A. Glossary

The following terms are defined in this Glossary:

- superscalar processor
- pipeline
- pipeline latency
- pipeline repeat rate
- out-of-order execution
- dynamic scheduling
- instruction fetch, decode, issue, execution, completion, and graduation
- active list
- free list and busy registers
- register renaming and unnaming
- nonblocking loads and stores
- speculative branching
- logical and physical registers
- register files
- ANDES architecture

A.1 Superscalar Processor

A superscalar processor is one that can fetch, execute and complete more than one instruction in parallel. By implication, a superscalar processor has more than one pipeline (see below).

A.2 Pipeline

In the processor **pipeline**, the execution of each instruction is divided into a sequence of simpler suboperations. Each suboperation is performed by a separate hardware section called a **stage**, and each stage passes its result to a succeeding stage.

Normally, each instruction only remains in each stage for a single cycle, and each stage begins executing a new instruction as previous instructions are being completed in later stages. Thus, a new instruction can often begin during every cycle.

Pipelines greatly improve the rate at which instructions can be executed, as long as there are no dependencies. The efficient use of a pipeline requires that several instructions be executed in parallel, however the result of any instruction is not available for several cycles after that instruction has entered the pipeline. Thus, new instructions must not depend on the results of instructions which are still in the pipeline.

A.3 Pipeline Latency

The **latency** of an execution pipeline is the number of cycles between the time an instruction is issued and the time a dependent instruction (which uses its result as an operand) can be issued.

In the R10000 processor, most integer instructions have a single-cycle latency, load instructions have a 2-cycle latency for cache hits, and floating-point addition and multiplication have a 2-cycle latency. Integer multiply, floating-point square-root, and all divide instructions are computed iteratively and have longer latencies.

A.4 Pipeline Repeat Rate

The **repeat rate** of the pipeline is the number of cycles that occur between the issuance of one instruction and the issuance of the next instruction to the same execution unit. In the R10000 processor, the main five pipelines all have repeat rates of one cycle, but the iterative units have longer repeat delays.

A.5 Out-of-Order Execution

The "program order" of instructions is the sequence in which they are fetched and decoded. In the R10000 processor, instructions may be issued, executed, and completed **out of program order**. They are always graduated in program order.

A.6 Dynamic Scheduling

The R10000 processor can issue instructions to functional units out of program order; this capability is known as **dynamic scheduling** or **dynamic issuing**.

The R10000 processor can dynamically issue an instruction as soon as all its operands are available and the required execution unit is not busy. Thus, an instruction is not delayed by a stalled previous instruction unless it needs the results of that previous instruction.

A.7 Instruction Fetch, Decode, Issue, Execution, Completion, and Graduation

In general, instructions are *fetched*, *decoded*, and *graduated* in their original program order, but may be *issued*, *executed*, and *completed* out of program order, as shown in Figure A-1.

- **Instruction fetching** is the process of reading instructions from the instruction cache.
- **Instruction decode** includes register renaming and initial dependency checks. For branch instructions, the branch path is predicted and the target address is computed.
- An instruction is **issued** when it is handed over to a functional unit for *execution*.
- An instruction is **complete** when its result has been computed and stored in a temporary physical register.
- An instruction **graduates** when this temporary result is committed as the new state of the processor. An instruction can graduate only after it and all previous instructions have been successfully completed.

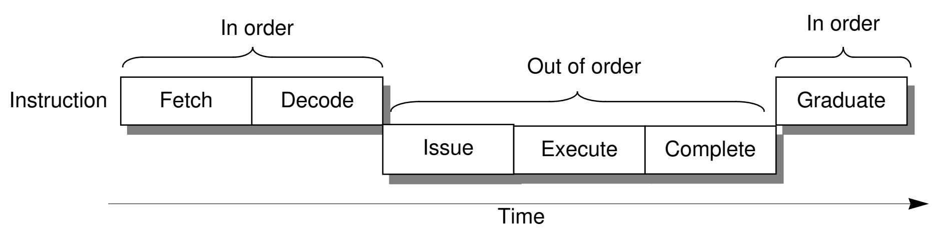

Figure A-1 Dynamic Scheduling

A.8 Active List

The R10000 processor's **active list** is a program-order list of decoded instructions. For each instruction, the active list indicates the physical register which contained the *previous* value of the destination register (if any). If this instruction graduates, that previous value is discarded and the physical register is returned to the free list. The active list records status, such as those instructions that have completed, or those instructions that have detected exceptions. Instructions are appended to the bottom of the list as they are decoded and instructions are removed from the top as they graduate.

A.9 Free List and Busy Registers

A **busy-bit table** indicates whether or not a result has been written into each of the physical registers. Each register is initially defined to be busy when it is moved from the **free list** to the active list; the register becomes available (“not busy”) when its instruction completes and its result is stored in the register file.

The busy-bit table is read for each operand while an instruction is decoded, and these bits are written into the queue with the instruction. If an operand is busy, the instruction must wait in the queue until the operand is “not busy.” The queues determine when an operand is ready by comparing the register number of the result coming out of each execution unit with the register number of each operand of the instructions waiting in the queue.

With a few exceptions, the integer and address queues have integer operand registers, and the floating-point queue has floating-point operand registers.

A.10 Register Renaming

As it executes instructions, the processor generates a myriad of *temporary* register results. These temporary values are stored in register files together with *permanent* values. The temporary values become new permanent values when their corresponding instructions graduate.

Register renaming is used to resolve data dependencies during the dynamic execution of instructions.

To ensure each instruction is given correct operand values, the logical register numbers (**names**) used in the instruction are mapped to physical registers. Each time a new value is put in a logical register, it is assigned to a new physical register. Thus, each physical register has only a single value. Dependencies are determined using these physical register numbers.

An example of register renaming is shown below. The following Doubleword Shift Left Logical instruction,

opcode	rs	rt	dest	sa	function
spec	-	r2	r3	2	DSLL

```
DSLL                r3,r2,2
```

has one register operand (*r2*) plus a 5-bit shift count of value two stored in the *sa* field; the value in *r2* is shifted left by two and this value is stored in *r3*.

The physical execution of the instruction above, with register renaming, is given below:

```
Physical execution        Rename operation
p3←p2 shift left 2        r3 = p3
```

When the DSLL instruction is executed, the *l*ogical destination register *r3* is assigned a new physical register, p3, from the free list.

Register renaming also allows exceptions to be handled in a precise manner. *Out-of-order execution* means that an instruction can change its result register even before all prior instructions have been completed. However, if any of the prior instructions cause an exception, the original register value must be restored. Since each new register value is loaded into a new physical register (physical register values are not overwritten until the physical register is placed in the free list), previous values remain unchanged in the original physical registers and these previous values can be restored.†

An instruction can be aborted up until the time it graduates, and all register and memory values can be restored to a precise state following any exception. This state is restored by *unnaming* the temporary physical registers assigned to subsequent instructions.

Registers are **unnamed** by writing the old destination register into the mapping table and returning the new destination register to the free list. Unnaming is done in reverse program order, in case a logical register was used more than once. After renaming, the register files contain only the permanent values which were created by instructions prior to the exception.

Once an instruction has graduated, all previous values are lost.

A.11 Nonblocking Loads and Stores

Loads and stores are **nonblocking**; that is, cache misses do not stall the processor. All other parts of the processor may continue to work on non-dependent instructions while as many as four cache misses are being processed.

† This same technique is used to reverse mispredicted speculative branches.

A.12 Speculative Branching

Normally, about one of every six instructions is a branch. Since four instructions are fetched each cycle, the R10000 processor encounters, on average, a branch instruction every other cycle, as shown in Figure A-2.

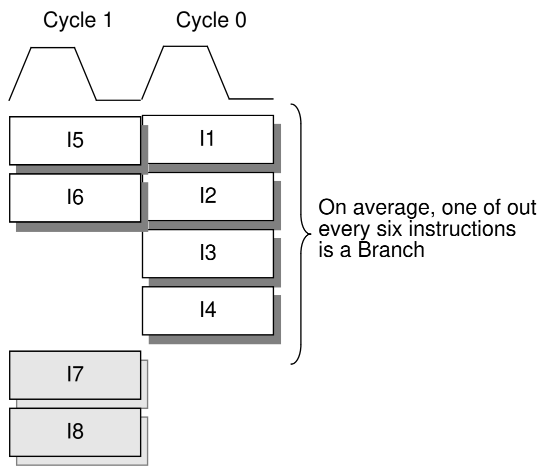

Figure A-2 Speculative Branching

When a branch instruction was encountered in previous processors, the instruction fetch and instruction issue halted until it was determined whether or not to take the branch. For instance, a branch delay slot was designed into the MIPS architecture to handle the intrinsic delay of a branch and to keep the pipeline filled.

Since the processor fetches up to four instructions each clock cycle, there is not enough time to resolve branches without stalling the fetch/decode circuitry. The processor therefore **predicts** the outcome of every branch and **speculatively executes** the branch based on this branch prediction.

The branch prediction circuit consists of a 512-entry RAM, using a 2-bit prediction scheme: two bits are assigned to a branch instruction, and indicate whether or not the branch was taken the last time it occurred. The four possible prediction states are: strongly taken, weakly taken, weakly not taken, strongly not taken. If the branch was taken the last two times, there is a good probability it will be taken this time too — or the inverse.†

The R10000 processor can speculate up to four branches deep. Shadow copies of the mapping tables are kept every time a prediction is made, allowing the R10000 processor to recover from a mispredicted branch in a single cycle.

† Simulations have shown the R10000 branch prediction algorithm to be over 90% accurate.

A.13 Logical and Physical Registers

Register renaming (described above) distinguishes between **logical registers**, which are referenced within instruction fields, and **physical registers**, which are actually located in the hardware register file. The programmer is only aware of logical registers; the implementation of physical registers is entirely transparent.

Logical register numbers are dynamically mapped onto physical register numbers. This mapping uses *mapping tables* which are updated after each instruction is decoded; each new result is written into a new physical register. This value is temporary and the previous contents of each logical register can be restored if its instruction must be aborted following an exception or a mispredicted branch.

Register renaming simplifies dependency checks. Logical register numbers can be ambiguous when instructions are executed out of order, since a succession of different values may be assigned to the same register. But physical register numbers uniquely identify each result, making dependency checking unambiguous.

The queues and execution units use physical register numbers. Integer and floating-point registers are implemented with separate renaming hardware and multi-port register files.

A.14 Register Files

The R10000 processor has two 64-bit-wide register files to store integer and floating-point values. Each file contains 64 registers. The integer register file has seven read and three write ports; the floating-point register file has five read and three write ports.

The integer and floating-point pipelines each use two dedicated operand ports and one dedicated result port in the appropriate register file. The Load/Store unit uses two dedicated integer operand ports for address calculation. It must also load or store either integer or floating-point values, sharing a result port and a read port in both register files.

These shared ports are also used to move data between the integer and floating-point register files, to store branch and link return addresses, and to read the target address for branch register instructions.

A.15 ANDES Architecture

The R10000 processor uses the MIPS ANDES architecture, or *Architecture with Non-sequential Dynamic Execution Scheduling*.