

To optimize an iformat design for a particular application program, the iformat system selects custom templates from operation issue statistics obtained from scheduling the program. The iformat system then generates an iformat based on a combination of the custom templates and an abstract ISA specification.

The system uses a re-targetable compiler to generate the operation issue statistics for a particular processor design. As shown in FIG. 12, a module called the MDES extractor 560 generates a machine description in a format called MDES.

This machine description retargets the compiler 564 to the processor design based on its abstract ISA specification 510 and datapath specification 514. The compiler 564 then schedules a given application program 566 and generates operation issue statistics 568 regarding the usage of the operation groups in the instruction format templates. The system then uses the frequency of use of the operations in each template by the application program to compute customized templates as shown in step 569. The customization process is automated in that it selects custom templates by minimizing a cost function that quantifies the static or dynamic code size and the decode cost (e.g., measured in chip area).

The process of selecting instruction templates in the iformat based on scheduling statistics may be conducted as a stand-alone process, or may be conducted in conjunction with the automated iformat design process. In the latter case, it may be used to provide an initial input specification of the desired ILP constraints to the automated iformat design process. Additionally, it may be used to optimize an existing iformat design.

The system may also perform additional optimization by using variable-length field encodings to further reduce the instruction size. These optimized designs can lead to dramatic reductions in code size, as shown in the detailed description below.

7.2 Implementation of the Input Specification

The principal input of the iformat design process is an Abstract Instruction Set Architecture (ISA) specification 510. In the current implementation, the user or another program module may provide this specification as an ArchSpec in textual form.

An ArchSpec reader module converts the textual form of the ArchSpec to an abstract ISA spec data structure, which contains a machine-readable set of tabular parameters and constraints, including register file entries, operation groups, and exclusion/concurrency relationships.

7.3 Instruction Syntax

VLIW processors issue instructions having multiple instruction fields. An instruction field is a set of bit positions intended to be interpreted as an atomic unit within some instruction context. Familiar examples are opcode fields, source and destination register specifier fields, and literal fields. Bits from each of these fields flow from the instruction register to control ports in the data path. For example, opcode bits flow to functional units, and source register bits flow to register file read address ports. Another common type of instruction field is a select field. Select fields encode a choice between disjoint alternatives and communicate this context to the decoder. For example, a select bit may indicate whether an operand field is to be interpreted as a register specifier or as a short literal value.

An operation is the smallest unit of execution; it comprises an opcode, source operands, and destination operands. Each operand may support one or more operand types. A set of possible operand types is called an io-set. A list of io-sets,

one per operand, form an operation's io-format. For example, suppose an add operation permits its left source operand to be either an integer register or a short literal value, and suppose its right source and destination operands source and sink from integer registers. The corresponding io-sets are {gpr, s}, {gpr}, {gpr}. The io-format is simply this list of io-sets, which are abbreviated in shorthand notation as follows:

gpr s, gpr : gpr

Closely related operations such as add and subtract often have the same io-format. One reason for this is that related operations may be implemented by a single, multi-function unit (macro-cell). As discussed above, to simplify the instruction format design process, related operations are grouped into operation groups.

The instruction format assigns sets of op groups (called super groups) to slots of an instruction. The processor issues operations within an instruction from these slots concurrently. To fully specify an operation, the instruction format specifies both an op-group and an opcode (specific to that opgroup). In effect, this organization factors a flat opcode name space into a multi-tier encoding. In rare cases, this factorization may increase the encoding length by one bit per level. However, it should be noted that this approach does not preclude a flat encoding space: placing each operation in its own op-group eliminates the factorization. More importantly, hierarchical encoding often gives the same benefits as variable-length field encoding, but is simpler to implement.

7.4 The Instruction Format Tree

In a flat, horizontal instruction format, all instruction fields are encoded in disjoint positions within a single, wide instruction. A hierarchical instruction format allows exclusive instruction fields (those that are not used simultaneously in any instruction) to be encoded in overlapping bit positions, thereby reducing the overall instruction width. In the instruction format design system shown in FIG. 12, the hierarchical relationship between instruction fields is represented by an instruction format tree (if-tree). The leaves of an if-tree are instruction fields; where each leaf points to a control port in the data path, such as a register file address port, or an opcode input of a FU.

FIG. 13 illustrates the structure of an if-tree used in the current implementation. The overall structure of the tree defines how each instruction is built. Each part of the tree represents a node, with the lowest nodes (the cut-off-box-shaped nodes) forming the tree's leaves. The oval-shaped nodes are "OR" nodes, while the boxed-shaped nodes are "AND" nodes. The OR nodes denote a selection between the children of the node such that only one choice (one branch) extends to the next level. Conversely, an AND node allows all of the components of the node to form new branches. Stated another way, each level of the tree is either a conjunction (AND) or disjunction (OR) of the subtrees at the lower level.

The root node 632 of the tree is the overall machine instruction. This is an OR node representing a choice of instruction templates. A template select field (template ID) is used to identify the particular template. This select field is illustrated as the leaf node labeled "steer" connected to the instruction node 632.

Individual instructions are based on instruction templates, which are the AND-type child nodes of the root node (See, e.g., templates 634 and 636). The templates each encode the sets of operations that issue concurrently. Since the number of combinations of operations that may issue concurrently is astronomical, it is necessary to impose some structure on the

encoding within each template. Hence, each template is partitioned into one or more operation issue slots. Every combination of operations assigned to these slots may be issued concurrently.

In addition, each template has a consume to end-of-packet bit field (CEP) that indicates whether the next instruction directly follows the current instruction or it starts at the next packet boundary. This capability is used to align certain instructions (e.g. branch targets) to known address boundaries. Each template also specifies the number of spare bits that may be used to encode the number of no-op cycle to follow the current instruction. These spare bits may arise due to a need for packet alignment or quantized allocation.

The next level of the tree defines each of the concurrent issue slots. Each slot is an OR node supporting a set of operation groups, called a super group (i.e., nodes 638, 640, 642), that are all mutually exclusive and have the same concurrency pattern. A select field chooses among the various operation groups within a super group. Again, this select field is illustrated as the leaf node labeled "steer" connected to super group 640.

Below each super group lie operation groups as defined in the input specification as described above. Each operation group (e.g., operation group 643) is an OR node that has a select field ("steer") to choose among the various operation formats supported by operation group. FIG. 13 shows this situation where one operation format allows a literal field on the left port, while the other allows it on the right port.

Each operation format (e.g., IO format descriptors 644, 646) is an AND node consisting of the opcode field 654, the predicate field (if any) 656, and a sequence of source and destination field types (shown as IO sets 648, 660, 652). The traditional three-address operation encoding is defined at this level.

Each IO set is an OR node consisting of a singleton or a set of instruction fields that identify the exact kind and location of the operand. IO sets with multiple choices (e.g., 650) have a select field to identify which instruction field is intended. For example, one of the IO set nodes 650 represents a selection between instruction fields 660, 662, which is controlled via a multiplexor select field 664. The other IO sets each have only one kind of field, and thus, have a single child node representing that field (nodes 658, 666). The instruction fields point to the datapath control ports 668.

In implementing an instruction format, one principal design choice is whether to use a single, fixed-length instruction format, or allow variable-length instructions. The iformat design system supports both fixed and variable length instructions. The use of variable-length instructions produces more-compact code but increases decode complexity. The trade-off between code size and instruction decode complexity is a primary design consideration. A single, fixed-length instruction format simplifies decode logic and the data path for dispersal of operations to functional units, but it often results in poor code density, since the single format must accommodate the worst-case (longest) instruction. For example, if the longest instruction in a fixed-length instruction format is 128 bits long, then all of the instructions in the instruction set must be 128 bits long. In order to maintain a constant instruction length, many instructions will require the use of wasted bits whose sole purpose is to fill in unused space in the instructions. These wasted bits lead to increased code size. Conversely, variable-length instructions can accommodate both wide and compact, restricted instruction formats without wasting bits, which results in a reduction in code size. By using variable-length instructions, the instruction format can accommodate the

widest instructions where necessary, and make use of compact, restricted instruction formats, such as instructions that do not encode long literals.

FIG. 14 shows the format of an instruction and its building blocks. At the heart of the instruction is an instruction template 670. An instruction template encodes sets of operations that issue concurrently. Each template includes multiple concurrent slots 672, where each slot comprises a set of exclusive operation groups 674. Since all of the operations in an operation group are exclusive, all of the operations in each slot are also exclusive. Each template encodes the cross-product of the operations in each of its slots.

The length of each template is variable, depending in part on the length and number of the slots in the template. For example, some templates might have two slots, while other templates might have three or four slots. Furthermore, the width of each slot will depend on the width of the widest operation group within that slot, plus overhead, as shown in the lower portion of FIG. 14. There is considerable similarity and overlap among the opcodes within an operation group by construction, so very little encoding space is wasted within the operation group. But the opcode field now must be split into an operation group selection field 676 and an opcode selection field 678 within the operation group. With logarithmic encoding, this requires at most one additional bit for encoding the opcode. For example, 15 opcodes may be encoded in 4 bits, while splitting them into 3 operation groups of 5 opcodes each requires $\lceil \log_2(3) \rceil + \lceil \log_2(5) \rceil + 5$ bits. In addition, every slot has a reserved no-op encoding.

In cases where an op group has alternative operation formats, there is yet another select field to select the operation format.

Each instruction also includes a consume to end-of-packet bit 680, and a template specifier 682. The template specifier identifies the template. An instruction format having templates will need $\lceil \log_2(t) \rceil$ bits to encode the template specifier. This template specifier is in a fixed position within every instruction, and from its value, the instruction sequencer in the processor's control path determines the overall instruction length, and thus the address of the subsequent instruction.

In the current implementation, the length of the instruction is variable, but each length is a multiple of a predetermined number of bits called a quantum. For instance, if the quantum is 8 bits, the length of the instruction could be any number equal to or above some minimum value (say 32 bits) that is divisible by 8, such as 64 bits, 72 bits, 80 bits, etc. One or more dummy bits may be placed as appropriate within the instruction to ensure that the length of the instruction falls on a quantum boundary.

The iformat system builds the levels of the if-tree in an incremental fashion. It constructs the top three levels, consisting of the instruction, the templates, and the super groups from the abstract ISA specification, and optionally, custom templates. It constructs the middle layers, including the operation groups, the operation formats, and the field types from the abstract ISA specification. Finally, it constructs the instruction fields from the contents of the various field types in the abstract ISA specification and the individual control ports in the datapath that each field is supposed to control.

7.5 Instruction Templates

A primary objective of the instruction format design system is to produce a set of instruction templates that support the encoding of all of the sets of operation groups that can be issued concurrently. To initiate the template design process, the instruction format design system starts

out with the architecture specification, which defines the exclusion and concurrency constraints for a particular design. In one implementation, the architecture specification directly provides the exclusion relationships between operation groups. However, the iformat design process needs to know which opcodes can be issued concurrently, i.e., the concurrency relationship, rather than which opcodes must be exclusive.

In such an implementation, the concurrency relationship is taken to be the complement of the exclusion relationship. One way of determining the concurrency relation is to take the complement of the exclusion relations among opcodes implied by the architecture specification and treat each set of concurrent opcodes as a potential candidate for becoming an instruction template. While this provides an excellent starting point, it unfortunately does not lead to a practical solution, since the number of combinations of operations that may issue concurrently quickly becomes intractable. For example, a typical VLIW machine specification may include 2 integer ALUs, 1 floating point ALU and 1 memory unit, with 50 opcodes each. In such a machine the total number of distinct 4-issue instructions is $50^2 \times 50 \times 50 = 6,250,000$. Specializing instructions to 1, 2, and 3-issue templates would add many more. It is therefore necessary to impose some structure on the encoding within each template.

Our current implementation uses several mechanisms to reduce the complexity of the problem. These mechanisms represent iformat design decisions and affect the final instruction format layout and size. In most cases there may also be a tradeoff between the simplicity and orthogonality of the field layout (and hence the decode hardware) and the size of the instruction template. These tradeoffs will be described as the design process is detailed below.

As a first axiom, all templates must satisfy an exclusion constraint between two opcodes, i.e. these opcodes must never occupy separate slots in any template. This is because these opcodes may share hardware resources during execution, and therefore, the scheduler should never put these opcodes together within the same instruction. On the other hand, a concurrency constraint between two opcodes implies that the scheduler is free to issue these opcodes together in a single instruction and therefore there should be some template in which these two opcodes are allowed to occur together. In particular, that template may contain additional slots that can be filled with noops, if necessary. Therefore, it is unnecessary to generate a special template for each concurrency constraint, but rather all that is needed is a set of templates that can effectively cover all possible sets of concurrently scheduled opcodes.

The problem becomes greatly simplified when the concurrency of operation groups is considered instead of individual opcodes. As introduced above, operation groups are defined as sets of opcode instances that are generally similar in nature in terms of their latency and connectivity to physical register files and are expected to be mutually exclusive with respect to operation issue. All opcodes within an operation group must be mutually exclusive by definition. Furthermore, the instruction format is designed so that all opcodes within an operation group share the same instruction fields. Thus, the operation group is an obvious choice for the primary building block for creating templates.

Another simplification involves classifying mutually-exclusive operation groups into equivalence classes called super groups based on the constraints provided in the architecture specification. FIG. 15 illustrates an example that shows how the operation groups (shown as letters) and exclusion relations are used in the template selection pro-

cess. The process starts with the ILP constraints 681, which define a set of exclusion relationships 683 between operation groups 684. From these exclusion relationships, the iformat design system builds a boolean exclusion matrix 686. In the exclusion matrix 686, the rows and columns are matched up with respective operation groups, e.g., "A" corresponds to the operation group A, "B" corresponds to the operation group B, etc. The 1's in the matrix indicate an exclusion relationship, while a blank indicates that the corresponding operation groups may be issued concurrently. (The blanks are actually 0's in the real matrix—blanks are used here for clarity). The system then builds a concurrency matrix 688 from the exclusion matrix 686. The concurrency matrix 688 is the complement of the exclusion matrix 686. The "?"s along the diagonal of the concurrency matrix 688 can be interpreted as either a 1 or 0.

The rows in the concurrency matrix determine a set of concurrency neighbors for each operation group. A graphical representation of the relationships defined by the concurrency matrix 688 is shown in concurrency graph 692. Each node represents an operation group, while each connecting "edge" represents a concurrency relation. A clique is a set of nodes from a graph where every pair of nodes is connected by an edge. For instance, there are 16 cliques in the concurrency graph 692.

After the concurrency matrix is generated, the system compares the rows in the concurrency matrix to identify equivalent operation groups. The super groups are formed from the equivalent operation groups. Two operation groups are said to be equivalent if they have the same set of concurrency neighbors. Note that two mutually exclusive operation groups that have the same set of concurrency neighbors can replace each other in any template without violating any exclusion constraint and therefore can be treated equivalently. Similarly, two concurrent operation groups that have the same set of concurrency neighbors (other than themselves) can always be placed together in a template without violating any exclusion constraints and therefore can be treated equivalently.

An example of pseudocode for performing equivalence checking and partitioning into super groups is illustrated below.

```

ProcedureFindSuperGroups (BitMatrix concur)
1: // "concur" is a (numNodes x numNodes) boolean matrix
2: //First, initialize supergroup hash table and id counter
3: HashMap<BitVector, int> SGmap
4: int SGcount = 0;
5: for (i = 0 to numNodes-1) do
6: //extract each node's vector of neighbors w/ and w/o self
7: BitVector AND-group = concur.row(i).set_bit(i);
8: BitVector OR-group = concur.row(i).reset_bit(i);
9: //Check for existing AND-style supergroup for this node
10: if (SGmap(AND-group) is already bound) then
11:   SGid(i) = SG-AND;
12:   SGid(i) = SGmap(AND-group);
13: //Check for existing OR-style supergroup for this node
14: else if (SGmap(OR-group) is already bound) then
15:   SGid(i) = SG-OR;
16:   SGid(i) = SGmap(OR-group);
17: //If neither neighbor relation is present, start a new
18: //supergroup with the new neighbor relations
19: else
20:   SGid(i) = SGcount;
21:   SGmap(AND-group) = SGmap(OR-group) = SGcount;
22:   SGcount = SGcount + 1;
23:   endif
24: endfor

```

The equivalence check and the partitioning can be performed quickly by employing the pigeon-hole principle. The

algorithm hashes each operation group using its set of neighbors in the concurrency matrix as the key. The neighbor relations (neighbor keys) for each operation group (each row) are converted to bitvectors. The algorithm hashes in two ways: once by treating each operation group as concurrent with itself (AND-style) thereby finding equivalent concurrent operation groups, and the second time by treating each operation group as exclusive with itself (OR-style) thereby finding equivalent exclusive operation groups. This hashing approach results in two bitvectors for each operation group—one with the “?” entry changed to a 1 (AND-style), and one with the “?” entry changed to a 0 (OR-style).

Bitvectors (operation groups) that hash to the same bucket necessarily have the same concurrency neighbors and therefore become part of the same super group. For example in FIG. 15, operation groups A, B, and C have the same concurrency neighbors and thus form the super group {A, B, C}. The other super groups, {P, Q}, {X, Y}, and {M, N}, are similarly formed. The set of all distinct super groups is defined by all the distinct neighbor keys. This partitioning leads to a reduced-concurrency (super group) graph 694, comprising the super groups and their concurrency relations. Instruction templates 696 are obtained from the reduced concurrency graph, as described below.

Each operation group identifies whether it is an AND-type or OR-type super group. This information is used in the final template expansion, where each operation group from an AND-type super group is given a separate slot, while all operation groups from an OR-type super group are put into the same slot.

In the concurrency matrix 690 shown in FIG. 15, the “?” entries of the “A”, “B”, and “C” operation group bitvectors have been changed to 0’s so that their corresponding bitvectors are identical. Thus, “A”, “B”, and “C” form an OR-type super group {A, B, C}, and each operation group is placed in the same slot.

FIG. 16 shows a case with an AND-type and an OR-type super group. In order to obtain identical bitvectors, the “A”, “B”, and “C” operation groups are treated as being concurrent with themselves. As a result, they form an AND-type super group and are placed in separate template slots. In contrast, the “M”, “N”, “X”, and “Y” operation groups are treated as exclusive with themselves and form two different sets of OR-type super groups {M, N} and {X, Y}, which each occupy a single slot.

For a homogenous VLIW-style machine with multiple, orthogonal functional units this process yields tremendous savings by reducing the complexity of the problem to just a few independent super groups. The resulting instruction templates closely match super groups to independent issue slots for each functional unit. For a more heterogeneous machine with shared resources, the resulting number of templates may be larger and the decoding is more complex but partitioning the operation groups into super groups still reduces the complexity of the problem significantly.

7.6 Concurrency Cliques and Templates

Once the super groups have been determined, each clique in the reduced concurrency graph is a candidate for an instruction template since it denotes a set of super groups that may be issued in parallel by the scheduler. A clique is a subgraph in which every node is a neighbor of every other node. Clearly, enumerating all cliques would lead to a large number of templates. On the other hand, unless the concurrency among super groups is restricted in some other way, it is necessary to choose a set of templates that cover all possible cliques of the super group graph to ensure that the scheduler is not restricted in any way other than that specified in the ArchSpec.

As an example, suppose super groups A, B and C only have pairwise concurrency constraints, i.e., {AB}, {AC}, and {BC}. These pairwise concurrencies can be covered in one of two ways. First, the pairwise concurrency constraints can be treated as three independent templates AB, AC, and BC, each requiring two issue slots. A second possibility is to treat the pairwise concurrencies as being simultaneously concurrent, thereby requiring only one template (ABC) with three issue slots. Strictly speaking, this allows more parallelism than what was intended. If the compiler never scheduled all three operations simultaneously, the second design would end up carrying one noop in every instruction thereby wasting one-third of the program space. On the other hand, the first design requires additional decoding logic to select among the three templates and more complex dispersal of the instruction bits to the various functional units.

In the present scheme, this tradeoff is made towards initially choosing a reduced number of possibly longer templates. This is partly due to the fact that the ArchSpec does not directly specify concurrency in most instances, but rather specifies exclusion relations among operation groups that are then complemented to obtain concurrency relations. During the initial template design phase, choosing the maximally concurrent templates covers all possible concurrency relations with as few templates as possible.

The maximally concurrent templates may be determined by finding the cliques of the super group graph. An example of a simple reduced super group concurrency graph is shown in FIG. 17. The graph comprises super groups 1–7, and their interconnecting edges. The maximal cliques for such a simple graph can be determined by hand by simply identifying sets of nodes that are completely connected—that is each node in a clique must connect to the remaining nodes in the clique. For instance, {1, 3, 7} is a clique, while {2, 4, 5, 6} is not (nodes 5 and 6 are not connected). In the supergraph of FIG. 6, there are seven maximal cliques, and thus seven maximally concurrent templates.

It is necessary to use computational means to calculate the cliques for more complex super group graphs. The instruction format designer uses the same approach for finding cliques as the datapath synthesizer described above.

7.7 Set-Up of Bit Allocation Problem

Once the templates are selected, the iformat system constructs the lower levels of the IF tree. The templates form the upper level of the tree. For each of the operation groups in a template, the system extracts the inputs and outputs for each operation based on their I/O formats in the abstract ISA specification and adds this information to the IF tree. Using the extracted I/O formats, the system enumerates the instruction fields for each of the operation groups associated with the templates. Next, it builds field conflicts, partitions instruction fields into superfields, and extracts bit width requirements.

7.7.1 Instruction Fields

As shown in FIG. 13, the instruction fields form the leaves of the if-tree. Each instruction field corresponds to a datapath control port such as register file read/write address ports, predicate and opcode ports of functional units, and selector ports of multiplexors. Each field reserves a certain number of instruction bits to control the corresponding control port.

The iformat designer assigns each field to a control port by traversing the if tree to find the operation group associated with the field, and then extracting the functional unit assigned to the operation group in the datapath specification.

The following sub-sections describe various kinds of instruction fields. FIG. 20 is annotated with letters S, A, L,

op and C to illustrate examples of the information flowing from these fields in the instruction register to the control ports in the data path.

Select fields (S). At each level of the if-tree that is an OR node, there is a select field that chooses among the various alternatives. The number of alternatives is given by the number of children, n , of the OR node in the if-tree. Assuming a simple binary encoding, the bit requirement of the select field is then $\log_2(n)$ bits.

Different select fields are used to control different aspects of the datapath. The root of the if-tree has a template select field that is routed directly to the instruction unit control logic in order to determine the template width. It also specifies where the supergroup select fields are positioned. Therefore, this field must be allocated at a fixed position within the instruction. Together with the template select fields, the select fields at super group and operation group levels determine how to interpret the remaining bits of the template and therefore are routed to the instruction decode logic for the datapath. The select fields at the level of field types (IO sets) are used to control the multiplexors and tristate drivers at the input and output ports of the individual functional units to which that operation group is mapped. These fields select among the various register and literal file alternatives for each source or destination operand.

Register address fields (A). The read/write ports of various register files in the datapath need to be provided address bits to select the register to be read or written. The number of bits needed for these fields depends on the number of registers in the corresponding register file.

Literal fields (L). Some operation formats specify an immediate literal operand that is encoded within the instruction. The width of these literals is specified externally in the ArchSpec. Dense ranges of integer literals may be represented directly within the literal field, for example, an integer range of -512 to 511 requires a 10-bit literal field in 2's complement representation. On the other hand, a few individual program constants, such as 3.14159, may be encoded in a ROM or a PLA table whose address encoding is then provided in the literal field. In either case, the exact set of literals and their encodings must be specified in the ArchSpec.

Opcode fields (op). The opcode field bits are used to provide the opcodes to the functional unit to which an operation group is assigned. It is possible to use the internal hardware encoding of opcodes in the functional unit directly as the encoding of the opcode field, in which case the width of the opcode field is the same as the width of the opcode port of the corresponding functional unit and the bits are steered directly to it. This mechanism may be used when all the opcodes supported by a functional unit are present in the same operation group or the same super group.

Under some templates, however, the functional unit assigned to a given operation group may have many more opcodes than those present within the operation group. In this case, opcode field bits may be saved by encoding the hardware opcodes in a smaller set of bits determined by the number of opcodes in that operation group and then decoding these bits before supplying to the functional unit. In this case, the template and opgroup specifier bits are used to provide the context for the decoding logic.

Miscellaneous control fields (C). Some additional control fields are present at the instruction level that help in proper sequencing of instructions. These consists of the consume to end-of-packet bit (Eop) and the field that encodes the number of no-op cycles following the current instruction.

7.7.2 Computing Field Conflicts

Before performing graph coloring, the system computes the pairwise conflict relation between instruction fields, which are represented as an undirected conflict graph.

In the if-tree, two leaf nodes (instruction fields) conflict if and only if their least-common ancestor is an AND node. The system computes pairwise conflict relations using a bottom-up data flow analysis of the if-tree. The procedure in the implementation maintains a field set, F , and a conflict relation, R . Set F_n is the set of instruction fields in the subtree rooted at node n . Relation R_n is the conflict relation for the subtree rooted at node n .

The procedure processes nodes in bottom-up order as follows:

leaf node: At a leaf node, f , the field set is initialized to contain the leaf node, and the conflict relation is empty.

or-node: At an OR-node, the field set is the union of field sets for the node's children. Since an OR-node creates no new conflicts between fields, the conflict set is the union of conflict sets for the node's children.

and-node: At an AND-node, the field set is the union of field sets for the node's children. An AND-node creates a new conflict between any pair of fields for which this node is the least-common ancestor; i.e. there is a new conflict between any two fields that come from distinct subtrees of the AND-node. Formally,

$$C_n = \bigcup_{i,j \in \text{children}} C_i \cup \{(x, y) | x \in C_j, y \in C_k, j \neq k\}$$

This method can be implemented very efficiently, by noting that the sets can be implemented as linked lists. Because the field sets are guaranteed to be disjoint, each union can be performed in constant time by simply linking the children's lists (each union is charged to the child). Similarly, the initial union of children's conflict sets can be done in constant time (charged to each child). Finally, forming the cross-product conflicts between fields of distinct and-node children can be done in time proportional to the number of conflicts. Since each conflict is considered only once, the total cost is equal to the total number of conflicts, which is at most n^2 . For an if-tree with n nodes and E conflicts, the overall complexity is $O(n+E)$ time.

7.7.3 Assigning Field Affinities

As introduced above, the ifformat system is capable of aligning instruction fields that correspond to the same control port to the same bit position in a process called affinity allocation. Such alignment may simplify the multiplexing and decoding logic required to control the corresponding datapath control ports since the same instruction bits are used under different templates. On the other hand, such alignment may waste some bits in the template thereby increasing its width.

In order to make use of affinity allocation, the ifformat designer groups instruction fields that point to the same datapath control port into a superfield. All instruction fields within a superfield are guaranteed not to conflict with each other since they use the same hardware resource and therefore must be mutually exclusive.

The superfield partitioning only identifies instruction fields that should preferably share instruction bits. However, sometimes it is deemed essential that certain instruction fields must share the same bits. For example, if the address bits of a register read port are aligned to the same bit positions under all templates, then these address bits may be steered directly from the instruction register to the register

file without requiring any control logic to select the right set of bits. This forced sharing of bit positions can avoid the need for a multiplexor in the critical path of reading operands out of a register file, thereby enhancing performance.

To handle such a constraint, the iformat system allows a user or other program module to specify a subset of fields within a superfield that must share bits. One way to specify this is in the form of a level mask that identifies the levels of the if-tree below which all instruction fields that are in the same superfield must share bit positions. This mask is a parameter to the bit allocation process described in the next section.

7.8 Resource Allocation

Once the instruction fields have been assigned to the leaves and the pairwise conflicts have been determined, we are ready to begin allocating bit positions to the instruction fields. In this problem, instruction fields are thought of as resource requesters. Bit positions in the instruction format are resources, which may be reused by mutually exclusive instruction fields. Fields required concurrently in an instruction must be allocated different bit positions, and are said to conflict. The resource allocation problem is to assign resources to requesters using a minimum number of resources, while guaranteeing that conflicting requesters are assigned different resources. The current implementation of resource allocation uses a variation of graph coloring.

Once the if-tree and instruction field conflict graph are built, the iformat system can allocate bit positions in the instruction format to instruction fields. Pseudocode for the resource allocation is shown below:

```
ResourceAlloc(nodeRequests, conflictGraph)
// compute resource request for each node+neighbors
foreach (node ∈ conflictGraph)
  Mark(node)=FALSE;
  TotalRequest(node)=Request(node)+Request
    (NeighborsOf(node));
// sort nodes by increasing remaining total resource
  request
// compute upper-bound on resources needed by allocation
  resNeeded=0; Stack=EMPTY;
for (k from 0 to NumNodes(conflictGraph))
  find (minNode ∈ unmarked nodes) such that
    TotalRequest(minNode) is minimum;
  Mark(minNode)=TRUE;
  push(minNode,Stack);
  resNeeded=max(resNeeded, TotalRequest:
    (minNode));
  foreach (nhbr ∈ NeighborsOf(minNode))
    TotalRequest(nhbr)-=Request(minNode);
// process nodes in reverse order (i.e., decreasing total
  request)
while (Stack is not EMPTY)
  node=pop(Stack);
  AllResources={0 . . . resNeeded-1};
  // available bits are those not already allocated to any
  neighbor
  AvailableRes(node)=AllResources-AllocatedRes
    (NeighborsOf(node));
  // select requested number of bits from available positions
  // according to one of several heuristics
  AllocatedRes(node)=Choose Request(node) resources
    from AvailableRes(node)
    ☑H1: Contiguous Allocation
    ☑H2: Affinity Allocation
  return resNeeded
```

In the above pseudocode, the total resource request for a node and its neighbors is computed by the first loop. The heuristic repeatedly reduces the graph by eliminating the node with the current lowest total resource request (node plus remaining neighbors). At each reduction step, we keep track of the worst-case resource limit needed to extend the coloring. If the minimum total resources required exceed the current value of k, we increase k so that the reduction process can continue. The graph reduction is performed by the second loop. Nodes are pushed onto a stack as they are removed from the graph. Once the graph is reduced to a single node, we begin allocating bit positions (resources) to nodes. Nodes are processed in stack order, i.e. reverse reduction order. At each step, a node is popped from the stack and added to the current conflict graph so that it conflicts with any neighbor from the original graph that is present in the current conflict graph. The existing allocation is extended by assigning bit positions to satisfy the current node's request, using bit positions disjoint from bit positions assigned to the current node's neighbors.

7.8.1 Allocation Heuristics

During bit allocation, the current node's request can be satisfied using any bit positions disjoint from positions allocated to the node's neighbors in the current conflict graph. The current implementation applies several heuristics to guide the selection of bits.

Left-most allocation. The number of required bit positions computed during graph reduction is the number needed to guarantee an allocation. In practice, the final allocation often uses fewer bits. By allocating requested bits using the left-most available positions, we can often achieve a shorter instruction format.

Contiguous allocation. Since bit positions requested by an instruction field generally flow to a common control point in the data path, we can simplify the interconnect layout by allocating requested bits to contiguous positions.

Affinity allocation. Non-conflicting instruction fields may have affinity, meaning there is an advantage to assigning them the same bit positions. For example, consider two non-conflicting fields that map to the same register file read address port. By assigning a single set of bit positions to the two fields, we reduce the interconnect complexity and avoid muxing at the read address port. As discussed earlier, each node has a set of affinity siblings. During allocation, we attempt to allocate the same bit positions to affinity siblings. This heuristic works as follows. When a node is first allocated, its allocation is also tentatively assigned to the node's affinity siblings. When a tentatively allocated node is processed, we make the tentative allocation permanent provided it does not conflict with the node's neighbors' allocations. If the tentative allocation fails, we allocate available bits to the current node using the previous heuristics, and we then attempt to re-allocate all previously allocated affinity siblings to make use of the current node's allocated bits. Because nodes are processed in decreasing order of conflict, tentative allocations often succeed.

A heuristics diagram for the resource allocation is as follows:

```
if node is tentatively allocated then
  make tentative allocation permanent, if possible
if node is (still) not allocated then
  try to use a sibling allocation
if node is (still) not allocated then {
  allocate either contiguously, or left-most available
  for each sibling of node {
    if sibling is allocated then
      try to use node's allocation in place of existing
      allocation
```

```

else
    tentatively allocate sibling, using node's allocation
} // for

```

7.9 Template-based Assembly

Once the complete structure of the instruction templates has been determined, we can proceed to assemble the code. All subsequent discussion is essentially to improve the quality of the templates. In this section, we briefly outline the process of assembly with a given set of templates.

A program that has been scheduled and register-allocated consists of a sequence of operations each of which has been assigned a time of issue. Multiple operations scheduled within the same cycle need to be assembled into a single instruction. Any instruction template that covers all the operations of an instruction may be used to assemble that instruction. Clearly, the shortest template is preferred to avoid increasing the codesize unnecessarily since longer templates would have to be filled with noops in the slots for which there are no operations in the current instruction.

The process of template selection for an instruction has the following steps. First, the specific compiler-opcode of each scheduled operation in the instruction is mapped back to its operation group. Each operation group keeps a record of the set of templates that it can be a part of. Finally, the intersection of all such sets corresponding to the operation groups present in the current instruction gives the set of templates that may be used to encode the current instruction. The shortest template from this set is chosen for assembly. The exact opcode and register bits are determined by mapping the compiler mnemonics to their machine encodings by consulting the if-tree.

7.10 Design of Application-specific Instruction Formats

As discussed above, the initial design produces a minimal set of maximally concurrent instruction templates that cover all possible concurrency relations implied by the ArchSpec. In practice, this tends to produce a few long templates since the processor designs we are interested in have quite a bit of expressible instruction-level parallelism (ILP). But not all that parallelism is used at all times by the scheduler. If we assemble programs using only these long templates, a lot of noops would have to be inserted in the low ILP parts of the code.

One fix to this problem is to customize the templates to the program being compiled. There are several aspects to such customization:

- (1) Identify the most frequently used combinations of operations in the program and design shorter templates for them which allow fewer concurrent operations in them. An extension of this view also takes into account the most frequently used operation formats and creates new opgroups that incorporate just those.
- (2) Use variable length encoding wherever there is a need to select one out of many choices in the instruction format. We may use variable length template selection bits according to the frequency of use of each template. Likewise, different operation groups within a slot and different opcodes within an operation group may be given a variable length encoding according to their frequency of use. There is, of course, a tradeoff between the codesize reduction and the increase in decode complexity.
- (3) Sometimes, the decode complexity may be improved dramatically by doing affinity-based allocation of similar instruction fields across templates. This reduces the degree of multiplexing needed to route the same information represented at different positions in different

templates. This amounts to reordering the positions of various operation groups within these templates.

- (4) The instruction fetch and decode hardware is usually designed with a certain quantum of instruction information in mind. A quantum is a unit of data (e.g., an integer multiple of bytes) used to specify the width of the data path in the instruction fetch and decode hardware. Rounding the instruction templates up to the next quantum usually frees up extra bit space. One or more of the above strategies can then take advantage of this extra bit space without increasing the width of the instruction.

7.11 Schedule-based template customization

The instruction format information is not needed until the program is ready to be assembled. The compiler is driven by a machine-description that only depends on the specified ArchSpec and the structure of the datapath. This implies that the exact schedule of the program may be used to customize the various available templates. To customize templates for a particular application program, the iformat system uses operation issue statistics from a scheduled version of the program to determine the frequency of use of the various combinations of operations. It then selects frequently used combinations of operations as possible candidates for new templates. Finally, it performs a cost/benefit analysis to select new "custom" templates.

FIG. 18 is a flow diagram illustrating a process of selecting custom templates from operation issue statistics. The process begins by extracting usage statistics from a scheduled application program 700. This is done by mapping the scheduled opcodes of an instruction back to their operation groups as shown in step 702. The process then generates a histogram of combinations of operation groups from the program as shown in step 704.

A static histogram records the frequency of static occurrences of each combination within the program and may be used to optimize the static codesize. A dynamic histogram weights each operation group combination with its dynamic execution frequency and may be used to improve the instruction cache performance by giving preference to the most frequently executed sections of the code. One implementation uses the static histogram in the optimization to give preference to the overall static code size. In alternative implementations, the dynamic histogram or both the dynamic and static histograms may be used to optimize the dynamic code size of the combined dynamic/static code size, respectively.

Based on the frequency of use data in the histogram, the customization process selects combinations of opgroups as potential candidates for templates (706) and evaluates their cost/benefit (708) in terms of code size/decode complexity, which is quantified in a cost function. The process iteratively selects a set of templates, evaluates their cost/benefit, and ultimately returns a set of custom templates that meet a predetermined optimization criteria (710, 712). As noted above, the criteria may include, for example, a minimized static or dynamic code size or a minimized code size and decode complexity. An example of this criteria is discussed below.

In the current implementation, the problem of determining custom templates is formulated as follows. Let us assume that T_1, \dots, T_n are the instruction templates that are required to conform with the ArchSpec. Suppose C_1, \dots, C_m are distinct combinations of operation groups occurring in the program. Let the width of each combination be w_i and its frequency of occurrence be f_i . Also, in case of unoptimized assembly, suppose each combination C_i maps to an

initial template T_i with width v_i . Assuming that variable length encoding is not used for the template selection field, the initial size of the program is,

$$W = \sum_{i=1}^m f_i \cdot (v_i + \lceil \log_2 n \rceil)$$

Now suppose we include C_i as a custom template. This is taken to be in addition to the initial set of templates since those must be retained to cover other possible concurrency relations of the machine as specified in the ArchSpec. The additional template has a smaller width w_i but it increases the size of the template selection field (and hence the decode logic). The other significant increase in decode cost is due to the fact that now the same operation may be represented in two different ways in the instruction format and hence the instruction bits from these two positions would have to be multiplexed based on the template selected. This cost may be partially or completely reduced by performing affinity allocation as discussed above.

If X_i represents a 1/0 variable denoting whether combination C_i is included or not, the optimized length of the program is denoted by,

$$\begin{aligned} W_{cost} &= \sum_{i=1}^m f_i \cdot (X_i \cdot w_i + (1 - X_i) \cdot v_i + \lceil \log_2 (n + \sum X_i) \rceil) \\ &= \sum_{i=1}^m f_i \cdot (v_i - X_i \cdot (v_i - w_i) + \lceil \log_2 (n + \sum X_i) \rceil) \end{aligned}$$

It is clear that we should customize all those operation group combinations into additional templates that provide the largest weighted benefit until the cost of encoding additional templates and their decoding cost outweigh the total benefits. One possible strategy is to pick the k most beneficial combinations where k is a small fixed number (e.g. $k < 16$). The decode complexity directly impacts chip area needed for decode logic. With an increase in the number of templates, the complexity of the decode logic tends to grow, unless affinity constraints are used to align operation group occurrences from different templates to the same template slots. The chip area occupied by selection logic may be quantified as another component of the cost function.

7.12 Variable Length Field Encodings

Variable length field encoding is an important technique for reducing the overall instruction format bit length. The simplest use of variable length fields is in encoding a steering field that selects one of a set of exclusive fields of differing lengths. For example, the instruction formats have an opgroup steering field to select one of many opgroups available within a single issue slot. Suppose we have 32 opgroups available within a particular issue slot, and that the opgroups' encodings require lengths from 12 to 29 bits. With fixed-length encodings, we require an additional 5 bits to encode the opgroup selection, bringing the overall size of the issue slot to 34 bits. Using a variable-length encoding, we can allocate short encodings to opgroups having the greatest overall width, while using longer encodings for opgroups having smaller width. Provided there is enough "slack" in the shorter opgroups to accommodate longer encodings, the overall bit requirement can be reduced significantly. In our example, we may be able to achieve a 30 bit encoding for the issue slot.

One approach to designing variable-length encodings uses entropy coding, and in particular, a variant of Huffman encoding. Entropy coding is a coding technique typically

used for data compression where an input symbol of some input length in bits is converted to a variable length code, with potentially a different length depending on the frequency of occurrence of the input symbol. Entropy coding assigns shorter codes to symbols that occur more frequently and assigns longer codes to less frequent codes such that the total space consumed of the coded symbols is less than that of the input symbols.

Let F be a set of exclusive bit fields, and let w_i denote the bit length of field $i \in F$. An encoding for the steering field for F is represented as a labeled binary tree, where each element of F is a tree leaf. The edge labels (zero or one) on the path from the root to a leaf i denotes the binary code for selecting i . A fixed-length steering code is represented by a balanced tree in which every leaf is at the same depth. Variable-length encodings are represented by asymmetric trees.

For a tree T representing a code for F , we define $d_T(x)$ to be the depth of x , i.e., the code length for choice x . The total cost of encoding a choice x is the sum of the bit requirement for x and the code length for x :

$$cost_T(x) = d_T(x) + W(x)$$

The overall cost for encoding the set of fields F together with its steering field is equal to the worst-case single field cost:

$$C(T) = \max_{x \in F} \{cost_T(x)\}$$

The goal is to find a code T of minimal cost. This problem is solved by the algorithm shown below:

```

Huffman (Set C, Weights W)
1:  N = |C|;
2:  //insert elements of C into priority queue
3:  for x ∈ C do
4:      enqueue(x, Q);
5:  endif
6:  for i = 1 to n-1 do
7:      z = new node;
8:      x = extract_min(Q);
9:      y = extract_min(Q);
10:     z.left = x; z.right = y;
11:     W(z) = max {W(x), W(y)} + 1;
12:     enqueue(z, Q);
13:  endif
14:  return extract_min(Q);
    
```

7.13 Extracting an Abstract ISA Specification from a Concrete ISA Specification

As outlined above, the iformat design process may be used to generate an instruction format specification from a datapath specification and an abstract ISA specification. In an alternative design scenario, the iformat design process may be used to generate optimized concrete ISA specification programmatically from an initial concrete ISA specification and a list of frequently occurring combinations of operation group occurrences and ILP constraints. The initial concrete ISA specification includes an instruction format specification and a register file specification and mapping. The register file specification and mapping provides: 1) the register file types; 2) the number of registers in each file; and 3) a correspondence between each type of operand instruction field in the instruction format and a register file.

In order to optimize the instruction format in this scenario (and thereby the concrete ISA specification), the iformat design process programmatically extracts an abstract ISA

specification from the concrete ISA specification (see step 554 in FIG. 12). It then proceeds to generate the bit allocation problem specification, and allocate bit positions programmatically as explained in detail above. The operation group occurrences and ILP constraints (e.g., concurrency sets of the operation group occurrences) may be provided as input from the user (e.g., starting with a custom template specification at block 556 in FIG. 12), or may be generated programmatically from operation issue statistics 568 in step 569 shown in FIG. 12 and described above.

Given a Concrete ISA Specification, this step extracts the information corresponding to an Abstract ISA Specification. The Instruction Format, which is part of the Concrete ISA Specification, consists of a set of Instruction Templates, each of which specifies sets of mutually exclusive opcodes that can be issued in parallel. From this information one can define the corresponding Operation Group Occurrences and a Concurrency Set consisting of these Operation Group Occurrences. All of the Instruction Templates, together, define the opcode repertoire, the Operation Groups and the ILP specification that form part of the Abstract ISA Specification. The Instruction Format Specification directly provides the I/O Format for each opcode as needed by the Abstract ISA Specification. The Register File Specification in the Concrete ISA Specification directly provides the Register File Specification that completes the Abstract ISA Specification.

8.0 Overview of Control Path Design System

The control path design system is a programmatic system that extracts values for control path parameters from an instruction format and data path specification and creates a control path specification in a hardware description language, such as AIR.

FIG. 19 is a block diagram illustrating a general overview of the control path design system. The inputs to the control path design synthesizer (CP synthesizer) 800 include a data path specification 802, an instruction format specification 804, and ICache parameters 806. The CP synthesizer selects the hardware components for the control path design from a macrocell database 808 that includes generic macrocells for a sequencer, registers, multiplexors, wiring buses, etc. in AIR format. The macrocell database also includes a machine description of certain macrocells, referred to as mini MDES. The mini-mdes of a functional unit macrocell, for example, includes the functional unit opcode repertoire (i.e., the opcodes executable by the functional unit and their binary encoding), a latency specification, internal resource usage, and input/output port usage.

Implemented as a set of program routines, the CP synthesizer extracts parameters from the data path, the instruction format, and instruction cache specifications and synthesizes the control path including the IUDatapath, control logic for controlling the IUDatapath, and decode logic for decoding the instructions in the instruction register.

The CP synthesizer builds the IUDatapath based on the instruction width requirements extracted from the instruction format specification. It instantiates macrocells in the IUDatapath by computing their parameters from the maximum and minimum instruction sizes and the instruction cache access time.

It then constructs the control logic for controlling the IUDatapath based on the computed IUDatapath parameters and the ICache parameters. The ICache parameters provide basic information about the instruction cache needed to construct the instruction fetch logic. These parameters include the cache access time and the width of the instruction packet, which is the unit of cache access.

The control path design process synthesizes the decode logic for decoding the instruction in the instruction register by scanning the instruction format and data path control ports. It also determines the interconnect between the bit positions in the instruction register and the control ports in the data path.

The CP synthesizer is programmed to optimize the design of the instruction unit for a pre-determined control path protocol. As part of this process, it may optimize the instruction pipeline (the IUDatapath) by selecting macrocells that achieve a desired instruction issue rate, such as one instruction to the decode logic per cycle, and by minimizing the area occupied by the macrocells. It also minimizes the area of the control logic, such as the area that the IU control logic and decode logic occupies.

The output of the control path design process is a data structure that specifies the control path hardware design in the AIR format 810. The AIR representation of the IUDatapath includes the macrocells for each of the components in the IUDatapath. This may include, for example, a prefetch buffer for covering the latency of sequential instruction fetching, and other registers used to store instructions before issuing them to the decode logic. The AIR representation includes a macrocell representing the sequencer and the control logic specification (e.g., a synthesizable behavioral description, control logic tables, etc.) representing the control logic for each of the components in the IUDatapath. Finally, the AIR representation includes a decode logic specification (e.g., decode logic tables) representing the instruction decode logic and the interconnection of this decode logic between the instruction register and the control ports enumerated in the data path specification. Conventional synthesis tools may be used to generate the physical logic (such as a PLA, ROM or discrete logic gates) from the control and decode logic specifications.

8.1 The Relationship between the Control Path and the Control Ports in the Data Path

Before describing aspects of the control path in more detail, it is instructive to consider the state of the processor design before the CP synthesizer is executed. As noted above, one input of the control path design process is the data path specification. Provided in the AIR format, the data path input 802 specifies instances of the functional unit macrocells and register file macrocells in the data path. It also specifies instances of the macrocells representing the wiring that interconnects the read/write data ports of the register files with input and output data ports of the functional units. At this phase in the design of the processor, the control ports in the data path are enumerated, but are not connected to other components. For example, the opcode input of the functional units and the address inputs of the register files are enumerated, but are not connected to the control path hardware.

FIG. 20 illustrates an example of a processor design, showing the relationship between the data path (in dashed box 820) and the control path. The data path includes a register file instance, gpr, a functional unit (FU) cell instance, and an interconnect between the gpr and functional unit. The interconnect comprises data buses 822-830 that carry data between the FU and gpr, a multiplexor 832 that selects between input sources (e.g., gpr and literal pseudo-register Sext), and tri-state buffer 834 that drives output data from the FU onto a data bus 830. The data read ports of the gpr, dr0 and dr1, provide data to the data input ports of the FU, i0 and i1, via buses 822-828 and multiplexor 832. The output port of the FU, o0, provides data to the data write port, dw0, via tri-state buffer 834 and data bus 830.

The control ports that are enumerated, yet remain unconnected before the control path design, include the read and write address ports of the gpr, ar0, ar1 and aw0, and the opcode input port, op, of the FU. Some data ports in a FU or gpr may map to more than one data port in the gpr or FU, respectively. This sharing may be controlled via control ports of a multiplexor 832 or tri-state buffer 834.

Also, a control port of the gpr or FU may map to more than one bit position in the instruction. This type of sharing may be controlled via control ports of a multiplexor 836, for example. However, the hardware logic to control this sharing is left to be specified in the control path design process.

The mapping between the instruction fields in an instruction and the control ports in the data path is specified in the instruction format specification. The datapath specification enumerates the control ports in the data path and provides the information needed to map these control ports to the instruction fields. The instruction format specification specifies the specific bit positions and encodings of the fields in the instruction fields.

The following sections describe in more detail how an implementation of the control path design process generates the control path.

8.2 The Control Path Protocol

The control path design process synthesizes a specific control path design based on a predefined control path protocol. In the current implementation, the control path protocol defines a method for fetching instructions from an instruction cache and dispatching them sequentially to an instruction register that interfaces with the processor's decode logic. It also defines the type of macrocells that the control path will be constructed from and enumerates their parameters. The CP synthesizer program then selects the macrocells and computes specific values for their parameters based on information extracted from the instruction format and datapath.

The example in FIG. 20 helps to illustrate the control path protocol used in the current implementation. It is important to note that a number of design choices are made in defining the protocol, and these design choices will vary with the implementation. The illustrated protocol represents only one possible example.

To get a general understanding of the control path protocol, consider the flow of an instruction through the control path in FIG. 20. The sequencer 900 initiates the fetching of instructions into the IUDatapath. The MAR 902 in the sequencer stores the address of the next instruction to be fetched from the instruction cache 904. Using the contents of the MAR, the sequencer initiates the fetching of instructions from the cache for both a sequential mode and a branch mode.

In order to specify values for the widths of components in the IUDatapath, the CP synthesizer extracts information about the instruction widths from the instruction format specification. The protocol specifies the types of parameters that need to be extracted from this information.

The parameters extracted from the instruction format include:

- Q_i // quantum (bytes) (greatest common denominator of all possible instruction widths, fetch widths)
- W_{min} // minimum instruction width (quanta)
- W_{max} // maximum instruction width (quanta)

The parameter, Q_i , is a unit of data used to express the size of instruction and fetch widths in an integer multiple of bytes and is referred to as a quantum. This parameter is not critical to the invention, but it does tend to simplify the design of other components such as the alignment network

because it is easier to control shifting in units of quanta rather than individual bits. The parameters to be extracted also include, W_{min} , the minimum instruction width in quanta, and W_{max} , the maximum instruction width in quanta.

The protocol also defines parameters relating to the instruction cache (ICache) as follows:

W_A // instruction packet width (quanta) ($W_A \geq W_{max}$, $W_A = 2^n$)

W_L // cache line size (quanta) ($W_L \geq W_A$, $W_L = 2^n$)

T_A // cache access time (cycles)

The instruction packet defines the amount of data that the control path fetches from the ICache with each fetch operation. In the protocol of the current implementation, the size of the instruction packet is defined to be at least as large as the widest instruction and is expressed as a number of quanta that must be a power of two. However, the packet need not be that large if the widest instruction is infrequent. In instruction format designs where the widest instruction is infrequent, the size of the control path can be reduced because the extra cycles needed to fetch instructions larger than the packet size will rarely be incurred. The computation of the packet size can be optimized by finding the smallest packet size that will provide a desired fetch performance for a particular application or a set of application programs.

The protocol specifies the method for fetching instructions from the ICache and the types of components in the IUDatapath. In the current implementation, the protocol includes a prefetch packet buffer, an On Deck Register (OnDeckReg or ODR) and an instruction register (IR). As shown in FIG. 20, the sequencer 900 is connected to the instruction cache 904 via control lines 906. These control lines include ICache address lines used to specify the next instruction to be fetched into the IUDatapath. Through these control lines, the sequencer 900 selects the packet and initiates the transfer of each packet of instructions from the instruction cache to a First-In, First-Out (FIFO) buffer 908.

The cache access time T_A is an ICache parameter provided as input to the control path design process. It is the time taken in cycles between the point when an address is presented to the address port of the ICache and when the corresponding data is available on its data port for reading. The cache line size parameter defines the width of a cache line in quanta. The control path design process selects a cache line size that is greater or equal to the packet size and is expressed as a number of quanta that must be a power of two. Although not necessary, this implies that in our current implementation a cache line contains an integral number of instruction packets.

The IUDatapath begins at the ICache and flows into the FIFO 908 via data lines 910. The number of data lines is defined as the instruction packet size in quanta. The FIFO 908 temporarily stores packets of instructions on their way to the instruction register 912. The objective in designing the FIFO is to make it deep enough to cover the latency of sequential instruction fetching from the instruction cache. The control path must be able to issue instructions to the instruction register to satisfy a desired performance criterion. In this case, the protocol defines the performance criterion as a rate of one instruction issue per clock cycle of the processor. Note, one instruction may contain several operations that are issued concurrently.

The IU Control 903 is responsible for controlling the flow of instruction packets from the FIFO 908 to a register that holds the next packet of instructions to be issued to the instruction register, called the ODR 914. In the example shown in FIG. 20, the IU Control 903 controls the flow of

instruction packets from the FIFO to the ODR 914 through control lines 916 to the FIFO 908, and control lines 918 to a multiplexor 920. The control lines 916 from the IU Control to the FIFO are used to accept new instruction packets from the ICache and to instruct the FIFO to transfer the next instruction packet to the ODR via data lines 922 from the FIFO to the multiplexor 920 and data lines 924 from the multiplexor to the ODR. As explained above, the size of this data path is defined via the instruction packet size parameter.

The IU Control 903 issues control signals 918 to the multiplexor 920 to select an instruction packet either from the FIFO 908 or directly from the instruction cache 904. The data path 926 is useful in cases where the FIFO has been cleared, such as when the processor has executed a branch instruction and needs to load the instruction packet containing the target of the branch into the ODR as quickly as possible.

The size of the FIFO (in packets) is another parameter in the control path protocol. The size of the FIFO depends upon the maximum and minimum instruction widths of instructions in the instruction format as well as the ICache access time. The width of an instruction may be as large as the maximum instruction width, and may be as small as the minimum instruction width in the instruction format specification. This constraint is merely a design choice in the current implementation, and is not necessary. The minimum instruction width plays an important role in determining the size of the FIFO because, in an extreme case, the ODR may be filled entirely with instructions of minimum size. In this case, the FIFO needs to be large enough to be filled with instruction packets already in flight from the ICaches as each of the instructions is issued sequentially from the ODR. The maximum instruction width also has an impact on the size of the FIFO because, in the opposite extreme, the ODR may contain a single instruction. In this case, the FIFO must be able to supply an instruction packet to the ODR at the desired performance rate, namely, once per clock cycle, while hiding the ICache access latency.

The parameters associated with the instruction fetch process include the size of the FIFO and the branch latency. These parameters are computed as shown below. The necessary FIFO size can be computed based on IUdatapath parameters and the instruction fetch policy. In case the policy does not allow for stalling the processor due to interrupts, then the FIFO size can be reduced further.

$$N_{FIFO} // \text{size of prefetch FIFO (packets)} (N_{FIFO} = \lceil T_A * W_{max} / W_A \rceil)$$

$$T_B // \text{branch latency } (T_B = T_{dpath} + T_A + 1)$$

The IU Control 903 controls the transfer of each instruction from the ODR 914 to the instruction register 912. The IU Control provides control signals via control lines 927 to the ODR, which in turn transfers the next instruction to the instruction register 912 via data lines 928 and an alignment network 930. The alignment network is responsible for ensuring that each instruction is left aligned in the instruction register 912. In the example shown in FIG. 20, the alignment network is comprised of a multiplexor for each quantum in the instruction register. Each of these multiplexors indicates where the next quantum of data will originate from in the ODR 914 or the IR 912. The IU Control 903 provides multiplexor select controls via control lines 932 based on parameters fed back from the decode logic via control lines 934.

The control path protocol outlines the operation of the alignment network. There are two principle modes of operation that the protocol of the alignment network must address: sequential instruction fetch mode; and branch target instruc-

tion fetch mode. FIG. 21 illustrates the operation of the shift network protocol for sequential instruction fetching, and FIG. 22 illustrates the operation of the shift network for branch target instruction fetching. Before describing the operation of the shift network in more detail, we begin by describing the relevant parameters associated with the shift network. The parameters in the current implementation are as follows:

W_{IR} // width of instruction register (quanta) ($W_{IR} = W_{max}$)

W_{curr} // width of current instruction (quanta)

$W_{consumed}$ // width of already used part in ODR (quanta)

P_{target} // position of branch target in ODR (quanta)

As noted previously, the shift network controls where each bit of data in the instruction register comes from. This data may come from the IR, the ODR, or in some cases, from both the ODR and the top instruction packet in the FIFO. With each cycle, the shift network ensures that the next instruction to be executed is left aligned in the instruction register. In doing so, it may shift unused bits within the instruction register itself, it may transfer bits from the ODR, and finally it may also transfer bits from the top of the FIFO. In particular, if the instruction register contains unused bits from the previous cycle representing part of the next instruction, it shifts these unused bits over to the left, and then fills in the rest of the instruction register with the next group of bits sufficient to fully load the register.

As noted above, the FIFO transfers instructions to the OnDeck register in packets. A packet remains in the ODR, and is incrementally consumed as the alignment network transfers portions of the bits in the ODR into the instruction register. The IU Control supplies control signals via control lines 936 to the instruction register 912 to issue the current instruction to the decode logic. The PC 938 in the sequencer specifies the memory address of the instruction currently being issued for execution.

8.2.1 The Alignment Network Protocol

FIG. 21 illustrates the two principle cases that occur in the shift network protocol for sequential instruction fetching. The first case is where the width of the current instruction in the instruction register, W_{curr} , is less than the remaining, unconsumed portion of the ODR, $W_A - W_{consumed}$. FIG. 21 illustrates an example of this scenario by showing the transition of the state of the instruction register, ODR, and FIFO from one cycle to the next. In the first cycle 1000, the current instruction occupies the left-most section (see section 1002) of the instruction register, while a part of the next instruction occupies the remaining section 1004. Also, a portion 1006 of the ODR is already consumed, and the remaining section 1008 contains valid data. In this case, the shift network shifts the unused portion 1004 to the left of the instruction register (see section 1010 representing the transfer of the bits from the right of the instruction register to the left-most position). In addition, the shift network transfers enough bits to fill in the remainder of the instruction register (see section 1012) from the left-most valid data portion 1008 in the ODR.

In the next cycle 1014, the instruction register contains the current instruction, aligned to the left, and a portion of the next instruction. The length of the current instruction becomes known only after decoding. The ODR contains a consumed portion 1016, which includes portions that the shift network already transferred in previous cycles. It also contains a remaining valid data portion 1018. The FIFO remains unchanged in this case.

The bottom diagrams 1030, 1032 in FIG. 21 illustrate the case where the width of the current instruction is greater than the valid data portion ($W_A - W_{consumed}$). In this case, the

current instruction occupies a relatively large section 1034 of the instruction register and the remaining portion 1036 contains part of the next instruction. The consumed portion 1038 of the ODR is relatively large compared to the remaining valid data portion 1040. As a result, the shift register needs to transfer data from three sources: the unused portion 1036 of the instruction register (shown being transferred in graphic 1042), the entire valid data portion remaining in the ODR 1040 (shown being transferred in graphic 1044), and finally, a portion in the top packet of the FIFO that is needed to fill in the rest of the instruction register (shown being transferred in graphic 1046). Since the ODR is fully consumed, the top packet of the FIFO needs to be advanced to the ODR. However, this example shows that a portion of the packet in the top of the FIFO is already consumed when the packet is transferred into the ODR (see section 1048 being transferred into the ODR), which leaves a consumed portion 1050 in the OnDeck register.

FIG. 22 illustrates the two principle cases that occur in the shift network protocol for branch target instruction fetching. When the processor executes a branch instruction, the control path should load the instruction containing the target of the branch as quickly as possible. There are a variety of schemes to accomplish this objective. Even within the specific protocol described and illustrated thus far, there are alternative ways to define the target fetch operation. In the example shown in FIG. 22, the target of a branch is allowed to reside anywhere in an instruction packet. This may result in the case where the next portion of valid data to be loaded into the instruction register (the target data) spans two instruction packets. One way to avoid this case is to require the application program compiler to align branch targets at the beginning of instruction packets. However, the example shown in FIG. 22 is more general and handles the case where the target data spans instruction packets.

The top diagrams 1100, 1102 illustrate the case where the target data is entirely within an instruction packet. This case is defined as a packet where the width of the instruction register, W_{IR} , is less than or equal to the width of a packet, W_A , less the position of the target instruction relative to the start of the packet, P_{target} . In the first cycle 1100, the current instruction occupies the left-most portion 1104 of the instruction register. In the shift operation, the entire contents of the instruction register are considered invalid. As such, the shift network fills the instruction register with new bits sufficient to fill it entirely (as shown in graphic 1106). The starting bit in the ODR for this shift operation is identified by P_{target} (see invalid portion 1108 in the ODR, which has a width P_{target}). Since the width of the instruction register plus P_{target} is still less than or equal to W_A , all of the new data comes from the ODR. After the shift, the consumed portion of the ODR occupies the left-most portion 1110 and some valid data for the next instruction may reside in the remaining portion 1112.

The bottom two diagrams 1120, 1122 show the case where the target data spans an instruction packet. This case is defined as a packet where the width of the instruction register, W_{IR} , is greater than the width of a packet, W_A , less the width of the offset to the target instruction inside the packet, P_{target} . In the first diagram 1120, the current instruction occupies the left-most portion 1124 of the instruction register. In the shift operation, the entire contents of the instruction register are considered invalid. As such, the shift network fills the instruction register with new bits sufficient to fill it entirely, but to do so, it must take bits from the ODR and the next packet from the ICache (as shown in graphics 1126 and 1128). The starting bit in the ODR for this shift

operation is identified by P_{target} (see invalid portion 1130 in the ODR, which has a width P_{target}). Since the width of the instruction register plus P_{target} is greater than W_A , some of the new data comes from the ODR and some comes from the next packet from the ICache. To get the target data into the instruction register, the control path may require two cycles. The shift network transfers valid bits from the ODR (as identified by P_{target}) to the IR and transfers the next packet (1132) from the ICache into the ODR. It then transfers valid bits from the ODR (1128) sufficient to fill the IR. This leaves a portion of the bits in the ODR 1134 ($W_{IR} - (W_A - P_{target})$) invalid.

The shift network protocol outlined above specifies how the IU Control logic controls the select ports of the multiplexors in the shift network in order to make the selection of the appropriate quanta in the IR, ODR, and FIFO. Further details about the synthesis of the shift network are provided below.

The final aspect of the control path protocol is the decode logic. Referring again to the example in FIG. 20, the decode logic (e.g., decode units 940-944) interfaces with the instruction register, decodes the current instruction, and dispatches control signals to the control ports in the data path. The CP synthesizer computes decode tables from the instruction format design as explained below.

8.3 Control Path Design

FIG. 23 is a flow diagram illustrating the operation of a software implementation of the CP synthesizer illustrated in FIG. 19. The CP synthesizer is implemented in the C++ programming language. While the software may be ported to a variety of computer architectures, the current implementation executes on a PA-RISC workstation or server running under the HP-UX 10.20 operating system. The functions of the CP synthesizer software illustrated in FIG. 23 are described in more detail below.

8.3.1 Collecting Parameter Values

The CP synthesizer begins by collecting and adjusting input parameters, Q_1 , W_{imax} , W_{imin} , W_A , T_A , and W_L as shown in step 1200. It calculates Q_1 as the greatest common denominator of all possible instruction widths and fetch widths. It extracts W_{imax} , W_{imin} from the instruction format, and derives W_A and possibly adjusts W_L as defined above. The ICache access time T_A is one of the ICache input parameters to the control path design.

The CP synthesizer computes $\#W_{curr}$ bits, a parameter that defines the number of bits needed to represent the length of the current instruction in quanta. The length of the current instruction may be zero or as large as W_{imax} . Therefore, W_{curr} bits is computed as $\lceil \log_2(W_{imax} + 1) \rceil$. The IU Control receives W_{curr} from the decode logic (See lines 934 in FIG. 20) and uses it to compute the appropriate shift amount for the shift and align network. The sequencer also uses this number to update the PC with the address of the next instruction to execute. The CP synthesizer determines the number of instruction register multiplexor selection bits $\#IRmux_{sel}$ bits as shown in step 1200, from the following expression: $\#IRmux_{sel} \text{ bits} = \lceil \log_2(W_A + W_{imax} - W_{imin}) \rceil$ in bits. This is the number of bits needed to select between $(W_A + W_{imax} - W_{imin})$ input quanta choices for each quantum multiplexor placed before the instruction register.

8.3.2 Allocating the Instruction Register and Sequencer

Next, the CP synthesizer selects an instruction register from the macrocell database as shown in step 1202, and sets the width of the instruction register equal to W_{imax} .

The CP synthesizer also selects a sequencer from the macrocell database in step 1204. The sequencer includes logic to process the branch addressing, logic to handle

interrupts and exceptions and logic to issue instruction fetching from the ICache. The choice of the sequencer depends on the architectural requirements specified during the design of the datapath and the instruction format, i.e., whether the processor needs to handle interrupts and exceptions, branch prediction, and control and data speculation. It is independent of the design of the instruction unit data path itself. Therefore, we assume that we have a set of predesigned sequencer macrocells available in the macrocell database from which one is selected that matches the architectural parameters of the datapath and the instruction format.

8.3.3 Building the Instruction Decode Logic

The CP synthesizer generates decode logic from the instruction format specification, which is provided in the IF tree 1206. This section describes how the CP synthesizer generates the decode tables programmatically.

The CP synthesizer generates the decode logic by creating decode tables that specify the inputs and outputs of the decode logic. In building a decode table, the CP synthesizer specifies the input bit positions in the instruction register, the input values for these bit positions, the corresponding control ports, and finally, the output values to be provided at these control ports in response to the input values. There are two general cases: 1) creating decode table entries for select fields (e.g., bits that control multiplexors and tri-state drivers); and 2) creating decode table entries for logic that converts opcodes. In the first case, the CP synthesizer generates the address selection logic needed to map bit positions in the instruction register with shared address control ports in the data path. It also generates the appropriate select values based on the select field encoding in the instruction template. In the second case, the CP synthesizer generates the opcode input values needed to select a particular opcode in a functional unit based on the opcode field encoding in the instruction template. Both of these cases are described further below.

The implementation divides the decode logic into two types of components: the template decode logic (synthesized in step 1208) and the FU decode logic, one per FU macrocell (synthesized in step 1210). The template decode logic is responsible for decoding all the information that is relevant for the entire instruction including the template width, the end-of-packet bit and the position of register file address port bits. The FU decode logic decodes all the information that is relevant for one FU macrocell including its opcode and the select ports of the data multiplexors and tri-state drivers. In step 1208, the CP synthesizer constructs a decode table for a template decode programmable logic array (PLA). As shown in the example FIG. 20, the template decode PLA provides information (W_{curr} and EoP parameter values) to the IU Control to drive the instruction shifting network. It converts the template ID into W_{curr} and feeds this information to the IU Control. It also provides the consume to end-of-packet (EoP) bit to the IU Control.

Based on the template ID, the template decoder also generates the mux select inputs in cases where instruction fields from different templates map to the same control ports in the datapath. For example, it computes select values for the mux select ports of register file address port multiplexors (RF port $addrmux_{sel}$; see, e.g., multiplexor 836 in FIG. 20).

To illustrate decode logic generation for select fields, consider the example of the RF address port multiplexors. The CP synthesizer builds a decode table for the address port multiplexors by traversing the IF tree to find the template specifier fields. The template specifier in the instruction identifies the template to the decode logic. This is significant

because a number of different bit positions may map to the same register file address port depending on the instruction template. The Table 1 shows an example of this scenario.

TABLE 1

Template	Bit Positions	Mux Inputs	Mux select
T1	0-3	I1	00
T2	10-13	I2	01
T3	1-3, 10	I3	10
T4	10-13	I4	11

In the example shown above, four different sets of bit positions map to the same register file address ports, depending on the instruction template. The decode logic, therefore, needs to generate the appropriate mux select signal to map the appropriate bit positions in the instruction to the register file address ports depending on the template specifier bits.

For each template, the CP synthesizer traverses the IF tree to the template specifier field and adds the bit encoding to the decode table as an input. It finds the corresponding bit positions from different templates that map to the same register file address ports and assigns them to the input ports of a multiplexor. Finally, it assigns mux select values so that the decode logic instructs the mux to select the appropriate mux inputs depending on the template specifier.

To illustrate decode logic generation for opcode fields, consider an example where the bits used to encode the opcode field in the instruction do not match the number of bits used to encode the opcode on the functional unit macrocell. The CP synthesizer functional unit constructs the FU decode PLA in step 1210 in a similar fashion as the template decode PLA. In particular, it builds a decode table that maps instruction register bits to data path control ports of the functional units in the data path. It traverses the IF tree to find the fields for the FU opcode fields. The CP synthesizer finds the instruction register ports that these fields have been assigned, and maps them to the opcode control ports.

The opcode field in the IF tree identifies the desired operations in an operation group and the corresponding functional unit to the decode logic. The opcode in the instruction field may need to be translated into a different form so that it selects the proper operation in the functional unit. Table 2 shows an example of this scenario.

TABLE 2

Opcode encoding	FU input
00	0000
01	1011
10	1100
11	0010

In the above example, the instruction selects one of four different operations to be executed on a given functional unit in the data path. The functional unit, however, supports more operations, and thus, uses a four bit input code to select an operation. In this case, the CP synthesizer generates a decode table for decode logic that will select the proper operation based on the opcode encoding in the instruction register. To accomplish this, it traverses the IF tree to find the opcode field, and the corresponding bit encoding, control port assignment, and bit position for this field. The opcode field in the IF tree is annotated with information that maps a bit encoding in the instruction to a particular input encoding for a functional unit in the data path. The CP synthesizer assigns the inputs of the decode logic to the bit positions of

the opcode field, and assigns the outputs of the decode logic to the opcode control ports of the functional unit.

The FU decode logic for the control ports of the muxes and tri-states in the interconnect between the functional units and register files is generated based on the select fields at the 10 set level in the IF tree in a similar fashion as described above for the RF address MUXes.

Once the decode logic tables are created, a variety of conventional logic synthesizer tools may be used to create hardware specific decode logic from the decode tables which is not necessarily restricted to a PLA-based design.

8.3.4 Assembling the Instruction Unit

In step 1212, the CP synthesizer builds the remainder of the instruction unit, including the IUDatapath and the control logic between the IUDatapath and sequencer. In this step, the CP synthesizer allocates the FIFO, ODR, and alignment network by selecting AIR macrocells from the macrocell database and instantiating them. It maps the control ports of these components in the IUDatapath to the control outputs of the IU Control logic. The IU Control logic controls the behavior of the IUDatapath at each cycle by providing specific bit values for each of the control ports of the IUDatapath components. The logic may be specified as a behavioral description of a finite state machine (FSM). From this description, conventional logic synthesis may be used to generate the FSM logic that forms the IUDatapath control logic.

When it allocates the sequencer macrocell, the CP synthesizer allocates the sequencer ports responsible for ICache control and addressing and connects it to the corresponding ICache ports (see, e.g., 906, FIG. 20). The number of address lines depends on #W_{ICaddr} bits, the number of ICache address bits. The memory address register (MAR) 902 drives the address port of the ICache while a fetch request bit (FReq) generated by the IU Control logic controls when new instruction packet fetches are initiated.

The CP synthesizer allocates the FIFO (908, FIG. 20) by computing the size of the FIFO as described above and constructing a macrocell instance from the macrocell database with N_{FIFO} packet registers of width W_A and a number of control and data ports. The data output of the ICache is connected to the data input of the FIFO. The various FIFO control ports are driven by the corresponding ports of the IU Control logic (916, FIG. 20).

The CP synthesizer also allocates the ODR (914, FIG. 20) by constructing a macrocell instance of a register having a width W_A and having corresponding control and data ports. It synthesizes the ODR's input side multiplexor (920, FIG. 20) by constructing a multiplexor from the macrocell database having a width W_A. The two inputs of the multiplexor 920 are connected to the FIFO and the ICache respectively. The selection control and the ODR load control ports are driven by the corresponding ports from the IU Control logic (918, 926, FIG. 20).

The CP synthesizer additionally synthesizes the branch FU control and address lines to interconnect the branch control ports of the sequencer with control ports of the branch FU.

It further allocates the instruction register shift network (930, FIG. 20), and connects its control ports to the IU Control logic (932, FIG. 20). FIG. 24 illustrates aspects of the IUDatapath to illustrate how the CP synthesizer allocates the shift network. In what follows, we assume that the various quanta in the IR, the ODR, the FIFO, and the cache are logically numbered sequentially starting from 0 as shown in FIG. 24.

As explained above, the shift network has a multiplexor for each quantum in the instruction register numbered 0 through W_{IR}-1. In the following discussion, k represents the number of a given multiplexor (0 ≤ k ≤ W_{IR}-1).

Each quantum multiplexor k selects among all quanta between the following two extremes:

- 1) k+W_{imin} (last inst. was minimum size); and
- 2) k+W_A+W_{IR}-1 (last inst. was maximum size and all of ODR W_A+was consumed).

The CP synthesizer creates instances for each multiplexor with enough input ports to select among the number of quanta reflected above. This number is (k+W_A+W_{IR}-1)-(k+W_{imin})+1=W_A+W_{IR}-W_{imin}.

The choices for IU selection control for a quantum mux k is given by:

- 1) k+W_{curr} (sequential access and k+W_{curr}<W_{IR});
- 2) k+W_{curr}+W_{consumed} from ODR/FIFO (sequential access and k+W_{curr} ≥ W_{IR}); and
- 3) k+W_{IR}+P_{target} from ODR/FIFO (branch target access).

The choices for IU selection control for ODR/FIFO quantum k is given by:

- 1) k+W_A from FIFO (advance FIFO by a full packet);
- 2) (k-W_{IR}) % W_A from I-Cache output (load directly from I-Cache); and
- 3) no shift (disable ODR load/FIFO advance).

The CP Synthesizer generates the IU Control logic to control the shift network according to the constraints given above. The design of the IU Control logic is discussed below.

8.3.5 Building IU Control Logic

The instruction fetch protocol described above is implemented in control logic that keeps track of the packet inventory—the packets in flight, packets in the prefetch buffer, and the unconsumed part of the ODR. It also issues instruction cache fetch requests, FIFO load and advance requests, and an ODR load request at the appropriate times, and provides the appropriate selection control for the shift and align network and other multiplexors in the instruction pipeline. Finally, the control logic is also responsible for flushing or stalling the pipeline upon request from the sequencer due to a branch or an interrupt.

The control logic is expressed in the following pseudocode.

```

Pseudocode for IU Control Logic
Module IU Control (cachePktRdy, flushPipe, EOP: in boolean; Wcurr: in integer)
1: // Design time constants: pktSize (WA), invSize
   (TA, Wmax/WA)
2: // Internal state: numFIFOPkts(0), numCachePkts(0),
   Wconsumed(WA)
3: if (numFIFOPkts + numCachePkts < invSize) then
4:   Request I-Cache fetch; //launch fetches to keep
    
```

-continued

```

5:   numCachePkts++;           inventory constant
6:   endif
7:   if (cachePktRdy) then     //packets are ready TA
8:     numCachePkts--;         cycles later
9:     if (Wconsumed ≥ WA && numFIFOPkts > 0) then
10:      Load cachePkt into ODR; //put pkt directly into
11:      Wconsumed = 0;          ODR, if empty
12:     else                    //otherwise, save pkt in
13:      Load cachePkt into FIFO; FIFO
14:      numFIFOPkts--;
15:     endif
16:   endif
17:   if (Wconsumed ≥ WA && numFIFOPkts > 0) then //draw next pkt from FIFO
18:     Load FIFOPkt into ODR;
19:     Wconsumed = WA;
20:     advance FIFO;
21:     numFIFOPkts--;
22:   endif
23:   if (flushPipe) then      //branch or interrupt
24:     flush l-cache and FIFO; processing
25:     numCachePkts=0;
26:     numFIFOPkts=0;
27:     Wconsumed = WA;
28:   elseif (EOP) then       // skip to end-of-packet
29:     Shift IR to align to next pack boundary;
30:     Wconsumed = WA;
31:   else                    // shift to next
32:     Shift IR by Wcurr;      instruction
33:     adjust Wconsumed;
34:   endif

```

The control logic is expressed as pseudocode that consists of a sequence of conditions and various actions to be performed under those conditions. The logic keeps track of the inventory of packets internally including those in flight in the instruction cache pipeline (numCachePkts) and those sitting in the prefetch buffer (numFIFOPkts). This is used to issue a fetch request whenever the inventory size falls below the threshold (line 3). The corresponding instruction packet is ready to be read at the output of the cache T_A cycles after the fetch is initiated (line 7). This packet may be loaded directly into the ODR if the rest of the pipeline is empty (line 9), or it may be saved in the FIFO (line 12). These packets are later loaded into the ODR as needed (line 17).

Upon encountering a taken branch signal or an interrupt signal from the sequencer (flushPipe), the control logic flushes the instruction pipeline by resetting the internal state (line 23). This enables the pipeline to start fetching instructions from the new address from the next cycle. Otherwise, the next instruction in sequence needs to be aligned into the instruction register (line 28). If the end-of-packet (EOP) bit is set, the current packet residing in the ODR is considered to be fully consumed and the IR is shifted to the next packet available. Otherwise, the IR is shifted by the width of the current instruction. In either case, the multiplexors of the shift and alignment network in front of the IR are provided with the appropriate selection control as described above.

The control logic shown above may be synthesized into a finite-state machine (FSM) using standard synthesis tools that translate a functional description such as that given above and produce a concrete implementation in terms of gates or PLA logic along with control registers to keep track of the sequential state.

While we have illustrated a specific control path protocol, it is important to note that the control path synthesizer program can be adapted for a variety of different protocols. Both the structural and procedural aspects of the protocol may vary. The protocol may specify that the alignment network is positioned between the instruction register and

the decode logic. In this protocol, for example, the instruction register has a wider width (e.g., a width of one packet) and the alignment network routes varying width instructions from the instruction register to the decode logic. This protocol is based on a procedural model of "in-place" decoding, where instructions are not aligned in the IR, but rather, fall into varying locations in the IR. The protocol procedure defines a methodology to determine the start of the next instruction to be issued from the IR.

The procedural model may be based on a statistical policy where the width of the control path pipeline is optimized based on the width of the templates in the instruction format. In this approach, the control path designer minimizes the width of the pipeline within some performance constraint. For example, the width is allowed to be smaller than the widest instruction or instructions as long as the stall cycles needed to issue these instructions do not adversely impact overall performance. When the width of the pipeline is less than the widest instruction, one or more stall cycles may be necessary to issue the instruction to the decode logic. Performance is estimated based on the time required to issue each instruction and the corresponding frequency of the instruction's issuance.

9.0 Generating a Structural Description

The system produces a structural description of the processor hardware at the RTL-level in a standard hardware description language such as VHDL. This description can be linked with the respective HDL component libraries pointed to by the macrocell database and processed further for hardware synthesis and simulation.

CONCLUSION

While the invention is described in the context of a specific implementation, the scope of the invention is not limited to this implementation. A number of design variations are possible.

One possible variation is the manner in which the ILP constraints are specified. As noted above, the ILP constraints

may be specified as exclusion sets, concurrency sets, or some combination of both. The form of other input data structures, such as the register file specification and macrocell library may vary as well. These data structures may be provided in an external file form, such as a textual file (e.g., the ArchSpec which is in a tabular form using the HMDDES database language) or in an internal form (e.g., a separate user interface to specify register file data structures and a component-level interface to the standard HDL macrocell databases). The above description provides a number of constructs for specifying an opcode repertoire, the I/O formats of the opcodes and the desired ILP among the operations. However, these constructs are not critical to the implementation of the invention.

The AIR form of the datapath represents only one possible way to specify the output of the datapath design process. Other types of hardware description languages may be used as well, such as VHDL or Verilog. Indeed, the AIR form can be easily translated to one of these external textual formats. The current implementation produces VHDL output.

In view of the many possible implementations of the invention, it should be recognized that the implementations described above are only examples of the invention and should not be taken as a limitation on the scope of the invention. Rather, the scope of the invention is defined by the following claims. We therefore claim as our invention all that comes within the scope and spirit of these claims.

We claim:

1. A method for programmatic design of a VLIW processor from an input specification including specified processor operations, I/O formats for the specified operations, instruction level parallelism constraints among the specified operations, and a register file specification of the processor, the method comprising:

based on the specified processor operations, and the instruction level parallelism constraints, programmatically generating a datapath description of the processor from a macrocell library, the datapath description including functional unit instances, register file instances and an interconnect between the functional unit and register file instances; and

based on the datapath description, the I/O formats, and the instruction level parallelism constraints, programmatically generating an instruction format specification, including instruction templates representing VLIW instructions executable by the processor, instruction fields for each of the templates, and bit positions and bit encodings for the instruction fields.

2. The method of claim 1 further including: programmatically extracting a machine description suitable to re-target a compiler from the datapath description and the input specification.

3. The method of claim 2 further including: from the compiler, re-targeted using the machine description of the processor, generating operation issue statistics for the specified operations; using the operation issue statistics, selecting custom templates; and

using the custom templates as input to programmatically generate the instruction format specification of the processor.

4. The method of claim 1 wherein programmatically generating the instruction format specification includes: programmatically constructing a bit allocation problem specification identifying instruction fields that are to be assigned to bit positions in the instruction format of the

processor, bit width requirements of the instruction fields, and instruction field conflict constraints; and programmatically allocating bit positions in the processor to the instruction fields in the bit allocation problem specification.

5. The method of claim 4 including:

programmatically extracting a machine description suitable to re-target a compiler from the datapath description and the input specification;

from the compiler, re-targeted using the machine description of the processor, generating operation issue statistics for the specified operations;

using the operation issue statistics, selecting custom templates; and

using the custom templates as input to programmatically construct the bit allocation specification problem of the processor.

6. The method of claim 1 including:

using the instruction format specification, programmatically generating a controlpath description with components from the macrocell library, where the control path description includes a hardware description of an instruction unit datapath for transferring instructions from an instruction cache to an instruction register, a description of control logic for coupling an instruction sequencer to the instruction unit data path, and a description of decode logic for decoding instructions in the instruction register and issuing the instructions to control ports in the datapath.

7. A computer readable medium having software for performing the method of claim 1.

8. An automated VLIW processor design method comprising:

receiving as input a concrete instruction set architecture specification of a processor including an instruction format specification, and register file specification, wherein the instruction format specification includes instructions, instruction fields within each of the instruction templates, and bit positions and encodings for the instruction fields; and

wherein the register file specification enumerates register files in the processor, including a number of registers in each of the register files, and a correspondence between operand instruction fields in the instruction format specification and a register file;

programmatically extracting an abstract instruction set architecture specification including specified processor operations and instruction level parallelism constraints among the specified operations.

9. The method of claim 8 including:

based on the specified processor operations, and the instruction level parallelism constraints, programmatically generating a datapath description of the processor from a macrocell library, the datapath description including functional unit instances, register file instances and an interconnect between the functional unit and register file instances.

10. The method of claim 9 including:

programmatically extracting a machine description suitable to re-target a compiler from the datapath description and the abstract instruction set architecture specification.

11. The method of claim 9 including:

using the instruction format specification, programmatically generating a controlpath description with compo-

nents from the macrocell library, where the control path description includes a hardware description of an instruction unit datapath for transferring instructions from an instruction cache to an instruction register, a description of control logic for coupling an instruction sequencer to the instruction unit data path, and a description of decode logic for decoding instructions in the instruction register and issuing the instructions to control ports in the datapath.

12. A computer readable medium having software for performing the method of claim 11.

13. An automated VLIW processor design method comprising:

reading a datapath description of a VLIW processor in a hardware description language, the datapath description including functional unit instances, register file instances and a description of hardware components that form an interconnect path between the functional unit and register file instances; and

programmatically extracting an abstract instruction set architecture specification of the VLIW processor, the abstract instruction set architecture including processor operations, I/O formats for the specified operations, instruction level parallelism constraints among the specified operations, and a register file specification of the processor.

14. The method of claim 13 including:

based on the datapath description, the I/O formats, and the instruction level parallelism constraints, programmatically generating an instruction format specification, including instruction templates representing VLIW instructions executable by the processor, instruction fields for each of the templates, and bit positions and bit encodings for the instruction fields.

15. The method of claim 14 wherein programmatically generating the instruction format specification includes:

programmatically constructing a bit allocation problem specification identifying instruction fields that are to be assigned to bit positions in the instruction format of the processor, bit width requirements of the instruction fields, and instruction field conflict constraints; and

programmatically allocating bit positions in the processor to the instruction fields in the bit allocation problem specification.

16. The method of claim 13 including:

programmatically extracting a machine description suitable to re-target a compiler from the datapath description and the input specification.

17. The method of claim 14 including:

using the instruction format specification, programmatically generating a controlpath description with components from the macrocell library, where the control path description includes a hardware description of an instruction unit datapath for transferring instructions from an instruction cache to an instruction register, a description of control logic for coupling an instruction sequencer to the instruction unit data path, and a description of decode logic for decoding instructions in the instruction register and issuing the instructions to control ports in the datapath.

18. A computer readable medium having software for performing the method of claim 13.

19. A system for programmatic design of a VLIW processor from an input specification including specified processor operations, I/O formats for the specified operations, instruction level parallelism constraints among the specified

operations, and a register file specification of the processor, the system comprising:

a datapath synthesizer for reading the specified processor operations, I/O formats of the operations and the instruction level parallelism constraints, and for generating a datapath description of the processor from a macrocell library, the datapath description including functional unit instances, register file instances and an interconnect between the functional unit and register file instances; and

an instruction format designer for reading the datapath description, the I/O formats, and the instruction level parallelism constraints, and for generating an instruction format specification, including instruction templates representing VLIW instructions executable by the processor, instruction fields for each of the templates, and bit positions and bit encodings for the instruction fields.

20. The system of claim 19 further including:

an MDES extractor for extracting a machine description suitable to re-target a compiler from the datapath description and the input specification.

21. The system of claim 20 further including:

a custom template module for selecting custom templates using operation issue statistics for an application program generated by the compiler, re-targeted based on the machine description; and

wherein the custom templates are used as input to the instruction format designer to generate the instruction format specification of the processor.

22. The system of claim 19 wherein the instruction format designer includes:

a module for constructing a bit allocation problem specification identifying instruction fields that are to be assigned to bit positions in the instruction format of the processor, bit width requirements of the instruction fields, and instruction field conflict constraints; and

a bit allocation module for allocating bit positions in the processor to the instruction fields in the bit allocation problem specification.

23. The system of claim 22 including:

an MDES extractor for extracting a machine description suitable to re-target a compiler from the datapath description and the input specification; and

a custom template module for selecting custom templates using operation issue statistics for an application program generated by the compiler, re-targeted based on the extracted machine description; and

wherein the custom templates are used as input to the instruction format designer to generate the instruction format specification of the processor.

24. The system of claim 19 including:

a control path synthesizer for reading the instruction format specification, and for generating a controlpath description with components from the macrocell library, where the control path description includes a hardware description of an instruction unit datapath for transferring instructions from an instruction cache to an instruction register, a description of control logic for coupling an instruction sequencer to the instruction unit data path, and a description of decode logic for decoding instructions in the instruction register and issuing the instructions to control ports in the datapath.

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION

PATENT NO. : 5,909,544
DATED : June 1, 1999
INVENTOR(S) : Graham et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

On the title page section [75] Inventors please delete "Micheil" and insert therefor
--Michiel--.

Signed and Sealed this
Sixteenth Day of November, 1999

Attest:



Q. TODD DICKINSON

Attesting Officer

Acting Commissioner of Patents and Trademarks



US006192384B1

(12) **United States Patent**
Dally et al.

(10) Patent No.: **US 6,192,384 B1**
(45) Date of Patent: **Feb. 20, 2001**

(54) **SYSTEM AND METHOD FOR PERFORMING COMPOUND VECTOR OPERATIONS**

(75) Inventors: **William J. Dally**, Stanford; **Scott Whitney Rixner**, Mountain View, both of CA (US); **Jeffrey P. Grossman**; **Christopher James Buchler**, both of Cambridge, MA (US)

(73) Assignees: **The Board of Trustees of the Leland Stanford Junior University**, Stanford, CA (US); **The Massachusetts Institute of Technology**, Cambridge, MA (US)

(*) Notice: Under 35 U.S.C. 154(b), the term of this patent shall be extended for 0 days.

(21) Appl. No.: **09/152,763**

(22) Filed: **Sep. 14, 1998**

(51) Int. Cl.⁷ **G06F 9/28**

(52) U.S. Cl. **708/200; 712/1; 712/10; 712/16; 712/20; 712/21; 708/3**

(58) Field of Search **710/131-132; 711/147; 712/22, 21, 16, 10; 708/3, 200**

(56) **References Cited**

U.S. PATENT DOCUMENTS

- 4,807,183 * 2/1989 Kung et al. 710/132
- 5,327,548 * 7/1994 Hardell, Jr. et al. 711/147
- 5,522,083 * 5/1996 Gove et al. 712/22
- 5,692,139 * 11/1997 Slavenburg et al. 710/131

OTHER PUBLICATIONS

Rixner et al., "A bandwidth-efficient architecture for media processor." Proceedings on Annual ACM/IEEE Interna-

tional Symposium on Microarchitecture, p. 3-13, Nov., 1998.*

Borkar, et al. "iWarp: an integrated solution to high-speed parallel computing." Proceedings on Supercomputing, p. 330-339, Nov., 1988.*

* cited by examiner

Primary Examiner—Meng-Ai T. An

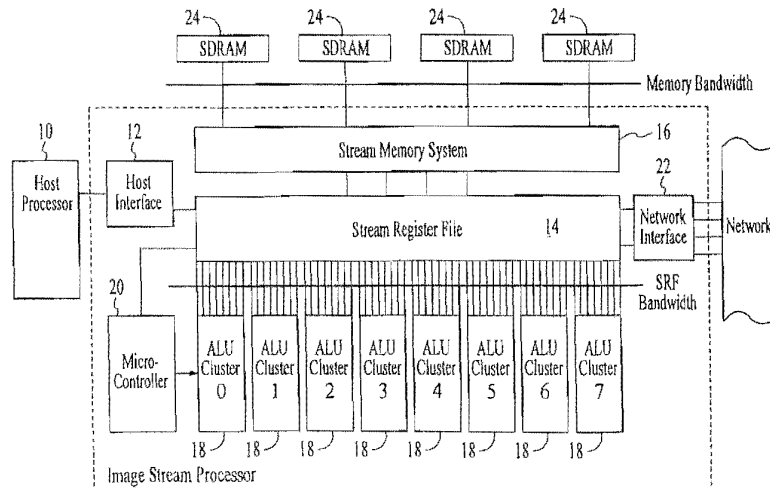
Assistant Examiner—Wen-Tai Lin

(74) *Attorney, Agent, or Firm*—Pillsbury Madison & Sutro, LLP

(57) **ABSTRACT**

A processor particularly useful in multimedia applications such as image processing is based on a stream programming model and has a tiered storage architecture to minimize global bandwidth requirements. The processor has a stream register file through which the processor's functional units transfer streams to execute processor operations. Load and store instructions transfer streams between the stream register file and a stream memory; send and receive instructions transfer streams between stream register files of different processors; and operate instructions pass streams between the stream register file and computational kernels. Each of the computational kernels is capable of performing compound vector operations. A compound vector operation performs a sequence of arithmetic operations on data read from the stream register file, i.e., a global storage resource, and generates a result that is written back to the stream register file. Each function or compound vector operation is specified by an instruction sequence that specifies the arithmetic operations and data movements that are performed each cycle to carry out the compound operation. This sequence can, for example, be specified using microcode.

29 Claims, 5 Drawing Sheets



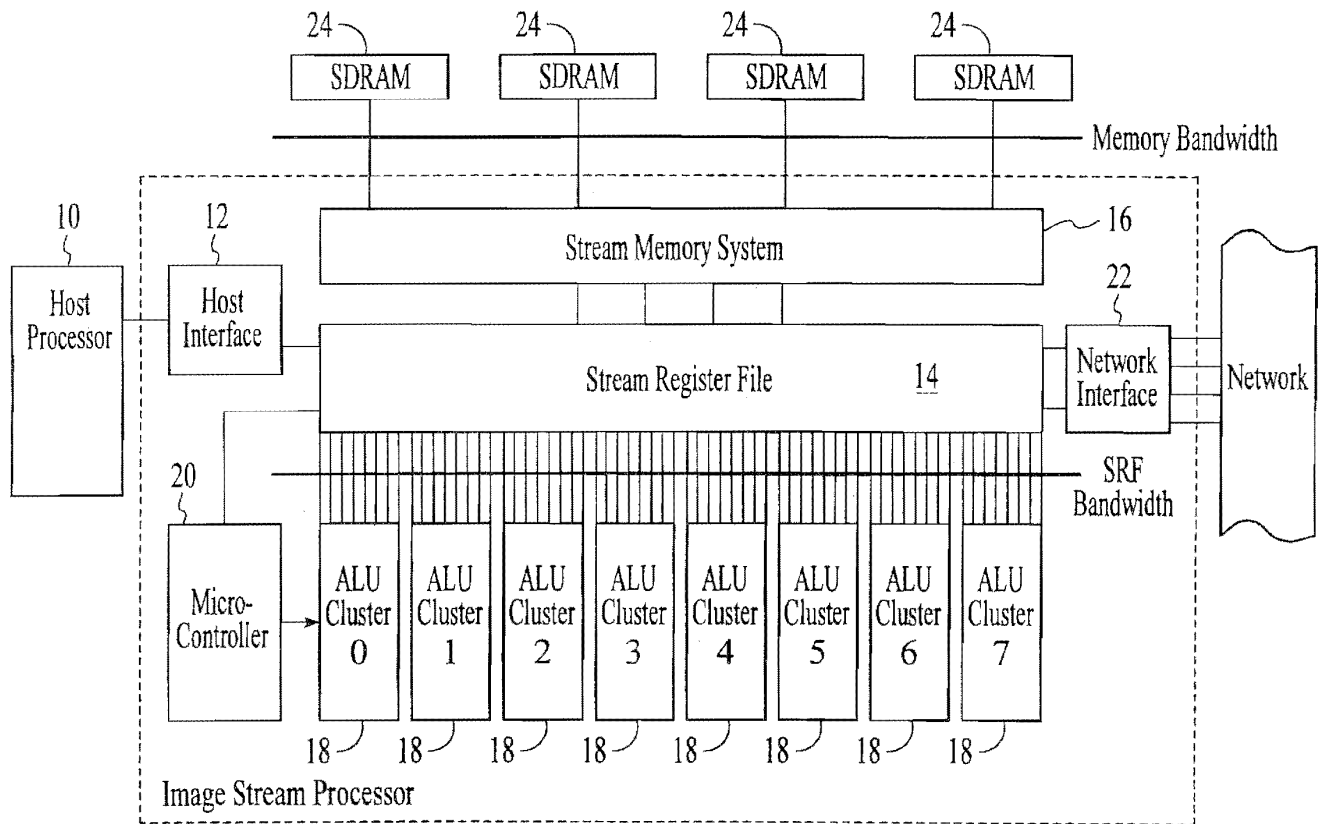


FIG. 1

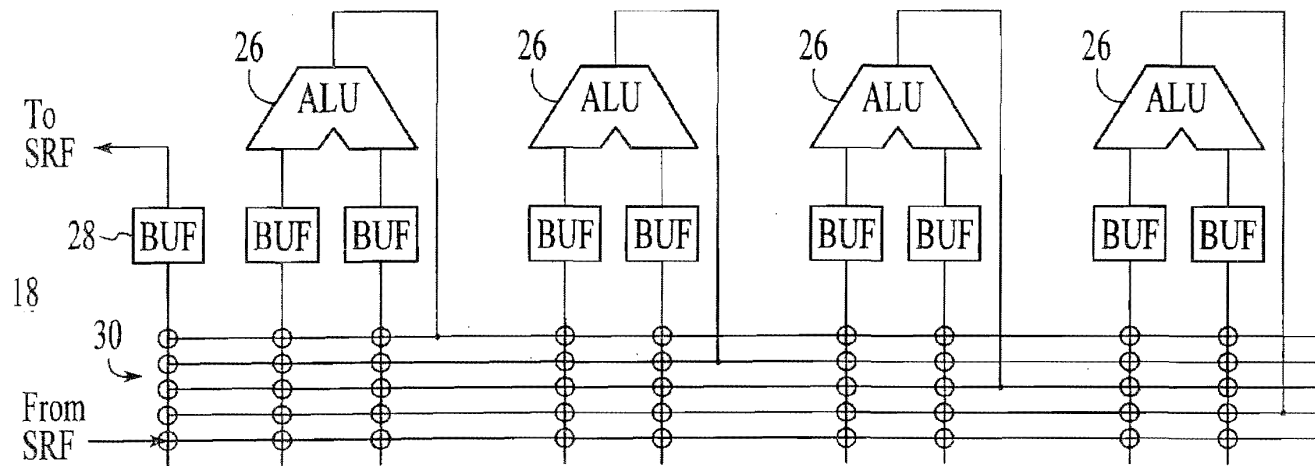


FIG. 2

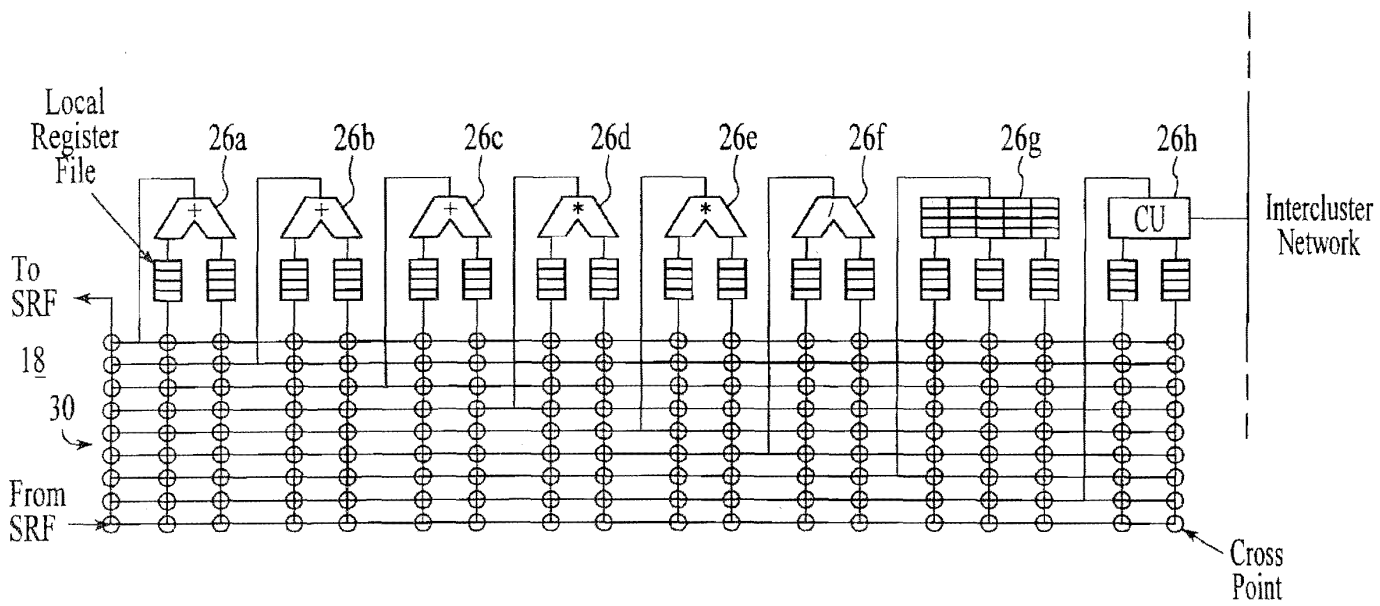
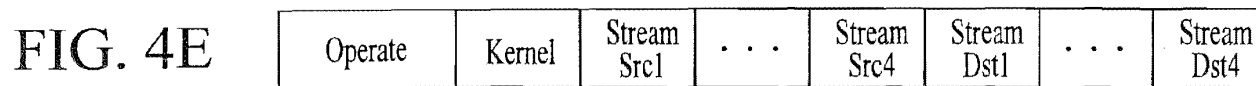
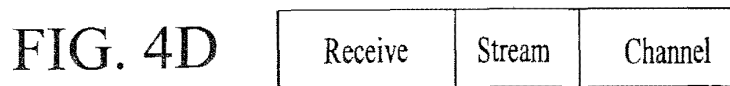
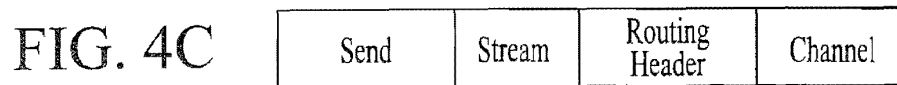
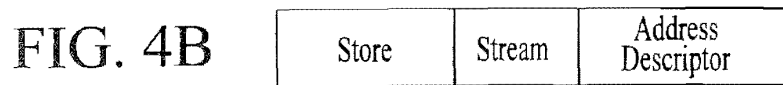
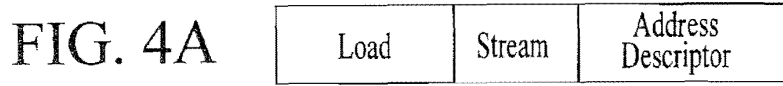


FIG. 3



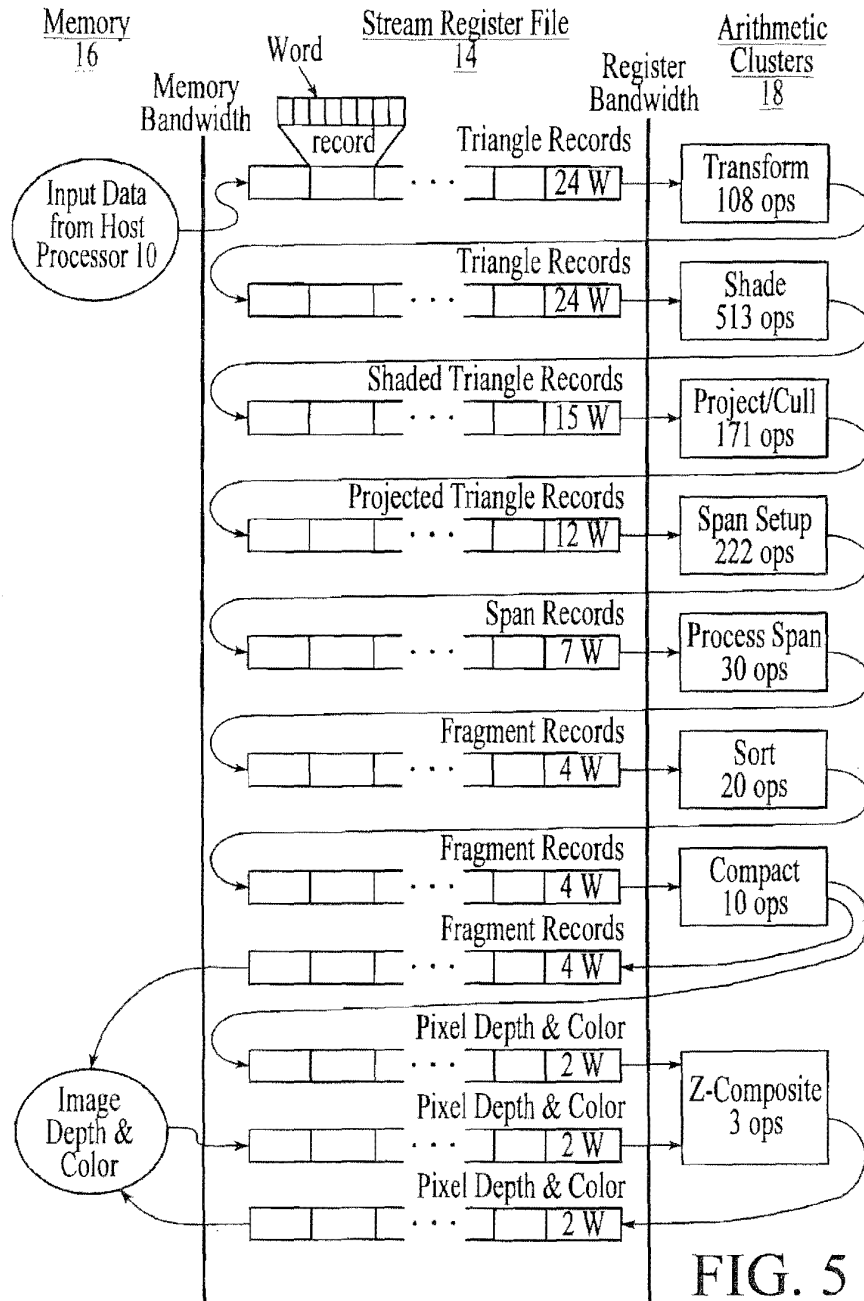


FIG. 5

WEST

SYSTEM AND METHOD FOR PERFORMING COMPOUND VECTOR OPERATIONS

This invention was made in conjunction with U.S. Government support under U.S. Army Grant No. DABT63-96-C-0037.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention is directed to computer architectures. More specifically, the invention is directed to pipelined and parallel processing computer systems which are designed to efficiently handle continuous streams of instructions and data.

2. Description of Related Art

Providing adequate instruction and data bandwidth is a key problem in modern computer systems. In a conventional scalar architecture, each arithmetic operation, e.g., an addition or multiplication, requires one word of instruction bandwidth to control the operation and three words of data bandwidth to provide the input data and to consume the result (two words for the operands and one word for the result). Thus, the raw bandwidth demand is four words per operation. Conventional architectures use a storage hierarchy consisting of register files and cache memories to provide much of this bandwidth; however, since arithmetic bandwidth scales with advances in technology, providing this instruction and data bandwidth at each level of the memory hierarchy, particularly the bottom, is a challenging problem.

Vector architectures have emerged as one approach to reducing the instruction bandwidth required for a computation. With convention vector architectures, e.g., the Cray-1, a single instruction word specifies a sequence of arithmetic operations, one on each element of a vector of inputs. For example, a vector addition instruction VADD VA, VB, VC causes each element of an, e.g., sixty-four element vector VA to be added to the corresponding element of a vector VB with the result being placed in the corresponding element of vector VC. Thus, to the extent that the computation being performed can be expressed in terms of vector operations, a vector architecture reduces the required instruction bandwidth by a factor of the vector length (sixty-four in the case of the Cray-1).

While vector architectures may alleviate some of the instruction bandwidth requirements, data bandwidth demands remain undiminished. Each arithmetic operation still requires three words of data bandwidth from a global storage source shared by all arithmetic units. In most vector architectures, this global storage resource is the vector register file. As the number of arithmetic units is increased, this register file becomes a bottleneck that limits further improvements in machine performance.

To reduce the latency of arithmetic operations, some vector architectures perform "chaining" of arithmetic operations. For example, consider performing the above vector addition operation and then performing the vector multiplication operation VMUL VC VD VE using the result. With chaining, the vector multiply instruction consumes the elements computed by the vector add instruction in VC as they are produced and without waiting for the entire vector add instruction to complete. Chaining, however, also does not diminish the demand for data bandwidth—each arithmetic operation still requires three words of bandwidth from the vector register file.

BRIEF SUMMARY OF THE INVENTION

In view of the above problems of the prior art, it is an object of the present invention to provide a data processing

system and method which can provide a high level of performance without a correspondingly high memory bandwidth requirement.

It is another object of the present invention to provide a data processing system and method which can reduce global storage resource bandwidth requirements relative to a conventional scalar or vector processor.

It is a further object of the present invention to provide a parallel processing system and method which minimizes the number of external access operations each processor conducts.

It is yet another object of the present invention to provide a parallel processing system and method which utilizes granular levels of operation of a higher order than individual arithmetic operations.

It is still another object of the present invention to provide a parallel processing system and method which is capable of simultaneously exploiting multiple levels of parallelism within a computing process.

It is yet a further object of the present invention to provide a single-chip processing system which reduces the number of off-chip memory accesses.

The above objects are achieved according to a first aspect of the present invention by providing a processor having a tiered storage architecture to minimize global bandwidth requirements. The processor has a stream register file through which the processor's arithmetic units transfer streams to execute processor operations. Load and store instructions transfer streams between the stream register file and a stream memory; send and receive instructions transfer streams between stream register files of different processors; and operate instructions pass streams between the stream register file and computational kernels.

Each of the computational kernels is capable of performing compound vector operations. A compound vector operation performs a sequence of arithmetic operations on data read from the stream register file, i.e., a global storage resource, and generates a result that is written back to the stream register file. Each function or compound vector operation is specified by an instruction sequence that specifies the arithmetic operations and data movements that are performed each cycle to carry out the compound operation. This sequence can, for example, be specified using microcode.

Because intermediate results are forwarded directly between arithmetic units and not loaded from or stored to the stream register file, bandwidth demands on the stream register file are greatly reduced and global storage bandwidth requirements are minimized.

For example, consider the problem of performing a transformation on a sequence of points, a key operation in many graphics systems when, e.g., adjusting for perspective or moving from a model space to a world space. In its most basic form, the operation requires reading three words of data for each point (x, y, z) , performing a 4×4 vector-matrix multiply, taking the reciprocal of a number, performing three multiplies, and writing the resulting point (x', y', z') in the new coordinate system. Without optimizations, the perspective transformation requires thirty-two arithmetic operations for each point—nineteen multiplications, twelve additions and one reciprocal operation. On conventional vector architectures, this would require ninety-six words of vector register bandwidth per point.

In contrast, a compound vector architecture as described in greater detail below can perform the perspective trans-

formation in a single operation. The compound vector operation requires only six words of global bandwidth storage per point: three words to read the coordinates of the original point (x, y, z) and three words to write the coordinates of the transformed point (x', y', z'). All of the intermediate results are forwarded directly between arithmetic units and thus do not require global storage bandwidth. This sixteen-fold reduction in vector register bandwidth greatly improves the scalability of the architecture. In effect, the compound vector architecture moves the vector register file access outside of a function such as perspective transformation.

BRIEF DESCRIPTION OF THE DRAWINGS

The above and other objects of the present invention will become readily apparent when reading the following detailed description taken in conjunction with the appended drawings in which:

FIG. 1 is a block diagram of a graphics processor according to a preferred embodiment of the invention;

FIG. 2 is a diagram of an arithmetic cluster used in the graphics processor;

FIG. 3 is a diagram of an arithmetic cluster having variegated functional elements;

FIGS. 4A-4E show the structure of the instruction set of the graphics processor; and

FIG. 5 depicts the flow of data between kernels in the graphics processor when performing a triangle rendering operation.

DETAILED DESCRIPTION OF PRESENTLY PREFERRED EMBODIMENTS

First, the overall architecture of an exemplary computer system employing a preferred embodiment of the present invention will be described.

Central to the operation of this preferred embodiment are the concepts of streams and kernels. A stream is a sequence of elements made up of a collection of related data words. A stream may be received by a computation kernel which executes the same operation on all of the elements in the stream to produce another stream that can be output or sent to other kernels for further processing.

Kernels are relatively small computational units that may only access local variables, read input streams and write to output streams. They cannot make arbitrary memory references. In a preferred embodiment of the invention, the computation kernels are expressed in a C-like programming language and compiled into microcode programs that sequence the operation of arithmetic clusters to carry out compound stream operations on each element in a stream. The operations implemented by the kernels are called compound operations because in contrast to conventional vector or stream operations which perform only one operation on each vector element, each kernel performs multiple arithmetic operations on each stream element. A compound stream operation is a small program that has access to the record at the head of each of its input streams and to its local variables. The kernel reads the input streams and writes to the output streams using explicit instructions. The length and record size of each stream can be different and the number of input and output streams need not be the same.

With this foundation in mind, FIG. 1 shows a preferred embodiment of the present invention used in a high speed graphics coprocessor. Here, a host processor 10 provides data to the graphics coprocessor via a host interface 12. The

data from the host processor 10 is stored in a stream register file 14 which is the center of activity in the graphics coprocessor. The host interface 12, stream memory 16, arithmetic clusters 18, microcontroller 20 and network interface 22 all interact by transferring streams of data and instructions to and from the stream register file 14.

The system has a three-level storage hierarchy consisting of the stream memory 16 as a global storage unit, the stream register file 14 as an intermediate storage unit, and local register files 28 (see FIG. 2) in the arithmetic clusters 18 as local storage units. The stream memory 16 holds persistent data; the stream register file 14 stores streams as they are passed to, from and between computation kernels, and the arithmetic clusters 18 use the local register files to store intermediate results produced during computations within the cluster so they do not need to recirculate through the stream register file 14.

The stream register file 14 is preferably a 64 kB memory organized to handle streams of data and instructions (of course, the size of the stream register file may be varied according to the application). An array of eighteen 64 word stream buffers are used to allow read/write access to eighteen streams simultaneously. The internal memory array is thirty-two 32-bit words (i.e., 1024 bits) wide so that it can fill or empty half a stream buffer each cycle. Each stream client may access its dedicated stream buffer every cycle if there is data available to be read or space available to be written. The clients of eight of the stream buffers are the eight clusters 18, and these stream buffers are accessed eight words at a time. The remaining ten stream buffers are accessed a single word at a time.

The stream memory system 16 can perform two simultaneous memory transfers between four thirty-two bit wide SDRAM banks 24 and the stream register file 14 via four stream buffers (two for data and two for indices) in the stream register file 14.

The eight arithmetic clusters 18 connected to the stream register file 14 are controlled by the microcontroller 20. Each cluster 18 operates on one record of a stream so that eight records can be processed simultaneously. An exemplary internal structure of an arithmetic cluster, shown in FIG. 2, includes four functional elements 26 each buffered by one of the local register files 28 which stores kernel constants, parameters and local variables, thereby reducing the bandwidth load on the stream register file 14.

The local register files 28 themselves are fed by a crosspoint switch 30 which distributes outputs of the functional elements 26 to inputs thereof as intermediate data for use in subsequent arithmetic operations. The output of each functional element 26 is connected to one of the input lines of the crosspoint switch 30, and the input of each local register file 28 is fed by a corresponding output line of the crosspoint switch 30. Additionally, one of the crosspoint input lines is fed by the stream register file 16 to provide the contents of the stream dedicated to that cluster, and one of the crosspoint output lines is returned to the stream register file 16 for writing into that stream.

A specific implementation of the arithmetic cluster 18 structure is shown in FIG. 3 in which three adders 26a-26c, two multipliers 26d and 26e, a divider/square root unit 26f, a 128 entry scratchpad register file 26g, and an inter-cluster communication unit 26h (hereinafter collectively referred to as functional elements 26) are employed as functional elements 26.

The scratch pad register file 26g can be indexed with a base address specified in an instruction word and an offset

specified in a local register and may be used for coefficient storage, short arrays, small lookup tables and some local register spilling. The adders 26a-26c and multipliers 26d and 26e have latencies of four and five cycles, respectively, are fully pipelined and perform single precision floating point arithmetic, 32-bit integer arithmetic, and 8-bit or 16-bit parallel subword integer operations. The adders 26a-26c also are able to perform 32-bit integer and parallel subword integer shift operations. The divider/square root unit 26f is not pipelined and operates only on single precision floating point and 32-bit integers.

Finally, the intercluster communication unit 26h performs data transfer among clusters using arbitrary communication patterns. This is particularly useful in applications such as Fast Fourier Transforms where interaction is required between adjacent stream elements.

The microcontroller 20 receives kernels as compiled VLIW microcode programs from the host processor 10. The microcontroller 20 executes each of the kernels as an independent process using the arithmetic clusters 18 for performing computational operations.

The network interface 22 connects the stream register file 14 to four bidirectional links that can be used to connect the graphics processor to other like processors.

Preferably, a substantial portion of the graphics coprocessor, particularly including the stream register file 14, arithmetic clusters 18 and microcontroller 20, are implemented on a single chip using VLSI techniques. This is particularly advantageous because it allows accesses within the arithmetic clusters 18 and accesses to the stream register file 14 to be internalized, thus freeing up more of the pin bandwidth to be used for communication with the stream memories 24. In fact, it appears that a coprocessor as disclosed herein can be implemented on a 1 cm² 0.25 μm CMOS chip operating at 400 MHz and perform up to 16 billion operations per second.

The application-level instruction set used by the host processor 10 to program the graphics coprocessor is shown in FIGS. 4A-4E. The set consists of two complementary Load and Store instructions which are used to move streams between the stream register file 14 and the stream memory 16. As shown in FIGS. 4A and 4B, each instruction consists

FIGS. 4C and 4D show send and receive instructions which allow streams to be passed from the stream register file of one graphics coprocessor to that of another. These instructions are particularly advantageous because they allow multiple processors to operate in cooperation and provide extensibility and scalability. The Send instruction shown in FIG. 4C includes the stream to be sent, a routing header identifying the external coprocessor to which the stream is sent, and a channel indicator designating the communications channel used so that a single node can discriminate between arriving messages. Similarly, the Receive instruction of FIG. 4D includes the stream to be received and a channel indicator designating the communications channel for node discrimination of multiple messages.

Finally, the Operate instruction invokes a kernel to perform its compound stream operation on one or more input streams to generate one or more output streams. The instruction includes a kernel field designating the kernel to be activated, up to four input stream designators which identify streams to be used to provide input data to the kernel's compound stream operation, and up to four output stream designators which identify streams to which results of the compound stream operations are provided.

The host processor 10 issues these application-level instructions to the coprocessor with encoded dependency information which specifies the system resources and data needed to execute the instructions. The host interface 12 buffers these instructions and, when their requirements are satisfied, issues them to the coprocessor. The host interface 12 also maps the coprocessor to the host's address space so that the host can read and write to the stream memory 16 and execute programs that issue the appropriate application-level instructions to the coprocessor.

Using this architecture, substantial improvements in memory bandwidth use minimization can be realized. Consider, for example, the point transformation example given in the Summary of the Invention section above. The above structure may be used to perform the operations necessary to carry out the transformation as show in TABLE I below.

TABLE I

Cycle	From SRF		ALU Cluster	ALU Cluster	ALU Cluster	ALU Cluster
	14	To SRF 14	18a	18b	18c	18d
1	x					
2	y		$x_1 = a_{11}x$	$x_2 = a_{12}x$	$x_3 = a_{13}x$	$x_4 = a_{14}x$
3	z		$y_1 = a_{21}y$	$y_2 = a_{22}y$	$y_3 = a_{23}y$	$y_4 = a_{24}y$
4			$z_1 = a_{31}z$	$z_2 = a_{32}z$	$z_3 = a_{33}z$	$z_4 = a_{34}z$
5			$t_1 = x_1 + y_1$	$t_2 = x_2 + y_2$	$t_3 = x_3 + y_3$	$t_4 = x_4 + y_4$
6			$u_1 = z_1 + a_{41}$	$u_2 = z_2 + a_{42}$	$u_3 = z_3 + a_{43}$	$u_4 = z_4 + a_{44}$
7			$x_p = t_1 + u_1$	$y_p = t_2 + u_2$	$z_p = t_3 + u_3$	$w = t_4 + u_4$
8						$w_1 = 1/w$
9			$x' = x_p * w_1$	$y' = y_p * w_1$	$z' = z_p * w_1$	
10		x'				
11		y'				
12		z'				

of an instruction descriptor which identifies a starting location, the stream to be loaded into the stream register file 14 or stored in the stream memory 16, and an address descriptor which specifies the record size, base address in memory and addressing mode, e.g., constant stride, indexed or bit-reversed. Optionally, the length of a stream in the stream register file 14 may be included.

In the first operation cycle, the x-coordinate of the point is loaded from the register file 14. In the next operation cycle, the y-coordinate is loaded and the x-coordinate is multiplied by appropriate elements in the transformation matrix. Similarly, in the following operation cycle, the z-coordinate is loaded and the y-coordinate is multiplied by the appropriate matrix elements, and so on. During the

WEST

7

computations, the program parameters such as the transformation matrix entries and intermediate results are stored in the local register files associated with the functional elements 26 which will consume them. Also, various values are distributed over the crossbar switch 30. For example, at the end of cycle 8 w , the reciprocal of w , is distributed to three of the arithmetic clusters 18 to be used in calculating x' , y' and z' .

In this way, four arithmetic clusters 18 can calculate the point transformation in just twelve operational cycles, a great improvement over conventional architectures. In practice, further optimizations would be performed to eliminate blank spots in the table at the beginning and end of the sequence using, e.g., loop unrolling or software pipelining. Also, in an actual implementation the functional elements 26 will have latencies of several cycles, e.g., two cycles for the adders 26a-26c, four cycles for the multipliers 26d and 26e, and eight cycles for the divider 26f, and the operation schedule would need to be rolled out to account for arithmetic latency. The resulting spaces can also be filled using unrolling.

Consider, as another example, triangle rendering—a common procedure in graphics processing which is exemplified by the C++ code below and whose dataflow is shown in FIG. 5:

```
void render_triangle_stream() {
    // Make sure kernels loaded into coprocessor controller
    int transform = load_microcode("transform.uc");
    int shade = load_microcode("shade.uc");
    int proj_cull = load_microcode("proj_cull.uc");
    int span_setup = load_microcode("span_setup.uc");
    int process_span = load_microcode("process_span.uc");
    int sort = load_microcode("sort.uc");
    int comp = load_microcode("comp.uc");
    int z_composite = load_microcode("z_composite.uc");
    // Triangle rendering on series of triangle streams
    for (int ii = 0; ii < NUM_TRIANGLE_STREAMS; ii++) {
        stream_load(mem_model_tri, srf_model_tri);
        stream_op(transform, srf_model_tri, srf_world_tri);
        stream_op(shade, srf_world_tri, srf_shaded_tri);
        stream_op(proj_cull, srf_shaded_tri, srf_screen_tri);
        stream_op(span_setup, srf_screen_tri, srf_spans);
        stream_op(process_span, srf_spans, srf_fragments);
        stream_op(sort, srf_fragments, srf_sort_idx);
        stream_op(comp, srf_sort_idx, srf_buf_idx, srf_pix);
        stream_load(mem_buf_pix{srf_buf_idx}, srf_pix2);
        stream_op(z_comp, srf_pix, srf_pix2, srf_out_pix);
        stream_store(srf_out_pix, mem_buf_pix{srf_buf_idx});
        update_descriptors();
    }
}
```

Here, each library function has a one-to-one correspondence with an application-level instruction. The load_microcode function loads the microcode routine denoted by its argument and returns the starting address of the code. Memory load and store instructions are respectively issued to the coprocessor by the stream_load and stream_store functions. Finally, an Operate instruction is issued by the stream_op function to cause the corresponding microcode kernel to run on each element of the specified source streams. For example, the first stream_op function shown in the code initiates a compound stream operation on the coprocessor by issuing an Operate instruction specifying the start address of the transform microcode. The instruction also specifies one input stream, srf_model_tri, and one output stream, srf_world_tri.

The arguments of the stream load, store and operate instructions are specified by stream descriptors. Each

8

memory stream descriptor, e.g., mem_model_tri, includes a base address, length, record length, mode and stride or index stream. Each register stream descriptor, e.g., srf_model_tri, includes a base location in the stream register file 16, record length, and stream length. These descriptors are produced by C++ code running on the host processor.

As shown in FIG. 5, the first arithmetic step in the process is to transform the triangle from model space to world space—a slightly more complicated version of the simple transform described in the summary section above. For this transformation, there is a single input stream and a single output stream. Each stream consists of twenty-four elements—for each of the three triangle vertices, the three dimensional vertex coordinates; a perspective coordinate; the vertex color, and a normal vector for the vertex expressed as a three dimensional coordinate. With this stream structure, the transformation computation can be expressed as the single compound stream operation shown in pseudocode below:

```
loop over all triangles {
    loop over three vertices {
        // read vertex data from input stream
        [x, y, z, w, color, nx, ny, nz] = input_stream();
        // compute transformed vertex coordinates
        tx = r11 * x + r12 * y + r13 * z + r14 * w;
        ty = r21 * x + r22 * y + r23 * z + r24 * w;
        tz = r31 * x + r32 * y + r33 * z + r34 * w;
        // compute transformed normal vector
        tnx = n11 * nx + n12 * ny + n13 * nz;
        tny = n21 * nx + n22 * ny + n23 * nz;
        tnz = n31 * nx + n32 * ny + n33 * nz;
        // write vertex data to output stream
        output_stream() = {tx, ty, tz, w, color, tnx, tny, tnz};
    }
}
```

Now, a typical data set might consist of average triangles covering twenty-five pixels with a depth complexity of 5. Rendering each triangle might require 1929 arithmetic operations, 666 references to stream register file 16 and 44 references to stream memory 18. With a conventional architecture in which three memory references are required for each arithmetic operation (one for reading the arithmetic instruction, one for reading the operands and one for writing the result), at least 5787 references would be necessary. Thus, by capturing locality within the kernels, coding the triangle rendering application to take advantage of the above-described architecture, references to memory outside the kernels are reduced by a factor of more than 8.

Moreover, once the kernels are programmed by microcode from the host processor 10, the entire triangle rendering process shown in FIG. 5 can be performed with only eleven application-level instructions: a Load instruction reads the triangle stream from the stream memory 16; seven Operate instructions sequence the kernels from transform to compact; a Load instruction uses the index vector computed by compact to read the old Z-values of the pixels in question; an Operate instruction performs Z-compositing; and a Store instruction writes the visible pixels and their Z-values back to the stream memory 16.

Additional efficiency could be realized by using more than one coprocessor in a multiprocessing arrangement. For example, when performing the triangle rendering process described above, one coprocessor could be used to run the first three kernels and transmit the result to a second coprocessor to run the remaining five kernels simply by inserting a Send and complementary Receive instruction at the appro-

appropriate position in the sequence of application-level instructions. The remaining resources of the two coprocessors may be used to render other triangles or to execute unrelated processes.

Kernels such as the transformation kernel listed above are written in a C-like microassembly language, and the kernel compiler (preferably on the host processor 10) takes this C-like code and generates VLIW microcode instructions that enable the microcontroller 20 to control the functional elements 26a-26h. The only flow control operations permitted in the kernels are iterative loops (although some control operations such as conditional branching may preferably be implemented in alternative ways as described in the U.S. patent application to William Dally, Scott Rixner, J. P. Grossman, and Chris Buehler, filed concurrently herewith and entitled SYSTEM AND METHOD FOR PERFORMING COMPOUND VECTOR OPERATIONS, incorporated herein by reference) and the compiler applies several common high-level optimizations such as loop unrolling, iterative copy propagation and dead code elimination. It then performs list scheduling starting with the largest, most deeply nested block, and within each block operations with the least slack are scheduled first.

The stream memory 16, stream register file 14 and local register files 28 have bandwidth ratios of 1:32:272. That is, for each word read from memory, thirty-two words may be accessed from the stream register file 14 and 272 words may be read from or written to the local register files 28 in the functional elements 26a-26h. In other words, the coprocessor can perform 40.5 arithmetic operations per four byte word of memory bandwidth and 1.2 arithmetic operations per word of stream register file bandwidth. The bandwidths of the stream memory 16 and stream register file 14 are limited by chip pin bandwidth and by available global chip wiring, respectively, while the bandwidth of the local register files 28 is set by the number of functional elements 26a-26h.

TABLE II compares the memory, global register and local register bandwidth requirements of the stream architecture of the coprocessor with a prior art vector processor and a prior art scalar processor for the above-described triangle transformation kernel. The figures for the scalar architecture were generated by compiling the transformation kernel for an UltraSPARC II using version 2.7.2 of the gcc compiler.

TABLE II

References	Stream	Scalar	Vector
Memory	5.5	342 (62.2)	48 (8.7)
Global Register File	48	1030 (21.5)	261 (5.4)
Local Register File	355	N/A	N/A

The entries for the scalar and vector processors should be self-explanatory. For the stream architecture, the 5.5 stream memory access figure was obtained by averaging the 44

memory references for the entire pipeline over eight kernels. The global register file reference figure is based on the 24 words read from the stream register file 14 and the 24 words written to the stream register file 14. Finally, the kernel executes 108 arithmetic operations which use 355 words of data from local register file 28. As can be seen from TABLE II, the memory bandwidth requirements of the scalar processor are 62.2 times higher than that of the stream architecture and the global register bandwidth requirements of the scalar processor are 21.5 times higher than that of the stream processor. The memory bandwidth requirements of the vector processor are 8.7 times that of the stream processor, and the global register bandwidth requirements of the vector processor are 5.4 times that of the stream processor.

Three image processing kernels, FFT, triangle transform and blockwarp (taken from an image-based rendering application), were used to generate the performance results shown in TABLE III below. FFT performs one stage of an N-point Fast Fourier Transform; triangle transform is the triangle vertex transformation described above; and Blockwarp performs a 3-D perspective transformation on 8x8 blocks of 3-D pixels to warp them from model space into screen space. As can be seen from the Table, the mean speed increase when moving from execution of each kernel on a single cluster to execution on eight clusters is over 7.5.

TABLE III

Kernel	Single Cluster	Eight Clusters	Speedup
FFT (cycles/butterfly)	4.19	0.75	5.59
Transform (cycles/triangle)	177	22.13	8
Blockwarp (cycles/block)	2890	275	10.5
Harmonic Mean			7.52

The vertex transformations are independent of one another, so there is no overhead lost to communication between clusters when executing that kernel, and the net speedup is exactly 8. The FFT requires exchanges of data between kernels, so the speedup when executing that kernel is somewhat less than 8. Execution of the Blockwarp kernel on eight clusters eliminates a loop in the process, resulting in a speedup of more than 8.

TABLE IIV shows the bandwidth used by each of the above kernels at each level of the memory hierarchy. The kernels require an average of 9.4 times as much local register bandwidth as stream register bandwidth. The throughput in the blockwarp kernel is worse than in the other kernels because it performs a divide when computing each pixel. The non-pipelined divider creates a bottleneck because all subsequent calculations are dependent on the divide result. Fully one-third of the execution cycles are spent waiting for results from the divider without issuing any arithmetic operations, even with loop unrolling to hide the latency to dependent calculations.

TABLE IV

Kernel	Stream Register File (GB/s)	Local Register File (GB/s)	Operations per Cycle	Arithmetic Op's (GOPS)
FFT	21.45	165.66	18.76	7.51
Transform	10.41	77.02	14.64	5.86
Blockwarp	4.19	46.59	8.73	3.49
Harmonic Mean	7.87	74.10	12.70	5.08

11

Thus, a processing system according to the present invention exposes the parallelism and locality of data processing tasks such as image processing and the like in a manner that is well-suited to current technologies. A programmer may describe an application as streams of records passed through computation kernels, and individual stream elements may be operated on in parallel by the arithmetic units acting under the control of the microcontroller as computational means to exploit data parallelism. Instruction parallelism may be exploited within the individual computation kernels by the microcontroller acting as program executing means. Finally, control parallelism may be exploited by partitioning an application across multiple processing systems by the host processor acting as control means. Locality is exposed both by recirculating streams through a stream register file and also within the computation kernels which access streams in order and keep a small set of local variables. Moreover, the combined effect of exploiting parallelism on each level is multiplicative. This enables the system architecture to make efficient use of a large number of arithmetic units without global bandwidth becoming a bottleneck.

As will be apparent from reading the above explanation, exploiting parallelism as used above and in the appended claims means performing computations, program execution or process control to take advantage of redundancy of content and similarity of structure in data, programs or processes flow to realize operational efficiencies in comparison with conventional architectures.

Modifications and variations of the preferred embodiment will be readily apparent to those skilled in the art. For example, the number of operative units such as arithmetic clusters, functional units within the clusters, memory banks and the like need not be as set forth herein and may readily be adapted depending on a particular application. Further, variations on the instruction set described above as well as new processor instructions may be provided. A larger number of simplified clusters may be provided, or a smaller number of more powerful clusters may be used. Such variations are within the scope of the present invention as defined by the appended claims.

What is claimed:

1. A data processing system comprising:
 - a controller;
 - at least one arithmetic cluster capable of independently and sequentially performing compound arithmetic operations, responsive to commands directly operatively provided from the controller, on data presented at an input thereof and providing resultant processed data at an output thereof, and capable of utilizing intermediate data generated as a result of performing the operations in subsequent operations without retrieving the intermediate data from a source external to that arithmetic cluster; and
 - a stream register file directly operatively coupled to the cluster and being selectively readable and writable, responsive to commands from the controller, by each of the at least one arithmetic cluster for holding the resultant processed data of the at least one arithmetic cluster.
2. The system of claim 1, wherein at least one arithmetic cluster includes a plurality of functional elements each capable of performing an individual arithmetic operation independently of other functional elements, and capable of providing results thereof to at least one of itself and other functional elements for use in subsequent arithmetic operations.
3. The system of claim 2, wherein the plurality of functional elements are connected to a crossbar switch for

12

providing results of arithmetic operations performed by each functional element to other functional elements.

4. The system of claim 3, wherein an arithmetic cluster includes a local storage unit for storing data to be used by a functional element within the arithmetic cluster during a compound vector operation.

5. The system of claim 4, wherein:

the local storage unit is connected to an input of the functional element within the arithmetic cluster; and data stored in the local storage unit is directly accessible only by the functional element to which it is connected.

6. The system of claim 4, wherein data stored in the local storage unit is accessible by a plurality of functional elements in the arithmetic cluster containing that local storage unit and plurality of functional elements.

7. The system of claim 3, wherein the crossbar switch is a sparse crossbar switch.

8. The system of claim 2, wherein the plurality of functional elements includes a scratchpad register file.

9. The system of claim 2, wherein the plurality of functional elements includes an intercluster communication unit for communicating with other arithmetic clusters.

10. The system of claim 1, wherein an arithmetic cluster includes a local storage unit for storing data to be used by the arithmetic cluster in subsequent arithmetic operations.

11. The system of claim 1, further comprising a host processor capable of selectively reading and writing the stream register file.

12. The system of claim 11, further comprising:

a network interface connected to the stream register file for exchanging data between the stream register file and another system.

13. The system of claim 1, wherein the at least one arithmetic cluster is a plurality of arithmetic clusters each capable of independently and sequentially performing compound arithmetic operations, responsive to commands from the controller, on data presented at respective inputs thereof and providing resultant processed data at respective outputs thereof, and capable of utilizing intermediate data generated as a result of performing the operations in subsequent operations without retrieving the intermediate data from a source external to that arithmetic cluster.

14. The system of claim 1, further comprising a global storage unit being selectively readable and writable, responsive to commands from the controller, only by the stream register file.

15. The system of claim 14, wherein the stream register file is selectively and independently writable, responsive to the controller, by at least two of the controller, the global storage unit and an arithmetic cluster.

16. The system of claim 14, wherein the global storage unit is selectively readable and writable, responsive to the controller, by the stream register file in independent, simultaneous transfers.

17. A method of processing data comprising:

performing multiple arithmetic operations simultaneously and independently in each of a plurality of arithmetic clusters responsive to commands directly operatively provided from a controller, at least some of the arithmetic operations utilizing data generated and supplied by the arithmetic clusters without retrieving the generated data from a source external to the arithmetic clusters; and

reading data used by the arithmetic clusters from and writing data generated by the arithmetic clusters to a stream register file connected directly to the plurality of arithmetic clusters.

13

18. The method of claim 17, wherein the reading and writing are performed for data generated by multiple arithmetic clusters in the plurality of arithmetic clusters independently and simultaneously.

19. The method of claim 17, wherein performing multiple arithmetic operations includes utilizing data generated and supplied by the arithmetic clusters without retrieving the generated data from a source external to an arithmetic clusters utilizing that data.

20. The method of claim 17, wherein performing multiple arithmetic operations includes performing individual arithmetic operations simultaneously and independently in each of a plurality of functional elements, at least some of the functional elements utilizing data generated and supplied by the functional elements without retrieving the data generated by the functional elements from a source external to an arithmetic cluster containing those functional elements.

21. The method of claim 17, further comprising storing at least some data generated by a functional element in a local storage unit.

22. The method of claim 21, further comprising retrieving data stored in the local storage unit only by a functional element which stored that data.

14

23. The method of claim 21, further comprising retrieving data stored in the local storage unit by plural functional units within an arithmetic cluster containing the plural functional elements.

24. The method of claim 17, further comprising exchanging data between arithmetic clusters.

25. The method of claim 17, further comprising exchanging data from the stream register file to an external system.

26. The method of claim 17, further comprising exchanging data between the stream register file and a global storage unit.

27. The method of claim 26, wherein exchanging data includes exchanging multiple data elements between the stream register file and the global storage unit independently and simultaneously.

28. The system of claim 1, wherein cluster instructions and at least one of data input and output streams are provided to the at least one cluster responsive to a stream instruction.

29. The system of claim 8, wherein the scratchpad register file is independently addressable for the cluster which it is in using a computed address.

* * * * *



US005909544A

United States Patent [19] Anderson, II et al.

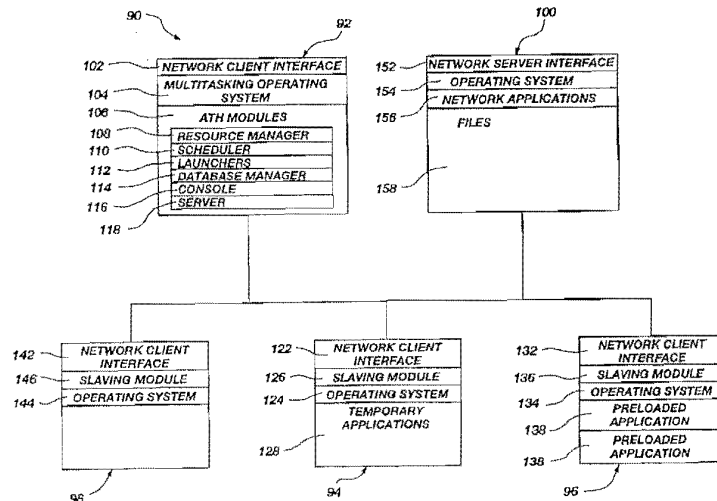
[11] Patent Number: **5,909,544**
[45] Date of Patent: ***Jun. 1, 1999**

- [54] AUTOMATED TEST HARNESS
- | | | | |
|-----------|---------|----------------|------------|
| 5,313,616 | 5/1994 | Cline et al. | 395/500 |
| 5,315,711 | 5/1994 | Barone et al. | 395/200.38 |
| 5,335,342 | 8/1994 | Pope et al. | 395/575 |
| 5,371,883 | 12/1994 | Gross et al. | 395/575 |
| 5,388,211 | 2/1995 | Hornbuckle | 395/712 |
| 5,630,049 | 5/1997 | Cardoza et al. | 395/183.01 |
| 5,671,414 | 9/1997 | Nicolet | 395/684 |
| 5,684,952 | 11/1997 | Stein | 395/712 |
| 5,805,897 | 9/1998 | Glowny | 395/712 |
- [75] Inventors: **Michell M. Anderson, II**, Orem;
Howard K. Bangert, Highland;
Marlon T. Borup, Orem; **James E. Byer**, American Fork; **Darin L. Cable**, Spanish Fork; **Ross W. Doxey**, Orem; **Richard S. Graham**, Springville; **Todd D. Hale**, American Fork; **Britt J. Hawley**; **Richard W. Lamplugh**, both of Orem; **Rick L. Pray**, Woodland Hills, all of Utah
- [73] Assignee: **Novell Inc.**, Provo, Utah
- [*] Notice: This patent issued on a continued prosecution application filed under 37 CFR 1.53(d), and is subject to the twenty year patent term provisions of 35 U.S.C. 154(a)(2).
- [21] Appl. No.: **08/518,160**
[22] Filed: **Aug. 23, 1995**
[51] Int. Cl.⁶ **G06F 13/00**
[52] U.S. Cl. **395/200.38; 395/200.47; 395/200.5; 395/712**
[58] Field of Search **395/183.03, 183.14, 395/712, 200.47, 200.49, 200.5, 200.51, 200.52, 200.38, 200.39, 200.41**
- [56] **References Cited**
U.S. PATENT DOCUMENTS
5,021,997 6/1991 Archie et al. 395/183.07
5,157,782 10/1992 Tuttle et al. 395/575
5,218,605 6/1993 Low et al. 371/16
- Primary Examiner*—Meng-Al T. An
Assistant Examiner—Walter D. Davis, Jr.
Attorney, Agent, or Firm—Madson & Metcalf

[57] **ABSTRACT**

An apparatus and method for temporarily slaving and configuring a plurality of hardware resources such as computers, microprocessor-based devices, and the like, over a network, and then emancipating the resources to operate independently. Resources or targets may be enslaved at an operating system level. A controller may configure a plurality of hardware resources such as computers, microprocessor-based devices, and the like, to operate autonomously over a network. Resources or targets may be enslaved at an operating system level, configured with commands from a controller, and emancipated to operate independently. Emancipated resources may download applications, read and write files, communicate with other devices, and otherwise operate as independent computers. Data corresponding to test instructions may be downloaded from, and data corresponding to results may be recorded and saved on, a network server by a resource operating independently. Upon completion of a test or a series of tests, a resource may again report back to a controller, be enslaved and reconfigured, and be emancipated to operate other test software.

27 Claims, 6 Drawing Sheets



WEST

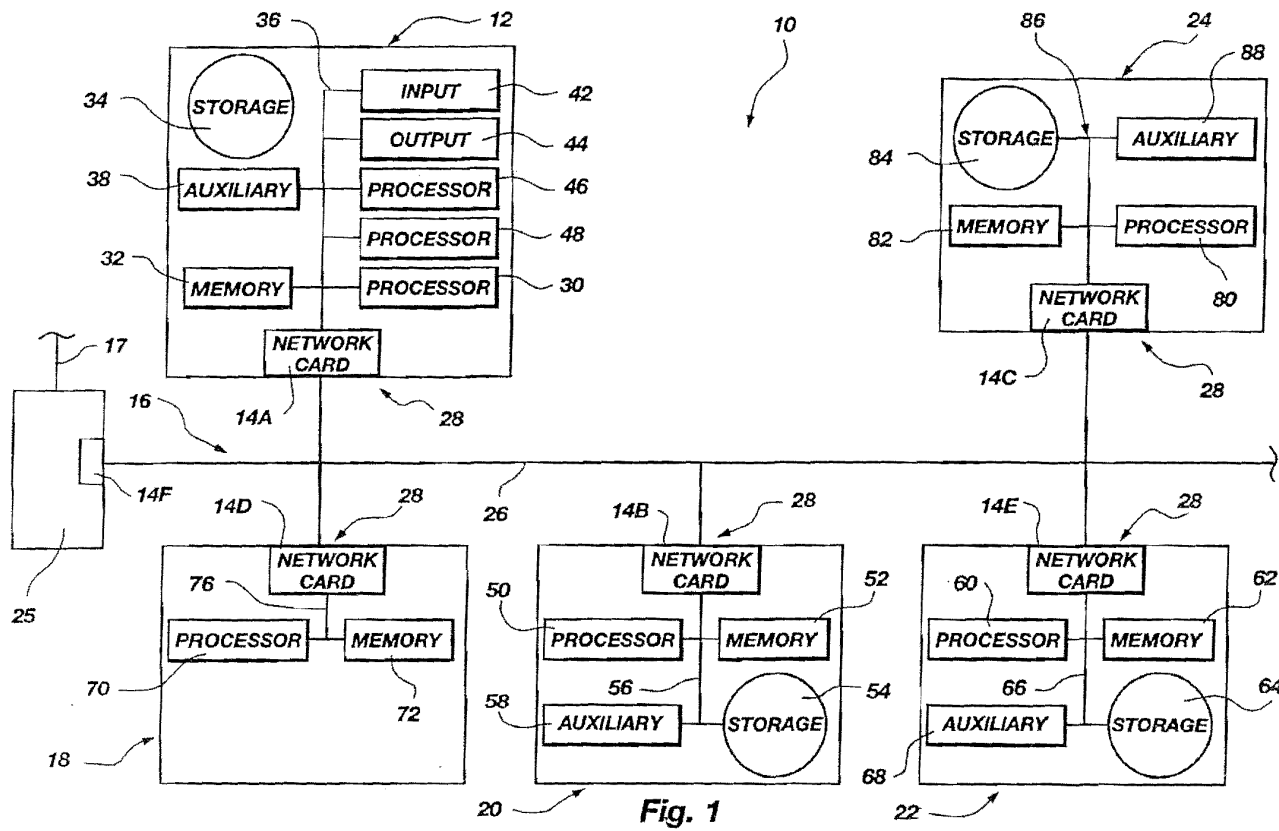


Fig. 1

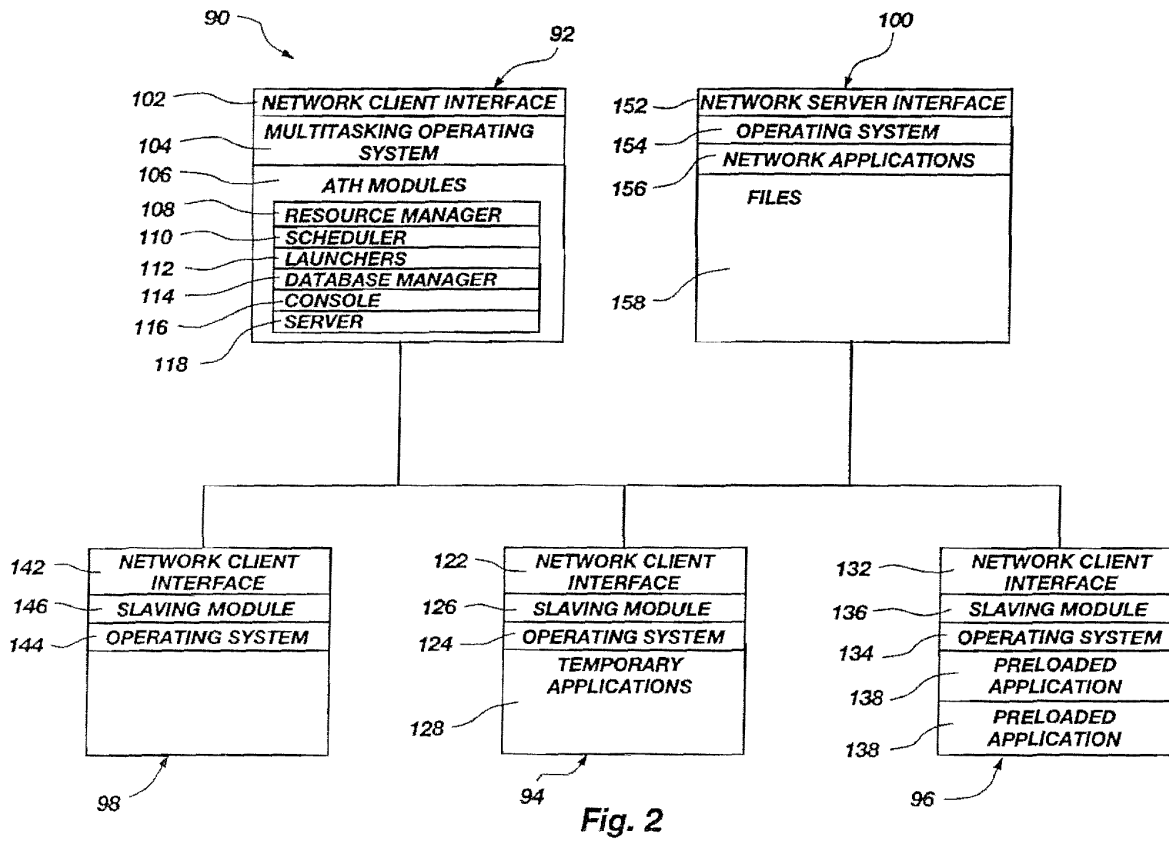
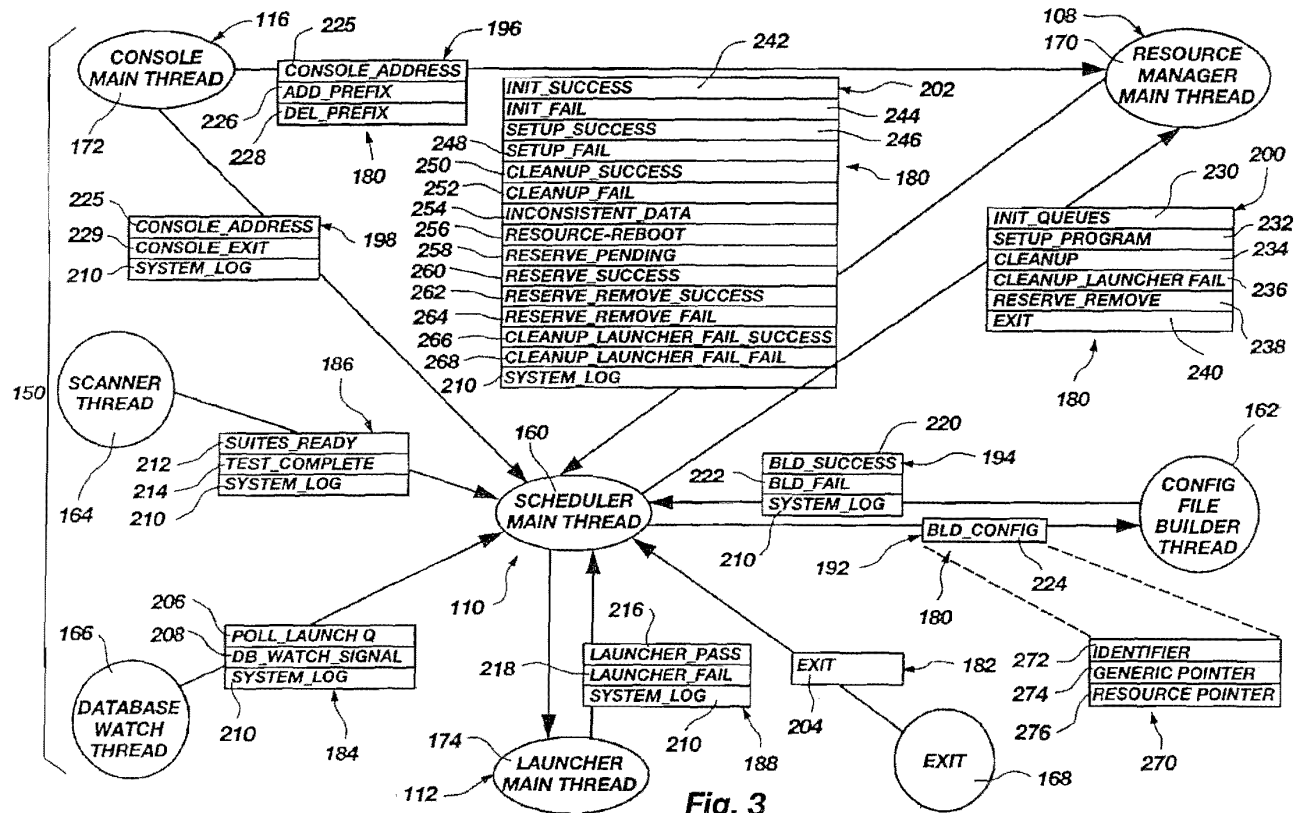


Fig. 2



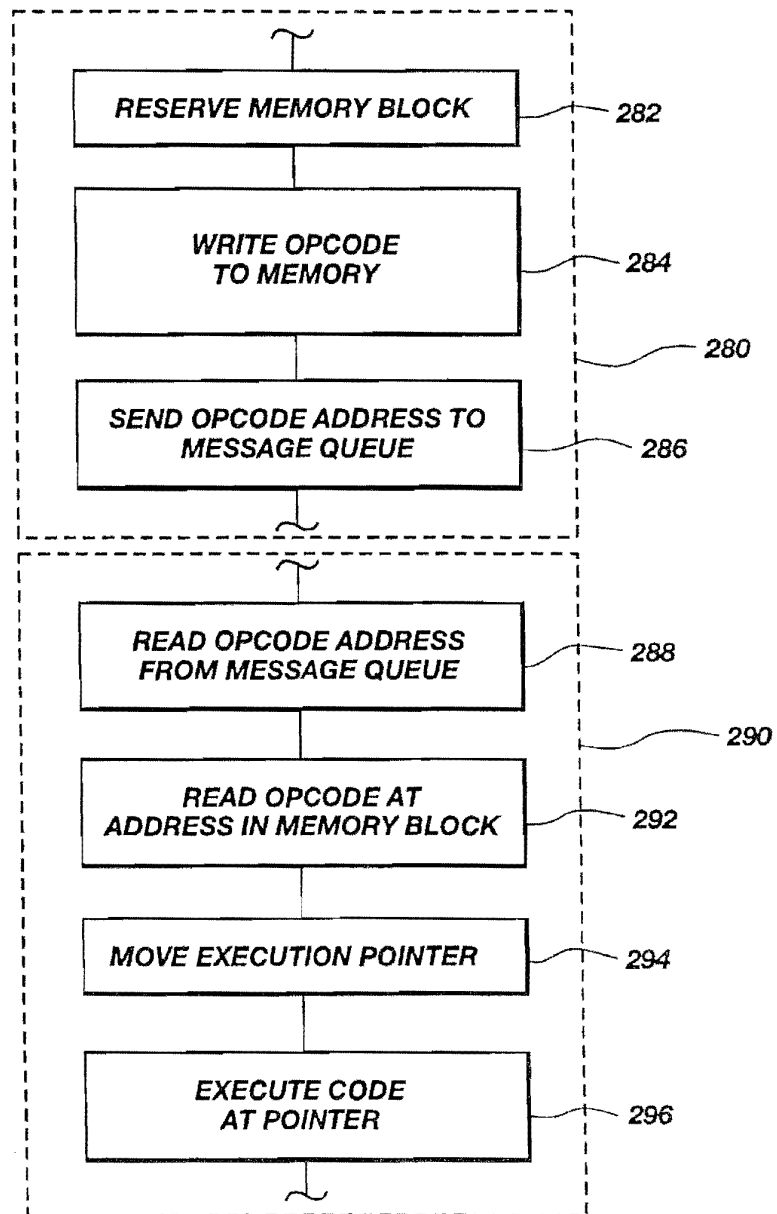


Fig. 4

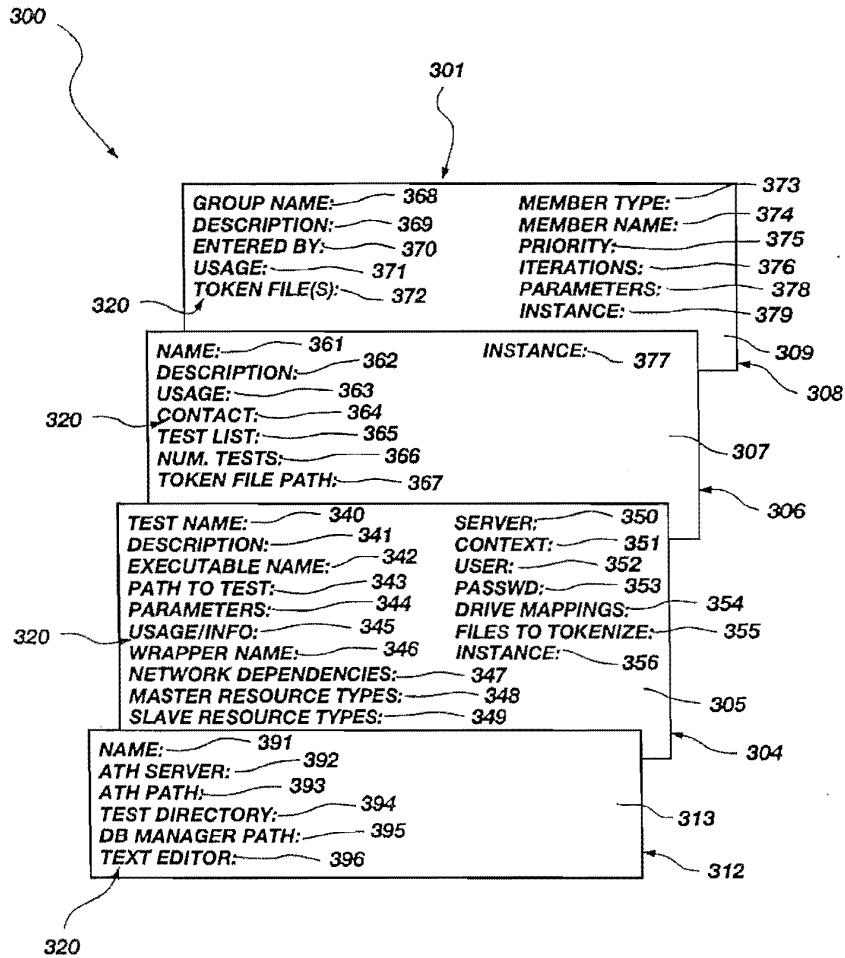


Fig. 5

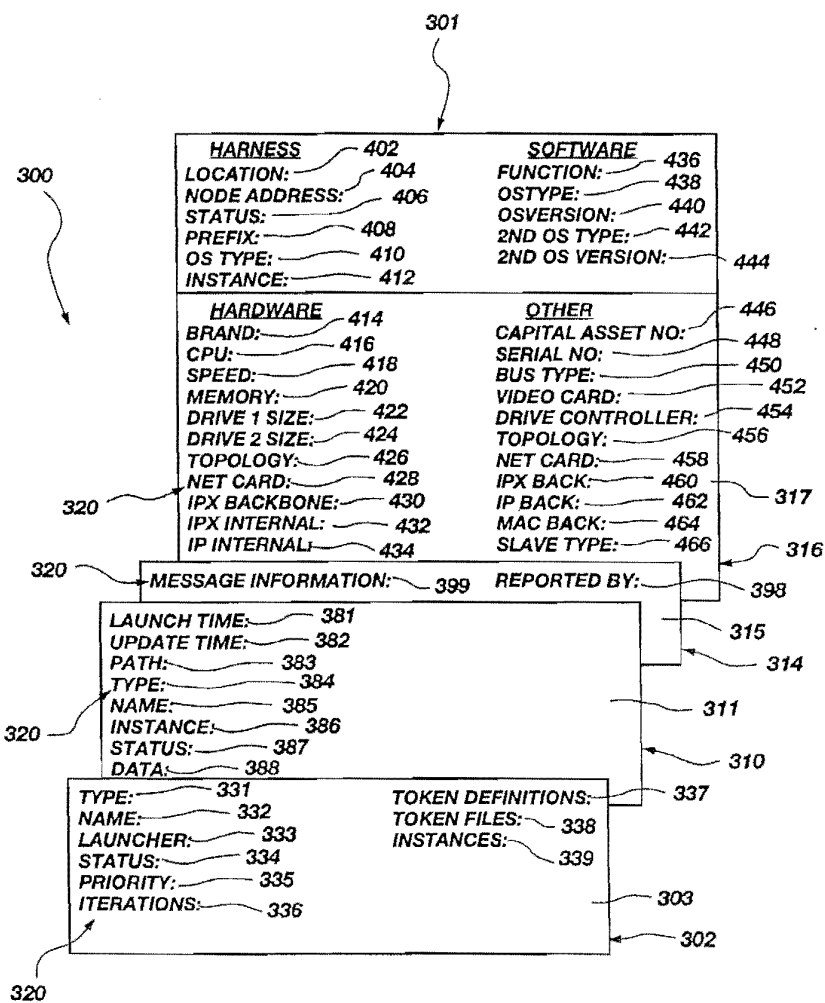


Fig. 6

AUTOMATED TEST HARNESS

BACKGROUND

1. The Field of the Invention

This invention relates to computer networks and, more particularly, to novel systems and methods for temporarily slaving operating systems of a plurality of processors for configuration purposes or for downloading software, and then later emancipating these slaved devices (e.g. computers, microprocessor-based devices, instruments, etc.) to operate independently, including running software, downloading files, and uploading files over the network.

2. The Background Art

Networking typically involves several microprocessor-based devices (nodes) exchanging data with one another over a transmission medium, a communication link such as, for example, a hard wire, fiberoptic cable, radio frequency transmitter/receiver, or other physical communication mechanism. A link between a node and a network or between routers of a wide area network (WAN) may be wireless. Network cards provide a mechanical and data connection between the communication link (e.g. fiberoptic, wire, or wireless) and the processor of the node or device to be connected to the network. Communication may occur by various protocols (rules), with steps (activities) executed by devices satisfying those protocols.

Networks may be defined at several levels of grouping and communication. For convenience, the terms primary, secondary and tertiary network are used here to indicate the extent and complexity of a network. A primary network is the most fundamental connecting of two nodes in some manner. A local area network (LAN) may be thought of as a primary network. A secondary network is a routed network, one including at least two primary networks connected by a router for forwarding messages between the primary networks. A tertiary network (sometimes called an internetwork) is one including two primary or secondary networks, a router in one being separated by an intermediate network from a router in the other. Thus, as the name network implies, a tertiary network can extend from router to router via intervening primary networks virtually forever, within the constraints of the laws of electrophysics and communications protocols.

A server is a computer connected to a network via a network card and programmed to act as a traffic manager and storage device for files of data being transmitted over the network by the various connected nodes. A hard wire interconnected to a group of network cards with attached computers, with one of those computers acting as a server, is a typical network. In other networks, every device may be a server.

Computer networks are capable of facilitating many functions, but not certain other functions. For example, as computer network technology has grown, so has the need to pass information over the networks.

Likewise, a user at each node, typically, must configure the node, from physically connecting hardware and loading an operating system of the processor to loading an application and instructing the application how to store and retrieve data representing information. A high degree of automation of tasks normally done by users is not typically available between devices (nodes) across networks. One command line at a time is typically required to be sent and responded to.

Unfortunately, a user logging a controller computer on to a network cannot automatically download executables to a

remote, general-purpose, computer for execution. The remote computer must be logged on and configured by the user just as any other computer.

Similarly, a slaved computer may be logged on to a network, configured, loaded with an application and then enslaved to communicate with a master to receive information from a master for use by an application previously launched on the slave computer. Instructions and responses are typically on a command-by-command basis.

Configuration may be thought of as programming a slave with executable codes, and providing information for establishing connections and protocols, identifying other devices to be addressed, identifying a server and possibly a router, providing the address of the slave to the network server, and so forth. Thus, once configured, a node (e.g. computer or other processor-based device on a network) may send and receive messages. Also, once an application has been launched (loaded and instructed to run), the slave may send and receive information related to the application. However, if a slave is basically a general purpose processor with an operating system and memory, or optionally including a storage device, it must be manually loaded with software and instructed to run by a user before communicating with a master.

What is needed is a system that enables a controller to act as a user or otherwise take control of a processor to instruct it. For example, a processor needs to be controlled long enough to be instructed to download "executables," (files containing a coded instruction capable of being "understood" by an operating system of a processor to result in an operable instruction executable by the processor), and to launch (run) the executables. Slaving an operating system and configuring the host (remote computer) need to be followed by a high degree of independent operation by the remote computer.

A scheme is needed for slaving and configuring, followed by downloading entire executables to a general purpose computer, that is, a target, located remotely from a controller, particularly over a network. For example, a controller may need to automatically perform all tasks normally required of a user seated at a keyboard associated with the target. For example, a user boots up a computer with some operating system, logs on to a network, navigates through a hierarchy of directories, locates and loads an executable file, and runs the "executable." For systems having an input/output (I/O) interface for dealing with humans, great utility may be obtained by providing for a remote computer to replace the user, thus providing automation of many manual tasks associated with setting up a computer, running software, and managing files.

Similarly, for processor-based apparatus having I/O interfaces embedded, without user interfaces as monitors and keyboards, great utility could be obtained by providing for automatic access to the operating system by a remote computer across a network. For example, automatic loading of executables from a remote computer, automatic upgrading, and any similar task accessing the operating system of a microprocessor-based device on a network could make management and upgrading of such devices tractable on a large scale.

Likewise, the need may be for downloading information to the memory of a device by which the device may operate a resident executable. For example, when a single computer connects to several peripheral devices, the single computer typically may control those various devices directly. Also, a controlling computer may simply provide data correspond-

ing to responses, commands, or information needed by an application running on a slaved computer. However a need exists for autonomous devices to be temporarily accessed remotely with data for controlling or supporting resident executables, after which the devices return to their autonomous operating condition. The need, when compared with the conventional term "slaved computer" is for an "emancipated computer" that may be temporarily and remotely enslaved at an operating system level, configured, and emancipated to operate independently over the network. The emancipated computer or other device needs to be able to access files of input information or load itself with an executable, or launch an executable, after which it may report back to a master to be enslaved again, and reconfigured to operate again as an emancipated device.

A system is needed that is transparent to a user for tracking large numbers of such emancipated slave computers. Thus, a system is needed that will facilitate a controller accessing numerous resources (computers), monitoring their capabilities, taking temporary control of any available one, commanding the selected resource to configure itself, load and run software, and pass input and output information properly, and report to the controller when it is again available to be assigned another project. Again, single commands with reporting back for another command are not the intention. What is needed is true emancipation, maximizing the use of the emancipated slave resource, with minimum necessity for control by the controller, while at the same time permitting the controller to communicate with the resource at an operating system level when needed. Thus a general purpose computer may be commanded to completely re-configure itself as needed without human intervention.

Thus maximum utility of all resources may be achieved by simply programming the logical options, defaults, backups for failed options, and the like, in order to keep all resources operating all the time with tasks that each can complete. Thus, regardless of processor, speed, memory, architecture or operating system, each resource needs to be able to be put to work effectively for every minute that it is available, without waiting for a user to monitor, schedule, and program the resource manually. All this capability over network can make possible testing systems having great speed, throughput, and flexibility from an assembly of any number of available resources of virtually any type. Moreover, human intervention may be only required at some minimal degree, yet a human operator could access the controller for information on the system status, results, and the like at any time.

BRIEF SUMMARY AND OBJECTS OF THE INVENTION

In view of the foregoing, it is a primary object of the present invention to provide a system for commanding resource computers to download test software and parametric data from a computer server (central repository) to the resource (test) computer or device selected from a plurality of resource computers over a primary, secondary, or tertiary network.

It is an object of the invention to provide a system for uploading and storing test result data from a test computer to a central repository on the network.

It is an object of the invention to automatically, according to a method transparent to a user, without requiring intervention by a user, manage a plurality of resource computers, over a network from a controlling computer, the startup and

configuration of one or more test computers selected from the plurality of resource computers, transfer software files and parametric data files to the test computer or cause the test computer to download those files, provide surrogate commands to the test computers as if from a user or programming computer, and emancipate the test computer to operate the software of the software files and return result data corresponding to test data to a system database repository remote from the test computer.

It is an object of the invention to permit transfer of data by bit streams, files, fields, or entire databases.

It is an object of the invention to transfer data to a controller, or to a remote processor operating as a database manager, or directly to a database in a storage device, over a network, without requiring synchronous communication as in conventional master/slave systems, for example, without requiring that a master send an instruction to obtain each packet of information from a slave, or without requiring a master to do anything to enable a resource to complete a function and store data for access later by other computers, such as the controller.

It is an object of the invention to provide a system for tracking and scheduling of available resource computers connected in a network, including monitoring such parameters as, for example, the location, name, operating system, memory, speed, processor characteristics, memory capacity and other operational characteristics, of each resource computer, and using that information to allocate those resource computers to run applications, such as for example, test applications and collect data, such as test data.

It is an object of the invention to provide over a network a server (whether having a separate central processor, or time-sharing with the controller) for storing and retrieving files to be used temporarily by a controlling computer on the network and files to be loaded to and from a selected test (resource) computer of a plurality of resource computers connected to the network.

It is an object of the invention to provide database management, including an information format and designated fields to be stored in a database, for application, such as test applications, datafiles of parameters for controlling the test applications, and datafiles of test results obtained by running the test applications, all to be uploadable and downloadable over a network between a plurality of resource computers, a server, and a controller connected in a network.

It is an object of the invention to provide scheduling and queuing for a plurality of resource computers connected in a network, the resource computers being general purpose, program inable, digital computers, wherein application files of testing software, and parametric datafiles for operating the testing software are not required to be resident, but may be made available or downloaded by each selected, available, resource computer as prompted by a controlling computer connected to the network to communicate with the individual operating systems of the resource computers.

It is an object of the invention to provide resource management of a plurality of resource computers connected in a network according to the performance parameters of each resource, such as, for example, its memory, speed, processor characteristics, memory capacity and other operational characteristics.

It is an object of the invention to provide selection of a resource computer by a controller, loading and launching of applications by the resource computer under limited prompting or direction from the controller.

It is an object of the invention to provide a scheduler for selecting a test, having an application, from a queue of tests, selecting criteria for choosing a resource computer, and passing to a resource manager such criteria, and to provide a resource manager to track performance characteristics of a plurality of resource computers connected to a network, and to select a suitable resource computer for running the application based on the criteria, for example, by tracking each resource, and matching each resource at any given time with a selected application to run on that resource, the application being selected from a plurality of applications identified by the scheduler from the launch queue.

It is an object of the invention to provide a method for a resource manager to catalog and track a plurality of resource computers, for a launcher to configure a selected resource of the plurality of resource computers, for emancipating the selected resource to run alone, for the selected resource to load or have loaded on it an application, for receiving therefrom (such as by a controller or a server) data corresponding to test results generated by a test application in a suitable format for storing and retrieval.

It is an object of the invention to provide a method for configuring a plurality of resource computers remotely over a network by a controlling computer to load and run application software and create data corresponding to results generated by the application software for storage in a storage device, such as, for example, by transfer of a datafile from the application software to a server, or by transfer of data from the processor to the controller for logging into a database on a server, which may have a database manager operably connected thereto for managing datafiles.

It is an object of the invention to communicate over a network with the operating system of a general purpose computer, temporarily control the general purpose computer to configure it for running application software selected and downloaded by a controlling computer, and then emancipate the general purpose computer to operate substantially independently.

It is an object of the invention to provide for monitoring by a controller the availability and abilities of a plurality of resource computers, and for communication by the controller with the operating systems of individual resource computers over a network for configuring the resource computers, providing for loading and launching of application software thereon.

Consistent with the foregoing objects, and in accordance with the invention as embodied and broadly described herein, a test harness may include a controller, a slave or target, and a network interconnecting the master and one or more slaves. A test harness may also be thought of as a system of related software modules for controlling and managing the resources of a system of hardware components (such as computers) to render one processor a controller and another processor a slave at an operating system level.

A resource or target may be a computer controlled across a network as a slave during a setup procedure controlled by commands from the controller. The slave is thereafter emancipated to operate independently, capable of downloading and running software, reading and writing files, and the like, without direction or control from the controller. A computer may include a processor capable of hosting an operating system, with memory for temporary storage of data and instructions used by the processor during operation.

An operating environment, or simply environment, may be an operating system, such as, for example DOS, OS/2™, Windows™, and so forth. However, an environment may be

another executable program operating within or under the operating environment hosted by a processor. By a "different" operating system is meant not a copy of an operating system running on a different processor, but an operating system using a programming structure, command system, protocol, combination of the foregoing, or the like, different from that of another operating system. For example DOS by Microsoft™ and Windows by Microsoft™, OS/2™, Macintosh™, Next Step™, are all separate, distinct, "different," operating systems for communicating with and controlling a processor, its associated memory, and possibly other devices such as a storage device over a bus of a computer.

A method and apparatus are disclosed in one embodiment of the present invention as including a network having a controller and a plurality of resources. An apparatus made in accordance with one embodiment of the invention may include a server, a database manager, a scheduler, a resource manager, a launcher, and a plurality of resources, each resource containing a processor and memory for hosting an application.

A method practiced in accordance with an embodiment of the invention may include tracking a plurality of resources, scheduling a plurality of applications having executables for controlling a processor and for providing data corresponding to operations of the processor, selecting a resource to host an application selected from the plurality of applications, configuring remotely over a network the selected resource, launching the application on the resource, running by the resource the application, providing output data from the resource, recording in a database for later retrieval the output data, and managing the database for identification and selection of the output data.

In one embodiment of an apparatus made in accordance with the invention, an apparatus for running test software may include a network for communicating data. A target may be operably connected to the network, the target comprising a first network interface, a first processor and a first memory device. The first processor may be programmable to host an operating system to communicate instructions to the first processor and to communicate data to and from the memory device. The first network interface may be operably connected to the first processor to communicate data between the first processor and the network;

A controller may be operably connected to the network to communicate data with the target over the network. The controller may include a second network interface and may be operably connected to the network to communicate data between the controller and the network. Likewise the controller may include a second processor operably connected to the second network interface for controlling the first processor's operating system.

The controller may include a second memory device for storing data communicated to and from the second processor, and a storage device for storing files. The files may include test applications containing instructions executable by the first processor, control applications containing instructions executable by the second processor, test datafiles containing data corresponding to test parameters used by the first processor in running the test applications, and result datafiles containing data corresponding to results obtained by the first processor while running test applications.

The apparatus may include software modules hosted on the controller. Modules may include a resource manager operable on the second processor for accumulating, tracking,

and controlling storage of data corresponding to identification, performance characteristics, and availability of the target to run test applications. The resource manager may also service likewise a plurality of such targets.

The apparatus may include a server for storing and retrieving files, and a database manager for storing to and retrieving from a database records corresponding to data communicated between the first processor and the second processor. The apparatus may include a plurality of databases accessible by the database manager.

The controller may include a launcher in one or more instantiations for communicating with the operating system of the target. The launcher, or the target after configuration, may download selected applications and application (test) datafiles to the target. In general, another type of application may be substituted for a test application, an application datafile for a test datafile of inputs, and a result datafile for test result datafile. The controller may include a scheduler for acquiring data corresponding to a queue of tests to be run, selecting test applications to be loaded onto the target, and controlling matching of the test applications to the target.

The apparatus may include a plurality of targets, wherein each target of the plurality of targets may host an operating system different from the operating system of any or all other targets of the plurality of targets. The target (or each target of a plurality of targets) may include a storage device operably connected to the first processor for temporarily storing test datafiles, result datafiles, and test applications while running tests. The target may be programmed to host an operating system, and an environment operating under the operating system, (or on top of the operating system).

In general, when the term "test application" is used herein, the word "application" may be substituted. Although in one preferred embodiment, the apparatus and method may be used to automate the execution of software testing applications, the method and apparatus are equally valid for other software applications.

The apparatus may be arranged to have the second processor programmed with a plurality of software modules. For example, the plurality of software modules may include a server for storing and retrieving files, a database manager for storing and retrieving records from a plurality of databases corresponding to tests, suites of tests, groups of tests, suites, or other groups, target performance characteristics, test results, system parameters and errors, launchers and launches, status of targets, and the like communicated between the first processor and the second processor. Other modules may include a resource manager for tracking identification, characteristics, and availability of the target to run test applications, a launcher for communicating with the target and for initiating downloading, by the controller or the target, of selected test applications and test datafiles to the target, and a scheduler for acquiring data corresponding to a queue of tests to be run, selecting test applications to be loaded by the target, under direction of the launcher, and for spawning launchers as needed.

An apparatus may use processes and operational codes (opcodes) to provide a logical flow between processes, each process operating according to opcodes placed in a queue of the process by another process. Each process may be hosted on an individual device (computer, processor) alone, or several processes may be hosted on a single processor, timesharing the processor. This is one benefit of a multi-tasking operating system. An entire system (automated test harness) may be queue driven. Processes simply pass

opcodes to queues of other processes. Each process may simply keep checking certain queues to determine whether anything needs to be done. When a process finds information in a queue, the process executes itself based upon the opcode found in the queue.

In general, an opcode may be a small data structure that may be passed from one process to a queue to be picked up by another process. Alternatively, an opcode may similarly pass from one thread to another thread of the same process. An opcode may include a block of memory allocated to contain a name, a pointer to identify other data structures needed by the particular opcode. For example, an opcode may invoke certain steps within a process. Those steps may expect certain information to be provided. That information may be provided in the data structure to which a pointer is pointing, addressing. In general, a data structure may be configured in a variety of ways.

A process may operate on data contained in a data structure, and then pass to another process (by means of an opcode) a pointer indicating that data structure. Thereafter, the other process may access the same data structure operating on, using, adding to, or deleting the information, or the like. Thus, an opcode may pass a pointer and data between multiple processes or threads during the running of an application.

Pointers may be chained. That is, a series of pointers may identify a series of next data structures. A data structure identified by a first pointer may in turn contain information interpreted as pointers. These pointers in the data structure, then, may point at, and thus chain to other data structures.

In an opcode as implemented in one preferred embodiment of an automated test harness in accordance with the invention, a generic first pointer may exist without practical limitation as to the nature of the data structure to which it points. By contrast, a second pointer may introduce a certain efficiency by pointing at a single, specific type of data structure. For example, the second pointer, by its very location in an opcode, may be bound to specific data that identifies its function. This may save process steps, and create processes by chaining opcodes, each data structure containing a pointer to the following data structure.

A method practiced in accordance with the invention may include a method of running test software on a plurality of targets, also called resources, processors, or computers. One may think of resources as hardware resources having the capability to host an operating system. The operating system may be enslaved and then emancipated by a master or controller. Emancipated means that a formerly enslaved resource's operating system is again rendered autonomous to operate independently of the master or controller.

The method may include operably connecting a target or resource to a network, the target comprising a first network interface, a first processor and a first memory device, the processor being programmable to host an operating system to communicate instructions to the processor and to communicate data to and from the memory device. The target's first network interface may be operably connected to the processor to communicate data between the processor and the network.

The method may include operably connecting a controller to the network to communicate with the target over the network. The controller may include a second network interface operably connected to the network to communicate data between the controller and the network. A second processor may be operably connected to the second network interface for controlling the operating system of the target.

A second memory device may be operably connected to the second processor for storing data communicated to and from the second processor, while a storage device may be connected for storing files.

The method may include loading by the target, at the instance of a launcher hosted on the controller to be in communication with an operating system hosted on the target, certain files, including executable files downloaded by the target from a server. These files may include test applications containing instructions executable by the first processor, test datafiles containing data corresponding to test parameters used by the first processor, and the like. Similarly, control applications containing instructions executable by the second processor may be downloaded by the controller.

The method may include running the test applications on the target, creating datafiles containing data corresponding to results obtained by the first processor while running the test applications. The method may also include uploading the datafiles of results to the server.

Temporary slaving may be accomplished by loading a slave module onto each target to operate on the operating system of the target, and a master module to operate on the operating system of controller. The slave module feeds back to the master module all data necessary for the master module to interact with the target's operating system as a user. Thus prompts, screens, and the like generated by the operating system of the target are communicated by the slave module to the master module of the controller. The master module is programmed to act based on data provided by the slave, sending instructions to the slave and commands for forwarding to the operating system of the target (resource).

The slave module receives executables from the master module, along with commands to be sent by the slave module to the command line of the operating system. Thus, the slave module is effectively provided with commands destined for the operating system, and is itself instructed to send the commands to the operating system for execution.

The temporary slaving operation may be conducted by any suitable software modules operating on the controller and the target. For example, a slaving program such as NCONTROL™ is a Novell™ slaving program that has been found suitable for facilitating the temporary slaving operation.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects and features of the present invention will become more fully apparent from the following description and appended claims, taken in conjunction with the accompanying drawings. Understanding that these drawings depict only typical embodiments of the invention and are, therefore, not to be considered limiting of its scope, the invention will be described with additional specificity and detail through use of the accompanying drawings in which:

FIG. 1 is a schematic block diagram of one embodiment of an apparatus made in accordance with the invention;

FIG. 2 is a schematic block diagram of one embodiment of software modules hosted on the apparatus of FIG. 1;

FIG. 3 is a schematic block diagram of one embodiment of processes and threads run on a controller, and operational codes passed between processes and threads running on the apparatus of FIG. 1;

FIG. 4 is a schematic block diagram of one embodiment of a process by which the operational codes of FIG. 3 may operate; and

FIGS. 5 and 6 are schematic block diagrams illustrating one embodiment of records of databases accessed by the apparatus of FIG. 1.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

It will be readily understood that the components of the present invention, as generally described and illustrated in the Figures herein, could be arranged and designed in a wide variety of different configurations. Thus, the following more detailed description of the embodiments of the system and method of the present invention, as represented in FIGS. 1 through 6, is not intended to limit the scope of the invention, as claimed, but it is merely representative of the presently preferred embodiments of the invention.

The presently preferred embodiments of the invention will be best understood by reference to the drawings, wherein like parts are designated by like numerals throughout. Reference is first made to FIG. 1, which illustrates one preferred embodiment of a schematic diagram of hardware components, while FIG. 2 illustrates a block diagram of software modules operable on the apparatus from the block diagram of FIG. 1.

FIG. 3 illustrates the processes and threads of certain software modules of FIG. 2, and particularly illustrates certain opcodes passed between the threads for establishing the operating logic of the apparatus of FIG. 1. Since the apparatus and processes may be queue driven according to messages or opcodes sent and received as appropriate, logic may be complex, but a multitude of logical paths may be determined by reference to the FIGS. 1-6.

Each operational code or opcode may be used by a sending and receiving thread or process according to the logic of the flow chart of FIG. 3. Data for inputs to and outputs from the components of apparatus of FIG. 1 may be stored in any suitable data structure or memory device. For example, in one embodiment, data may be stored in databases configured as illustrated by the representative records and fields of the databases of FIGS. 5-6.

Those of ordinary skill in the art will, of course, appreciate that various modifications to the detailed schematic diagrams of FIGS. 1-6 may easily be made without departing from the essential characteristics of the invention, as described in connection with the block diagram of FIGS. 1-6. Thus, the following description of the details of the schematic diagrams of FIGS. 3-6 is intended only as an example, and it simply illustrates one presently preferred embodiment of a schematic diagram that is consistent with the foregoing description of FIGS. 1-2 and the invention as claimed herein. The operation of various components and modules corresponding to each of the functional blocks of FIGS. 1-2 are outlined in FIGS. 3-6 and are numbered with like numerals.

Referring to FIG. 1, an apparatus 10, alternately referred to as a system 10 or an automated test harness 10 may include a controller 12. The controller 12 may be connected to a network interface 14A, or the network interface 14A may be integral to the controller 12. Regardless, the controller 12 may be connected to a network 16. The controller 12 may host a slaving software module for slaving a number of resources 18, alternately referred to as targets 18, or target computers 18. The resources 18 may have alternate embodiments including the resource 20 and resource 22. That is, a resource 18 may include all of the components of a general purpose computer. Nevertheless, the resource 18 need only contain a processor and sufficient memory to host an oper-

11

ating system for temporary enslavement to the controller 12, followed by later emancipation for running independently.

The resources 18, 20, 22, and controller 12 may be connected to a server 24 by means of a carrier 26, typically a physical line 26 or cable 26, although the carrier 26 may be a wireless carrier 26.

The controller 12 and resources 18, together with the server 24 may be configured in a variety of topologies. For example, the network 16 may be arranged as a star, a ring, a mesh having more than one connection on a single node, or the like. Alternatively, the network 16 may be configured as a bus having a single line 26 to which all nodes 28 connect. In general, a node 28 may be used to designate any microprocessor based device connectable to the network 16.

Similarly, the nodes 28 may be configured to operate under any one of a multitude of available protocols. For example, popular protocols that might be employed may include an internetwork packet exchange (IPX) or transmission control protocol/internet protocol (TCP/IP), sometimes called transport control protocol/interface program, sequenced packet exchange (SPX), token ring, and other protocols that are currently available or may be made available in the future.

The cable 26 of the network 16 may be of any suitable local area network (LAN) or wide area network (WAN). For example, a token ring cable system, a twisted pair, a co-axial cable, a 10 base T, fiber optic lines, microwave or other radio frequency (RF), wireless transmission media, infrared or laser communication beams, and the like. The controller 12 may operate under any suitable operating system, but in one preferred embodiment should include a true multi-tasking operating system.

Although the controller 12 may include multiple processors, a multi-tasking operating system operating on a single processor 30 may be selected from OS/2™, UNIX, Windows NT™, Windows 95™, Macintosh™ Operating System, Next Step™, or the like. An object-oriented operating system may be suitable, but is not required. The conventional Windows™ operating system is not truly multi-tasking, and as currently configured would not be suitable.

By contrast, the resources 18, 20, 22 need not have multi-tasking operating systems. Although any resource 18, 20, 22 may include an entire general purpose computer, and may host any operating system such as Windows 95, Next Step, Macintosh Operating System, OS-2, or the like. In short, the resources 18, 20, 22 may be configured with any operating system capable of operating on a processor.

For example, the Novell™ embedded systems technology (NEST) devices and systems represent a very limited capability. However, NEST devices are capable of hosting an operating system, and thereby operating under the direction of the controller 12 in the automated test harness 10 long enough to be configured for subsequent independent operation.

That is, each resource 18 may include in a minimal configuration a central processing unit and sufficient memory, either on the individual chip associated with the processor, or independent thereof, capable of independent operation once loaded. Thus, each resource 18 need only be capable of executing instructions. In general an "executable" may be thought of as any data that may be understood by an operating system and thus resulting eventually in a machine code instruction executable by a processor.

As used here, a resource 18 may refer to any of the resources 18, 20, 22 or the like. Thus, a resource 18 may be the generalized expression for a resource 18, 20, 22 or the like.

Referring to FIG. 1, a controller 12 may include a processor 30 connected to a memory device 32 and a storage

12

device 34. Connections may be made by a bus 36. Other auxiliary devices 38 may also connect to the bus 36. In addition, the controller 12 may include an input device 42, such as a keyboard, mouse, graphical user interface with some interactive sensor for receiving user inputs, or the like. Similarly, an output device 44 may be connected to the bus 36, such as a monitor for supporting a graphical user interface, or the like.

The controller 12 may have multiple processors 46, 48 in addition to the processor 30. Nevertheless, the processor 30 may simply be configured to host a multi-tasking operating system and thereby carry on all processing of the controller 12.

The resource 20 may be the most common embodiment of the resource 18 connected to the network 16. The resource 20 may include a processor 50 connected to a memory device 52 and storage device 54 for operational storage of data, and long-term non-volatile storage, respectively. The bus 56 connecting the processor 50 to the memory device 52 and the storage device 54 may also accommodate connections to other auxiliary device 58 whether singular or plural. Similarly, the resource 22 may be configured with different software packages to operate differently than the resource 20, as discussed below, but may also include a processor 60 connected to a memory device 62 and a storage device 64 by the bus 66. Auxiliary devices 68 may also be connected to the bus 66 as needed. The resource 18, needing only a processor 70 and a sufficient memory device 74 to provide for operations of the processor 70 may or may not include a bus 76, depending upon the configuration of the memory device 74 with respect to the processor 70. The resource 18 need only host an operating system, and need not be complex, or sophisticated. That is, certain devices exist that may be desirable to temporarily control by means of a controller 12 remotely connected over an extensive network 16. Any microprocessor-based device may be a suitable resource 18. For example, numerous types of data acquisition, instrumentation, process control, and other devices exist on networks 16. Temporary slaving, followed by emancipation for independent operation, may be desirable and suitable in an automated test harness 10 in accordance with the invention.

All nodes 28 connected to the network 16 may include network cards 14A, 14B, 14C, 14D, 14E, 14F as illustrated, for providing a hardware interface with the network 16. Different networks 16 provide for different amounts of embedded hardware and software for accomplishing the functions of a network card 14A, 14B, 14C, 14D, 14E, 14F. However, in general, a network card 14 designates any one of the network cards 14A-14F.

The server 24 may be a general purpose computer and may include a processor 80 connected to a memory device 82 and storage device 84 by a bus 86. Auxiliary devices 88 may include additional storage devices 84, additional memory devices 82, and the like as needed. Inasmuch as a server 24 provides file access and management for all the nodes 28 on the network 16, the server 24 may include various auxiliary devices 88.

The software system hosted by the nodes 28 of the automated test harness 10 may include a system of controller modules 92 and resource modules 94, 96, 98 as illustrated in FIG. 2. Also, server modules 100 may be hosted by the server 24. The controller modules 92 may include a network client interface 102 for driving the network card 14A. Thus, the network client interface 102 facilitates communication by the controller 12 over the network 16. The controller modules 92 also include a multi-tasking operating system for controlling the processor 30 of the controller 12.

An automated test harness 10 may include modules 106 (ATH Modules) or processes 106 hosted on the controller

12. Processes 106 may include a resource manager 108 for monitoring all of the resources 18 connected to the network 16. A scheduler 110 may be included for scheduling tests according to the availability of resources 18 and the requirements for each specific test. Launchers 112 may be "spawned" by the scheduler 110. A launcher 112 may be instructed by the scheduler 110 to run, and may be spawned as needed. Multiple instances of the launcher 112 may run simultaneously.

The data base manager 114 may be unique to the controller 12. Alternatively, a variety of suitable data base management software modules are available. For example, Lotus Notes™ has been shown to provide an "engine" for an effective data base manager. A console 116 may include software for interfacing with a user. For example, the console 116 may include a graphical user interface including icons, windows, and other mechanisms for selection by a user with a minimum of command structures.

The ATH modules 106 may include a server 118. Although a set of server modules 100 are hosted by the server 24, the controller 12, in one currently preferred embodiment of an apparatus 10 in accordance with the invention, may include a server module 118. Thus, the controller 12 may include all of the modules required to operate the automated test harness 10 and the server 24.

Thus, in general, all of the server modules 100 may actually be incorporated into the server module 118 in the controller 12. Nevertheless, the server 24 will be discussed as a separate hardware component of the automated test harness 10, hosting the server modules 100. The resource modules 94 may include a network client interface 122 for communicating with the network 16, as well as an operating system 124. A slaving module 126 may be included to permit enslaving of the operating system 124 to the controller 12 temporarily for configuring the resource 20 prior to emancipating the resource 20 to operate independently. In addition, temporary applications 128 may be hosted in the resource 20, typically by the resource 20 downloading the temporary applications 128 from the server 24 in response to commands received in a slave mode from the controller 12.

The resource modules 96 corresponding to the resource 22, may include a network client interface 132, an operating system 134, a slaving module 136, and preloaded applications 138. Although the resource modules 96 may be identical to those of the resource modules 94, for the sake of explanation and discussion, the resource 22 is configured to have preloaded applications 138, rather than temporary applications 128 as hosted by the resource 20. Thus, although not taking full advantage of the capabilities of the automated test harness 10, a resource 22 that is completely configured, including being loaded with the software to be run, may be fed information and commands one line at a time by the controller 12. Nevertheless, a more useful automated test harness 10 may be configured using resources such as the resource 20, in which temporary applications 128 are loaded by the resource 20 as a result of the controller 12 slaving the operating system 124 temporarily to instruct the resource 20 by command line instructions. That is, once the operating system 124 has been slaved to the controller 12, the operating system 124 may be instructed on what to load and how to load it. Thereafter, the operating system 124 and the resource 20 may be emancipated to operate independently, as though they had been programmed by a user individually. Nevertheless, the resource 20 at that point (emancipated) can operate to run software, download and upload files, with a very high degree of autonomy. This is substantially different than other slaving in which individual command lines must be passed to a preloaded executables in response to queries, or prompts.

The resource modules 98 represent a minimum configuration in which a processor 70 of a resource 18 need include

only sufficient processor and memory capability to host an operating system 144. The resource module 98 may include a network client interface 142 and a slaving module 146 for facilitating communication between the controller 12 and the processor 70, or more correctly the operating system 144 hosted on the processor 70.

The network server modules 100 may include a network server interface 152 corresponding to the network client interfaces 102, 122, 132, 142. The network server interface 152 facilitates communication by the server 24 over the network 16. Thus, the controller modules 92 hosted by the controller 12 correspond to the resource module 94, 96, 98 and server modules 100 of the resources 20, 22, 18 and server 24, respectively.

The server modules 100 may include a network server interface 152 for facilitating communication by the operating system 154 over the network 16, while network applications 156 may operate "on top of" the operating system 154 for accomplishing the functional requirements of the server 24. Of course, a number of files 158 are typically stored on the storage device 84 of the server 24. Accordingly, the network applications 156 may perform management, storage, retrieval, indexing, and other associated functions for the files 158, serving the files 158 to each resource 18, 20, 22 or to the controller 12, as needed.

In general, the operating system 104 of the controller 12, or of the controller modules 92 hosted by the controller 12, should be a multi-tasking operating system such as an OS-2 or UNIX operating system, as discussed above. In addition, a network interface should render the controller 12 a node 28 on the network 16 served by the server 24. For example, the controller 12 may be set up as an OS-2 requester, or as a client in a network 16. The data base manager 114 of the controller modules 92 may require that the controller 12 host either the server software or the client software for the data base manager program. For example, the notes server or client may be associated with the Lotus Notes™ data base manager for managing all data base files associated with the automated test harness 10.

Each of the network interfaces 102, 122, 132, 142 may be configured to facilitate communication by the controller 12 at an operating system level with the resources 18, 20, 22. Thus, the controller 12, communicating as a node 28 of the network 16 may instruct the resources 18, 20, 22 to load executables. Nevertheless, as mentioned previously, the resource 22 may be preloaded with applications. This configuration gives less flexibility but may be used by the automated test harness 10. Nevertheless, even in such a configuration, the resource 22 may be enslaved by the controller 12, communicating with the operating system 34 to reconfigure the resource 22.

An explanation of the relationships between the various opcodes 180 and the threads 150 from which and to which each may be directed is explained below. However, referring to FIG. 3, the sources and destinations of all opcodes 180 may be illustrated by schematic block diagram. Since the logic of the automated test harness 10 is so intertwined among the various threads 160, 170, 172, 174, and opcodes 180, all opcodes 180 and threads 150 need to be identified, and their relationships identified, before the operation and logic of each may be properly explained with reference to each of the other opcodes 180 and threads.

In addition, Tables 1-3 below illustrate logical relations between various opcodes 180. Entries in Tables 1-3 are arranged more-or-less according to which process 106 in the second column is sending an opcode 180 in the third column. In certain circumstances, a thread 150, in general, of the scheduler 110 may send an opcode 180 to another general thread 150 of the scheduler 110.

For each opcode 180 identified in the middle or third ("opcode") column, the process 106 in the second ("from")

15

column is the sending process 106, and the process 106 in the fourth ("to") column receives the opcode 180. The sending process 106 identified in the second column may send an opcode 180 upon receiving an opcode 180, or occurrence of some other initiating event, identified in the first ("initiated by") column.

The receiving process 106 of the fourth ("to") column may send a return opcode 180 identified in the fifth ("returns") column. A returned opcode 180 is typically sent to the sending process 106 of the second ("from") column, but may be sent to another process 106, either of which will be identified in the sixth or last ("to") column.

16

One may note that no return opcodes are shown in Table 2, and thus no fifth and sixth columns. This is merely for presentation here. That is, only the opcode RESOURCE_REBOOT 256 sent to the scheduler 110 has an associated return. The CLEANUP 234 opcode is returned to the resource manager 108.

TABLE 1

OPCODES FROM SCHEDULER TO RESOURCE MANAGER					
INITIATED BY	FROM	OPCODE	TO	RETURNS	TO
	SCH	INIT_	RM 108	INIT_	SCH
	110	QUEUES 230		SUCCESS 242	110
	SCH	INIT_	RM 108	INIT_	SCH
	110	QUEUES 230		FAIL 244	110
BLD_	SCH	SETUP_	RM 108	SETUP_	SCH
SUCCESS 220	110	PROGRAM 232		FAIL 248	110
BLD_	SCH	SETUP_	RM 108	SETUP_	SCI
SUCCESS 220	110	PROGRAM 232		SUCCESS 246	110
BLD_	SCH	SETUP_	RM 108	RESERVE_	SCH
SUCCESS 220	110	PROGRAM 232		PENDING 258	110
BLD_	SCH	SETUP_	RM 108	RESERVE_	SCI
SUCCESS 220	110	PROGRAM 232		SUCCESS 260	110
TEST_	SCH	CLEANUP 234	RM 108	CLEANUP_	SCH
COMPLETE 214	110			SUCCESS 250	110
TEST_	SCH	CLEANUP 234	RM 108	CLEANUP_	SCH
COMPLETE 214	110			FAIL 248	110
RESOURCE_	SCH	CLEANUP 234	RM 108	CLEANUP_	SCH
REBOOT 256	110			SUCCESS 250	110
RESOURCE_	SCH	CLEANUP 234	RM 108	CLEANUP_	SCH
REBOOT 256	110			FAIL 248	110
LAUNCHER_	SCH	CLEANUP_	RM 108	CLEANUP_	SCH
FAIL 218	110	LAUNCHER_		LAUNCHER_	110
		FAIL 236		FAIL_	
LAUNCHER_	SCH	CLEANUP_	RM 108	CLEANUP_	SCH
FAIL 218	110	LAUNCHER_		LAUNCHER_	110
		FAIL 236			
				SUCCESS 266	
(LAUNCH	SCH	RESERVE_	RM 108	RESERVE_	SCH
ENTRY	110	REMOVE 238		REMOVE_	110
DELETED)				SUCCESS 262	
(LAUNCH	SCH	RESERVE_	RM 108	RESERVE_	SCH
ENTRY	110	REMOVE 238		REMOVE_	110
DELETED)				FAIL 264	
CONSOLE_	SCH	EXIT 204	RM 108		
EXIT 229	110				

50

TABLE 2

OPCODES FROM RESOURCE MANAGER TO SCHEDULER			
INITIATED BY	FROM	OPCODE	TO
INIT_QUEUES 230	RM 108	INIT_SUCCESS 242	SCH 110
INIT_QUEUES 230	RM 108	INIT_FAIL 244	SCH 110
SETUP_PROGRAM 232	RM 108	SETUP_SUCCESS 246	SCH 110
SETUP_PROGRAM 232	RM 108	SETUP_FAIL 248	SCH 110
CLEANUP 234	RM 108	CLEANUP_SUCCESS 250	SCH 110
CLEANUP 234	RM 108	CLEANUP_FAIL 252	SCH 110
(UNIDENTIFIED	RM 108	INCONSISTENT_	SCH 110
OPCODE)		DATA 254	
(RESOURCE REBOOTED)	RM 108	RESOURCE_REBOOT 256	SCH 110
SETUP_PROGRAM 232	RM 108	RESERVE_PENDING 158	SCH 110

TABLE 2-continued

OPCODES FROM RESOURCE MANAGER TO SCHEDULER					
INITIATED BY	FROM	OPCODE			TO
(R_P 158 SENT PREVIOUSLY)	RM 108	RESERVE_SUCCESS 160			SCH 110
RESERVE_REMOVE 238	RM 108	RESERVE_REMOVE_			SCH 110
		SUCCESS 262			
RESERVE_REMOVE 238	RM 108	RESERVE_REMOVE_			SCH 110
		FAIL 264			
CLEANUP_LAUNCHER_	RM 108	CLEANUP_LAUNCHER_			SCH 110
FAIL 236		FAIL_SUCCESS 266			
CLEANUP_LAUNCHER_	RM 108	CLEANUP_LAUNCHER_			SCH 110
FAIL 236		FAIL_FAIL 268			

15

Referring to FIG. 3, the controller 12 may operate several processes 108, 110, 112, 116, 118. In addition, each process 108, 110, 112, 116, 118 may include multiple threads. For example, the scheduler 110 may include a main thread 160 communicating with a configuration file builder thread 162, a scanner thread 164, a data base watch thread 166, and an exit thread 168. In one embodiment, the data base watch thread 166 may be incorporated, included in, the main thread 160. Similarly, the exit thread 168 may be included in the main thread 160. In general, the individual threads 162, 164, 166, 168 may be included within a main thread 160 or separated out to operate on the multi-tasking operating system 104.

The scheduler 110 may be made to operate on the controller 12, providing feedback and prompts to a user in a window of a graphical user interface of the console main thread 172, providing visible output to a user on an output device 44 of the controller 12. The output device 44 may be, for example, a monitor associated with the controller 12.

The main thread 170 of the resource manager 108 may provide management of all the resources 18 connected to the network 16. That is, the resource manager 108 tracks all resources 18, their status, their capabilities, their physical and software limitations, and provides information regarding these resources 18 and their availability to the scheduler 110, and more particularly to the main thread 160.

The console 116, although it may have multiple threads, in one embodiment presently preferred, may include a main thread 172 for implementing a graphical user interface for a user operating the controller 12. The console main thread 172 communicates to the scheduler main thread 160 and the resource manager main thread 170. By contrast, the scheduler main thread 160 and resource manager main thread 170 may communicate back and forth with one another. However, in one embodiment, the console main thread 172 may simply send out information and not receive information from the main threads 160, 170. Alternatively, other tasks and functions may be provided for or executed by the console 116.

TABLE 3

OTHER OPCODES					
INITIATED BY	FROM	OPCODE	TO	RETURNS	TO
RESERVE_	SCH	SUITES_	SCH	SETUP_	RM
SUCCESS 260	110	READY 121	110	PROGRAM 232	108
	SCH	TEST_	SCH	CLEANUP 234	RM
	110	COMPLETE 214	110		108
SETUP_	SCH	BLD_	SCH	BLD_	SCH
SUCCESS 246	110	CONFIG 224	110	SUCCESS 220	110
SETUP_	SCH	BLD_	SCH	BLD_	SCH
SUCCESS 246	110	CONFIG 224	110	FAIL 222	110
BLD_	SCH	BLD_	SCH	(LAUNCHER 112	
CONFIG 224	110	SUCCESS 220	110	INITIATED)	
BLD_	SCH	BLD_	SCH		
CONFIG 224	110	FAIL 222	110		
	ANY	SYSTEM_	SCH		
		LOG 210	110		
	LAU	LAUNCHER_	SCH		
	112	PASS 216	110		
	LAU	LAUNCHER_	SCH		
	112	FAIL 218	110		
	CON	CONSOLE_	SCH		
	116	ADDRESS 225	110		
	CON	CONSOLE_	RM 108		
	116	ADDRESS 225			
(USER ON	CON	ADD_	RM 108		
CONSOLE GUI)	116	PREFIX 226			
(USER ON	CON	DEL_	RM 108		
CONSOLE GUI)	116	PREFIX 228			
(USER ON	CON	CONSOLE_	SCH		
CONSOLE GUI)	116	EXIT 229	110		

The launcher 112 may include a launcher main thread 174, and may be configured to have other threads. However, the launcher 112 may be "spawned" in multiple versions by the scheduler main thread 160. That is, the launcher 112 may be simultaneously running in more than one instantiation, to accommodate the multiple resources 18 that must be configured and run.

Each of the threads 160, 162, 164, 166, 168, 170, 172, 174 may communicate with one another by a series of opcodes 180. For example, the opcodes 180 may be referred to as opcodes 180, and may be illustrated by the opcodes 182 communicating between the exit thread 168 and the main thread 160. Similarly, the opcodes 184 may communicate between the data base watch thread 166 and the main thread 160. The opcodes 186 may be passed from the scanner thread 164 to the main thread 160, while the opcodes 188 are passed from the launcher main thread 174 to the scheduler main thread 160. Similarly, the opcodes 192 pass from the scheduler 110 to the scheduler main thread 160 to the configuration file builder thread 162, the opcodes 194 pass from the configuration file builder thread 162 back to the scheduler main thread 160, the opcodes 196 pass from the console main thread 172 to the resource manager main thread 170, and the opcodes 198 pass from the console main thread 172 to the scheduler main thread 160. The opcodes 200 may pass from the scheduler main thread 160 to the resource manager main thread 170 while the opcodes 202 pass from the resource manager main thread 170 to the scheduler main thread 160.

In general, each opcode 180 may have a generic opcode structure 270. In one presently preferred embodiment of an apparatus 10 in accordance with the invention, an opcode structure 270 may include three pieces of information, each comprising three long (32 bit) words. The first piece of information is an identifier 272 that may be a name, or a number that uniquely identifies a particular opcode 180. A generic pointer 274 may follow the identifier 272 and may include an address for identifying generic data that may be used by the process receiving the opcode 180. Following the generic pointer 274, a resource pointer 276 may contain a memory address associated with data stored in the memory device 32 of this controller 12, or similar devices specifically for use in identifying and managing resources 18. The purpose of a generic pointer 274 and resource pointer 276 rather than a single pointer, is to simplify logic and speed up operation.

As referenced earlier, the data base watch thread 166 and the exit thread 168 may be incorporated directly into the scheduler main thread 160. Similarly, the scanner thread 164 in one embodiment of an apparatus 10 in accordance with the invention, may be incorporated into the scheduler main thread 160. However, in general, opcodes may each be made to operate similarly, or even identically. At a source thread, an opcode indicates to the source thread to reserve a memory block, that is, a specific segment of memory in a memory device such as the memory device 32. The source thread then writes the identifier 272, a generic pointer 274, and a resource pointer 276 into the reserved block of memory. The source thread then sends the address of the opcode 180 to a destination thread by writing the address to a message queue associated with the destination thread. A destination thread periodically reads all messages in a message queue associated with a destination thread. Upon reading the message received from the source thread, the destination thread receives the address of the opcode 180. The destination thread then reads the opcode in the memory block, ascertaining the identifier 272, and the pointers 274, 276. The

destination thread then may move an execution pointer in the opcode associated with the thread that has received the opcode 180, and may begin executing the opcode at the designated location. Thus, the opcode identifier 272 has served to move an execution pointer within the coded executable of the destination thread. The destination thread then executes the opcode using the data pointed to by the pointers 274, 276. In certain circumstances, the destination thread may return an opcode 180 to the source thread, or to another thread in the controller 12, and particularly the controller modules 92.

In one embodiment, opcodes 182 may include an EXIT 204. The data base watch thread 166 may send opcodes 184 to the scheduler main thread 160, including a POLL_LAUNCHQ 206, a DB_WATCH_SIGNAL 208, and a SYSTEM_LOG 210. The scanner thread 164 may send opcodes 186 including a SUITES_READY 212, a TEST_COMPLETE 214, and a SYSTEM_LOG 210 to the main thread 160. The launcher main thread 174 of any individual launcher 112 "spawned" by the scheduler 110, may return to the scheduler main thread 160 any of the opcodes 188 including a LAUNCHER_PASS 216, LAUNCHER_FAIL 218, SYSTEM_LOG 210, and others as appropriate.

The configuration file builder thread 162 may send the opcodes 194 to the main thread 160, which opcodes 194 may include a BLD_SUCCESS 220, a BLD_FAIL 222, or SYSTEM_LOG 210. The opcodes 194 may be sent by the configuration file builder thread 162 in response to one of the opcodes 192, which may include a BLD_CONFIG 224 opcode. The console main thread 172 may send a CONSOLE_ADDRESS 225 to the resource manager main thread 170. In addition, an ADD_PREFIX 226 or DEL_PREFIX 228 may be sent from the console main thread 172 to the resource manager main thread 170.

The console main thread 172 may receive no opcodes 180 from other threads. Rather, the console main thread 172 may receive its principal direction from inputs from a graphical user interface gathering inputs from a user. The console main thread 172 does send a CONSOLE_EXIT 229 opcode to the scheduler main thread 160, indicating that a system exit is in order. The CONSOLE_ADDRESS 225 may also be sent from the console main thread 172 to the scheduler main thread 160. The SYSTEM_LOG 210 may be sent by any thread to the scheduler main thread 160.

The scheduler main thread 160 may send an INIT_QUEUES 230 opcode to the resource manager main thread 170. Similarly, a SETUP_PROGRAM 232 may be forwarded to the resource manager main thread 170, which like all main threads may be referred to as the main thread 170.

A CLEANUP 234 or CLEANUP_LAUNCHER_FAIL 236 may be sent from the scheduler main thread 160 to the resource manager main thread 170. A RESERVE_REMOVE 238 or an EXIT 240 may be passed from the scheduler main thread 160 to the resource manager main thread 170.

In the return direction from the resource manager main thread 170 back to the scheduler main thread 160, a host of opcodes 202 may be sent, including INIT_SUCCESS 242, INIT_FAIL 244, SETUP_SUCCESS 246, SETUP_FAIL 248, CLEANUP_SUCCESS 250, and CLEANUP_FAIL 252. Each of the opcodes 202 provides information from the resource manager 108 to the scheduler 110 to indicate the status of a given resource 18 selected for a particular test.

In addition, the resource manager main thread 170 may send INCONSISTENT_DATA 254, RESOURCE_REBOOT 256, RESERVE_PENDING 258, RESERVE_

SUCCESS 260, RESERVE_REMOVE_SUCCESS 262, or RESERVE_REMOVE_FAIL 264 in response to various opcodes 200 received by the resource manager main thread 170.

The resource manager main thread 170 may send a CLEAN_UP_LAUNCHER_FAIL_SUCCESS 266 or a CLEANUP_LAUNCHER_FAIL_FAIL 268 to the scheduler main thread 160. As with other threads in communication with the scheduler 160, the resource manager main thread 170 may send a SYSTEM_LOG 210 to the scheduler main thread 160.

Referring now to FIG. 4, before returning to the definitions and further relationships illustrated in FIG. 3, the operation of a source thread 280 and a destination thread 290 are shown with respect to each other and an opcode 180 contained in the opcode structure 270 of FIG. 3.

In general, any thread 150 may be regarded as a source thread 280 and any other thread 150 may be regarded as a destination thread 290. Each opcode 180 may be used by a source thread 280 as illustrated in FIG. 4. At an appropriate point in the execution of the source thread 280, the reserve memory 282 step may be executed. The effect of the reserve memory 282 may be to reserve a segment or block of memory, typically in the memory device 32 of the controller 12. The write opcode 284 step may then follow, in which the source thread 280 writes an opcode 180 including all of the elements of the opcode structure 270 to the memory block reserved by the reserve memory 282 step.

After writing the identifier 272, generic pointer 274, if applicable, and resource pointer 276, as required, the source thread 280 executes a SEND_ADDRESS 286 step. The SEND_ADDRESS 286 step may include sending the address in the memory device 32 or other memory device to which the opcode 180 may be stored, to a message queue readable by a destination thread 290. A destination thread 290 may include an operation read address 288, that may have the effect of reading the code address written by the source thread 280 in the SEND_ADDRESS operation 286. The destination thread 290 periodically may read the message queue. Thus, the READ_ADDRESS 288 operation will effectively read the code address from the message queue.

The destination thread 150 next may read the opcode 180 itself at the address in the memory block, as designated by the code address read in the READ_ADDRESS operation 288.

The effect of the READ opcode 292 may be to provide an identifier 272 from the opcode structure 270 of the opcode 180, which identifier 272 indicates a location for an execution pointer in the destination thread 290. The destination thread 290 then executes a move pointer operation 294 in which the execution pointer of the processor 30 may be moved to the appropriate location designated by the identifier 272. The destination thread 290 then executes 296. The executing 296 operation effectively executes the code beginning at the execution pointer designated in the MOVE_POINTER 294 operation. The code of the destination thread 290 may have a return, or may itself write a new opcode 180 and return it to the source thread 280 or to some other thread 150. In sending a return opcode 180, the destination thread 290 may use the same operations or steps 282, 284, 286 since in such an operation, the destination thread 290 becomes the new source thread 280 for the returned opcode 180.

Referring again to FIG. 3, the operational codes 180 or opcodes 200, remembering that auxiliary thread 150 such as

the threads 164, 166, 168 may optionally be incorporated into the main thread 160 of the scheduler 110. It may be instructive to discuss the opcodes 180 associated with the scheduler 110. The EXIT opcode 204 initiates a scheduler 160 to signal the resource manager 108 with an EXIT opcode 240 to exit the system 10. Alternatively, the EXIT opcode 204 may be sent from within the scheduler main thread 160, or may be replaced by the CONSOLE_EXIT opcode 229.

The POLL_LAUNCHQ 206 may be used to instruct the main thread 160 to periodically, or upon occurrence of a signalling event, poll the launch queue data base 302 for a new record 303, indicating an addition to the launch queue.

The DB_WATCH_SIGNAL 208 opcode may be used to indicate to the main thread 160 that certain of the data bases 300 have had records 301 recently written to them. Since the apparatus 10 in one currently preferred embodiment may be queue-driven, one methodology for prompting a thread 150 to execute an instruction, may be to provide reading of data bases 300, or reading of fields in data bases written with the specific purpose of operating as flags to indicate occurrence of an awaited event or triggering event.

The SYSTEM_LOG 210 may be sent by the data base watch thread 166 to the main thread 160 to produce a message that may eventually be saved in a system log data base 310. Since the scheduler 110 manages most of the interaction between the processor 30 of the controller 12 and the data bases 300, the main thread 160 may be tasked with the operation of reporting or writing all entries into the system log data base 310. The system log data base 310 may be normally used to store reports of system errors. Thus, the SYSTEM_LOG 210 opcode may be normally sent to the scheduler main thread 160 to report system errors. No initiation opcode 180 may be required to send the SYSTEM_LOG 210, and no return opcode 180 need be returned in reply or as a direct result.

The SUITES_READY 212 opcode may be sent to the scheduler main thread 160 when a test is ready for running on a resource 18. The SUITES_READY 212 may also be sent as a result of a RESERVE_SUCCESS 260 opcode received. Thus, the main thread 160 may typically receive the SUITES_READY 212 opcode, and may actually initiate it within the main thread 160 in response to a RESERVE_SUCCESS 260 initiation opcode 180 received from the resource manager 108. The scheduler main thread 160 may send a SETUP_PROGRAM 232 return opcode to the resource manager main thread 170.

The TEST_COMPLETE 214 opcode may be sent when the scheduler 110, and more particularly the scamer thread 164 has detected that a test has completed running. Thus, no initiation opcode 180 may be required, but the main thread 160 may then send, in response, or as a result, a CLEANUP 234 return opcode to the resource manager main thread 170.

The LAUNCHER_PASS 216 opcode may be returned by a launcher main thread 174 to the scheduler 110, and more particularly to the scheduler main thread 160 when a test has been successfully launched. A successful launch of a test 305 indicates that the launcher 112 was able to configure a resource 18, also referred to as a target 18 or target resource 18, at an operating system level, and the subject resource 18 has successfully loaded the test program and the necessary data. The test 305 is therefore running. No initiation opcode 180 may be required for the LAUNCHER_PASS 216, but the launcher 112 is itself "spawned" by the scheduler main thread 160, which may be itself an initiating event. No return opcode 180 may be necessary from the main thread 160.

The LAUNCHER_FAIL 218 may be sent by the launcher main thread 174 to the scheduler main thread 160 if a test 305 has not been successfully launched. Some reasons why a launch may fail may include the failure of a "login" command from a launcher 112 to a resource 18, failure of a "map" command to map the necessary drives, or perhaps more properly, for example, virtual drives on the storage devices 54, 64 or memory devices 52, 62, 72 of the resources 18. As discussed above, the resource 18 refers generally to all resources 18, 20, 22, and the like. The LAUNCHER_FAIL 218 requires no initiation opcode 180, since a launcher is "spawned" by the scheduler 110. No return opcode 180 may be required.

The BLD_SUCCESS 220 may be sent from the configuration file builder thread 162 to the scheduler main thread 160 when the scheduler 110, and more specifically, the configuration file builder thread 162 has successfully organized the information necessary to run a test 305. The initiation opcode BLD_CONFIG 224 may be first received by the configuration file builder thread 162 from the main thread 160. No return opcode 180 may be required.

The receipt of the BLD_SUCCESS 220 may be an initiating event for the scheduler main thread 160. The response by the scheduler main thread 160 may be to "spawn" a launcher 112, which will itself initiate a test 305, setting up the proper resources 18 to conduct the test 305.

The BLD_FAIL 222 opcode may be returned to the main thread 160 from the configuration file builder thread 162 when the information necessary to operate a test 305 has not been successfully organized by the configuration file builder thread 162. For example, if datafiles are not present, if required entries are not available, if a token value is not defined, if an operating system refuses to function properly, or if any flaw in logic or data, then the configuration file builder thread 162 may not be able to provide all of the configuration information needed by the launcher 112. The initiation opcode 180 may be a BLD_CONFIG 224 opcode, but no return opcode 180 may be necessary.

The BLD_CONFIG 224 may be sent to the configuration file builder thread 162 of the scheduler 110 by the main thread 160. The function of the opcode 224 may be to compile and organize all information associated with the opcode 224 so that a launcher 112 may be "spawned" by the main thread 160, and will have all of the data necessary to run a test 305. An initiation opcode 180 for the opcode 224 may be the SETUP_SUCCESS 246 received from the resource manager main thread 170. Thus, the resource manager main thread 170, having identified that the hardware resources 18 are available, properly equipped, and otherwise ready for employment in a test 305, sends the initiation opcode SETUP_SUCCESS 246 to the main thread 160 of the scheduler 110. Possible return opcodes sent from the scheduler 110 to the configuration file builder thread 162 may include the BLD_SUCCESS 220 or the BLD_FAIL 222 opcodes.

Thus, the opcodes 212, 214, 220, 222, 224 originate in a thread of the scheduler 110 and may be sent to the originating thread or another thread of the scheduler 110. The SYSTEM_LOG 210, in contrast, may be returned by any process 106 to the scheduler main thread 160.

The CONSOLE_ADDRESS 225 may be sent by the console main thread 172 to the resource manager main thread 170. This opcode 225 may be sent during an initialization for the automated test harness 10 for the purpose of sending an address of global memory, available to all processes 106 and threads 150, to the resource manager 108,

and more particularly to the main thread 170. Similarly, the CONSOLE_ADDRESS 225 opcode may be sent from the console 116 to the scheduler 110, typically from the main thread 172 to the main thread 160. Since global memory may be used for all the data structures associated with any opcode, the block of global memory established by the CONSOLE_ADDRESS 225 may be available to all processes 106. No initiation opcode 180 may be necessary, since the opcode 225 may be sent during the start-up phase for the automated test harness 10. Similarly, no return opcode 180 may be necessary.

The main thread 172 may also send an ADD_PREFIX 226 or a DEL_PREFIX 228 to the resource manager main thread 170. The opcode ADD_PREFIX 226 may be a signal to the resource manager 108 that another resource prefix should be added to a list. That is, a prefix may be an initiation opcode 180 in the name of a service advertising protocol (SAP) in a slaving program protocol. For example, a slaving program such as NCONTROL™ is a Novell™ slaving program that has been found suitable for facilitating the temporary slaving operation for setup, configuring the resource 18 through its operating system by the controller 12. After the setup operation, the resource 18 may be "emancipated," left to operate independently, loading applications and files over the network as needed by applications running on the resource 18.

Thus, a prefix may be added to a SAP string by a resource manager 108 in response to the opcode 226. The resource manager 108 then uses the prefix to know which resources 18 are to be used for a given task. The resource manager 108 may handle multiple prefixes. An initiation opcode 180 is not necessarily required, although the opcode 226 may be initiated by another opcode 180. However, in one embodiment of an apparatus 10 made in accordance with the invention, a user may make a selection from a graphical user interface hosted by the console 116. A selection of the feature designated to add a prefix may be selected by a user. In response to the selection by a user, the console 116 sends the ADD_PREFIX 226 to the scheduler 110.

The DEL_PREFIX 228 effectively instructs the resource manager 108 to cease looking for resources 18 that are advertising with the SAP prefix associated with the opcode 228. As with the ADD_PREFIX 226, the opcode 228 has no return opcode 180. Similarly, no initiation opcode 180 may be required. However, as with the opcode 226, a selection by a user of an icon, such as a delete button, selection box, or the like presented on screen of a graphical user interface of the console 172 may be used to initiate the opcode 228.

The CONSOLE_EXIT 229 opcode may be sent to the scheduler 110 to signal a system exit. As with other opcodes 180 initiated by the console 116, the opcode 229 may be initiated by a user selection from the graphical user interface presented on a screen of a monitor associated with the console 116. Thus, no initiation opcode 180, that is opcode 180 associated with initiation, is required. Similarly, no return opcode 180 may be appropriate. The appropriate response by the scheduler 160 may be a system exit.

The INIT_QUEUES 230 may be sent by the scheduler 110 to synchronize with the resource manager 108, thus ensuring that all queues are properly set up, identified, and functioning for receiving messages for the appropriate threads 150. No initiation opcode 180 may be required, although possible return opcodes may include INIT_SUCCESS 242 or INIT_FAIL 244 opcodes returned by the resource manager 108.

The SETUP_PROGRAM 232 opcode sent by the scheduler 110 has the effect of requesting resources 18 needed for

25

a test 305. The information associated with the opcode 232 identifies resource information structures identifying the nature of the needed resources 18. Thus, a "specification" of sorts may be associated with the opcode 232. An initiation opcode BLD_SUCCESS 220 gives rise to the opcode 232. Thus, when configuration by the thread 162 is complete, the scheduler main thread 160 makes a resource request of the resource manager 108 with the opcode 232. Possible return opcodes may include SETUP_SUCCESS 246, SETUP_FAIL 248, RESERVE_PENDING 258, or RESERVE_SUCCESS 260 sent by the resource manager 108 to the scheduler 160.

The CLEANUP 234 opcode may be used by the scheduler 110 to instruct the resource manager 108 to free resources 18, making those resources 18 available for use in a new test 305. This is usually done when a resource 18 has been allocated or reserved for a particular test 305 needing the capabilities of the selected resource 18. With each opcode 234, a list of information structures associated with the selected resource 18 is associated. Although information may be sent with an opcode 180, information may also be stored and pointed to by the pointers 274, 276 associated with the opcode 234. Initiation opcodes 180 giving rise to the CLEANUP 234 opcode may include TEST_COMPLETE 214, from the scanner thread 164, or RESOURCE_REBOOT 256 sent by the resource manager 108 in response to a reboot, typically occurring outside of the control of the automated test harness 10. Possible return opcodes 180 may include CLEANUP_SUCCESS 250, and CLEANUP_FAIL 252.

The CLEANUP_LAUNCHER_FAIL 236 bears some resemblance to the CLEANUP 234 opcode. However, the opcode 236 may be sent after the scheduler 110 receives a LAUNCHER_FAIL 218 opcode from the launcher 112. Thus, the effects are the same, although the initiation sources are different. The initiation opcode 180 for the opcode 236 may include the LAUNCHER_FAIL 218, while possible return opcodes may include CLEANUP_LAUNCHER_FAIL_SUCCESS 266, or CLEANUP_LAUNCHER_FAIL_FAIL 268.

The RESERVE_REMOVE 238 opcode may be sent by the scheduler 110 to the resource manager 108 when an entry is deleted from the launch queue data base 302. That is, to the extent that an entry in the launch queue data base 302 has certain resources 18 reserved for the test 305 specified, those resources 18 need to be released when a test 305 is canceled. Thus, the opcode 238 instructs the resource manager 108 to release the reserve resources 18 and delete the reservation entry from any internal tables maintained by the resource manager 108. The opcode 238 may have associated with it an instance identification of a deleted entry in the launch queue data base 302. The instance identification, sometimes referred to as instance ID, may be a unique number assigned by an automated test harness 10 to each test 305 that is to be run. Thus, with each instance of a launcher 112, or of a test 305 to be run by a launcher 112, an instance ID may be assigned. Thus, any test 305 may be tracked according to its unique instance ID. Possible initiation opcodes 180 for the opcode 238 may be used, but in one embodiment of an apparatus 10 in accordance with the invention, the deletion of an entry from the launch queue data base 302 may be detected by the scheduler main thread 160 monitoring the launch queue 302. Thus, the main thread 160 may then initiate the opcode 238 upon detection of deletion of an entry. Possible return opcodes 180 may include RESERVE_REMOVE_SUCCESS 262 and RESERVE_REMOVE_FAIL 264, returned by the resource manager 108 to the

26

scheduler 110. Although return opcodes 180 are often returned to an initiating or source thread 280, such need not be the case. A return opcode 180, in general, may be simply an output opcode 180 associated with a thread 150 operating in accordance with another received opcode 180.

The EXIT 240 opcode may be sent to the resource manager 108 to signal a system exit. The initiation opcode 180 may be controlled by an exit thread 168 sending an EXIT 204 opcode to the scheduler main thread 160. However, in one presently preferred embodiment of an apparatus 10 made in accordance with the invention, all control over the EXIT 204 opcode may reside in the console 116. Thus, an appropriate initiation opcode 180 may be a CONSOLE_EXIT 229 sent from the console main thread 170 to the scheduler main thread 160. No return opcode 180 may be required, since a resource manager 108 may be programmed to properly log off or otherwise exit all resources 18 from the system 10.

The INIT_SUCCESS 242 opcode may be sent from the resource manager 108 to the scheduler 110 to instruct the scheduler 110 to complete synchronization, thus ensuring that all queues (message queues of all threads 150) are functioning. Thus, all protocols are properly operating to synchronize messaging between the resource manager 108 and scheduler 110. Initiation opcodes 180 may include INIT_QUEUES 230, although no return opcode 180 may be required.

The INIT_FAIL 244 opcode 180 may be sent by the resource manager 108 when initialization fails. Initiation opcodes 180 may include INIT_QUEUES 230, although no return opcode 180 may be required, similar to the opcode 242.

The SETUP_SUCCESS 246 opcode 180 may be sent when the resource manager 108 is successful in allocating all of the resources requested by the scheduler 110 for a specified test 305. Initiation opcodes 180 may include SETUP_PROGRAM 232 from the scheduler 110, but no return opcode 180 may be required.

The SETUP_FAIL 248 may be sent to the scheduler 110 when an error occurs with respect to information supplied with or associated with the SETUP_PROGRAM 232 opcode 180 received by the resource manager 170. Thus if the request for information identified with a request for preparation with a test 305 is improper, the opcode 232 acts as an initiation opcode 180 for the opcode 248.

The CLEANUP_SUCCESS 250 opcode may be returned when the resource manager 108 has been successful in freeing up the necessary resources 18 identified in data associated with the CLEANUP 234 opcode. Thus, the CLEANUP 234 operates as an initiation opcode 180, although no return opcode 180 may be required. Identification of resources 18, as discussed above, may occur by specification of any or all of several parameters identifying the capacity of a resource 18 required to run a test 305. Similarly, the CLEANUP_FAIL 252 opcode may be returned by the resource manager 108 in response to a CLEANUP 234 opcode. When the resource manager 108 is unable to free up the necessary resources 18 identified in data associated with the opcode 234, the opcode 252 may be appropriate. No return opcodes 180 may be required from the scheduler 110 in response to the opcode 252.

An INCONSISTENT_DATA 254 opcode may be sent by the resource manager 108 when an unrecognized opcode 180 is received. Theoretically, an unrecognized opcode 180 should not occur, particularly in a compiled code or in a previously debugged code. However, the opcode 254 serves

as a backup, an initiation opcode 180 being any undefined opcode 180 received by the resource manager 108. No return opcode 180 may be required.

A RESOURCE_REBOOT 256 may be sent when the resource manager 108 detects that a resource 18 has been rebooted. Rebooting should not normally occur, and therefore indicates that a person or program independent of the apparatus 10 has rebooted a resource 18 that was originally logged on to the network 16 as an available resource 18 for the apparatus 10. Also, if a resource 18 has been temporarily enslaved by a controller 12, it may be unloaded (have its software removed) and then be loaded again. Thus, if the slaving software (e.g. NControl™) has been unloaded from the resource 18 in question and then loaded again, the resource 18 has effectively been removed from the apparatus 10 and then replaced. The opcode 256 treats this operation as a reboot. No initiation opcode 180 may be required, since the initiating event may be an outside action, typically by a user, detected by the resource manager 108 upon polling of its resources 18. Possible return opcodes 180 sent by the scheduler 110 in response to the opcode 256 may include CLEANUP 234.

The RESERVE_PENDING 258 opcode may be sent by the resource manager 108 to the scheduler 110 if requested resources 18 are not available to the resource manager 108. The opcode 258 indicates to the scheduler 110 that the resource manager 108 will notify the scheduler 110 when the appropriate resources 18 become available to run the designated test 305. Thus, whenever the resource manager 108 has been unable to locate sufficient resources 18 having the appropriate capabilities to run a test 305, the opcode 258 may be returned. Thereafter, the resource manager 108 waits for sufficient resources 18 to become available to conduct the requested tests 305.

When the resource manager 108 has previously sent a RESERVE_PENDING 258 opcode to the scheduler 110, it may subsequently send a RESERVE_SUCCESS 260 opcode. Thus, whereas an initiation opcode 180 of SETUP_PROGRAM 232 may result in a RESERVE_PENDING 258 opcode from the resource manager 108, no return opcode 180 is sent immediately. Rather, the resource manager 108 simply tracks the resources 18 in view of the pending request, and sends the opcode 260 when the proper resources 18 are available. Thus, once the resource manager 108 has "collected" sufficient resources 18, those resources 18 are designated in the data associated with the opcode 260. Thus, although no initiation opcode 180 or return opcode 180 may be required, a SETUP_PROGRAM 232 may be regarded as a quasi-initiation opcode 180, but cannot control the timing of the opcode 260 being returned.

The RESERVE_REMOVE_SUCCESS 262 may be sent when the resource manager 108 has been able to remove a test reservation (an entry in its table of resources 18 maintained for allocation to tests 305) associated with tests 305, from the launch queue 302. That is, when the resource manager 108 will no longer attempt to reserve resources 18 for a test 305, it may send the opcode 262 without ever devoting those resources 18 to that test 305. The initiation opcode 180 may be a RESERVE_REMOVE 238, although no return opcode 180 may be necessary for the opcode 262 or the opcode 264.

The RESERVE_REMOVE_FAIL 264 opcode may be sent when the resource manager 108 has been unable to remove a test reservation from the table of resources 18 maintained. Thus, the entry may never have existed, or the appropriate resources 18 may have been located and allo-

cated previously, making the initiation opcode 180, opcode 238, unnecessary. No return opcode 180 may be required.

The CLEANUP_LAUNCHER_FAIL_SUCCESS 266 may be sent when the resource manager 108 has successfully freed up those resources 18 to be allocated to a test 305 that previously failed to launch properly. Similarly, the CLEANUP_LAUNCHER_FAIL_FAIL 268 opcode may be sent when the resources 18 have not been successfully freed up. Either opcode 266, 268 may be initiated by the initiation opcode CLEANUP_LAUNCHER_FAIL 236, and may require no return opcode 180.

Several data bases 300 are associated with the data base manager 114 of the automated test harness 10. Each data base 300 may be comprised of a number of records 301, each record 301 containing some number of fields 320 for containing data.

Referring now to FIGS. 5-6, the data bases 300 are designated by a reference number although it is proper to speak of a database 300 or a record 301 in the database 300 interchangeably in certain circumstances. The launch queue data base 302 stores data associated with each launch. A launch may be made by a launcher 112 "spawned" by the scheduler 110 in order to run either a test 305 (actually a test identified by a record 305) from the test data base 304, a suite 307 of tests 305, (identified by the records 307 from the suite data base 306), or a group 309 (or group identified by record 309) of tests 305, suites 307, or groups 309, from the group data base 308. After running a test 305, or a suite 307 or group 309 of tests 305, the automated test harness 10 may store the results in a launch history data base 310. Other data bases 300 or files may be filled with any information desired from a test 305. However, in general, the automated test harness 10 may run any test 305 without regard to what operations are conducted or what data may be generated by the test 305. Thus, a user may determine any number of executable files and output files for a test 305, independent of the automated test harness 10. Thus, the launch history data base 310 concerns primarily the history of the automated test harness 10 and launching and completing tests 305.

To set up the automated test harness 10, certain settings may be saved in a settings data base 312. The settings data base 312 facilitates rapid set-up and reconfiguration of the automated test harness 10, including the controller 12, and any associated hardware and software.

The system log data base 314 may be used to store errors encountered during attempts by the automated test harness 10 to launch and run a test 305 or tests 305.

Information particular to each resource 18 in the apparatus 10 may be stored in a resource data base 316. Thus, when the resource manager 108 seeks resources 18 to run a test 305 scheduled by the scheduler 110, the resource manager 108 may make a determination of the suitability of any resource 18, based on a record 317 of the resource data base 316.

Referring now to FIGS. 5-6, fields 320 in each of the data bases 300 may be configured in a variety of ways. In one presently preferred embodiment of an apparatus 10 made in accordance with the invention, the fields 320 of the launch queue may include a type 331 designating the type of launch, whether a test 305, suite 307, or group 309. A name 332 of the launch may correspond to the test name 340, suite name 361, or group name 368 as appropriate. The launcher 333 indicates the launcher 112 responsible for the test 305 in question. The status 334 may store information relative to the current status in operation of the automated test harness

10 occupied by the launch 332 in question. The priority 335 indicates a designation of importance assigned by a user. Items with lowest priority numbers, highest priority, will be launched first from first received in the launch queue 302 to last received in the launch queue 302, followed by all launches next in priority. The iterations field 336 indicates how many times an individual launch should be run. That is, one may speak of tests 305 or launches 333, but each may refer to an instance 339 corresponding to one instantiation of a launcher 112 tasked by the main thread 160 of the scheduler 110 to slave resources 18, configure those resources 18 to operate, and then emancipate those resources 18 to act in accordance with instructions obtained in consequence of the launcher 112 setting up the resources 18.

Token definitions 337 may be stored in the token definitions field 337 and token files may be identified in the token files field 338. Tokens are those parameters or variables replaceable during any individual operation with specific data, but defined by place holders within a code.

Each record 305 in the test data base 304 may include a test name 340, a description 341, and an executable name 342. A path directing a resource 18 to the actual software to operate a test 305 may be stored at the path field 343, along with parameters 344 for a specific instance 339. Usage information between a programmer and a user may be stored at field 345, while a wrapper name, indicating the designation of a test 305, may be stored in 346. Network dependencies, including types of networks and protocols may be stored in the network dependencies field 347. The type of the controller 12 may be stored at master resource types 348, while resource 18 types may be identified at slave resource types 349. A server 24 may be designated by the server 350 field, whereas the context 351, user 352, and password 353 may be filled with individualized information particular to a user and testing scenario.

The drive mapping 354 field may enable additional drives to be mapped on a computer such as a controller 12 or server 24 on a network 16. The files to tokenize 355 allows each token in a user-specified file to be replaced with a specified value defined elsewhere and stored in memory device 32. The instance 356 may be a unique serial number assigned to each test name 340 and peculiar to the particular instantiation of that test name 340 by a launcher 112.

Similarly, each suite 307 may have a suite name 361, also referred to simply as a name 361, in the name field 361. A text description may be provided in the description field 362, along with instructions for usage in the usage field 363. The individual responsible, typically a user, and possible a directing individual requesting certain testing, may be identified as a contact in the contact field 364. All test names 340 that are included within a suite 361 may be listed, separated by some delimiter, in the test list field 365. The number of tests 305 may be input in number tests field 366, or the number of tests 305 may simply be determined automatically by the coding for reading the delimiters in the test list field 365. To the extent that certain code may be tokenized, a token file path field 367 may be filled with information to direct a resource 18 to the proper token files to fill in dummy variables.

Each group 309 of tests 305, group 309 of suites 307, or group 309 of groups 309, may be assigned a group name in the group name field 368. A description, such as a textual description may be stored in the description field 369. A contact or responsible individual may be identified in the entered by field 370, while usage instructions may be stored in the usage field 371. Similar to the test data base 304 and suite data base 306, token files may be identified in a token file field 372.

Since, unlike the test data base 304 and suite data base 306, the group data base 308 may accept any type of testing grouping, a member type, whether a test 305, suite 307, or group 309, within a group name 368 may be identified in the member type field 373. A name, such as a test name 340, suite name 361, or group name 368, from a test 305, suite 307, or group 309, respectively, associated with the group name 368 in question, may be stored in the member main field 374. A priority may be assigned by a user in the priority field 375, and a group 309 may be repeated, just as a test 305 or suite 307 may be repeated, some number of iterations designated in the iterations field 376. Parameters to be passed may be stored in the parameters field 378, while the instance number unique to the group name 368 and its instantiation by the launcher 112 may be identified by an instance field 379. The instance fields 339, 356, and 379 may be used as indices to associate any testing instantiation with the test data and any record 311 in the launch history data base 310.

The launch history data base 310 may include a launch time set by a clock, and recorded in a launch time field 381, as well as a most recent time in which the record 311 of the data base 310 in question was last updated or stored in an update time field 382. With each instantiation of a launch by the launcher 112, a type may be stored in the type field 384 and a name in the name field 385, corresponding to the type (test 305, suite 307, group 309) and the name 340, 361, 368 corresponding thereto in the name field 385. The corresponding instance 356, 377, 379 may be stored in the instance field 386. Thus, the launch data base 302 may be indexed to the individual test 305, suite 307, or group 309, as is the launch history data base 310. The status field 387 may be used to store information regarding the status of the launch being recorded, while the data field 388 may be used to gather other data pertinent to the individual launch.

The automated test harness data base 312, also referred to as the ATH data base 312, may be used to specify particular, standardized, configurations of the automated test harness 10. Each record 313 of the ATH data base 312 may include a name field 391 for identifying a standardized setup configuration, an automated test harness server identifier in the ATH server field 392, the path in an ATH path field 393, with a test directory or subdirectory for the location of executables associated with a test 305 stored in the test directory field 394. The path to the databases 300 may be specified in the DB manager path field 395, while a text editor of choice may be specified by a user or for a user in the text editor field 396 to enable modification of coding, comments, and other text strings by an appropriate engine.

The system log data base 314 may contain various error messages. In general, however, a message information field 397 may contain the text of a message or additional debugging information relating to possible sources of a message, or both. The reported by field 398 may store information regarding a process 106 or a hardware resource 18 responsible for generating a particular error message.

The resources 18 may be specified in substantial detail. Similarly, any particular testing set-up may be provided with a specification for resources 18 according to any or all fields 320 in the resource data base 316.

The resource data base 316 may contain information regarding the automated test harness 10, in general. However, in one embodiment of the apparatus 10 made in accordance with the invention, the resource database 316 contains information touching configuration of the hardware suite employed in the resources 18, and, optionally, the

31

server 24. In addition, certain other information, such as specifications related to peripheral equipment attached to a resource 18, may be significant to an individual laboratory using the automated test harness 10.

General information concerning the automated test harness 10 may include a location field 402 for specifying a geographical location such as a laboratory room number or other geographical identifier. A node address field 404 may store the actual network node address of the controller 12 on the network 16. The status field 406 may identify the state of readiness or configuration, while the prefix field 408 may contain a prefix for uniquely identifying files pertaining to the automated test harness.

The operating system type field 410 may contain information regarding the particular operating system type associated with a controller 12, or may be made to list a number of operating systems that may be hosted by the controller 12. The instance field 412 may temporarily contain one of the instances from an instance field 356, 377, 379 associated with a testing regimen associated with a resource 18 identified by a record 317 in the data base 316. Thus, the instance field 412 does not contain a permanently associated number unique to a resource 18, but rather the designation of the current instance to which the resource 18 is dedicated.

Each of the data bases 302, 304, 306, 308, 310, 312, 314, 316 has associated with it a series of individual records 303, 305, 307, 309, 311, 313, 315, 317, respectively. Each record 303, 305, 307, 309, 311, 313, 315, 317 has associated with it a plurality of fields 320.

Related to the hardware of a resource 18 may be the brand field 414 identifying the actual brand name of the resource 18. The CPU field 416 and the speed field 418 in the record 317 of the resource data base 316 identify exactly the CPU identifying number or name and the speed or megahertz at which the processor 70 of the resource 18 operates.

The drive size field 422 and the drive size field 424 may identify the size and megabytes of hard drives associated with the resource 18. Topology field 426 identified the topology discussed above under which a resource 18 may be connected to the network 16, while the net card field 428 identifies the type of network card 14C, 14B, 14E associated with a particular resource 18.

The IPX backbone field 430 may be the identifier for the network protocol used by the automated test harness 10 for communicating over an internetwork through a router 25 connecting the network 16 to an internetwork 17 (refer to FIG. 1). The IPX internal field 432 and IP internal field 434 may contain identifying numbers, names, or other strings to identify the protocols used within the automated test harness 10 over the network 26 between the network cards 14A, 14B, 14C, 14D, 14E, 14F. IPX is a potential protocol that has been used in various networks, and indicates in general the network protocol of the network 16. Similarly, the IP internal field 434 relates to the internetwork protocol and may be replaced by some other protocol as appropriate.

Relating to the software suite of the automated test harness 10, the function field 436 may contain data relating to software that may be or has been organically hosted on a resource 18. The OS type field 438 and OS version field 440 identify the operating system type as well as the current version number hosted by each resource 18. For some resources 18, operating system types may be changed readily, in which event a second OS field 442 and second OS version field 444 may be used to store information relating to a second operating system under which a resource 18 may operate.

32

Other information that may be pertinent to a laboratory or organization operating an automated test harness 10 may include a capital asset number field 446 for containing a capital inventory number such as is assigned by most companies in controlling capital assets. Similarly, a machine serial number may be stored in a serial number field 448, while the bus type may be identified in a bus type field 450 and the video card type and drive controller may be designated in a video card field 452 and drive controller field 454.

Alternate topology may be designated by another topology field 456, while an additional network card 14 may be hosted in certain types of systems, such as a wireless card and a wired card, one of which may be a default card. Thus, a net card field 458 may be used to designate an additional network card 14, associated with a resource 18, or an alternative network card 14. Similarly, a network backbone protocol for the network 16 or internetwork 17 may be designated in the IPX back field 460 and the IP back field 462, respectively. Similarly, the data link layer addressing information associated with the MAC-LAYER protocols according to the International Organization for Standardization Open Systems Interconnection model (ISO/OSI) model may be designated in a MAC back field 464. The software associated with the slaving model 446 for slaving the operating system 144 of a resource 18 to the controller 12 of the automated test harness 10, may be specified in a slave type field 466.

Other fields may be created in records 303-317 in the data bases 302-316, respectively, or other new data bases or records may be created as convenient. However, in one embodiment of an apparatus 10 made in accordance with the invention, the foregoing fields 320, records 303-317, and data bases 302-316 may be used to identify data determined to be useful in operating an automated test harness 10.

From the above discussion, it will be appreciated that the present invention provides an apparatus and method for temporarily slaving a resource 18 or target computer 18 to a controller 12, during which event the controller 12 operates as a command line controller communicating with the operating system of the resource 18 to configure the resource 18, and after which the resource 18 may load and run software for conducting tests independently.

The present invention may be embodied in other specific forms without departing from its spirit or essential characteristics. The described embodiments are to be considered in all respects only as illustrative, and not restrictive. The scope of the invention is, therefore, indicated by the appended claims, rather than by the foregoing description. All changes which come within the meaning and range of equivalency of the claims are to be embraced within their scope.

What is claimed and desired to be secured by United States Letters Patent is:

1. An apparatus for running test software, the apparatus comprising:

- a network for communicating data;
- a target operably connected to the network, the target comprising:
 - a first network interface operably connected to the network,
 - a first processor connected to the first network interface and provided with a first operating system,
 - a first memory device operably connected to a store data transferred to and from the first processor; and
- a controller operably connected to a network to communicate data with the target over the network, the controller comprising:

a second network interface operably connected to the network to communicate data between the controller and the network,

a second processor operably connected to the second network interface for selectively, based on resource information received from the target and temporarily providing operating system command line instructions for controlling the operating system of the target, while the operating system of the target is continuously operating, to configure the target,

a second memory device for storing data communicated to and from the second processor, and

a storage device connected to the network for storing files of data.

2. The apparatus of claim 1 wherein the controller is further provided with a launcher for communicating with the target, and for enslaving and controlling the operating system of the target during a setup operation to configure the target.

3. The apparatus of claim 2 wherein the second processor is programmed to emancipate the target to operate independently after the setup operation, and the target is configured to load and run an application independently of the controller.

4. The apparatus of claim 1 wherein the controller is further provided with a resource manager for managing data corresponding to identification and performance characteristics of the target, and to availability of the target to run applications.

5. The apparatus of claim 1 wherein the controller further comprises a server for storing and retrieving files.

6. The apparatus of claim 5 wherein the files comprise datafiles selected from applications for running on the first processor, control applications containing executables for controlling loading and running of the applications, application datafiles containing data corresponding to parameters used by the first processor in running the applications, and result datafiles containing data corresponding to results output by the first processor while running the applications.

7. The apparatus of claim 1 wherein the controller is further provided with a database manager for storing and retrieving application datafiles and result datafiles communicated between the first processor and the storage device.

8. The apparatus of claim 1 wherein the controller is further provided with a scheduler for acquiring data corresponding to a queue of applications to be run, scheduling to run an application associated with the queue, confirming that all data needed to run the application has been assembled, and monitoring the target to determine completion of running of the application.

9. The apparatus of claim 8 further comprising a database manager for storing, and for retrieving over the network, application datafiles and result datafiles.

10. The apparatus of claim 9 further comprising a resource manager for acquiring and storing data corresponding to identification and characteristics of the target, and to availability of the target to run applications.

11. The apparatus of claim 10 further comprising a launcher for communicating commands to the operating system of the target for configuring the target prior to emancipation of the target to operate independently.

12. The apparatus of claim 11 further comprising a server for transferring files communicated over the network to and from the storage device.

13. The apparatus of claim 1 further comprising a plurality of targets, the target being a first target of the plurality of targets.

14. The apparatus of claim 13 wherein a second target of the plurality of targets is provided with a second operating system different from the first operating system.

15. The apparatus of claim 1 wherein the target further comprises a storage device operably connected to the first processor for storing application datafiles, result datafiles, and applications.

16. The apparatus of claim 1 wherein the target is provided with an operating system for downloading an executable to be run on the first processor.

17. The apparatus of claim 16 wherein the executable is selected from a command file, an interpretable language file, a batch file, and an instruction file written in machine code.

18. The apparatus of claim 1 wherein the target is provided with an operating system for receiving executables downloaded by the controller to be run on the first processor.

19. The apparatus of claim 1 further comprising a second target operably connected to the network and having a second operating system different from the first operating system.

20. The apparatus of claim 1 wherein the second memory device is provided with a plurality of queues for receiving operational codes communicated by a plurality of processes running on the second processor, each operational code being readable by a process of the plurality of processes for controlling an operation of the process.

21. The apparatus of claim 1 wherein the second processor is programmed to run simultaneously a plurality of software modules, the plurality of software modules comprising:

- a database manager for storing and retrieving application datafiles to be used as inputs for applications executable by the second processor, and result datafiles containing outputs of the applications;
- a resource manager for storing data corresponding to identification, characteristics, and availability of the target to run the applications;
- a launcher for communicating with the operating system of the target and for configuring the target to operate independently of the controller in running applications and uploading result files to the server, and for selectively downloading applications to the target and instructing the target to download applications to the target result datafiles to the server; and
- a scheduler for acquiring data corresponding to applications to be run, selecting applications to be identified to the target for downloading to the target, and for spawning instantiations of the launcher.

22. A method of running software on a plurality of computers, the method comprising:

- connecting a target to a network, the target comprising:
 - a first processor having a first operating system, and
 - a first network interface operably connected to the processor to communicate data between the processor and the network,
- connecting a controller to the network to communicate over the network with the first operating system, the controller comprising:
 - a second network interface operably connected to the network to communicate data between the controller and the network,
 - a second processor having a second operating system and operably connected to the second network interface for selectively, based on resource information received from the target, and temporarily provide operating system command line instructions for controlling the first operating system of the target, and

35

a memory device for storing data communicated to and from the second processor;
connecting a server, containing a storage device, to the network for storing and retrieving files transferred over the network;
enslaving the first operating system of the target to be controlled by the controller;
transmitting operating system command line instructions from the controller to the first operating system of the target to be executed by the first operating system of the target to configure the target;
emancipating the first operating system of the target to operate independently of the controller; and
loading by the target, independently of the controller, a file from the server onto the target.
23. The method of claim 22 wherein the first operating system and the second operating system are different from each other.

36

24. The method of claim 22 further comprising running an application of the applications on the first processor.
25. The method of claim 24, further comprising creating by the target a result datafile containing data corresponding to results obtained by the first processor while running the applications.
26. The method of claim 25, further comprising independently uploading by the target the result datafile to the server.
27. The method of claim 22 wherein the file is selected from applications containing instructions executable by the first processor, control applications containing instructions executable by the first processor for controlling running of applications, a batch file interpretable by the first operating system, and application datafiles containing data corresponding to parameters used by the first processor in running applications.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION

PATENT NO. : 5,909,544
DATED : June 1, 1999
INVENTOR(S) : Graham et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

On the title page section [75] Inventors please delete "Micheil" and insert therefor
--Michiel--.

Signed and Sealed this
Sixteenth Day of November, 1999

Attest:



Q. TODD DICKINSON

Attesting Officer

Acting Commissioner of Patents and Trademarks

WEST Generate Collection

Print

L1: Entry 4 of 5

File: USPT

Jun 1, 1999

DOCUMENT-IDENTIFIER: US 5909544 A
TITLE: Automated test harness

Detailed Description Text (34):

In addition, Tables 1-3 below illustrate logical relations between various opcodes 180. Entries in Tables 1-3 are arranged more-or-less according to which process 106 in the second column is sending an opcode 180 in the third column. In certain circumstances, a thread 150, in general, of the scheduler 110 may send an opcode 180 to another general thread 150 of the scheduler 110.

Detailed Description Text (43):

Each of the threads 160, 162, 164, 166, 168, 170, 172, 174 may communicate with one another by a series of opcodes 180. For example, the opcodes 180 may be referred to as opcodes 180, and may be illustrated by the opcodes 182 communicating between the exit thread 168 and the main thread 160. Similarly, the opcodes 184 may communicate between the data base watch thread 166 and the main thread 160. The opcodes 186 may be passed from the scanner thread 164 to the main thread 160, while the opcodes 188 are passed from the launcher main thread 174 to the scheduler main thread 160. Similarly, the opcodes 192 pass from the scheduler 110 to the scheduler main thread 160 to the configuration file builder thread 162, the opcodes 194 pass from the configuration file builder thread 162 back to the scheduler main thread 160, the opcodes 196 pass from the console main thread 172 to the resource manager main thread 170, and the opcodes 198 pass from the console main thread 172 to the scheduler main thread 160. The opcodes 200 may pass from the scheduler main thread 160 to the resource manager main thread 170 while the opcodes 202 pass from the resource manager main thread 170 to the scheduler main thread 160.

Detailed Description Text (45):

As referenced earlier, the data base watch thread 166 and the exit thread 168 may be incorporated directly into the scheduler main thread 160. Similarly, the scanner thread 164 in one embodiment of an apparatus 10 in accordance with the invention, may be incorporated into the scheduler main thread 160. However, in general, opcodes may each be made to operate similarly, or even identically. At a source thread, an opcode indicates to the source thread to reserve a memory block, that is, a specific segment of memory in a memory device such as the memory device 32. The source thread then writes the identifier 272, a generic pointer 274, and a resource pointer 276 into the reserved block of memory. The source thread then sends the address of the opcode 180 to a destination thread by writing the address to a message queue associated with the destination thread. A destination thread periodically reads all messages in a message queue associated with a destination thread. Upon reading the message received from the source thread, the destination thread receives the address of the opcode 180. The destination thread then reads the opcode in the memory block, ascertaining the identifier 272, and the pointers 274, 276. The destination thread then may move an execution pointer in the opcode associated with the thread that has received the opcode 180, and may begin executing the opcode at the designated location. Thus, the opcode identifier 272 has served to move an execution pointer within the coded executable of the destination thread. The destination thread then executes the opcode using the data pointed to by the pointers 274, 276. In certain circumstances, the destination thread may return an opcode 180 to the source thread, or to another thread in the controller 12, and particularly the controller modules 92.

Detailed Description Text (48):

The console main thread 172 may receive no opcodes 180 from other threads. Rather, the console main thread 172 may receive its principal direction from inputs from a graphical user interface gathering inputs from a user. The console main thread 172 does send a CONSOLE.sub.--EXIT 229 opcode to the scheduler main thread 160, indicating that

a system exit is in order. The CONSOLE.sub.-- ADDRESS 225 may also be sent from the console main thread 172 to the scheduler main thread 160. The SYSTEM.sub.-- LOG 210 may be sent by any thread to the scheduler main thread 160.

Detailed Description Text (51):

In the return direction from the resource manager main thread 170 back to the scheduler main thread 160, a host of opcodes 202 may be sent, including INIT.sub.-- SUCCESS 242, INIT.sub.-- FAIL 244, SETUP.sub.-- SUCCESS 246, SETUP.sub.-- FAIL 248, CLEANUP.sub.-- SUCCESS 250, and CLEANUP.sub.-- FAIL 252. Each of the opcodes 202 provides information from the resource manager 108 to the scheduler 110 to indicate the status of a given resource 18 selected for a particular test.

Detailed Description Text (59):

Referring again to FIG. 3, the operational codes 180 or opcodes 180, remembering that auxiliary thread 150 such as the threads 164, 166, 168 may optionally be incorporated into the main thread 160 of the scheduler 110. It may be instructive to discuss the opcodes 180 associated with the scheduler 110. The EXIT opcode 204 initiates a scheduler 160 to signal the resource manager 108 with an EXIT opcode 240 to exit the system 10. Alternatively, the EXIT opcode 204 may be sent from within the scheduler main thread 160, or may be replaced by the CONSOLE.sub.-- EXIT opcode 229.

Detailed Description Text (65):

The LAUNCHER.sub.-- PASS 216 opcode may be returned by a launcher main thread 174 to the scheduler 110, and more particularly to the scheduler main thread 160 when a test has been successfully launched. A successful launch of a test 305 indicates that the launcher 112 was able to configure a resource 18, also referred to as a target 18 or target resource 18, at an operating system level, and the subject resource 18 has successfully loaded the test program and the necessary data. The test 305 is therefore running. No initiation opcode 180 may be required for the LAUNCHER.sub.-- PASS 216, but the launcher 112 is itself "spawned" by the scheduler main thread 160, which may be itself an initiating event. No return opcode 180 may be necessary from the main thread 160.

Detailed Description Text (66):

The LAUNCHER.sub.-- FAIL 218 may be sent by the launcher main thread 174 to the scheduler main thread 160 if a test 305 has not been successfully launched. Some reasons why a launch may fail may include the failure of a "login" command from a launcher 112 to a resource 18, failure of a "map" command to map the necessary drives, or perhaps more properly, for example, virtual drives on the storage devices 54, 64 or memory devices 52, 62, 72 of the resources 18. As discussed above, the resource 18 refers generally to all resources 18, 20, 22, and the like. The LAUNCHER.sub.-- FAIL 218 requires no initiation opcode 180, since a launcher is "spawned" by the scheduler 110. No return opcode 180 may be required.

Detailed Description Text (67):

The BLD.sub.-- SUCCESS 220 may be sent from the configuration file builder thread 162 to the scheduler main thread 160 when the scheduler 110, and more specifically, the configuration file builder thread 162 has successfully organized the information necessary to run a test 305. The initiation opcode BLD.sub.-- CONFIG 224 may be first received by the configuration file builder thread 162 from the main thread 160. No return opcode 180 may be required.

Detailed Description Text (70):

The BLD.sub.-- CONFIG 224 may be sent to the configuration file builder thread 162 of the scheduler 110 by the main thread 160. The function of the opcode 224 may be to compile and organize all information associated with the opcode 224 so that a launcher 112 may be "spawned" by the main thread 160, and will have all of the data necessary to run a test 305. An initiation opcode 180 for the opcode 224 may be the SETUP.sub.-- SUCCESS 246 received from the resource manager main thread 170. Thus, the resource manager main thread 170, having identified that the hardware resources 18 are available, properly equipped, and otherwise ready for employment in a test 305, sends the initiation opcode SETUP.sub.-- SUCCESS 246 to the main thread 160 of the scheduler 110. Possible return opcodes sent from the scheduler 110 to the configuration file builder thread 162 may include the BLD.sub.-- SUCCESS 220 or the BLD.sub.-- FAIL 222 opcodes.

Detailed Description Text (71):

Thus, the opcodes 212, 214, 220, 222, 224 originate in a thread of the scheduler 110 and may be sent to the originating thread or another thread of the scheduler 110. The

SYSTEM.sub.-- LOG 210, in contrast, may be returned by any process 106 to the scheduler main thread 160.

Detailed Description Text (78):

The SETUP.sub.-- PROGRAM 232 opcode sent by the scheduler 110 has the effect of requesting resources 18 needed for a test 305. The information associated with the opcode 232 identifies resource information structures identifying the nature of the needed resources 18. Thus, a "specification" of sorts may be associated with the opcode 232. An initiation opcode BLD.sub.-- SUCCESS 220 gives rise to the opcode 232. Thus, when configuration by the thread 162 is complete, the scheduler main thread 160 makes a resource request of the resource manager 108 with the opcode 232. Possible return opcodes may include SETUP.sub.-- SUCCESS 246, SETUP.sub.-- FAIL 248, RESERVE.sub.-- PENDING 258, or RESERVE.sub.-- SUCCESS 260 sent by the resource manager 108 to the scheduler 160.

Detailed Description Text (82):

The EXIT 240 opcode may be sent to the resource manager 108 to signal a system exit. The initiation opcode 180 may be controlled by an exit thread 168 sending an EXIT 204 opcode to the scheduler main thread 160. However, in one presently preferred embodiment of an apparatus 10 made in accordance with the invention, all control over the EXIT 204 opcode may reside in the console 116. Thus, an appropriate initiation opcode 180 may be a CONSOLE.sub.-- EXIT 229 sent from the console main thread 170 to the scheduler main thread 160. No return opcode 180 may be required, since a resource manager 108 may be programmed to properly log off or otherwise exit all resources 18 from the system 10.



US005887166A

United States Patent [19]
Mallick et al.

[11] **Patent Number:** 5,887,166
[45] **Date of Patent:** Mar. 23, 1999

[54] **METHOD AND SYSTEM FOR
CONSTRUCTING A PROGRAM INCLUDING
A NAVIGATION INSTRUCTION**

5,421,014 5/1995 Bucher .
5,452,459 9/1995 Drury et al .
5,524,247 6/1996 Mizuno .
5,524,250 6/1996 Chesson et al .

[75] **Inventors:** Soumya Mallick; Robert G.
McDonald; Edward L. Swarouth, all
of Austin, Tex.

Primary Examiner—Majid A. Banabkan
Attorney, Agent, or Firm—Casimer K. Salys; Brian F.
Russell; Andrew J. Dillon

[73] **Assignee:** International Business Machines
Corporation, Armonk, N.Y.

[57] **ABSTRACT**

[21] **Appl. No.:** 767,491

[22] **Filed:** Dec. 16, 1996

[51] **Int. Cl.⁶** G06F 9/00

[52] **U.S. Cl.** 395/672

[58] **Field of Search** 395/670, 672,
395/677, 674

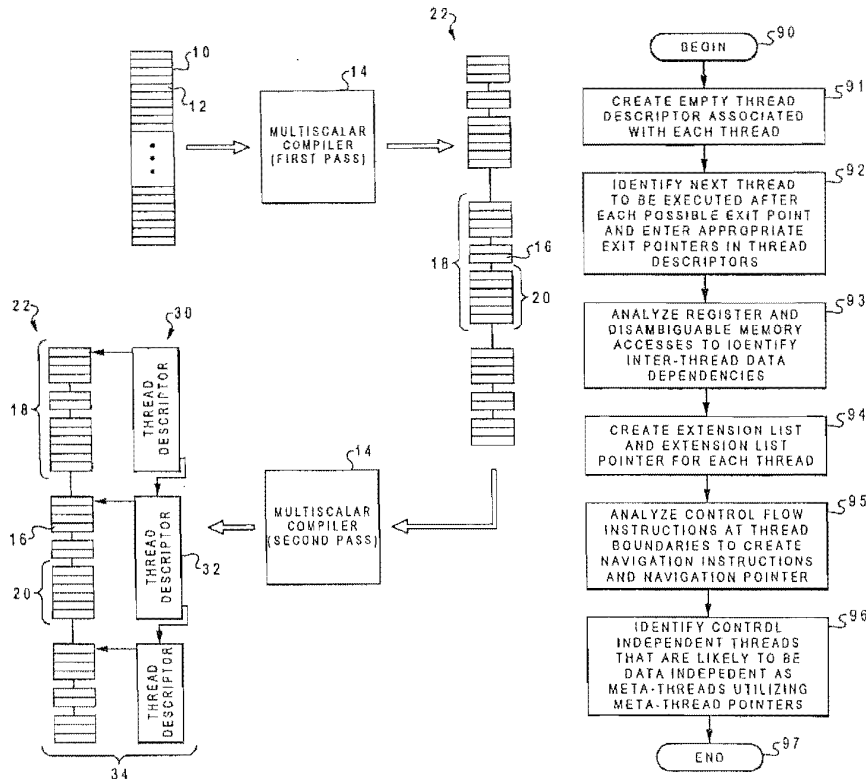
A method and system are provided for constructing a program executable by a processor including one or more processing elements for executing threads and a thread scheduler for assigning threads to the processing elements for execution. According to the method, a plurality of threads are provided that each include at least one control flow instruction. From one or more control flow instructions within the plurality of threads, a condition upon which execution of a particular thread depends is determined. In response to the determination, at least one navigation instruction executable by the thread scheduler is created that indicates that the particular thread is to be assigned to one of the processing elements for execution in response to the condition.

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,179,702 1/1993 Spix et al .
5,339,415 8/1994 Strout, II et al .
5,404,898 4/1995 Stowers .

23 Claims, 18 Drawing Sheets



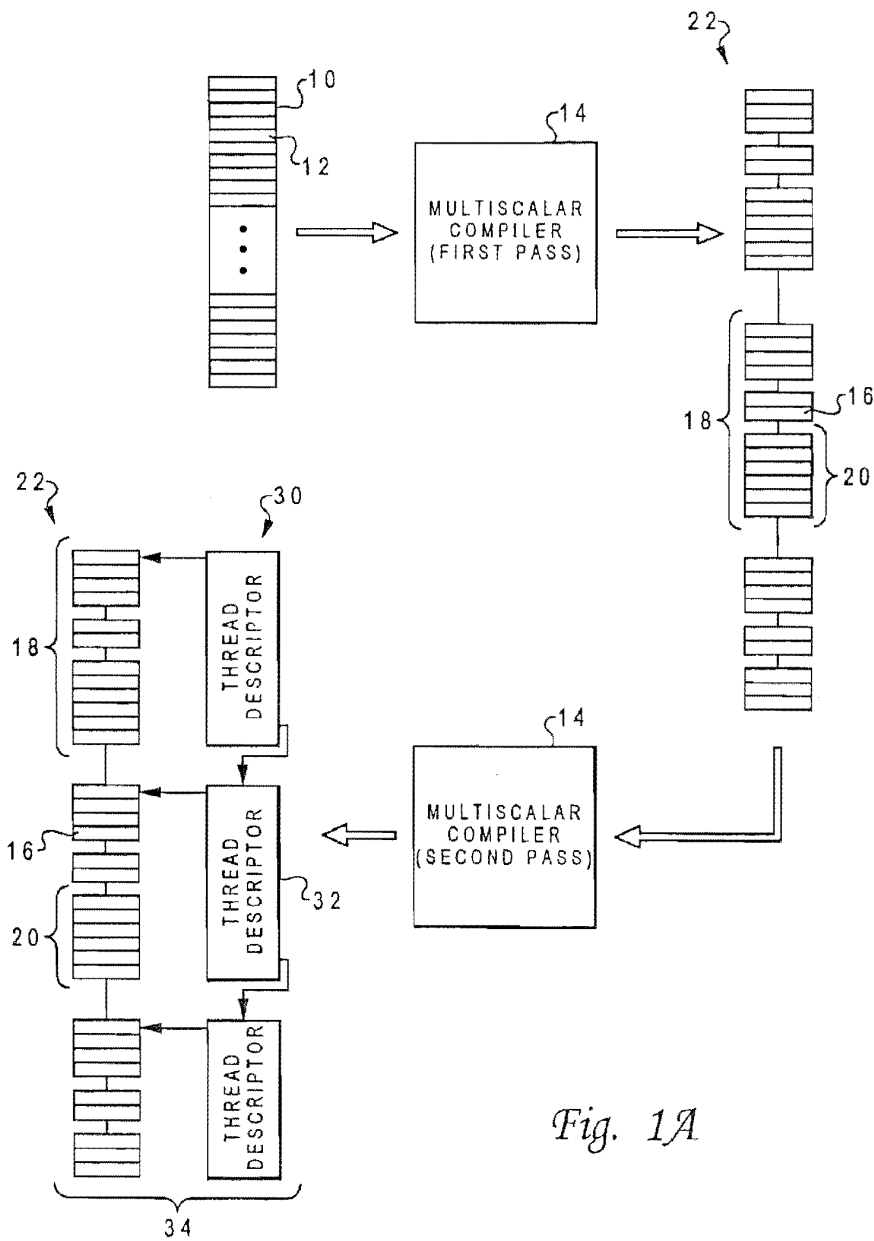


Fig. 1A

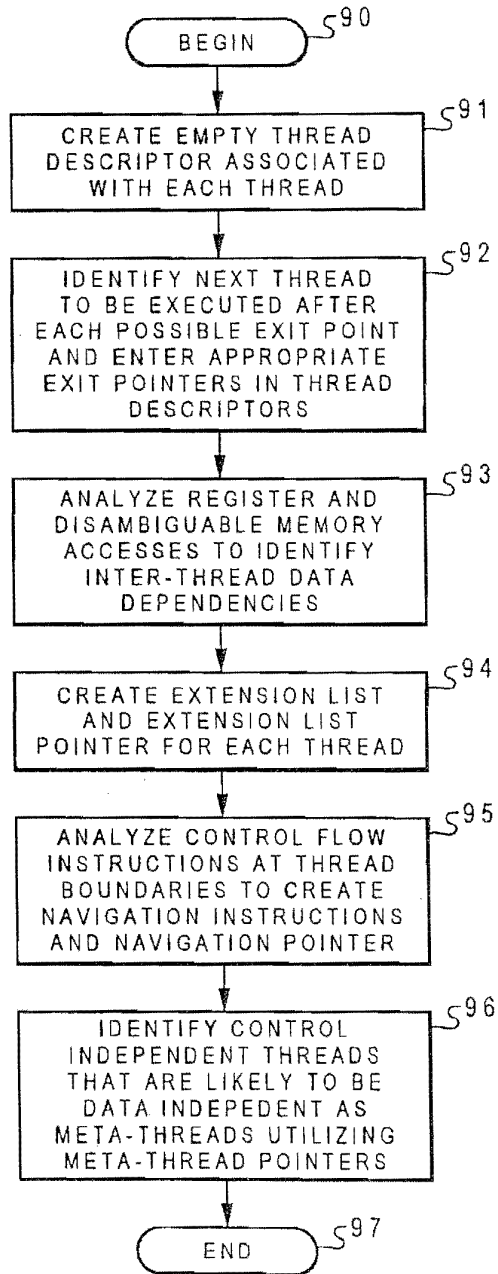


Fig. 1B

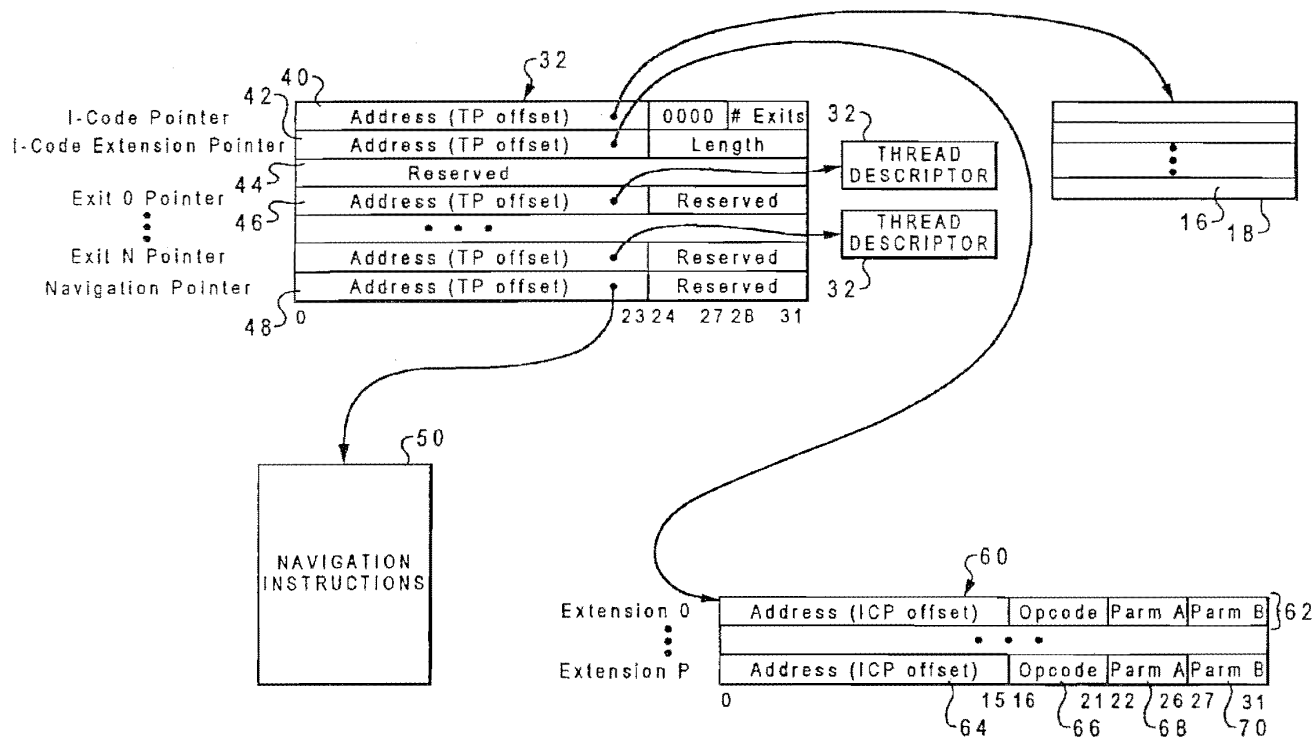


Fig. 2

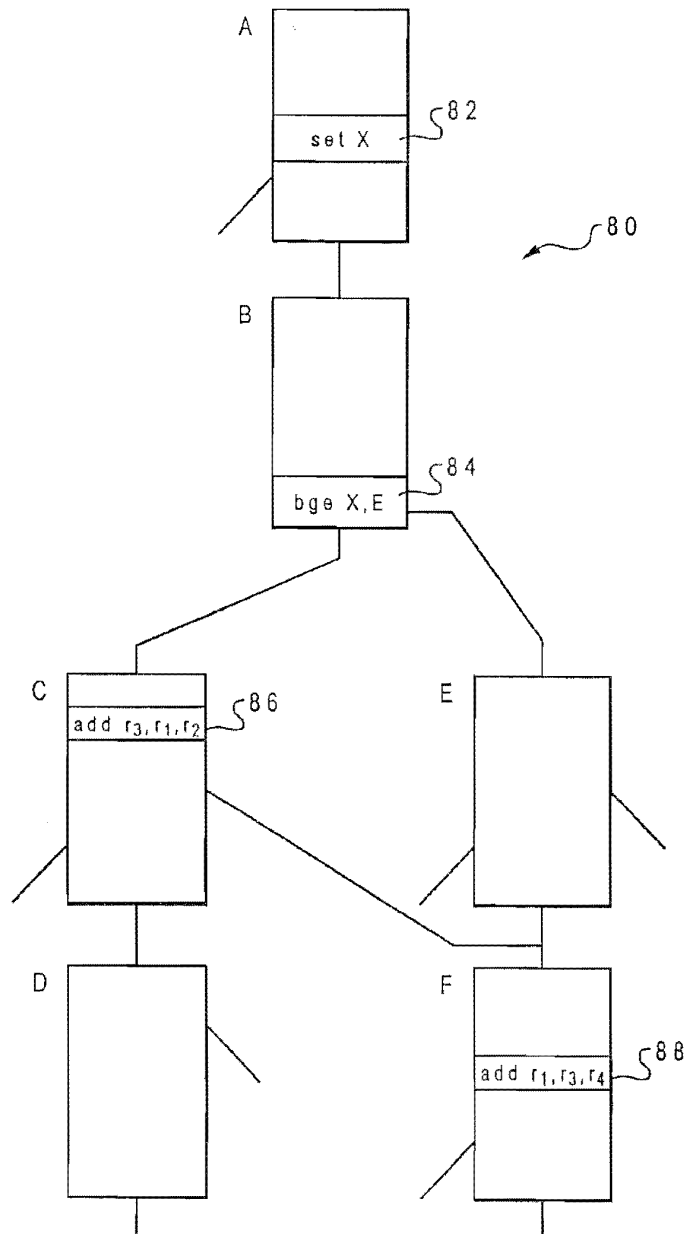


Fig. 3

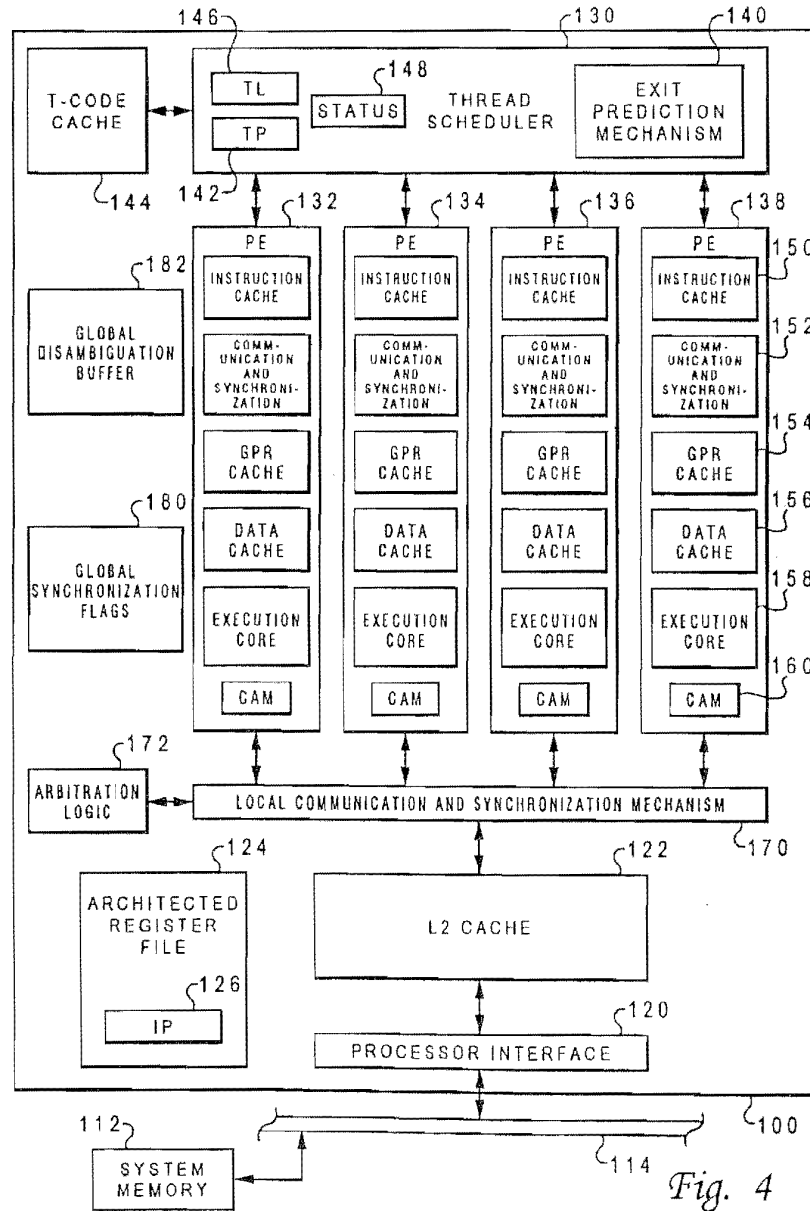


Fig. 4

Global Synchronization Flags

SF0	0
SF1	1
SF2	0
SF3	0
SF4	0
SF5	1
SF6	0
SF7	0
SF8	1
SF9	1
SF10	0
SF11	0
⋮	⋮
SF30	0
SF31	0

180

Fig. 5

Concurrent Thread pipeline

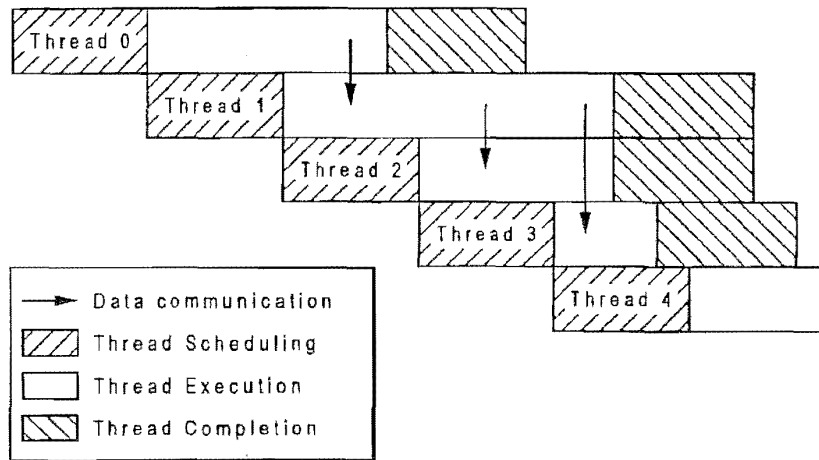


Fig. 6

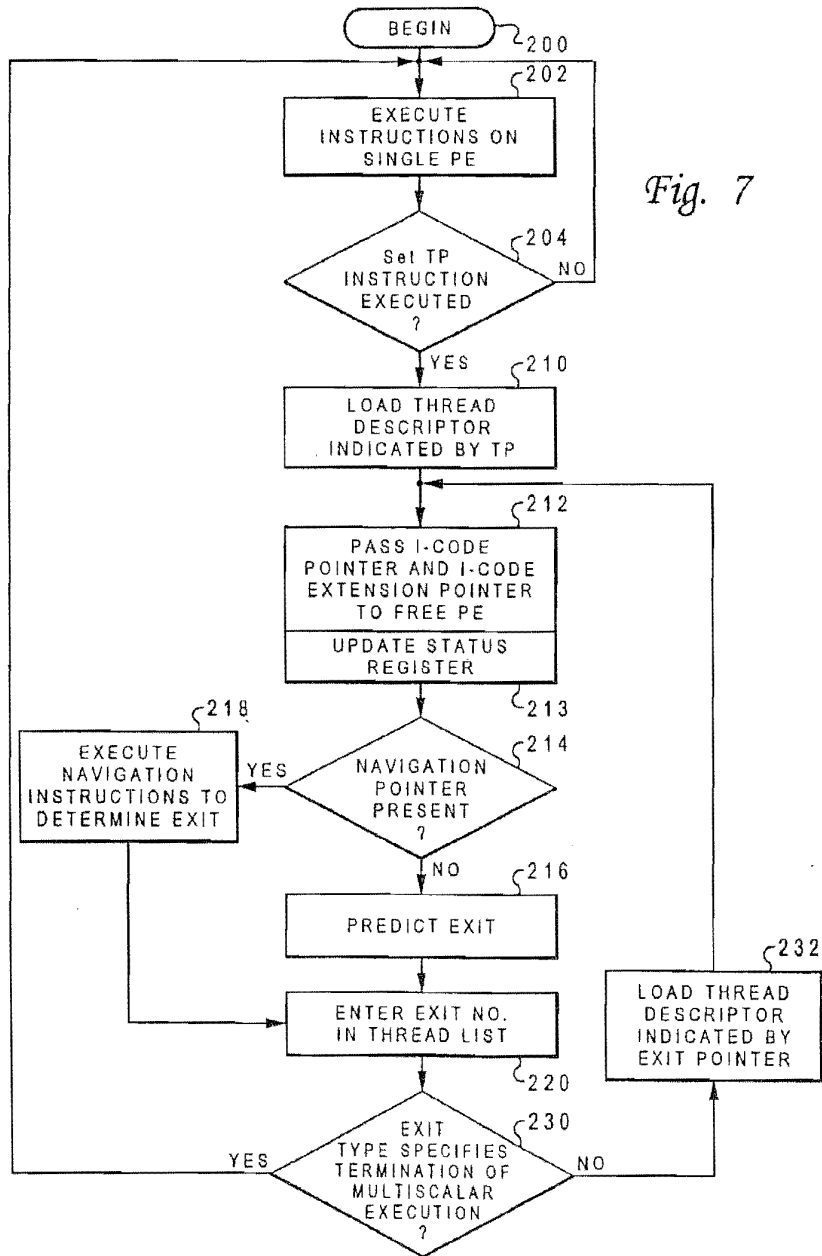


Fig. 7

Fig. 8

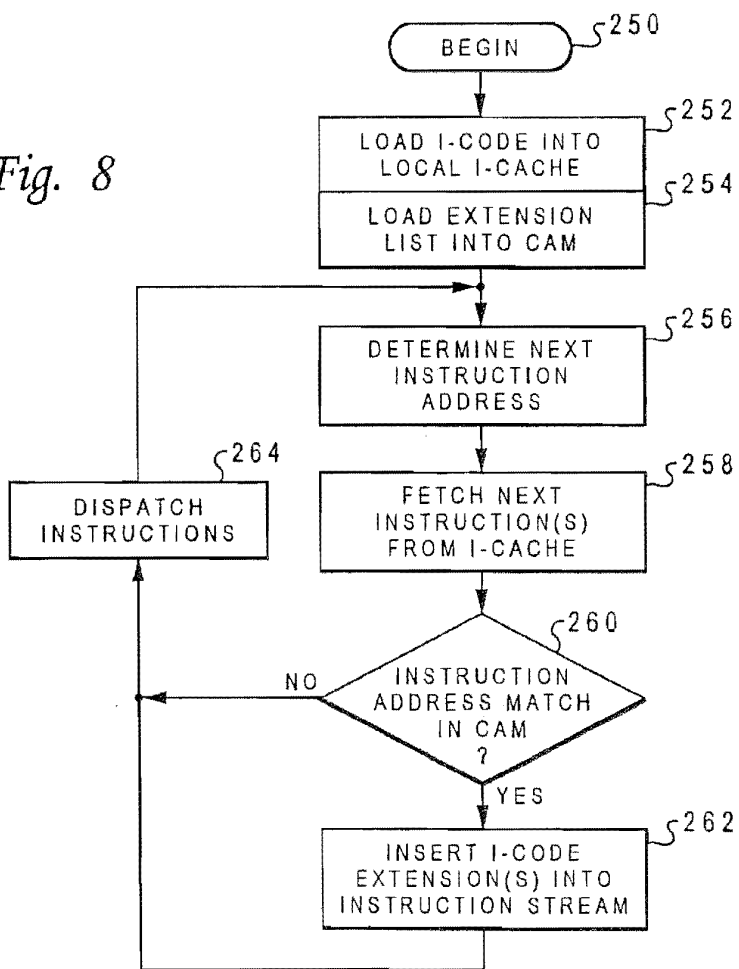
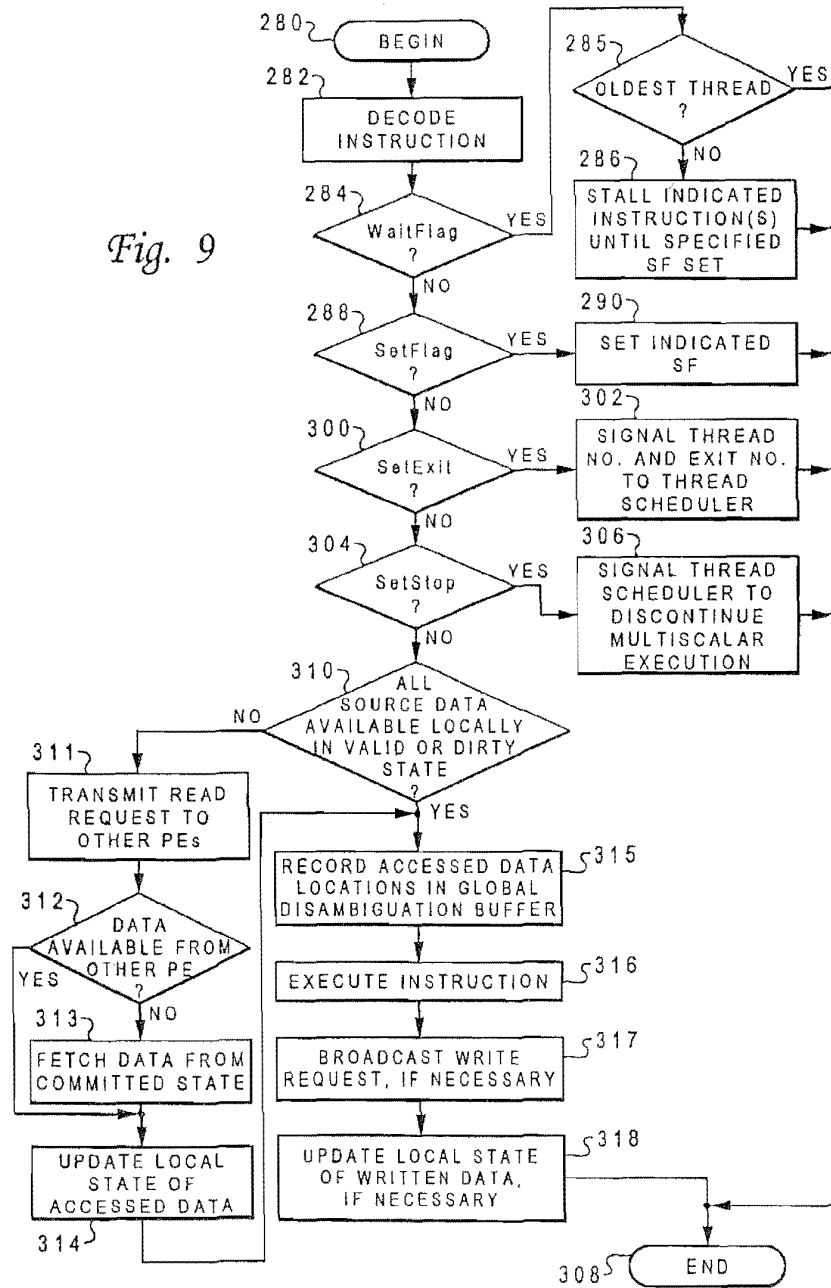


Fig. 9



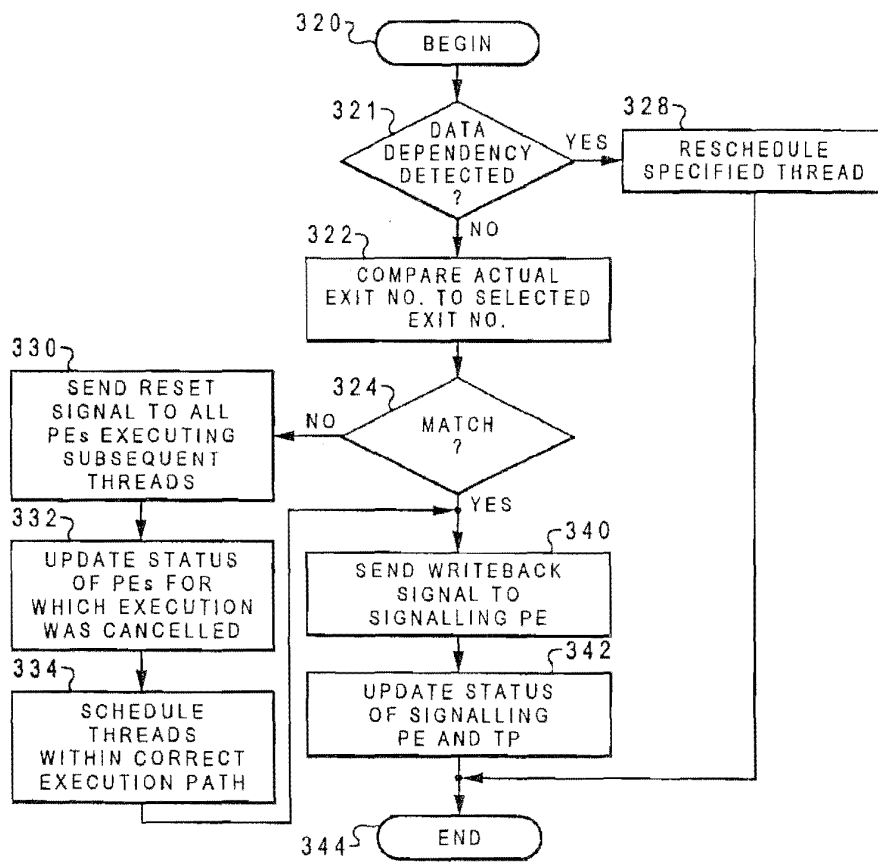


Fig. 10

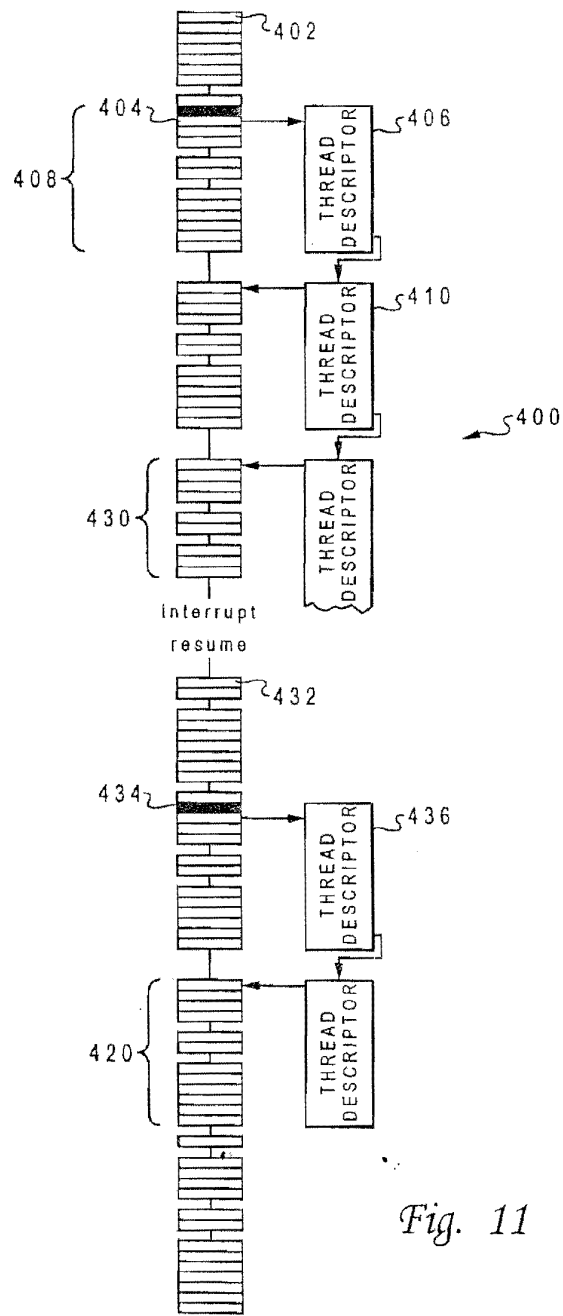


Fig. 11

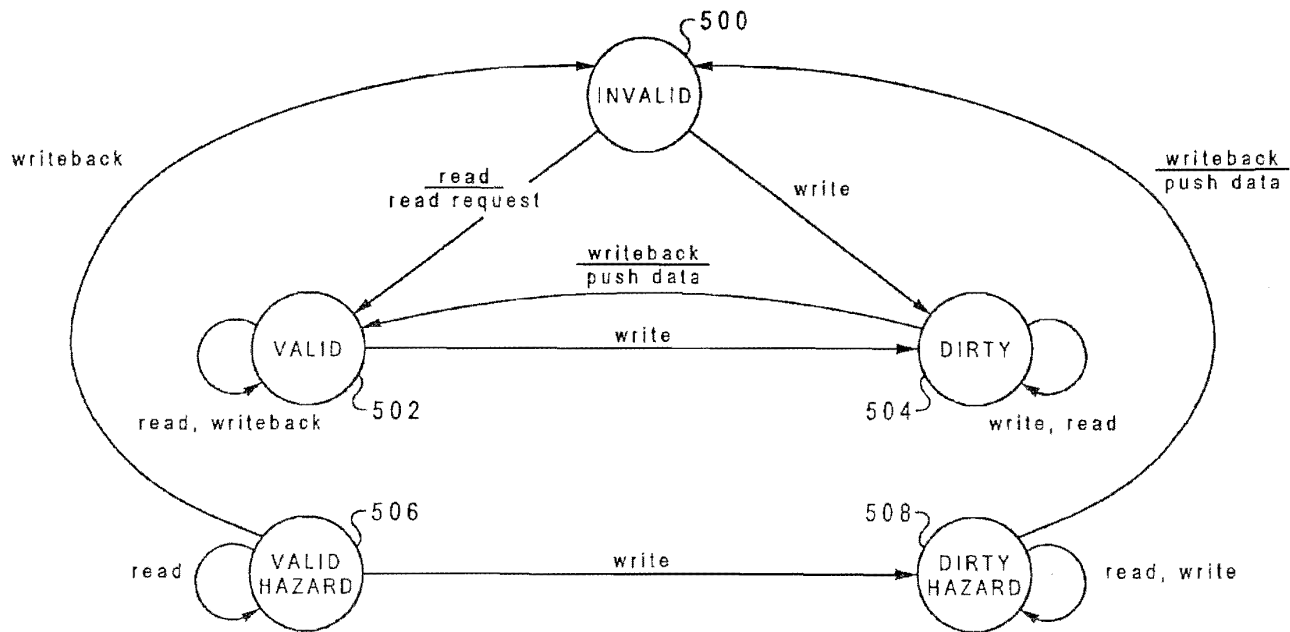


Fig. 12

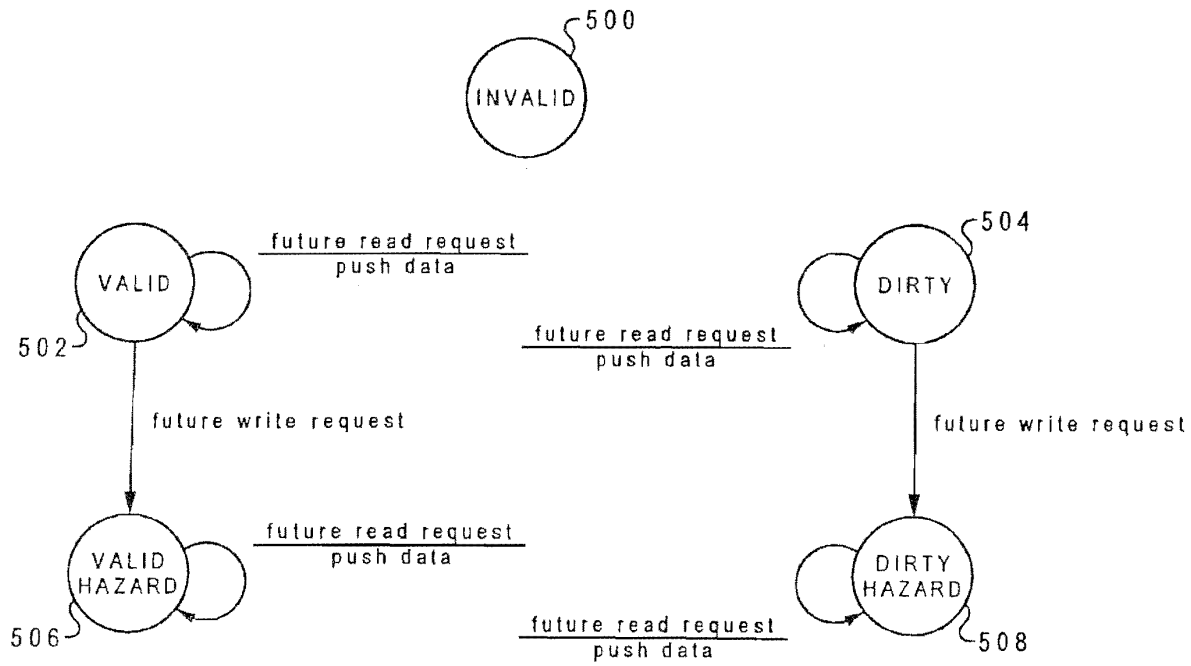


Fig. 13

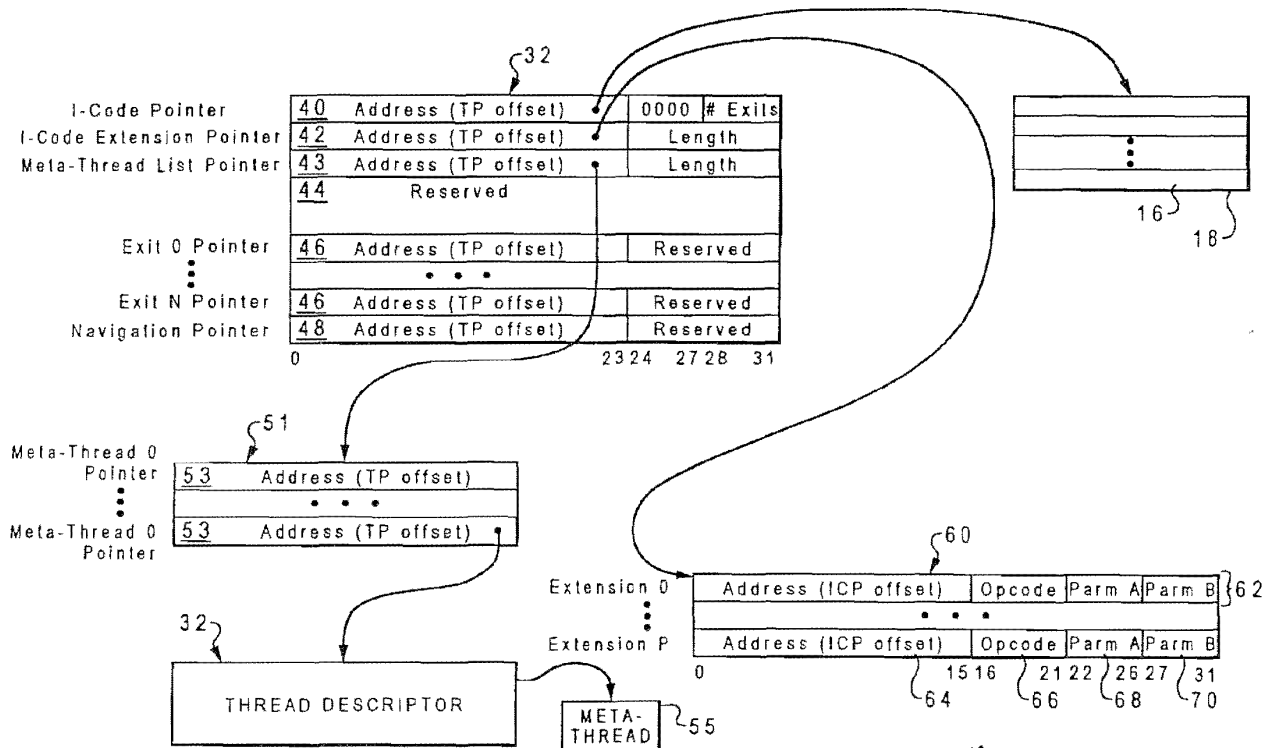


Fig. 14

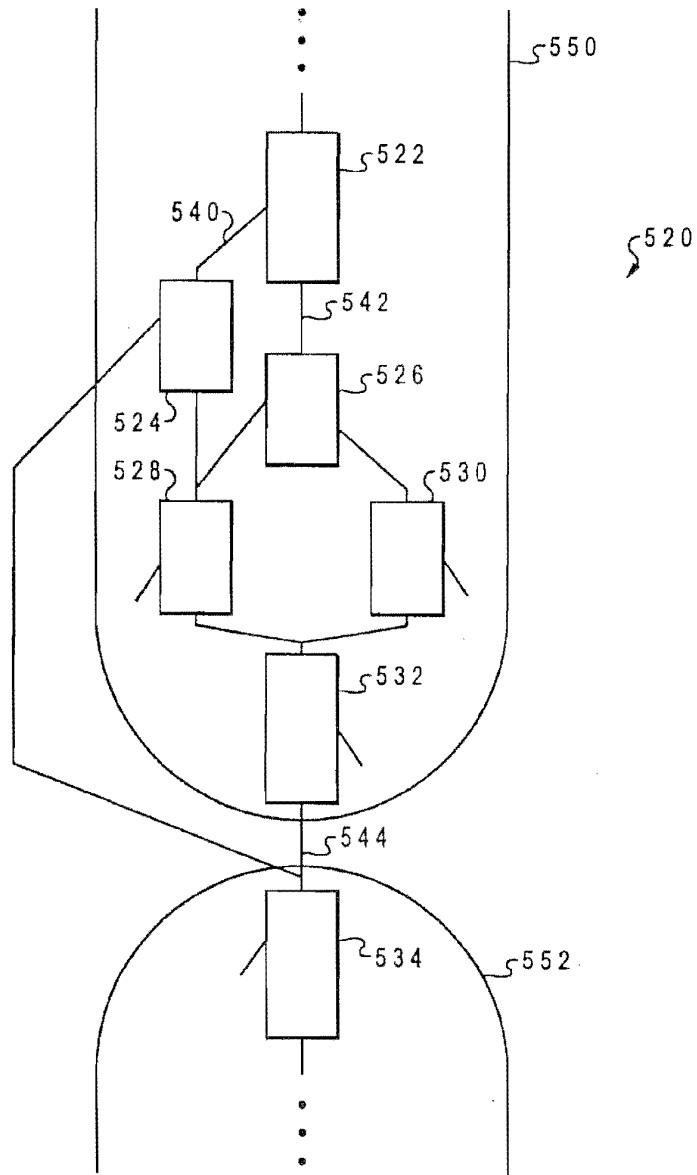
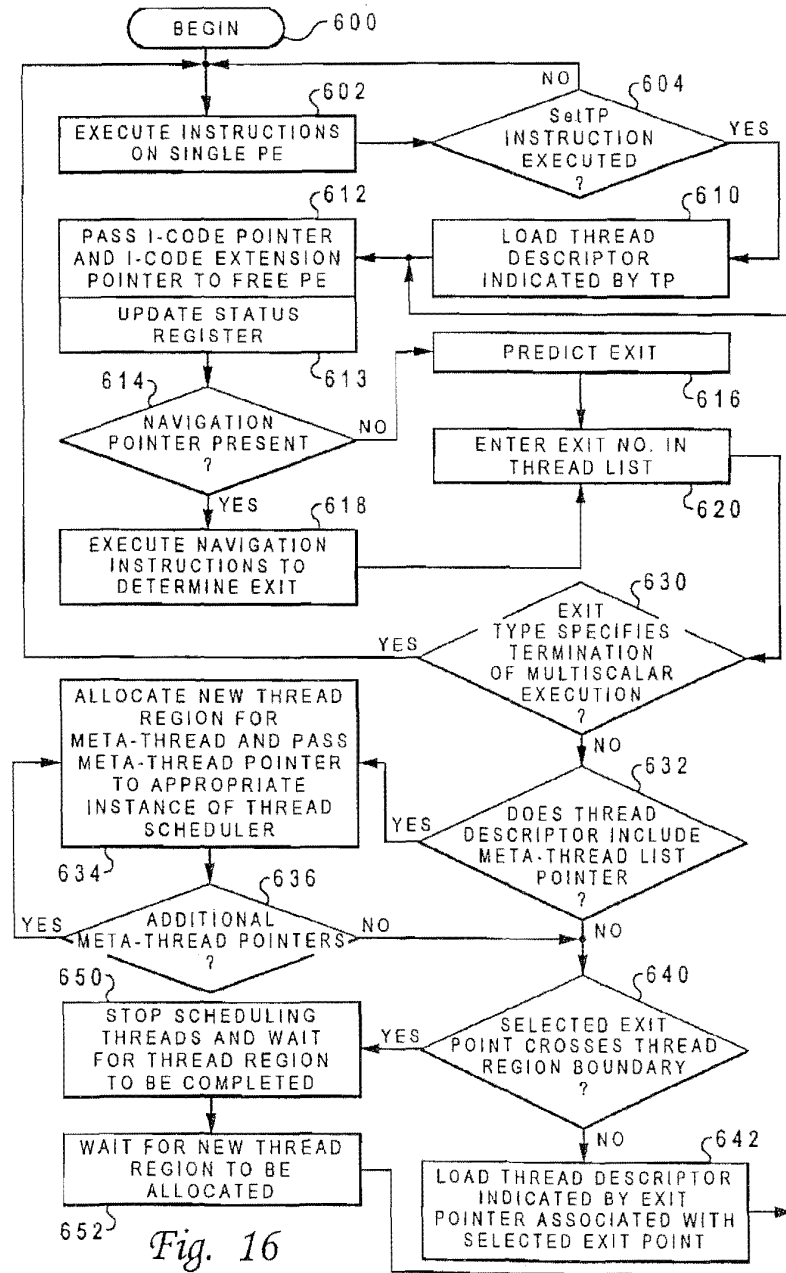


Fig. 15



652 Fig. 16

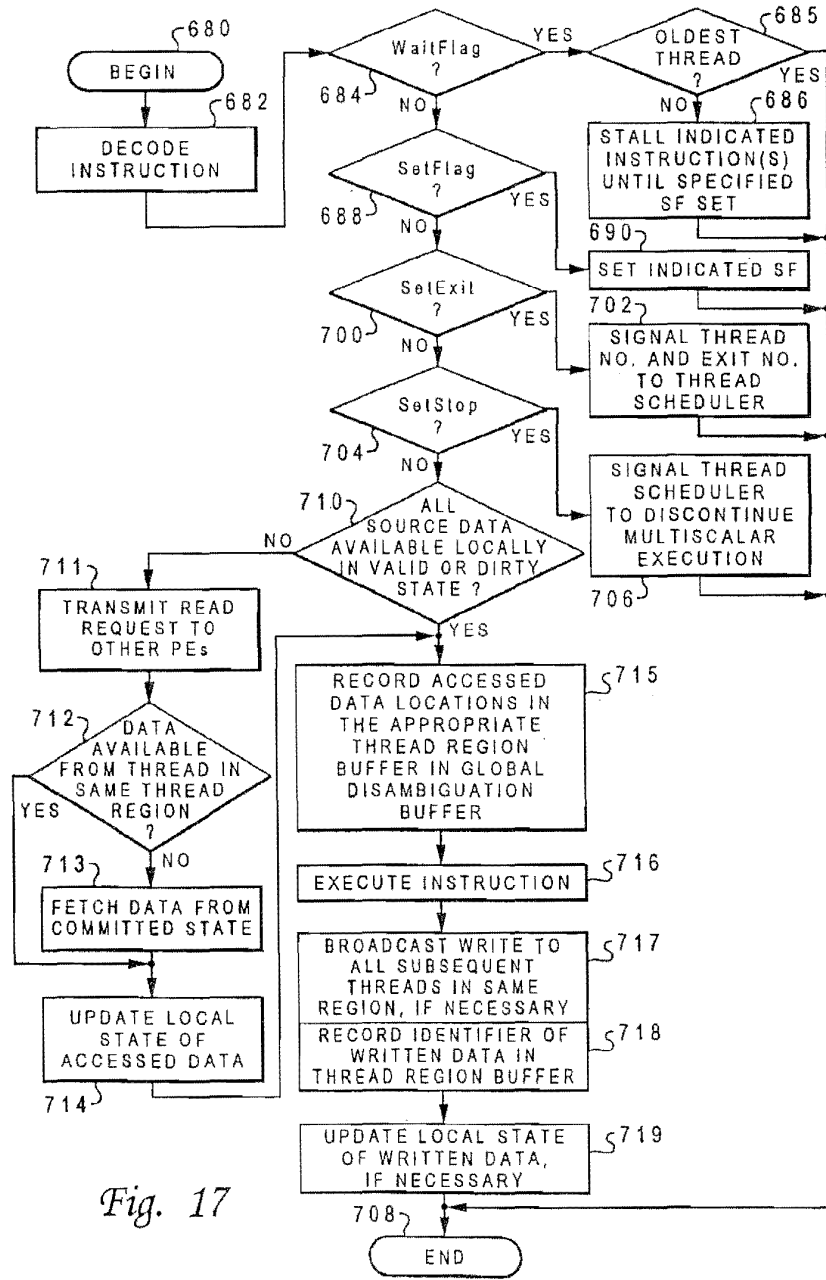


Fig. 17

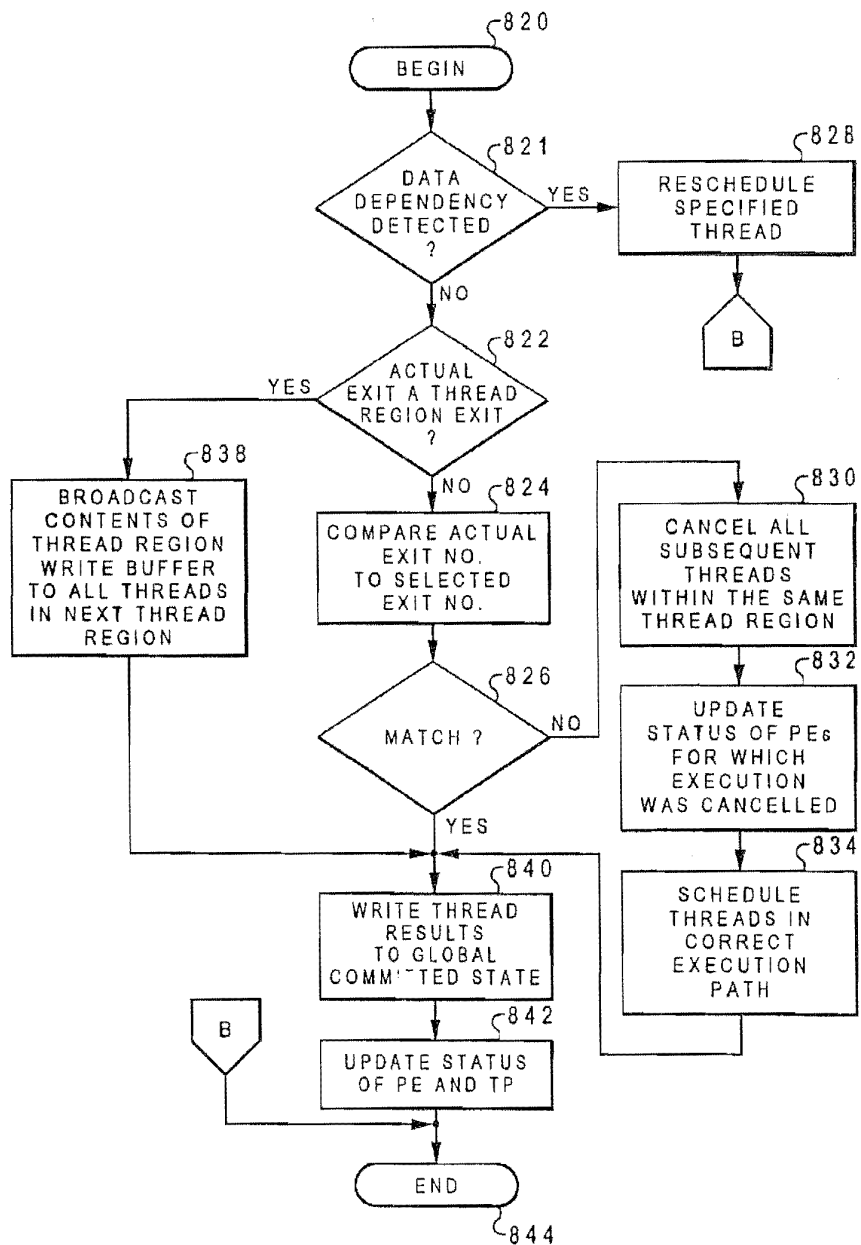


Fig. 18

1

**METHOD AND SYSTEM FOR
CONSTRUCTING A PROGRAM INCLUDING
A NAVIGATION INSTRUCTION**

BACKGROUND

1. Technical Field

The technical field of the present specification relates in general to a method and system for data processing and in particular to a method and system for multiscalar data processing.

2. Cross-Reference to Related Applications

This application is related to the following patent applications, which are incorporated herein by reference:

(1) application Ser. No. 08/767,488 (Attorney Docket No. AT9-96-223), entitled "METHOD AND SYSTEM FOR CONSTRUCTING A MULTISCALAR PROGRAM INCLUDING A PLURALITY OF THREAD DESCRIPTORS THAT EACH REFERENCE A NEXT THREAD DESCRIPTOR TO BE PROCESSED," filed of even date herewith;

(2) application Ser. No. 08/767,488 (Attorney Docket No. AT9-96-187), entitled "PROCESSOR AND METHOD FOR DYNAMICALLY INSERTING AUXILIARY INSTRUCTIONS WITHIN AN INSTRUCTION STREAM DURING EXECUTION," filed of even date herewith;

(3) application Ser. No. 08/767,489 (Attorney Docket No. AT9-96-224), entitled "METHOD AND SYSTEM FOR CONCURRENTLY EXECUTING MULTIPLE THREADS CONTAINING DATA DEPENDENT INSTRUCTIONS," filed of even date herewith;

(4) application Ser. No. 08/767,487 (Attorney Docket No. AT9-96-224), entitled "METHOD AND SYSTEM FOR EXECUTING A PROGRAM WITHIN A MULTISCALAR PROCESSOR BY PROCESSING LINKED THREAD DESCRIPTORS," filed of even date herewith; and

(5) application Ser. No. 08/767,490 (Attorney Docket No. AT9-96-186), entitled "METHOD AND SYSTEM FOR CONSTRUCTING A PROGRAM INCLUDING OUT-OF-ORDER THREADS AND PROCESSOR AND METHOD FOR EXECUTING THREADS OUT-OF-ORDER," filed of even date herewith.

3. Description of the Related Art

In the development of data processing systems, it became apparent that the performance capabilities of a data processing system could be greatly enhanced by permitting multiple instructions to be executed simultaneously. From this realization, several processor paradigms were developed that each permit multiple instructions to be executed concurrently.

A superscalar processor paradigm is one in which a single processor is provided with multiple execution units that are capable of concurrently processing multiple instructions. Thus, a superscalar processor may include an instruction cache for storing instructions, at least one fixed-point unit (FXU) for executing fixed-point instructions, a floating-point unit (FPU) for executing floating-point instructions, a load/store unit (LSU) for executing load and store instructions, a branch processing unit (BPU) for executing branch instructions, and a sequencer that fetches instructions from the instruction cache, examines each instruction individually, and opportunistically dispatches each instruction, possibly out of program order, to the appropriate execution unit for processing. In addition, a superscalar processor typically includes a limited set of architected registers that temporarily store operands and results of

2

processing operations performed by the execution units. Under the control of the sequencer, the architected registers are renamed in order to alleviate data dependencies between instructions.

State-of-the-art superscalar processors afford a performance of between 1 and 2 instructions per cycle (IPC) by, among other things, permitting speculative execution of instructions based upon the dynamic prediction of conditional branch instructions. Because superscalar processors have no advance knowledge of the control flow graph (CFG) (i.e., the control relationships linking basic blocks) of a program prior to execution, IPC performance is necessarily limited by branch prediction accuracy. Thus, increasing the performance of the superscalar paradigm requires not only improving the accuracy of the already highly accurate branch prediction mechanism, but also supporting a broader instruction issue bandwidth, which requires exponentially complex sequencer circuitry to analyze instructions and resolve instruction dependencies and antidependencies. Because of the inherent difficulty in overcoming the performance bottlenecks of the superscalar paradigm, the development of increasingly aggressive and complex superscalar processors has a diminishing rate of return in terms of IPC performance.

An alternative processing paradigm is that provided by parallel and multiprocessing data processing systems, which although having some distinctions between them, share several essential characteristics. Parallel and multiprocessor data processing systems, which each typically comprise multiple identical processors and are therefore collectively referred to hereinafter as multiple processor systems, execute programs out of a shared memory accessible to the processors across a system bus. The shared memory also serves as a global store for processing results and operands, which are managed by a complex synchronization mechanism to ensure that data dependencies and antidependencies between instructions executing on different processors are resolved correctly. Like superscalar processors, multiple processor systems are also subject to a number of performance bottlenecks.

A significant performance bottleneck in multiple processor systems is the latency incurred by the processors in storing results to and retrieving operands from the shared memory across the system bus. Accordingly, in order to minimize latency and thereby obtain efficient operation, compilers for multiple processor systems are required to divide programs into groups of instructions (tasks) between which control and data dependencies are identified and minimized. The tasks are then each assigned to one of the multiple processors for execution. However, this approach to task allocation is not suitable for exploiting the instruction level parallelism (ILP) inherent in many algorithms. A second source of performance degradation in multiple processor systems is the requirement that control dependencies between tasks be resolved prior to the dispatch of subsequent tasks for execution. The failure of multiple processor systems to provide support for speculative task execution can cause processors within the multiple processor systems to incur idle cycles while waiting for inter-task control dependencies to be resolved. Moreover, the development of software for multiple processor systems is complicated by the need to explicitly encode fork information within programs, meaning that multiple processor code cannot be easily ported to systems having diverse architectures.

Recently, a new aggressive "multiscalar" paradigm, comprising both hardware and software elements, was proposed to address and overcome the drawbacks of the conventional

superscalar and multiple processor paradigms described above. In general, the proposed hardware includes a collection of processing units that are each coupled to a sequencer, an interconnect for interprocessor communication, and a single set of registers. According to the proposed multiscalar paradigm, a compiler is provided that analyzes a program in terms of its CFG and partitions a program into multiple tasks, which comprise contiguous regions of the dynamic instruction sequence. In contrast to conventional multiple processor tasks, the tasks created by the multiscalar compiler may or may not exhibit a high degree of control and data independence. Importantly, the compiler encodes the details of the CFG in a task descriptor within the instruction set architecture (ISA) code space in order to permit the sequencer to traverse the CFG of the program and speculatively assign tasks to the processing units for execution without examining the contents of the tasks.

According to the proposed multiscalar paradigm, register dependencies are resolved statically by the compiler, which analyzes each task within a program to determine which register values each task might possibly create during execution. The compiler then specifies the register values that might be created by each task within an associated register reservation mask within the task descriptor. The register reservations seen by a given task are the union of the register reservation masks associated with concurrently executing tasks that precede the given task in program order. During execution of the program, a processing unit executing an instruction dependent upon a register value that might be created by a concurrently executing task stalls until the register value is forwarded or the reservation is released by the preceding task. Upon release of the register or receipt of a forwarded register value by the stalled processing unit, the reservation for the register is cleared within the register reservation mask of the stalled processing unit and the stalled processing unit resumes execution. In order to trigger the forwarding of register values, the compiler adds tag bits to each instruction within a task. The tag bits associated with the last instruction in a task to create a particular register value indicate that the register value is to be forwarded to all concurrently executing tasks subsequent to the task in program order. Release of a register, on the other hand, is indicated by a special release instruction added to the base ISA or created by overloading an existing instruction within the ISA.

In contrast to register dependencies, the proposed multiscalar paradigm does not attempt to statically resolve memory dependencies and permits load and store instructions to be executed speculatively. A dynamic check must then be made to ensure that no preceding task stores to a memory location previously loaded by a subsequent task. If such a dependency violation is detected, the execution of the task containing the speculative load and all subsequent tasks are aborted and appropriate recovery operations are performed. Further details of the proposed multiscalar architecture may be found in G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar Processors," *Proc. ISCA '95 Int'l Symposium on Computer Architecture*, June 1995, pp. 414-425.

The proposed multiscalar paradigm overcomes many of the deficiencies of other paradigms in that the multiscalar paradigm affords a wide instruction window from which instructions can be dispatched utilizing relatively simple scheduling hardware, is less sensitive to inter-task data dependencies and mispredicted branches, and is capable of exploiting the LLP believed to be present in most sequential programs. However, the proposed multiscalar architecture

also has several deficiencies. First, backward compatibility of code binaries is sacrificed due to the insertion of release and other multiscalar instructions into the program to handle task synchronization. Second, multiscalar simulations have shown that the insertion of a large amount of multiscalar instructions that do no useful work into a program can actually degrade multiscalar performance to such an extent that better performance may be obtained with a conventional superscalar processor. Third, the attachment of additional bits to each instruction in the program, which was proposed in order to trigger the forwarding of processing results from a predecessor task to subsequent tasks, necessitates an increased instruction path width and additional hardware complexity. Fourth, the proposed multiscalar paradigm has no mechanism for handling dependencies between loads and stores to memory. Fifth, in the proposed multiscalar architecture, all tasks except the oldest are executed speculatively, meaning that even if task prediction accuracy is 90%, the prediction accuracy for tasks beyond the fifth task drops below 60%.

As should thus be apparent, it would be desirable to provide an enhanced multiscalar architecture that overcomes the foregoing and other deficiencies of the proposed multiscalar processor paradigm.

SUMMARY

It is therefore one object of the present disclosure to provide an improved method and system for data processing.

It is another object of the present disclosure to provide an improved method and system for multiscalar data processing.

The foregoing objects are achieved as is now described. A method and system are provided for constructing a program executable by a processor including one or more processing elements for executing threads and a thread scheduler for assigning threads to the processing elements for execution. According to the method, a plurality of threads are provided that each include at least one control flow instruction. From one or more control flow instructions within the plurality of threads, a condition upon which execution of a particular thread depends is determined. In response to the determination, at least one navigation instruction executable by the thread scheduler is created that indicates that the particular thread is to be assigned to one of the processing elements for execution in response to the condition.

The above as well as additional objects, features, and advantages of an illustrative embodiment will become apparent in the following detailed written description.

BRIEF DESCRIPTION OF THE DRAWINGS

The novel features believed characteristic of the invention are set forth in the appended claims. The invention itself however, as well as a preferred mode of use, further objects and advantages thereof, will best be understood by reference to the following detailed description of an illustrative embodiment when read in conjunction with the accompanying drawings, wherein:

FIG. 1 illustrates a conceptual diagram of a process for constructing a multiscalar program, wherein the multiscalar program includes separate Instruction Code (I-Code) and Thread Code (T-Code) streams;

FIG. 1B depicts a high level logical flowchart of an illustrative embodiment of the process by which a multiscalar compiler builds the T-Code stream of the multiscalar program;

FIG. 2 depicts an illustrative embodiment of a thread descriptor within the T-Code stream depicted in FIG. 1;

FIG. 3 illustrates an exemplary multiscalar program fragment that includes possibly dependent instruction set architecture (ISA) instructions synchronized by SetFlag and WaitFlag extension instructions, wherein the program fragment further includes an inter-thread control dependency that may be resolved by executing a set of T-Code navigation instructions created by the multiscalar compiler;

FIG. 4 is a block diagram depiction of an illustrative embodiment of a multiscalar data processing system;

FIG. 5 illustrates a more detailed depiction of the global synchronization flags (SFs) illustrated in FIG. 4;

FIG. 6 depicts a timing diagram of the pipelined processing of the threads of a multiscalar program, wherein the thread pipeline includes thread scheduling, thread execution, and thread completion stages;

FIG. 7 is a high level logical flowchart of a method of thread scheduling when threads are processed according to logical program order;

FIG. 8 is a high level logical flowchart of a method for fetching and dispatching instructions within a processing element, which illustrates the dynamic insertion of extension instructions into the instruction stream of the processing element;

FIG. 9 is a high level logical flowchart depicting a method of executing instructions within a processing element when threads are processed in logical program order;

FIG. 10 is a high level logical flowchart illustrating a method of completing threads when threads are processed in logical program order;

FIG. 11 illustrates the execution of the Thread Code (T-Code) and Instruction Code (I-Code) streams comprising a multiscalar program, wherein multiscalar execution of the multiscalar program is initiated by a SetTP instruction embedded within the I-Code stream;

FIG. 12 depicts a state diagram of the protocol utilized by the processing elements (PEs) within the multiscalar processor illustrated in FIG. 4 to maintain local register and memory data coherency in response to local events;

FIG. 13 illustrates a state diagram of the snooping protocol utilized by the PEs within the multiscalar processor depicted in FIG. 4 to maintain local register and memory data coherency in response to external events;

FIG. 14 depicts an illustrative embodiment of a T-Code thread descriptor utilized to support out-of-order execution of threads;

FIG. 15 illustrates the partitioning of threads within a multiscalar program into multiple thread regions;

FIG. 16 is a high level logical flowchart depicting a method of scheduling threads for out-of-order execution;

FIG. 17 is a high level logical flowchart illustrating a method of executing instructions within a processing element when threads are processed out-of-order; and

FIG. 18 is a high level logical flowchart depicting a method of completing threads when threads are processed out-of-order.

DETAILED DESCRIPTION

The multiscalar processing paradigm disclosed herein overcomes numerous deficiencies of the previously proposed multiscalar paradigm through improvements to both the multiscalar hardware and software architectures. In order to facilitate an understanding of the operation of the multi-

scalar processor hardware, an introduction to the improved multiscalar software architecture will first be given. Software Architecture

With reference now to the figures and in particular with reference to FIG. 1A, there is a conceptual diagram of a process for constructing a multiscalar program is illustrated. As depicted, an ordinary high level language (e.g., C++) program 10 containing a number of high level instructions 12 is input into multiscalar compiler 14 for processing.

During a first pass, multiscalar compiler 14 translates each of high level instructions 12 into one or more executable instruction set architecture (ISA) instructions 16 arranged in a particular program order. In addition, multiscalar compiler 14 partitions ISA instructions 16 into one or more threads 18, which each contain a logically contiguous group of ISA instructions 16. As utilized hereinafter, the term thread refers to a set of one or more logically contiguous instructions within a multiscalar program that have a single entry point and multiple possible exit points. In other words, when a thread is executed, the first instruction within the thread is always executed, but there are multiple possible execution paths out of the thread. Importantly, the multiscalar software architecture disclosed herein permits each ISA instruction 16 to be included within more than one thread 18 and does not utilize the explicit programmed forks required by conventional multiple processor software architectures. Threads 18 can be distinguished from basic blocks 20 in that basic blocks 20 are sets of sequential ISA instructions terminated by a branch instruction. Basic blocks 20 have only two exit points, but may have two or more entry points. The set of threads 18 produced by the first pass of multiscalar compiler 14 forms Instruction Code (I-Code) stream 22.

Because threads 18 are not necessarily substantially data and control independent (in contrast to those processed in parallel and multiprocessor systems), information describing the CFG of program 10 and inter-thread data dependencies must be made available to a multiscalar processor during execution in order to permit concurrent execution of multiple threads. Accordingly, during a second pass multiscalar compiler 14 generates a Thread Code (T-Code) stream 30 including a number of thread descriptors 32 that are each associated with a respective one of threads 18. Each thread descriptor 32 provides the information needed to support multiscalar thread scheduling, thread prediction, and thread synchronization, including (as depicted in FIG. 1) pointers to both the corresponding thread 18 and subsequent thread descriptors 32. I-Code stream 22 and T-Code stream 30 together comprise a multiscalar program 34 executable by the multiscalar data processing system described below with reference to FIG. 4.

With reference now to FIG. 2, there is depicted a more detailed diagram of an illustrative embodiment of a thread descriptor 32 associated with a thread 18. As illustrated, thread descriptor 32 is a data structure containing a number of 32-bit entries. The first 32-bit entry contains a 24-bit I-Code pointer 40 that indicates the address of the first ISA instruction 16 within thread 18 relative to the address indicated by a hardware-maintained thread pointer (TP). As described above, the ISA instruction 16 pointed to by I-Code pointer 40 will be the first instruction executed within thread 18. The first 32-bit entry also includes 4 bits that indicate the number of possible exit points within the associated thread 18.

As illustrated, thread descriptor 32 also includes at least two 32-bit entries that each contain a 24-bit exit pointer 46. Each exit pointer 46 is associated with a possible exit point of thread 18 and indicates a TP-relative address of a thread

descriptor 32 associated with the next thread 18 to be executed if the associated exit point of the current thread 18 is taken or predicted as taken. The 32-bit entries containing exit pointers 46 also include an 8-bit reserved section that may be subsequently defined to provide further exit information. Future improvements to the multiscalar architecture disclosed herein may also be supported by defining the reserved 32-bit entries indicated at reference numeral 44.

Thread descriptor 32 further contains a 24-bit I-Code Extension pointer 42 that points to an extension list 60 containing auxiliary extension instructions that are to be dynamically inserted into thread 18 by the multiscalar processor hardware during execution. The length of (i.e., number of entries within) extension list 60 is specified by the final 8 bits of the 32-bit entry. Referring now to extension list 60, each of extension list entries 62 contains a 26-bit address identifier 64 that indicates, relative to I-Code pointer 40, the address of an ISA instruction 16 within thread 18. The indicated instruction address specifies the location within thread 18 at which the extension instruction defined by 6-bit opcode 66 is to be dynamically inserted. Finally, each extension list entry 62 can optionally include parameters 68 and 70. Depending upon the type of extension instruction defined by opcode 66, parameters 68 and 70 can be utilized to indicate whether the extension instruction is to be executed prior to, subsequent to, or in conjunction with the ISA instruction 16 indicated by address identifier 64. As will be appreciated by those skilled in the art, multiple extension instructions may be associated with a single ISA instruction address.

Following is a description of a number of instruction extensions that can be inserted into extension lists 60 by multiscalar compiler 14 in order to support thread scheduling, thread prediction, and thread synchronization:

- SetExit: Marks a possible exit point of a thread;
- SetStop: Marks a possible exit point at which multiscalar execution terminates if the possible exit point is taken;
- SetFlag: Sets a specified hardware-maintained synchronization flag (SF) to indicate that register or memory data is available for use by subsequent threads;
- WaitFlag: Delays execution of one or more specified instructions within a thread until a specified SF is set; and
- ChainFlag: Sets a second SF in response to a first SF being set.

In order to minimize penalties attributable to inter-thread data hazards, multiscalar compiler 14 utilizes SetFlag and WaitFlag extension instructions to resolve every inter-thread register data dependency (although hardware support is also available as discussed below with reference to FIG. 4). Accordingly, multiscalar compiler 14 preferably creates a SetFlag extension instruction in the extension list 60 of the thread that produces a data value and creates a WaitFlag extension instruction in the extension list 60 of the thread that consumes the data value. In addition, if the execution path between two threads is not control-independent, multiscalar compiler 14 creates SetFlag extension instructions within the alternative execution path(s) in order to ensure that the consuming thread can proceed as soon as the data dependency (or possible data dependency) is resolved.

For example, referring to FIG. 3, there is illustrated a fragment of a multiscalar program for which multiscalar compiler 14 will create SetFlag and WaitFlag extension instructions. As depicted, thread C contains ISA instruction 86, which specifies that the sum of registers GPR1 and GPR2 is to be calculated and stored within GPR3. Thread F contains ISA instruction 88, which specifies that the sum of GPR3 and GPR4 is to be calculated and stored within GPR1.

Thus, in the present example, thread C is a producer of the value of GPR3 and thread F is a consumer of the value of GPR3. During compilation of multiscalar program 80, multiscalar compiler 14 inserts a WaitFlag extension instruction in extension list 60 of thread F that is associated with the instruction address of ISA instruction 88. The WaitFlag extension instruction specifies that it is to be inserted into thread F prior to ISA instruction 88 so that execution of ISA instruction 88 (and possibly other instructions within thread F) is stalled until a specified SF is set. In addition, multiscalar compiler 14 inserts a SetFlag extension instruction in extension list 60 of thread C that is associated with the instruction address of ISA instruction 86. The SetFlag extension instruction specifies that it is to be inserted into thread C following ISA instruction 86. Furthermore, multiscalar compiler 14 inserts a SetFlag extension instruction into extension list 60 of thread E so that, if control passes from thread B to thread E to thread F during execution, the execution of thread F is not unnecessarily stalled by the WaitFlag extension instruction.

In contrast to possible register data dependencies, which are always detected and synchronized utilizing SetFlag and WaitFlag extension instructions, multiscalar compiler 14 only utilizes the SetFlag and WaitFlag extension instructions to synchronize disambiguable memory data accesses (i.e., memory data accesses known to be dependent because the target addresses can be statically determined). Other memory data accesses are assumed to be independent by multiscalar compiler 14 and are monitored by the multiscalar processor hardware described below in order to prevent data inconsistencies.

Referring again to FIG. 2, thread descriptor 32 may optionally include an entry containing a 24-bit navigation pointer 48 that points to a set of navigation instructions 50. In accordance with the illustrative embodiment of a multiscalar data processing system described below with reference to FIG. 4, navigation instructions 50 may be utilized by the multiscalar processor's thread scheduling hardware to traverse the CFG of I-Code stream 22 in a non-speculative fashion.

With reference again to FIG. 3, multiscalar program 80 also illustrates a scenario in which multiscalar compiler 14 may create a set of navigation instructions 50 in order to facilitate non-speculative thread scheduling. As depicted, thread A of multiscalar program 80 contains ISA instruction 82, which sets a variable X to a particular value. Thread B contains ISA instruction 84, which causes control to pass to thread E if X has a value greater than or equal to 0 and to pass to thread C if X has a value less than 0. If multiscalar program 80 were executed in the previously proposed multiscalar processor, the sequencer hardware would simply predict one of the exits of thread B and speculatively assign the indicated one of threads C and E to a processing element prior to the execution of ISA instruction 84. In contrast, according to the multiscalar paradigm disclosed herein, multiscalar compiler 14 identifies ISA instruction 82 as a condition setting instruction and ISA instruction 84 as an inter-thread control flow instruction that depends upon the condition set by ISA instruction 82. Multiscalar compiler 14 then inserts a navigation pointer 48 into thread B's thread descriptor 32 that points to a set of navigation instructions 50 also created by multiscalar compiler 14. The set of navigation instructions 50 created by multiscalar compiler 14 for thread B may be expressed as follows:

```

if x < 0
  fork C
else
  fork D
endif;

```

By making these navigation instructions available to the thread scheduler hardware at runtime through navigation pointer 48, the thread scheduler can schedule one of threads C and E to a processing element for non-speculative execution. Thus, in this instance, the penalty for exit misprediction is totally eliminated. Multiscalar compiler 14 can also provide such control flow information for other types of inter-thread control flow instructions, including if-then-else and loop constructs. Importantly, the navigation instructions 50 generated by multiscalar compiler 14 can alternatively be accessed by an extension pointer 64 within extension list 60. Furthermore, navigation instructions 50 can be executed within a processing element of the multiscalar processor on behalf of the thread scheduler.

With reference now to FIG. 1B, there is depicted a high level logical flowchart that summarizes the method by which multiscalar compiler 14 constructs T-Code stream 30 in an illustrative embodiment. As illustrated, the process begins at block 90 in response to multiscalar compiler 14 translating high level instructions 12 into ISA instructions 16 and partitioning ISA instructions 16 into one or more threads 18, which as described above each include a single entry point and a plurality of possible exit points. The process then proceeds to block 91, which depicts multiscalar compiler 14 creating an empty thread descriptor 32 associated with each thread 18. The process proceeds from block 91 to block 92, which depicts multiscalar compiler 14 identifying the next thread to be executed in program order following each possible exit point of threads 18. Multiscalar compiler 14 utilizes the exit information to insert appropriate exit pointers and exit counts within thread descriptors 32. Next, the process passes to block 93, which illustrates multiscalar compiler 14 identifying inter-thread data dependencies by analyzing the register IDs and memory addresses accessed by ISA instructions 16. As depicted at block 94, multiscalar compiler 14 utilizes the exit information ascertained at block 92 and the data dependency information collected at block 93 to create an extension list 60 associated with each respective thread 18. As described above, extension lists 60 contain the extension instructions utilized by the multiscalar processor hardware to resolve identified inter-thread data dependencies and to identify possible exit points of threads. Multiscalar compiler 14 also creates an I-Code extension pointer 42 within each thread descriptor 32 that references the associated extension list 60. The process then proceeds from block 94 to block 95, which illustrates multiscalar compiler 14 analyzing the control flow instruction(s) adjacent to each thread boundary to determine if the conditions upon which the control flow instructions depend can be resolved prior to prediction of an exit point of the threads. As described above with reference to FIG. 3, in response to detection of a control flow condition that can be resolved prior to exit prediction, multiscalar compiler 14 creates a set of navigation instructions 50 executable by or on behalf of the thread scheduler and inserts a navigation pointer 48 within the thread descriptor 32. The process proceeds from block 95 to optional block 96, which is described below with reference to FIG. 14, and thereafter terminates at block 97.

Referring again to FIG. 2, in order to permit selective multiscalar execution of multiscalar program 34, I-Code

stream 22 preferably includes at least one SetTP instruction near the beginning that triggers concurrent execution of threads 18 by initializing the value of the hardware TP. In order to maintain software compatibility with prior processor paradigms, the SetTP instruction preferably overloads a seldom used instruction within the ISA, such as an alternative form of a noop or branch instruction. I-Code stream 22 preferably also includes SetTP instructions at locations scattered throughout I-Code stream 22. The additional SetTP instructions permit concurrent execution of threads 18 to be resumed following an exception or other interruption of multiscalar execution and are ignored by hardware if threads 18 are being executed concurrently.

Having provided an overview of an illustrative embodiment of the improved multiscalar software architecture, the hardware architecture will now be described.

Hardware Architecture

Referring now to FIG. 4, there is depicted an illustrative embodiment of a multiscalar data processing system. As illustrated, the multiscalar data processing system includes a multiscalar processor 100, which is coupled to system memory 112 and other unillustrated components of the multiscalar data processing system via system bus 114. As depicted, multiscalar processor 100 includes processor interface circuitry 120, which comprises the latches and support circuitry necessary to communicate data and instructions between system bus 114 and unified level two (L2) cache 122. As a unified cache, L2 cache 122 stores a copy of a subset of both the data and instructions residing in system memory 112 for use by multiscalar processor 100 during execution. Coherency between the data stored within L2 cache 122 and system memory 112 is maintained utilizing a conventional cache coherency protocol. Multiscalar processor 100 further includes architected register file 124, which in addition to providing register storage for data and condition information, includes instruction pointer (IP) 126, which indicates the instruction address at which multiscalar processor 100 is currently executing non-speculatively. As described in greater detail below, multiscalar processor 100 is capable of executing multiple threads concurrently, only one of which is typically executing non-speculatively. Thus, IP 126 marks the current point of execution in this non-speculative thread. In contrast to information maintained within the execution circuitry of multiscalar processor 100, information within architected register file 124, L2 cache 122, and processor interface circuitry 120 is in a committed state, meaning that this information constitutes a non-speculative, consistent machine state to which multiscalar processor 100 can return upon interruption.

Still referring to FIG. 4, the execution circuitry of multiscalar processor 100 includes thread scheduler 130 and a scalable number of identical processing elements (PEs), which in the illustrative embodiment include PEs 132, 134, 136, and 138. In accordance with the multiscalar software architecture described above, thread scheduler 130 processes thread descriptors within the T-Code stream of a multiscalar program in order to assign multiple threads to PEs 132-138 for concurrent execution. In order to reduce access latency, thread scheduler 130 is equipped with a T-Code cache 44 that stores the thread descriptors, thereby establishing separate fetch paths for the I-Code and T-Code streams. As noted above, ordinarily only one of PEs 132-138 executes non-speculatively at a time. The non-speculative thread, which is the earliest occurring thread in program order among the executing threads (and the thread that contains the instruction to which IP 126 points), is indicated by thread pointer (TP) 142 maintained by thread scheduler 130.

Thread scheduler 130 also includes exit prediction mechanism 140, which is utilized by thread scheduler 130 to predict exits of threads. In a first embodiment of multiscalar processor 100, exit prediction mechanism 140 comprises a static prediction mechanism that predicts one of the possible exits of a thread based upon information supplied by multiscalar compiler 14. For example, multiscalar compiler 14 could be constrained to list the statically predicted exit within the thread descriptor as Exit 0, thereby indicating to exit prediction mechanism 140 that this exit should be selected. Exit prediction mechanism 140 can alternatively be implemented as a history-based dynamic prediction mechanism like that utilized in a superscalar processor to predict branch resolutions.

As illustrated, thread scheduler 130 further includes a thread list (TL) 146 that records, in association with an arbitrary thread number, the exit number of each exit selected by thread scheduler 130. The thread number is utilized to identify the thread containing the selected exit in communication between thread scheduler 130 and PEs 132-138. In the illustrative embodiment, thread scheduler 130 tracks which of PEs 132-138 is (are) free utilizing a 4-bit status register 148 in which the state of each bit indicates whether a corresponding one of PEs 132-138 is free or busy. Status register 148 is updated each time a thread is scheduled to or completed by one of PEs 132-138.

Referring to PEs 132-138, the central component of each of PEs 132-138 is an execution core 158 that executes instructions contained within an assigned thread. In a preferred embodiment, execution core 158 contains superscalar circuitry that supports intra-thread branch speculation and includes multiple execution units capable of executing multiple ISA instructions out-of-order during each cycle. However, based upon design and cost considerations, execution core 158 of PEs 132-138 can alternatively employ any one of a number of diverse hardware architectures. For example, execution core 158 may comprise a single execution resource that executes ISA instructions sequentially. Regardless of which hardware architecture is utilized to implement execution core 158, each execution core 158 includes an instruction sequencer that fetches and dispatches instructions and at least one execution resource that executes instructions.

Local storage is provided to each execution core 158 by an associated instruction cache 150, data cache 156, and GPR cache 154, which respectively store the ISA instructions, memory data values, and data and condition register values required by the associated execution core 158 during execution. Each execution core 158 is also coupled to CAM 160 that stores the extension list associated with the thread executing within the associated execution core 158. Extension instructions in the extension list are dynamically inserted into the thread executed by the associated execution core 158 in accordance with the method described below with respect to FIG. 8.

Each of PEs 132-138 further includes communication and synchronization logic 152, which is coupled to both GPR cache 154 and data cache 156. Communication and synchronization logic 152 maintains register and memory data coherency (i.e., the availability of data to the associated PE) through inter-PE and PE-L2 communication across local communication and synchronization mechanism 170, which, in order to reduce latency, preferably includes four concurrent address busses for register communication and at least one address bus for memory communication. Communication across local communication and synchronization mechanism 170 is performed under the arbitrating control of

arbitration logic 172. Further details of local communication and synchronization mechanism 170 may be found in J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach," second ed., Morgan Kaufmann Publishers, Inc., pp. 655-693, which is incorporated herein by reference. The inter-PE and PE-L2 communication conducted by communication and synchronization logic 152 is governed by the data coherency protocol depicted in FIGS. 12 and 13.

Referring now to FIGS. 12 and 13, two state diagrams are shown that together illustrate the data coherency protocol implemented by multiscalar processor 100 for both register and memory data. For clarity, FIG. 12 shows the portion of the data coherency protocol relating to local (intra-PE) events, while FIG. 13 shows the portion of the data coherency protocol relating to external (inter-PE) events received from local communication and synchronization mechanism 170. Because the data coherency protocol includes five states, the state of each data word in data cache 156 and each register within GPR cache 154 is preferably tracked utilizing three status bits. Those skilled in the art will appreciate from the following description that the data coherency protocol could alternatively be implemented within multiscalar processor 100 utilizing a directory-based coherency mechanism.

With reference first to FIG. 12, when execution of a multiscalar program begins, all data locations within GPR cache 154 and data cache 156 of each of PEs 132-138 are initially in invalid state 500. In response to receipt of an instruction within a thread, an execution core 158 within a PE requests data required for execution of the instruction from its local GPR cache 154 or data cache 156. If the data location associated with the requested data is in invalid state 500, meaning that the requested data is not present locally, communication and synchronization logic 152 broadcasts a read request indicating the register number or memory address of the required data on local communication and synchronization mechanism 170, which is snooped by each of PEs 132-138. As depicted in FIG. 13, the communication and synchronization logic 152 within PEs that have the requested register or memory data in any of valid state 502, dirty state 504, valid hazard state 506, or dirty hazard state 508 responds to the read request by indicating ownership of the requested data. PEs for which the requested data is in invalid state 500 do not respond. Based upon thread issue order information obtained from thread scheduler 130, arbitration logic 172 signals the responding PE executing the nearest preceding thread in program order to place the requested data on local communication and synchronization mechanism 170. However, if no PEs respond to the read request broadcast on local communication and synchronization mechanism 170, the communication and synchronization logic 152 within the requesting PE retrieves the required register or memory data from architected register file 124 or L2 cache 122, respectively. Referring again to FIG. 12, once the requested data is read into GPR cache 154 or data cache 156 of the requesting PE, communication and synchronization logic 152 updates the state of the data location from invalid state 500 to valid state 502. Data in valid state 502 is "owned" by the PE and hence can be utilized as an operand for subsequent instructions.

As depicted, communication and synchronization logic 152 updates a register or memory data location in invalid state 500 or valid state 502 to dirty (modified) state 504 in response to the local execution of a store or other instruction that writes data to the data location. A register or memory location in dirty state 504 does not change state in response

to a local execution of an instruction that writes to the data location. Dirty state 504 is similar to valid state 506 in that data locations in dirty state 504 are also owned a PE and thus can be utilized as a source of operands for subsequent instructions. However, in contrast to data locations in valid state 502, data locations in dirty state 504 are written back to architected register file 124 and L2 cache 122 (i.e., the committed state) by communication and synchronization logic 152 in response to a receipt of a writeback signal during thread completion in order to update modified data locations. Importantly, following thread completion, data locations in valid state 502 do not undergo a state transition, leaving GPR cache 154 and data cache 156 "primed" with valid data that can be accessed by a subsequent thread executed locally or within another PE.

Referring again to FIG. 13, the data coherency protocol utilizes valid hazard state 506 and dirty hazard state 508 to mark data locations that have been written by PEs executing future threads in logical program order. Thus, communication and synchronization logic 152 updates a data location in valid state 502 to valid hazard state 506 and updates a data location in dirty state 504 to dirty hazard state 508 in response to receipt of a write request from a PE executing a future thread. The semantics of valid hazard state 506 and dirty hazard state 508 in response to both local and external events are the same as those of valid state 502 and dirty state 504, respectively, except in response to a writeback signal. Because valid hazard state 506 marks locally unmodified data locations that have been written by future threads (and therefore may not be valid after execution of the current thread), data locations in valid hazard state 506 are updated to invalid state 500 in response to receipt of a writeback signal by communication and synchronization logic 152. Similarly, data locations in dirty hazard state 508 are updated to invalid state 500 after the contents of the data locations are written back to architected register file 124 or L2 cache 122.

Still referring to FIG. 13, communication and synchronization logic 152 updates the state of all local data locations to invalid state 500 in response to the receipt of a reset signal generated in response to the occurrence of an exception or the detection of a data or control hazard. As discussed above, setting the state of all local data locations to invalid state 500 discards all of the data within GPR cache 154 and data cache 156.

With reference again to FIG. 4, multiscalar processor 100 further includes a global disambiguation buffer 182 coupled to PEs 132-138 that verifies inter-thread data consistency, that is, that the execution of a multiscalar program obtains the same results as those obtained under sequential, scalar execution.

In the illustrative embodiment of multiscalar processor 100, memory data inconsistencies can occur because execution cores 158 queue store instructions and preferentially perform load instructions such that memory data latency is minimized. This practice, which tacitly assumes that memory accesses are data independent, can lead to data inconsistency if memory accesses are, in fact, dependent between threads. In order to detect an inter-thread memory data inconsistency, global disambiguation buffer 182 stores the target addresses and thread numbers of load instructions and the target addresses and thread numbers of store instructions such that the relative execution order of the load and store instructions is retained. Global disambiguation buffer 182 then compares the target address of each store instruction executed by PEs 132-138 with the buffered load addresses. If a target address match is found and (1) the

thread number of the load instruction follows the thread number of the store instruction in logical program order, and (2) there is no intervening store to the target address within the thread containing the load instruction, thereby indicating that the load instruction was dependent upon a store instruction, global disambiguation buffer 182 signals that a data inconsistency (hazard) has been detected by generating a cancellation signal. In response to a cancellation signal generated by global disambiguation buffer 182, all threads subsequent to the thread containing the load instruction are cancelled and the thread containing the load instruction is reexecuted utilizing the correct memory data.

The cancellation of threads in response to the detection of a data inconsistency can be handled in at least two ways, depending upon design considerations. In a first embodiment, the cancellation signal sets a consistency bit within thread scheduler 130 that is associated with the PE executing the thread that loaded the inconsistent data. As discussed below with reference to FIG. 10, the consistency bit is subsequently processed during the completion of the thread that loaded the inconsistent data. This approach has the advantage of requiring that the consistency bit be checked only a single time during thread processing. However, if data inconsistencies occur relatively frequently or early in the execution of a thread, this approach permits a large amount of useless work to be performed prior to thread cancellation. Alternatively, in a second embodiment, the cancellation signal generated by global disambiguation buffer 182 can set a bit within the PE executing the thread that loaded the inconsistent data. Although this embodiment requires each of PEs 132-138 to check its consistency bit during each cycle, thereby increasing latency, the second embodiment has the advantage of detecting and correcting for data inconsistencies as early as possible, so that the number of processor cycles consumed by useless work is minimized.

In order to correct for possible errors by multiscalar compiler 14 in identifying inter-thread register dependencies with SetFlag/WaitFlag extension instructions or in order to permit multiscalar compiler 14 to insert SetFlag/WaitFlag extension instruction in only the statistically most likely execution paths, global disambiguation buffer 182 preferably further include facilities that ensure inter-thread register data consistency. Similar to the facilities that handle memory data accesses, the register data facilities store the register number and thread number of instructions that read and write register data in a manner that preserves the relative execution order of the "read" and "write" instructions. Global disambiguation buffer 182 then compares the register number into which data is written by an instruction with all of the numbers of registers previously read by threads subsequent in program order to the thread containing the "write" instruction. If the comparison reveals that a "write" instruction in an earlier thread was executed subsequent to a "read" instruction that referenced the same register and the thread containing the "read" instruction does not include an intervening "write" to the same register, global disambiguation buffer 182 signals that a data inconsistency has occurred so that appropriate corrective action can be taken in the manner discussed above with respect to the detection of a memory data inconsistency.

Multiscalar processor 100 finally includes global synchronization flags (SFs) 180, which comprise a shared resource utilized by PEs 132-138 to provide inter-thread data consistency support for register and disambiguable memory accesses. Although not required for data correctness, which is guaranteed by global disambiguation buffer 182, the data

consistency support provided by global SFs 180 improves processor performance by inhibiting data speculation for identified dependencies, thereby avoiding the performance penalty incurred by misspeculation.

With reference now to FIG. 5, there is illustrated a more detailed representation of global SFs 180, which include 32 1-bit flags that are assigned to threads during compilation by multiscalar compiler 14 in order to ensure inter-thread data consistency for register and disambiguable memory accesses. A SF is cleared (set to logical zero) when the thread to which the SF is assigned is scheduled by thread scheduler 130 to one of PEs 132-138 for execution. The SF is set to logical one in response to an occurrence of a synchronization event, such as the execution of a SetFlag extension instruction in response to the production of a data value. Setting the SF notifies subsequent threads stalled by a WaitFlag extension instruction that computation dependent upon the occurrence of the synchronization event can then be performed. Importantly, the oldest (non-speculative) thread ignores all WaitFlag extension instructions since inter-thread data consistency for register and disambiguable memory accesses is guaranteed.

Multiscalar Operation

Referring now to FIG. 6, there is depicted a conceptual timing diagram of the pipelined processing of threads by multiscalar processor 100. As illustrated, the processing of threads by processor 100 is divided into thread scheduling, thread execution, and thread completion stages. During multiscalar execution, stages in the processing of a thread are overlapped with the same and different stages in the processing of other threads in order to mask the effects of latency.

During the thread scheduling stage of thread processing, the thread is assigned by thread scheduler 130 to one of PEs 132-138 for execution. As discussed above and as is described below in greater detail with reference to FIG. 7, once thread scheduler 130 has selected an exit point of a scheduled thread by prediction or execution of navigation code, thread scheduler 130 assigns the thread indicated by the selected exit point to one of PEs 132-138 for execution.

During the thread execution stage, a PE executes an assigned thread. It is during the execution stage that a PE communicates with PEs executing preceding threads in order to request required register or memory data. As described below with reference to FIG. 8, it is also during the thread execution stage that extension instructions are dynamically inserted into the execution stream of a PE. If execution of a thread confirms the exit selected by thread scheduler 130, the thread enters the thread completion stage. However, if upon execution a different exit of the thread is taken then was selected by thread selector 130, all subsequent threads are cancelled.

As described in greater detail below with reference to FIG. 10, during the completion stage of thread processing all modified register and memory locations of successfully completing threads are written back to the architected state maintained within architected register file 124 and L2 cache 122. Because all required data is forwarded to PEs executing subsequent threads during the thread execution stage, the thread completion stage is completely overlapped with other processing stages, thereby hiding latency.

With reference now to FIG. 7, there is illustrated a high level logical flowchart of a method of scheduling threads for execution in accordance with the illustrative embodiment of a multiscalar data processing system depicted in FIG. 4. The process shown in FIG. 7 will be described with reference to the exemplary multiscalar program depicted in FIG. 11. As

illustrated, the process begins at block 200, which represents the operating system of the multiscalar data processing system depicted in FIG. 4 loading multiscalar program 400 in response to a selected command. The process then proceeds from block 200 to block 202, which depicts multiscalar processor 100 executing ISA instructions on a single one of PEs 132-138 beginning with ISA instruction 402. Next, the process proceeds to block 204, which illustrates a determination of whether or not a SetTP instruction, such as ISA instruction 404, has been executed. If not, scalar execution of ISA instructions continues on a single one of PEs 132-138, as indicated by the process returning from block 204 to block 202.

Referring again to block 204, in response to execution of SetTP instruction 404, which specifies the base address of thread descriptor 406, the process proceeds from block 204 to block 210. Block 210 depicts multiscalar processor 100 initiating multiscalar execution of multiscalar program 400 by loading the base address of thread descriptor 406 into TP 142 of thread scheduler 130. Next, as illustrated at block 212, thread scheduler 130 passes the I-Code pointer and I-Code extension pointer specified within thread descriptor 406 to a free one of PEs 132-138 in conjunction with a thread number that does not conflict with a thread number currently allocated within TL 146. As illustrated at block 213, status register 148 is then updated to indicate that the PE to which the thread was assigned is busy.

The process proceeds from block 213 to block 214, which depicts a determination of whether or not thread descriptor 406 includes a navigation pointer. As described above, the presence of a navigation pointer within thread descriptor 406 indicates that multiscalar compiler 14 has created a set of navigation instructions that may be executed in order to resolve the inter-thread control dependency that determines which of the possible exit points of thread 406 will be taken. In response to a determination by thread scheduler 130 that thread descriptor 406 does not include a navigation pointer, the process proceeds to block 216, which illustrates exit prediction mechanism 140 predicting an exit of thread 408. The process then proceeds from block 216 to block 220. However, in response to a determination at block 214 that thread descriptor 406 includes a navigation pointer, thread scheduler 130 loads the set of navigation instructions pointed to by the navigation pointer and executes the navigation instructions in order to determine an exit of thread 408, as illustrated at block 218. As will be appreciated by those skilled in the art, the execution of navigation instructions by thread scheduler 130 entails either the inclusion of simple arithmetic and control flow execution circuitry within thread scheduler 130 or the execution of the navigation instructions within one of PEs 132-138 on behalf of thread scheduler 130. Following a determination of an exit of thread 408 at either of blocks 216 or 218, the process proceeds to block 220, which illustrates entering the selected exit number within TL 146 in association with the thread number. The process then passes to block 230.

Block 230 depicts a determination of whether or not the exit selected at one of blocks 216 and 218 was marked in thread descriptor 406 as a termination point of multiscalar execution. If so, the process returns to block 202, which depicts multiscalar processor 100 again executing ISA instructions within multiscalar program 400 utilizing only a single one of PEs 132-138. However, in response to a determination at block 230 that the selected exit was not marked by multiscalar compiler 14 as a termination point of multiscalar execution, the process proceeds to block 232. Block 232 illustrates thread scheduler 130 loading thread

descriptor 410, the thread descriptor pointed to by the exit pointer in thread descriptor 406 associated with the selected exit. Thereafter, the process returns to block 212, which has been described.

Referring now to FIG. 8, there is depicted a high level logical flowchart of a method of fetching and dispatching instructions within each of PEs 132-138 of multiscalar processor 100. Although the described process is individually employed by each of PEs 132-138, only PE 132 will be referred to for the sake of simplicity. As illustrated, the process begins at block 250 in response to receipt by PE 132 of an I-Code pointer, I-Code extension pointer, and thread number from thread scheduler 130. The process then proceeds to blocks 252 and 254, which illustrate PE 132 loading the I-Code specified by the I-Code pointer into instruction cache 150 and loading the extension list specified by the I-Code extension pointer into CAM 160. Next, the process passes to block 256, which depicts the instruction sequencer within execution core 158 determining the instruction address of the next ISA instruction to be executed. As depicted at block 258, one or more instructions are then fetched from instruction cache 150 utilizing the instruction address calculated at block 256. The process proceeds from block 258 to block 260, which illustrates a determination of whether or not the instruction address of any of the instructions fetched at block 258 matches an instruction address associated with an instruction extension stored within CAM 160. If not, the process proceeds to block 264. However, in response to a determination that an instruction address of a ISA instruction fetched from instruction cache 150 has a match within CAM 160, CAM 160 furnishes the opcode of the instruction extension to the instruction sequencer of execution core 158, which inserts the extension instruction opcode into the instruction stream at a point indicated by the extension instruction. The process then passes to block 264, which illustrates the instruction sequencer of execution core 158 dispatching one or more ISA instructions and instruction extensions to the execution resources for execution. Thereafter, the process returns to block 256, which has been described.

With reference now to FIG. 9, there is illustrated a high level logical flowchart of a method of instruction execution within execution core 158 of PE 132. As illustrated, the process begins at block 280 in response to the execution resources of execution core 158 receiving at least one instruction dispatched by the instruction sequencer. Thereafter, the process proceeds to block 282, which illustrates the execution resources of execution core 158 decoding the instruction. A determination is then made at block 284 whether or not the dispatched instruction is a WaitFlag extension instruction. If so, the process passes to block 285, which depicts a determination by execution core 158 whether or not the thread being executed is the oldest (non-speculative) thread. For example, execution core 158 can determine if it is executing the oldest thread by interrogating thread scheduler 130, which tracks the ordering of threads executing within PEs 132-138. In response to a determination that execution core 158 is executing the oldest thread, the WaitFlag extension instruction is simply discarded since data consistency is guaranteed. However, in response to a determination that execution core 158 is not executing the oldest thread, the process proceeds to block 286, which illustrates execution core 158 executing the WaitFlag extension instruction by stalling execution of at least one instruction until the specified one of global SFs 180 is set. According to a preferred embodiment, the WaitFlag extension instruction specifies whether the subsequent ISA

instruction or all ISA instructions within the thread are to be stalled. The process then terminates at block 308 until the next instruction is received by the execution resources.

Returning to block 284, in response to a determination that the dispatched instruction is not a WaitFlag extension instruction, the process proceeds to block 288, which illustrates a determination of whether or not the dispatched instruction is a SetFlag extension instruction. If so, the process passes to block 290, which depicts execution core 158 setting one of global SFs 180 indicated by the SetFlag extension instruction. The process thereafter passes to block 308 and terminates until the next instruction is received by the execution resources.

If a determination is made at block 288 that the dispatched instruction is not a SetFlag extension instruction, the process proceeds to block 300, which illustrates a determination of whether or not the dispatched instruction is a SetExit extension instruction. If so, the process proceeds to block 302, which depicts execution core 158 signalling the thread number of the thread under execution and the exit number marked by the SetExit extension instruction to thread scheduler 130. Execution core 158 preferably determines the appropriate exit number from a parameter of the SetExit extension instruction within extension list 60. PE 132 then terminates execution of the thread at block 308 and initiates the thread completion process illustrated in FIG. 10 by transmitting the thread number and exit number to thread scheduler 130.

In response to a determination at block 300 that the dispatched instruction is not a SetExit extension instruction, the process proceeds to block 304, which depicts a determination of whether or not the dispatched instruction is a SetStop extension instruction. If so, the process passes to block 306, which illustrates PE 132 signalling thread scheduler 130 to halt multiscalar execution of the multiscalar program. Thereafter, PE 132 terminates execution of the thread at block 308 and initiates the thread completion process illustrated in FIG. 10 in the manner which has been described. Thus, as illustrated in FIG. 11, if a SetStop extension instruction is executed at the exit of thread 420, execution of multiscalar program 400 continues in a scalar fashion on a single PE.

Referring again to FIG. 9, in response to a determination at block 304 that the dispatched instruction is not a SetStop extension instruction, the process passes to blocks 310-317, which illustrates the execution of an ISA instruction by execution core 158. Referring first to block 310, in response to a read signal from execution core 158, a determination is made whether or not all of the source data required to execute the ISA instruction is available locally within GPR cache 154 and data cache 156 in any of data coherency states 502-508. If so, the process proceeds to block 315, thereby signifying that execution core 158 can access the required data locally. However, in response to a determination that the required data is not owned locally, the process proceeds to block 311, which depicts communication and synchronization logic 152 transmitting a read request on local communication and synchronization mechanism 170 that indicates the required memory address or register number. As described above, PEs having the requested data in any of data coherency states 502-508 will respond to the read request by indicating ownership of the requested data. Arbitration logic 172 then signals the responding PE executing the nearest preceding thread in logical program order to place the requested data on local communication and synchronization mechanism 170. As illustrated at block 312, if a PE responds to the read request, the process proceeds to

block 314. However, if none of PEs 132-138 responds to the read request, the process passes to block 313, which illustrates the PE fetching the required data from the committed state, that is, from either L2 cache 122 or architected register file 124. The process then proceeds to block 314, which illustrates communication and synchronization logic 152 updating the data coherency state of the local data location containing the requested data to valid state 502. Thereafter, the process passes to block 315.

Block 315 depicts communication and synchronization logic signalling global disambiguation buffer 182 with the memory addresses and register numbers accessed to obtain data for the ISA instruction. As described above, global disambiguation buffer 182 records these data location identifiers for subsequent comparison with data locations written by threads that precede the current thread in program order. The process then proceeds to block 316, which illustrates the execution resources of execution core 158 executing the ISA instruction, possibly generating result data that is written to a local data location. As illustrated at block 317, communication and synchronization logic then broadcasts a write request indicating the register number(s) or memory addressee(s), if any, written in response to execution of the ISA instruction. As described above with reference to FIG. 13, the communication and synchronization logic 152 within PEs that are executing threads subsequent to a the signalling thread in program order and that have the indicated data locations) in valid state 502 or dirty state 504 updates the state of the indicated data locations to the appropriate one of valid hazard state 506 and dirty hazard state 508. The data location identifiers broadcast at block 317 are also processed by global disambiguation buffer 182 in order to check for data dependencies. The process proceeds from block 316 to block 317, which illustrates communication and synchronization logic 152 updating the local state of data locations written in response to execution of the ISA instruction, if necessary. Thereafter, the process passes to block 308 and terminates until the next instruction is dispatched to the execution resources of execution core 158 for execution.

With reference now to FIG. 10, there is depicted a high level logical flowchart of a method of thread completion within multiscalar processor 100. According to the illustrative embodiment, threads are completed according to logical program order. As illustrated, the process begins at block 320 in response to receipt by thread scheduler 130 of a thread number and exit number from one of PEs 132-138. The process then proceeds to block 321, which illustrates a determination of whether or not a data dependency was detected during execution of the specified thread. If so, the process passes to block 328, which illustrates thread scheduler sending a reset signal to the signalling PE to invalidate the local data and rescheduling the specified thread for execution within the signalling PE. Thereafter, the process terminates at block 344. Referring again to block 321, in response to a determination that no data dependency was detected during the execution of the specified thread, the process proceeds to block 322.

Block 322 depicts thread scheduler 130 comparing the actual exit number received from the signalling PE with the selected exit number associated with the indicated thread number in TL 146. As illustrated at block 324, a determination is then made whether or not the actual exit number indicated by the signalling PE matches the predicted exit number associated with the thread number in TL 146. If so, the process passes to block 340, which is described below. However, if the actual exit number does not match the exit number recorded in TL 146, the process proceeds to block

330, which depicts thread scheduler 130 sending a reset signal to all PEs executing threads subsequent to the specified thread in program order. Thus, as illustrated at block 330, the occurrence of a control (but not data) hazard requires the cancellation of all subsequent speculative threads. The process then passes to block 332, which depicts thread scheduler 130 updating status register 148 to mark the PEs for which execution was cancelled as free. Next, the process proceeds to block 334, which illustrates thread scheduler 130 scheduling the threads (in accordance with the method depicted in FIG. 7) within the correct execution path. The process then proceeds to block 340.

Block 340 depicts thread scheduler 130 sending a write-back signal to the signalling PE. In response to receipt of the writeback signal, the PE writes back all data locations in dirty state 504 and dirty hazard state 508 to the appropriate one of architected register file 124 and L2 cache 122. In addition, the state of updated locations within L2 cache 122 are marked as valid. The process then passes from block 340 to block 342, which illustrates thread scheduler 130 updating status register 148 to indicate that the signalling PE is free. In addition, TP 142 is updated to point to the thread descriptor indicated by the exit pointer associated with the actual exit point of the completed thread. Thereafter, the process terminates at block 344.

In the hereinbefore described process of thread processing, exceptions occurring during the execution of a multiscalar program are only taken in scalar execution mode. Thus, as illustrated in FIG. 11 at reference numeral 430, PEs 132-138 simply quit execution of threads and return to an idle state in response to the occurrence of an exception. An appropriate exception handler is then executed on one of PEs 132-138. Thereafter, scalar execution of the ISA instructions within multiscalar program 400 is resumed on a single one of PEs 132-138, as depicted at reference numeral 432. Execution of ISA instructions continues in scalar mode until the execution of SetTP instruction 434, which as described above, initializes TP 142 with the base address of thread descriptor 436, thereby restarting concurrent execution of multiple threads.

Out-of-Order Operation

Heretofore, it has been assumed that threads within a multiscalar program are assigned by thread scheduler 130 to PEs 132-138 according to logical program order. However, even greater levels of ILP may be achieved by scheduling threads to PEs 132-138 for speculative out-of-order execution, if a high percentage of the out-of-order threads are data independent from preceding threads.

In order to support out-of-order thread execution, it is desirable to make a number of enhancements to the software and hardware architectures described above. First, referring now to FIG. 14, there is depicted an illustrative embodiment of a thread descriptor generated by multiscalar compiler 14 to support out-of-order execution of threads. As is apparent upon comparison of FIGS. 2 and 14, the thread descriptor 32 illustrated in FIG. 14 is identical to that depicted in FIG. 2, except for the inclusion of meta-thread list pointer 43. Meta-thread list pointer 43 is a 24-bit pointer that indicates, relative to TP 142, the base address of meta-thread list 51, which contains one or more 24-bit meta-thread pointers 53. As illustrated, each meta-thread pointer 53 specifies the base address of a thread descriptor 32 associated with a meta-thread 55 that is to be scheduled to one of PEs 132-138 for out-of-order execution. Unlike the thread 18 to which I-Code pointer 40 points, the meta-threads 55 indirectly specified by meta-thread pointers 53 do not logically follow the thread preceding thread 18 in logical program order.

Instead, meta-threads 55 are threads identified by multiscalar compiler 14 at block 96 of FIG. 1B as control independent from preceding threads once the execution path has reached thread 18 (i.e., each meta-thread 55 will be executed regardless of which exit of thread 18 is taken). Thus, meta-threads 55 can be executed out-of-order with respect to the logical ordering of threads under the assumption that hardware within multiscalar processor 100 will detect and correct for any unidentified data dependencies between meta-threads 55 and preceding threads.

According to the illustrative embodiment, data dependencies between meta-threads and preceding threads are handled at thread completion on a thread region-by-thread region basis, where each meta-thread defines a thread region including the meta-thread and all subsequent threads that logically precede the next meta-thread, if any, in program order. For example, with reference now to FIG. 15, there is illustrated a multiscalar program 520 including threads 522-534, which are depicted in logical program order. As illustrated, thread 522 includes a first possible exit point 540, which if taken causes thread 524 to be executed, and a second possible exit point 542, which if taken causes thread 526 to be executed. Because thread 534 will be executed regardless of which of possible exit points 540 and 542 is actually taken during execution, multiscalar compiler 14 designates thread 534 as a meta-thread child of thread 522 by creating a meta-thread pointer 43 in the thread descriptor 32 associated with thread 522. As illustrated, thread 522 and all logically subsequent threads preceding meta-thread 534 comprise a first thread region 552, and meta-thread 534 and all logically subsequent threads preceding the next meta-thread comprise a second thread region 552.

In order to permit multiscalar processor 100 to identify the boundary between first thread region 550 and second thread region 552, multiscalar compiler 14 creates, within the thread descriptor of thread 532, an exit pointer associated with possible exit point 544 that specifies the base address of the thread descriptor of meta-thread 534 (as would be the case for in-order thread execution). In addition, multiscalar compiler 14 indicates that possible exit point 544 of thread 532 crosses a thread region boundary between first thread region 550 and second thread region 552 by creating a region boundary exit identifier within the 8-bit reserved section following the exit pointer.

Two principal hardware enhancements are made to multiscalar processor 100 in order to support out-of-order thread processing. First, thread scheduler 130 is modified to include four instances of the thread scheduling hardware hereinbefore described. Each instance of thread scheduler 130 is associated with a particular one of the four thread regions in which PEs 132-138 may possibly be executing. A separate TL 146 is utilized by each instance of thread scheduler 130 to track the exit predictions made within the associated thread region. In contrast to TL 146, TP 142, status register 148, and exit prediction mechanism 140 are shared between the four instances of thread scheduler 130.

Second, global disambiguation buffer 182 preferably includes four thread region buffers that are each associated with a respective one of the four possible thread regions in which PEs 132-138 can execute. Like the embodiment of global disambiguation buffer 182 described above with respect to in-order execution, each thread region buffer accumulates the register numbers and memory addresses from which threads within the associated thread region read data and the register numbers and memory addresses to which threads within the associated thread region write data. These data location identifiers are utilized to detect intra-

region data consistency in the manner described above. In addition, as described below with reference to FIG. 18, the identifiers of data locations written by threads within a thread region are utilized during thread completion to verify that all inter-region data dependencies are observed.

Referring now to FIG. 16 there is depicted a high level logical flowchart of a method of scheduling threads in a multiscalar processor that supports out-of-order thread execution. FIG. 16 illustrates the steps performed by each of the four instances of thread scheduler 130 to schedule threads within its associated thread region. As illustrated, the process begins at block 600 and thereafter proceeds to blocks 602-620, which illustrate the first instance of thread scheduler 130 loading a thread descriptor, initiating execution of the associated thread within one of PEs 132-138, selecting one of the exits of the thread, and storing the exit selection within TL 146, in the manner which has been described above with reference to blocks 202-220 of FIG. 7.

The process proceeds from block 620 to block 630, which illustrates a determination of whether or not the exit type of the selected exit specifies that multiscalar execution is to be terminated. If so, the process returns to block 602, which illustrates the resumption of scalar execution by a single one of PEs 132-138. However, in response to a determination at block 630 that the exit type of the selected exit does not specify the termination of multiscalar execution, the process proceeds to block 632, which illustrates the first instance of thread scheduler 130 determining whether the currently loaded thread descriptor includes a meta-thread list pointer 43. If not, the process passes to block 640, which is described below. However, in response to a determination that the thread descriptor includes a meta-thread list pointer 43, the process proceeds to block 634, which depicts the first instance of thread scheduler 130 allocating a new thread region and passing a meta-thread pointer 53 within meta-thread list 51 to a second instance of thread scheduler 130 so that the second instance of thread scheduler 130 can load the thread descriptor associated with the meta-thread 55 and begin the thread scheduling process illustrated in FIG. 16 at block 612. The process then proceeds from block 634 to block 636, which illustrates a determination by the first instance of thread scheduler 130 whether or not additional meta-thread pointers are present within meta-thread list 51. If so, the process returns to block 634, which illustrates the first instance of thread scheduler 130 passing a next meta-thread pointer 53 to a third instance of thread scheduler 130. Referring again to block 636, in response to a determination that all meta-thread pointers 53 within meta-thread list 51 have been passed to other instances of thread scheduler 130, the process proceeds from block 636 to block 640.

Block 640 illustrates a determination of whether or not the exit type of the selected exit point indicates that the exit point of the current thread defines a boundary between two thread regions. If not, the process proceeds to block 642, which illustrates the first instance of thread scheduler 130 loading the thread descriptor indicated by the exit pointer associated with the selected exit point. The process then returns to block 612, which illustrates the first instance of thread scheduler 130 processing the new thread descriptor. Returning to block 640, in response to a determination that the exit type of the selected exit point indicates that the selected exit point defines a thread region boundary, the process proceeds to block 650, which depicts the first instance of thread scheduler 130 discontinuing the scheduling of threads and waiting for the associated thread region to be completed. Of course, if a data or control hazard is detected within the thread region while the first instance of

thread scheduler 130 is waiting at block 650, the first instance of thread scheduler 130 recovers from the detected hazard by scheduling the appropriate thread(s). Following block 650, the process passes to block 652, which illustrates the first instance of thread scheduler 130 waiting for a new thread region to be allocated in the manner described above with reference to block 634. In response to receipt of a meta-thread pointer 53 by the first instance of thread scheduler 130, the process returns to block 612, which has been described.

With reference now to FIG. 17, there is illustrated a high level logical flowchart of a method of executing instructions within the PE of a multiscalar processor that supports out-of-order thread execution. As illustrated, the process begins at block 680 in response to receipt of an instruction dispatched to the execution resources of execution core 158 in accordance with the method described above with reference to FIG. 8. The process then proceeds to blocks 682-706, which correspond to blocks 282-306 of FIG. 9 and accordingly are not further described here.

Referring now to block 704, in response to a determination that the dispatched instruction is not a SetStop extension instruction, thereby indicating that the dispatched instruction is an ISA instruction, the process proceeds to block 710. Block 710 illustrates a determination of whether or not all of the source data required to execute the dispatched ISA instruction are available locally in any of data coherency states 502-508. If so, the process passes to block 715, which is described below. However, in response to a determination that all of the source data required to execute the ISA instruction are not available locally within GPR cache 154 and data cache 156, the process proceeds to block 711, which depicts communication and synchronization logic 152 transmitting a read request on local communication and synchronization mechanism 170 that indicates the memory address or register number containing the required data as well as the number of the thread region in which the PE is executing. A PE snooping local communication and synchronization mechanism 170 responds to the read request if the PE is executing an earlier thread within the same thread region and owns the requested data in one of data coherency states 502-508. As illustrated at block 712, if the required data is available from another PE executing a thread in the same thread region as the requesting PE, the process passes to block 714. However, in response to a determination at block 712 that the required data is not available from another PE executing within the same thread region, the process proceeds to block 713, which illustrates the requesting PE fetching the required data from L2 cache 122 or architected register file 124. The process then passes to block 714, which depicts communication and synchronization logic 152 updating the data state of the accessed data to valid state 502. Thereafter, the process proceeds to block 715.

Block 715 illustrates communication and synchronization logic 182 transmitting the identifier of each data locations accessed to obtain an operand for the ISA instruction to the appropriate thread region buffer within global disambiguation buffer 182. Next, as depicted at block 716, the execution resources of execution core 158 execute the ISA instruction. The process then proceeds to block 717, which illustrates communication and synchronization logic 152 broadcasting a write request on logic communication and synchronization mechanism 170 that indicates to all subsequent threads within the same thread region each memory address or register number, if any, written in response to execution of the ISA instruction. In addition, as depicted at block 718, communication and synchronization logic 152 records the

register number or memory address of each data location written by the ISA instruction in the thread region buffer associated with the current thread region. As described below with respect to FIG. 18, the information within the thread region buffer is utilized to correct for inter-region data dependencies upon the completion of all threads within the current thread region. The process then proceeds from block 717 to block 718, which illustrates communication and synchronization logic 152 updating the local state of data locations written in response to execution of the ISA instruction. Thereafter, the process terminates at block 708.

Referring now to FIG. 18, there is depicted a high level logical flowchart of a method of thread completion within a multiscalar processor that supports out-of-order thread execution. As illustrated, the process begins at block 820, in response to receipt of a thread number and exit number by the instance of thread scheduler 130 associated with the thread region to which the executed thread belongs. The process proceeds from block 820 to block 821, which depicts a determination of whether or not a data dependency was detected during execution of the specified thread. If so, the process proceeds to block 828, which illustrates the instance of thread scheduler 130 sending a reset signal to the signalling PE to invalidate all local data and rescheduling the specified thread for execution by the signalling PE. The process then passes to block 844 through page connector B and terminates.

Referring again to block 821, in response to a determination at block 821 that no data dependency was detected during the execution of the specified thread, the process proceeds to block 822, which illustrates a determination of whether or not the exit type of the exit pointer associated with the actual exit point of the executed thread indicates that the exit point defines a thread region boundary. If so, the process proceeds to block 838, which illustrates the instance of thread scheduler 130 causing the identifiers of all data locations written by threads within the current thread region to be broadcast from the thread region buffer associated with the current thread region to all threads within the immediately subsequent thread region. As described above with reference to FIG. 13, PEs executing threads within the subsequent thread region utilize the broadcast write requests to update the data coherency state of data locations in valid state 502 and dirty state 504 to valid hazard state 506 and dirty hazard state 508, respectively. In addition, the identifiers of data locations written by threads within the current thread region are transferred to the thread region buffer associated with the immediately subsequent thread region so that global disambiguation buffer 182 can check for inter-thread data dependencies between the immediately subsequent thread region and the current thread region. The process then passes to block 840.

With reference again to block 822, in response to a determination that the actual exit taken by the executed thread does not define a thread region boundary, the process proceeds to block 824, which depicts the instance of thread scheduler 130 comparing the actual exit number received from the signalling PE with the exit number associated with the thread number in TL 146. A determination is then made at block 826 whether or not the actual exit number indicated by the signalling PE matches the selected exit number associated with the thread number in TL 146. If so, the process passes to block 840, which is described below. If the actual and selected exit numbers do not match, however, the process proceeds from block 824 to block 830, which illustrates the instance of thread scheduler 130 sending a reset signal to all PEs that are executing threads within the

current thread region that are subsequent to the completed thread. Thus, in contrast to the in-order execution case, the detection of a control hazard during out-of-order execution requires only the cancellation of all subsequent threads within the same thread region and not all subsequent threads. The process proceeds from block 830 to block 832, which illustrates the instance of thread scheduler 130 updating status register 148 to mark the PEs for which execution was cancelled as free. Next, the process passes to block 834, which illustrates the instance of thread scheduler 130 scheduling threads within the correct execution path in accordance with the method depicted in FIG. 16. The process then passes to block 840.

Block 840 illustrates the instance of thread scheduler 130 transmitting a writeback signal to the signalling PE, which in response to receipt of the writeback signal, writes back dirty (modified) registers and memory addresses to L2 cache 122 and architected file 124. The process then proceeds to block 842, which illustrates the instance of thread scheduler 130 updating status register 148 to indicate that the signalling PE is free. In addition, TP 142 is updated to point to the thread associated with the exit point of the completed thread. The process then terminates at block 844.

As will be appreciated from the foregoing description, the multiscalar software and hardware architectures disclosed herein provide numerous advantages over prior art superscalar, multiprocessor, and multiscalar data processing systems. By providing linked thread descriptors within a T-Code stream that is parallel to, yet separate from the I-Code stream, the present multiscalar software architecture avoids the performance degradation experienced in prior art multiscalar systems due to an increase in program length. Maintaining separate processing paths for the T-Code and I-Code streams and providing hardware and software support for the dynamic insertion of auxiliary instructions within the I-Code stream ensures backward compatibility between the multiscalar software architecture described herein and scalar object code executable by conventional processors. The dynamic insertion of auxiliary instructions within the I-Code stream and the possibility of including a single instruction within multiple threads further permits a single instruction to be associated with multiple instruction extensions. Thus, an instruction within a first thread, which produces a particular register value and is therefore associated with a SetFlag extension instruction within the extension list of the first thread, may also be included in a second thread and associated with a second SetFlag extension instruction within the extension list of the second thread.

Furthermore, the data consistency support provided by the SetFlag/WaitFlag paradigm permits multiple instructions to be synchronized utilizing a single execution control facility that may be employed for both register accesses and disambiguable memory accesses. In contrast to prior art data processing systems, the hardware and software architectures herein disclosed support both speculative and non-speculative execution of multiple threads through the generation of navigation instructions executable by the thread scheduler. The execution of navigation instructions by the thread scheduler reduces the amount of speculative work that is discarded in response to exit mispredictions, thereby enhancing IPC performance.

Moreover, from the foregoing description of out-of-order thread processing, it should be apparent that partitioning multiscalar programs into thread regions in this manner has a number of advantages. First, inter-region thread interaction is minimized through the use of different protocols for inter-region and intra-region thread interaction. According

to the illustrative embodiment, the inter-thread data coherency communication and SetFlag/WaitFlag extension instructions are utilized during the thread execution stage of out-of-order thread processing to maintain data coherency and register data consistency between threads within the same thread region. However, because threads in different thread regions are executed under the assumption of inter-region data and control independence, data coherency communication between threads in different thread regions is eliminated and verification of register data consistency is deferred until the thread completion stage of thread processing, which is performed according to the logical program order of thread regions.

Second, delaying the verification of data consistency until thread writeback has the advantage that computation performed by a meta-thread is not discarded in response to speculative execution of threads within a mispredicted execution path upon which execution of the meta-thread is seemingly dependent. For example, with reference again to FIG. 15, if an instruction in thread 534 has an apparent register data dependency upon an instruction in thread 526 and possible exit point 542 of thread 522 is predicted, thread 534 and subsequent threads within thread region 552 are not cancelled if it is determined that the exit point of thread 522 was mispredicted.

Third, the recovery activities performed in response to the detection of data hazard during out-of-order thread processing entail a potentially smaller performance penalty than those performed in response to the detection of a control or data hazard during in-order thread processing. As described above and as illustrated at block 330 of FIG. 10, for in-order thread processing the detection of a control hazard during thread writeback entails the cancellation of all threads subsequent to the thread being processed. In contrast, the detection of a control hazard between threads within a thread region only requires that subsequent threads within the same thread region be cancelled. Thus, the discarding of control independent work is eliminated.

Fourth, thread regions permit greater utilization of a limited shared resource, such as SFs 180, by allocating a separate instance of the shared resource to each thread region. For example, assume that SFs 180 include four instances of 32 SFs each, where each instance of SFs 180 is identified by a respective one of thread regions 0-3 so that a PE must transmit both a thread region number and a SF number in order to set a SF. In addition, referring again to FIG. 15, assume that thread 522, which is in thread region 0, contains a "write" instruction having an associated SetFlag extension instruction that sets SF4 and that thread 532, which is also in thread region 0, contains a "read" instruction having an associated WaitFlag extension instruction that delays execution of the "read" instruction until SF4 is set. In this exemplary embodiment, data consistency for the "read" instruction in thread 532 is guaranteed even if meta-thread 534, which is scheduled to one of PEs 132-138 for execution immediately following thread 522, contains an instruction having an associated SetFlag extension instruction that targets SF4. Thus, organizing threads into thread regions prevents contention for shared resources between threads in different regions and minimizes the complexity of the processor hardware required to track utilization of shared resources by out-of-order threads.

While an illustrative embodiment has been particularly shown and described, it will be understood by those skilled in the art that various changes in form and detail may be made therein without departing from the spirit and scope of the illustrative embodiment. For example, although aspects

of the illustrative embodiment have been described with respect to specific "method steps" implementable within a data processing system, those skilled in the art will appreciate from the foregoing description that the illustrative embodiment can alternatively be implemented as a computer program product for use with a data processing system. Such computer program products can be delivered to a computer via a variety of signal-bearing media, which include, but are not limited to: (a) information permanently stored on non-writable storage media (e.g., CD-ROM); (b) information alterably stored on writable storage media (floppy diskettes or hard disk drives); or (c) information conveyed to a computer through communication media, such as through a computer or telephone network. It should be understood, therefore, that such signal-bearing media, when carrying computer readable instructions that direct the method functions of the illustrative embodiment, represent alternative embodiments.

What is claimed is:

1. A method of constructing a program executable by a processor, said processor including one or more processing elements for executing threads and a thread scheduler for assigning threads to said one or more processing elements for execution, said method comprising:

providing a plurality of threads, each of said plurality of threads including at least one control flow instruction; determining, from one or more control flow instructions within said plurality of threads, a condition upon which execution of a particular thread among said plurality of threads depends; and

in response to said determination, creating at least one navigation instruction, said at least one navigation instruction indicating that said particular thread is to be assigned to one of said one or more processing elements in response to said condition, wherein said plurality of threads and said at least one navigation instruction together comprise said program.

2. The method of claim 1, said method further comprising constructing a plurality of data structures that are each associated with a respective one of said plurality of threads, wherein each of said plurality of data structures specifies a plurality of possible exit points of an associated thread.

3. The method of claim 1, wherein said step of creating at least one navigation instruction comprises the step of creating a loop construct.

4. The method of claim 1, wherein said step of creating at least one navigation instruction comprises the step of creating an if-then-else construct.

5. The method of claim 1, wherein said method further comprises the steps of:

providing a plurality of instructions of a selected instruction set architecture; and

assigning each of said plurality instruction to at least one of said plurality of threads.

6. A method of executing a program within a processor including one or more processing elements and a thread scheduler, said method comprising:

providing a program including a plurality of threads and at least one navigation instruction, said at least one navigation instruction indicating that a particular thread is to be assigned to one of said one or more processing elements in response to a particular condition;

executing said at least one navigation instruction to determine if said particular condition is present; and

in response to a determination that said particular condition is present, assigning said particular thread to one of said one or more processing elements for execution.

7. The method of claim 6, and further comprising the step of:

thereafter, executing said particular thread within said one of said one or more processing elements.

8. The method of claim 6, and further comprising:

in response to a determination that said particular condition does not exist, assigning a thread subsequent to said particular thread in logical program order to one of said one or more processing elements without assigning said particular thread.

9. The method of claim 6, and further comprising:

speculatively assigning a thread among said plurality of threads that is not associated with said at least one navigation instruction to one of said one or more processing elements.

10. The method of claim 6, said at least one navigation instruction comprising a loop construct, wherein said executing step comprises the step of comparing a value of a loop iteration variable to a second value.

11. The method of claim 6, said at least one navigation instruction comprising an if-then-else construct wherein said executing step comprises the step of determining whether an if statement within said if-then-else construct is logically true.

12. A system for constructing a program executable by a processor including one or more processing elements for executing threads and a thread scheduler for assigning threads to said one or more processing elements for execution, said system comprising:

means, responsive to receipt of a plurality of threads, each of said plurality of threads including at least one control flow instruction, for determining, from one or more control flow instructions within said plurality of threads, a condition upon which execution of a particular thread among said plurality of threads depends; and

means, responsive to said determination, for creating at least one navigation instruction, said at least one navigation instruction indicating that said particular thread is to be assigned to one of said one or more processing elements in response to said condition, wherein said plurality of threads and said at least one navigation instruction together comprise said program.

13. The system of claim 12, said system further comprising means for constructing a plurality of data structures that are each associated with a respective one of said plurality of threads, wherein each of said plurality of data structures specifies a plurality of possible exit points of an associated thread.

14. The system of claim 12, wherein said means for creating at least one navigation instruction comprises means for creating a loop construct.

15. The system of claim 12, wherein said means for creating at least one navigation instruction comprises means for creating an if-then-else construct.

16. The system of claim 12, wherein said system further comprises:

means, responsive to receipt of a plurality of instructions of a selected instruction set architecture, for assigning each of said plurality instruction to at least one of said plurality of threads.

17. A processor, comprising:

one or more processing elements for executing threads; means, responsive to loading at least one navigation instruction, said at least one navigation instruction indicating that a particular thread is to be assigned to one of said one or more processing elements in

29

response to a particular condition, for determining if said particular condition is present; and

means, responsive to a determination that said particular condition is present, for assigning said particular thread to one of said one or more processing elements for execution.

18. The processor of claim 17, said processor further comprising:

means, responsive to a determination that said particular condition does not exist, for assigning a thread subsequent to said particular thread in a logical program order to one of said one or more processing elements without assigning said particular thread.

19. The processor of claim 17, said processor further comprising:

means for speculatively assigning a thread among said plurality of threads that is not associated with said at least one navigation instruction to one of said one or more processing elements.

20. The processor of claim 17, said at least one navigation instruction comprising a loop construct, wherein said means for determining comprises means for comparing a value of a loop iteration variable to a second value.

21. The processor of claim 17, said at least one navigation instruction comprising an if-then-else construct, wherein said means for determining comprises means for determining whether an if statement within said if-then-else construct is logically true.

22. A computer program product for creating a program executable by a processor including one or more processing elements for executing threads and a thread scheduler for

30

assigning threads to said one or more processing elements, said computer program product comprising:

signal bearing means;

instruction code within said signal bearing means for causing a data processing system to determine, from one or more control flow instructions within a plurality of threads, a condition upon which execution of a particular thread among said plurality of threads depends; and

responsive to said determination, instruction code within said signal bearing means for causing said data processing system to create at least one navigation instruction, said at least one navigation instruction indicating that said particular thread is to be assigned to one of said one or more processing elements in response to said condition, wherein said plurality of threads and said at least one navigation instruction together comprise said program.

23. A computer program product, comprising:

a program executable by a processor including one or more processing elements for executing instructions and a thread scheduler for assigning threads to said one or more processing elements, said program including at least one navigation instruction, said at least one navigation instruction indicating that said particular thread is to be assigned to one of said one or more processing elements in response to said condition; and

signal bearing means bearing said program.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION

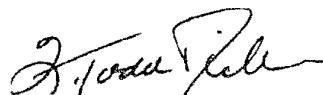
PATENT NO. : 5,887,166
DATED : Mar. 23, 1999
INVENTOR(S) : *Mallick et al.*

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

In col. 1, line 23, please change "08/767,488" to -- 08/767,492 --.

Signed and Sealed this
Sixteenth Day of November, 1999

Attest:

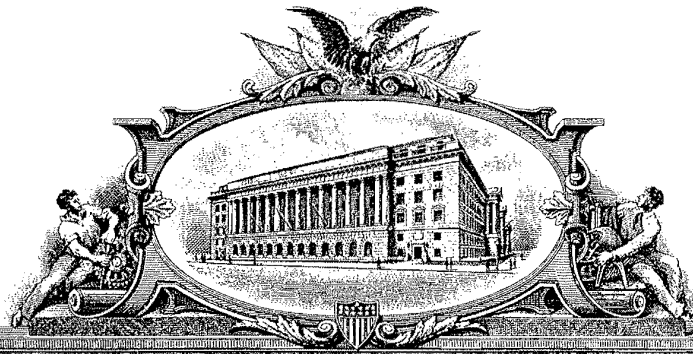


Attesting Officer

Q. TODD DICKINSON

Acting Commissioner of Patents and Trademarks

TW 7551726



THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office

October 19, 2015

THIS IS TO CERTIFY THAT ANNEXED IS A TRUE COPY FROM THE
RECORDS OF THIS OFFICE OF THE FILE WRAPPER AND CONTENTS
OF:

APPLICATION NUMBER: 09/556,474

FILING DATE: *April 21, 2000*

PATENT NUMBER: 6,630,935

ISSUE DATE: *October 07, 2003*

By Authority of the
Under Secretary of Commerce for Intellectual Property
and Director of the United States Patent and Trademark Office



Sylvia Holley
SYLVIA HOLLEY
Certifying Officer

Parts (2) of (2) Parts



3/8 reconsideration
2/27/03

PATENT APPLICATION

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

RECEIVED
FEB 26 2003
Technology Center 2600

Applicants: Ralph Clayton Taylor et al.
Serial No.: 09/556,474
Filing Date: April 21, 2000
Confirmation No.: 6798

Examiner: Mackly Monestime
Art Group: 2676
Docket No.: 0100.0000650
Our File No.: 00100.00.0650

Title: GEOMETRIC ENGINE INCLUDING A COMPUTATIONAL MODULE FOR USE IN A VIDEO GRAPHICS CONTROLLER

Box Non-Fee Amendment
Commissioner for Patents
U.S. Patent and Trademark Office
Washington, D.C. 20231

Certificate of First Class Mailing
I hereby certify that this paper is being deposited with the United States Postal Service as first-class mail in an envelope addressed to: Box Non-Fee Amendment, Commissioner for Patents, Washington, D.C.20231, on this date.

2/19/03 *Margaret Caruso*
Date Margaret Caruso

AMENDMENT

Dear Sir:

In response to the Office Action dated November 19, 2002, Applicants submit the following remarks.

REMARKS

Applicants respectfully traverse and request reconsideration.

As a preliminary matter, Applicants wish to thank the Examiner for the notice that claims 9-18 have been allowed.

Claims 1-2, 6 and 7 stand rejected under 35 U.S.C. §102(b) as being anticipated by U.S. Patent No. 5,887,166 (Mallick et al.). The Mallick reference describes a method and system for constructing a program including a navigation instruction for a multiscale program. As such, the cited reference teaches a process for constructing a multiscale program and teaches using

during a first pass, a multiscalar compiler that translates each high level instruction into one or more executable instruction set architecture ("ISA") instructions arranged in a particular program order. The multiscalar compiler partitions the ISA instructions into one or more threads. When a thread is executed, the first instruction within the thread is always executed, but there are multiple possible execution paths out of the thread. During a second pass, the multiscalar compiler generates a thread code stream including a number of thread descriptors that are each associated with a respective one of the threads. Each thread descriptor provides the information needed to support multiscalar thread scheduling, thread prediction, and thread synchronization, including pointers to both corresponding threads and subsequent thread descriptors. A thread descriptor is a data structure containing a number of 32 bit entries. The multiscalar compiler utilizes a set flag and wait flag extension instructions to resolve inter-thread register data dependency.

As to claims 1-2 and 7, the Mallick reference has been cited as teaching each and every claim limitation of these claims. However, Applicants respectfully submit that the Mallick reference is silent as to specific aspects of Applicants' claimed invention. For example, among other differences, Mallick does not disclose the claimed thread controllers nor the claimed arbitration module. For example, the Office Action cites items 30 and 32 of Mallick as the claimed thread controllers. However, items 30 and 32 are merely thread descriptors which are data structures. In contrast, Applicants' claimed thread controllers may be state machines or any other suitable structure that may contain for example op codes and other information if desired. The thread descriptors 30 and 32 as such cannot be the claimed the thread controllers.

In addition, the Office Action states that the claimed arbitration module is allegedly taught as items no. 172 and 130 of Mallick. Applicants claimed arbitrations module, among

other things, is coupled to a plurality of thread controllers and utilizes an application's specific prioritization scheme to provide op codes from the thread controllers to the computation engine in an order to minimize idle time of the computation engine.

The Office Action cites col. 7, lines 45-48 and col. 10, lines 57-60 as allegedly teaching the claimed arbitration module. However, Applicants respectfully note that col. 7, lines 45-48, refers to the multiscalar compiler 14 and does not refer to the arbitration logic 172 or the thread scheduler 130. As such, Applicants are unsure as to why this particular passage has been cited and respectfully requests clarification of the same. Col. 10, lines 57-60, refers to the thread scheduler 130, but notes that the thread scheduler 130 uses a T-code cache that stores the thread descriptors thereby establishing separate fetch paths for the I-code and T-code streams to reduce access latency. This is not an application specific prioritization scheme that provides op codes from the thread controllers in a way that minimizes idle time, but instead is a mechanism which stores thread descriptors in a cache to provide separate fetch paths. This is a different operation from that claimed by Applicants. As such, these claims are believed to be in condition for allowance.

As to claim 6, the Office Action cites FIG. 1, items 30 and 32 of Mallick, as allegedly disclosing a thread controller that includes at least one of a transform thread controller, a clip thread controller, a barycentric controller, and an attribute thread controller. However, Applicants respectfully submit that they are unable to find reference to such thread controllers as claimed or to reference items 30 and 32. Since it appears that Mallick is silent as to graphics transformations, clipping operations, barycentric controllers or attribute thread controllers, Applicants respectfully submit that this claim is also in condition for allowance. If the rejection

is maintained, Applicants respectfully request a showing by column and line number of where the Mallick reference teaches such specific thread controllers as claimed.

Claim 3 stands rejected under 35 U.S.C. §103(a) as being unpatentable over Mallick in view of U.S. Patent No. 5,909,544 (Anderson II et al.). The Office Action cites column 7, lines 9-27 of Mallick as allegedly teaching that each op code includes a controller identity, type of operation, and wherein each of the plurality of thread controllers maintain latency data for operation codes of a corresponding thread and wherein each of the plurality of thread controllers releases operation codes to the arbitration module in accordance with the latency data. Applicants respectfully submit that the cited portion of Mallick fails to teach, among other things, the thread controller that variation latency data as claimed. The cited portion of Mallick instead teaches merely that the thread descriptor may contain an extension pointer that points to an extension list containing auxiliary extension instructions that are to be dynamically inserted into threads by multiscalar processor hardware during execution. It does not teach that the op codes include a controller identity. Applicants respectfully request the column and line showing where the op codes in Mallick include a controller identity and a type of operation as claimed.

In addition, Applicants claim that the thread controllers maintain latency data for operation codes. The Office Action previously indicated that the thread controllers are items 30 and 32 of Mallick. However, the thread descriptors do not include latency data and the thread controllers do not hold op codes and release them based on the latency data to an arbitrator. Applicants are unable to find reference in the cited portion to the latency data as alleged in the Office Action. As claimed, the thread controllers maintain latency data and release op codes to an arbitration module in accordance with the latency data. Such as structure and operation is not


taught or suggested by the cited reference. As such, these claims are also believed to be in condition for allowance.

Claims 4-5 and claim 8 stand rejected under 35 U.S.C. §103(a) as being unpatentable over Mallick in view of U.S. Patent No. 6,192,384 (Dally et al.). Dally is directed to a system and method for performing compound vector operations. Applicants agree that Mallick does not disclose a computation module that comprises a vector engine that performs structure operations. The Office Action alleges that one of ordinary skill would combine the teachings of Dally with those of Mallick, but the Office Action does not appear to provide any factual support for the conclusion that one would combine the references. In particular, the Applicants respectfully submit that references cannot be combined with the knowledge of Applicants' claimed invention, but there must be some motivation to combine the references found elsewhere other than Applicants' own claimed invention. The Office Action cites col. 2, lines 35-45, of Dally. This cited portion merely states that the Dally system performs a compound vector operation and generates a result that is written back to the stream register file. It appears that the Dally reference teaches an opposite approach to that of Applicants' claimed invention since it does not appear to contemplate multiple thread controllers or arbitration modules coupled to a plurality of thread controllers. As such, these claims are also believed to be in condition for allowance.

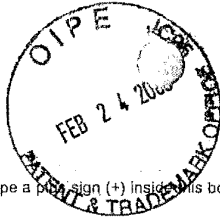
Accordingly, Applicants respectfully submit that the claims are in condition for allowance, and that an early Notice of Allowance be issued in this application. The Examiner is invited to contact the below-listed attorney if the Examiner believes that a telephone conference will advance the prosecution of this application.

Respectfully submitted,

Date: February 19, 2003

By 
Christopher J. Reckamp
Registration No 34,414

Vedder, Price, Kaufman & Kammholz
222 North LaSalle Street
Chicago, Illinois 60601
PHONE: (312) 609-7599
FAX: (312) 609-5005



2676

Please type a plus sign (+) inside this box →

PTO/SB/21 (08-00)
 Approved for use through 10/31/2002. OMB 0651-0031
 U.S. Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

<h1>TRANSMITTAL FORM</h1> <p><i>(to be used for all correspondence after initial filing)</i></p>	Application Number	09/556,474
	Filing Date	April 21, 2000
	First Named Inventor	Ralph Clayton Taylor
	Group Art Unit	2676
	Examiner Name	M. Monestime
Total Number of Pages in This Submission	Attorney Docket Number	0100.0000650

RECEIVED
 FEB 26 2003
 Technology Center 2600

ENCLOSURES (check all that apply)		
<input type="checkbox"/> Fee Transmittal Form	<input type="checkbox"/> Assignment Papers (for an Application)	<input type="checkbox"/> After Allowance Communication to Group
<input type="checkbox"/> Fee Attached	<input type="checkbox"/> Drawing(s)	<input type="checkbox"/> Appeal Communication to Board of Appeals and Interferences
<input checked="" type="checkbox"/> Amendment / Reply	<input type="checkbox"/> Licensing-related Papers	<input type="checkbox"/> Appeal Communication to Group (Appeal Notice, Brief, Reply Brief)
<input type="checkbox"/> After Final	<input type="checkbox"/> Petition	<input type="checkbox"/> Proprietary Information
<input type="checkbox"/> Affidavits/declaration(s)	<input type="checkbox"/> Petition to Convert to a Provisional Application	<input type="checkbox"/> Status Letter
<input type="checkbox"/> Extension of Time Request	<input checked="" type="checkbox"/> Power of Attorney, Revocation Change of Correspondence Address	<input checked="" type="checkbox"/> Other Enclosure(s) (please identify below):
<input type="checkbox"/> Express Abandonment Request	<input type="checkbox"/> Terminal Disclaimer	Form PTO/SB/08A; copies of cited references; return receipt postcard.
<input checked="" type="checkbox"/> Information Disclosure Statement	<input type="checkbox"/> Request for Refund	
<input type="checkbox"/> Certified Copy of Priority Document(s)	<input type="checkbox"/> CD, Number of CD(s) _____	
<input type="checkbox"/> Response to Missing Parts/ Incomplete Application	Remarks	
<input type="checkbox"/> Response to Missing Parts under 37 CFR 1.52 or 1.53		

SIGNATURE OF APPLICANT, ATTORNEY, OR AGENT	
Firm or individual name	Christopher J. Reckamp Reg. No. 34,414
Signature	
Date	2-19-03

CERTIFICATE OF MAILING			
I hereby certify that this correspondence is being deposited with the United States Postal Service with sufficient postage as first class mail in an envelope addressed to: Commissioner for Patents, Washington, DC 20231 on this date: 2-19-03			
Typed or printed name	Margaret Caruso		
Signature		Date	2-19-03

Burden Hour Statement: This form is estimated to take 0.2 hours to complete. Time will vary depending upon the needs of the individual case. Any comments on the amount of time you are required to complete this form should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, Washington, DC 20231. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Assistant Commissioner for Patents, Washington, DC 20231.

#1037629

Please type a plus sign (+) inside this box

Approved for use through 10/31/2002. OMB 0651-0035
U.S. Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

5104
2013
PTO/SB/124 (10-00)

<p align="center">CHANGE OF CORRESPONDENCE ADDRESS</p> <p align="center"><i>Application</i></p> <p>Address to: Assistant Commissioner for Patents Washington, D.C. 20231.</p>	Application Number	09/556,474
	Filing Date	04/21/2000
	First Named Inventor	Taylor et al.
	Group Art Unit	2676
	Examiner Name	M. Monestime
	Attorney Docket Number	0100.0000650

PATENT & TRADEMARK OFFICE
FEB 24 2003

RECEIVED
FEB 24 2003
Technology Center 2600
Label here
23418
PATENT TRADEMARK OFFICE

Please change the Correspondence Address for the above-identified application to:

Customer Number
Type Customer Number here

OR

<input type="checkbox"/> Firm or Individual Name			
Address			
Address			
City	State	ZIP	
Country			
Telephone	Fax		

This form cannot be used to change the data associated with a Customer Number. To change the data associated with an existing Customer Number use "Request for Customer Number Data Change" (PTO/SB/124).

I am the :

Applicant/Inventor.

Assignee of record of the entire interest.
Statement under 37 CFR 3.73(b) is enclosed. (Form PTO/SB/96).

Attorney or Agent of record.

Registered practitioner named in the application transmittal letter in an application without an executed oath or declaration. See 37 CFR 1.33(a)(1). Registration Number _____

Typed or Printed Name
Christopher J. Reckamp, Reg. No. 34,414

Signature
Christopher J. Reckamp

Date
2-19-03

NOTE: Signatures of all the inventors or assignees of record of the entire interest or their representative(s) are required. Submit multiple forms if more than one signature is required, see below*.

*Total of 1 forms are submitted.

Burden Hour Statement: This form is estimated to take 3 minutes to complete. Time will vary depending upon the needs of the individual case. Any comments on the amount of time you are required to complete this form should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, Washington, DC 20231. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Assistant Commissioner for Patents, Washington, DC 20231.



4/19/03
2/19/03

PATENT APPLICATION

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicants: Ralph Clayton Taylor et al. Examiner: Mackly Monestime
Serial No.: 09/556,474 Art Group: 2676
Filing Date: April 21, 2000 Our File No.: 00100.00.0650
Confirmation No.: 6798 Docket No.: 0100.0000650

Title: GEOMETRIC ENGINE INCLUDING A COMPUTATIONAL MODULE FOR USE IN A VIDEO GRAPHICS CONTROLLER

Box Non-Fee Amendment
Commissioner for Patents
U.S. Patent and Trademark Office
Washington, D.C. 20231

Certificate of First Class Mailing
I hereby certify that this paper is being deposited with the United States Postal Service as first-class mail in an envelope addressed to: Box Non-Fee Amendment, Commissioner for Patents, Washington, D.C. 20231, on this date.

2/19/03 Margaret Caruso
Date Margaret Caruso

INFORMATION DISCLOSURE STATEMENT IN ACCORDANCE WITH 37 C.F.R. §§1.97 AND 1.98

Pursuant to 37 CFR §§1.97 and 1.98, Applicants hereby respectfully submit that the below references were known to Applicants less than three months ago and submit the following statement consisting of:

- I. A list of documents
- II. General remarks.

A copy of each listed document is enclosed herewith, along with Form PTO/SB/08A.

I. Documents


A. U.S. Patents

<u>Patent No.</u>	<u>Inventor</u>	<u>Issue Date</u>
4,964,042	Sterling et al.	October 16, 1990
6,018,353	Deering et al.	January 25, 2000
6,091,506	Payne et al.	July 18, 2000

II. General Remarks

The submission of the above documents is not an admission that the information is prior art, analogous or otherwise material. It is respectfully requested that the above listed documents be considered and made of record in the present application.

Respectfully submitted,

By: 
Christopher J. Reckamp
Registration No. 34,414

Date: Feb. 19, 2003

Vedder, Price, Kaufman & Kammholz
222 North LaSalle Street
Chicago, Illinois 60601
Phone: (312) 609-7599
Fax: (312) 609-5005

Substitute for form 1449A/PTO		Complete if Known	
INFORMATION DISCLOSURE STATEMENT BY APPLICANT <i>(use as many sheets as necessary)</i>		Application Number	09,556,474
		Filing Date	April 21, 2000
		First Named Inventor	Ralph Clayton Taylor et al.
		Group Art Unit	2676
		Examiner Name	M. Monestime
Sheet 1 of 1	Attorney Docket Number	0100.0000650	

U.S. PATENT DOCUMENTS						
Examiner Initials*	Cite No. ¹	U.S. Patent Document		Name of Patentee or Applicant of Cited Document	Date of Publication of Cited Document MM-DD-YYYY	Pages, Columns, Lines, Where Relevant Passages or Relevant Figures Appear
		Number	Kind Code ² (if known)			
MM		4,964,042		Sterling et al.	10/16/1990	
MM		6,018,353		Deering et al.	01/25/2000	
MM		6,091,506		Payne et al.	07/18/2000	
MM		6,212,542		Kahle et al.	04/03/2001	

FOREIGN PATENT DOCUMENTS								
Examiner Initials*	Cite No. ¹	Foreign Patent Document			Name of Patentee or Applicant of Cited Document	Date of Publication of Cited Document MM-DD-YYYY	Pages, Columns, Lines, Where Relevant Passages or Relevant Figures Appear	T ⁶
		Office ³	Number ⁴	Kind Code ⁵ (if known)				

Examiner Signature	<i>Markby Monestime</i>	Date Considered	4/23/03
--------------------	-------------------------	-----------------	---------

*EXAMINER: Initial if reference considered, whether or not citation is in conformance with MPEP 609. Draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.
¹ Unique citation designation number. ² See attached Kinds of U.S. Patent Documents. ³ Enter Office that issued the document, by the two-letter code (WIPO Standard ST.3). ⁴ For Japanese patent documents, the indication of the year of the reign of the Emperor must precede the serial number of the patent document. ⁵ Kind of document by the appropriate symbols as indicated on the document under WIPO Standard ST. 16 if possible. ⁶ Applicant is to place a check mark here if English language Translation is attached.

Burden Hour Statement: This form is estimated to take 2.0 hours to complete. Time will vary depending upon the needs of the individual case. Any comments on the amount of time you are required to complete this form should be sent to the Chief Information Officer, U. S. Patent and Trademark Office, Washington, DC 20231. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Assistant Commissioner for Patents, Washington, DC 20231.

United States Patent [19]

Sterling et al.

[11] **Patent Number:** 4,964,042

[45] **Date of Patent:** Oct. 16, 1990

[54] **STATIC DATAFLOW COMPUTER WITH A PLURALITY OF CONTROL STRUCTURES SIMULTANEOUSLY AND CONTINUOUSLY MONITORING FIRST AND SECOND COMMUNICATION CHANNELS**

[75] **Inventors:** Thomas L. Sterling, Crofton, Md.; Ellery Y. Chan, Melbourne, Fla.

[73] **Assignee:** Harris Corporation, Melbourne, Fla.

[21] **Appl. No.:** 231,673

[22] **Filed:** Aug. 12, 1988

[51] **Int. Cl.:** G06F 9/30; G06F 15/82

[52] **U.S. Cl.:** 364/200; 364/228.3; 364/232.22; 364/253; 364/260.2; 364/736

[58] **Field of Search** ... 364/200 MS File, 900 MS File, 364/736

[56] **References Cited**

U.S. PATENT DOCUMENTS

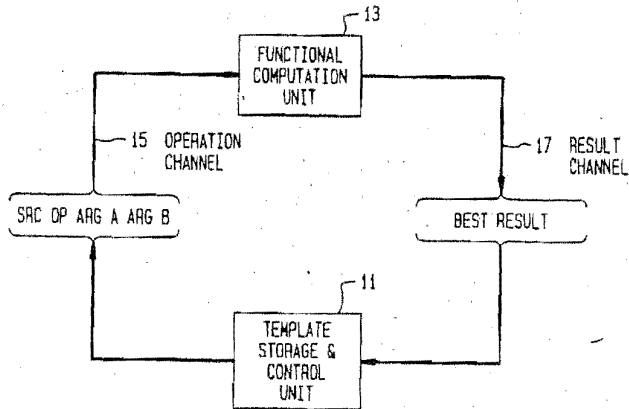
4,153,932	5/1979	Dennis et al.	364/200
4,379,326	4/1983	Anastas et al.	364/200
4,809,159	2/1989	Sowa	364/200
4,814,978	3/1989	Dennis	364/200
4,841,436	6/1989	Asano et al.	364/200
4,893,234	1/1990	Davidson et al.	364/200
4,901,274	2/1990	Maejima et al.	364/900

Primary Examiner—Thomas C. Lee
Attorney, Agent, or Firm—Evenson, Wands, Edwards, Lenahan & McKewon

[57] **ABSTRACT**

An associative architecture for a static data flow processing system comprises a functional computation unit in which data processing operations are executed, a data processing execution control structure (template) storage and control unit and communication channels through which the functional computation unit and the template storage and control unit communicate with one another. The template storage and control unit controls the supply of data to be processed by the functional computation unit and includes memory for storing a plurality of templates. Each template storage and control unit assembles data processing messages for application to a first of the communication channels for controlling the execution of a data processing operation by the functional computation unit. Each message contains the address of that template to which the result of the data processing operation is returned and stored in a return buffer, an opcode and either the data directly or the address of the template that contains the data to be processed by the functional computation unit. Each template also stores the status of a data processing execution cycle. Each template continuously monitors the communications channels for its address and, upon detecting its address, controllably interfaces prescribed information associated with the execution of a data processing operation with respect to the communication channels.

25 Claims, 22 Drawing Sheets



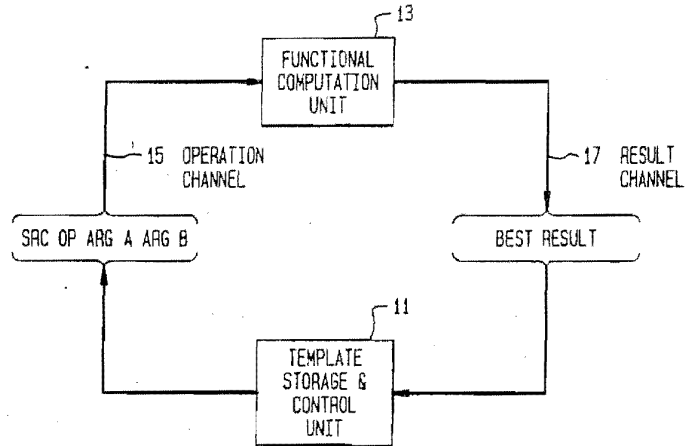


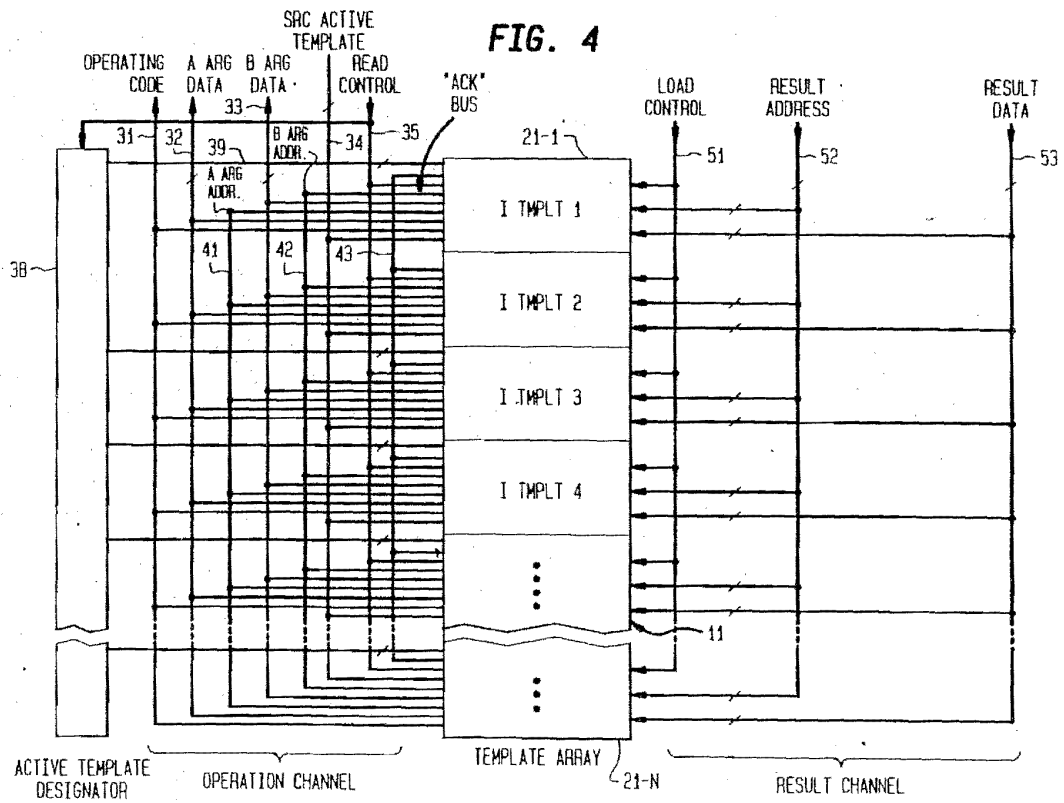
FIG. 1

LABEL/ADDRESS
TEMPLATE STATUS WORD
A ARG SOURCE
B ARG SOURCE
RESULT BUFFER

FIG. 2

OP CODE	A ARG AVL	B ARG AVL	ACK'S
---------	-----------	-----------	-------

FIG. 3



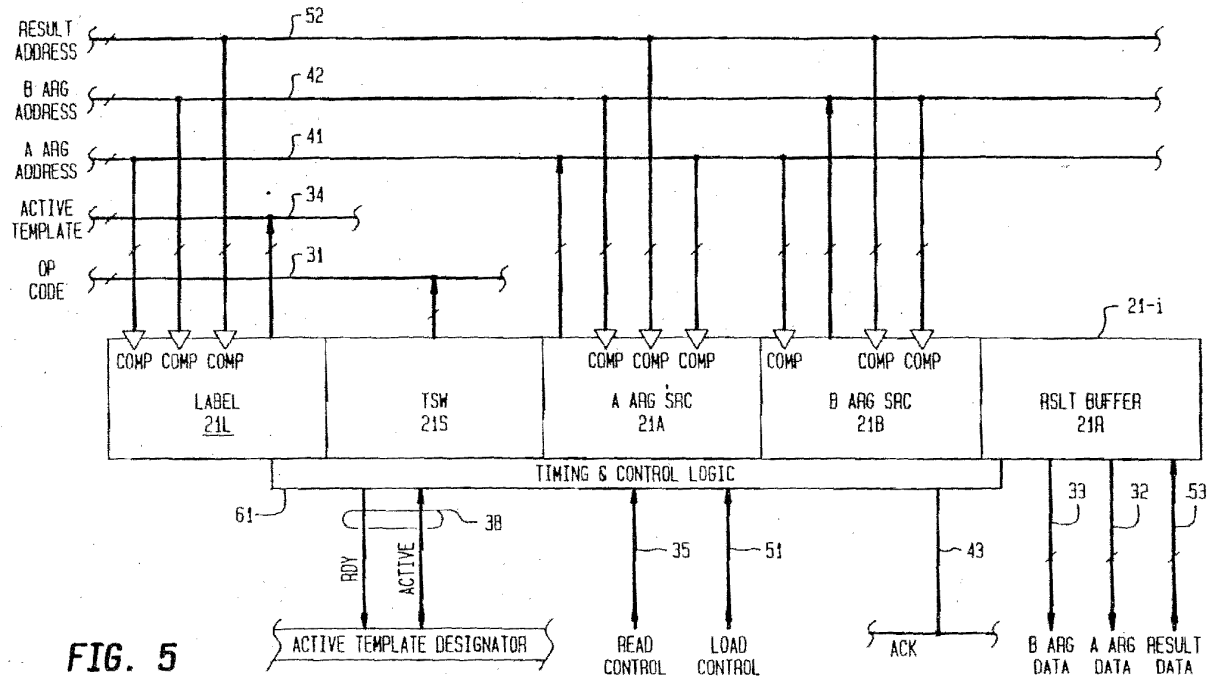


FIG. 5

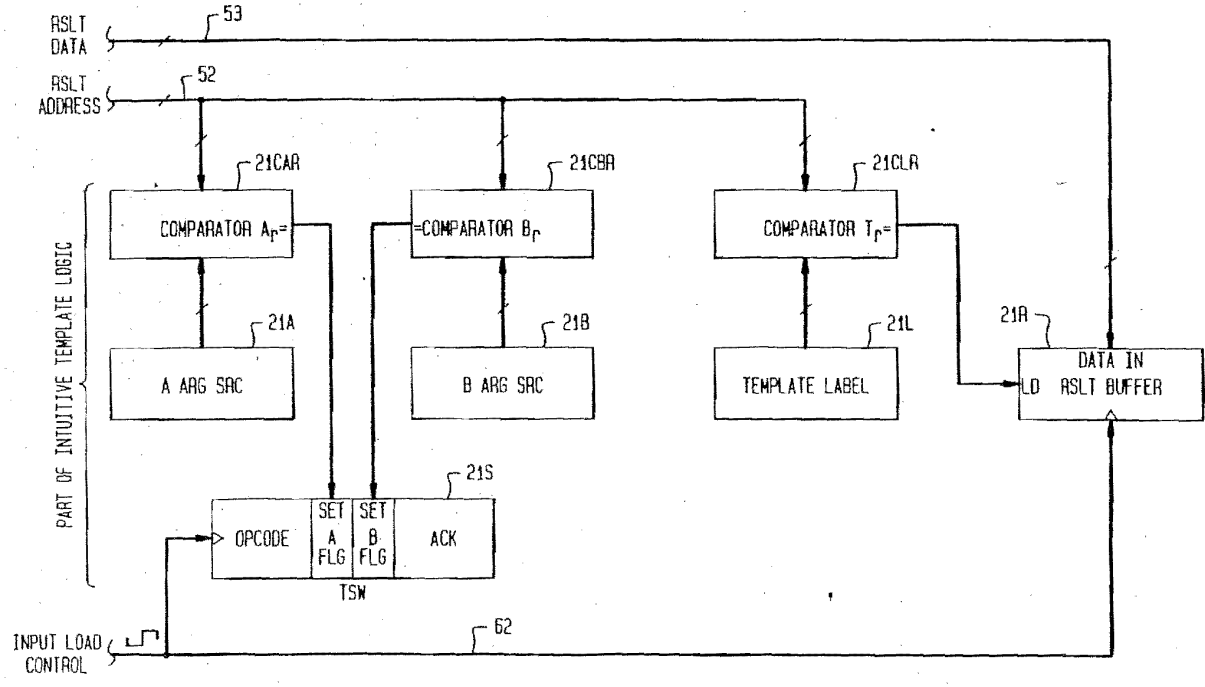


FIG. 6

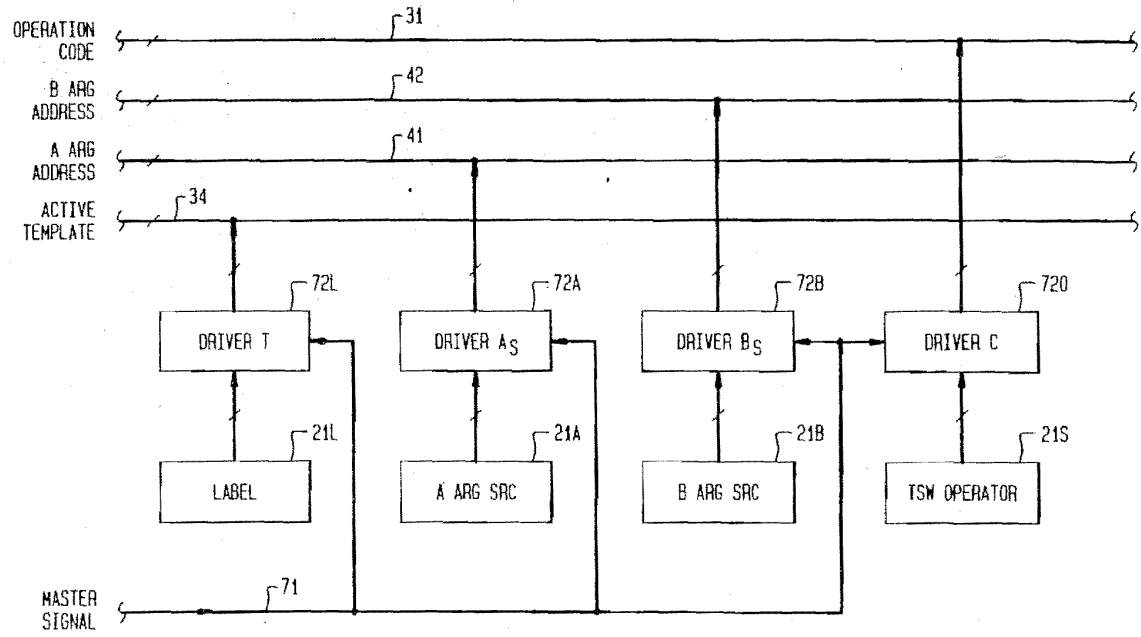


FIG. 7

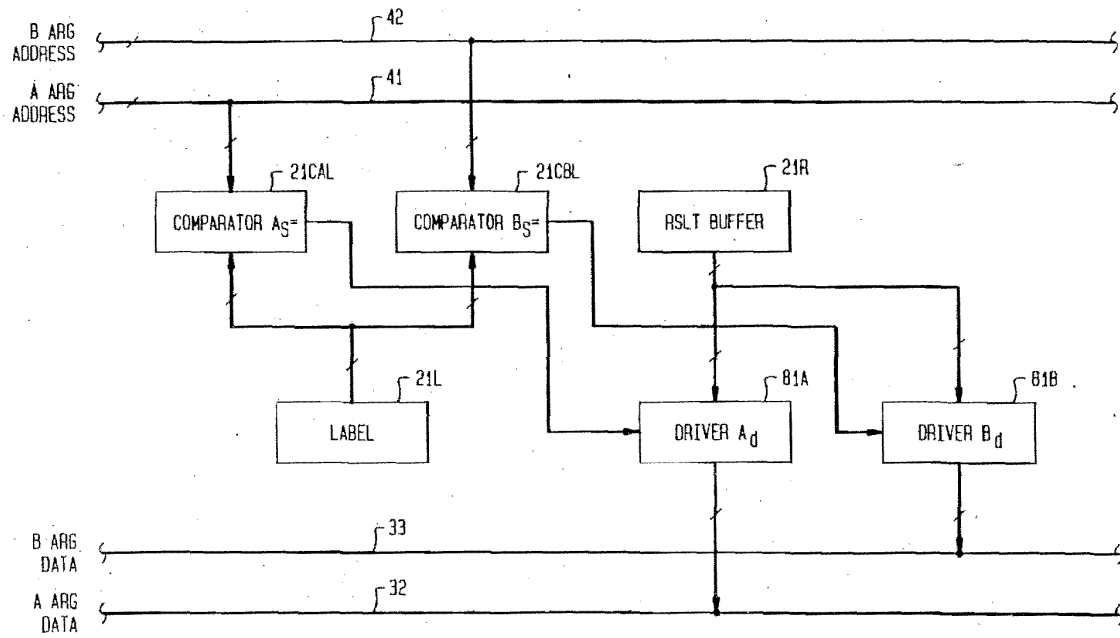


FIG. 8

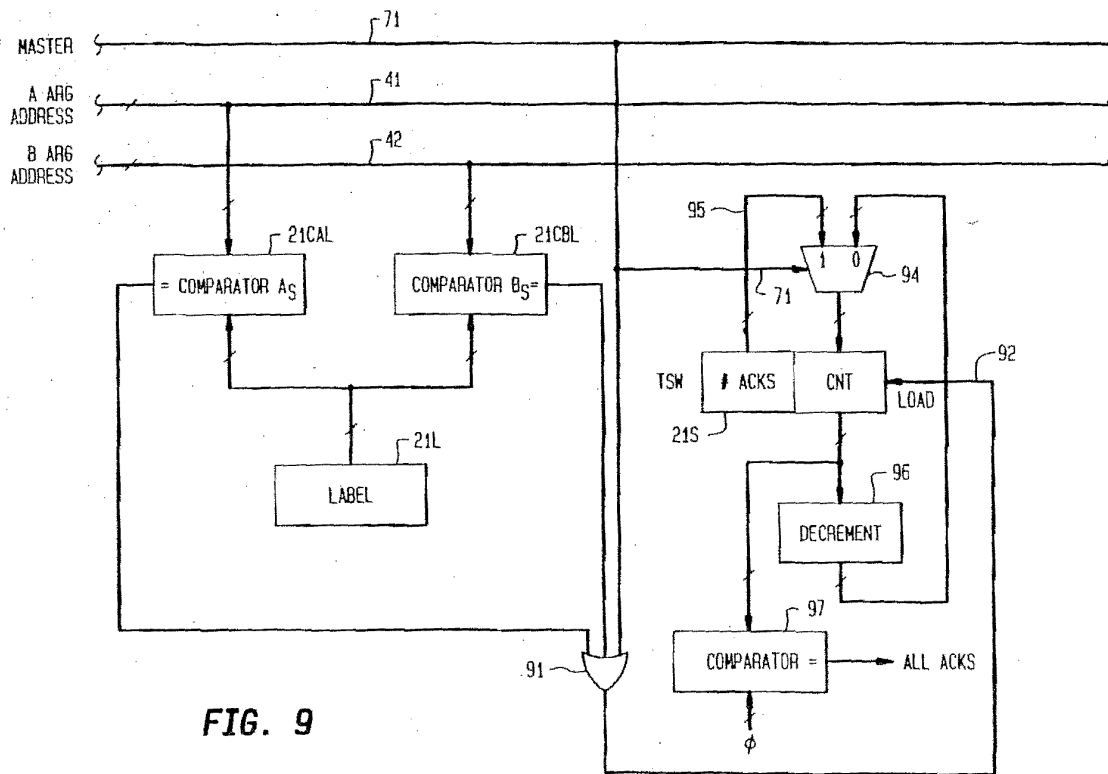


FIG. 9

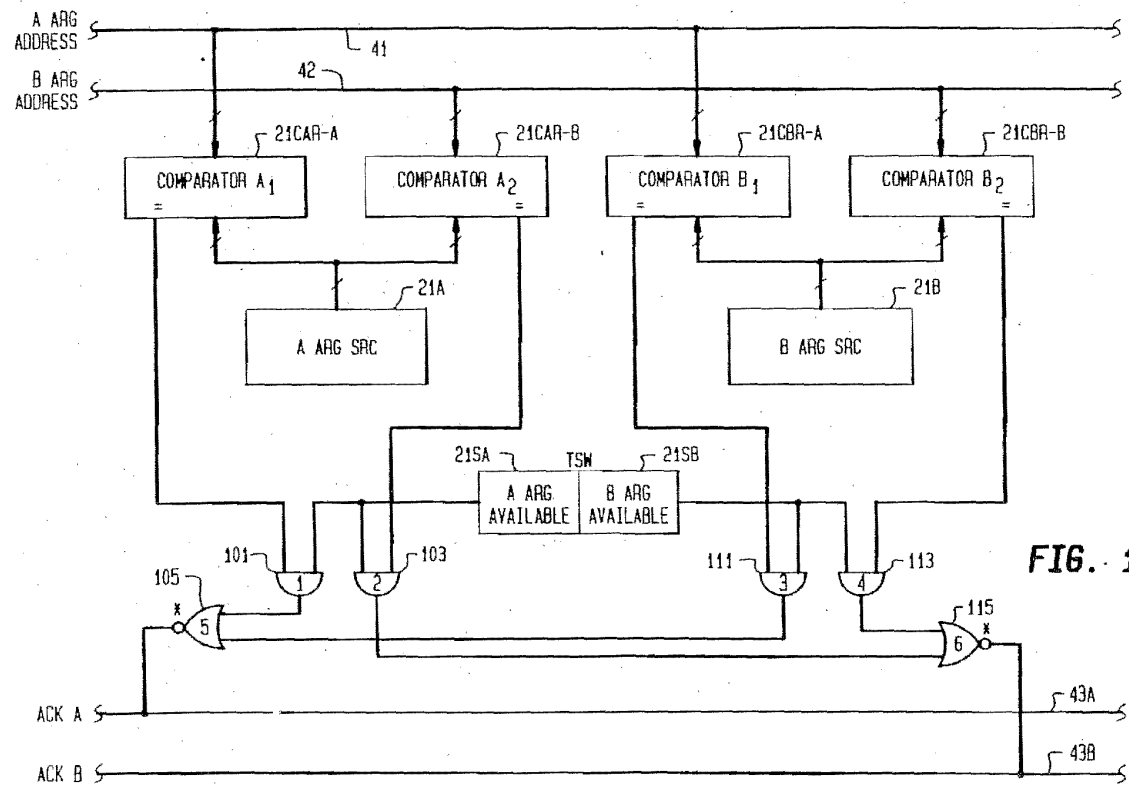


FIG. 10

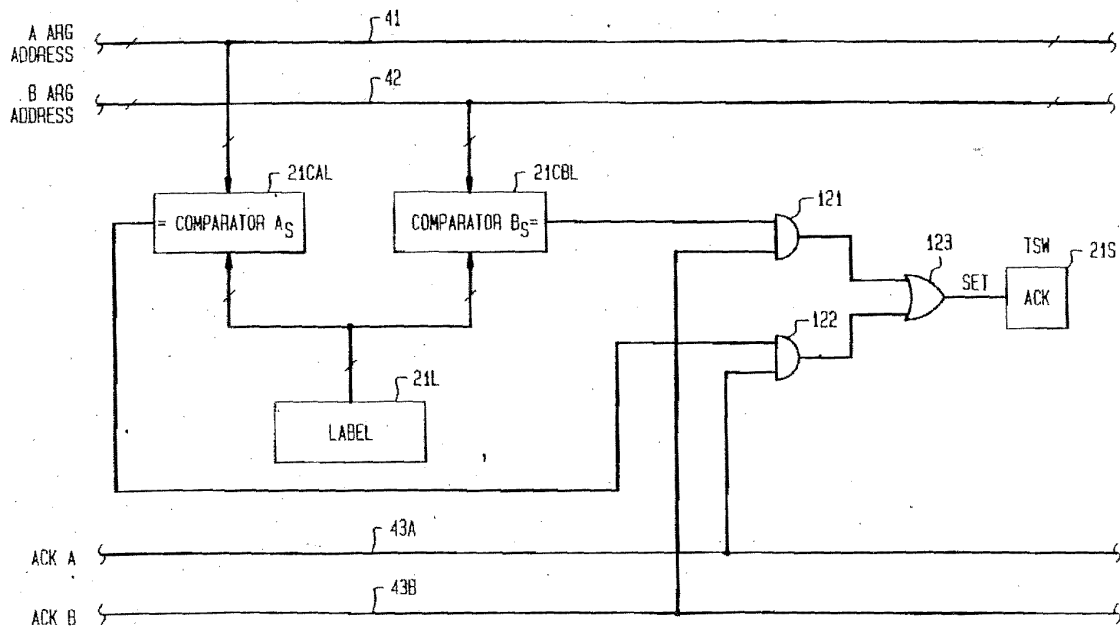
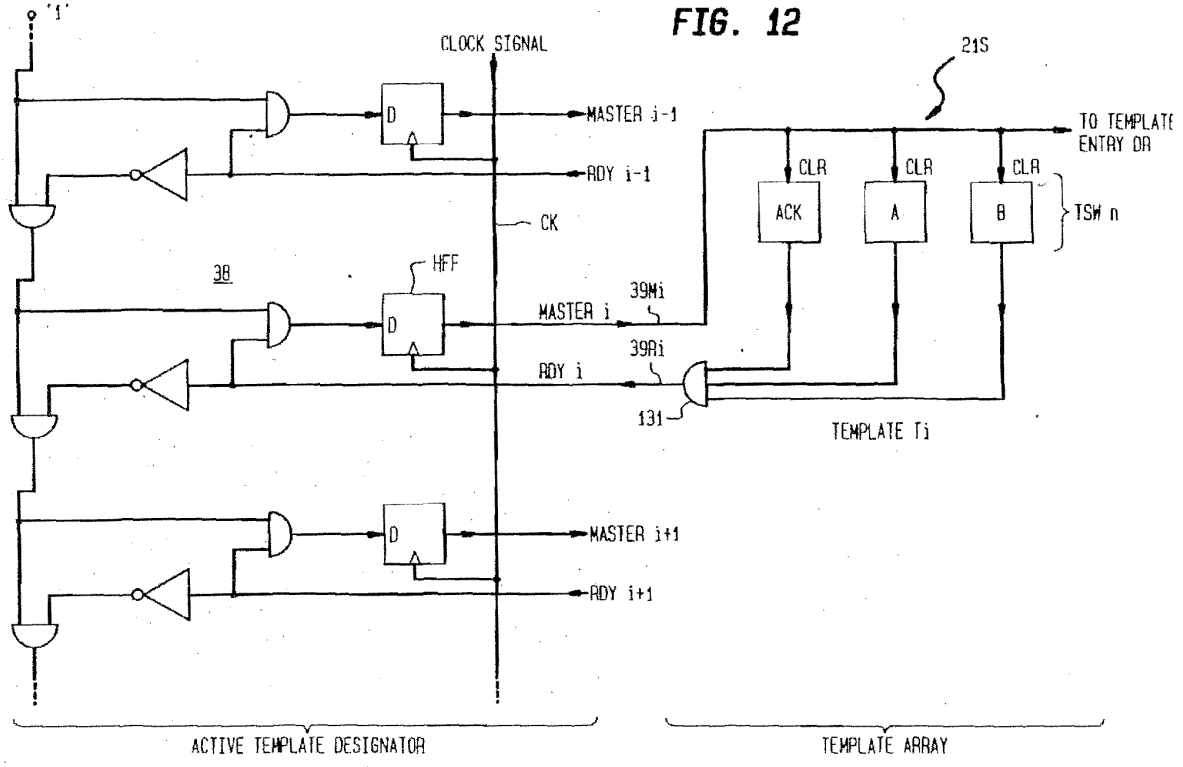


FIG. 11



U.S. Patent Oct. 16, 1990 Sheet 10 of 22 4,964,042

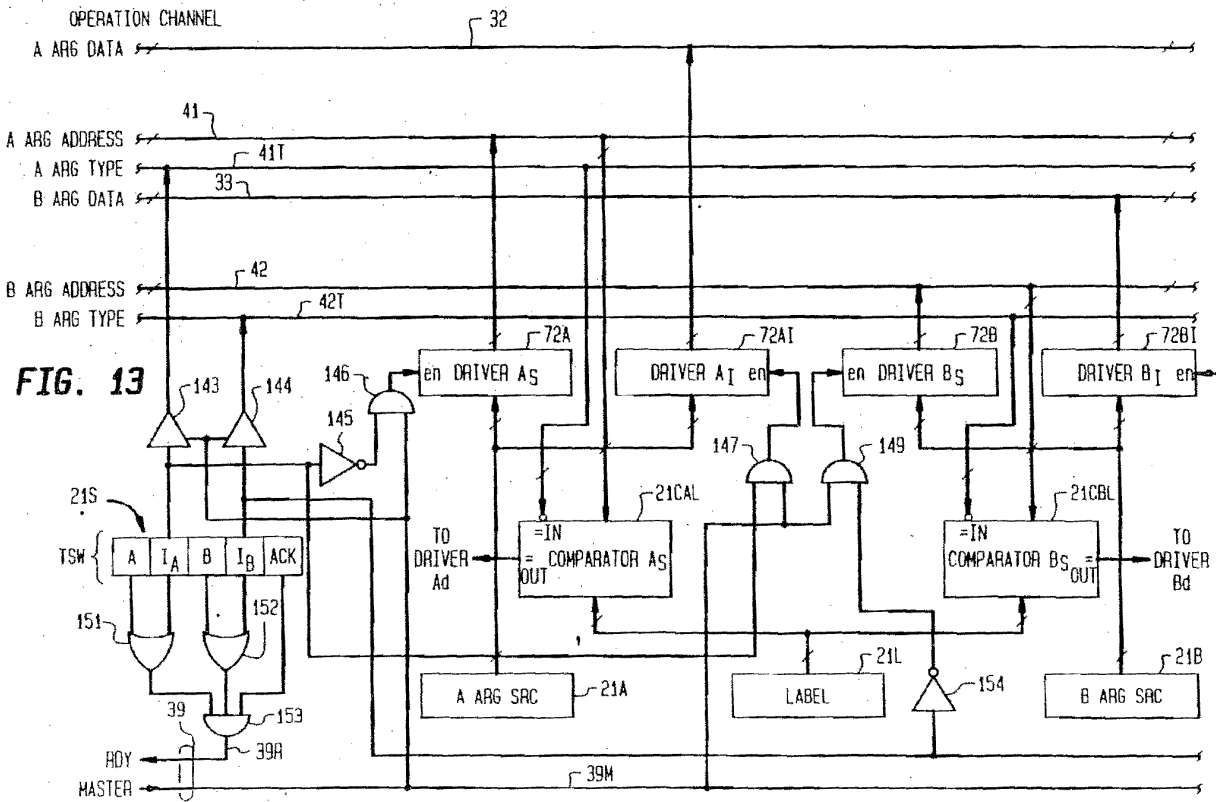


FIG. 13

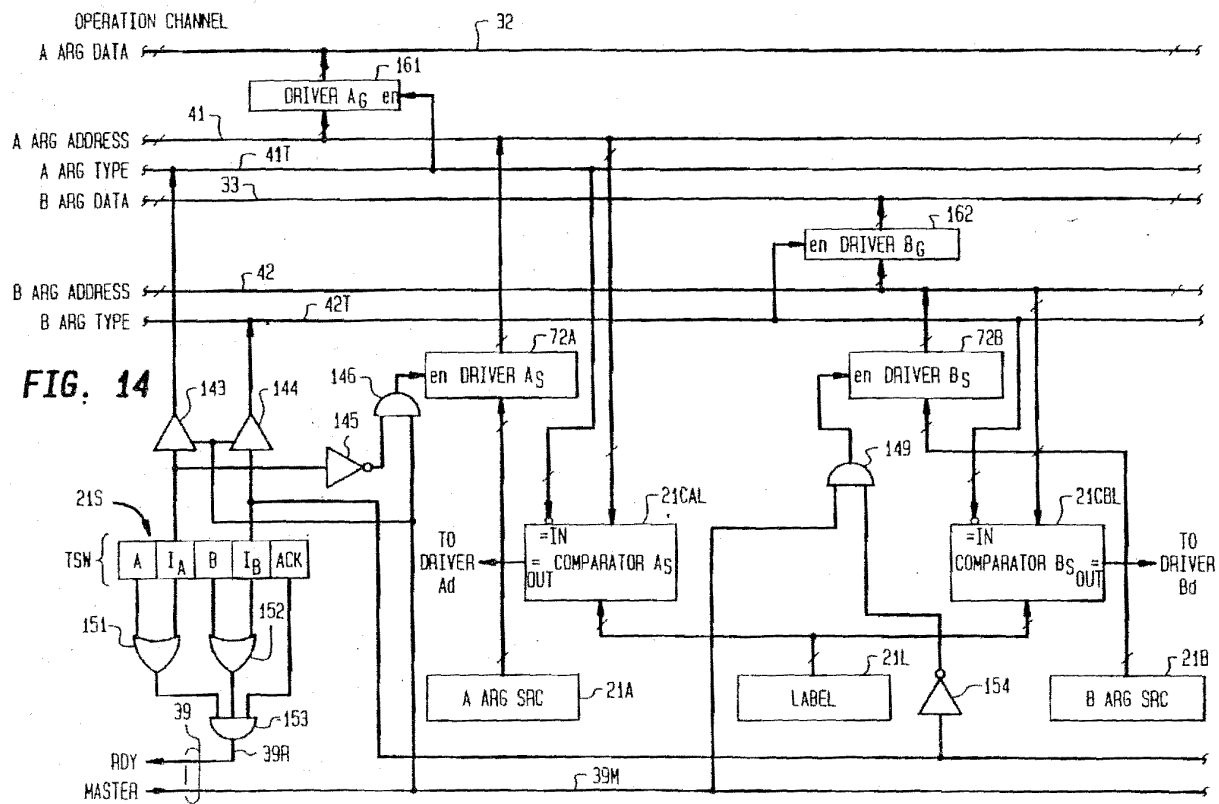


FIG. 14

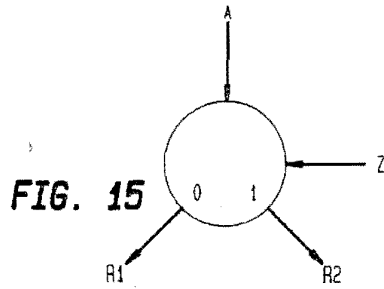
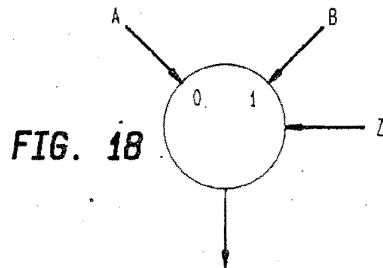
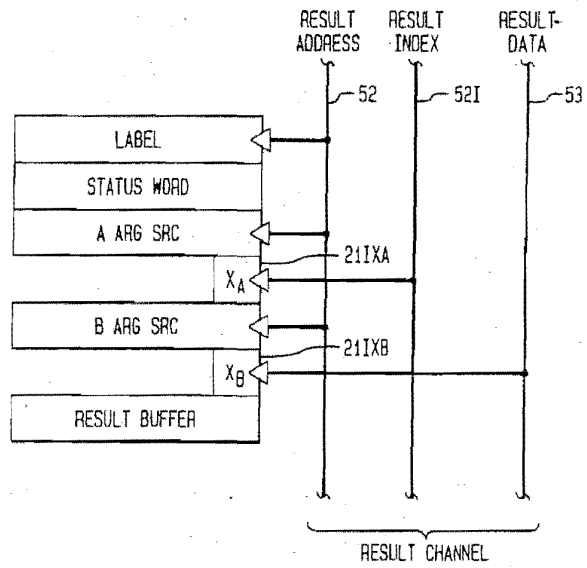


FIG. 16



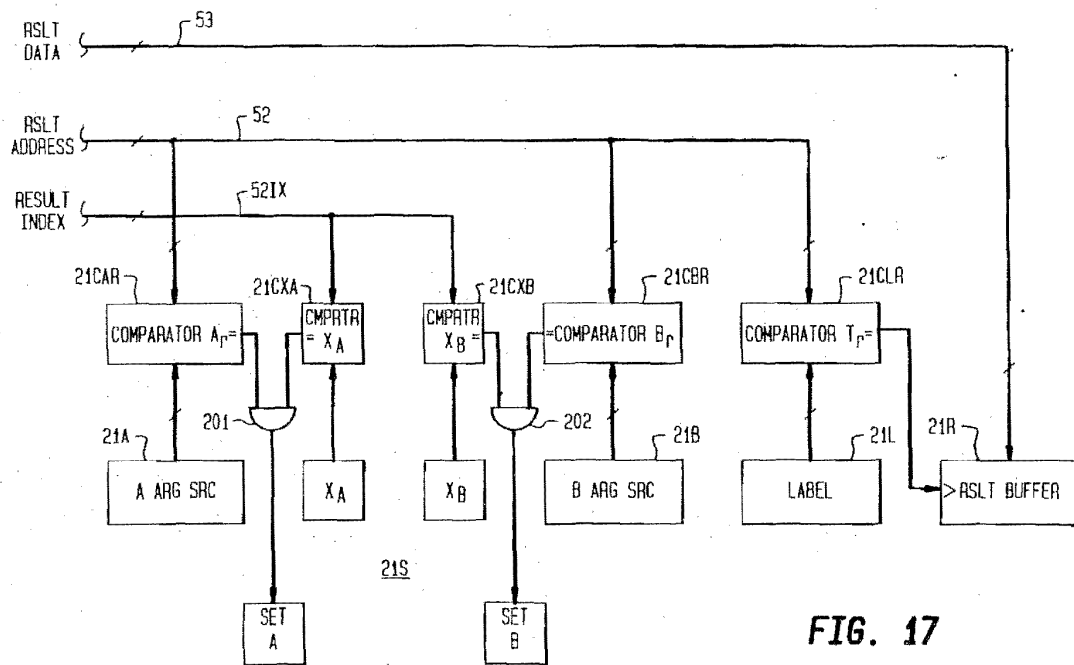
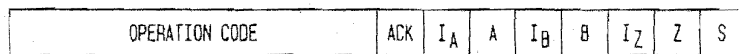
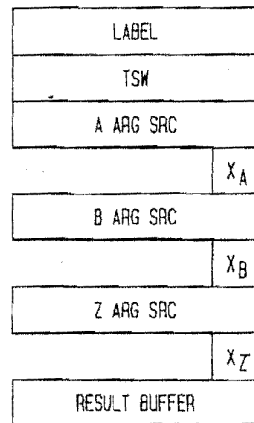
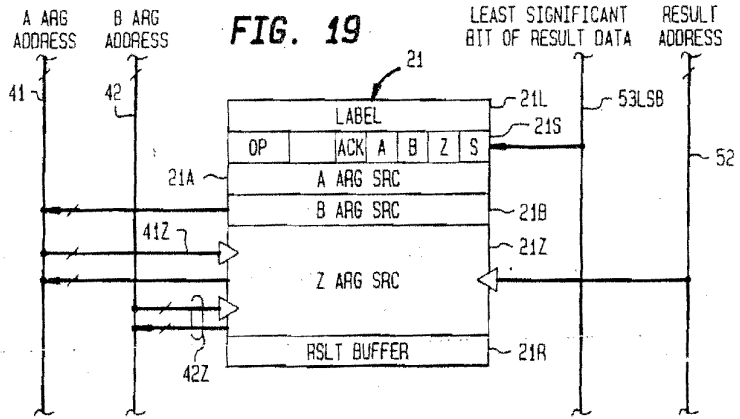


FIG. 17



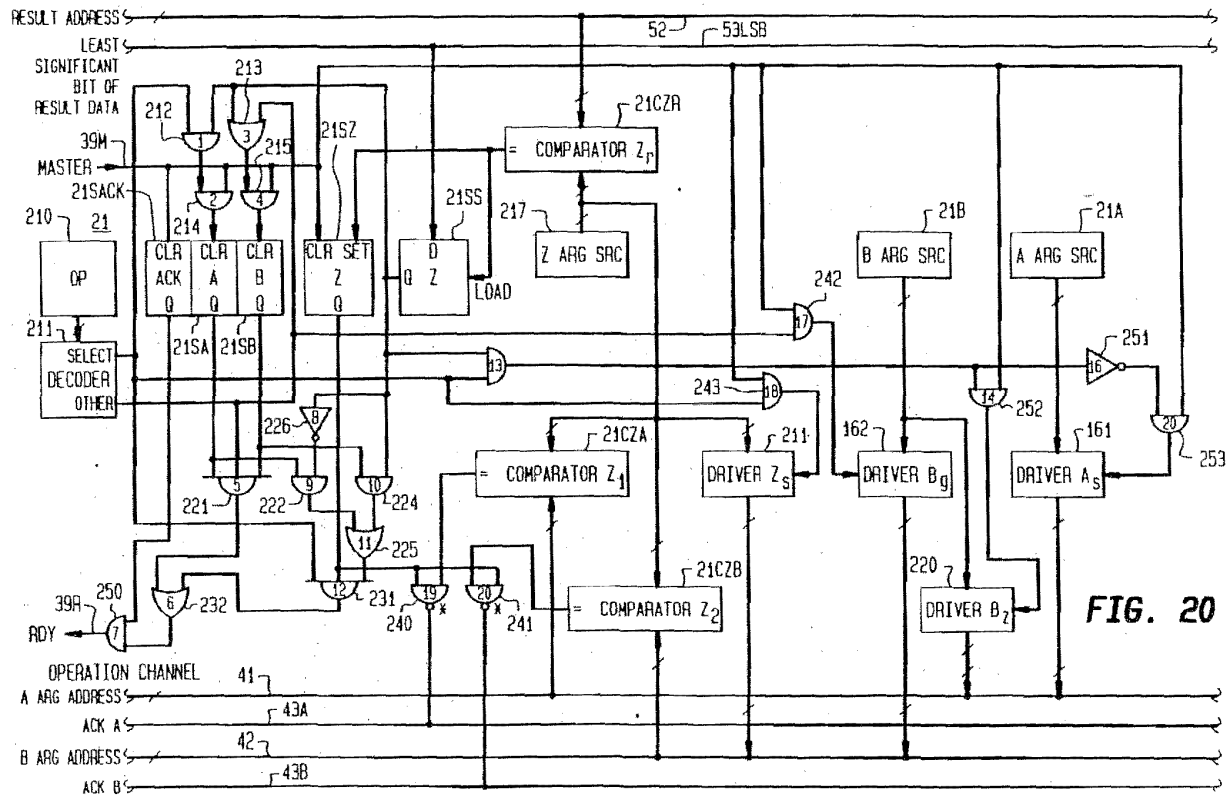


FIG. 20

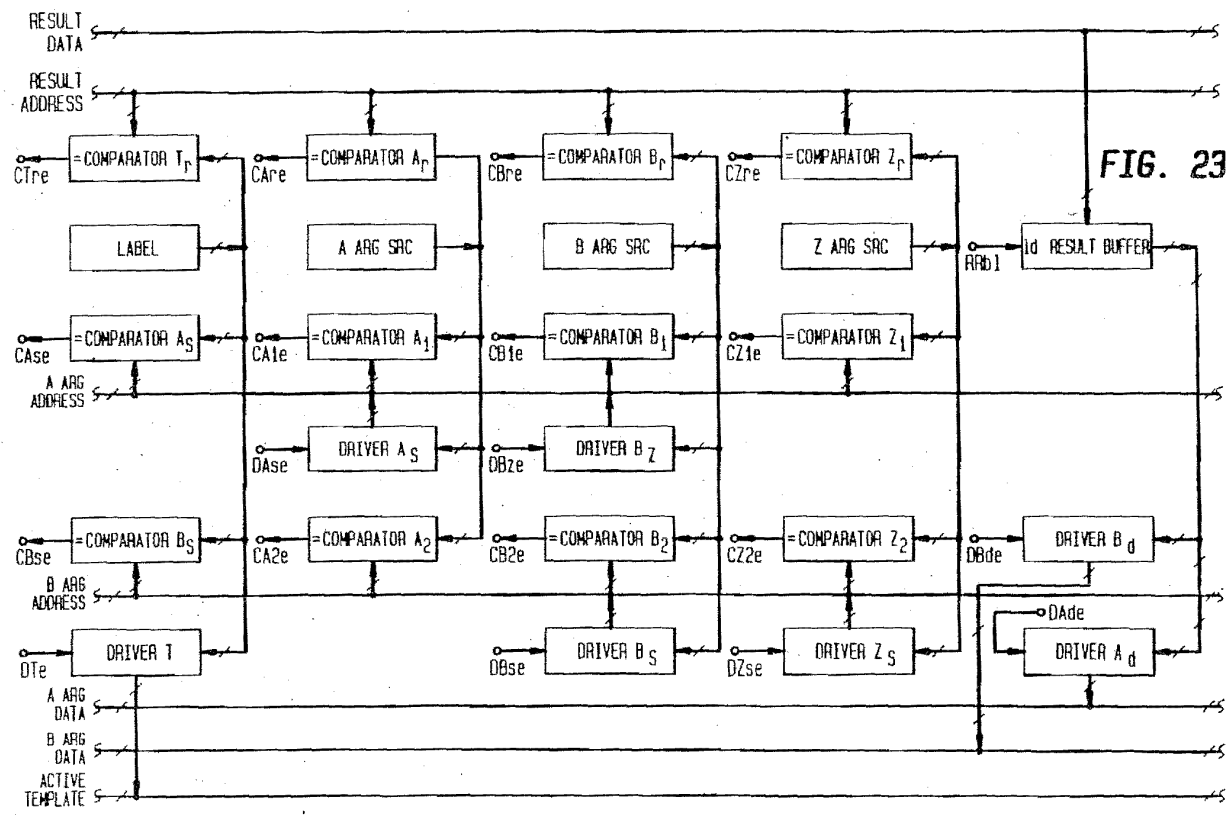


FIG. 23

U.S. Patent Oct. 16, 1990 Sheet 17 of 22 4,964,042

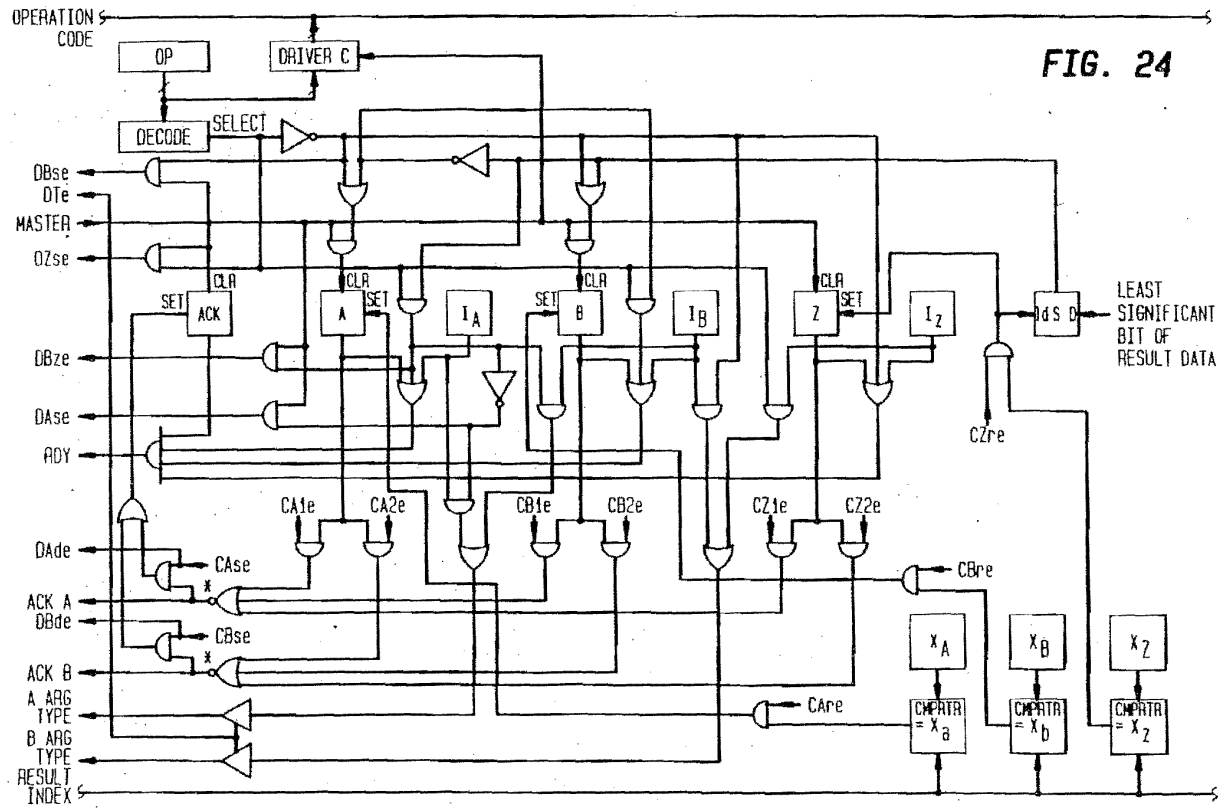


FIG. 24

U.S. Patent Oct. 16, 1990 Sheet 18 of 22 4,964,042

FIG. 25

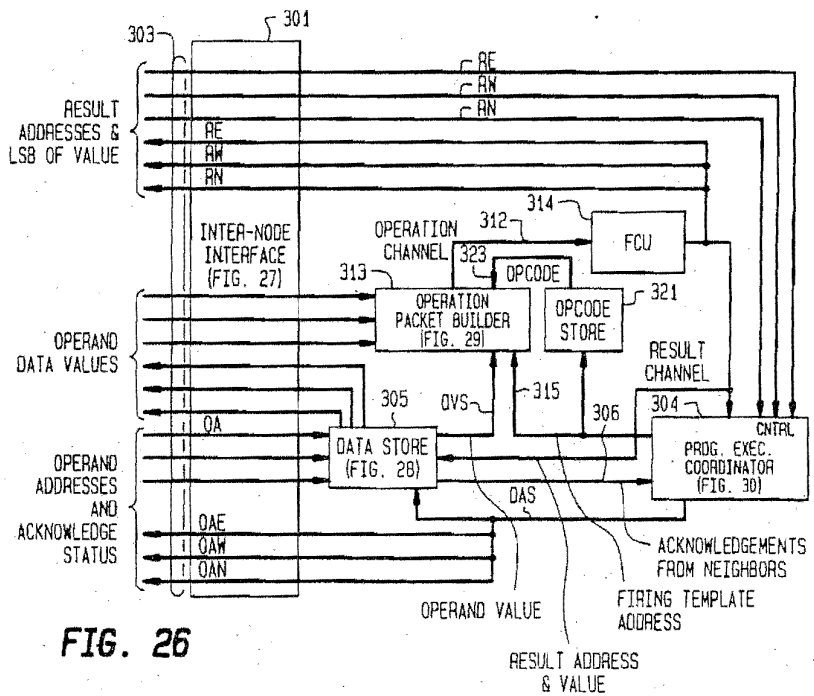
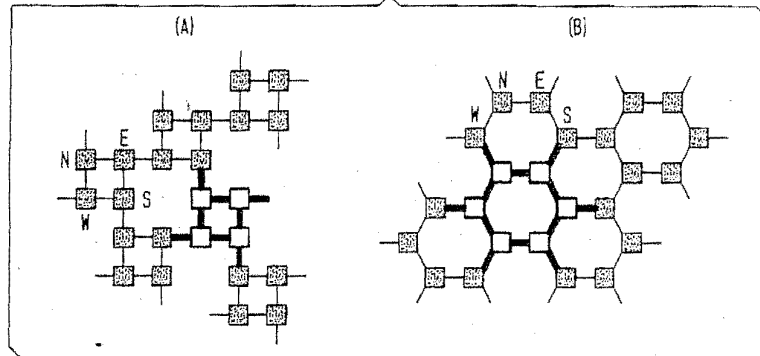


FIG. 26

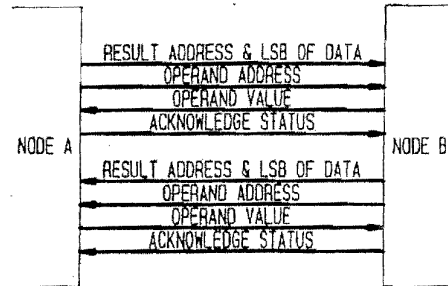


FIG. 27

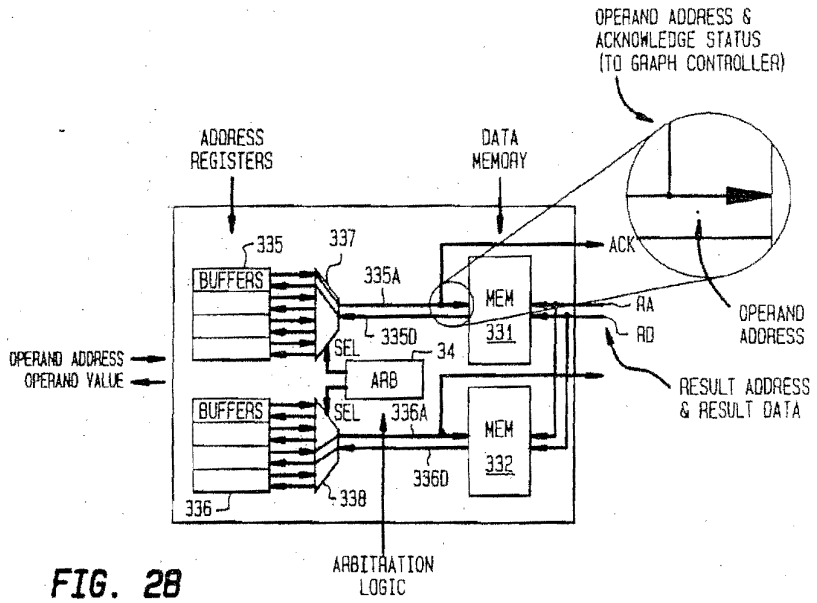
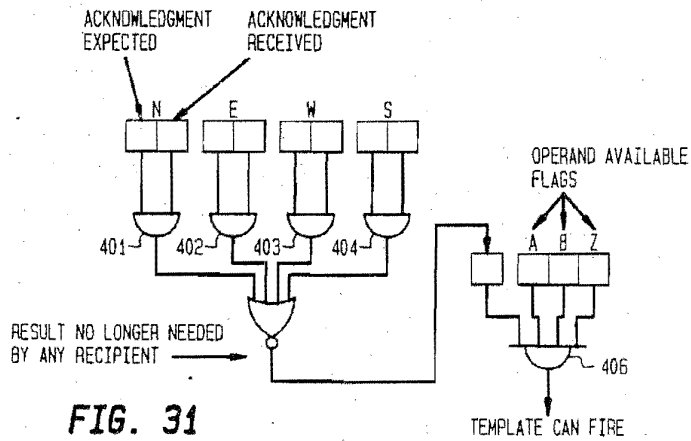
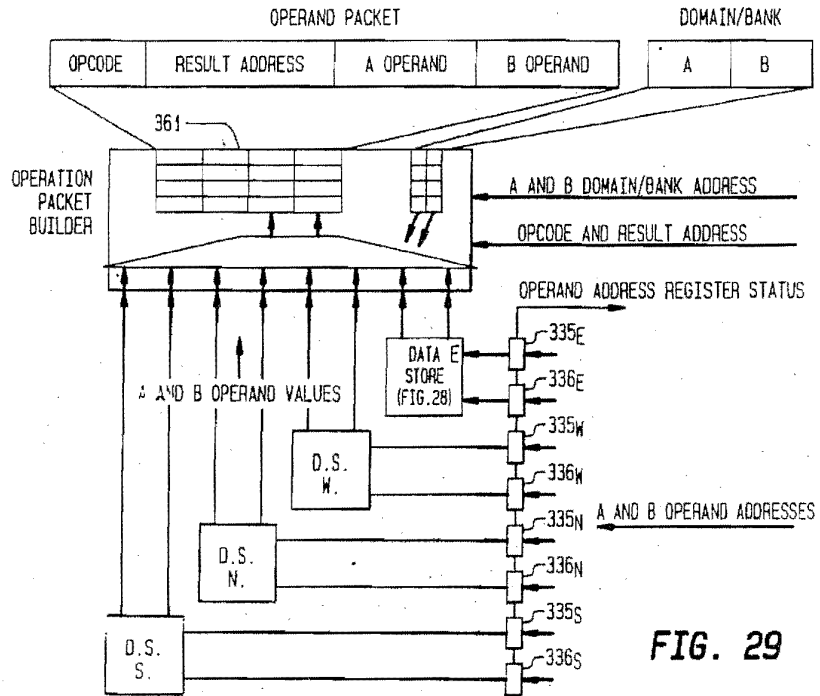


FIG. 28



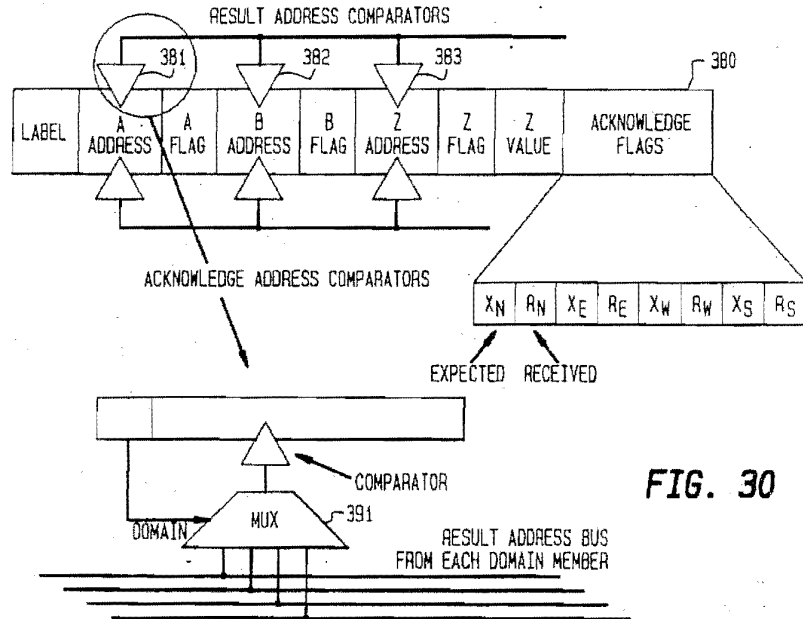


FIG. 30

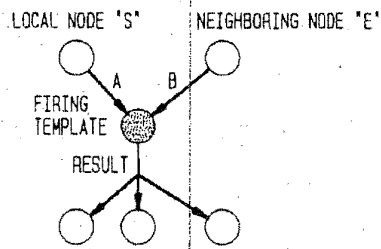


FIG. 32

**STATIC DATAFLOW COMPUTER WITH A
PLURALITY OF CONTROL STRUCTURES
SIMULTANEOUSLY AND CONTINUOUSLY
MONITORING FIRST AND SECOND
COMMUNICATION CHANNELS**

BACKGROUND OF THE INVENTION

The present invention relates in general to data processing systems and is particularly directed to a static dataflow computer architecture communications within which are effected through associative processing.

1. Field of the Invention

In static dataflow computer architectures, program execution is typically controlled by tokens, with directed information packets providing communication and synchronization among data execution control structures, or templates. Unfortunately, the amount of serial, temporal overhead required for a token-based processor to perform a single operation and the number of memory accesses per operation have effectively prevented static data flow computers from being employed for practical parallel data processing.

2. Summary of the Invention

In accordance with the present invention, the substantial temporal overhead and memory bandwidth requirements of token-based static data flow computer architectures are substantially reduced by replacing token-based processor communications with associative processing, similar to that used for associative memories, through which plural data execution control structures, or templates, of the system are interconnected with one another and with the data processing resources of the system, so that they may monitor and respond to operations carried out with respect to all other components of the system simultaneously, thereby increasing data processing execution speed and enhancing the efficient use of system memory.

For this purpose, in accordance with a first embodiment of the present invention involving a single processing node architecture, the static dataflow data processing system includes a functional computation unit, in which data processing operations are controllably executed, a template storage and control unit, and a pair of communication channels through which the functional computation unit and the template storage and control unit communicate with one another. The template storage and control mechanism controls the supply of data to be processed by the functional computation unit and includes memory for storing a plurality of templates. The template storage and control unit assembles data processing messages for application to a first of the communication channels for controlling the execution of a data processing operation by the functional computation unit. Each message contains first information representative of the identification of that template (its address) to which the result of the data processing operation is to be returned (via the second communication channel and stored in a return buffer within that template dedicated for the purpose), second information (an opcode) representative of the data processing operation to be performed by the functional computation unit, and third information representative of the data (either the data directly or the address of the template that contains the data) to be processed by the functional computation unit. Each template also stores the status of a data processing execution cycle. In accordance with the associative architecture of the present inven-

tion, each template is coupled to and continuously monitors the first and second communications channels for the presence of its address having been asserted thereon and, in response to detecting the presence of its address, controllably interfacing prescribed information associated with the execution of a data processing operation with respect to the first and second communication channels.

Each template monitors the first communications channel and asserts the contents of its return buffer onto the first communications channel in response to recognizing its address, so that the data stored in the return buffer may be employed as an operand for the execution of a data processing operation by the functional computation unit. In its status entry a template contains acknowledgement information representative of whether any other template requires the use of the contents of the return buffer. The status entry also includes operand availability information indicating whether the result entry of another template, whose address is defined by the contents of a source address entry of that template, contains an operand required for the execution of a data processing operation defined in accordance with opcode entry of that same template.

Each template also contains a code indicating its readiness to "fire", i.e. to have a data processing message asserted onto the first communications channel, in accordance with the contents of the status entry, and includes means for indicating the readiness of the template to have a data processing message asserted on the first communications channel in response to the acknowledgement information being representative that no other template requires the use of the contents of the current result entry of that template as an operand, and that the operand availability information indicates that all operands required for the execution of a data processing operation defined in accordance with opcode are available.

The template storage and control unit includes means for clearing the contents of the acknowledgement and operand availability information within the status entry of the template in the course of causing a data processing message associated with that template to be asserted onto the first communications channel.

The second communications channel includes a data portion over which output data from the functional computation unit is conveyed and a result address portion over which the address of an output data recipient template is conveyed. Each template includes a comparator for comparing its operand source address entries with the contents of the address portion of the second communications channel and controllably causes the operand availability information of the status entry to indicate that an operand entry required for the execution of a data processing operation defined in accordance with an opcode entry of the respective template is available in the result entry of another template whose address matches one of the operand source address entries of the respective template.

The second communications channel also includes a result index portion for identifying one of the operand source entries of a template, and the comparator includes means for causing the operand availability information of the status entry to indicate that an operand entry required for the execution of a data processing operation defined in accordance with an opcode entry of that template is available in the result entry of an-

other template whose address matches the operand source address entry of the respective template as identified by the result index portion.

The first communications channel includes a data portion over which operands are conveyed, an address portion over which the address of a selected template is conveyed, an opcode portion over which the opcode entry of a selected template is conveyed, and an intra template address link over which operand source addresses are conveyable among the templates of said storage unit. Each template includes means for comparing its address with the contents of the intra template address link and causing the contents of its result entry to be asserted onto the data portion of the first communications channel, in response to detecting a match between its address and the contents of the intra template address link.

A respective template includes means for controllably asserting its operand source addresses onto the intra template address link in the course of the assertion of a data processing message, and the second communications channel includes a data portion over which output data is conveyed and a result address portion over which the address of an output data recipient template is conveyed. A template also includes means for controllably causing the operand address asserting means to assert an operand source address onto the intra template address link in accordance with the contents of the address portion of the second communications channel. A selected operand source address is asserted onto the intra template address link in accordance with the contents of a prescribed (least significant bit) portion of the data portion of the second communications channel.

In accordance with a second embodiment of the present invention, the associative template-based data processing mechanism is applied to a larger, system level architecture, comprised of multiple nodes, each having its own dedicated functional computation unit and template storage facility, wherein operand and result data are exchanged among the nodes of the system. The nodes preferably form a mesh topology, in which each node is connected with and may communicate with some number, e.g. three, nearest neighbor nodes with which it shares data resources in the course of execution of its own data processing operations and also in the course of the execution of data processing operations by those neighboring nodes. Namely, within an individual node, a data processing operation defined by a template stored within that node is always executed by the functional computation unit within that node. However, the operands required for and the results of that execution may be shared by templates in nearest neighbor nodes. In order to effect this sharing of data resources, the architecture of each node is configured to provide an inter-node associative communication capability, similar to that of an individual node, for those aspects of a template which may depend upon or be necessary for the execution of a template in any of its neighboring nodes, by assigning associative communication control functions to dedicated storage and supervisory units within each node.

For this purpose, the multi-node configuration of the associative data processing architecture of the present invention comprises a plurality of data processing nodes each of which includes its own dedicated functional computation unit and a program execution control unit which contains a plurality of templates, each template comprising a plurality of entries, including an address

for identifying that template, a plurality of operand source entries for specifying the addresses of operands to be employed in the execution of a data processing operation associated with that template, and the status of the template with respect to its associated data processing operation. Each node also contains an opcode store, coupled to the program execution control unit, for storing a plurality of opcodes respectively associated with the plurality of templates, and an opcode which defines a data processing operation to be performed by the functional computation unit. Also included within each node is an operand store, which is coupled to the program execution control unit, for storing a plurality of result entries in which output data produced by the functional computation unit as a result of its execution of a data processing operation requested by a template are stored.

Assembly of a data processing message is carried out by an operation packet builder, which is coupled to the program execution control unit, the opcode store and the operand store. The operation packet builder assembles a plurality of data processing messages to be forwarded to the functional computation unit for execution, a respective data processing message including the identification of a respective template, the contents of respective result entries identified by operand source addresses of said respective template, and the opcode associated with said respective template.

A first communications channel is coupled between the operation packet builder and the functional computation unit for conveying data processing request messages between the packet builder and the functional computation unit. A second communications channel is coupled between the functional computation unit, the program execution control unit, storage unit and the operand store, for conveying output data from the functional computation unit to the operand store and the identification of the template for which a data processing request message has been processed by the functional computation unit to the program execution control unit.

A first internode communication channel is coupled to the second communication channel of each node, for coupling the identification of the template for which a data processing request message has been processed by its associated functional computation unit to the program execution control unit in each node. A second internode communication channel is coupled to the operand store, the operation packet builder, and the program execution control unit of each node, for enabling the operand addresses of a template stored within the program execution control unit of a node to be presented to the operand store of every other adjacent node, and for enabling operand values stored in any node to be presented to the operation packet builder of any node.

The status entry of a respective template includes operand availability information representative of whether the result entry of another template in any neighboring node, whose address is defined by the contents of a source address entry of the template, contains an operand required for the execution of a data processing operation defined in accordance with opcode entry of the template.

The second communications channel includes a data portion over which output data from the functional computation unit is conveyed and a result address portion over which the address of said respective template

is conveyed, and the program execution control unit includes a comparator for comparing the operand source entries of the template with the contents of the address portion of the second communications channel and causes the operand availability information of the status entry to indicate that an operand required for the execution of a data processing operation defined in accordance with an opcode associated with the template is available in the operand store of that node which contains the template whose identification matches one of the operand source address entries of the template.

The second communications channel further includes a result index portion for identifying one of the operand entries of a template and the comparator outputs a signal which causes the operand availability information of the status entry to indicate that an operand entry required for the execution of a data processing operation defined in accordance with an opcode entry of the template is available in the operand store of a node containing the template whose address matches the operand source address entry of the template as identified by the result index portion.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 diagrammatically illustrates the general architecture of an associative template dataflow processing system in accordance with the present invention;

FIG. 2 diagrammatically illustrates the contents of a respective template stored within a storage and control unit of FIG. 1;

FIG. 3 shows the sub-entries within a template status word;

FIG. 4 diagrammatically illustrates the general organization of the system template storage and control unit architecture of FIG. 1;

FIG. 5 illustrates the interfacing of the respective bus links of the architecture of FIG. 4 with the respective entries of an individual template within a storage and control unit;

FIG. 6 shows circuitry within a template for handling result data and address signals;

FIG. 7 diagrammatically illustrates the operation of a 'master' template;

FIG. 8 diagrammatically illustrates the operation of an operand-supplying template;

FIG. 9 illustrates a circuit for modifying the template status word field through the use of a counter to track the number of dependent templates, remaining to be asserted;

FIGS. 10 and 11 diagrammatically illustrate a mechanism for generating an acknowledgement through the use of a dual argument acknowledgement bus and status word field;

FIG. 12 shows the configuration of a designator circuit for detecting whether a template is ready to be asserted and for selecting which ready template is the next in line to be asserted;

FIG. 13 diagrammatically shows a single level scheme, as a modification of the operand generation mechanism of FIG. 7, for supporting the use of immediate arguments;

FIG. 14 shows a dual level modification of the operand generation mechanism of FIG. 7, for handling both direct addressing and immediate type arguments;

FIG. 15 diagrammatically shows a switch template for initiating the execution of a data processing operation in which a primary data operand is made available

to one of two sets of prescribed recipient templates depending on the value of a control Boolean operand; FIG. 16 illustrates a switch template;

FIG. 17 illustrates the configuration of modified signal processing hardware within the template necessary to support a switch function;

FIG. 18 diagrammatically illustrates a select template for initiating the execution of a data processing operation in which one of its two data operands is made available, depending on the value of a third, control Boolean operand;

FIG. 19 illustrates a select template;

FIG. 20 shows signal processing logic for enabling a template to perform the select function;

FIGS. 21 and 22 diagrammatically show expanded template fields and template word status entries;

FIGS. 23 and 24 show the overall comparator and driver circuitry and their associated communication buses, together with combined control logic for implementing an associative template;

FIG. 25, illustrates an exemplary mesh topology of a multiple node architecture;

FIG. 26 diagrammatically illustrates the architecture of an individual node of a multi-node processor architecture;

FIG. 27 shows the bus structure of an inter-node communications link;

FIG. 28 diagrammatically illustrates the configuration of a data store;

FIG. 29 shows the configuration of an operation packet builder;

FIG. 30 diagrammatically illustrates the respective fields of an individual template stored within the program execution coordinator and the mechanism through which the program execution coordinator monitors the result address link from its local node and those of neighbor nodes for setting the A, B and Z flags; is generated.

FIG. 31 illustrates circuitry for handling acknowledgements; and

FIG. 32 diagrammatically illustrates a multi-node data flow operation.

DETAILED DESCRIPTION

Before describing in detail the particular improved computer architecture in accordance with the present invention, it should be observed that the present invention resides primarily in a novel structural combination of conventional signal processing and communication circuits and components and not in the particular detailed configurations thereof. Accordingly, the structure, control and arrangement of these conventional circuits and components have been illustrated in the drawings by readily understandable block diagrams which show only those specific details that are pertinent to the present invention, so as not to obscure the disclosure with structural details which will be readily apparent to those skilled in the art having the benefit of the description herein. Thus, the block diagram illustrations of the Figures do not necessarily represent the mechanical structural arrangement of the exemplary system, but are primarily intended to illustrate the major structural components of the system in a convenient functional grouping, whereby the present invention may be more readily understood.

Referring now to FIG. 1, the general architecture of a single node associative template dataflow processing system in accordance with the present invention is

shown as comprising a pair of operational (storage/control and execution) units 11 and 13 linked with one another by way of a pair of communication paths 15 and 17. By single node architecture is meant that all data processing operations with the system are executed within the confines of a self-contained node or processing unit, as contrasted with a multi-node environment, to be described infra, where multiple execution units have their own computational capabilities and share data resources through an inter-node communications architecture.

Within the single node system of FIG. 1, a data storage and control unit 11 and a functional computation unit 13 are mutually linked by way of an operation channel 15 and a result channel 17. Data storage and control unit 11 contains memory and associated control logic for storing and controllably interfacing a plurality of data processing execution control structures, termed templates, with each of operation channel 15 and result channel 17. A principal responsibility of unit 11 is the control of the presentation or transmission of data processing messages awaiting service in the templates to functional computation unit 13 over operation channel 15. Functional computation unit 13 performs arithmetic and logical operations on one and two argument value sets that are contained within data processing execution messages supplied over operation channel 15 from storage and control unit 11 and forwards the result of its data processing operation over result channel 17 to template storage and control unit 11.

The contents of a respective template, stored within template storage and control unit 11, are diagrammatically illustrated in FIG. 2 as a set of table entries comprising: an address, or label, (L) identifying the template and employed by various portions of the system to address that template; a status word (TSW), which contains a number of sub-entries (shown in FIG. 3) representative of the operational/control status of the template; a pair of argument value entries (A arg src and B arg src) corresponding to the labels or addresses of those templates within unit 11 from which the actual argument values of a data processing message are to be obtained; and a result buffer entry in which the result of a data processing operation executed by the functional computation unit is stored.

As shown in FIG. 3, the sub-entries within the template status word comprise an opcode, representative of the data processing operation to be performed, a pair of argument available flags (A arg avl and B arg avl), which indicate whether or not the respective A and B argument values to be employed in the data processing operation are currently resident within the templates whose labels correspond to the A arg src and B arg src entries, referenced above, and a set of acknowledgement flags representative of the status of other templates that use the contents of the result buffer as an argument value. As will be explained in detail below, the contents of the TSW field effectively determine whether or not the entries within the template are complete, so that a data processing message may be assembled and placed on the operation channel to be processed by functional computation unit 13.

FIG. 4 diagrammatically illustrates, in greater detail, the general organization of the system architecture of FIG. 1, referenced above. Storage and control unit 11 is shown as being comprised of a plurality or array of N templates, 21-1 . . . 21-N, which are interfaced with functional computation unit 13 through operation chan-

nel 15 and result channel 17, each of which is comprised of a multiple bus structure, as shown. Specifically, operation channel 15 contains a set of inter-unit buses including an opcode bus 31, an A argument data bus 32, a B argument data bus 33, an active template bus 34 and a read control bus 35, and a set of intra-unit buses, including an A argument address bus 41, a B argument address bus 42 and an acknowledge bus 43. Opcode bus 31 carries the opcode portion of a data processing message derived from the TSW field of one of templates 21-1 . . . 21-N which is currently invoking the execution of a data processing operation, while buses 32 and 33 carry the A and B argument values obtained from the source templates (A src and B src) specified by the A and B argument source entries of the template. Bus 34 indicates which template is currently active (having a data processing request serviced by functional computation unit 13), and bus 35 is used to provide timing and control signals to manage and synchronize the transmission of messages between template storage and control unit 11 and functional computation unit 13. For this purpose, template storage and control unit 11 includes an arbitration logic circuit, or active template designator 38, which is controlled by the timing signals on bus 35 and controllably designates, via one of links 39-1 . . . 39-N, which template is currently 'active'.

Within the set of intra-unit buses, A argument address bus 41 and B argument address bus 42 are employed by the active template to identify which templates contain the respective A and B argument values (operands) that are to be asserted on A and B argument data buses 32 and 33 during the forwarding of a data processing message to functional computation unit 13. Acknowledge bus 43 serves to propagate control/status information among the templates in the course of establishing whether a template is ready to be asserted.

Result channel 17 is comprised of a load control bus 51, a result address bus 52 and a result data bus 53. Load control bus 51 provides timing signals that direct the monitoring of the address and data buses and the loading of the data by the templates. Result address bus 52 contains the address or label of the template that initiated the data processing operation and to which the result of that operation is to be returned, while data bus 53 contains the actual result data that has produced by functional computation unit 13 and which is to be written into the result buffer of the initiating template.

FIG. 5 illustrates, in greater detail, the interfacing of the respective bus links of the architecture of FIG. 4 with the respective entries of an individual template 21-i within storage and control unit 11. Also shown is a timing and control logic circuit 61 that controls the storage and readout of the contents of the respective fields of the template. Within the intra-unit bus portion of operation channel 15, the A and B argument address buses 41 and 42 are coupled as inputs to a template label or address field 21L, so that their contents may be compared with the identity of the template, and, thereby determine whether or not the contents of that template's result buffer 21R are to be asserted onto either of A argument or B argument buses 32 and 33, respectively, which are connected as result buffer output links, as shown. A and B argument address buses 41 and 42 are also coupled as inputs to both A argument and B argument source fields 21A and 21B, respectively, so that their contents may be compared with the stored A and B argument source fields for purposes of handling an acknowledgement, as will be described infra. Ac-

knowledge bus 43 is coupled to timing logic and control logic circuit 61. Under the control of read control bus 35, buses 41 and 42 are also respectively coupled to receive the contents A and B argument source fields 21A and 21B, so that the contents of fields 21A and 21B may be asserted onto the respective A and B argument address buses.

Within the inter-unit portion of operation channel 15, the result address bus 52 is coupled as an input to template label field 21L, so that its contents may be compared with the identity of the template, and thereby determine whether or not the contents of the result data bus 53 are to be written or loaded into that template's result buffer 21R, in accordance with a load control signal supplied to timing and control logic circuit 61 over link 51. Result address bus 52 is further coupled to each of A argument source and B argument source fields 21A and 21B. When either of these argument source fields detects a match between the contents of result address bus 52 and itself, a respective (A or B) 'argument-available' flag within the template status word field is raised, indicating the availability of that argument for use in a message to functional computation unit 13.

Additional bus connections of the inter-unit bus portion of operation channel 15 include the coupling of the opcode portion of the template status word field 21S to opcode bus 31 and the coupling of the template address to active template bus 34. Timing and control logic circuit 61 also monitors the contents of template status word field 21S to control the generation of a 'ready' control signal on link 39R to active template designator 38 which, in turn, asserts an 'active' control signal on link 39A to inform timing and control logic circuit 61 when that template has become the current or active data structure.

To facilitate an understanding of the associative operation of the architecture of the single node embodiment of the present invention, in the description to follow, the manner in which a template monitors and responds to the contents of the respective bus portions of result and operation channels 15 and 17 will be explained in detail with reference to FIGS. 6-24, which diagrammatically illustrate the state and functionality of the stored contents of a template in the course of its interaction with the contents of one or more prescribed portions of one of the communication channels.

Result Handling (FIG. 6)

As shown in FIG. 6, result address bus 52 is coupled to each of respective comparators 21CAR, 21CBB and 21CLR wherein its contents are compared with A argument address field 21A, B argument address field 21B and the template address 21L. If the result address matches either of the argument address fields a respective A or B flag is set within the template status word field 21S. If the result address matches the template's address, comparator 21CLR supplies a load input to result buffer 21R causing the contents of result data bus 53 to be written into result buffer 21R. The loading operation of each of the template status word field 21S and result buffer 21R is controlled by a clock control signal from timing and control logic circuit 61 on link 62, which synchronizes the loading operation with the operation of functional computation unit 13.

There are two situations in which a template may respond to the contents of result address bus 52. As pointed out previously, when functional computation

unit 13 completes the execution of a data processing operation, the contents of result address bus 52 identify the template which initiated the data processing operation that produced the output data on result data bus 53. In this circumstance, comparator 21CLR detects a match between the template label 21L and the contents of result address bus 52, so that result buffer 21R is loaded with the result data.

The other situation involves the use of a result value by a template as one of its operands. In this case the template does not store the argument value itself, since it is already being saved by the template that initiated its production. However, it is necessary to store an indication that the argument value is now available (in another template). For this purpose, the template employs comparators 21CAR and 21CBB to determine whether or not the result address matches either or both of its A and B argument source fields. If a match occurs, in either case, a respective flag bit is set in the corresponding A/B availability field within the template status word field 21S, thereby indicating that the A/B argument is resident in the template whose label corresponds to the argument source address. All templates which employ the result data as an operand will set their corresponding flag bit(s) simultaneously.

Assertion of Data Processing Message (FIGS. 7 and 8)

During each cycle of operation of channel 15, two types of templates are involved—a master template (which initiates the assertion of a data processing message), and one or more operand templates (which provide the actual argument values).

In response to a message assert request from functional computation unit 13 on read control bus 35, active template designator 38 asserts an 'active' control signal on bus 39A to a 'master' template (the operation of which is diagrammatically shown in FIG. 7), which is ready to transmit a message and which the arbitration logic within designator 38 has determined to be next in line for service. This active control signal causes control logic 61 to assert a master template control signal on link 71, which is coupled to the enable inputs of each of respective output drivers 72L, 72A, 72B and 720 that are associated with the respective template address, A and B argument source and opcode portion of the template status word fields. Template address driver 72L is coupled to active template bus 34, drivers 72A and 72B are coupled to address buses 41 and 42, and opcode driver 720 is coupled to bus 31. As a consequence, the source and opcode portions of the operation channel are specified immediately by the contents of label field 21L and the opcode portion of the template status word field that are asserted onto the inter-unit buses of operation channel 15, while the addresses of the A and B arguments to be asserted on the A and B argument data buses of the inter-unit bus portion of operation channel 15 are asserted onto buses 41 and 42, respectively. Since buses 41 and 42 are contained within the intra-unit portion of operation channel 15, their contents are not applied directly to functional computation unit 13. Instead, they are used to select the templates in which the actual operand values are stored.

The operation of an operand-supplying template is diagrammatically illustrated in FIG. 8. As shown therein, A and B argument address buses 41 and 42 are respectively coupled to first input ports of comparators 21CAL and 21CBL, second input ports of which are coupled to template address field 21L. Should the con-

tents of either of buses 41 and 42 match the template address, a corresponding one or both of drivers 81A and 81B will be enabled, so as to cause the contents of result buffer 21R, which is the actual operand value to be employed in the execution of the data processing operation, to be asserted on the associated A and B argument data bus 32 and 33 of the inter-unit bus portion of the operation channel 15. It should be noted that the configuration shown in FIG. 8 will support the situation where a single template is requested to supply both the A and B arguments (as in the case of a multiply operation to compute the square of a number).

Acknowledgements

As pointed out supra, in the course of the generation of a data processing message by a master template, the operands are derived from the result buffers in one or more other templates. The contents of these other templates depend upon the contents of a previous template in terms of program flow. It often occurs that a template may be used repeatedly and, in the case of pipelined processing, continuously. It is necessary, therefore, to preserve the order of message assertion of a sequence of templates, so that no template can perform a new operation until all of its dependent templates, that require the use of a previously computed value stored in its result buffer as an operand, have performed their operations. Preserving the order of message assertion is accomplished in accordance with the present invention through the use of an acknowledgement mechanism which is incorporated into the template and which is examined before execution of the template may proceed. The discussion to follow will address two types of acknowledgement mechanisms, one using a counter to keep track of the number of dependent templates that have yet to be asserted before the template may become active, and a distributed mechanism through which a template determines the status of all of its dependent templates whenever its own result buffer is referenced by another template.

Counter-Defined Acknowledgement

FIG. 9 illustrates the manner in which the template status word field 21S is modified through the use of a counter used to track the number of dependent templates remaining to be asserted. Specifically, a first additional static sub-field, identified as #Acks, is used to indicate the number of dependent templates, while a variable Cnt sub-field is used to indicate the number of templates remaining to be asserted before the template may reexecute. The contents of the Cnt sub-field are coupled to a down counter or decrement circuit 96, the output of which is coupled to one input of a multiplexer 94. The output of multiplexer 94 is coupled to the count sub-field. A second input of multiplexer 94 is coupled to receive the contents of the #Ack sub-field of the template status word 21S. The output of the Cnt sub-field is also coupled to a zero reference comparator 97, to determine when the contents of the Cnt sub-field have been decremented to zero.

In operation, when the template has been designated as the master template and asserts a data processing message on operation channel 15, the contents of the Cnt sub-field is reset to the value of the #Ack sub-field. If the master control signal is asserted on link 71, multiplexer 94 couples the #Acks subfield to the Cnt sub-field and an active load signal is coupled from OR gate 91 to the load input of the register in which the Cnt

sub-field is resident. Otherwise, multiplexer 94 selects the decrement output to reduce the value in the Cnt sub-field. Whenever the result buffer is accessed by a master template, one of comparators 21CAL and 21CBL will supply an output through OR gate 91 to cause a load signal to be applied to the Cnt subfield. Because the master active control signal is not asserted at this time, multiplexer 94 couples the decremented count value to the Cnt sub-field. As a consequence, the contents of the Cnt sub-field are decremented every time the template's result buffer is accessed. This process continues until comparator 97 detects that the count value has reached zero, at which time it produces an output indicating the all dependent templates have been asserted.

Distributed Acknowledgement

FIGS. 10 and 11 diagrammatically illustrate a mechanism for generating an acknowledgement through the use of a dual argument acknowledgement bus and status word field. FIG. 10 illustrates comparator and logic circuitry by which a dependent template provides an indication of whether or not it has been asserted; FIG. 11 depicts comparator and logic circuitry for determining if all dependent templates have been asserted.

As shown in FIG. 10, A and B argument address buses 41 and 42 are respectively coupled to pairs of dual comparators 21CAR-A, 21CBA-A and 21CAR-B, 21CBA-B. Comparators 21CAR-A and 21CAR-B are coupled to compare the contents of each of the argument address buses 41 and 42 with the A argument source field 21A, while comparators 21CBA-A and 21CBA-B are coupled to compare the contents of each of the argument address buses 41 and 42 with the B argument source field 21B. If any of the comparators detects a match between its monitored argument address and the stored source address field (representative of earlier asserted templates), then the template is required to indicate status information on the appropriate acknowledgement line 43A, 43B. Acknowledgement link 43A is used in the case of an earlier-asserted template being referenced by a current master template on A Arg address bus 41, while acknowledgement link 43B is used in the case of an earlier-asserted template being referenced by a current master template on B Arg address bus 42.

The template is considered pending with respect to an earlier-asserted template if it has not been asserted since the prior template's most recent result value became available, which is determined by reference to the state of the A argument available and B argument available flags in the template status word sub-fields 21SA, 21SB. If the A argument available flag is set when the prior template referenced by the contents of the A argument source field 21A is detected, then one of AND gates 101 and 102 will be enabled, causing an active (low) signal to be asserted via one of NOR gates on either (open collector) Ack A line 43A or Ack B line 43B. Similarly, if the B argument available flag is set when the prior template referenced by the contents of the B argument source field 21B is detected, then one of AND gates 111 and 112 will be enabled, causing an active signal to be asserted.

Referring now to FIG. 11, which shows the mechanism for determining whether all dependent templates have been asserted, acknowledgement bus portions 43A and 43B are coupled to one input of respective AND gates 212 and 222, second inputs to which are coupled

to the outputs of comparators 21CAL and 21CBL, described previously with reference to FIG. 8. Whenever the result buffer is referenced by argument address signals on argument address bus lines 41 or 42, its corresponding comparator 21CAL, 21CBL will detect a match between the referenced template's address and the argument address, thereby providing an enabling input to one of AND gates 121 and 122. If either of comparators 21CAL, 21CBL detects a match, then it is known that the dependent templates are providing assertion status information on the acknowledgement lines 43A or 43B. An active acknowledgement line indicates that there are still pending dependent templates to be asserted. When the relevant acknowledgement line goes high (inactive) at the time the associated comparator 21CAL, 21CBL has detected a match, then it can be inferred that all dependent templates have been executed, so that the referenced template may execute (subject, of course to the availability of its own operand values). Upon this condition being satisfied, the output of OR gate 123 sets the acknowledgement flag within the status word field 21S. This flag will be reset upon execution of the template.

As pointed out previously, the order of assertion (becoming master/execution) of a template onto the operation channel is dependent upon the template being ready and not requiring execution of any other template, and it must be selected for assertion by the active template designator. FIG. 12 shows the configuration of such a designator circuit for detecting whether a template is ready to be asserted and for selecting which ready template is the next in line to be asserted.

For this purpose the active template designator comprises a daisy-chain arbitration logic circuit 38 which monitors the acknowledgement and A and B availability flags within the status word field via an AND gate 131, the output of which is asserted active (RDY) if all three flags are set. This RDY signal is coupled to a respective stage of a conventional linked AND gate daisy chain arbitration circuit 38. A constant active signal level is asserted at the input to the first (top, as viewed in FIG. 12) stage and is controllably propagated down the chain in dependence upon the assertion of the respective RDY signals from the template status word fields. For a template Ti, if RDY is asserted and there are no higher priority (up the chain) templates waiting to be asserted, then a hold flip-flop HFF is set by a clock signal CK, causing a MASTER i signal to become active, indicating that template Ti is the new master template. In response to this MASTER i signal, the template enables the appropriate output drivers and resets the status word flags.

In the foregoing description it has been assumed that template arguments are generated by the execution of other templates. However, it is occasionally necessary to specify the value of an argument as a constant, namely, an immediate argument. To successfully support an immediate argument, the entries in the template's Argument Source fields 21A and 21B must be capable of storing and using both template addresses and constant values. In addition, during the formation of a data processing message, a mechanism for applying the constant to the operation channel must also be provided. Finally, the presence or use of immediate arguments must not interfere with the acknowledgement mechanism. In the explanation to follow, two mechanisms for supporting immediate arguments will be described. The first, or single level scheme, shown in FIG.

13, uses additional drivers between the argument source field and the data bus. The second, or dual level scheme, shown in FIG. 14, employs additional logic to transfer immediate values from the address bus to the data bus.

Referring now to FIG. 13, there is shown a first modification of the operand generation mechanism described above with reference to FIG. 7, for supporting the use of immediate arguments. As shown in FIG. 13, the operation channel 15 is modified to include a pair of additional A and B argument type signal lines 41T and 42T to indicate whether or not the arguments are immediate. These argument type signals (A Arg Type and B Arg Type) are asserted by the master template. If none of these lines is active, its associated argument is immediate rather than requiring direct addressing to an argument source field. Each of links 41T and 42T is coupled to (controllably) disable (in the case of an immediate argument) a respective template address (label) comparator 21CAL, 21CBL that is monitoring its associated address bus, so as to prevent any template, other than the master template, from applying an argument value to a data bus.

A pair of additional drivers 72A1 and 72B1 are coupled to the respective A and B argument source entries 21A and 21B, for controllably asserting the immediate arguments directly onto the data buses. These drivers are controllably enabled by a pair of AND gates 147 and 148 which monitor a pair of status flag bits Ia and Ib that are incorporated into the template status word 21S. These additional bits are active (logical 1) when the A or B argument source fields 21A or 21B contain immediate arguments. Gates 147 and 148 are controlled by the active template designator (FIG. 12) asserting an active signal on master line 39M. Otherwise, if immediate flag bits Ia, Ib are not set, then via inverters 145, 154 and AND gates 146 and 149, drivers 72A and 72B are controllably enabled by the assertion of an active signal on master line 39M.

The use of the additional immediate flag bits within the status word field also affects the manner in which the RDY signal is generated. In addition to requiring the setting of the acknowledgement flag Ack, indicating that all dependent templates have been asserted, either the A or B availability flag is set, or the Ia or Ib flags are set indicating that the argument is immediate and its value is resident in the corresponding argument source field 21A, 21B. Logical circuitry for producing the RDY signal in the case of an expanded status word field to include immediate arguments includes OR gates 151, 152 and AND gate 153, as shown.

In addition to being coupled to disable comparators 21CAL, 21CBL, argument type signal lines 41T and 42T are coupled to controllably disable comparators 21CAR-A, 21CBA-A and 21CAR-B, 21CBB-B of FIG. 10, to prevent the assertion of false acknowledgement signals on the Ack A and Ack B lines.

FIG. 14 shows a second, or dual level, modification of the operand generation mechanism of FIG. 7, for handling both direct addressing and immediate type arguments, again using the A and B argument type lines 41T and 42T, shown in FIG. 13, but with reduced logic complexity. As shown in FIG. 14, a pair of additional A and B drivers 161 and 162 are coupled between the respective A and B address and A and B data buses. These additional drivers are controllably enabled directly by the respective A and B type lines. In the configuration of FIG. 14, when the Ia flag is set, the value is applied to the A Arg Address bus; however, driver

161 is enabled, so as to assert the immediate value onto the A data bus 32. Similarly, when the Ib flag is set, the value is applied to the B Arg Address bus; however, driver 162 is enabled, so as to assert the immediate value onto the B data bus 33.

The data processing operations of the associative communications architecture thus far described involve the use of fixed operators. In a practical system, however, function execution will involve conditional operators for flow control and decision making. In the discussion to follow, the manner in which the architecture described supra is modified to handle conditional operators, termed 'switch' and 'select', will be addressed.

Switch Template

A switch template, diagrammatically shown in FIG. 15, initiates the execution of a data processing operation in which a primary data operand A is made available to one of two sets of prescribed recipient templates (0,1) depending on the value of a control Boolean operand B. Any dependent template's argument source entry will reference only one of the two result values. Namely, the value returned is always the template's A argument, with the named result to which the value is returned being defined by the Boolean value of the template's B argument. If the Boolean operand value is false (logical 0), the A operand value will be returned to the first result, designated <switch label>0.0, where <switch label> is the contents of the switch template's label address field. Similarly, if the B operand is true (logical 1), the value of the A operand will be returned to the <switch label>0.1 result. Because each template that is dependent upon the switch template last acquired the most recent of a pair of values referenced by it, a switch template is inhibited from executing until all of its dependent templates, for both result values, have been executed.

The mechanism of a switch template, shown in FIG. 16, involves a minor modification of the basic template data structure shown in FIG. 2, specifically the addition of a result index line 52IX to the result channel, which is coupled to a result index bit (21IXA, 21IXB) that is appended to the result address to identify to which of the two result elements the returned value is directed. Generally, the result index will be a logical 0, referencing the first result element R0 of the template (since most templates have only single element results); for the infrequent case in which the second result element R1 is referenced, the result index is a logical 1.

The configuration of the modified signal processing hardware within the template necessary to support a switch function is illustrated in FIG. 17 as comprising comparators 21CXA, 21CXB which are coupled to compare the appended A and B index bits of the template status word field 21S with the contents of the result index line 52IX. The output of comparator 21CXA is logically ANDed with the output of comparator 21CAR in AND gate 201; the output of comparator 21CXB is logically ANDed with the output of comparator 21CBR in AND gate 202. For a template's A operand, comparator 21CAR compares its contents with the result address, as described previously, while comparator 21CXA compares the bit on the result index line 52I with the A index bit of the status word field. Only if both compare operations are true does the template recognize the presence of its A operand and updates its A availability bit. A similar operation is carried

out with respect to the B operand, using its dedicated comparator logic.

It should be noted that there is no need to specify which of the two result elements from a switch template is required by a master template argument since there is only one available at a time from a given switch and the current master template has already determined that the one available is the one that it requires. Moreover, since all dependent templates of a switch template must have used its previous results before it can be asserted, the distributed acknowledgement mechanism described supra need not distinguish between templates accessing result element R0 and those accessing result element R1, so that it requires no modification in order to support a switch template.

The assertion of a switch template proceeds in the same manner as a normal dyadic operator, placing its address (label), opcode and A and B argument source entries onto the operation channel 15. Functional computation unit 13 returns the contents of the A data bus 32 to result data bus 53, the label to result address bus 52 and the appended (least significant) bit of the contents of the B data bus 33 to result index line 52IX. Dependent template then determine whether or not they are able to use the result in accordance with the value of the index bit and the contents of their own extension bits, as explained above.

Select Template

A select template, diagrammatically shown in FIG. 18, and the mechanism for the execution of which is shown in FIG. 19, initiates the execution of a data processing operation in which one of its two data operands A and B is made available, depending on the value of a third, control Boolean operand Z. In order to be asserted, the control operand Z and the selected argument (A or B) are required. To accommodate the additional Z operand, the data structure is modified to include a Z argument source field 21Z and to add a corresponding Z argument available bit Z into the status word field 21S. In addition, the least significant bit (S) of the result data is latched as part of the template status word, so that the template can hold the Boolean value of the Z argument and can determine which of its A and B arguments is to be applied to the A argument address bus of the operation channel when the select template becomes a master template.

The signal processing logic for enabling the template data structure to perform the select function is shown in detail in FIG. 20 as comprising a first Z comparator 21CZR which compares the entry in the Z argument source field 21Z with the contents of the result bus 52, a second Z comparator 21CZA which compares the Z argument with the contents of the A argument address bus 43A and a third Z comparator 21CZB which compares the Z argument with the contents of the B argument address bus. The output of comparator 21CZR sets a Z available flip-flop 21SZ within the template status word 21S and loads an S flip-flop 21SS with the least significant bit 53LSB of the result data bus 53. The outputs of comparators 21CZA, 21CZB are coupled to NOR gates 240 and 241, respectively, which are controllably enabled by the Q output of Z latch 21SZ, which is further coupled to an input of AND gate 231. The Q output of S latch 21SS is coupled to NAND gate 212 and AND gates 223 and 224 and to OR gate 213. It is complemented by inverter 226 and applied to AND gate 222.

The opcode field 210 is coupled to a decoder 211 which provides a first "select" output to each of NAND gate 212 and AND gates 231, 223 and 243. A second "other" output of decoder 211 is coupled to OR gate 213 and to each of AND gates 221 and 242. The output of NAND gate 212 is coupled to one input of AND gate 214 a second input of which is coupled to master control line 39M, which serves as a clear or reset input for the contents of the template status word. Master control line 39M is further coupled to the clear inputs of acknowledgement latch 21SACK and Z latch 21SZ and to AND gates 215, 243, 252 and 253. A second input of AND gate 252 is coupled to the output of AND gate 223, which is also complemented by inverter 251 and applied to a second input of AND gate 253.

The output of AND gate 253 is applied to the enable input of driver 161, while the output of AND gate 242 is coupled to the enable input of driver 162. A pair of additional drivers 211 and 220 are coupled to Z argument source field 21Z and B argument source field 21B, respectively. The output of driver 211, which is enabled by the output of AND gate 243, is coupled to B argument address bus 42. The output of driver 220, which is enabled by the output of AND gate 252, is coupled to A argument address bus 41.

Logic circuitry for generating a RDY indication to the template designator 38 over line 39R includes AND gate 250, a first input of which is coupled to the Q output of Acknowledgement latch 21SACK, and a second input of which is coupled to OR gate 232. OR gate 232 is coupled to the outputs of each of AND gates 221 and 231. AND gate 221 receives the Q outputs of the A and B available latches 21SA and 21SB and the other output of decoder 211. The Q output of A available latch is also coupled to AND gate 222, while the Q output of B available latch is coupled to AND gate 224. The outputs of AND gates 222 and 224 are coupled via OR gate to AND gate 231.

In operation, opcode decoder 211 examines the contents of the opcode field and determines if the template is to operate as a select template (asserting its 'select' output bit) or if the template is another type (asserting its 'other' output bit). If the template is not a 'select' template, then gates 221, 232 and 250 cause a RDY signal to be placed on line 39R, when the A and B and ACK latches 21SA, 21SB, 21SACK are set (A and B arguments available and acknowledgement ACK flags in the status field are set).

On the other hand, if the opcode indicates that the template is a select template, the Z argument must be available; namely comparator 21CZR must have detected a match between the contents of the result bus and the Z argument source field 21ZZ, thereby setting latch 21SZ and enabling a second input of AND gate 231. The third input of AND gate 231 depends upon the value of the least significant bit S of the result data bus. If the bit is a 0, so that the Q output of latch is 0, AND gate 224 is disabled, while AND gate 222 receives a 1 on its input coupled to inverter 226. To be enabled the second input of AND gate 222 must indicate that the A argument is available (A availability latch 21SA is set). Alternatively, if the least significant bit S is a 1, then the B argument must be available, to enable the second input to AND gate 224.

As pointed out previously, when a template becomes a master template, it supplies operation data and reinitializes template status word 21S, which is ordinarily performed by clearing the A, B and ACK flags. For a

select template, however, not all flags are necessarily reset. When a template becomes a master only the argument available flag of the argument that is actually used is cleared. Control logic for this purpose includes gates 212-215. The A available flag is cleared if the template is not a 'select' template or, if it is a 'select' template, if the value of the S bit is a 0. The B available flag is cleared if the template is not a 'select' template or, if it is a 'select' template, if the value of the S bit is a 1.

In the course of execution of a select template the Z argument and one of the A and B arguments are employed. For this purpose the Z argument is applied to the B argument address bus of the operation channel 15 via driver 211. If the S bit is a 0, then driver 161 is enabled via AND gate 253, and the contents of the A argument source field is asserted onto A argument address bus 41. Thus, the A and Z arguments are asserted and the B argument is saved for later use, as described above. If the S bit is a 1, on the other hand, then driver 162 is enabled via AND gate 242, and the contents of the B argument source field is asserted onto B argument address bus 42. The B and Z arguments are asserted and the A argument is saved for later use.

Distributed acknowledgements are extendable to the Z argument for the select template through the use of comparators 21CZA and 21CZB to compare the contents of Z argument field 21Z with the A and B argument addresses on buses 41 and 42, respectively, to thereby determine the select template's response on the acknowledgement buses 43A and 43B. Gates 240 and 241 assert active signals on buses 43A and 43B when either comparator detects a match and the Z bit is set, indicating that the template has not been asserted since its Z argument became available.

When all of the data structure and signal processing mechanisms described thus far are incorporated into a single template architecture, the expanded template fields and template word status entries may be diagrammatically represented by the data structures shown in FIGS. 21 and 22, respectively. The comparator and driver circuitry and their associated communication buses, together with the combined control logic therefore are depicted in FIGS. 23 and 24, respectively. To simplify the circuitry, operand decoder employs only a single output D to indicate that the template possesses select functionality. The ready signal RDY may be represented by the Boolean expression:

$$RDY <- \neg ACK * [A + I_A + S * D] * [B + I_B + S * D] * [Z + I_Z + D]$$

Note that the participation of immediate values for the Z argument, including Z_X is shown by the logic of FIG. 24. When a select template becomes master, the values of the A and B argument type signal lines are defined in accordance with the value of the S bit of the status word field. I_A or I_B flags are asserted on A argument type line depending whether or not the template is a select template. If it is, whether the value of S is a 0 or a 1 is controlling. I_A will be asserted on the B argument type line if the template is a select template; otherwise, I_B will be asserted.

Multi-node Architecture

The associative template-based data processing mechanism described thus far is configured as a single, self-contained data processing station (or node), wherein all operands employed in the course of the

execution of data processing messages and all execution results are exchanged between one template data storage facility and one functional computation unit. In a larger, system level architecture, comprised of multiple nodes, each having its own dedicated functional computation unit and template storage facility, wherein operand and result data are exchanged among the nodes of the system, the associative communication mechanism of the present invention may be extended to facilitate parallel computation throughput.

In the description to follow, for purposes of providing an illustrative example of a multiple node architecture, the system configuration will be considered to have a mesh topology, such as those illustrated in FIG. 25, in which each node is connected with and may communicate with three nearest neighbor nodes. For reference purposes, the exemplary node of interest will be identified as a south (S) node, having neighboring north (N), east (E) and west (W) nodes with which it shares data resources in the course of execution of its own data processing operations and also in the course of the execution of data processing operations by those neighboring nodes. Namely, within an individual node (e.g. a south node), a data processing operation defined by a template stored within that node is always executed by the functional computation unit within that node. However, the operands required for and the results of that execution may be shared by templates in nearest neighbor (north, east and west) nodes. In order to effect this sharing of data resources, the architecture of each node is configured to provide an inter-node associative communication capability, similar to that of an individual node, for those aspects of a template which may depend upon or be necessary for the execution of a template in any of its neighboring nodes, by assigning associative communication control functions to dedicated storage and supervisory units within each node.

More particularly, as diagrammatically illustrated in FIG. 26, the architecture of an individual node (S) includes a multinode communications channel interface 301 containing inter-node communication links 303 that extend to its three neighboring nodes (N, E, W) and through which data resources are shared. An individual inter-node link is shown in FIG. 27 as containing an operand address segment OA, an operand value segment OV, a result segment R and an acknowledge segment A.

The operand address segment OA conveys the address of the template in which the operand is to be obtained, while the operand value segment OV conveys the actual operand value to be used in the execution of a data processing operation by the functional computation unit within the node to which the operand value is transmitted. The result segment R conveys the result address and an indication of the availability of the result of a data processing operation executed by the functional computation unit of one node to each of its nearest neighbor nodes. The acknowledge segment A conveys acknowledge information consisting of the acknowledge condition state and is used to set the acknowledge flags of a selected template.

Within the node (S) itself, each of operand address segments OAE, OAW and OAN from neighboring nodes E, W and N and an internal operand address link OAS from a program execution coordinator 304 (to be described below with reference to FIG. 30) within which the data flow graph topology of that node is

stored, is coupled to a data store 305 (to be described below with reference to FIG. 28). Data store 305 stores operands to be employed in the execution of data processing operations by templates contained within that node and templates of its neighboring nodes. It also stores acknowledgement information that is made available to the program execution coordinator 304 over link 306.

Within data store 305, operand addresses conveyed by local operand address link OAS and internode links OAE, OAW and OAN are employed to access operand values stored in an operand memory within the data store 305; accessed values are coupled links OVE, OVW, OVN, OVS to an operation packet builder 313 (to be described below with reference to FIG. 29), which essentially comprises a set of temporary holding registers in which the various component parts of a data processing message, intended for transmission over an internal execution message link 312 to a local functional computation unit 314, are assembled.

The results of instruction execution by functional computation unit 314 are coupled over an internal result communications channel RS to data store 305, program execution coordinator 304 and over respective portions RE, RW and RN of result segment R to the program execution coordinators in neighboring nodes E, W and N. The result value is stored in the operand memory within the data store 305, while the result address and the Z (least significant result value) bit are applied over result address segment R.

The opcodes of the instructions to be executed within the local node (S) are stored (in terms of template address) in an opcode store (memory) 321. In response to a template (opcode) address coupled onto link 325 by program execution coordinator 304, opcode store 321 couples the opcode over link 323 to operation packet builder 313, wherein data processing messages are assembled, as noted above. Operation packet builder 313 is also coupled (over link 325) to receive the template address from program execution coordinator 304, in order to identify the template originating the data processing execution request. As noted above, the operand values of data processing messages that are assembled by operation packet builder 313 are coupled over internode operand value links OVE, OVW and OVN (from neighboring nodes E, W and N) and local operand value link OVS from data store 305.

Data Store (FIG. 28)

Data store 305 contains a pair of dual-port operand (result value) data memories 331 and 332, in which operands (result values) are stored for use by any of the four interconnected (E, W, N, S) nodes. The use of a pair of redundant data memories (and attendant access control circuitry) allows two operands to be resolved simultaneously, thereby increasing the availability of operand data. Specifically, each of the two memories performs a read and a write in the same cycle. Thus, the functional computation unit writes into its node's operand memory which two read addresses are being served.

The writing of result values into memories 331 and 332 occurs as a result of operation of the local functional computation unit, with the result value being coupled over result data input link RD and the template address being coupled over link RA, to identify that location in each of the data memories in which the result value is to be stored.

The reading of operand data values out of memories 331 and 332 and the handling of acknowledgements is effected through respective sets of buffer registers 335 and 336 and associated multiplexers 337 and 338 under the supervision of an arbitration logic circuit 341.

More particularly, each of buffers register sets 335 and 336 contains four data and address register units 351E, 351W, 351N, 351S and 352E, 351W, 351N, 351S, respectively, coupling operand address links OAE, OAW, OAN, OAS and operand value links OVE, OVW, OVN, OVS with data store operand data links 355D, 356D and data store operand address links 355A, 356A, through which operand values stored within data memories 331 and 332 are accessed. Each data and address register unit contains a pair of registers, one for storing operand data that is read out from memory and another for storing acknowledgement status information to be forwarded to the program execution coordinator and an operand address for accessing the operand value that is to be read out of memory and coupled to the operation packet builder of the requesting node.

Which operand address link will be serviced is handled by multiplexers 337 and 338 under control of an arbitration logic circuit 341, which is preferably implemented as a round-robin arbitration circuit to ensure that no requesting link will be locked out. Whenever the contents of an operand address register are coupled over one of links 355A, 356A to access an operand from one of data memories 351, 352 the associated acknowledgement status information is coupled over link 306 to program execution coordinator 304.

Operation Packet Builder (FIG. 29)

As noted previously, the operation packet builder 313 assembles data processing operation message packets for delivery to functional computation unit 314. For this purpose, the packet builder is coupled to receive an opcode from opcode store 321, a result address from data store 305 and, for each operand, an additional neighbor interface address which specifies a neighbor node and from which of the memories of the data store an operand is to be accessed. Because the delay between the initiation of the execution of a data processing operation and the arrival of an operand can vary, and because there are multiple (four in the present example) sources of operand values, throughput can be enhanced by preparing, concurrently, a plurality of template data processing messages. To accommodate multiple template messages, the operation packet builder is essentially configured as a set of buffers in which opcode and result addresses are temporarily stored, while operands are being fetched from the data store. Once the operand data values have been obtained from the data store, the assembled message packet is transmitted to the functional computation unit.

The buffer circuitry of which operation packet builder 313 is configured is diagrammatically shown in FIG. 29 as a first set of four packet registers 361, 362, 363 and 364. Each register encompasses a packet field that contains the opcode, result (originating template) address, the A operand and the B operand. Associated with the first register set is a second register set, containing four neighbor/data memory registers 371, 372, 373 and 374. Each of the registers of the second set stores three bits, for designating in which of the four nodes the operand value is stored and which of the data memories (331 or 332) of that operand value-storing node contains the operand. Once an operand value (A

operand or B operand) has been accessed from the data store 305 and stored in the operand field of the designated packet register, the contents of the address register within the data store that had been storing operand address is cleared, so that it may be used by another template. Whether or not an operand address register has been cleared so that it may receive a new operand address is indicated to a dispatcher logic unit within program execution coordinator 304 be described below with reference to FIG. 41.

Program Execution Coordinator (Representative Template Shown in FIG. 30)

As noted previously, the program execution coordinator stores data flow program execution topology, maintains the program control state and determines the order in which program templates are asserted. In addition, it receives acknowledge synchronization signals from the program execution coordinators of neighboring nodes indicating whether or not any of their templates still require the availability of an operand that is associated with a template to be executed in that node. Finally, the program execution coordinator monitors the result link from the functional computation units in each of its neighboring nodes in order to synchronize the operation of the node with the completion of operations in other nodes and to update its control state.

For this purpose, the program execution coordinator is comprised of memory for storing each field of the templates of that node, except for the result values (which are retained in the data store, as explained supra), the opcodes and (condition determination) comparator logic, coupled with the respective fields of the templates, for monitoring, in substantially the same manner as a single node architecture described above, the result and acknowledge signal communication links of the local and neighboring nodes.

Result Bus Comparison

Referring now to FIG. 30 there are diagrammatically illustrated the respective fields of an individual one of the templates stored within the program execution coordinator and the mechanism through which the program execution coordinator monitors the result address link from its local node and those of neighbor nodes for setting the A, B and Z flags. Because of the static allocation of dataflow templates, the operand referenced by an operand address field can come from only one of the local node and the three neighboring nodes; it is not variable. As a consequence, it is necessary to monitor only one of the four result address buses, the identity of which is specifiable by the two most significant bits of the template address.

To this end, rather than provide respective A and B operand address and Z address comparators for each of the four result buses, each template utilizes only one comparator for each of the respective operand and Z address fields, with the input to the comparator being defined by a multiplexed connection to each bus. As shown in FIG. 30, respective comparators 381, 382 and 383 are coupled to compare the A operand, B operand and Z address values of a respective template 300 with the outputs of respective multiplexers 391, 392 and 393. Each multiplexer has four inputs coupled to the result links of nodes E, W, N, S and a select input which is coupled to the two most significant bits of the template address, so as to designate which of the result links will be coupled to its associated comparator. The use of a

multiplexer and comparator pair significantly reduces the hardware complexity (transistor count and power consumption) that would be encountered using a separate comparator for each bus.

Acknowledgement Handling

In the single node architecture, described above, acknowledgement signaling is essentially defined by two operations: 1-generating the acknowledge condition state; and 2-setting an acknowledge flag. Since, in a single node architecture all templates reside in the same node, both of these operations may be carried out at the same time. In a multinode architecture, however, a dependent template may be located in any one of four different nodes, the data processing operations within which are being carried out by independent functional computation units. Still, whenever a node creates an acknowledge condition signal, that signal needs to go to only the node to which the operand data request is directed. All templates in one node that use the results of a template in another node monitor each other to determine whether every template in that node is finished with the operand of interest. When the operand is no longer required by any template in that node an acknowledge condition signal is generated.

The handling of acknowledgements is effected by expanding the template acknowledgement field (shown in FIG. 30) and logically operating on a set of four acknowledge flags and associated mask bits that make up the expanded field, using the circuitry shown in FIG. 31. As shown in FIG. 30 the expanded acknowledge field includes a masking 'expected' bit X, which is set if an acknowledgement is still expected, and a 'received' bit R, which indicates that an expected acknowledgement has in fact been received, for each of the four nodes. For each of the nodes E, W, N, S, a respective one of AND gates 410, 402, 404, 404 is coupled to the X and R bits and has its output coupled to NOR gate 405. If there are no templates within a neighboring node that require the use of a template in the local node then no acknowledgement is expected and the X bit is not set. If this mask bit has been set, then upon a change in state of the acknowledge 'received' bit R, the output of its associated AND gate will change state, thereby applying a 0 to that node's input to NOR gate 405. Upon the acknowledgements for all four nodes having been satisfied, the output of NOR gate changes state to one bit, thereby asserting a one on its input to template-ready AND gate 406. Other inputs of AND gate 406 are coupled to receive the A, B and Z flags of the template 380. Upon each of the flags (A and B operand, Z bit and Acknowledge) being set, the output of AND gate 406 changes state, indicating that the template may be asserted.

Template Execution

The execution of a template in the multi-node architecture proceeds as follows. Again, considering the local node of interest to be the south node S, let it be assumed that the A operand is to be obtained from the local node and its B operand from neighboring node E; in addition, the result values are used by two local (node S) recipient templates and one neighboring (node E) template, as diagrammatically illustrated in FIG. 32.

An execution cycle begins with the template being in condition to be asserted to its local functional computation unit. The program execution coordinator dispatching logic, described above with reference to FIG. 31,

selects the pending template for execution when a communication port to the local data store 305_S (which contains the A operand) and a similar port to the data store 305_E in the east node E containing the B operand are available and transmits the address of the pending template to the opcode store 321 and to operation packet builder 313_S. The contents of the A and B operand address fields with the program execution coordinator 313_S are asserted onto the A and B operand buses; the A operand address is applied to the available access port of data store 305_S and the B operand address is applied to the available access port of data store 305_E in neighboring node E.

As explained above, for each of the operand fetches an acknowledgement condition state signal is generated; the other templates in local node S monitor the operation channel 301 and if either of their A and B operand fields matches either of the A and B operand addresses that have been asserted onto the operation channel, these templates assert an active signal on the wired-OR acknowledge signal line, indicating that these templates still require that operand to be available. Otherwise, their respective acknowledge lines are not asserted active. As pointed out above, the states of these lines tell the source template whether or not there are other templates in the local node S for which that operand must remain available.

Once the pending template has been selected for execution, operation packet builder 313 selects an available buffer and stores the identity of the data store ports that are to supply the A and B operands. The identity of the asserting template is stored in the result field and applies its address to the opcode store 321, so that the opcode associated with that template is read out of the opcode store and loaded in the opcode field in the packet builder buffer.

As described previously, the data store arbitration logic is preferably implemented as a round-robin mechanism. Consequently, within neighboring node E from which the B operand is to be accessed, each of the address buffers is examined. When the port for the asserted template is accessed, the data memory reads the contents of its addressed value and returns it to the dedicated output buffer of the local node, which supplies the operand directly to the packet builder. At the same time, the acknowledge condition state that accompanies the operand address is coupled over link 306 to the program execution coordinator. The address selects the operand source template and the acknowledge condition state is loaded into the acknowledge flag associated with local node S.

When the contents of each of the fields of the requesting template's buffer within the operation packet builder 313 have been filled, the buffer's ready flag is set (the output of AND gate 406 is enabled). The functional computation unit detects the assertion of the ready flag on the operation channel and acquires the contents of the data processing message within the buffer.

After processing the instruction, functional computation unit 314 places the result value and the address of the asserted template on the result channels to each of the nodes. The result value is stored in each of the redundant pair of data memories 351, 352 within the data store 305_S of the local node S and the result address is distributed over the result bus to the program execution coordinators of each of the four nodes E, W, N, S.

Within each of these nodes, the result address comparators of each template monitor the result buses of the

nodes from which their operands are derived. The A operand comparator of the source template in local node S monitors the local result bus. When the operand source templates are asserted, each comparator determines the availability of the result value by detecting a match between the contents of its operand address field and the result address bus. In response to a match, the corresponding A or B flag is set.

Thereafter, as other templates, which are dependent upon the results of the local template's execution, are asserted they access the data memory in local node S to obtain its result value and they return acknowledge condition state signals to the acknowledge ports of the program execution coordinator. When a dependent template in neighboring node E is asserted, there are no other templates using local node S's result value as operands, so that the acknowledge condition state that is returned to the local program execution coordinator causes the corresponding acknowledge condition flag to be set. When the first of the two dependent templates in node S is asserted, the acknowledge flag will not be set since the second dependent template has yet to be asserted and still requires the result value to be available. Once the second template is asserted, however, the acknowledge flag in node S is set (there are no other templates in node S requiring the result value to remain available for their use).

With both the neighboring node template and the local node's two operations completed, both the A and B flags are now set, indicating that both operands are available. The acknowledge flag in local node S and that associated with the neighbor node E containing the dependent template are set; also, the masks (X) bits of the remaining two acknowledge flags (for nodes W and N) are set since these nodes contain no templates that are dependent on the template of interest in node S. As a consequence, each of AND gates 401-404 provides an enable input to ready AND gate 406 (FIG. 32), so that its output changes state indicating that the template is ready to be asserted again.

As will be appreciated from the foregoing description, the present invention provides a computer architecture which significantly reduces the substantial temporal overhead and memory bandwidth requirements of token-based static data flow computer architectures by replacing token-based processor communications with associative processing, similar to that used for associative memories, through which plural data execution control structures, or templates, of the system are interconnected with one another and with the data processing resources of the system, so that they may monitor and respond to operations carried out with respect to all other components of the system simultaneously, thereby increasing data processing execution speed and enhancing the efficient use of system memory.

While we have shown and described several embodiments in accordance with the present invention, it is to be understood that the same is not limited thereto but is susceptible to numerous changes and modifications as known to a person skilled in the art, and we therefore do not wish to be limited to the details shown and described herein but intend to cover all such changes and modifications as are obvious to one of ordinary skill in the art.

What is claimed is:

1. A data processing system comprising:
 - first means for controllably executing a data processing operation on data supplied thereto;

second means for controlling the supply of data to be processed by said first means and including means for storing a plurality of data processing execution control structures, each respective one of said data processing execution control structures containing first information representative of the identification of that data processing execution control structure, second information representative of a data processing operation to be performed by said first means, third information representative of data to be processed by said first means, fourth information representative of the status of a data processing execution cycle, and fifth information representative of the result of a data processing operation carried out by said first means;

a first communications channel for coupling data and control messages from said second means to said first means; and

a second communications channel for coupling the results of a data processing operation carried out by said first means to said second means; and wherein

each of said plurality of data processing execution control structures stored by said second means is associatively coupled with and simultaneously and continuously monitors said first and second communications channels, and said second means includes means for asserting onto said first communications channel a data processing control message containing first and second information derived from a selected data processing execution control structure requesting the execution of a data processing operation by said first means, and third information derived from fifth information stored within prescribed ones of said plurality of data processing execution control structures; and wherein

said first means includes means for asserting onto said second communications channel a data processing output message containing the identification of said selected data processing execution control structure and the result of the data processing operation carried out in accordance with the second and third information asserted onto said first communications channel.

2. A data processing system comprising:

a functional computation unit in which data processing operations are executed on operand data in accordance with an opcode supplied thereto, so as to produce output data representative of the result of the execution of a data processing operation;

a storage unit in which are stored a plurality of data processing execution control structures each of which comprises a plurality of entries including an address for identifying that data processing execution control structure, an opcode for defining a data processing operation to be performed by said functional computation unit, a plurality of operand source addresses for specifying the addresses of data processing execution control structures containing operands to be employed in the execution of said defined data processing operation, the status of said data processing execution control structure with respect to its associated data processing operation and a result entry in which the output data produced by said functional computation unit as a result of its execution of a data processing operation

tion requested by that data processing execution control structure is stores;

a first communications channel, coupled between said storage unit and said functional computation unit, and being monitored simultaneously by each of the data processing execution control structures of said storage unit, for conveying data processing request messages from said storage unit to said functional computation unit;

a second communications channel, coupled between said functional computation unit and said storage unit, and being monitored simultaneously by each of the data processing execution control structures of said storage unit, for conveying output data from said functional computation unit to said storage unit; and

a control unit, coupled with said storage unit, for controllably causing a data processing message to be asserted onto said first communication channel in accordance with the contents of a selected one of said data processing execution control structures, said data processing message including the contents of the address and opcode entries of said selected data processing execution control structure and operands specified in accordance with the operand source address entries of said selected data processing execution control structure, and for causing output data, produced by said functional computation unit as a result of a data processing operation executed in accordance with said data processing message and asserted onto said second communications channel by said functional computation unit, to be captured in the result entry of said selected data processing structure.

3. A data processing system according to claim 2, wherein a respective data processing execution control structure includes means for monitoring said first communications channel and asserting the contents of its result entry onto said first communications channel in response to recognizing its address having been asserted thereon, so that said result may be employed as an operand for the execution of a data processing operation by said functional computation unit.

4. A data processing system according to claim 3, wherein the status entry of a respective data processing execution control structure includes acknowledgement information representative of whether any other data processing execution control structure of said storage unit requires the use of the contents of the result entry of said respective data processing execution control structure as an operand.

5. A data processing system according to claim 4, wherein the status entry of a respective data processing execution control structure includes operand availability information representative of whether the result entry of another data processing execution control structure, whose address is defined by the contents of a source address entry of said respective data processing execution control structure, contains an operand required for the execution of a data processing operation defined in accordance with opcode entry of said respective data processing execution control structure.

6. A data processing system according to claim 5, wherein a respective data processing execution control structure further includes means for indicating the readiness of said data processing execution control structure to have a data processing message asserted on,

said first communications channel in accordance with the contents of said status entry.

7. A data processing system according to claim 6, wherein said indicating means includes for indicating the readiness of said data processing execution control structure to have a data processing message asserted on said first communications channel in response to said acknowledgement information being representative that no other data processing execution control structure of said storage unit requires the use of the contents of the result entry of said respective data processing execution control structure as an operand, and that said operand availability information is representative that all operands required for the execution of a data processing operation defined in accordance with opcode entry of said respective data processing execution control structure are available.

8. A data processing system according to claim 7, wherein said control unit includes means for clearing the contents of the acknowledgement and operand availability information within the status entry of said respective data processing execution control structure in the course of causing a data processing message associated with said respective data processing execution control structure to be asserted onto said first communications channel.

9. A data processing system according to claim 2, wherein the status entry of a respective data processing execution control structure includes operand availability information representative of whether the result entry of another data processing execution control structure, whose address is defined by the contents of a source address entry of said respective data processing execution control structure, contains an operand required for the execution of a data processing operation defined in accordance with opcode entry of said respective data processing execution control structure.

10. A data processing system according to claim 9, wherein said second communications channel includes a data portion over which said output data is conveyed and a result address portion over which the address of an output data recipient data processing execution control structure is conveyed, and wherein a respective data processing execution control structure includes means for comparing its operand source address entries with the contents of the address portion of said second communications channel and causing said operand availability information of said status entry to indicate that an operand entry required for the execution of a data processing operation defined in accordance with an opcode entry of said respective data processing execution control structure is available in the result entry of another data processing execution control structure whose address matches one of the operand source address entries of said respective data processing execution control structure.

11. A data processing system according to claim 10, wherein said second communications channel further includes a result index portion for identifying one of the operand source entries of a data processing execution control structure and said comparing means includes means for causing said operand availability information of said status entry to indicate that an operand entry required for the execution of a data processing operation defined in accordance with an opcode entry of said respective data processing execution control structure is available in the result entry of another data processing execution control structure whose address matches the

operand source address entry of said respective data processing execution control structure as identified by said result index portion.

12. A data processing system according to claim 2, wherein a data processing execution control structure further includes means for controllably enabling the contents of an operand source address entry to be directly asserted as an operand for the execution of said defined data processing operation.

13. A data processing system according to claim 2, wherein said first communications channel includes a data portion over which operands are conveyed, an address portion over which the address of a selected data processing execution control structure is conveyed and an opcode portion over which the opcode entry of a selected data processing execution control structure is conveyed, and further including an intra data processing execution control structure address link over which operand source addresses are conveyable among the data processing execution control structures of said storage unit, and wherein a respective data processing execution control structure includes means for comparing its address with the contents of said intra data processing execution control structure address link and causing the contents of its result entry to be asserted onto said data portion of said first communications channel, in response to detecting a match between its address and the contents of said intra data processing execution control structure address link.

14. A data processing system according to claim 13, wherein a respective data processing execution control structure includes means for controllably asserting its operand source addresses onto said intra data processing execution control structure address link in the course of the assertion of a data processing message, and wherein said second communications channel includes a data portion over which said output data is conveyed and a result address portion over which the address of an output data recipient data processing execution control structure is conveyed, and wherein a respective data processing execution control structure includes means for controllably causing said operand address asserting means to assert an operand source address onto said intra data processing execution control structure address link in accordance with the contents of the address portion of said second communications channel.

15. A data processing system according to claim 14, wherein said controllably causing means includes means for controllably causing said operand address asserting means to assert a selected operand source address onto said intra data processing execution control structure address link in accordance with the contents of a prescribed portion of the data portion of said second communications channel.

16. A data processing system comprising:
first means for controllably executing a data processing operation on data supplied thereto;
second means for controlling the supply of data to be processed by said first means and including means for storing a plurality of data processing execution control structures, each respective one of said data processing execution control structures containing first information representative of the identification of that data processing execution control structure, second information representative of a data processing operation to be performed by said first means, third information representative of data to be processed by said first means, fourth information

representative of the status of a data processing execution cycle, and fifth information representative of the result of a data processing operation carried out by said first means;

a first communications channel for coupling data and control messages from said second means to said first means; and

a second communications channel for coupling the results of a data processing operation carried out by said first means to said second means; and wherein

each of said plurality of data processing execution control structures stored by said second means is associatively coupled with and simultaneously and continuously monitors said first and second communications channels for the presence of said first information having been asserted thereon and, in response to detecting the presence of its identification, controllably interfaces prescribed information associated with the execution of a data processing operation, and said second means includes means for asserting onto said first communications channel a data processing control message containing first and second information derived from a selected data processing execution control structure requesting the execution of a data processing operation by said first means, and third information derived from fifth information stored within prescribed ones of said plurality of data processing execution control structures; and wherein

said first means includes means for asserting onto said second communications channel a data processing output message containing the identification of said selected data processing execution control structure and the result of the data processing operation carried out in accordance with the second and third information asserted onto said first communications channel.

17. A data processing system comprising:
a plurality of data processing nodes each of which includes

a functional computation unit in which data processing operations are executed on operand data in accordance with an opcode supplied thereto, so as to produce output data representative of the result of the execution of a data processing operation,

a program execution control unit which contains a plurality of data processing execution control structures, each of which data processing execution control structures comprises a plurality of entries including an address for identifying that data processing execution control structure, a plurality of operand source entries for specifying the addresses of operands to be employed in the execution of a data processing operation associated with that data processing execution control structure, and the status of said data processing execution control structure with respect to its associated data processing operation,

opcode storage means, coupled to said program execution control unit, for storing a plurality of opcodes respectively associated with said plurality of data processing execution control structures, a respective opcode defining a data processing operation to be performed by said functional computation unit,

operand storage means, coupled to said program execution control unit, for storing a plurality of result entries in which output data produced by said functional computation unit as a result of its execution of a data processing operation requested by a data processing execution control structure is stored,

data processing message assembly means, coupled to said program execution control unit, said operand storage means and said operand storage means, for assembling a plurality of data processing messages to be forwarded to said functional computation unit for execution, a respective data processing message including the identification of a respective data processing execution control structure, the contents of respective result entries identified by operand source addresses of said respective data processing execution control structure, and the opcode associated with said respective data processing execution control structure, and

a first communications channel, coupled between said data processing message assembly means and said functional computation unit, and being monitored simultaneously by each of the data processing execution control structures of said program execution control unit, for conveying data processing request messages from said data processing message assembly means to said functional computation unit,

a second communications channel, coupled between said functional computation unit, said program execution control unit storage unit and said operand storage means, and being monitored simultaneously by each of the data processing execution control structures of said program execution control unit, for conveying output data from said functional computation unit to said operand storage means and the identification of the data processing execution control structure for which a data processing request message has been processed by said functional computation unit to said program execution control unit;

first internode communication channel means, coupled to the second communication channel of each of said plurality of nodes, for simultaneously coupling the identification of the data processing execution control structure for which a data processing request message has been processed by its associated functional computation unit to the program execution control unit in each of said nodes; and

second internode communication channel means, coupled to the operand storage means, data processing message assembly means and program execution control means of each of said nodes, for enabling the operand addresses of a data processing execution control structure stored within the program execution control unit of a node to be simultaneously presented to the operand storage means of each of every other node, and for enabling operand value stored in any node to be simultaneously presented to the data processing message assembly means of any node.

18. A data processing system according to claim 17, wherein the status entry of a respective data processing execution control structure includes operand availability information representative of whether the result entry of another data processing execution control

structure in any of said plurality of nodes, whose address is defined by the contents of a source address entry of said respective data processing execution control structure, contains an operand required for the execution of a data processing operation defined in accordance with opcode entry of said respective data processing execution control structure.

19. A data processing system according to claim 18, wherein, within each node, said second communications channel includes a data portion over which output data from said functional computation unit is conveyed and a result address portion over which the address of said respective data processing execution control structure is conveyed, and said program execution control unit includes means for comparing the operand source entries of said respective data processing execution control structure with the contents of the address portion of said second communications channel and causing said operand availability information of said status entry to indicate that an operand required for the execution of a data processing operation defined in accordance with an opcode associated with said respective data processing execution control structure is available in the operand storage means of that one of said nodes which contains the data processing execution control structure whose identification matches one of the operand source address entries of said respective data processing execution control structure.

20. A data processing system according to claim 19, wherein said second communications channel further includes a result index portion for identifying one of the operand entries of a data processing execution control structure and said comparing means includes means for causing said operand availability information of said status entry to indicate that an operand entry required for the execution of a data processing operation defined in accordance with an opcode entry of said respective data processing execution control structure is available in the operand storage means of a node containing the data processing execution control structure whose address matches the operand source address entry of said respective data processing execution control structure as identified by said result index portion.

21. A data processing system according to claim 17, wherein, with a node, said operand storage means includes means for monitoring said first internode communications channel means and asserting thereon the contents of an operand entry, in response to recognizing the address of a data processing execution control structure contained within the program execution control unit of that node having been asserted on said first internode communications channel means, so that said operand entry may be employed as an operand for the execution of a data processing operation by a functional computation unit in one of said nodes.

22. A data processing system according to claim 21, wherein the status entry of a respective data processing execution control structure contained within the program execution control unit of a node includes acknowledgement information representative of whether another data processing execution control structure of any of said nodes requires the use of the operand contained within the operand storage means of said node whose address corresponds to identity of said respective data processing execution control structure.

23. A data processing system according to claim 22, wherein the status entry of said respective data processing execution control structure includes operand avail-

ability information representative of whether the operand storage means of any node has an address, which is defined by a source address entry of said respective data processing execution control structure and contains an operand required for the execution of a data processing operation, defined in accordance the opcode associated with said respective data processing execution control structure.

24. A data processing system according to claim 23, wherein the program execution control unit of a node includes means for indicating the readiness of said respective data processing execution control structure to have a data processing message asserted on said first communications channel in accordance with the contents of said status entry.

25. A data processing system according to claim 24, wherein said indicating means includes for indicating

the readiness of said respective data processing execution control structure to have a data processing message asserted on said first communications channel, in response to said acknowledgement information being representative that no other data processing execution control structure in any of said plurality of nodes requires the use of the contents of a storage location of the operand storage means of said node, the address of which storage location is the identity of said respective data processing execution control structure and that said operand availability information is representative that all operands required for the execution of a data processing operation defined in accordance with the opcode of said respective data processing execution control structure are available.

* * * * *

20

25

30

35

40

45

50

55

60

65



US006018353A

United States Patent [19]

[11] Patent Number: **6,018,353**

Deering et al.

[45] Date of Patent: ***Jan. 25, 2000**

[54] **THREE-DIMENSIONAL GRAPHICS ACCELERATOR WITH AN IMPROVED VERTEX BUFFER FOR MORE EFFICIENT VERTEX PROCESSING**

5,745,125	4/1998	Deering et al.	345/503
5,767,856	6/1998	Peterson et al.	345/503
5,793,371	8/1998	Deering	345/418
5,821,949	10/1998	Deering	345/505
5,842,004	11/1998	Deering et al.	345/501
5,867,167	2/1999	Deering	345/419
5,870,094	2/1999	Deering	345/419

[75] Inventors: **Michael F. Deering**, Los Altos;
Michael Neilly, Menlo Park, both of Calif.

OTHER PUBLICATIONS

[73] Assignee: **Sun Microsystems, Inc.**, Palo Alto, Calif.

OpenGL Architecture Review Board, "OpenGL Reference Manual," The Official Reference Document for OpenGL, Release 1, Nov. 1992, pp. 1-5.

[*] Notice: This patent issued on a continued prosecution application filed under 37 CFR 1.53(d), and is subject to the twenty year patent term provisions of 35 U.S.C. 154(a)(2).

"The OpenGL Machine," The OpenGL Graphics System Diagram, 1992 Silicon Graphics, Inc., 4 pages.

[21] Appl. No.: **08/885,280**

Primary Examiner—**Kec M. Tung**
Attorney, Agent, or Firm—**Conley, Rose & Tayon; Dan R. Christen; Jeffrey C. Hood**

[22] Filed: **Jun. 30, 1997**

[57] ABSTRACT

Related U.S. Application Data

[63] Continuation-in-part of application No. 08/511,294, Aug. 4, 1995, Pat. No. 5,793,371, and application No. 08/511,326, Aug. 4, 1995, Pat. No. 5,842,004.

A vertex accumulation buffer for improved three-dimensional graphical processing is disclosed. The accumulation buffer may include two individual buffers (buffers A and B) that each comprise a plurality of individual storage locations that are each configured to store vertex parameter values such as XYZ values, normal values, color information, and alpha information. The individual buffers serve to double buffer the vertex parameter values stored in the accumulation buffer. The storage locations may be written to on an individual basis without overwriting the other storage locations in the buffer.

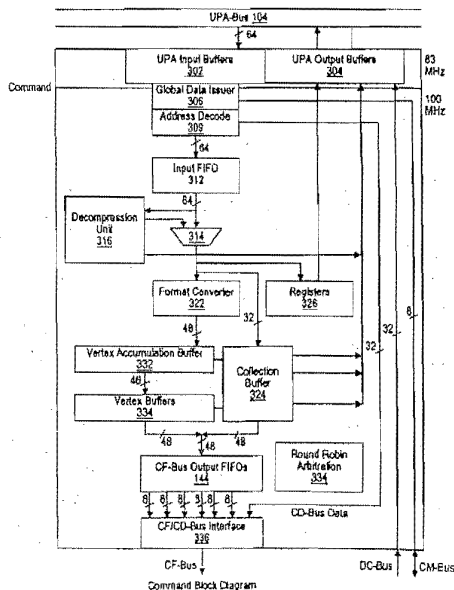
[51] Int. Cl.⁷ **G06F 13/00**
 [52] U.S. Cl. **345/511; 345/202**
 [58] Field of Search **345/508, 511, 345/513, 519, 501, 503, 509, 202**

[56] References Cited

U.S. PATENT DOCUMENTS

5,740,409 4/1998 Deering 345/503

20 Claims, 25 Drawing Sheets



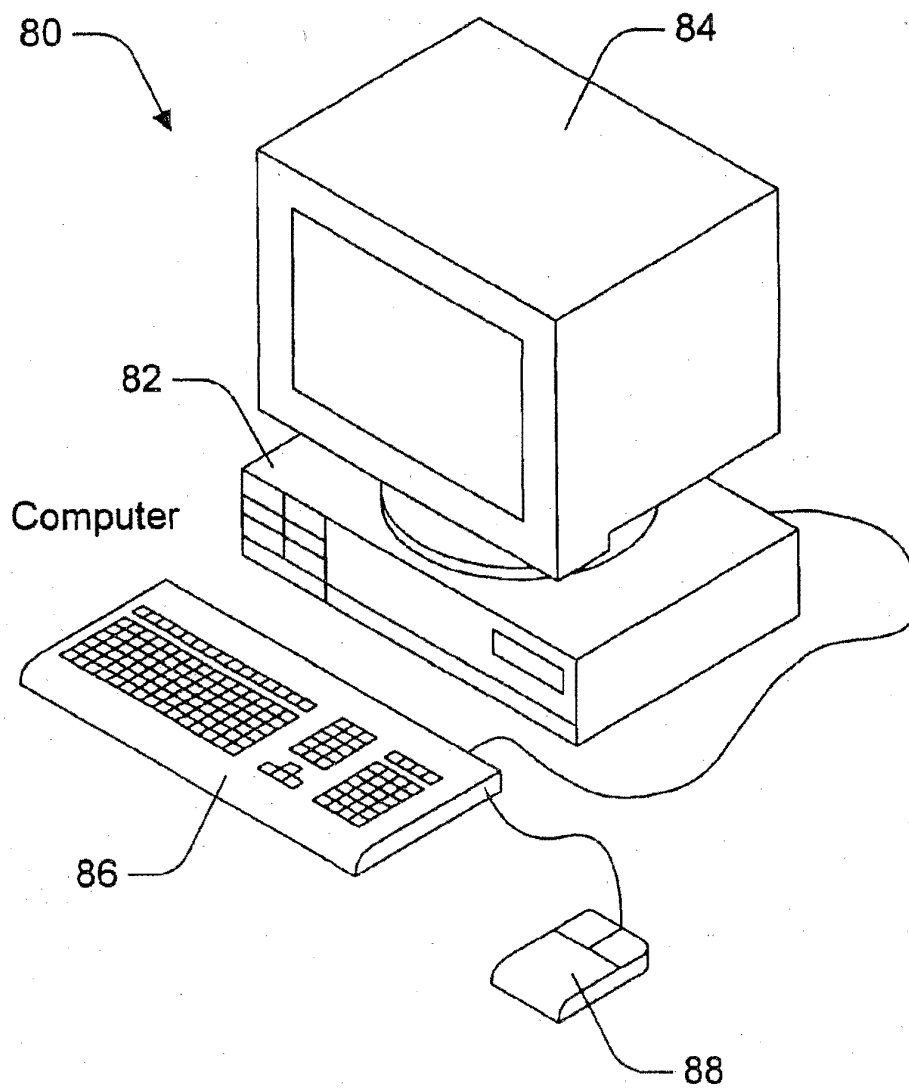


FIG. 1

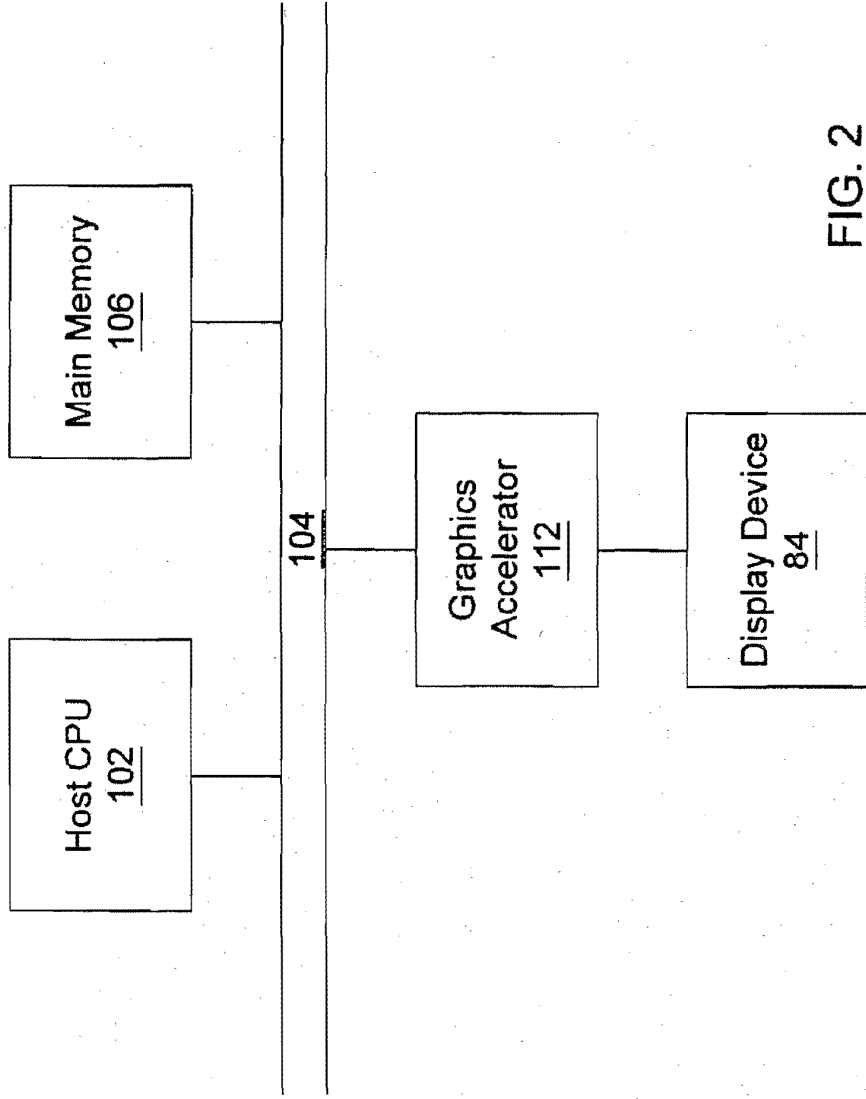


FIG. 2

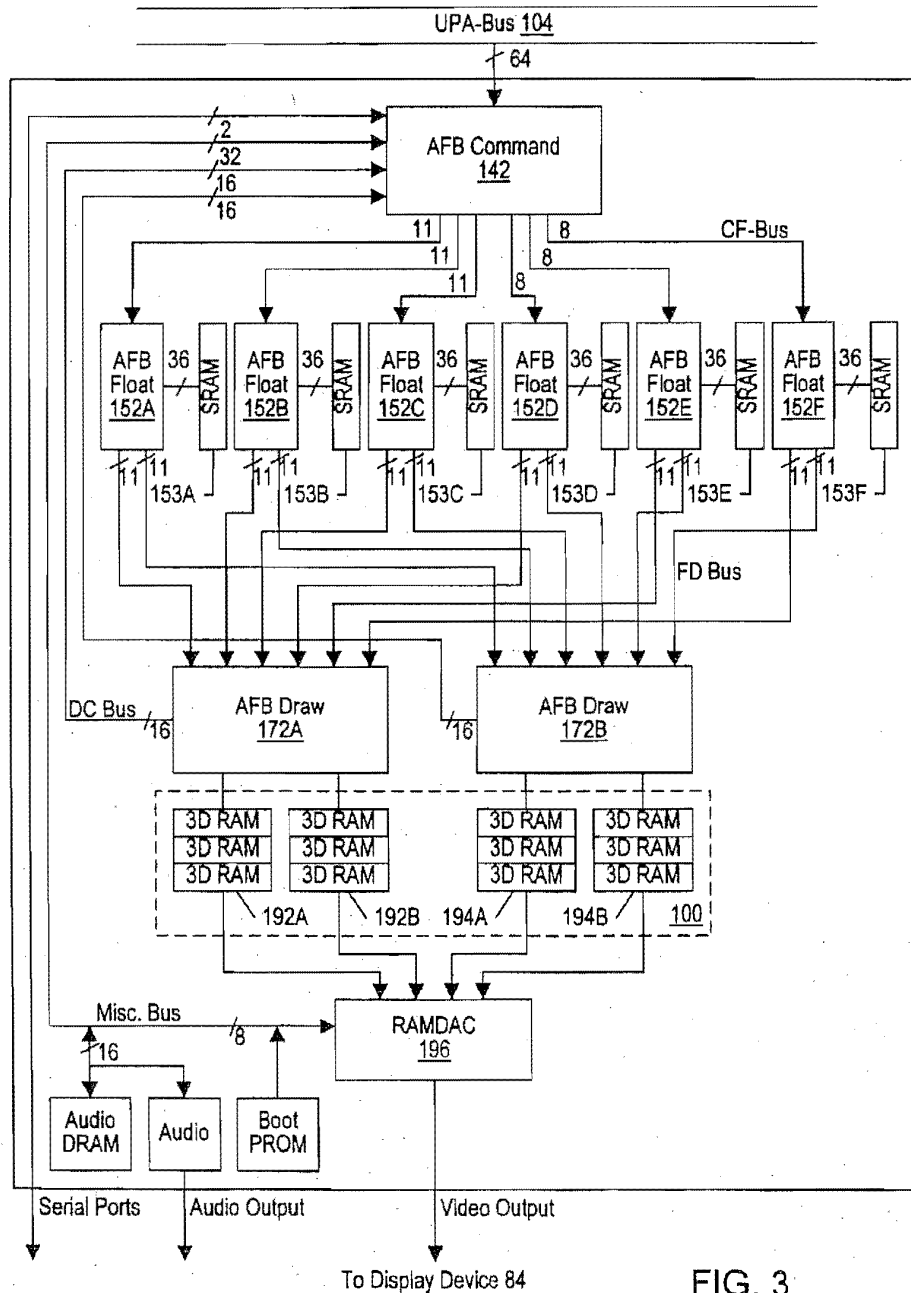


FIG. 3

WEST

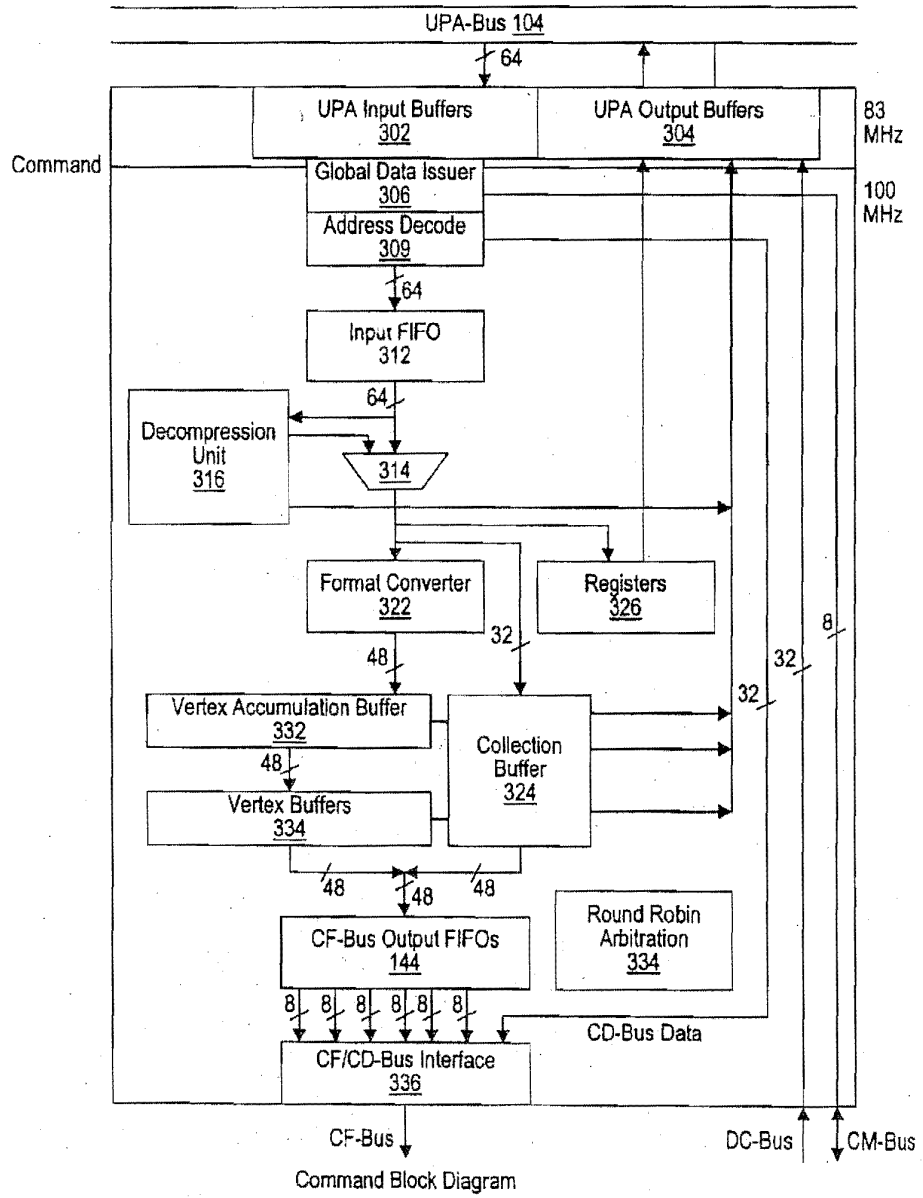
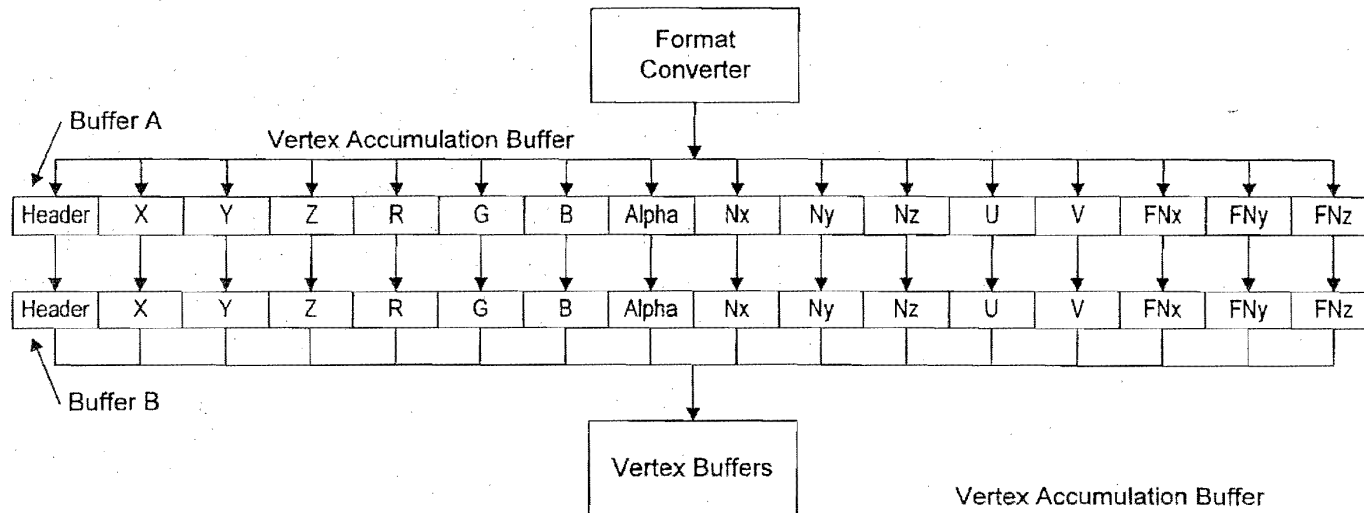
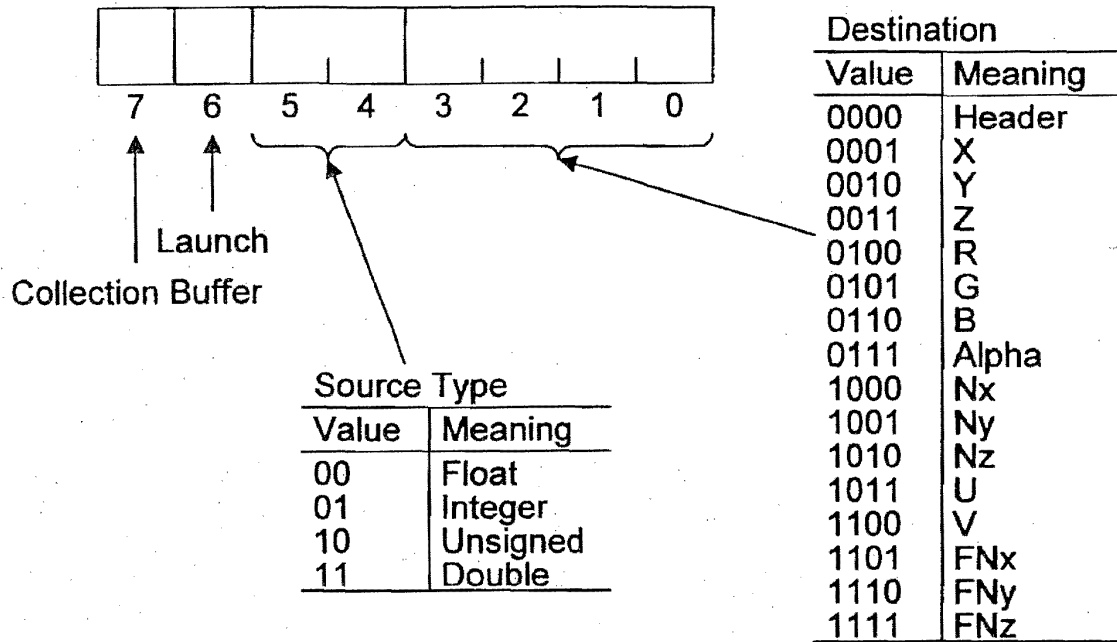


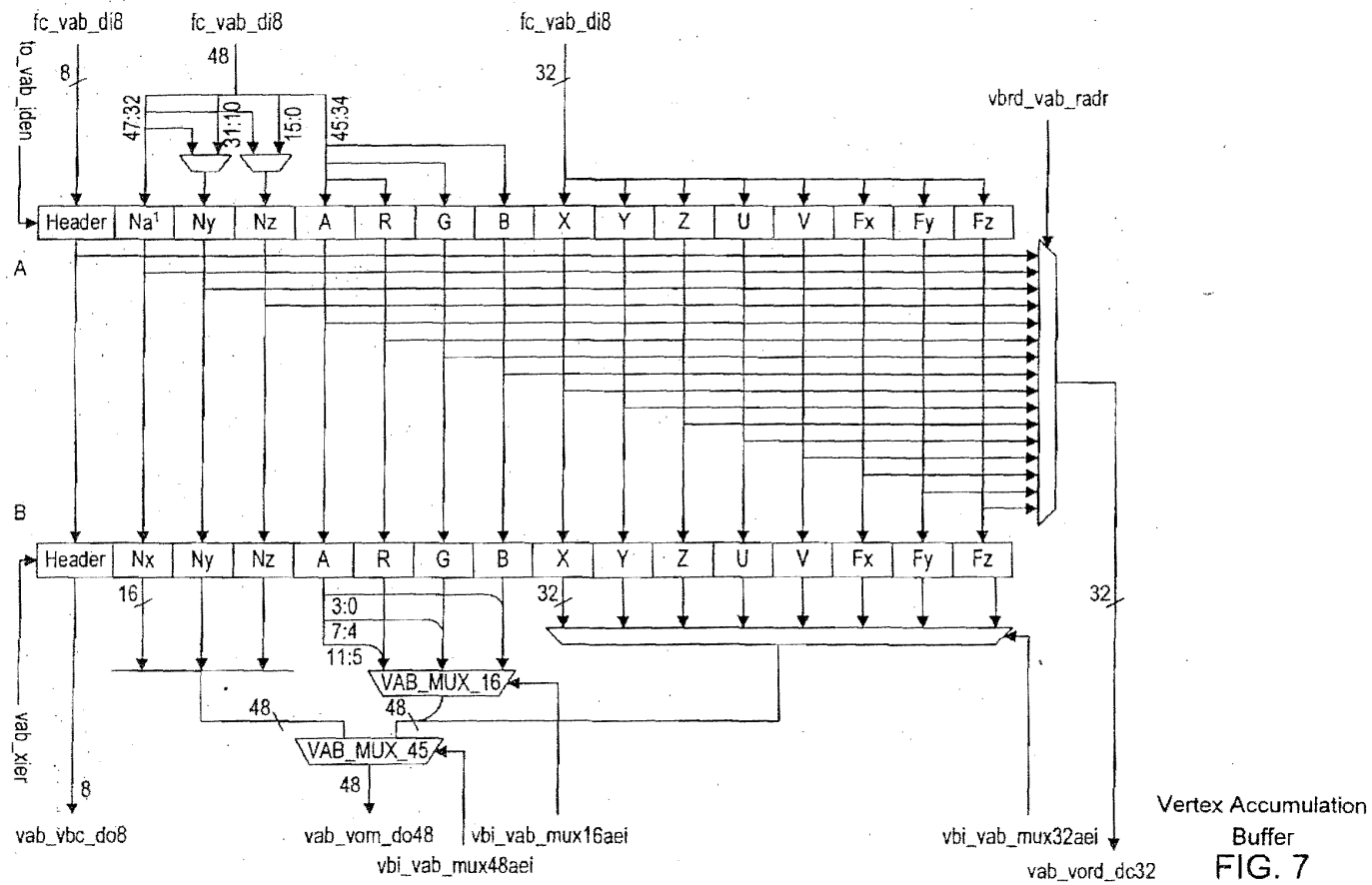
FIG. 4



Vertex Accumulation Buffer
FIG. 5



Format Converter Opcodes
FIG. 6

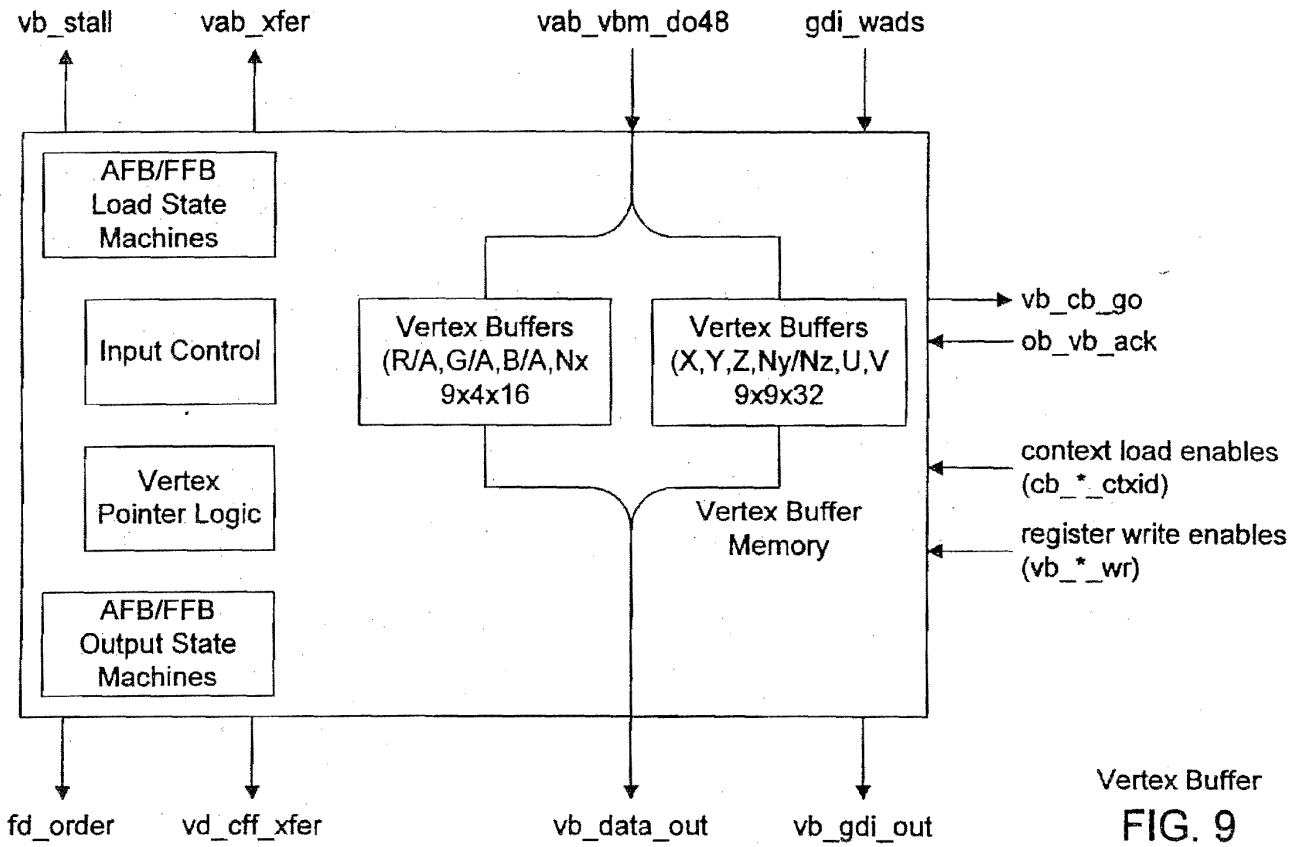


Vertex Accumulation Buffer
FIG. 7

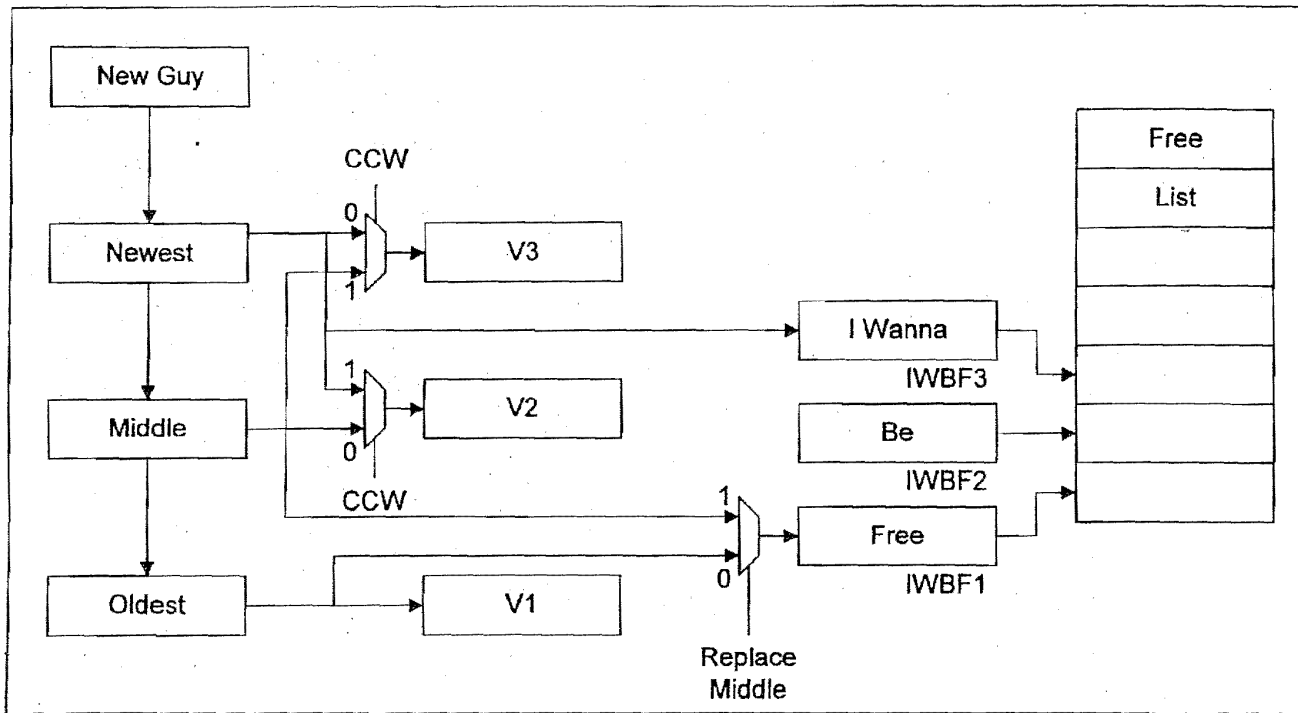
fc_vab_iden	VAB register loaded (A buffer)
0000 0000 0000 0000	None
0000 0000 0000 0001	Header *Header may be loaded at any time
0000 0000 0000 001x	X
0000 0000 0000 010x	Y
0000 0000 0000 100x	Z
0000 0000 0001 000x	R
0000 0000 0010 000x	G
0000 0000 0100 000x	B
0000 0000 1000 000x	A
0000 0001 0000 000x	NX
0000 0010 0000 000x	NY
0000 0100 0000 000x	NZ
0000 1000 0000 000x	U
0001 0000 0000 000x	V
0010 0000 0000 000x	FNX
0100 0000 0000 000x	FNY
1000 0000 0000 000x	FNZ
0000 0111 0000 000x	48 bit Normal (Nx,Ny,Nz)

Load enables to the Vertex Accumulation Buffer

FIG. 8

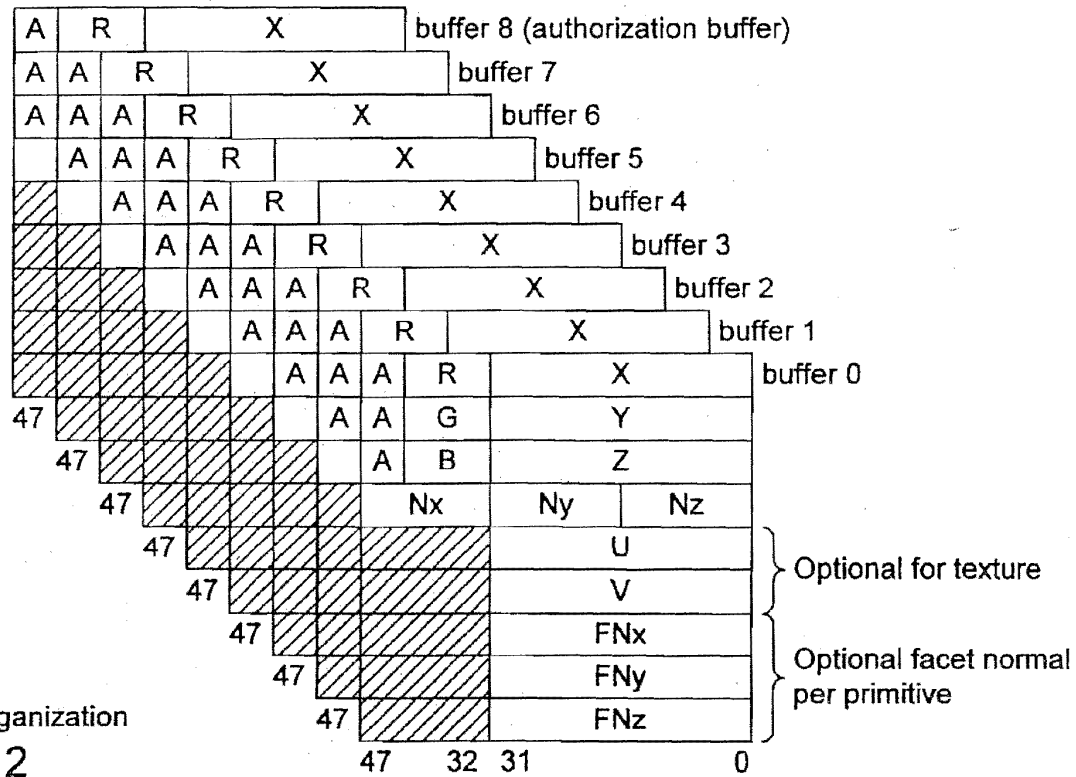


Vertex Buffer
FIG. 9



Vertex Buffer Control Logic

FIG. 11



Vertex Buffer Organization
FIG. 12

AFB Primitives

- Quads
- Triangles
- Lines
- Dots

Primitives that rely on the Primitive Control Register. The Format Converter asserts the fc_vb_launch signal for these primitives.

FFB Primitives

- Triangles
- Lines
- Dots
- Polygons
- Fast Fill
- Rectangles
- Vertical Scroll

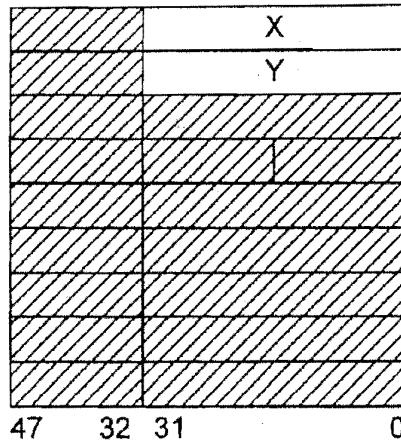
Primitives that use the vb_1dsm.

Primitives that rely on the FFB Opcode Register. The Format converter asserts a combination of the fc_vb_launch, fc_vb_ebxi and fc_vb_nxgo signals.

Primitives that use the vb_fldsm.

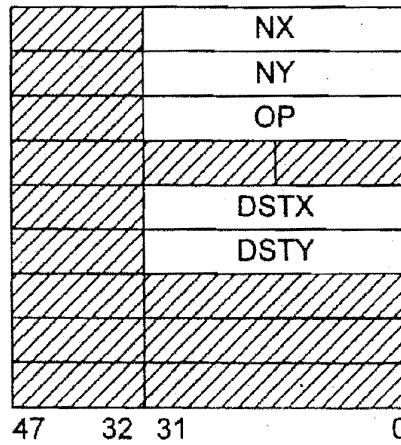
Selection Of Load State Machine

FIG. 13



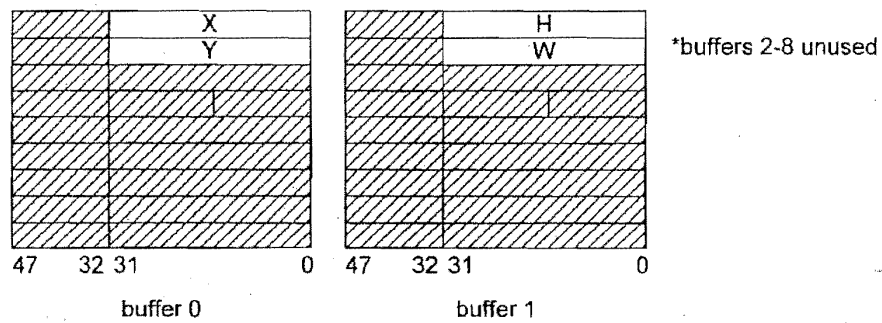
buffers 0-2 will be used
buffers 3-5 may be used
buffers 6-8 are unused

Vertex Buffer Storage Of FFB Polygons
FIG. 14

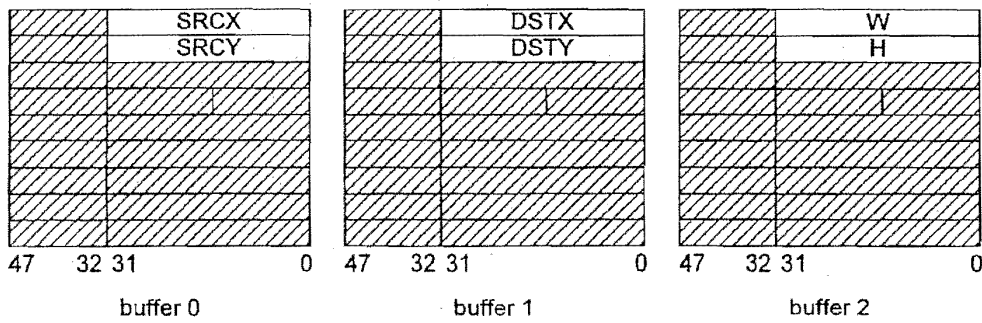


buffers 0-1 will be used
buffers 2-4 may be used
buffers 5-8 are unused

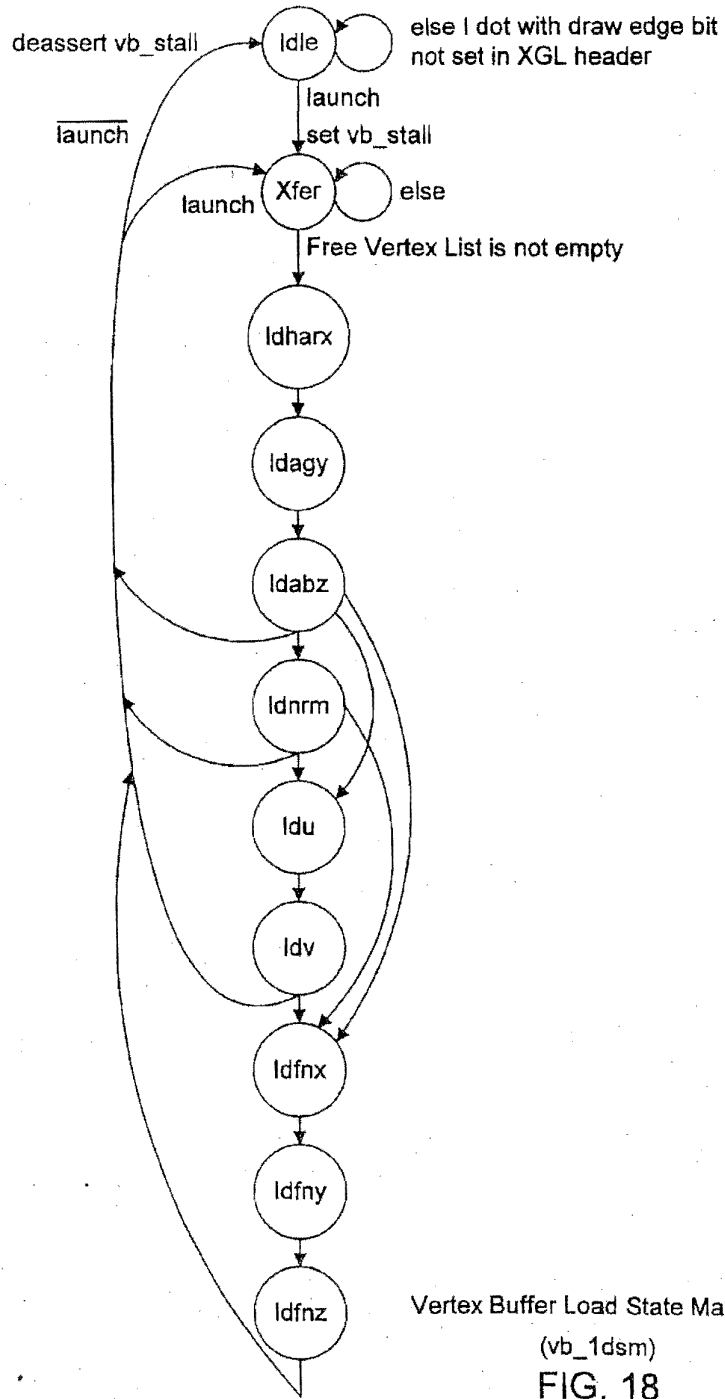
Vertex Buffer Storage Of FFB Fast Fill Primitives
FIG. 15

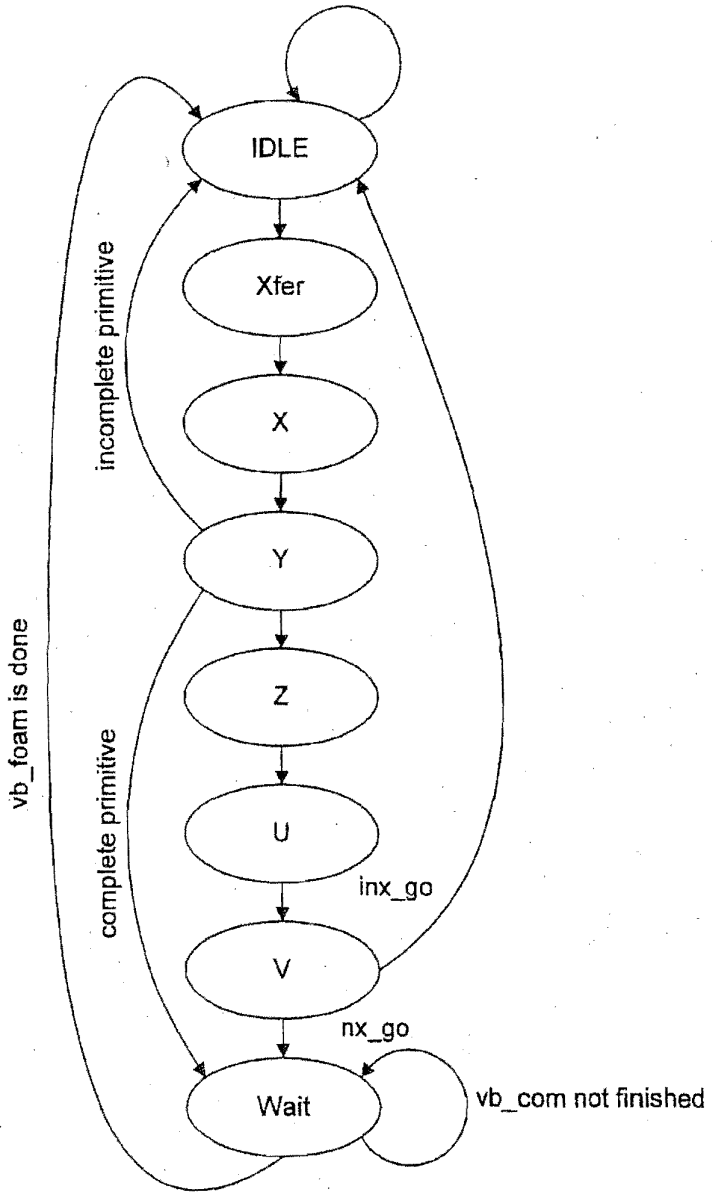


Vertex Buffer Storage Of FFB Rectangles
FIG. 16



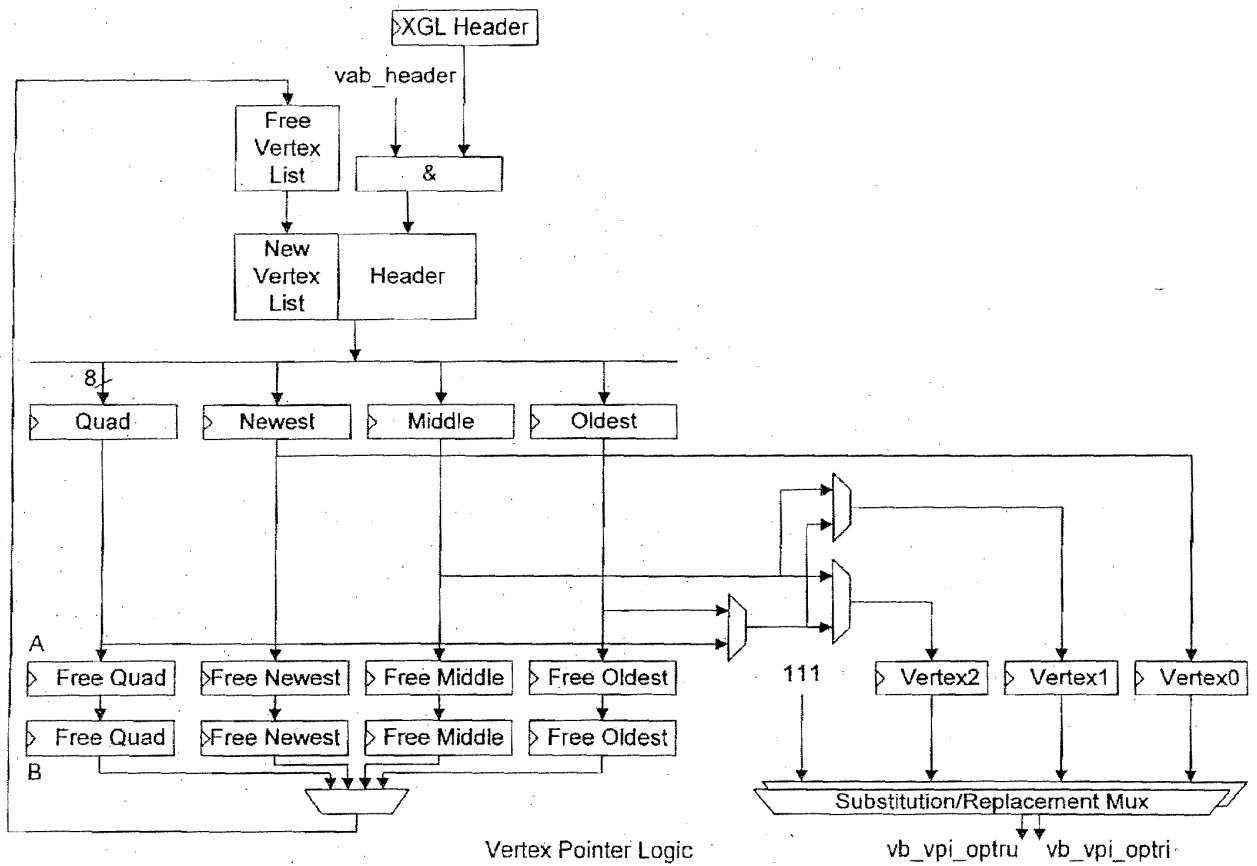
Vertex Buffer Organization For Vertical Scroll
FIG. 17



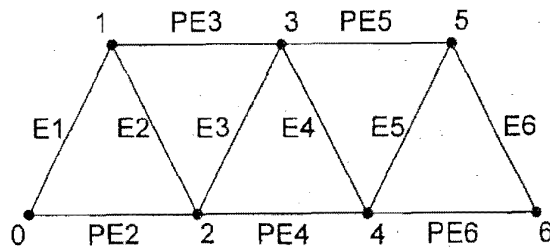


Vertex Buffer FFB Load State Machine

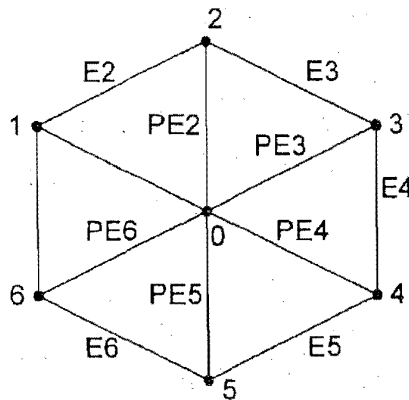
FIG. 19



Vertex Pointer Logic
FIG. 20



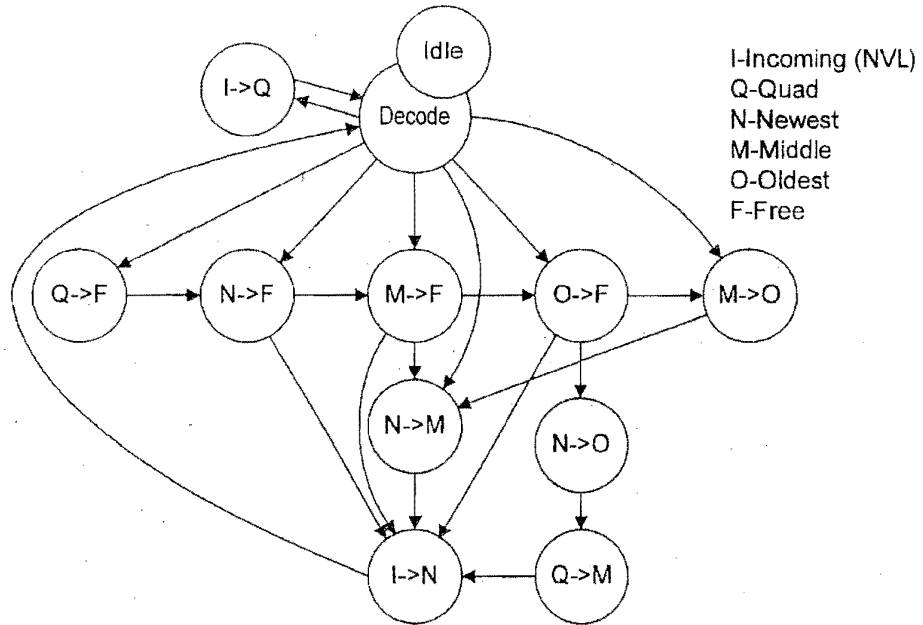
- 0 Restart
- 1 RO
- 2 RO
- 3 RO
- 4 RO
- 5 RO
- 6 RO



- 0 Restart
- 1 RM
- 2 RM
- 3 RM
- 4 RM
- 5 RM
- 6 RM

Edge Logic Explanation

FIG. 21

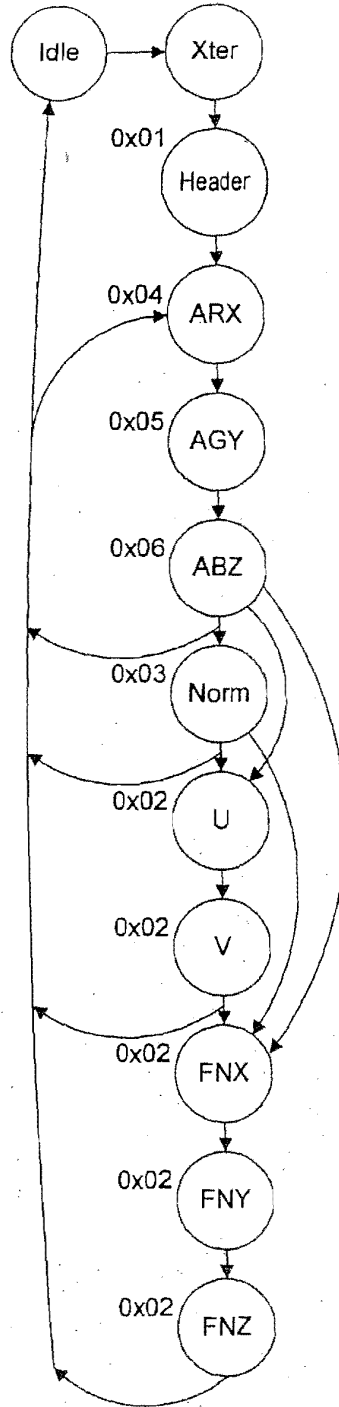


Valid transitions

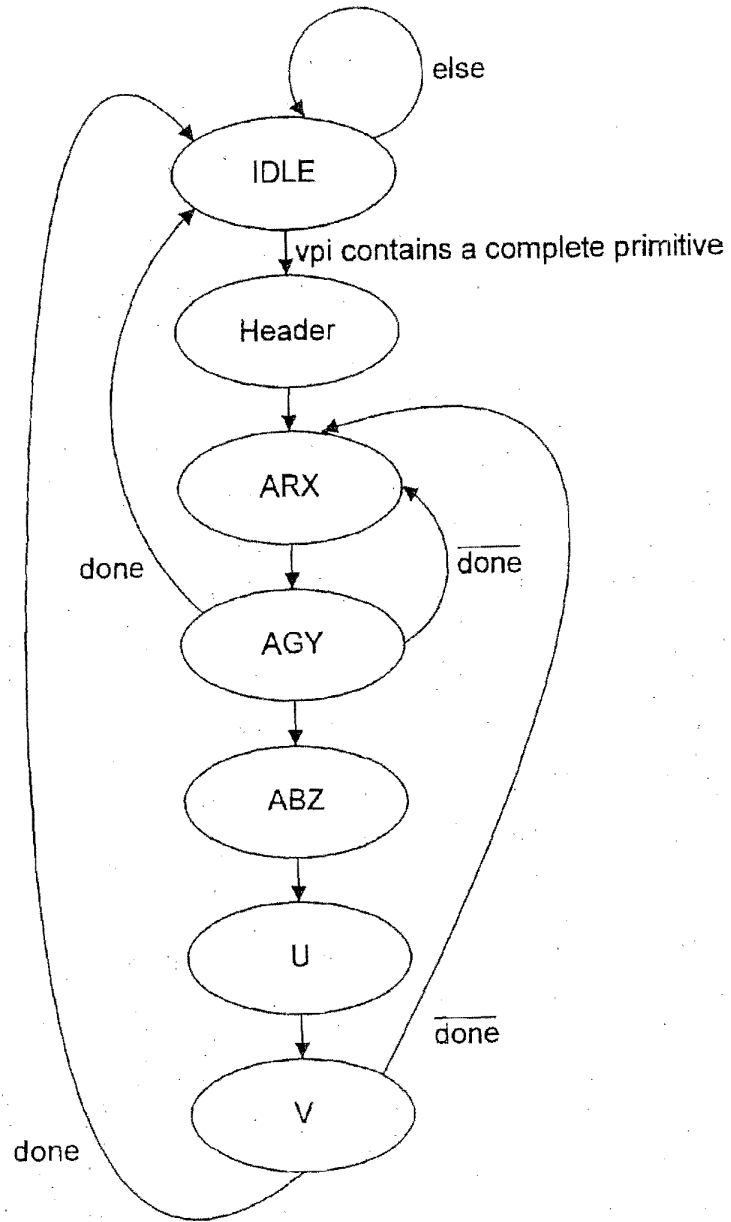
Condition	Transitions
Quad	
Restart	Q->F, N->F, M->F, O->F, I->N
Restart+1 cycle	N->M, I->N
Restart+2 cycle	M->O, N->M, I->N
*Restart+3cycle	I->Q
Replace Oldest	M->F, O->F, N->O, Q->M, I->N
*Replace Oldest + 1	I->Q
Triangle	
Restart	N->F, M->F, O->F, I->N
Restart + 1	M->O, N->M, I->N
*Restart + 2	M->O, N->N, I->N
*Replace Oldest	M->F, N->M, I->N
*Replace Middle	
Line	
Restart	N->F, M->F, I->N
Move	N->F, M->F, I->N
*Draw	N->M, I->N
Dot	
*Restart	N->F, I->N
*Draw	N->F, I->N

*Indicates a complete primitive

vb_vplsm
FIG. 22



vb_osm
FIG. 23



Vertex Buffer FFB Output State Machine (vb_fosm)

FIG. 24

Substitution Replication Control Register

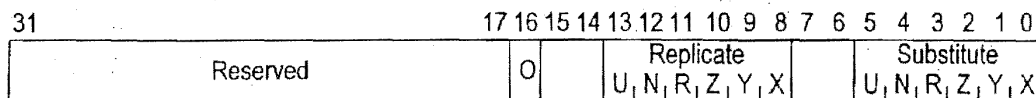


FIG. 25a

Primitive Control Register

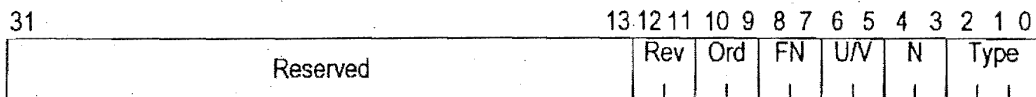


FIG. 25b

XGL Header Register

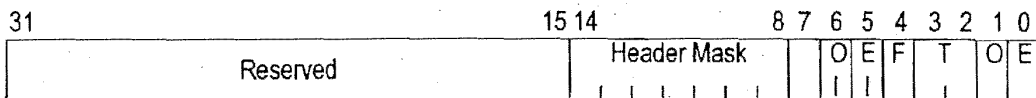


FIG. 25c

Float Enable Mask Register



FIG. 25d

Address	Source	Contents	Signals used to load
0x0060_1D60	NVL	{0000, nvl[4], 0000, nvl[3]}	cb_nvl_ctxld
0x0060_1D64	NVL	{0000, nvl[2], 0000, nvl[1]}	cb_nvl_ctxld
0x0060_1D68	NVL	{0000, nvl[0], 0000, rptr, 0, wptr, fcount}}	cb_nvl_ctxld
0x0060_1D6C	FVL	{0, rptr, 0, wptr, 0, fcount, fv1[4:0]}	cb_fvl_ctxld
0x0060_1D70	QNMO	{quad, new, middle, oldest}	cb_vpl_ctxld
0x0060_1D74	*reserved*	reserved for misc state bits	
0x0060_1D78	Free A	{000, freequada, 000, freenewa, 000, freemiddlea, 000, freeolda}	cb_vpl_ctxld
0x0060_1D7C	Free B	{000, freequadb, 000, freenewb, 000, freemiddleb, 000, freeoldb}	cb_vpl_ctxld

Vertex Buffer State Registers Address Map
FIG. 26

Address	Description
0x0060_1D80	Vertex Buffer 0
0x0060_1DC0	Vertex Buffer 1
0x0060_1E00	Vertex Buffer 2
0x0060_1E40	Vertex Buffer 3
0x0060_1E80	Vertex Buffer 4
0x0060_1EC0	Vertex Buffer 5
0x0060_1F00	Vertex Buffer 6
0x0060_1F40	Vertex Buffer 7
0x0060_1F80	Vertex Buffer 8
0x0060_1FC0	Vertex Accumulation Buffer

Vertex Buffer Memory and VAB Context Address Map

FIG. 27

**THREE-DIMENSIONAL GRAPHICS
ACCELERATOR WITH AN IMPROVED
VERTEX BUFFER FOR MORE EFFICIENT
VERTEX PROCESSING**

CONTINUATION DATA

This application is a continuation-in-part of application Ser. No. 08/511,294, filed Aug. 4, 1995 now U.S. Pat. No. 5,793,371, entitled METHOD AND APPARATUS FOR GEOMETRIC COMPRESSION OF THREE-DIMENSIONAL GRAPHICS DATA, and assigned to the assignee of this application.

This application is a continuation-in-part of application Ser. No. 08/511,326, filed Aug. 4, 1995 now U.S. Pat. No. 5,842,004, entitled METHOD AND APPARATUS FOR DECOMPRESSION OF COMPRESSED GEOMETRIC THREE-DIMENSIONAL GRAPHICS DATA, and assigned to the assignee of this application.

U.S. application Ser. No. 08/511,294, filed Aug. 4, 1995 now U.S. Pat. No. 5,793,371, entitled METHOD AND APPARATUS FOR GEOMETRIC COMPRESSION OF THREE-DIMENSIONAL GRAPHICS DATA, and assigned to the assignee of this application, is hereby incorporated by reference as though fully and completely set forth herein.

U.S. application Ser. No. 08/511,326, filed Aug. 4, 1995 now U.S. Pat. No. 5,842,004, entitled METHOD AND APPARATUS FOR DECOMPRESSION OF COMPRESSED GEOMETRIC THREE-DIMENSIONAL GRAPHICS DATA, and assigned to the assignee of this application, is hereby incorporated by reference as though fully and completely set forth herein.

FIELD OF THE INVENTION

The present invention relates to improved vertex pointer logic for assembling polygons from received geometry data in a three-dimensional graphics accelerator.

DESCRIPTION OF THE RELATED ART

A three dimensional (3-D) graphics accelerator is a specialized graphics rendering subsystem for a computer system which is designed to off-load the 3-D rendering functions from the host processor, thus providing improved system performance. In a system with a 3-D graphics accelerator, an application program executing on the host processor of the computer system generates three-dimensional geometry data that defines three-dimensional graphics elements for display on a video output device. The application program causes the host processor to transfer the geometry data to the graphics accelerator. The graphics accelerator receives the geometry data and renders the corresponding graphics elements on the display device.

Applications which display three-dimensional graphics require a tremendous amount of processing capabilities. For example, for a computer system to generate smooth 3-D motion video, the computer system is required to maintain a frame rate or update rate of between 20 to 30 frames per second. This requires a 3-D graphics accelerator capable of processing over a million graphics primitives per second.

In general 3-D graphics accelerators have had three major bottlenecks which limit performance. A first bottleneck is the transfer of geometric primitive data from main memory to the graphics accelerator over the system bus. A second bottleneck is the vertex processing requirements (such as transformation, lighting, and set-up) which are performed on

the geometric primitives by the graphics accelerator prior to rendering. A third bottleneck is the speed at which pixels from processed primitives can be filled into the frame buffer.

Vertex processing operations are typically performed by dedicated hardware in the graphics accelerator. This hardware is commonly pipelined, such that each stage of the pipeline effectuates a distinct operation on the vertices of the received geometric primitive. The operations may be performed in either fixed or floating-point math.

SUMMARY OF THE INVENTION

The present invention comprises improved vertex processing in a graphics accelerator.

A vertex accumulation buffer for improved three-dimensional graphical processing is disclosed. In one embodiment, the accumulation buffer may include two individual buffers (buffers A and B) that each comprise a plurality of individual storage locations. The individual storage locations are each configured to store vertex parameter values such as XYZ values, normal values, color information, and alpha information. The individual buffers serve to double buffer the vertex parameter values stored in the accumulation buffer. The storage locations may be written to on an individual basis without overwriting the other storage locations in the buffer.

In another embodiment, the vertex accumulation buffer may comprise a first buffer for storing a plurality of vertex values. The plurality of vertex values may include XYZ position values, red, green, and blue values, alpha values and normal values. The vertex accumulation buffer may further comprise a second buffer configured to receive and store copies of the plurality of vertex values. The first buffer may include a plurality of outputs (corresponding to each of the stored vertex values). The outputs may be coupled to corresponding inputs on the second buffer. The first buffer may be adapted to receive and store new vertex values. The old vertex values may remain unchanged in the first buffer until a new value overwrites the stored value. A graphics system configured to utilize the vertex accumulation buffer is also contemplated.

BRIEF DESCRIPTION OF THE DRAWINGS

A better understanding of the present invention can be obtained when the following detailed description of the preferred embodiment is considered in conjunction with the following drawings, in which:

FIG. 1 illustrates a computer system which includes a three dimensional (3-D) graphics accelerator according to the present invention;

FIG. 2 is a simplified block diagram of the computer system of FIG. 1;

FIG. 3 is a block diagram illustrating the 3-D graphics accelerator according to the preferred embodiment of the present invention;

FIG. 4 is a block diagram illustrating the command chip in the 3-D graphics accelerator according to the preferred embodiment of the present invention;

FIG. 5 illustrates the vertex accumulation buffer;

FIG. 6 illustrates format converter op-codes;

FIG. 7 is a more detailed diagram illustrating the vertex accumulation buffer;

FIG. 8 illustrates the valid assertions of the load enable lines to the vertex accumulation buffer;

FIG. 9 is a block diagram of the vertex buffer;

FIG. 10 illustrates organization of one of the vertex buffers;

FIG. 11 illustrates the vertex buffer control logic;

FIG. 12 is a more detailed diagram illustrating vertex buffer organization;

FIG. 13 lists the types of primitives supported by the vertex buffer as well as the primary control registers and state machines that handle the respective primitives;

FIG. 14 illustrates vertex buffer storage of FFB polygons;

FIG. 15 illustrates vertex buffer storage of FFB fast fill primitives;

FIG. 16 illustrates vertex buffer storage of FFB rectangles;

FIG. 17 illustrates vertex buffer organization for vertical scroll;

FIG. 18 illustrates the vertex buffer load state machine;

FIG. 19 illustrates the vertex buffer FFB load state machine;

FIG. 20 illustrates the vertex pointer logic;

FIG. 21 illustrates the relationship of edge bits to triangles;

FIG. 22 illustrates the vertex pointer logic state machine;

FIG. 23 illustrates the state diagram for the vertex buffer output state machine;

FIG. 24 illustrates the vertex buffer FFB output state machine;

FIGS. 25a-d illustrates user defined registers;

FIG. 26 illustrates the vertex buffer state registers address map; and

FIG. 27 illustrates the vertex buffer memory and VAB context address map.

DETAILED DESCRIPTION OF THE EMBODIMENTS

FIG. 1—Computer System

Referring now to FIG. 1, a computer system 80 which includes a three-dimensional (3-D) graphics accelerator according to the present invention is shown. As shown, the computer system 80 comprises a system unit 82 and a video monitor or display device 84 coupled to the system unit 82. The display device 84 may be any of various types of display monitors or devices. Various input devices may be connected to the computer system, including a keyboard 86 and/or a mouse 88, or other input. Application software may be executed by the computer system 80 to display 3-D graphical objects on the video monitor 84. As described further below, the 3-D graphics accelerator in computer system 80 includes a lighting unit which exhibits increased performance for handling of incoming color values of polygons used to render three-dimensional graphical objects on display device 84.

FIG. 2—Computer System Block Diagram

Referring now to FIG. 2, a simplified block diagram illustrating the computer system of FIG. 1 is shown. Elements of the computer system which are not necessary for an understanding of the present invention are not shown for convenience. As shown, the computer system 80 includes a central processing unit (CPU) 102 coupled to a high speed bus or system bus 104. A system memory 106 is also preferably coupled to the high speed bus 104.

The host processor 102 may be any of various types of computer processors, multi-processors and CPUs. The system memory 106 may be any of various types of memory subsystems, including random access memories and mass

storage devices. The system bus or host bus 104 may be any of various types of communication or host computer buses for communication between host processors, CPUs, and memory subsystems, as well as specialized subsystems. In the preferred embodiment, the host bus 104 is the UPB bus, which is a 64 bit bus operating at 83 MHz.

A 3-D graphics accelerator 112 according to the present invention is coupled to the high speed memory bus 104. The 3-D graphics accelerator 112 may be coupled to the bus 104 by, for example, a cross bar switch or other bus connectivity logic. It is assumed that various other peripheral devices, or other buses, may be connected to the high speed memory bus 104, as is well known in the art. It is noted that the 3-D graphics accelerator may be coupled to any of various buses, as desired. As shown, the video monitor or display device 84 connects to the 3-D graphics accelerator 112.

The host processor 102 may transfer information to and from the graphics accelerator 112 according to a programmed input/output (I/O) protocol over the host bus 104. Alternately, the graphics accelerator 112 accesses the memory subsystem 106 according to a direct memory access (DMA) protocol or through intelligent bus mastering.

A graphics application program conforming to an application programmer interface (API) such as OpenGL generates commands and data that define a geometric primitive such as a polygon for output on display device 84. As defined by the particular graphics interface used, these primitives may have separate color properties for the front and back surfaces. Host processor 102 transfers these commands and data to memory subsystem 106. Thereafter, the host processor 102 operates to transfer the data to the graphics accelerator 112 over the host bus 104. Alternatively, the graphics accelerator 112 reads in geometry data arrays using DMA access cycles over the host bus 104. In another embodiment, the graphics accelerator 112 is coupled to the system memory 106 through a direct port, such as the Advanced Graphics Port (AGP) promulgated by Intel Corporation. As will be described below, graphics accelerator 112 is advantageously configured to more efficiently produce polygons to be rendered from received geometry data.

FIG. 3—Graphics Accelerator

Referring now to FIG. 3, a block diagram is shown illustrating the graphics accelerator 112 according to the preferred embodiment of the present invention. As shown, the graphics accelerator 112 is principally comprised of a command block 142, a set of floating-point processors 152A-152F, a set of draw processors 172A and 172B, a frame buffer 100 comprised of 3DRAM, and a random access memory/digital-to-analog converter (RAMDAC) 196.

As shown, the graphics accelerator 112 includes command block 142 which interfaces to the memory bus 104. The command block 142 interfaces the graphics accelerator 112 to the host bus 104 and controls the transfer of data between other blocks or chips in the graphics accelerator 112. The command block 142 also pre-processes triangle and vector data and performs geometry data decompression.

The command block 142 interfaces to a plurality of floating point blocks 152. The graphics accelerator 112 preferably includes up to six floating point processors labeled 152A-152F, as shown. The floating point processors 152A-152F receive high level drawing commands and generate graphics primitives, such as triangles, lines, etc. for rendering three-dimensional objects on the screen. The floating point processors 152A-152F perform transformation, clipping, face determination, lighting and

set-up operations on received geometry data. Each of the floating point processors 152A-152F connects to a respective memory 153A-153F. The memories 153A-153F are preferably 32 kx36-bit SRAM and are used for microcode and data storage.

Each of the floating point blocks 152A-F connects to each of two draw processors 172A and 172B. The graphics accelerator 112 preferably includes two draw processors 172A and 172B, although a greater or lesser number may be used. The draw processors 172A and 172B perform screen space rendering of the various graphics primitives and operate to sequence or fill the completed pixels into the 3DRAM array. The draw processors 172A and 172B also function as 3DRAM control chips for the frame buffer 100. The draw processors 172A and 172B concurrently render an image into the frame buffer 100 according to a draw packet received from one of the floating-point processors 152A-152F, or according to a direct port packet received from the command processor 142.

Each of the floating point blocks 152A-F preferably operates to broadcast the same data to the two drawing blocks 172A and 172B. In other words, the same data is always on both sets of data lines coming from each floating point block 152. Thus, when the floating point block 152A transfers data, the floating point block 152A transfers the same data over both parts of the FD-bus to the draw processors 172A and 172B.

Each of the respective drawing blocks 172A and 172B couple to frame buffer 100, wherein frame buffer 100 comprises four banks of 3DRAM memory 192A-B, and 194A-B. The draw processor 172A couples to the two 3DRAM banks 192A and 192B, and the draw processor 172B couples to the two 3DRAM banks 194A and 194B, respectively. Each bank comprises three 3DRAM chips, as shown. The 3DRAM memories or banks 192A-B and 194A-B collectively form the frame buffer 100, which is 1280x1024 by 96 bits deep. The frame buffer stores pixels corresponding to 3-D objects which are rendered by the draw processors 172A and 172B.

Each of the 3DRAM memories 192A-B and 194A-B couple to a RAMDAC (random access memory digital-to-analog converter) 196. The RAMDAC 196 comprises a programmable video timing generator and programmable pixel clock synthesizer, along with cross-bar functions, as well as traditional color look-up tables and triple video DAC circuits. The RAMDAC in turn couples to the video monitor 84.

The command block is preferably implemented as a single chip. Each of the floating point processors 152 are preferably implemented as separate chips. In the preferred embodiment, up to six floating point blocks or chips 152A-F may be included. Each of the drawing blocks or processors 172A and 172B also preferably comprise separate chips. For more information on different aspects of the graphics accelerator architecture of the preferred embodiment, please see related co-pending application Ser. No. 08/673,492 entitled "Three-Dimensional Graphics Accelerator With Direct Data Channels for Improved Performance", and related co-pending application Ser. No. 08/673,491 entitled "Three-Dimensional Graphics Accelerator Which Implements Multiple Logical Buses Using Common Data Lines for Improved Bus Communication", both filed on Jul. 1, 1996.

As described above, command block 142 interfaces with host bus 104 to receive graphics commands and data from host CPU 102. These commands and data (including polygons with both front and back surface properties) are passed in turn to floating point processors 152 for transformation,

lighting, and setup calculations. The output data is then provided to the draw chips for rendering into the frame buffer. As described further below, the command block includes improved vertex pointer logic according to the present invention, which more efficiently creates complete polygons from received geometry data.

FIG. 4—Command Block

As discussed above, the command preprocessor or command block 142 is coupled for communication over the host bus 104. The command preprocessor 142 receives geometry data arrays transferred from the memory subsystem 106 over the host bus 28 by the host processor 102. In the preferred embodiment, the command preprocessor 142 receives data transferred from the memory subsystem 106, including both compressed and non-compressed geometry data. When the command preprocessor 142 receives compressed geometry data, the command preprocessor 142 operates to decompress the geometry data to produce decompressed geometry data.

The command preprocessor 142 preferably implements two data pipelines, these being a 3D geometry pipeline and a direct port pipeline. In the direct port pipeline, the command preprocessor 142 receives direct port data over the host bus 104, and transfers the direct port data over the command-to-draw (CD) bus to the draw processors 172A-172B. As mentioned above, the CD bus uses or "borrows" portions of other buses to form a direct data path from the command processor 142 to the draw processor 172A-172B. The direct port data is optionally processed by the command preprocessor 142 to perform X11 functions such as character writes, screen scrolls and block moves in concert with the draw processors 172A-172B. The direct port data may also include register writes to the draw processors 172A-172B, and individual pixel writes to the frame buffer 3DRAM 192 and 194.

In the 3D geometry pipeline, the command preprocessor 142 accesses a stream of input vertex packets from the geometry data arrays. When the command preprocessor 142 receives a stream of input vertex packets from the geometry data arrays, the command preprocessor 142 operates to reorder the information contained within the input vertex packets and optionally delete information in the input vertex packets. The command preprocessor 142 preferably converts the received data into a standard format. The command preprocessor 142 converts the information in each input vertex packet from differing number formats into the 32 bit IEEE floating-point number format. The command preprocessor 142 converts 8 bit fixed-point numbers, 16 bit fixed-point numbers, and 32 bit or 64 bit IEEE floating-point numbers. For normal and color values, the command preprocessor 142 may convert the data to a fixed point value.

The command preprocessor 142 operates to accumulate input vertex information until an entire primitive is received. The command preprocessor 142 then transfers output geometry packets or primitive data over the command-to-floating-point (CF) bus to one of the floating-point processors 152A-152F. The output geometry packets comprise the reformatted vertex packets with optional modifications and data substitutions.

Referring now to FIG. 4, a block diagram illustrating the command processor or command block 142 is shown. As shown, the command block 142 includes input buffers 302 and output buffers 304 for interfacing to the host bus 104. The input buffers 302 couple to a global data issuer 306 and address decode logic 308. The global data issuer 306 connects to the output buffers 304 and to the CM bus and performs data transfers. The address decode logic 308 receives an input from the DC bus as shown. The address

decode logic 308 also couples to provide output to an input FIFO buffer 312.

In general, the frame buffer has a plurality of mappings, including an 8-bit mode for red, green and blue planes, a 32-bit mode for individual pixel access, and a 64-bit mode to access the pixel color together with the Z buffer values. The boot prom 197, audio chip 198 and RAMDAC 196 also have an address space within the frame buffer. The frame buffer also includes a register address space for command block and draw processor registers among others. The address decode logic 308 operates to create tags for the input FIFO 312, which specify which logic unit should receive data and how the data is to be converted. The input FIFO buffer 312 holds 128 64-bit words, plus a 12-bit tag specifying the destination of data and how the data should be processed.

The input FIFO 312 couples through a 64-bit bus to a multiplexer 314. Input FIFO 312 also provides an output to a geometry decompression unit 316. As discussed above, the command block 142 receives both compressed and non-compressed geometry data. The decompression unit 316 receives the compressed geometry data and operates to decompress this compressed geometry data to produce decompressed geometry data. The decompression unit 316 receives a stream of 32-bit words and produces decompressed geometry or primitive data. Then decompressed geometry data output from the decompression unit 316 is provided to an input of the multiplexer 314. The output of the multiplexer 314 is provided to a format converter 322, a collection buffer 324 and register logic 326. In general, the decompressed geometry data output from the decompression unit is provided to either the format converter 322 or the collection buffer 324.

In essence, the geometry decompression unit 316 can be considered a detour on the data path between the input FIFO 312 and the next stage of processing, which is either the format converter 322 or the collection buffer 324. For data received by the command processor 142 which is not compressed geometry data, i.e., non-compressed data, this data is provided from the input FIFO 312 directly through the multiplexer 314 to either the format converter 322, the collection buffer 324, or the register logic 326. When the command processor 142 receives compressed geometry data, this data must first be provided from the input FIFO 312 to the geometry decompression unit 316 to be decompressed before being provided to other logic.

Thus, the command block 142 includes a first data path coupled to the input buffers 302 or input FIFO 312 for transferring the non-compressed geometry data directly through the multiplexer 314 to either the format converter 322 or the collection buffer 324. The command block 142 also includes a second data path coupled to the input buffers 302 or input FIFO 312 for receiving compressed geometry data. The second data path includes a geometry decompression unit coupled to an output of the input FIFO 312 for receiving and decompressing the compressed geometry input data to produce decompressed geometry input data.

The format converter 322 receives integer and/or floating point data and outputs either floating point or fixed point data. The format converter 322 provides the command processor 142 the flexibility to receive a plurality of different data types while providing each of the floating block units 152A-152F with only a single data type for a particular word.

The format converter 322 provides a 48-bit output to a vertex accumulation buffer 332. The vertex accumulation buffer 332 in turn provides an output to vertex buffers 334. The

vertex accumulation buffer 332 and the vertex buffers 334 provide outputs to the collection buffer 324, which in turn provides an output back to the output buffers 304.

The vertex accumulation buffer 332 is used to store or accumulate vertex data required for a primitive that is received from the format converter 322. The vertex accumulation buffer 332 actually comprises two sets of registers, i.e., is double buffered. The first set of registers is used for composing a vertex, and the second set of registers is used for copying the data into one of the vertex buffers 334. As discussed further below, these two sets of registers allow for more efficient operation. Data words are written one at a time into the first or top buffer of the vertex accumulation buffer 332, and these values remain unchanged until a new value overwrites the respective word. Data is transferred from the first set of registers to the second set of registers in one cycle when a launch condition occurs.

The vertex buffers 334 are used for constructing or "building up" geometric primitives, such as lines, triangles, etc. Lines and triangles require two and three vertices, respectively, to complete a primitive. According to one embodiment of the invention, new primitives may be created by replacing a vertex of an existing primitive when the primitive being created shares one or more vertices with the prior created primitive. In other words, the vertex buffers 334 remember or maintain previous vertex values and intelligently reuse these vertex values when a primitive or triangle shares one or more vertices or other information with a neighboring primitive or triangle. This reduces the processing requirements and makes operation of the Open GL format operate more efficiently. In the preferred embodiment, the vertex buffers 334 can hold up to seven vertices. This guarantees maximum throughput for the worse case primitive, i.e., independent triangles. The vertex buffers 334 also operate at optimum speed for dots, lines and triangles and is substantially optimal for quad primitives.

Each of the vertex accumulation buffer 332 and the vertex buffers 334 are coupled to a collection buffer 324. The collection buffer 324 provides respective outputs to the output buffers 304 as shown. The vertex buffers 334 are coupled to provide outputs to CF bus output FIFOs 144. The collection buffer 324 is also coupled to provide outputs to the CF bus output FIFOs 144. The collection buffer 324 is used for sending all non-geometric data to the floating point blocks 152A-152F. The collection buffer 324 can hold up to 32 32-bit words. It is noted that the operation of copying data into the CF-bus output FIFOs 144 may be overlapped with the operation of copying new data into the collection buffer 324 for optimal throughput.

As mentioned above, the command block 142 includes a plurality of registers 326 coupled to the output of the multiplexer 314. The registers 326 also provide an output to the UPA output buffers 304. Register block 326 comprises 16 control and status registers which control the format and flow of data being sent to respective floating point blocks 152A-152F.

Each of the vertex buffers 334 and the collection buffer 324 provides a 48-bit output to CF-bus output FIFOs 144. The CF-bus output FIFOs 144 enable the command block 142 to quickly copy a primitive from the vertex buffers 334 into the output FIFO 144 while the last of the previous primitive is still being transferred across the CF-bus. This enables the graphics accelerator 112 to maintain a steady flow of data across each of the point-to-point buses. In the preferred embodiment, the CF-bus output FIFOs 144 have sufficient room to hold one complete primitive, as well as additional storage to smooth out the data flow. The CF

output FIFOs 144 provide respective 8-bit outputs to a bus interface block 336. The bus interface 336 is the final stage of the command processor 142 and couples to the CF-bus as shown. In addition, the CF/CD bus interface 336 provides "direct port" accesses to the CDC bus which are multiplexed on the CF-bus as mentioned above.

The command block 142 also includes round robin arbitration logic 334. This round robin arbitration logic 334 comprises circuitry to determine which of the respective floating point processors 152A-152F is to receive the next primitive. As discussed above, the graphics accelerator 112 of the present invention comprises separate point-to-point buses both into and out of the respective floating point processors 152A-152F. Thus, the round robin arbitration logic 334 is included to distribute primitives evenly between the chips and thus maintain an even flow of data across all of the point-to-point buses simultaneously. In the preferred embodiment, the round robin arbitration logic 334 utilizes a "next available round robin" arbitration scheme, which skips over a sub-bus that is backed up, i.e., full.

For information on another embodiment of the command processor 142, please see U.S. Pat. No. 5,408,605 titled "Command Preprocessor for a High Performance Three Dimensional Graphics Accelerator", which is hereby incorporated by reference in its entirety.

Vertex Buffer System

The Vertex Buffer organizes incoming vertices into primitives to be loaded into the CF bus output fifos for delivery to the AFB-Float ASICs. These manipulations include face orientation, substitution, replication, edge processing, and vertex ordering. These operations are handled by various pieces of the Vertex Buffer, which are discussed below.

Vertex Accumulation Buffer

The Vertex Accumulation buffer facilitates OpenGL operation, and also simplifies other operation of the graphics accelerator. FIG. 5 shows the Vertex Accumulation buffer together with the other modules in the AFB-Command chip to which it is connected. Data comes into the VAB from the Format Converter and is written to one of the Vertex Buffers.

Incoming data is written to Buffer A of the Vertex Accumulation Buffer. There is a 7-bit word for the header, three 32-bit words for X, Y and Z, four 12-bit words for R, G, B and Alpha, three 16-bit words for N_x , N_y , and N_z , two more 32-bit words for U and V (texture coordinates), and three 32-bit words for FN_x , FN_y , and FN_z (the facet normal). These words are written one at a time and remain unchanged until a new value overwrites the word. The feature of the words remaining the same "forever" allows a color, normal or Z value to be set in this buffer, with no need for other constant registers. It also permits the data to be written in any order.

When a "launch" condition occurs, the entire contents of Buffer A is written into Buffer A in one cycle. New values may then be written immediately to Buffer A while the contents of Buffer B is being copied into the appropriate Vertex Buffer. The transfer into the Vertex Buffer is accomplished 48 bits at a time (see FIG. 4-6 for the format of the 48-bit words). For OpenGL mode and some of the XGL modes, a write to an explicit address causes the launch condition. For bcopy mode in XGL the Format Converter Op-codes determine when to launch a vertex. For decompression mode the current mode and a counter determine when a launch condition has been reached.

A major advantage of this design over prior art designs is that there are no "dead cycles" during the data transfer on either side of the Vertex Accumulation Buffer.

Incoming Header Word

The incoming header word is defined to exactly match the XGL bit definition. The seven bits of this header word are defined as follows:

Bit 0	Draw edge
Bit 1	Draw previous edge
Bits 2-3	Triangle replace bits
Bit 4	Face orientation (CCW)
Bit 5	Edge is internal
Bit 6	Previous edge is internal

The individual bits have the following meanings:

Draw edge: For lines, this is the same as a move/draw bit. When zero the line starting position is specified and when one, a line is drawn from the previous point to the current point. For dots, the dot is not drawn when this bit is zero. When drawing triangle edges, this bit indicates that an edge is to be drawn from the newest vertex to the middle vertex.

Draw previous edge: This bit only applies while drawing triangle edges and indicates that an edge should be drawn from the newest vertex to the oldest vertex.

Triangle replace bits: A value of 00 in these two bits indicates to restart the triangle. The next two vertices received will complete the triangle, no matter what the value of the replace bits. That is to say, the replace bits are always ignored for the second and third vertices after a restart. A value of 01 indicates that the oldest of the three existing vertices is to be discarded in forming a new triangle. A value of 10 indicates that the middle of the three existing vertices is to be discarded in forming a new triangle.

Face Orientation: The face orientation bit is only used on a restart and is exclusive-Or'd with the CCW bit of the Primitive Control Register to determine the current winding bit used when outputting primitives.

Note: The CCW bit in both the GT and ZX graphics accelerators was specified assuming a left-handed coordinate system (X positive up, Y positive to the right, Z positive going away from the viewer) as needed by PHIGS. This is actually backwards for XGL, which uses a right-handed coordinate system (Z is now positive coming towards the viewer). AFB will differ from its predecessors by specifying the CCW bit for a right-handed coordinate system.

Edge is internal: This bit is used when drawing hollow triangles and indicates that the edge from the most recent vertex to the middle vertex is an internal edge (part of a larger polygon) and is not to be drawn.

Previous edge is internal: Same as the above, but for the edge from the most recent vertex to the oldest vertex.

Format Converter Controller

When running in "immediate mode," both XGL and OpenGL store data directly to the appropriate Vertex Accumulation Buffer registers based on the address to which the data is written. The addresses also specify to the Format Converter how the data is to be handled. However, when data is copied to AFB-Command in large blocks using bcopy, it can't be written to the required addresses that make immediate mode work. Some other way is required to specify how many words make up a vertex and how each word is to be treated.

The Format Converter Controller at the bottom of the Input FIFO contains opcodes to specify how incoming data streams should be dealt with. The op-code format is shown in FIG. 6. The Destination field (bits 3-0) specify which of the 16 Vertex Accumulation Buffer registers is to receive each data word. The Source Type field (bits 5-4) specifies whether the incoming data is 32-bit IEEE floating-point,

32-bit signed integer fraction, 32-bit unsigned integer fraction or 64-bit double-precision IEEE floating-point. The last word of a vertex has the launch bit set (bit 6), all other words must keep this bit clear (or they suddenly become the last word of the vertex). The launch bit eliminates the need for a count register, as was needed in prior architectures.

Data is directed to the Collection Buffer instead of the Vertex Accumulation Buffer if bit 7 is set. No conversions are performed on the data in this case, so the launch bit is the only other bit that affects the data.

There is no provision for skipping vertex data as in prior art designs, but that can be easily accomplished by writing to a location twice or by writing to a location that is not used in the primitive as sent to the AFB-Float chips.

The Vertex Accumulation Buffer is responsible for storing all converted data from the Format Converter. The VAB is organized as a double buffered set of registers: buffer A and buffer B as shown in FIGS. 5 and 7. The contents of buffer A are loaded by the Format Converter via a 16 bit load enable. The Format Converter indicates to the Vertex Buffer that it is done loading the VAB by asserting one of several "launch" signals. Also provided by the VAB is a 32 bit data path for reading the contents of the A buffer during register reads and context switches.

Each piece of data converted by the Format Converter gets placed into the Vertex Accumulation buffer. This is accomplished by the proper assertion of the 16 bit `fc_vab_iden` lines. FIG. 8 shows the only valid assertions of the load enable (`fc_vab_iden`) lines. Each line corresponds to a different register within the VAB. With the exception of two special cases the load enable lines are only asserted one at a time. A special case exists for normals. If the corresponding bits for all three normals are asserted then the two muxes seen above N_1 and N_2 in FIG. 7 will switch to the 48 bit path. This allows for loading of a single 48 bit normal from the Decompression Unit. When only one of the load enable bits corresponding to the normal registers is enabled then the upper 16 bits of the 48 bit path is used. Note also that the R, G, B and A registers use bits 45:34 of the 48 bit path. The other special case is that the Header register may be loaded in combination with any other register. This was done to accommodate certain quirks in the architecture (namely performance in the FFB compatibility mode).

The mux logic following the VAB is used to pack the data from the VAB as it is transferred into the Vertex Buffer. Header information is not stored in the Vertex Buffers. It is stored directly in the Vertex Pointer Logic. The next section explains the format of the data as stored in the Vertex Buffer Memory.

Context is read from the A buffer of the VAB via the `vbrd_vab_radr` and `vab_vbrd_d032` lines. The `vbrd_vab_radr` is a 4 bit address derived from the `gdi_wads` (GDI word address) which is used to select which of the VAB registers is to be read out onto the `vab_vbrd_d032` bus.

Vertex Buffer

1. Vertex Buffer Organization

The Vertex Buffer resides between the Vertex Accumulation buffer and CF bus output fifos. Data is loaded into the Vertex Buffer from the Vertex Accumulation buffer when a "launch" signal is received from the Format Converter. When enough vertices to assemble a complete primitive have been loaded into the Vertex Buffer Memory the primitive is loaded into the CF bus output fifos for delivery to the AFB-Float chips over the CF Bus. FIG. 9 diagrams the Vertex Buffer.

The Vertex Buffers gather vertices to form complete geometric primitives: dots, lines, triangles, or quadrilaterals.

There are seven vertex buffers; enough to run at maximum speed while gathering independent triangles; that is, three for the triangle currently being written out, three for the triangle being loaded in, and one extra for the overhead of moving the buffers around. Each word in the vertex buffer is 48 bits, to match the width of the data sent across CF-Bus to the AFB-Float chips. Data is transferred into each vertex buffer 48 bits at a time, even if this means reading from up to three separate values in the Vertex Accumulation Buffer. A diagram of one of the vertex buffers is shown in FIG. 10.

All vertices have an X Y Z coordinate and a color. There are three optional parts: the normal, the texture coordinate, and the facet normal. The facet normal actually applies to a full primitive, but the hardware is simpler with the facet normal attached to the vertex it came in with.

The seven vertex buffers are kept track of using three-bit pointers. These pointers are kept on one of six lists:

The Free list. These point to vertex buffers that are ready to receive data.

The New Guy vertex. A vertex transferred in from the Vertex Accumulation buffer gets put here first, along with the two-bit replacement code, until the previous primitive has been grouped is beginning to be transferred to the CF-Bus Output FIFOs. This vertex is then moved to one of the three working vertices.

The Newest vertex. This is the most recent vertex to be added to the working vertices.

The Middle vertex. This is the next to oldest working vertex.

The Oldest vertex. The vertex that has been a working vertex the longest.

The I Wanna Be Free list. When a vertex is taken from the New Guy vertex, either one vertex (the Oldest or Middle) will be recycled or all three in the case of a restart. These are placed on the "I Wanna Be Free" list until the primitive gets completely transferred, at which point they are moved to the free list.

Once a complete primitive is held in the Newest, Middle, and Oldest registers, these three pointers are transferred to the Vertex Output Pointers so that the primitive may be sent out while the next one is being put together. This is shown in FIG. 11. It is noted that all registers shown in the diagram are three bits wide; this is not a large piece of logic like most other block diagrams.

State Machines

The Vertex Buffer control logic is made up of a number of small state machines. The following list is an attempt to describe all of them.

The working registers, Newest/Middle/Oldest, has a state machine with the following states:

None—Only happens when logic is initialized.

Have 1 vertex—After "none" or a restart.

Have 2 vertices—After "have 1."

Have 3 vertices—After "have 2" or after "have 3" and a replace condition.

Have 3 vertices—transmitted—After the transfer to V1/V2/V3.

The V1/V2/V3 output registers get loaded all at once and are only temporaries to show the state of Newest/Middle/Oldest when the triangle was made complete. It has the following states:

Outputting V1—After a triangle launch.

Outputting V2—After "outputting V1."

Outputting V3—After "outputting V2."

Done outputting—After "outputting V3."

The "I wanna be free" list keeps track of which vertices in the V1/V2/V3 registers need to be sent to the free list. These need to be held until the complete triangle is output.

Have none—Default state, when all have been returned.

Have 3—Only occurs on a restart.

Have 2—After "have 3."

Have 1—After "have 2" or any replace.

The free list behaves like a FIFO and has a counter that goes from zero to seven. When hardware gets initialized, it holds all seven. At most other times it holds less.

The "New Guy" vertex has two states:

Have none.

Have 1.

When the Registers Get Clocks

The working registers, Newest/Middle/Oldest, are clocked any time a "New Guy" is present and they are not waiting to output a completed primitive (i.e., not in the "have 3 vertices" state). They are all three clocked at once except on a replace middle condition. The oldest register is not clocked when replacing the middle vertex.

The V1/V2/V3 output registers all get clocked whenever a completed primitive is to be output (i.e., "done outputting" and the working registers are in the "have 3 vertices" state). Note that clockwise triangles are transferred Newest to V3, Middle to V2, and Oldest to V1. When a triangle is counterclockwise, Newest goes to V2 and Middle goes to V3. This is done so that triangles are always clockwise when sent to AFB-Float.

The "I wanna be free" registers get clocked at the same time that the "New Guy" gets transferred into the working registers. They all get clocked on a restart. Only IWBF1 gets clocked for replace middle or replace oldest. Note that the value clocked into IWBF1 is either from the Middle or Oldest register depending on whether the replacement code is replace middle or replace oldest, respectively.

The free list gets values clocked in from the "I wanna be free" list when the completed primitive has been transmitted and the V1/V2/V3 registers are in the "done transmitting" state. They are transmitted one at a time. Since the fastest a vertex could possibly be created is three clocks, it is okay to take three clocks in the worst case to put register pointers back on the free list.

A value goes from the free list to the New Guy whenever there is at least one value on the free list and the New Guy is empty.

Please keep in mind that these registers are only used to index into the array of seven vertex buffers or for housekeeping purposes. The only ones that are actually used as indices are the "new guy" for writing data from the Vertex Accumulation buffer into a Vertex Buffer, and the V1/V2/V3 registers used when writing completed primitives to the CF-Bus Output FIFOs. All other registers are just there for housekeeping purposes.

Treatment of Lines and Dots

Lines behave similarly to triangles, but only the Middle (actually used as "Oldest" for lines) and Newest working registers get used and only two of the V1/V2/V3 registers are needed. The only replacement conditions are replace oldest or restart.

Dots just use one register, the Newest working register, and only one of the V1/V2/V3 registers. The only replacement condition is restart.

Quads

Dealing with quadrilaterals adds a little complexity to the design. Quads can be treated as triangles except when there is a facet normal or facet color. Then it is necessary to have four vertices present before anything can be output. This calls for a new Quads register added to the working registers

after Oldest and a V4 register for output. Unlike triangles, quad strips require two new vertices to create a new primitive.

Quads are still output as triangles to the AFB-Float chips.

5 First, V1, V2, and V3 are sent, then V3, V2 and V4.

Substitution and Replication

There are two cases where either the vertex color or the vertex normal is not the value actually output to the CF-Bus Output FIFO for a particular primitive. Substitution is where a different color or normal is output for all vertices. Replication is where the value in the last (or first) vertex is also used for the other vertices.

Substitution is done using an eighth vertex buffer called the substitution buffer. This is used for overriding the color during operations such as pick highlighting and to specify one facet normal for large polygons.

Replication is similar to substitution, except that the value comes from the Newest (or Oldest) vertex instead of the substitution register. This is needed when color interpolation is disabled, that is, when the color of the most recent vertex specifies the color of the entire triangle or line rather than having the color smoothly interpolated across the primitive. It is also used for faceted shading where one normal is used for all three vertices of a triangle.

25 The hardware performs substitution and replication by selecting the color fields from one vertex while selecting the XYZ values from another vertex while outputting a primitive to AFB-Float. If you look closely at FIG. 4-6 on page 4-27, you'll notice that 16 bits of color share a 48-bit field comes from the one vertex each time. For normals, the whole 48-bit field comes from the one vertex each time. The implementation involves simple multiplexing of the address lines.

Collection Buffer

Attributes and other non-geometric data do not go through the Vertex Accumulation buffer or the Vertex Buffers, but are gathered into the Collection Buffer. Once a full primitive has been gathered, it is sent to the CF-Bus Output FIFOs. All collection buffer data is packed, one and one-half 32-bit words per 48-bit word, as it is written to the CF-Bus Output FIFOs.

There are two types of passthrough data: AFB-Float attributes which are broadcast to all six AFB-Float chips, and data or attributes sent to AFB-Draw which go through a single AFB-Float chip, just like geometric data. For broadcast data, no output is expected from any of the AFB-Float chips. Also, for broadcast primitives, all six Output FIFOs must have enough room in them before the data may be copied.

50 The Collection Buffer does not behave quite like a FIFO. The first data written to it is always started at location zero. The input pointer points at the next location to receive any data and also contains the count of how many words are in the buffer. When a launch condition occurs, the input pointer is copied to a count register and the input and output pointers are cleared to zero. Now, the data is copied out from the locations pointed to by the output pointer, with the point being incremented until it matches the count register. The last word sent is marked with the last word bit set.

60 Since copying data from the Collection Buffer to the CF-Bus Output FIFO is guaranteed to be uninterruptable and since new data coming in cannot be copied in faster than the data is read out, the next input operation can be overlapped with the data output. It is still unclear whether we will have to wait one cycle between the write that causes the launch and the write of the first word of the next data packet, or if the next write can happen on the same cycle as the read.