SIMP (Single Instruction stream/Multiple instruction Pipelining): A Novel High-Speed Single-Processor Architecture

Kazuaki Murakami, Naohiko Irie, Morihiro Kuga, and Shinji Tomita

Department of Information Systems
Interdisciplinary Graduate School of Engineering Sciences
Kyushu University
Fukuoka, 816 JAPAN

Abstract

SIMP is a novel multiple instruction-pipeline parallel architecture. It is targeted for enhancing the performance of SISD processors drastically by exploiting both temporal and spatial parallelisms, and for keeping program compatibility as well. Degree of performance enhancement achieved by SIMP depends on; i) how to supply multiple instructions continuously, and ii) how to resolve data and control dependencies effectively. We have devised the outstanding techniques for instruction fetch and dependency resolution. The instruction fetch mechanism employs unique schemes of; i) prefetching multiple instructions with the help of branch prediction, ii) squashing instructions selectively, and iii) providing *multiple conditional modes* as a result. The dependency resolution mechanism permits *out-of-order execution* of sequential instruction stream. Our out-of-order execution model is based on *Tomasulo's algorithm* which has been used in single instruction-pipeline processors. However, it is greatly extended and accommodated to multiple instruction pipelining with; i) detecting and identifying multiple dependencies simultaneously, ii) alleviating the effects of control dependencies with both *eager execution* and *advance execution*, and iii) ensuring a precise machine state against branches and interrupts. By taking advantage of these techniques, SIMP is one of the most promising architectures toward the coming generation of high-speed single processors.

1. Introduction

The demand for high-speed single-processors forces more sophisticated instruction pipelines to be implemented in SISD (Single Instruction stream/Single Data stream) processors of a wide range from microprocessors to supercomputers. These conventional pipelined SISD processors exploit temporal parallelism in the process of instruction execution; i.e., the process is segmented into consecutive subprocesses (stages of a pipeline). The performance of these processors can be expressed as the program execution time;

$$E = N \times C \times T,$$

where N is the number of instructions that must be executed, C is the average number of cycles per instruction, and T is the cycle time. N and C depend on processor architectures; e.g., CISC architectures decrease N but increase C by improving functionality of instructions, while RISC architectures reduce C to nearly 1 but increase N by simplifying the instruction set. Although processor architectures may influence T in some degree, semiconductor technology mostly determines T. Since T has been decreasing constantly with advances in VLSI technology, conventional pipelined SISD processors have enjoyed speedups regardless of their architectures: CISC or RISC.

However, the physical lower limit of T obviously exists in any semiconductor technology, and therefore some architectural changes must be considered for SISD processors.

Some innovative approaches of such challenges are VLIW (Very Long Instruction Word) architectures [Fisher83], which are derivatives of SISD and exploit spatial parallelism (low-level parallelism). VLIW architecture decreases N by specifying two or more independent operations in an instruction, without increasing C by having each operation be a RISC-style instruction. We have already developed two VLIW processors: the QA-series (QA-1 and QA-2) [Hagiwara80; Tomita83,86]. To exploit spatial parallelism, the QA-series provide multiple functional units such as quadruple ALUs, quadruple memory access units, and a sequencer. QA-1 and QA-2 employ very long instruction formats of 160-bits and 256-bits respectively, for controlling every functional unit independently. However, VLIW architectures have a serious drawback; i.e., it is difficult to keep program compatibility, because their hardware architectures are exposed to compilers.

Other, somewhat old, approaches are found in single instruction-pipeline processors with multiple functional-units (called MFU processors), such as the CDC 6600 and the IBM 360/91. MFU processors have potential of reducing C to nearly 1 by having multiple pipelined-functional units busy, and keep N comparable to CISC processors. Unlike VLIW, program compatibility is preserved very easily. Nevertheless, the limit exists to performance enhancement because the instruction issue rate ($= 1/C$) can not exceed one instruction per cycle.

All these approaches attempt to exploit low-level fine-grained parallelism by utilizing multiple functional units. Key problems in the exploitation of low-level parallelism are; i) to detect data dependencies and control dependencies, ii) to resolve these hazards, and iii) to schedule the order of instruction execution. VLIW architectures rely solely on clever compilers which solve the problems by means of static code scheduling, such as trace scheduling [Fisher81] and software pipelining [Lam88]. MFU processors also solve the problems at run time by implementing dynamic code scheduling in hardware. Static and dynamic code scheduling methods differ in their domain; i.e., static code scheduling is done with a broad overview of program codes, but dynamic code scheduling is done with a peephole. However, these code scheduling methods are not mutually exclusive.

After evaluating the QA-series, we have studied the feasibility for enhancing the performance of SISD processors drastically by combining both temporal and spatial parallelisms, and for preserving program compatibility as well. As a result of this study, we have introduced the multiple instruction-pipeline parallel architecture: SIMP [Murakami88]. Given that P instruction pipelines are provided, SIMP processors ideally reduce C to $1/P$ by fetching P instructions per cycle, and keep N comparable to conventional single-pipelined SISD processors as well. As the first implementation of SIMP, we are now developing the SIMP processor prototype: 「 新 風 」 (in Japanese), whose English pronunciation is [ʃimpu :]. It implies "new streamline" processor.

The paper is organized in 6 sections. The following two sections present the rationale of SIMP architecture (section 2), and give some background regarding instruction-pipelining techniques (section 3). In section 4, we introduce the SIMP processor prototype and discuss its pipeline flow and instruction-set architecture. In section 5, we describe the instruction fetch mechanism and the out-of-order execution model, both of which are devised for the SIMP prototype. Section 6 offers a few concluding remarks.

2. Principles of SIMP Pipelined Instruction Execution

There are various models of pipelined instruction execution. They apply temporal parallelism (i.e., pipelining) and/or spatial parallelism (i.e., low-level parallelism) to a single instruction stream, as schematically depicted in Figure 1. We clarify SIMP architecture by comparing its instruction pipelining model with those of its counterparts such as linear-pipeline, MFU, and pipelined VLIW. In the discussion here, we assume that the process of instruction execution is decomposed into 5 stages: IF (instruction fetch), D (instruction decode), OF (operand fetch), E (execute), and W (result write).

2.1 Linear-Pipelined Processor

Most of conventional pipelined SISD processors employ single pipeline structure shown in Figure 1-a. Instructions are sequentially processed one by one from preceding stages to succeeding stages. Instructions should enter and leave each stage in-order with respect to the compiled code sequence. This pipeline structure is referred to as a *linear pipeline*.

Linear-pipelined processors exploit only temporal parallelism. *Pipeline interlock logic* is usually placed between critical stages to detect and resolve data and control dependencies; otherwise, the interlock logic is imposed on a compiler.

The maximum throughput of instruction execution is at most one instruction per cycle; i.e., C (the average number of cycles per instruction) cannot be less than 1. Thus, the ideal program execution time of linear-pipelined processors results in;

$E=N\times 1\times T$.

2.2 MFU processor

Some pipelined SISD processors such as the CDC 6600, the IBM 360/91 floating-point unit, and the CRAY-1 scalar unit, provide multiple functional units (MFUs) in E-stage (Figure 1-b). Each functional unit may or may not be pipelined. Although not shown in Figure 1-b, W-stage can be multiplied by the number of MFUs.

MFU processors can exploit both temporal and spatial parallelisms. To increase MFU utilization, most of pipeline interlock logic is localized to the stages prior to E-stage. It is referred to as *instruction issue logic* [Weiss84]. Simple instruction issue logic issues an instruction to an MFU sequentially. Complex issue logic such as *Tomasulo's algorithm* [Tomasulo67] is capable of dynamic code scheduling by allowing instructions to begin and/or complete execution nonsequentially.

Even if any instruction issue logic is employed, however, at most one instruction can be issued per cycle. Thus, the maximum throughput of instruction execution is the same as in linear-pipelined SISD processors; i.e., C cannot be less than 1. The ideal program execution time of MFU processors results in;

$E=N\times 1\times T$.

More complex issue logic capable of issuing multiple instructions per cycle are studied so as to reduce C less than 1 [Tjaden70; Acosta86; Pleszkun88]. If F functional units are provided and the same number of instructions can be issued each cycle, the ideal program execution time now results in;

$E=N\times(1/F)\times T$.

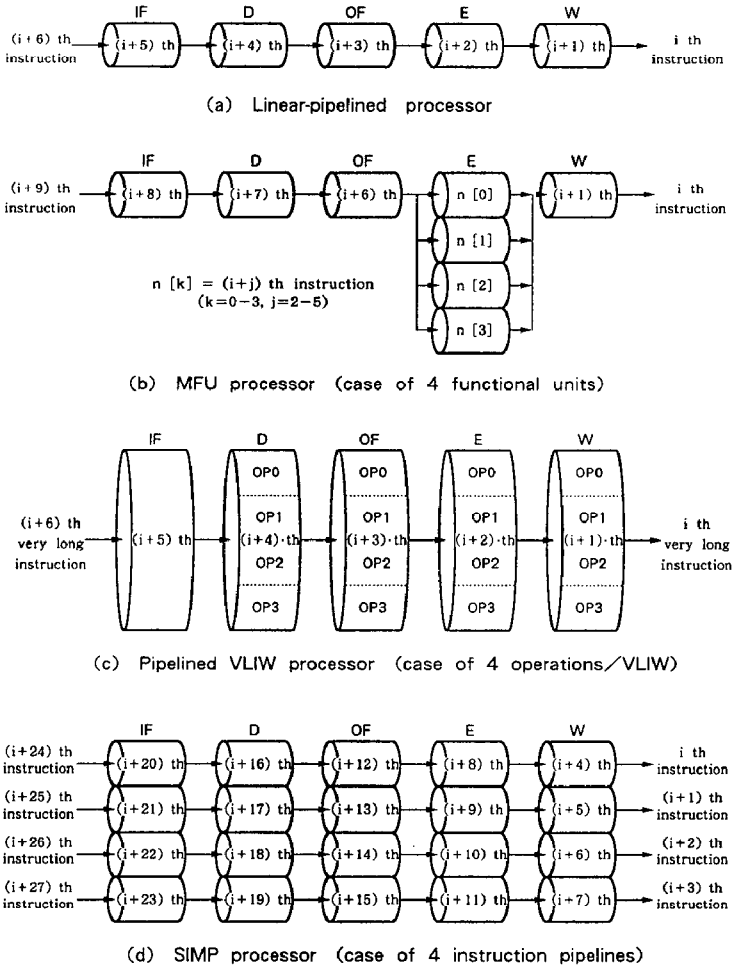

Figure 1. Pipelined Instruction Execution Models

2.3 Pipelined VLIW Processor

VLIW processors also provide multiple functional units. Unlike MFU processors, however, VLIW processors attempt to utilize functional units by specifying multiple independent operations in a single very long instruction. Original, non-pipelined, VLIW processors such as the QA-series have exploited only spatial parallelism. Since each operation can be pipelined, pipelined VLIW processors such as the Multiflow TRACE [Colwell87] and the Cydrome Cydra 5 [Rau89] can now exploit both temporal and spatial parallelisms (Figure 1-c).

A very long instruction is fetched each cycle, each operation field of the instruction is decoded in parallel, and each decoded operation is processed simultaneously and independently at the corresponding functional unit. A clever compiler is well aware of the architecture (e.g., the number of functional units, the number of pipeline stages, etc.), and is responsible for scheduling the order of operations strictly to prevent any hazard from causing incorrect results at run time. Therefore, no pipeline interlock logic or complex instruction issue logic need be implemented in hardware.

Given that F functional units are provided (e.g., $F=4$ in Figure 1-c), pipelined VLIW processors can reduce N' (the number of very long instructions to be executed) to N/F, where N is the number of short instructions to be executed substantially, by compacting F operations into one very long instruction. The ideal program execution time of pipelined VLIW processors results in;

$E=(N/F)\times 1\times T$.

2.4 SIMP Processor

SIMP processors employ multiple instruction-pipeline structure shown in Figure 1-d, and exploit both temporal and spatial parallelisms. All pipelines should be identical. Each pipeline may be either the linear pipeline shown in Figure 1-a, or the MFU-processor-like pipeline shown in Figure 1-b.

Regardless of the number of instruction pipelines provided, a single program counter (PC) controls the flow of instruction execution; i.e., a single instruction stream. Given that P instruction pipelines are provided (e.g., $P=4$ in Figure 1-d), instructions can be processed in blocks of P. We refer to this block as an "*instruction block*", which consists of successive P instructions in an object program. An instruction block starting with the instruction specified by c(PC) and ending with the instruction specified by $c(PC)+P-1$ is fetched each cycle, where c(PC) indicates the contents of PC. Each instruction of an instruction block is decoded and processed simultaneously, but *dependently*, at the corresponding instruction pipeline.

SIMP processors would appreciate the advantages of static code scheduling, but they should exploit spatial parallelism by scheduling instructions at run time. It is because, unlike VLIWs, SIMP processors should not expose their hardware architectures (e.g., the number of instruction pipelines) to such clever compilers as used for VLIW processors. In such a case, even if an ordinary compiler could increase the dependency distance, some dependencies may still remain in the compiled code until run time. As linear-pipelined and MFU processors do, SIMP processors must provide a way to resolve these remaining dependencies at run time.

From a standpoint of program compatibility, the above is a major difference between SIMP and VLIW. For VLIW, it is not impossible but difficult to have compiled codes portable among VLIW processors with the various number of functional units, because the instruction size reflects the number of functional units directly. On the other hand, in SIMP, one instruction corresponds to one instruction pipeline, and therefore the number of instruction pipelines installed is transparent to its ISP (Instruction-Set Processor) architecture.

The maximum throughput of instruction execution is at most one instruction block of P instructions per cycle; i.e., C can be reduced to $1/P$. Thus, the ideal program execution time of SIMP processors results in;

$E=N\times(1/P)\times T$.

3. Critical Issues Regarding SIMP

There are several critical issues to be resolved before implementing SIMP architecture, which issues are also common to most pipelined processors. We identify those issues by giving some background regarding instruction-pipelining techniques.

3.1 Branch Problem

The detrimental effects of branch instructions are severe, because fetching the next instruction is postponed until a branch decision (i.e., taken or not-taken) is resolved and a branch target address is generated. As a result, a lot of "bubbles" are forced into a pipeline, and the pipeline stalls. There are some techniques to reduce the branch penalty: delayed branch, branch prediction, multiple prefetching, and so on [Lee84]. A popular technique is *branch prediction*. By means of branch prediction, instructions fetched from a predicted branch path can be executed in a *conditional mode*. If the predicted path is incorrect, however, some prediction-miss handling should be done to nullify or squash the conditionally executed instructions.

SIMP processors also can adopt branch prediction with prediction-miss handling. However, branch prediction involves other critical issues unique to SIMP;

"How to determine predicted paths inside an instruction block which substantially consists of multiple instructions to be fetched simultaneously, and how to recover if one of the paths is incorrect."

3.2 Data Dependency

There are 3 types of data dependencies: flow (RAW: Read-After-Write), anti (WAR: Write-After-Read), and output (WAW: Write-After-Write). Two popular hardware solutions exist to the data dependency problem: *pipeline interlock logic* in linear-pipelined processors, and *instruction issue logic* in MFU processors.

Pipeline interlock logic is so simple and straightforward that, if it detects data dependencies, it just interlocks pipeline until the dependencies are resolved. The detrimental effects of data dependencies, however, can not be alleviated.

Simple instruction issue logic can issue an instruction to an MFU, only if its data dependencies are resolved; otherwise, the instruction and subsequent instructions are blocked from issuing. Complex issue logic such as Tomasulo's algorithm permits an instruction to be issued even when its dependencies are not resolved. Flow dependencies can be resolved by waiting and monitoring instruction results, and anti/output dependencies can be eliminated by renaming registers with tags. In addition, Tomasulo's algorithm allows subsequent instructions to bypass the instruction issued previously, while it waits until its dependencies have been resolved. Thus instructions can begin and/or complete their execution out-of-order with respect to the compiled code sequence. This scheme of instruction execution is referred to as *out-of-order execution* model [Weiss84; Patt85], and can minimize the effects of flow dependencies.

To utilize multiple instruction pipelines as well as multiple functional units, SIMP processors also can employ the out-of-order execution model. In such a case, there are critical issues unique to SIMP;

"How to detect data dependencies among multiple instructions simultaneously, and how to represent them."

3.3 Control Dependency

Even if above-mentioned techniques such as conditional-mode execution and out-of-order execution are employed, control dependencies still impede the execution of instructions. It is because the domain of out-of-order execution may be affected by the size of a *basic block* (in which only one control dependency occurs at the bottom). There are two schemes of the out-of-order execution model: *lazy* (or normal) *execution* and *eager execution*.

Lazy execution scheme disallows out-of-order execution to proceed beyond any branch instruction; i.e., the domain of out-of-order execution is limited to the basic block whose execution is certainly needed. Although the scheme is simple, the effect of out-of-order execution is diminished by small basic blocks.

On the other hand, eager execution scheme causes instructions to be executed out-of-order regardless of basic blocks they belong to. The scheme applies out-of-order execution to instructions fetched from a predicted path, and it therefore requires more sophisticated prediction-miss handling mechanisms, such as the *reorder buffer* of the RUU (Register Update Unit) [Sohi87] and the checkpoint repair mechanism [Hwu87], in order to avoid an imprecise machine state (described in section 3.4).

3.4 Imprecise Machine State

When instructions may finish out-of-order, an imprecise machine state can be introduced by branches and instruction-generated traps (e.g., an exception and a page fault). It is because, when a branch or trap occurs, out-of-order execution can modify a machine state (such as register file and memory) inconsistently with the sequential architectural specification. The problem caused by traps is well-known as *imprecise interrupt*, and there are many solutions to the problem [Smith85]. Also, there are

common solutions to both the imprecise interrupt problem and the problem caused by branches [Hwu87; Sohi87].

4. SIMP Processor Prototype

The SIMP processor prototype, the first implementation of SIMP, is a quadruple instruction-pipeline processor composed of 4 identical instruction pipelines. It is an ideal SIMP processor, in the sense that it is equipped with rich hardware mechanisms to resolve the issues discussed in section 3; i.e., branch prediction, conditional-mode execution, out-of-order execution, and reorder buffer. Note that its design never limits other implementations of SIMP architecture, especially on the number of instruction pipelines.

4.1 Processor Organization

Block diagram of the SIMP processor prototype is shown in Figure 2. Each instruction pipeline comprises 5 stages: IF (Instruction-block Fetch), D (Decode), I (register-read and Issue), E (Execute), and R (register-write and Retire). The prototype will be implemented in off-the-shelf TTL/CMOS chips, with a machine cycle time of 60 ns, or a pipeline cycle time of 120 ns (i.e., 2-cycle pipeline). Hereafter cycles refer to pipeline cycles. The SIMP processor prototype consists of the following main components.

(a) MBIC (Multiple-Bank Instruction Cache):

The MBIC is an instruction cache consisting of 4 independent banks of RAMs, where a bank width equals to an instruction size (4 bytes). A cache line contains 16 contiguous instructions from a *static instruction stream* (i.e.;object program) in memory. The MBIC also includes a BTB (Branch Target Buffer) for purpose of branch prediction.

(b) IBSU (Instruction-Block Supply Unit):

The IBSU is common to all instruction pipelines, and it plays a role of IF-stage. At each cycle, the IBSU fetches 4 successive instructions from the MBIC with the help of branch prediction and distributes them to all the IPUs.

(c) IPUs (Instruction Pipeline Units):

Four identical IPUs are installed. For an instruction received from the IBSU, each IPU performs a pipeline sequence from D-stage to R-stage individually. E-stage is equipped with 5 pipelined functional units: IALU (Integer ALU), IMUL (Integer MULtiplier), FALU (Floating-point ALU), FMUL (Floating-point MULtiplier), and DCAR (Data Cache Access Requester). They are constructed with 32-bit building blocks such as Advanced Micro Devices Am29323/32 and Weitek WTL2264/65.

To allow out-of-order execution and to ensure the precise machine state, E-stage provides a queue of 4 entries, called *WRB (Waiting and Reorder Buffer)*, which comprises a pair of WB (Waiting Buffer) and RB (Reorder Buffer). The WB corresponds to the *reservation stations* in the IBM 360/91 [Tomasulo67], and the RB corresponds to the *reorder buffer* proposed in [Smith85; Sohi87].

(d) DHRF (Dependency-Handling Register-File):

The DHRF is a register file with 8 read-ports and 4 write-ports. The DHRF is shared by all the IPUs, and is accessed three times (i.e., two source-register accesses at I-stage and one destination-register access at R-stage) per cycle by every IPU. The DHRF maintains a *WRT (Write Reservation Table)*, each entry of which is associated with a register, to detect flow dependencies. It also maintains a *CDT (Control Dependency Table)* to detect control dependencies. The DHRF provides 4 *BBs (Bypass Buffers)*, each of which contains a copy of the RB in each IPU.

(e) MPDC (Multiple-Port Data Cache):

The MPDC is a data cache with 4 load-ports and 4 store-ports. The MPDC is shared by all the IPUs, and is accessed twice (i.e., one LOAD access at E-stage and one STORE access at R-stage) per cycle by every IPU, if executing a LOAD/STORE instruction.

(f) IPCN (IPU Chaining Network):

The IPCN is an interconnection network among all the WRBs.

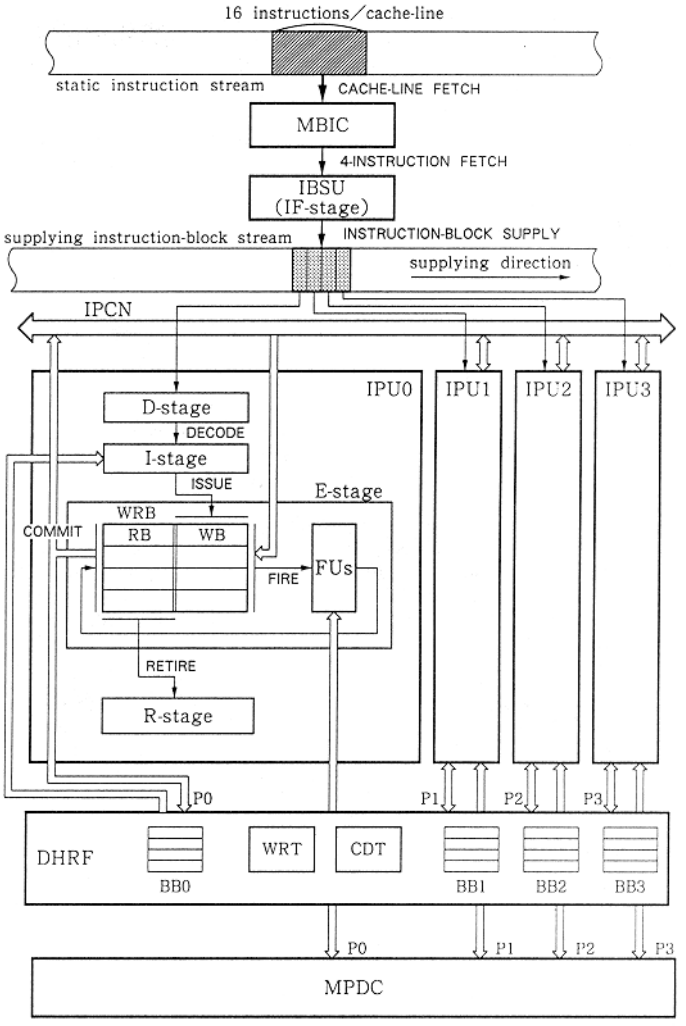

Figure2. Block Diagram of the SIMP Processor Prototype

It consists of 4 broadcast buses, where every IPU is the bus master of the corresponding bus. When one IPU places an execution result on its own bus, all the IPUs (including the sender itself) can catch the result from the bus. Thus the IPCN works like the common data bus used in the IBM 360/91 [Tomasulo67].

4.2 Pipeline Flow

During IF-stage, the IBSU fetches an instruction block of 4 successive instructions from a static instruction stream cached in the MBIC, and then supplies it for all the IPUs, one instruction per IPU. The instruction stream being supplied by the IBSU is called *"supplying instruction-block stream"*. Hereafter, the instruction block is a unit of management until it retires from the supplying instruction-block stream. At most 7 instruction blocks can reside in the stream; i.e., 1/D-stage, 1/I-stage, 4/E-stage, and 1/R-stage.

Once each IPU takes the corresponding instruction in an instruction block, it can accomplishes pipelined processing for the instruction individually. All the 4 instructions in the same instruction block, however, should flow through D, I, and R-stages on a lockstep basis, and in-order with respect to the instruction-block sequence. Only one slack exists in E-stage, where instructions can wait requisite operands and begin the execution

when the operands become available. Instructions can begin and complete the execution out-of-order with respect to both the instruction sequence in the instruction block and the instruction-block sequence. The out-of-order execution model is also referred to as "*local dataflow execution*", because instructions are executed in a dataflow-execution fashion and the scope is localized in E-stage.

A general pipeline flow following IF-stage is summarized below. A brief description of out-of-order execution, which involves I, E, and R-stages, follows in section 5.

(a) D (Decode) stage:

In each IPU, the decoder accepts an instruction from the IBSU and decodes an operation field of the instruction. It then forwards identifiers of requisite registers (up to 3: 2 sources and 1 destination) and an operation type (i.e., BRANCH, or not) of the decoded instruction to the DHRF.

(b) I (register-read and Issue) stage:

The DHRF receives the register identifiers and the operation type from every IPU. The DHRF updates the entries in the WRT, each of which is indexed by a destination-register identifier. It also updates the CDT, according to the operation types.

The DHRF then transmits the current contents of source registers, which are fetched from the register file or bypassed from BBs, to the IPU requesting the source operands. It appends some control information representing flow and control dependencies.

Each IPU accepts them and "*issues*" the instruction by forwarding them to the tail entry of the WRB, if the WRB is not full.

(c) E (Execute) stage:

At each machine cycle, every instruction in the WRB is checked to see if it can be executed. If an instruction has no "probable" flow dependency (which is defined in section 5.3), then the IPU can "*fire*" the instruction by dispatching it to one of functional units. Otherwise, the instruction must wait in the WRB until its probable flow dependencies have been resolved. An instruction can resolve its dependencies by monitoring the IPCN. When the firing instruction completes its execution, the result is stored in the WRB.

At the same time, if no control dependency exists to the instruction which has completed its execution, the IPU can "*commit*" the instruction by broadcasting the execution result to all the IPUs via the IPCN. Otherwise, the instruction must wait in the WRB until its control dependencies have been resolved.

(d) R (register-write and Retire) stage:

Each IPU forwards a committed result, if any, from the WRB to the associated BB in the DHRF. The DHRF updates the WRT and CDT, according to the committed results. For the instruction block at the head of the WRBs, if all 4 instructions are committed, then the instruction block can be "*retired*" from the IPUs. The retirement has the DHRF update the destination registers with the contents of the head entries of the BBs.

4.3 Balanced Instruction Set Computer

The SIMP architecture itself is not an ISP architecture but a vehicle useful for implementing several different target ISP architectures. We believe, however, that there is a class of ISPs most suitable for SIMP in a wide spectrum of ISPs from RISC to CISC. We refer to it as "*BISC (Balanced Instruction Set Computer)*", and apply it to the SIMP processor prototype. In order to utilize strengths of SIMP, we place the following constraints on the BISC architecture;

(1) Single-sized instructions: Single-sized instructions allow fetching and decoding multiple instructions simultaneously.
(2) LOAD/STORE architecture: Register-register operations relieve some of the burden to detect and resolve data dependencies.
(3) Fixed-cycle operations: Unlike RISCs emphasizing single-cycle operations, operations need not be completed in one cycle, but in some fixed cycles. If E-stage is pipelined by means of multiple arithmetic pipelines, it is no longer necessary to limit all instructions to single-cycle operations. Also, from a viewpoint of code scheduling, it is sufficient to fix the number of operation cycles for every instruction. Thus BISC can include some complex instructions requiring fixed multiple cycles, such as integer MULTIPLY/DIVIDE and floating-point instructions, which are usually excluded from RISCs. However, the other complex instructions, whose operations spend the variable number of cycles, are still excluded from BISC.
(4) Balanced execution time: Though the fixed-cycle operations can loosen RISCs' constraint of single-cycle operations, the unbalance of execution time of every instruction introduces problems of large pipeline latency and low pipeline throughput. The pipeline stall due to the unbalance is severe in SIMP, because all instructions in an instruction block had better flow through pipelines on a lockstep basis so as to avoid the imprecise machine state problem. It means that an instruction with the longest execution time in an instruction block determines the pipeline elapsed time for the instruction block. Therefore the execution time, especially the number of operation cycles in E-stage, of every instruction should be balanced within limits.

The ISP architecture for the SIMP processor prototype is a representative of BISC; i.e., 32-bit single-sized instructions, LOAD/STORE architecture, register-register operations specified by 3-operand formats, and balanced fixed-cycle operations ranging from 2 to 6 machine-cycles in E-stage. From a standpoint of the balanced execution time, DIVIDE instructions (which would require over 13 machine-cycles) are not provided, but MULTIPLY instructions, for both integer and floating-point (which require 2 and 6 machine-cycles respectively), are provided.

5 Resolution Algorithms

To resolve the critical issues discussed in section 3, we have already developed algorithms for branch problem resolution, data dependency resolution [Kuga89], control dependency resolution, and so on. Details of all the algorithms cannot be presented here due to insufficient space, however, and they will be reported elsewhere. We describe the outline of these algorithms below.

5.1 Branch Problem Resolution

During IF (Instruction-block Fetch) stage, the IBSU (Instruction-Block Supply Unit) supplies an instruction block of 4 successive instructions. Because of the presence of branch instructions, the supplying instruction-block stream is somewhat different from a *dynamic instruction stream*, which is a sequential execution order of instructions in a static instruction stream. The supplying instruction-block stream has the following characteristics;

(1) To fetch and distribute 4 instructions simultaneously, the IBSU simply makes an instruction block of 4 successive instructions in the static instruction stream. Thus the instruction block may include some instructions not to be executed (i.e., to be excluded from the dynamic instruction stream).
(2) The instruction block to be prefetched is determined with the help of branch prediction using a BTB (Branch Target Buffer). Thus the supplying instruction-block stream may contain some instruction blocks fetched from incorrectly predicted branch paths.

Conventional prediction-miss handling techniques nullify all the instructions executed in a conditional mode (i.e., flush the pipeline), if (and only if) a branch prediction is wrong. Such a *pipeline flushing scheme* is, however, inadequate to the supplying instruction-block stream because of the following reasons;

(1) Even if a prediction is accurate, some instructions must be nullified due to the above-mentioned characteristics 1.
(2) Although a prediction is wrong, it is not efficient to flush the pipeline regardless of whether or not the instruction to be refetched already exists in the pipeline. The inefficiency is

more severe in the SIMP processor prototype, because as many as 23 instructions can be executed in a conditional mode.

Hence, instead of the pipeline flushing scheme, we have introduced the selective instruction-squashing scheme, which squashes only the instructions to be excluded from the dynamic instruction stream whenever a branch decision is resolved [Murakami88]. The selective instruction-squashing scheme results in allowing the instructions fetched from multiple branch paths to be execute in *multiple conditional modes*.

The selective instruction-squashing scheme forces the IBSU to select instructions to be squashed. For that purpose, the IBSU maintains a queue, called *IBAT (Instruction-Block Address Table)*, each of whose entries is associated with an instruction block under pipelined execution. There are the following 4 patterns of squashing instructions selectively, as shown in Figure 3;

(1) Predict-Not-taken/Not-taken (Figure 3-a): When a prediction is "Not-taken" and found to be accurate, no squashing occurs.

(2) Predict-Taken/Taken (Figure 3-b) and Predict-Not-taken/Taken (Figure 3-c): When a branch is taken, the IBAT is associatively searched to see if the branch target instruction has already been supplied. If the target is present, all the instructions between the branch and the target must be squashed. In such a case, it is not necessary to refetch the target; otherwise, the pipeline must be flushed and the target refetched.

(3) Predict-Taken/Not-taken (Figure 3-d): When a prediction is "Taken" and found to be wrong, the IBAT is associatively searched to see if the sequential target instruction, which is incidentally fetched from the sequential stream, has already been supplied. The succeeding process is the same as that stated above.

5.2 Data Dependency Resolution

Our data-dependency resolution algorithm is an extension to Tomasulo's algorithm [Tomasulo67]. We describe the outline of our algorithm by comparing it with original Tomasulo's algorithm and the other extensions.

(a) Flow Dependency Detection and Representation:

(1) Tomasulo's algorithm: Instructions are sequentially issued one by one. Each register is assigned a busy bit which indicates if it is the destination register of an instruction in execution. Each register is also assigned a tag which identifies the instruction that must write the result into the register. When an instruction is issued, the source registers are checked to see if they are busy. An instruction whose source registers are busy obtains tags for the busy source registers.

(2) Our algorithm: We adopt a bitmap scheme, called *"multiple-dependency representation scheme"*, rather than Tomasulo's tag scheme, because the tagging substantially needs to be carried out in sequential. Four instructions are issued simultaneously. Each register is assigned an entry of the WRT (Write Reservation Table), which entry is a bitmap and identifies *all* the instructions that must write the results into the register. When an instruction is issued, it always obtains the bitmaps for its source registers. Thus every instruction is capable of recognizing its multiple flow dependencies caused by up to 3 preceding instructions in the same instruction block and by up to 3 preceding instruction blocks (i.e., up to 15 dependencies per source-register). It is another reason for the bitmap scheme, and detailed in section 5.3.

(b) Flow Dependency Resolution:

(1) Tomasulo's algorithm: An instruction whose source registers are busy is forwarded to a reservation station (RS). The instruction must wait in the RS until its flow dependencies have been resolved. It can resolve its dependencies by monitoring the common data bus (CDB) and by performing the tag-matching process. If a matching tag is found, the data on the CDB is the requisite source operand.

(2) Our algorithm: Every instruction is forwarded to the tail entry of the WRB (Waiting and Reorder Buffer) in the corresponding IPU (Instruction Pipeline Unit). The WB (Waiting Buffer) part of the WRB corresponds to reservation stations. If an instruction has any "probable" flow dependency (which is defined in section 5.3), then it must wait in the WRB until its probable flow dependencies have been resolved. Flow dependencies can be resolved by monitoring the IPCN (IPU Chaining Network) and by clearing bits in bitmaps representing dependencies. No tag-matching logic is needed. If a bit associated with a probable flow dependency is cleared, the data on the IPCN is the requisite source operand.

(c) Anti/Output Dependency Resolution:

(1) Tomasulo's algorithm: If the destination register is busy at issuing an instruction, the instruction renames its destination register by updating the tag of the destination register with a new tag. Every register as well as the RS monitors the CDB, and updates its contents and clears its busy bit if a matching tag is found. The direct register renaming scheme introduces imprecise interrupts, because the instruction that completes its execution is allowed to write the result into the destination

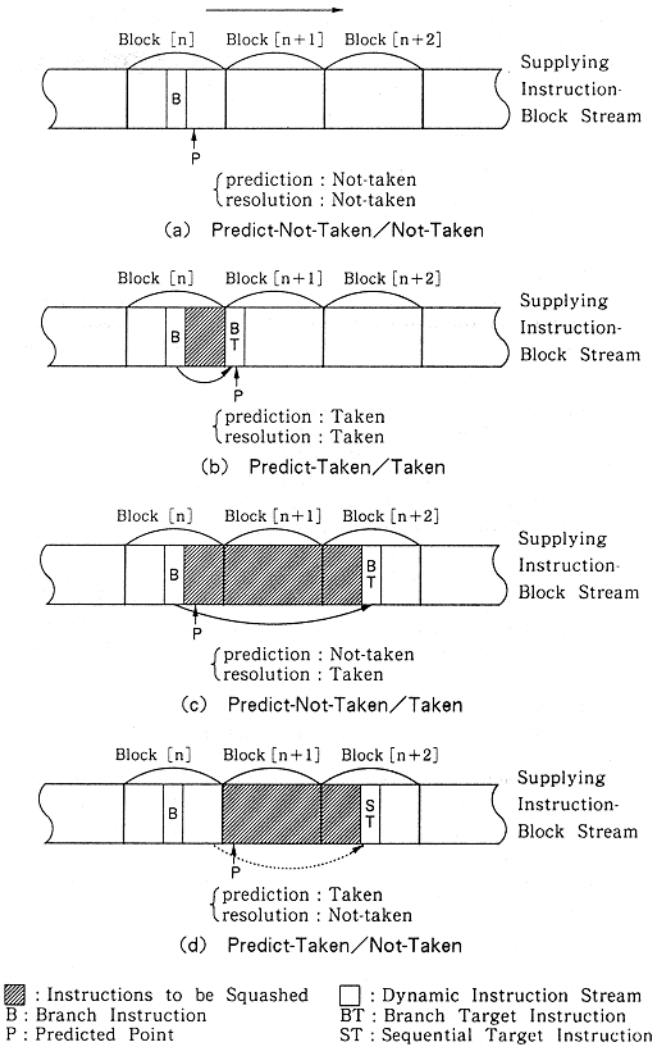

Figure 3. Selective Instruction-Squashing Scheme

register directly if a tag match occurs. In other words, the result disappears if no tag match occurs.

(2) Patt's and Sohi's algorithms: The registers are not renamed directly, but done indirectly through a buffer such as the result buffer [Patt85] or the reorder buffer of the RUU (Register Update Unit) [Sohi87]. That is, an instruction result is not written into the destination register straightforward, but stored in the buffer temporarily. The indirect register renaming scheme allows recovery from wrong branch predictions and traps, and therefore implements precise interrupts.

(3) Our algorithm: Basically the same as Patt's and Sohi's algorithms. The scheme of interest to us is the RUU, which acts as a queue and merges the RSs with the reorder buffer [Sohi87]. The WRB (Waiting and Reorder Buffer) in our algorithm corresponds to the RUU. Unlike the RUU, a separate WRB is located in every IPU and a set of all the WRBs acts as a queue.

5.3 Control Dependency Resolution

As stated briefly in section 3.3, there have been two hardware solutions of the basic block problem caused by control dependencies: lazy execution and eager execution. In addition to the eager execution scheme, we attempt a more eager execution scheme, called *"advance execution scheme"*.

(a) Lazy Execution:

When a branch instruction is present, subsequent instructions are blocked from firing (or dispatching) their executions until the control dependency due to the branch has been resolved [Tomasulo67].

(b) Eager Execution:

In the above case, subsequent instructions are allowed to fire their executions as soon as their flow dependencies are resolved, even before the control dependency has not been resolved. The scheme does not commit the instructions (i.e., update the machine state) until the control dependencies have been resolved [Sohi87].

(c) Advance Execution:

To take advantage of multiple conditional modes derived from the selective instruction-squashing scheme, the advance execution scheme allows an instruction to fire its execution (called *pre-execution*) even before its flow dependencies have not been definitely resolved because of the presence of control dependencies. It also allows the instruction to *re-execute* (or *backtrack*) when the pre-execution result is found invalid.

We first introduce two types into flow dependencies: *PFD (Probable Flow Dependency)* and *UFD (Uncertain Flow Dependency)*. As shown in Figure 4, a PFD occurs when the instruction **A** is going to write to the register *Rn* from which the instruction **D** is going to read, and when no control dependency covers **A**. On the other hand, a UFD occurs when the instruction **C** is going to write to *Rn* and some control dependencies covers **C**. From the viewpoint of the instruction **D**, at most 1 PFD and more than 1 UFD may be present per source operand at a time. To identify all the PFD and UFDs, we employ the multiple-dependency representation scheme mentioned in 5.2.

For this example, Tomasulo's tag scheme represents that **D** is flow-dependent on **C**, with a single tag to identify **C** which will supply **D** with the latest version of *Rn*. However, whether **C** is executed or not depends on the conditional branch instruction **B**. If the branch is taken and the branch target falls between **C** and **D**, then **C** is never executed, and therefore the tag identifying **C** has no meaning. The instruction **D** should access the correct version of *Rn*, which was generated by **A** and may be already written into the register file. In such a case, conventional pipeline flushing schemes recover from the incorrect sequence by squashing all the instructions following **B**, and by refetching instructions starting from the branch target. Then the instruction **D** is refetched, and now can obtain the correct version of *Rn* from the register file.

Since the selective instruction-squashing scheme need not refetch the instruction **D**, however, **D** cannot obtain the correct source operand with the tag. The tag scheme is inadequate to the selective instruction-squashing scheme consequently. The multiple-dependency representation scheme has just resulted from the selective instruction-squashing scheme.

Now an instruction can fire its pre-execution as soon as its PFDs are resolved, regardless of whether or not its control dependencies have been resolved. Resolution of its control dependencies may uncover any new PFD (which was one of UFDs), even after the instruction completed its pre-execution. In such a case, the instruction must backtrack and wait in a WRB until its PFDs have been resolved again. There is no limit to the number of the backtrack. The WRB commits the instruction when its PFDs and control dependencies have been resolved. The committed result is then forwarded, via the IPCN, to all the instructions waiting for it.

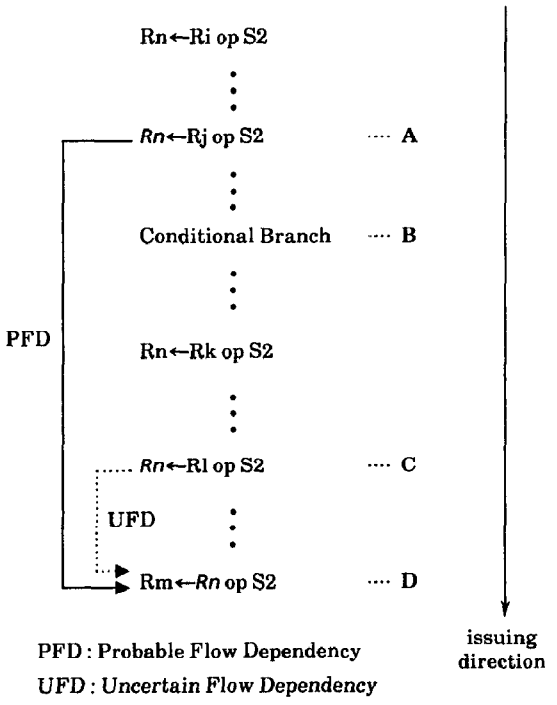

Figure 4. Probable and Uncertain Flow Dependencies

6. Conclusions

In this paper, we have introduced a novel multiple instruction-pipeline parallel architecture: SIMP. Our goal is to enhance the performance of SISD processors drastically by exploiting both temporal and spatial parallelisms, and to keep program compatibility as well. Key techniques to achieving high performance are the mechanisms of fetching multiple instructions simultaneously and continuously, and of resolving instruction dependencies effectively at run time. We have devised these mechanisms for the SIMP processor prototype which is a quadruple instruction-pipeline processor under development.

The dependency resolution mechanism permits out-of-order execution of sequential instruction stream. Our out-of-order execution model is based on Tomasulo's algorithm, but is greatly extended and accommodated to multiple instruction pipelining. The instruction fetch mechanism cooperates with the out-of-order execution model by supplying it with multiple instructions continuously with the help of branch prediction. In this paper, we have introduced several unique schemes such as selective instruction squashing, multiple conditional modes, multiple-dependency representation, advance execution, and so on.

Although SIMP does not limit the target ISP architectures, we have suggested a class of ISP architectures most suitable for SIMP: BISC. BISC is a LOAD/STORE architecture with single-sized instructions, but it loosens RISCs' constraint of single-cycle operations. We apply BISC to the SIMP processor prototype.

Currently we have several projects in progress regarding SIMP; the development of the SIMP processor prototype, the design of an optimizing C compiler capable of static code scheduling for the prototype, and the evaluation of SIMP architecture via software simulations.

Since we have not yet completed a software simulator of the SIMP processor prototype, we cannot report the performance estimate for the prototype here. The measurements obtained by a simple trace-driven simulator (see Table 1), however, show that a quadruple instruction-pipeline processor can achieve a performance of about 200-270% of a single instruction-pipeline processor [Irie88].

Table 1. Relative Speedups over Single Instruction Pipeline

Execution Model	Functional Unit	Number of Instruction-Pipelines			
		1	2	4	6
In-Order	Not Pipelined	1	1.45-1.58	2.04-2.23	2.36-2.62
	Pipelined	1.02-1.28	1.52-2.03	2.10-2.60	2.37-2.64
Out-of-Order	Not Pipelined	1	1.60-1.62	2.21-2.37	2.53-2.76
	Pipelined	1.03-1.33	1.71-2.27	2.39-2.72	2.59-2.98

We would like to emphasize that the results are very preliminary and the benchmarks are not optimized by static code scheduling. The results therefore encourage us to expect more performance enhancement in the SIMP processor prototype.

As our future work, we are planning a VLSI implementation of SIMP architecture. It will be done by implementing one or more instruction pipelines on a single-chip VLSI, and by constructing an SIMP processor using the chips as building blocks. We refer to the single-chip VLSI as *"pipeline-slice microprocessor"* on the analogy of bit-slice microprocessor. The processor organization using pipeline-slice microprocessors will support scalability toward more instruction pipelines. As bit-slice microprocessors can achieve a wide range of data width, pipeline-slice microprocessors will be able to achieve a wide range of performance.

We believe that SIMP architecture is one of the most promising architectures toward the coming generation of high-speed single processors.

Acknowledgements

We would like to thank the following students who have contributed or are now contributing to the project: T.Goto (currently with Toshiba Corp.), O.-B.Gwun, T.Hara, and T.Hironaka. We would also like to acknowledge the considerable contributions from A. Fukuda and T.Sueyoshi.

References

[Acosta86] R.D.Acosta, J.Kjelstrup, and H.C.Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Trans. Comput.*, vol.C-35, no.9, pp.815-828, Sept. 1986.

[Colwell87] R.P.Colwell, R.P.Nix, J.J.O'Donnell, D.B.Papworth, and P.K.Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *Proc. 2nd Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pp.180-192, Oct. 1987.

[Fisher81] J.A.Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Comput.*, vol. C-30, no.7, pp.478-490, July 1981.

[Fisher83] J.A.Fisher, "Very Long Instruction Word Architectures and the ELI-512," *Proc. 10th Ann. Int. Symp. Computer Architecture*, pp.140-150, June 1983.

[Hagiwara80] H.Hagiwara, S.Tomita, S.Oyanagi, and K.Shibayama, "A Dynamically Microprogrammable Computer with Low-Level Parallelism," *IEEE Trans. Comput.*, vol.C-29, no.7, pp.577-595, July 1980.

[Hwu87] W.W.Hwu and Y.N.Patt, "Checkpoint Repair for Out-of-order Execution Machines," *Proc. 14th Ann. Int. Symp. Computer Architecture*, pp.18-26, June 1987; also *IEEE Trans. Comput.*, vol.C-36, no.12, pp.1496-1514, Dec. 1987.

[Irie88] N.Irie, M.Kuga, K.Murakami, and S.Tomita, "Speedup Mechanisms and Performance Estimate for the SIMP Processor Prototype (in Japanese)," *IPSJ WGARC report* 73-11, Nov. 1988.

[Kuga89] M.Kuga, K.Murakami, and S.Tomita, "Low-level Parallel Processing Algorithms for the SIMP Processor Prototype (in Japanese)," *Proc. IPSJ Joint Symp. Parallel Processing'89*, pp.163-170, Feb. 1989.

[Lam88] M.Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *Proc. SIGPLAN'88 Conf. Programming Language Design and Implementation*, pp.318-328, June 1988.

[Lee84] J.K.F.Lee and A.J.Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, vol.17, no.1, pp.6-22, Jan. 1984.

[Murakami88]K.Murakami,A.Fukuda, T.Sueyoshi, and S.Tomita, "SIMP:Single Instruction stream/Multiple instruction Pipelining (in Japanese)," *IPSJ WGARC report* 69-4, Jan. 1988.

[Patt85] Y.N.Patt, W-M.Hwu, and M.Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction," *Proc. 18th Ann. Workshop on Microprogramming*, pp.103-108, Dec. 1985.

[Pleszkun88] A.R.Pleszkun and G.S.Sohi, "The Performance Potential of Multiple Functional Unit Processors," *Proc. 15th Ann. Int. Symp. Computer Architecture*, pp.37-44, May 1988.

[Rau89] B.R.Rau, D.W.L.Yen, W.Yen and R.A.Towle, "The Cydra 5 Departmental Supercomputer," *IEEE Computer*, vol.22, no.1, Jan. 1989.

[Smith85] J.E.Smith and A.R.Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *Proc. 12th Ann. Int. Symp. Computer Architecture*, pp.36-44, June 1985; also *IEEE Trans. Comput.*, vol.C-37, no.5, pp.562-573, May 1988.

[Sohi87] G.S.Sohi and S.Vajapeyam, "Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors," *Proc. 14th Ann. Int. Symp. Computer Architecture*, pp.27-34, June 1987.

[Tjaden70] G.S.Tjaden and M.J.Flynn, "Detection and Parallel Execution of Independent Instructions," *IEEE Trans. Comput.*, vol.C-19, no.10, pp.889-895, Oct. 1970.

[Tomasulo67] R.M.Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Res. Develop.*, vol.11, pp.25-33, Jan. 1967.

[Tomita83] S.Tomita, K.Shibayama, T.Kitamura, T.Nakata, and H.Hagiwara, "A User-Microprogrammable, Local Host Computer with Low-Level Parallelism," *Proc. 10th Ann. Int. Symp. Computer Architecture*, pp.151-157, June 1983.

[Tomita86] S.Tomita, K.Shibayama, T.Nakata, S.Yuasa, and H.Hagiwara, "A Computer with Low-Level Parallelism QA-2 - Its Applications to 3-D Graphics and Prolog/Lisp Machines -," *Proc. 13th Ann. Int. Symp. Computer Architecture*, pp.280-289, June 1986.

[Weiss84] S.Weiss and J.E.Smith, "Instruction Issue Logic for Pipelined Supercomputers," *Proc. 11th Ann. Int. Symp. Computer Architecture*, pp.110-118, June 1984; also *IEEE Trans. Comput.*, vol.C-33, no.11, pp.1013-1022, Nov. 1984.