Advanced Topics in Computer Architecture

Lecture 2
Pipelining: Basic and Intermediate Concepts

Marenglen Biba
Department of Computer Science
University of New York Tirana

Outline

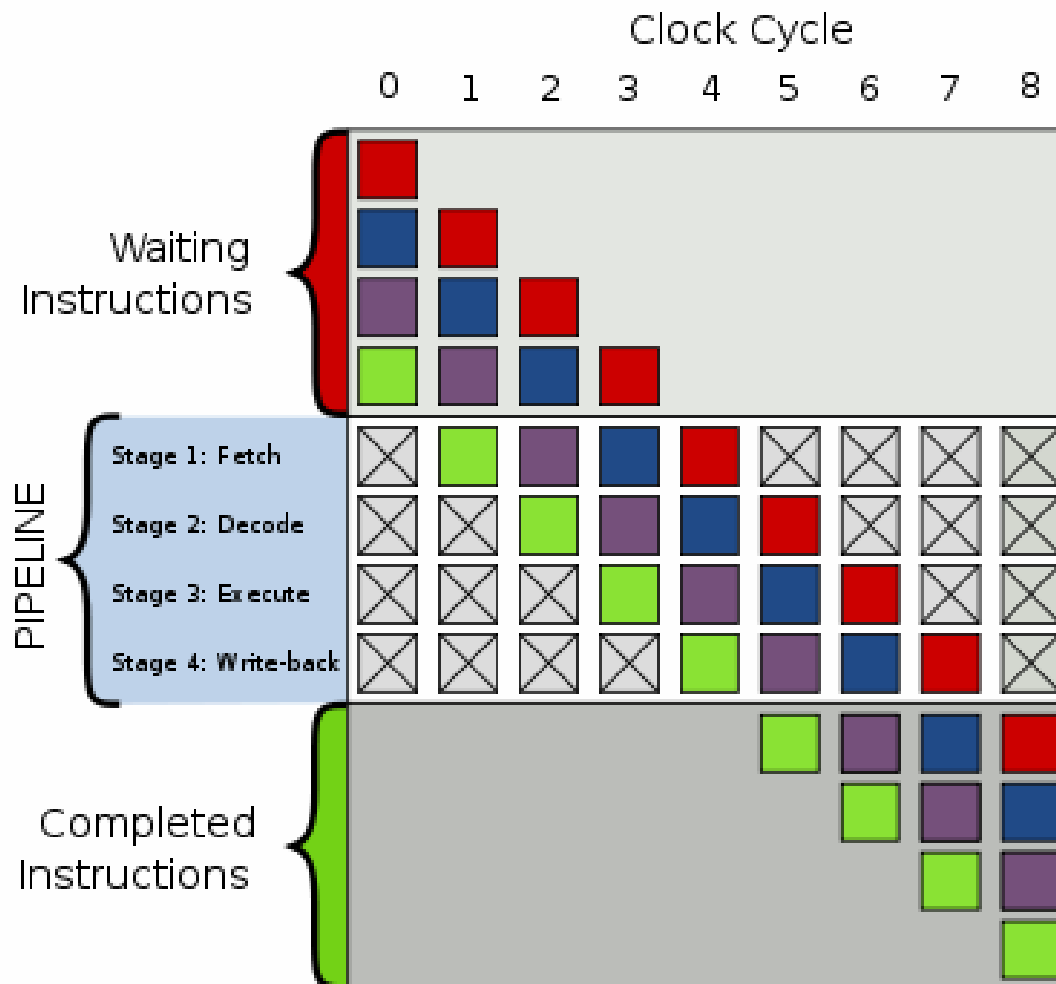
- Introduction
- The Major Hurdle of Pipelining—Pipeline Hazards
- How Is Pipelining Implemented?
- What Makes Pipelining Hard to Implement?
- Extending the MIPS Pipeline to Handle Multicycle Operations
- Putting It All Together: The MIPS R4000 Pipeline
- Crosscutting Issues
- Fallacies and Pitfalls

What Is Pipelining?

- *Pipelining* is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction.
- Today, pipelining is the key implementation technique used to make fast CPUs.
- A pipeline is like an assembly line. In an automobile assembly line, there are many steps, each contributing something to the construction of the car. Each step operates in parallel with the other steps, although on a different car.
- In a computer pipeline, each step in the pipeline completes a part of an instruction.
- Like the assembly line, different steps are completing different parts of different instructions in parallel. Each of these steps is called a *pipe stage* or a *pipe segment*.

Pipelining

- Example of 4-stage pipeline. The colored boxes represent instructions independent of each other



Throughput

- In an automobile assembly line, *throughput* is defined as the number of cars per hour and is determined by how often a completed car exits the assembly line.
- Likewise, the *throughput* of an instruction pipeline is determined by how often an instruction exits the pipeline.
- Because the pipe stages are hooked together, all the stages must be ready to proceed at the same time, just as we would require in an assembly line.
- The time required between moving an instruction one step down the pipeline is a *processor cycle*.

  Because all stages proceed at the same time, the length of a processor cycle is determined by the time required for the slowest pipe stage, just as in an auto assembly line, the longest step would determine the time between advancing the line.
- In a computer, this processor cycle is usually 1 clock cycle (sometimes it is 2, rarely more).

Speedup

- The pipeline designer's goal is to balance the length of each pipeline stage, just as the designer of the assembly line tries to balance the time for each step in the process.
- If the stages are perfectly balanced, then the time per instruction on the pipelined processor — assuming ideal conditions—is equal to:

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

- Under these conditions, the speedup from pipelining equals the number of pipe stages, just as an assembly line with n stages can ideally produce cars *n* times as fast.
- Usually, however:
  - the stages will not be perfectly balanced
  - furthermore, pipelining does involve some overhead.

Reducing CPI

- Pipelining yields a reduction in the average execution time per instruction.
- Depending on what you consider as the baseline, the reduction can be viewed as decreasing the number of clock cycles per instruction (CPI), as decreasing the clock cycle time, or as a combination.
- If the starting point is a processor that takes multiple clock cycles per instruction, then pipelining is usually viewed as reducing the CPI.
- This is the primary view we will take.
- If the starting point is a processor that takes 1 (long) clock cycle per instruction, then pipelining decreases the clock cycle time.

Pipelining and sequential streams

- Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream.
- It has the substantial advantage that, unlike some speedup techniques it is not visible to the programmer.
- Here we will cover the concept of pipelining using a classic five-stage pipeline;
  - later we will investigate the more sophisticated pipelining techniques in use in modern processors.
- Before we say more about pipelining and its use in a processor, we need a simple instruction set, which we introduce next.

The Basics of a RISC Instruction Set

- RISC architectures are characterized by a few key properties, which dramatically simplify their implementation:
  - All operations on data apply to data in registers and typically change the entire register (32 or 64 bits per register).
  - The only operations that affect memory are load and store operations that move data from memory to a register or to memory from a register, respectively.
  - Load and store operations that load or store less than a full register (e.g., a byte, 16 bits, or 32 bits) are often available.

- The instruction formats are few in number with all instructions typically being one size.

MIPS64

- We will use MIPS64, the 64-bit version of the MIPS instruction set.
- The extended 64-bit instructions are generally designated by having a D on the start or end of the mnemonic.
- For example DADD is the 64-bit version of an add instruction, while LD is the 64-bit version of a load instruction.
- Like other RISC architectures, the MIPS instruction set provides 32 registers, although register 0 always has the value 0.
- Most RISC architectures, like MIPS, have three classes of instructions:

RISC Instructions

ALU instructions

- These instructions take either two registers or a register, operate on them, and store the result into a third register.
- Typical operations include add (DADD), subtract (DSUB), and logical operations (such as AND or OR), which do not differentiate between 32-bit and 64-bit versions.

RISC Instructions

Load and store instructions

- These instructions take a register source, called the *base register*, and an immediate field (16-bit in MIPS), called the *offset*, as operands.
- The sum — called the *effective address* — of the contents of the base register and the sign-extended offset is used as a memory address.
- In the case of a load instruction, a second register operand acts as the destination for the data loaded from memory.

RISC Instructions

Branches and jumps

- Branches are conditional transfers of control.
- There are usually two ways of specifying the branch condition in RISC architectures:
  - with a set of condition bits (sometimes called a condition code) or
  - by a limited set of comparisons between a pair of registers or between a register and zero.
- MIPS uses the latter.

A Simple Implementation of a RISC Instruction Set

- To understand how a RISC instruction set can be implemented in a pipelined fashion, we need to understand how it is implemented *without* pipelining.
- This section shows a simple implementation where every instruction takes at most 5 clock cycles.
- We will extend this basic implementation to a pipelined version, resulting in a much lower CPI.

Clock cycles

- Every instruction in this RISC subset can be implemented in at most 5 clock cycles. The 5 clock cycles are as follows:

1. *Instruction fetch cycle* (IF):

   Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by adding 4 (since each instruction is 4 bytes) to the PC.

2. *Instruction decode/register fetch cycle* (ID):

   Decode the instruction and read the registers corresponding to register source specifiers from the register file.

Clock cycles

3. *Execution/effective address cycle* (EX):
   - The ALU operates on the operands prepared in the prior cycle, performing one of three functions depending on the instruction type.
4. *Memory access* (MEM):
   - If the instruction is a load, memory does a read using the effective address computed in the previous cycle. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.
5. *Write-back cycle* (WB):
   - Register-Register ALU instruction or Load instruction:
     - Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction).

The Classic Five-Stage Pipeline for a RISC Processor

- We can pipeline the execution described above with almost no changes by simply starting a new instruction on each clock cycle.
- Each of the clock cycles from the previous section becomes a *pipe stage* — a cycle in the pipeline.
- Although each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction and will be executing some part of the five different instructions.

Simple RISC Pipeline

Instruction number	Clock number 1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Figure A.1 Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its 5-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write back.

To start with, we have to determine what happens on every clock cycle of the processor and make sure we don't try to perform **two different operations with the same data path resource on the same clock cycle**.
For example, a single ALU cannot be asked to compute an effective address and perform a subtract operation at the same time. Thus, we must ensure that the **overlap of instructions in the pipeline cannot cause such a conflict.**
Fortunately, the simplicity of a RISC instruction set makes resource evaluation relatively easy.

RISC Pipeline

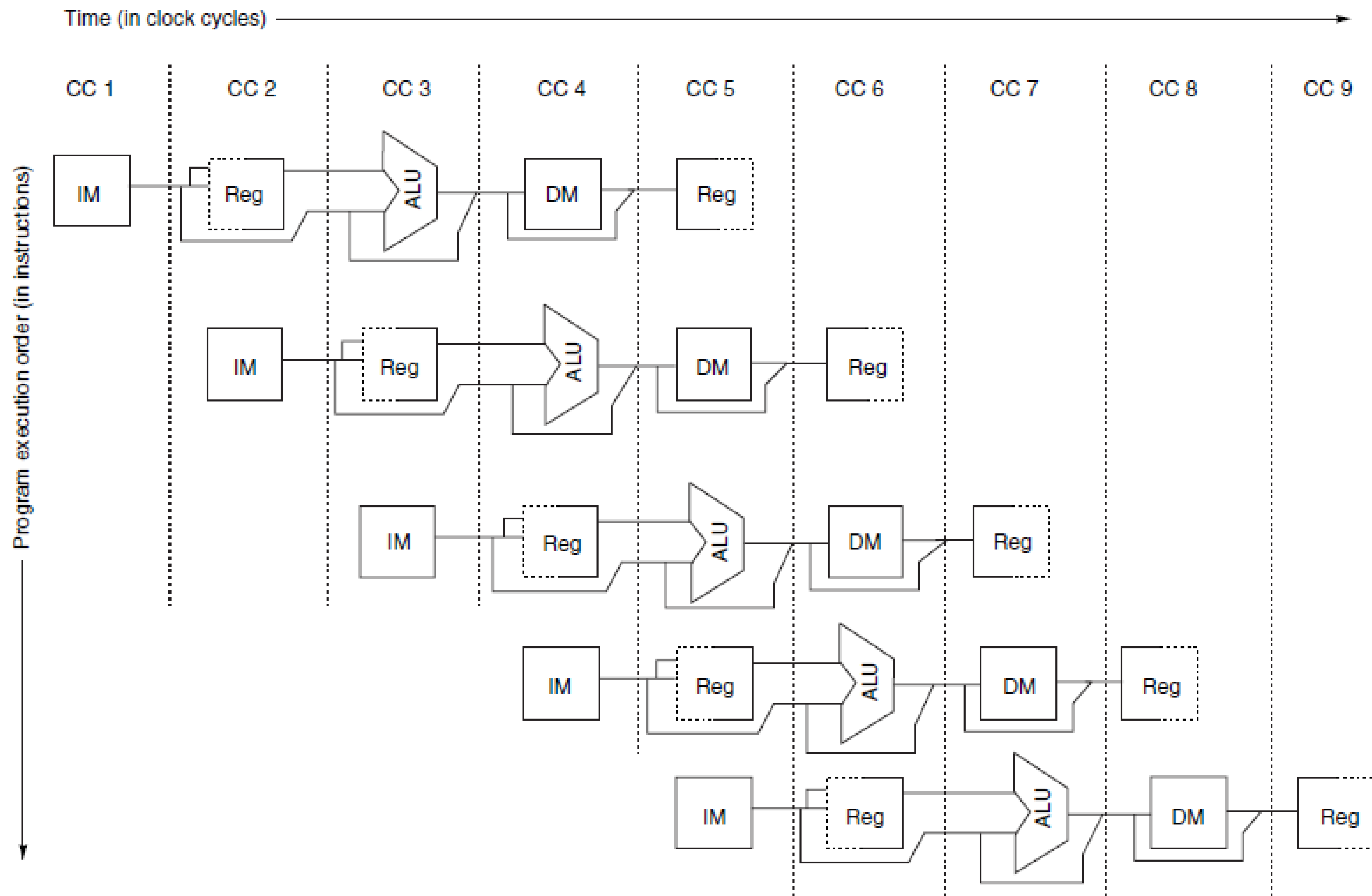


Figure A.2 The pipeline can be thought of as a series of data paths shifted in time. This shows the overlap among the parts of the data path, with clock cycle 5 (CC 5) showing the steady-state situation. Because the register file is used as a source in the ID stage and as a destination in the WB stage, it appears twice. We show that it is read in one part of the stage and written in another by using a solid line, on the right or left, respectively, and a dashed line on the other side. The abbreviation IM is used for instruction memory, DM for data memory, and CC for clock cycle.

Pipeline registers

- Although it is critical to ensure that instructions in the pipeline do not attempt to use the hardware resources at the same time, we must also ensure that instructions in different stages of the pipeline do not interfere with one another.

- This separation is done by introducing *pipeline registers* between successive stages of the pipeline, so that at the end of a clock cycle all the results from a given stage are stored into a register that is used as the input to the next stage on the next clock cycle.

- Pipeline drawn with these pipeline registers => next slide

Pipeline registers

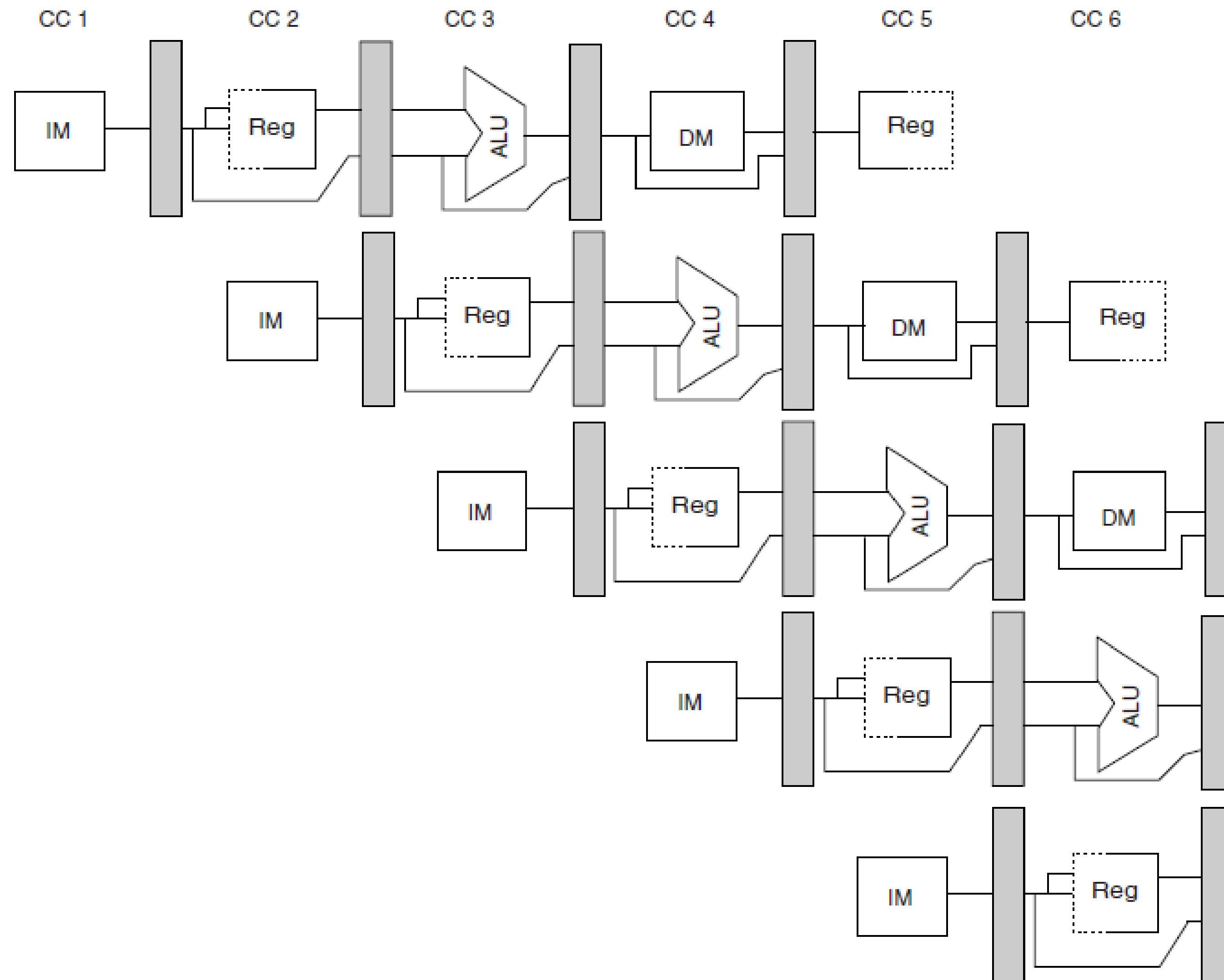


Figure A.3 **A pipeline showing the pipeline registers between successive pipeline stages.** Notice that the registers prevent interference between two different instructions in adjacent stages in the pipeline. The registers also play the critical role of carrying data for a given instruction from one stage to the other. The edge-triggered property of registers—that is, that the values change instantaneously on a clock edge—is critical. Otherwise, the data from one instruction could interfere with the execution of another!

Performance Issues in Pipelining

- Pipelining increases the CPU instruction throughput — the number of instructions completed per unit of time — but it does not reduce the execution time of an individual instruction.
- In fact, it usually slightly increases the execution time of each instruction due to overhead in the control of the pipeline.
- The increase in instruction throughput means that a program runs faster and has lower total execution time, even though no single instruction runs faster!

Pipeline imbalance

- The fact that the execution time of each instruction does not decrease puts limits on the practical depth of a pipeline.
- Limits arise from imbalance among the pipe stages and from pipelining overhead.
- Imbalance among the pipe stages reduces performance since the clock can run no faster than the time needed for the slowest pipeline stage.

Pipeline overhead

- Pipeline overhead arises from the combination of pipeline register delay and clock skew.
  - The pipeline registers add setup time, which is the time that a register input must be stable before the clock signal that triggers a write occurs, plus propagation delay to the clock cycle.
- In circuit designs, *clock skew* (sometimes *timing skew*) is a phenomenon in synchronous circuits in which the clock signal (sent from the clock circuit) arrives at different components at different times.
- This overhead affected the performance gains achieved by the Pentium 4 versus the Pentium III.

Pipeline: Example

- Consider an unpipelined processor.
- Assume that it has a 1 ns clock cycle and that it uses 4 cycles for ALU operations and branches and 5 cycles for memory operations.
- Assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively.
- Suppose that due to clock skew and setup, pipelining the processor adds 0.2 ns of overhead to the clock.
- Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

Pipeline: Example

- The average instruction execution time on the unpipelined processor is:

$$\begin{aligned}\text{Average instruction execution time} &= \text{Clock cycle} \times \text{Average CPI} \\ &= 1\text{ ns} \times ((40\% + 20\%) \times 4 + 40\% \times 5) \\ &= 1\text{ ns} \times 4.4 \\ &= 4.4\text{ ns}\end{aligned}$$

- In the pipelined implementation, the clock must run at the speed of the slowest stage plus overhead, which will be 1 + 0.2 or 1.2 ns; this is the average instruction execution time. Thus, the speedup from pipelining is:

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} \\ &= \frac{4.4\text{ ns}}{1.2\text{ ns}} = 3.7\text{ times}\end{aligned}$$

- This simple RISC pipeline would function just fine for integer instructions if every instruction were independent of every other instruction in the pipeline.
- In reality, instructions in the pipeline can depend on one another!!!

Outline

- Introduction
- The Major Hurdle of Pipelining—Pipeline Hazards
- How Is Pipelining Implemented?
- What Makes Pipelining Hard to Implement?
- Extending the MIPS Pipeline to Handle Multicycle Operations
- Putting It All Together: The MIPS R4000 Pipeline
- Crosscutting Issues
- Fallacies and Pitfalls

The Major Hurdle of Pipelining—Pipeline Hazards

- There are situations, called *hazards,* that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards:

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.
2. *Data hazards* arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
3. *Control hazards* arise from the pipelining of branches and other instructions that change the PC.

Stalling

- Hazards in pipelines can make it necessary to *stall* the pipeline. Avoiding a hazard often requires that some instructions in the pipeline be allowed to proceed while others are delayed.
- For the pipelines we discuss here, when an instruction is stalled:
  - *all instructions issued later than the stalled instruction — and hence not as far along in the pipeline — are also stalled.*
  - *all instructions issued earlier than the stalled instruction — and hence farther along in the pipeline — must continue, since otherwise the hazard will never clear.*
- As a result, no new instructions are fetched during the stall.

Performance of Pipelines with Stalls

- A stall causes the pipeline performance to degrade from the ideal performance.
- Let's look at a simple equation for finding the actual speedup from pipelining:

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}$$

$$= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

Performance of Pipelines with Stalls

- Pipelining can be thought of as decreasing the CPI or the clock cycle time. Since it is traditional to use the CPI to compare pipelines, let's start with that assumption.

- The ideal CPI on a pipelined processor is almost always 1. Hence, we can compute the pipelined CPI:

CPI pipelined = Ideal CPI + Pipeline stall clock cycles per instruction

= 1 + Pipeline stall clock cycles per instruction

Performance of Pipelines with Stalls

- If we ignore the cycle time overhead of pipelining and assume the stages are perfectly balanced, then the cycle time of the two processors can be equal, leading to:

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

- One important simple case is where all instructions take the same number of cycles, which must also equal the number of pipeline stages (also called the *depth of the pipeline*). In this case, the unpipelined CPI is equal to the depth of the pipeline, leading to:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

- *If there are no pipeline stalls, this leads to the intuitive result that pipelining can improve performance by the depth of the pipeline.*

Structural Hazards

- When a processor is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.

- If some combination of instructions cannot be accommodated because of resource conflicts, the processor is said to have a *structural hazard.*

Structural Hazards

- The most common instances of structural hazards arise when some functional unit is not fully pipelined.
  - Then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle.
- Another common way that structural hazards appear is when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute.
  - For example, a processor may have only one register-file write port, but under certain circumstances, the pipeline might want to perform two writes in a clock cycle. This will generate a structural hazard.

Structural Hazards

- When a sequence of instructions encounters this hazard, the pipeline will stall one of the instructions until the required unit is available. Such stalls will increase the CPI from its usual ideal value of 1.
- Some pipelined processors have shared a single-memory pipeline for data and instructions. As a result, when an instruction contains a data memory reference, it will conflict with the instruction reference for a later instruction, as shown in Figure A.4.
- To resolve this hazard, we stall the pipeline for 1 clock cycle when the data memory access occurs.
- A stall is commonly called a *pipeline bubble* or just *bubble,* since it floats through the pipeline taking space but carrying no useful work. We will see another type of stall when we talk about data hazards.

Structural Hazard

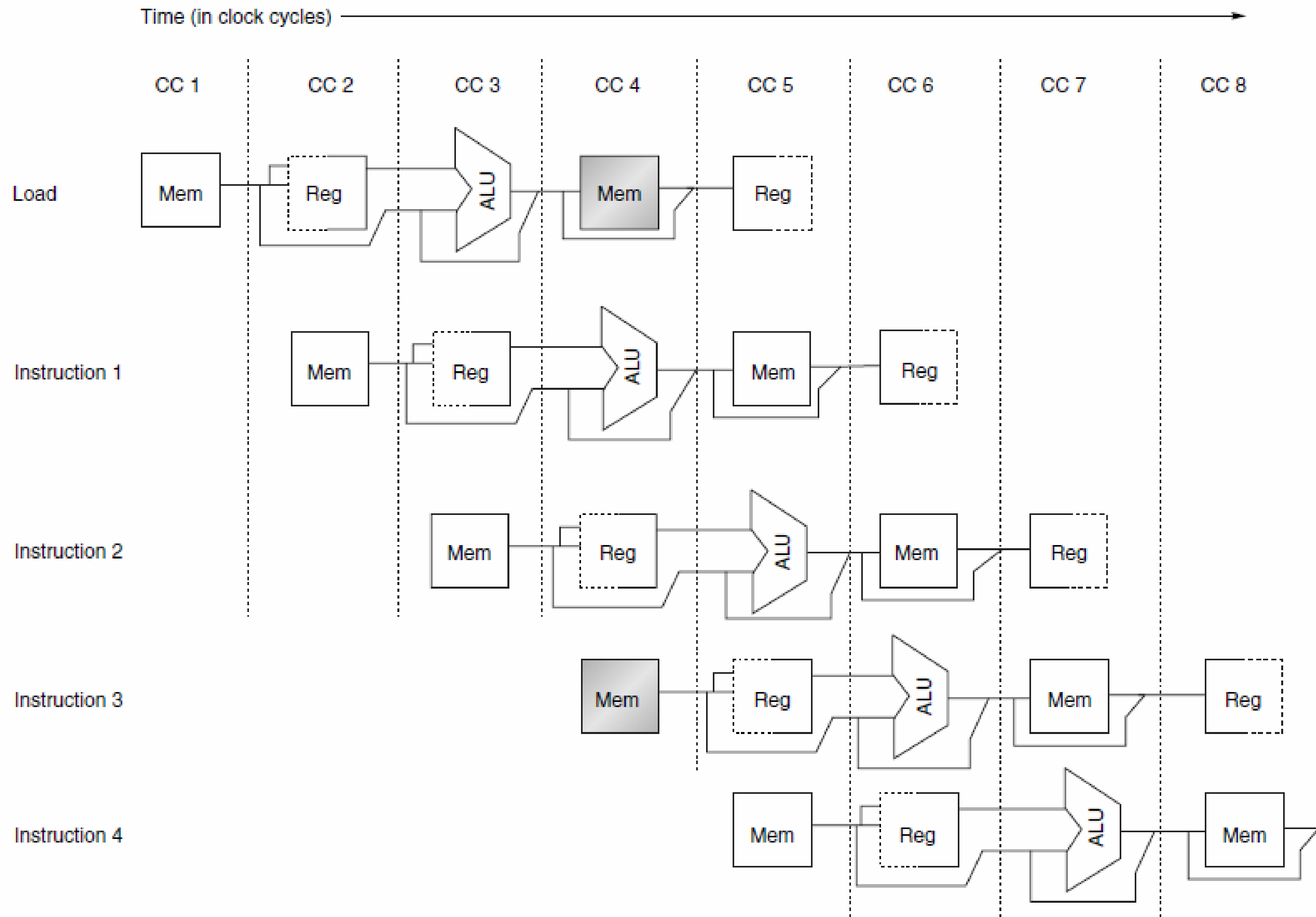


Figure A.4 A processor with only one memory port will generate a conflict whenever a memory reference occurs. In this example the load instruction uses the memory for a data access at the same time instruction 3 wants to fetch an instruction from memory.

Stalled Pipeline

Instruction	Clock cycle number									
	1	2	3	4	5	6	7	8	9	10
Load instruction	IF	ID	EX	MEM	WB					
Instruction $i + 1$		IF	ID	EX	MEM	WB				
Instruction $i + 2$			IF	ID	EX	MEM	WB			
Instruction $i + 3$				stall	IF	ID	EX	MEM	WB	
Instruction $i + 4$						IF	ID	EX	MEM	WB
Instruction $i + 5$							IF	ID	EX	MEM
Instruction $i + 6$								IF	ID	EX

Figure A.5 A pipeline stalled for a structural hazard—a load with one memory port. As shown here, the load instruction effectively steals an instruction-fetch cycle, causing the pipeline to stall—no instruction is initiated on clock cycle 4 (which normally would initiate instruction $i + 3$). Because the instruction being fetched is stalled, all other instructions in the pipeline before the stalled instruction can proceed normally. The stall cycle will continue to pass through the pipeline, so that no instruction completes on clock cycle 8. Sometimes these pipeline diagrams are drawn with the stall occupying an entire horizontal row and instruction 3 being moved to the next row; in either case, the effect is the same, since instruction $i + 3$ does not begin execution until cycle 5. We use the form above, since it takes less space in the figure. Note that this figure assumes that instruction $i + 1$ and $i + 2$ are not memory references.

Example

- Let’s see how much the load structural hazard might cost.
- Suppose that data references constitute 40% of the mix, and that the ideal CPI of the pipelined processor, ignoring the structural hazard, is 1.
- Assume that the processor with the structural hazard has a clock rate that is 1.05 times higher than the clock rate of the processor without the hazard. Disregarding any other performance losses, is the pipeline with or without the structural hazard faster, and by how much?

Example

- There are several ways we could solve this problem. Perhaps the simplest is to compute the average instruction time on the two processors:

$$\text{Average instruction time} = \text{CPI} \times \text{Clock cycle time}$$

- Since it has no stalls, the average instruction time for the ideal processor is simply the Clock cycle time ideal.
- The average instruction time for the processor with the structural hazard is:

$$\begin{aligned}\text{Average instruction time} &= \text{CPI} \times \text{Clock cycle time} \\ &= (1 + 0.4 \times 1) \times \frac{\text{Clock cycle time}_{\text{ideal}}}{1.05} \\ &= 1.3 \times \text{Clock cycle time}_{\text{ideal}}\end{aligned}$$

- Clearly, the processor without the structural hazard is faster; we can use the ratio of the average instruction times to conclude that the processor without the hazard is 1.3 times faster.

Instruction buffers

- As an alternative to the structural hazard, the designer could provide a separate memory access for instructions, either by:
  - splitting the cache into separate instruction and data caches, or by
  - using a set of buffers, usually called *instruction buffers*, to hold instructions

Data Hazards

- Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor.
- Consider the pipelined execution of these instructions:

DADD	R1,R2,R3
DSUB	R4,R1,R5
AND	R6,R1,R7
OR	R8,R1,R9
XOR	R10,R1,R11

- All the instructions after the DADD use the result of the DADD instruction.

Data Hazards

- As shown in Figure A.6 (next slide), the DADD instruction writes the value of R1 in the WB pipe stage, but the DSUB instruction reads the value during its ID stage.
- This problem is called a *data hazard*.
- Unless precautions are taken to prevent it, the DSUB instruction will read the wrong value and try to use it.

Data Hazards

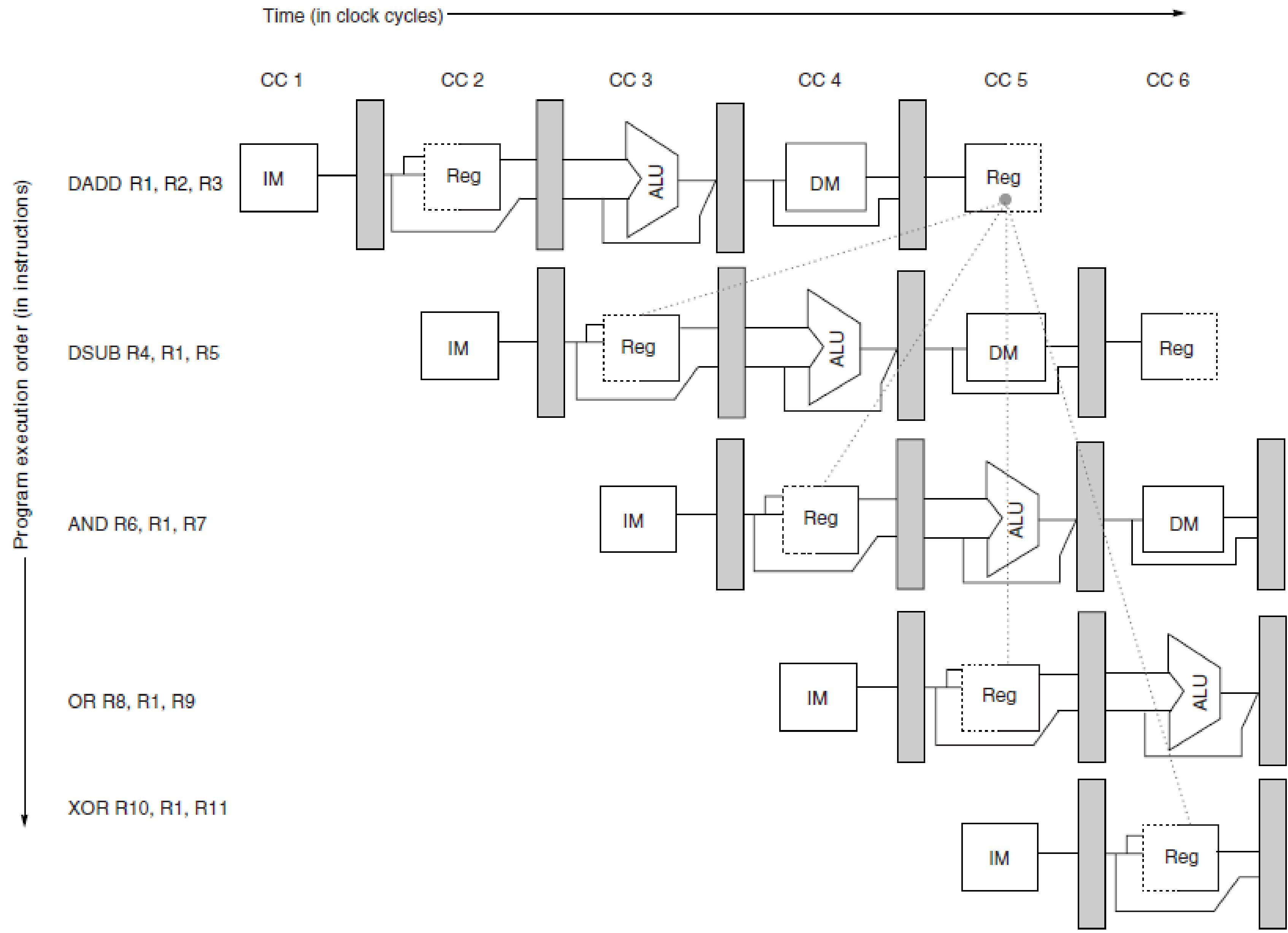


Figure A.6 The use of the result of the DADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

Minimizing Data Hazard Stalls by Forwarding

- The problem posed can be solved with a simple hardware technique called *forwarding* (also called *bypassing* and sometimes *short-circuiting*).
- The key insight in forwarding is that the result is not really needed by the DSUB until after the DADD actually produces it.
- If the result can be moved from the pipeline register where the DADD stores it to where the DSUB needs it, then the need for a stall can be avoided.
- Using this observation, forwarding works as follows:

Minimizing Data Hazard Stalls by Forwarding

1. The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.

2. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

Minimizing Data Hazard Stalls by Forwarding

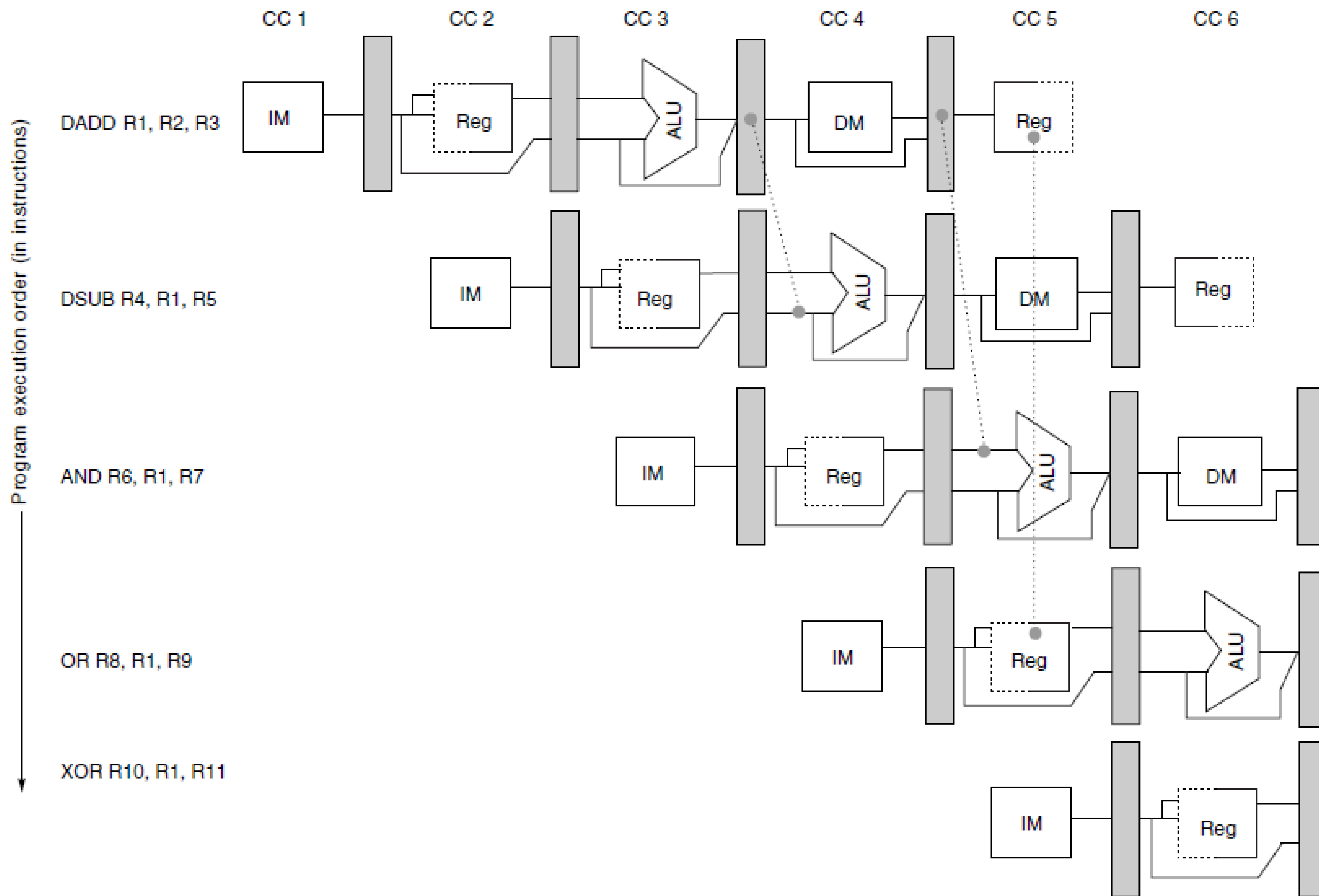


Figure A.7 **A set of instructions that depends on the DADD result uses forwarding paths to avoid the data hazard.** The inputs for the DSUB and AND instructions forward from the pipeline registers to the first ALU input. The OR receives its result by forwarding through the register file, which is easily accomplished by reading the registers in the second half of the cycle and writing in the first half, as the dashed lines on the registers indicate. Notice that the forwarded result can go to either ALU input; in fact, both ALU inputs could use forwarded inputs from either the same pipeline register or from different pipeline registers. This would occur, for example, if the AND instruction was AND R6, R1, R4.

Data Hazards Requiring Stalls

- Unfortunately, not all potential data hazards can be handled by bypassing.
- Consider the following sequence of instructions:

```
LD      R1,0(R2)
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
```

Data Hazards Requiring Stalls

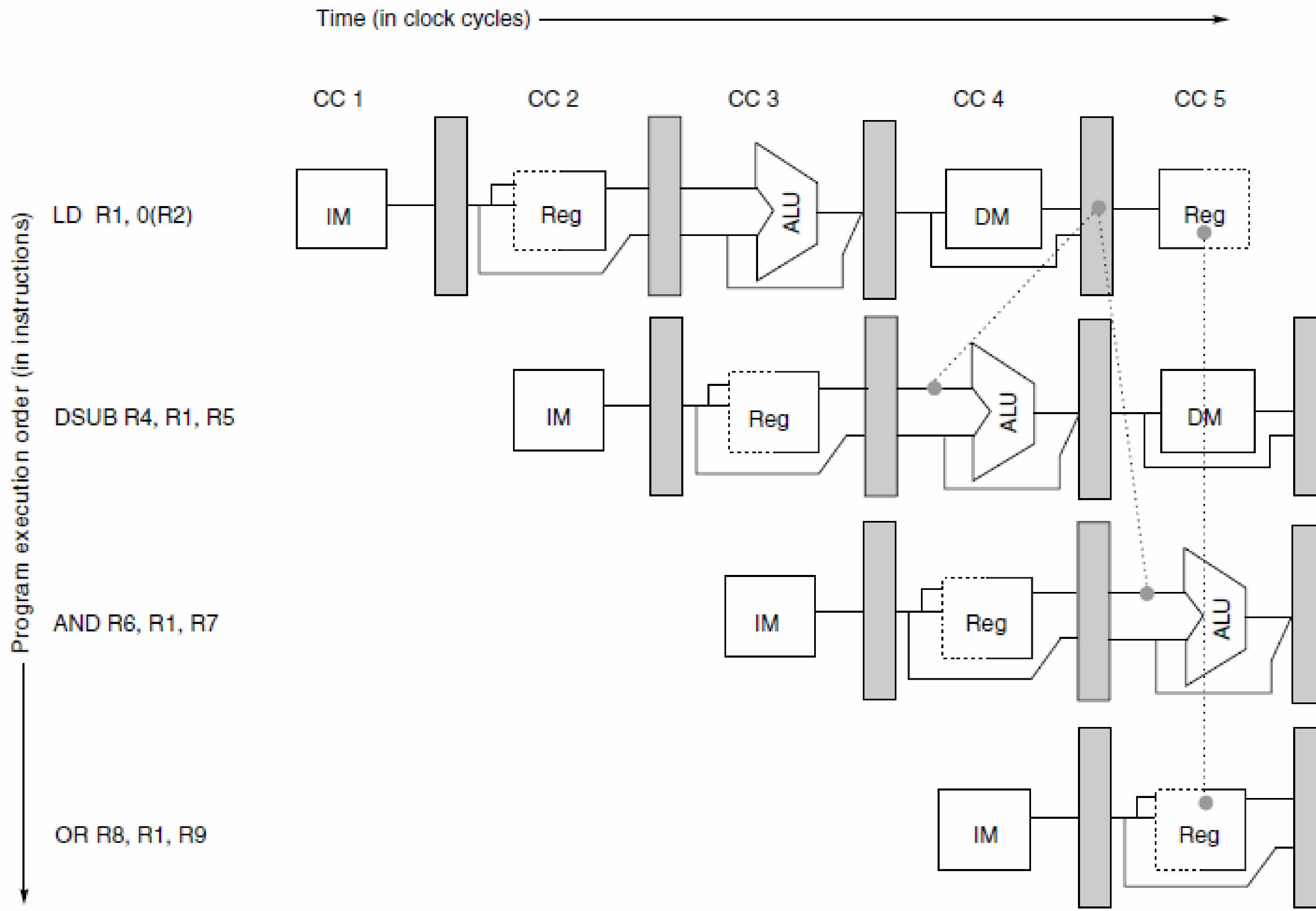


Figure A.9 The load instruction can bypass its results to the AND and OR instructions, but not to the DSUB, since that would mean forwarding the result in "negative time."

Pipeline interlock

- The load instruction has a delay or latency that cannot be eliminated by forwarding alone.
- Instead, we need to add hardware, called a *pipeline interlock*, to preserve the correct execution pattern.
- In general, *a pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared*.
- In this case, the interlock stalls the pipeline, beginning with the instruction that wants to use the data until the source instruction produces it.
- This pipeline interlock introduces a stall or bubble, just as it did for the structural hazard.
- The CPI for the stalled instruction increases by the length of the stall (1 clock cycle in this case).

Pipeline interlock

LD	R1,0(R2)	IF	ID	EX	MEM	WB			
DSUB	R4,R1,R5		IF	ID	EX	MEM	WB		
AND	R6,R1,R7			IF	ID	EX	MEM	WB	
OR	R8,R1,R9				IF	ID	EX	MEM	WB

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	stall	EX	MEM	WB		
AND	R6,R1,R7			IF	stall	ID	EX	MEM	WB	
OR	R8,R1,R9				stall	IF	ID	EX	MEM	WB

Figure A.10 In the top half, we can see why a stall is needed: The MEM cycle of the load produces a value that is needed in the EX cycle of the DSUB, which occurs at the same time. This problem is solved by inserting a stall, as shown in the bottom half.

Branch Hazards

- *Control hazards* can cause a greater performance loss for the MIPS pipeline than do data hazards.

- When a branch is executed, it may or may not change the PC to something other than its current value plus 4.

- Recall that if a branch changes the PC to its target address, it is a *taken* branch; if it falls through, it is *not taken,* or *untaken*.

- If instruction *i* is a taken branch, then the PC is normally not changed until the end of ID, after the completion of the address calculation and comparison.

Branch Hazards

- The simplest method of dealing with branches is to redo the fetch of the instruction following a branch, once we detect the branch during ID (when instructions are decoded).
- The first IF cycle is essentially a stall, because it never performs useful work.
- One stall cycle for every branch will yield a performance loss of 10% to 30% depending on the branch frequency, but there are some techniques to deal with this loss.

Branch Hazards

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

Figure A.11 A branch causes a 1-cycle stall in the five-stage pipeline. The instruction after the branch is fetched, but the instruction is ignored, and the fetch is restarted once the branch target is known. It is probably obvious that if the branch is not taken, the second IF for branch successor is redundant. This will be addressed shortly.

Outline

- Introduction
- The Major Hurdle of Pipelining—Pipeline Hazards
- How Is Pipelining Implemented?
- What Makes Pipelining Hard to Implement?
- Extending the MIPS Pipeline to Handle Multicycle Operations
- Putting It All Together: The MIPS R4000 Pipeline
- Crosscutting Issues
- Fallacies and Pitfalls

Unpipelined MIPS

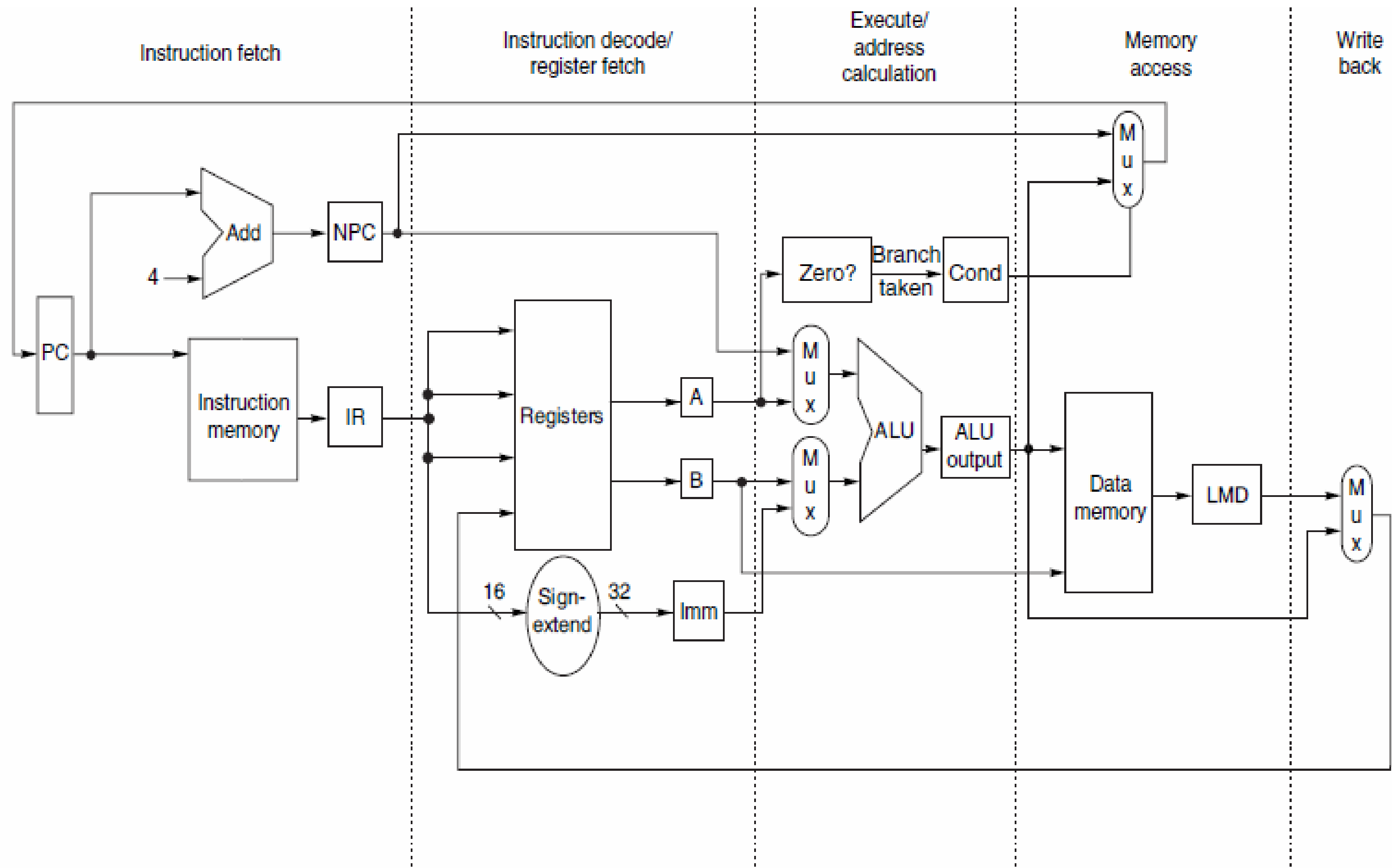


Figure A.17 **The implementation of the MIPS data path allows every instruction to be executed in 4 or 5 clock cycles.** Although the PC is shown in the portion of the data path that is used in instruction fetch and the registers are shown in the portion of the data path that is used in instruction decode/register fetch, both of these functional units are read as well as written by an instruction. Although we show these functional units in the cycle corresponding to where they are read, the PC is written during the memory access clock cycle and the registers are written during the write-back clock cycle. In both cases, the writes in later pipe stages are indicated by the multiplexer output (in memory access or write back), which carries a value back to the PC or registers. These backward-flowing signals introduce much of the complexity of pipelining, since they indicate the possibility of hazards.

Pipelined MIPS

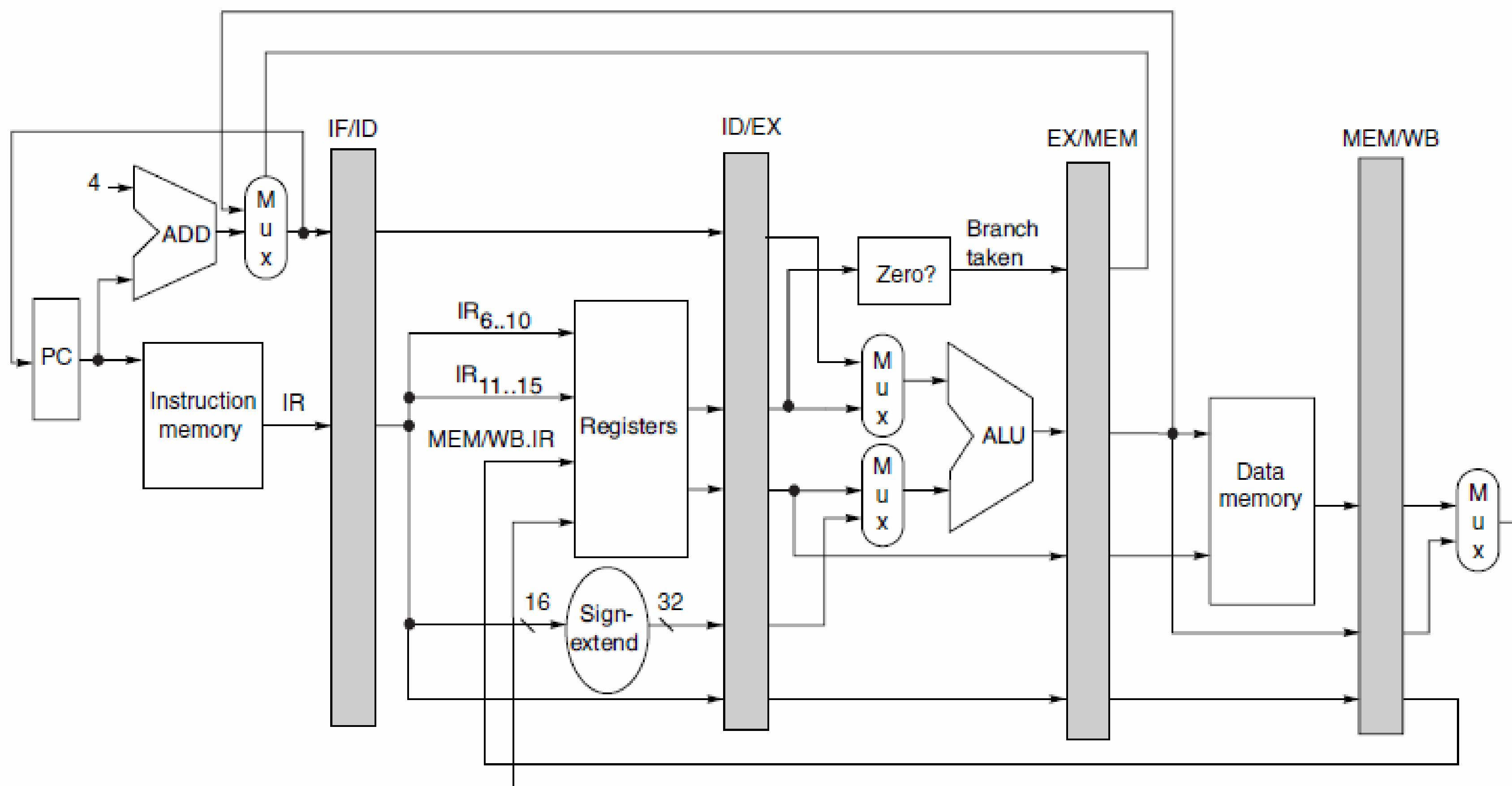


Figure A.18 The data path is pipelined by adding a set of registers, one between each pair of pipe stages. The registers serve to convey values and control information from one stage to the next. We can also think of the PC as a pipeline register, which sits before the IF stage of the pipeline, leading to one pipeline register for each pipe stage. Recall that the PC is an edge-triggered register written at the end of the clock cycle; hence there is no race condition in writing the PC. The selection multiplexer for the PC has been moved so that the PC is written in exactly one stage (IF). If we didn't move it, there would be a conflict when a branch occurred, since two instructions would try to write different values into the PC. Most of the data paths flow from left to right, which is from earlier in time to later. The paths flowing from right to left (which carry the register write-back information and PC information on a branch) introduce complications into our pipeline.

Events of the MIPS Pipeline

Stage	Any instruction		
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM.ALUOutput} else {PC+4});		
ID	ID/EX.A ← Regs[IF/ID.IR[rs]]; ID/EX.B ← Regs[IF/ID.IR[rt]]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field]);		
	ALU instruction	**Load or store instruction**	**Branch instruction**
EX	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B; or EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.Imm;	EX/MEM.IR to ID/EX.IR EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.B ← ID/EX.B;	EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm << 2); EX/MEM.cond ← (ID/EX.A == 0);
MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput] ← EX/MEM.B;	
WB	Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUOutput; or Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD;	

Figure A.19 Events on every pipe stage of the MIPS pipeline. Let's review the actions in the stages that are specific to the pipeline organization. In IF, in addition to fetching the instruction and computing the new PC, we store the incremented PC both into the PC and into a pipeline register (NPC) for later use in computing the branch-target address. This structure is the same as the organization in Figure A.18, where the PC is updated in IF from one of two sources. In ID, we fetch the registers, extend the sign of the lower 16 bits of the IR (the immediate field), and pass along the IR and NPC. During EX, we perform an ALU operation or an address calculation; we pass along the IR and the B register (if the instruction is a store). We also set the value of cond to 1 if the instruction is a taken branch. During the MEM phase, we cycle the memory, write the PC if needed, and pass along values needed in the final pipe stage. Finally, during WB, we update the register field from either the ALU output or the loaded value. For simplicity we always pass the entire IR from one stage to the next, although as an instruction proceeds down the pipeline, less and less of the IR is needed.

Implementing Control in the MIPS Pipeline

- The process of letting an instruction move from the instruction decode stage (ID) into the execution stage (EX) of this pipeline is usually called *instruction issue*;
  - an instruction that has made this step is said to have *issued*.
- For the MIPS integer pipeline, all the data hazards can be checked during the ID phase of the pipeline.
- If a data hazard exists, the instruction is stalled before it is issued.
- Likewise, we can determine what forwarding will be needed during ID and set the appropriate controls then.

Pipeline Hazard Detection

Situation	Example code sequence	Action
No dependence	LD **R1**,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R6,R7	No hazard possible because no dependence exists on R1 in the immediately following three instructions.
Dependence requiring stall	LD **R1**,45(R2) DADD R5,**R1**,R7 DSUB R8,R6,R7 OR R9,R6,R7	Comparators detect the use of R1 in the DADD and stall the DADD (and DSUB and OR) before the DADD begins EX.
Dependence overcome by forwarding	LD **R1**,45(R2) DADD R5,R6,R7 DSUB R8,**R1**,R7 OR R9,R6,R7	Comparators detect use of R1 in DSUB and forward result of load to ALU in time for DSUB to begin EX.
Dependence with accesses in order	LD **R1**,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,**R1**,R7	No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

Figure A.20 Situations that the pipeline hazard detection hardware can see by comparing the destination and sources of adjacent instructions. This table indicates that the only comparison needed is between the destination and the sources on the two instructions following the instruction that wrote the destination. In the case of a stall, the pipeline dependences will look like the third case once execution continues. Of course hazards that involve R0 can be ignored since the register always contains 0, and the test above could be extended to do this.

Outline

- Introduction
- The Major Hurdle of Pipelining—Pipeline Hazards
- How Is Pipelining Implemented?
- What Makes Pipelining Hard to Implement?
- Extending the MIPS Pipeline to Handle Multicycle Operations
- Putting It All Together: The MIPS R4000 Pipeline
- Crosscutting Issues
- Fallacies and Pitfalls

Further Complications

- Now that we understand how to detect and resolve hazards, we can deal with some complications that we have avoided so far.
- The first issue regards the challenges of exceptional situations where the instruction execution order is changed in unexpected ways.

Dealing with Exceptions

- Exceptional situations are harder to handle in a pipelined CPU because the overlapping of instructions makes it more difficult to know whether an instruction can safely change the state of the CPU.
- In a pipelined CPU, an instruction is executed piece by piece and is not completed for several clock cycles.
- *Unfortunately, other instructions in the pipeline can raise exceptions that may force the CPU to abort the instructions in the pipeline before they complete.*
- Before we discuss these problems and their solutions in detail, we need to understand what types of situations can arise and what architectural requirements exist for supporting them.

Types of Exceptions and Requirements

- The terminology used to describe exceptional situations where the normal execution order of instruction is changed varies among CPUs.
  - The terms *interrupt, fault,* and *exception* are used, although not in a consistent fashion. We use the term *exception* to cover all these mechanisms.
- I/O device request
- Invoking an operating system service from a user program
- Tracing instruction execution
- Breakpoint (programmer-requested interrupt)
- Integer arithmetic overflow

Types of Exceptions and Requirements

- FP arithmetic anomaly
- Page fault (not in main memory)
- Misaligned memory accesses (if alignment is required)
- Memory protection violation
- Using an undefined or unimplemented instruction
- Hardware malfunctions
- Power failure

Exceptions in different architectures

Exception event	IBM 360	VAX	Motorola 680x0	Intel 80x86
I/O device request	Input/output interruption	Device interrupt	Exception (level 0...7 autovector)	Vectored interrupt
Invoking the operating system service from a user program	Supervisor call interruption	Exception (change mode supervisor trap)	Exception (unimplemented instruction)— on Macintosh	Interrupt (INT instruction)
Tracing instruction execution	Not applicable	Exception (trace fault)	Exception (trace)	Interrupt (single-step trap)
Breakpoint	Not applicable	Exception (breakpoint fault)	Exception (illegal instruction or breakpoint)	Interrupt (breakpoint trap)
Integer arithmetic overflow or underflow; FP trap	Program interruption (overflow or underflow exception)	Exception (integer overflow trap or floating underflow fault)	Exception (floating-point coprocessor errors)	Interrupt (overflow trap or math unit exception)
Page fault (not in main memory)	Not applicable (only in 370)	Exception (translation not valid fault)	Exception (memory-management unit errors)	Interrupt (page fault)
Misaligned memory accesses	Program interruption (specification exception)	Not applicable	Exception (address error)	Not applicable
Memory protection violations	Program interruption (protection exception)	Exception (access control violation fault)	Exception (bus error)	Interrupt (protection exception)
Using undefined instructions	Program interruption (operation exception)	Exception (opcode privileged/reserved fault)	Exception (illegal instruction or break-point/unimplemented instruction)	Interrupt (invalid opcode)
Hardware malfunctions	Machine-check interruption	Exception (machine-check abort)	Exception (bus error)	Not applicable
Power failure	Machine-check interruption	Urgent interrupt	Not applicable	Nonmaskable interrupt

Figure A.26 The names of common exceptions vary across four different architectures. Every event on the IBM 360 and 80x86 is called an *interrupt*, while every event on the 680x0 is called an *exception*. VAX divides events into *interrupts* or *exceptions*. Adjectives *device*, *software*, and *urgent* are used with VAX interrupts, while VAX exceptions are subdivided into *faults*, *traps*, and *aborts*.

Requirements on exceptions

- The requirements on exceptions can be characterized on five semiindependent axes:
  - *Synchronous versus asynchronous*
  - *User requested versus coerced*
  - *User maskable versus user nonmaskable*
  - *Within versus between instructions*
  - *Resume versus terminate*

Synchronous versus asynchronous

- If the event occurs at the same place every time the program is executed with the same data and memory allocation, the event is *synchronous*.
- With the exception of hardware malfunctions, *asynchronous* events are caused by devices external to the CPU and memory.
- Asynchronous events usually can be handled after the completion of the current instruction, which makes them easier to handle.

User requested versus coerced

- If the user task directly asks for it, it is a *user-requested* event. In some sense, user-requested exceptions are not really exceptions, since they are predictable.
  - They are treated as exceptions, however, because the same mechanisms that are used to save and restore the state are used for these user-requested events.
- Because the only function of an instruction that triggers this exception is to cause the exception, user-requested exceptions can always be handled after the instruction has completed.
- *Coerced* exceptions are caused by some hardware event that is not under the control of the user program. Coerced exceptions are harder to implement because they are not predictable.

User maskable versus user nonmaskable

- If an event can be masked or disabled by a user task, it is *user maskable*.
- This mask simply controls whether the hardware responds to the exception or not.

Within versus between instructions

- This classification depends on whether the event prevents instruction completion by occurring in the middle of execution no matter how short—or whether it is recognized *between* instructions.
- Exceptions that occur *within* instructions are usually synchronous, since the instruction triggers the exception.
- It's harder to implement exceptions that occur within instructions than those between instructions, since the instruction must be stopped and restarted.
- Asynchronous exceptions that occur within instructions arise from catastrophic situations (e.g., hardware malfunction) and always cause program termination.

Resume versus terminate

- If the program's execution always stops after the interrupt, it is a *terminating* event.
- If the program's execution continues after the interrupt, it is a *resuming* event.
- It is easier to implement exceptions that terminate execution, since the CPU need not be able to restart execution of the same program after handling the exception.

Categories of actions for exceptions

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

Figure A.27 Five categories are used to define what actions are needed for the different exception types shown in Figure A.26. Exceptions that must allow resumption are marked as resume, although the software may often choose to terminate the program. Synchronous, coerced exceptions occurring within instructions that can be resumed are the most difficult to implement. We might expect that memory protection access violations would always result in termination; however, modern operating systems use memory protection to detect events such as the first attempt to use a page or the first write to a page. Thus, CPUs should be able to resume after such exceptions.

Restartable Pipelines

- The difficult task is implementing interrupts occurring within instructions where the instruction must be resumed.
- Implementing such exceptions requires that another program must be invoked to:
  - save the state of the executing program,
  - correct the cause of the exception, and
  - then restore the state of the program before the instruction that caused the exception can be tried again.
- This process must be effectively invisible to the executing program.
- If a pipeline provides the ability for the processor to handle the exception, save the state, and restart without affecting the execution of the program, the pipeline or processor is said to be *restartable*.
- While early supercomputers and microprocessors often lacked this property, almost all processors today support it, at least for the integer pipeline, because it is needed to implement virtual memory

Stopping and Restarting Execution

- As in unpipelined implementations, the most difficult exceptions have two properties:

(1) they occur within instructions (that is, in the middle of the instruction execution corresponding to EX or MEM pipe stages), and

(2) they must be restartable.

Stopping and Restarting Execution

- When an exception occurs, the pipeline control can take the following steps to save the pipeline state safely:
  1. Force a trap instruction into the pipeline on the next IF.
  2. Until the trap is taken, turn off all writes for the faulting instruction and for all instructions that follow in the pipeline.
  3. After the exception-handling routine in the operating system receives control it immediately saves the PC of the faulting instruction. This value will be used to return from the exception later.

Precise Exceptions

- After the exception has been handled, special instructions return the processor from the exception by reloading the PCs and restarting the instruction stream (using the instruction RFE in MIPS).
- If the pipeline can be stopped so that the instructions just before the faulting instruction are completed and those after it can be restarted from scratch, the pipeline is said to have *precise exceptions.*

Exceptions in the MIPS pipeline

Pipeline stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

Figure A.28 Exceptions that may occur in the MIPS pipeline. Exceptions raised from instruction or data memory access account for six out of eight cases.

Multiple Exceptions

- With pipelining, *multiple exceptions* may occur in the same clock cycle because there are multiple instructions in execution. For example, consider this instruction sequence:

LD	IF	ID	EX	MEM	WB	
DADD		IF	ID	EX	MEM	WB

- This pair of instructions can cause a data page fault and an arithmetic exception at the same time, since the LD is in the MEM stage while the DADD is in the EX stage.
- This case can be handled by dealing with only the data page fault and then restarting the execution.
- The second exception will reoccur (but not the first, if the software is correct), and when the second exception occurs, it can be handled independently.

Outline

- Introduction
- The Major Hurdle of Pipelining—Pipeline Hazards
- How Is Pipelining Implemented?
- What Makes Pipelining Hard to Implement?
- Extending the MIPS Pipeline to Handle Multicycle Operations
- Putting It All Together: The MIPS R4000 Pipeline
- Crosscutting Issues
- Fallacies and Pitfalls

Multicycle operations

- It is impractical to require that all MIPS floating-point operations complete in 1 clock cycle, or even in 2.
- Doing so would mean accepting a slow clock, or using enormous amounts of logic in the floating-point units, or both.
- Instead, the floating-point pipeline will allow for a longer latency for operations.
- This is easier to grasp if we imagine the floating-point instructions as having the same pipeline as the integer instructions, with two important changes.
  - First, the EX cycle may be repeated as many times as needed to complete the operation—the number of repetitions can vary for different operations.
  - Second, there may be multiple floating-point functional units.
- A stall will occur if the instruction to be issued will either cause a structural hazard for the functional unit it uses or cause a data hazard.

FP functional units

- Let's assume that there are four separate functional units in our MIPS implementation:
  1. The main integer unit that handles loads and stores, integer ALU operations, and branches
  2. FP and integer multiplier
  3. FP adder that handles FP add, subtract, and conversion
  4. FP and integer divider
- We also assume that the execution stages of these functional units are not pipelined.
  – Because EX is not pipelined, no other instruction using that functional unit may issue until the previous instruction leaves EX.
  – Moreover, if an instruction cannot proceed to the EX stage, the entire pipeline behind that instruction will be stalled.

Unpipelined functional units in MIPS

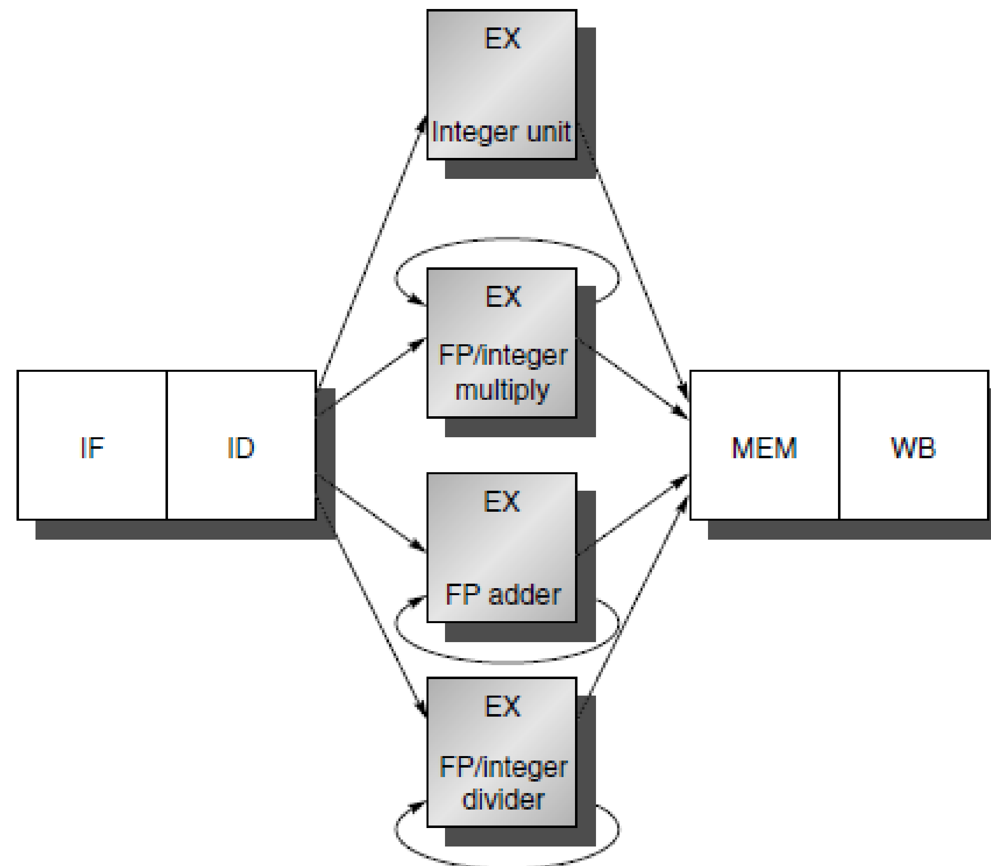


Figure A.29 The MIPS pipeline with three additional unpipelined, floating-point, functional units. Because only one instruction issues on every clock cycle, all instructions go through the standard pipeline for integer operations. The floating-point operations simply loop when they reach the EX stage. After they have finished the EX stage, they proceed to MEM and WB to complete execution.

Latency or repeat interval

- We can generalize the structure of the FP pipeline to allow pipelining of some stages and multiple ongoing operations.
- To describe such a pipeline, we must define both the latency of the functional units and also the *initiation interval* or *repeat interval.*
  - We define *latency* the same way we defined it earlier: the number of intervening cycles between an instruction that produces a result and an instruction that uses the result.
  - The *initiation or repeat interval* is the number of cycles that must elapse between issuing two operations of a given type.

Latency

- With this definition of latency, integer ALU operations have a latency of 0, since the results can be used on the next clock cycle, and loads have a latency of 1, since their results can be used after one intervening cycle.

- Since most operations consume their operands at the beginning of EX, the latency is usually the number of stages after EX that an instruction produces a result — for example, zero stages for ALU operations and one stage for loads.

Latencies and initiation intervals for functional units.

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Figure A.30 Latencies and initiation intervals for functional units.

Pipeline with multiple FP operations

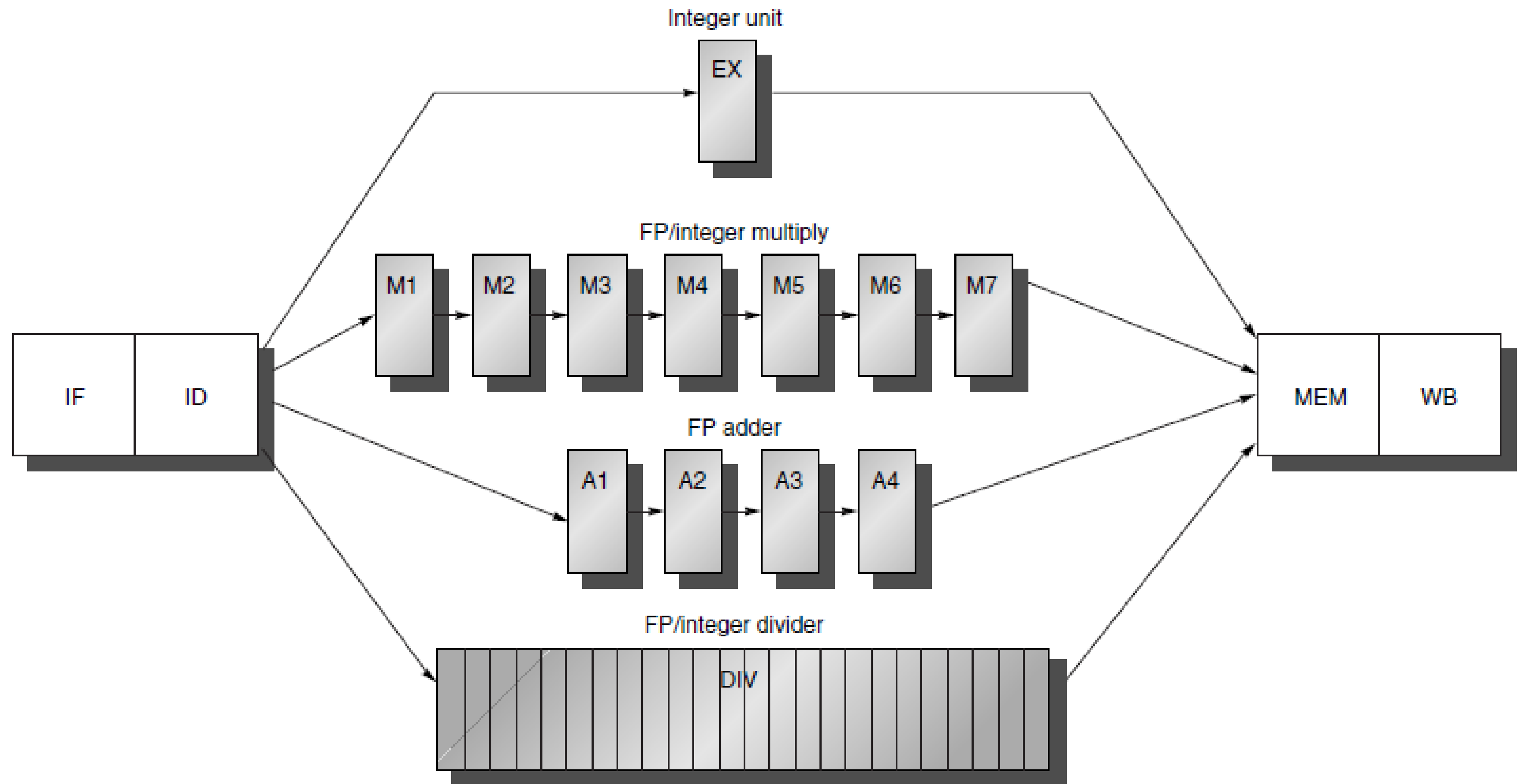


Figure A.31 A pipeline that supports multiple outstanding FP operations. The FP multiplier and adder are fully pipelined and have a depth of seven and four stages, respectively. The FP divider is not pipelined, but requires 24 clock cycles to complete. The latency in instructions between the issue of an FP operation and the use of the result of that operation without incurring a RAW stall is determined by the number of cycles spent in the execution stages. For example, the fourth instruction after an FP add can use the result of the FP add. For integer ALU operations, the depth of the execution pipeline is always one and the next instruction can use the results.

Pipeline Timing for independent FP operations

`MUL.D`	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
`ADD.D`		IF	ID	*A1*	A2	A3	**A4**	MEM	WB		
`L.D`			IF	ID	*EX*	**MEM**	WB				
`S.D`				IF	ID	*EX*	*MEM*	WB			

Figure A.32 The pipeline timing of a set of independent FP operations. The stages in italics show where data are needed, while the stages in bold show where a result is available. The "`.D`" extension on the instruction mnemonic indicates double-precision (64-bit) floating-point operations. FP loads and stores use a 64-bit path to memory so that the pipelining timing is just like an integer load or store.

Performance of a MIPS FP Pipeline

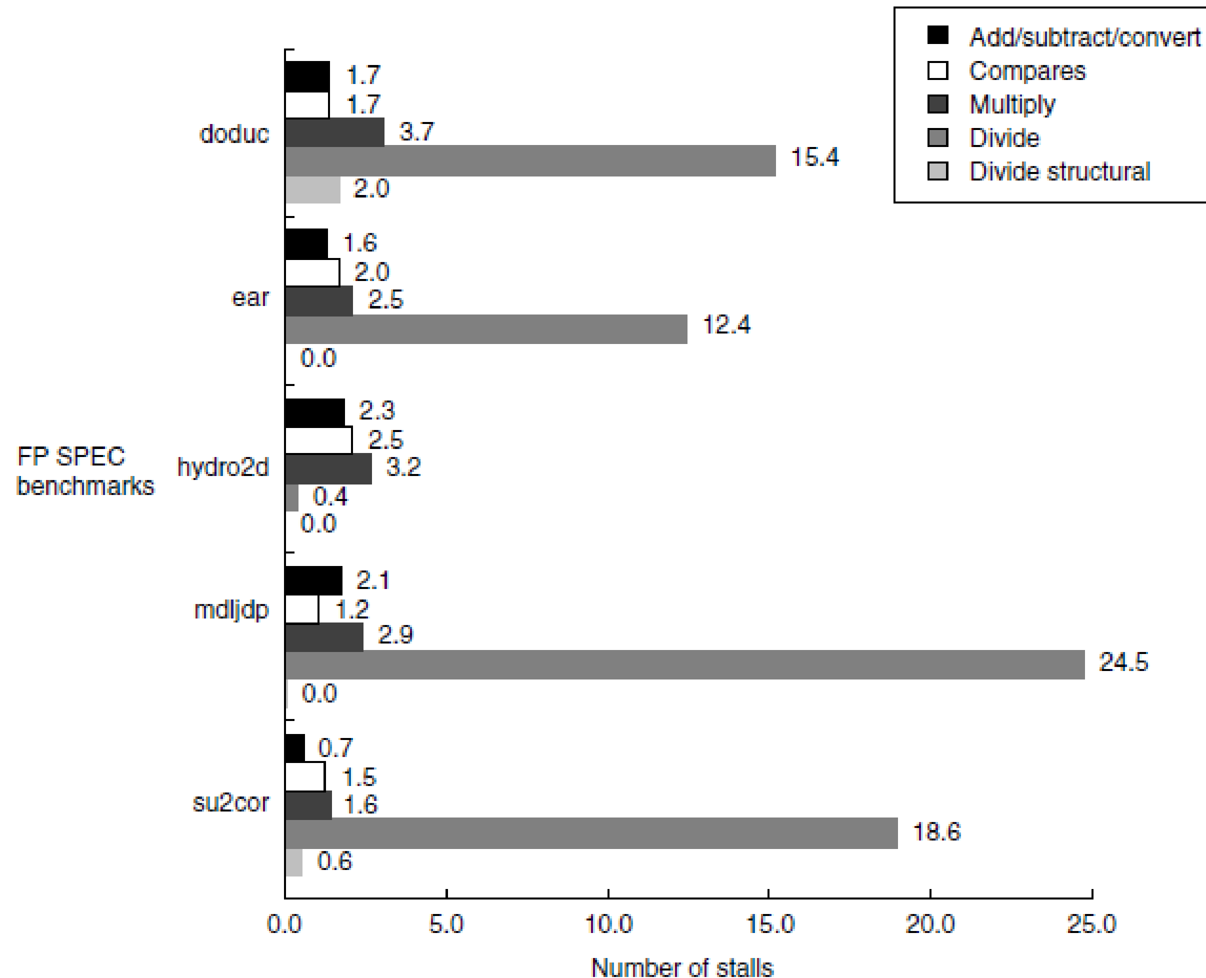


Figure A.35 Stalls per FP operation for each major type of FP operation for the SPEC89 FP benchmarks. Except for the divide structural hazards, these data do not depend on the frequency of an operation, only on its latency and the number of cycles before the result is used. The number of stalls from RAW hazards roughly tracks the latency of the FP unit. For example, the average number of stalls per FP add, subtract, or convert is 1.7 cycles, or 56% of the latency (3 cycles). Likewise, the average number of stalls for multiplies and divides are 2.8 and 14.2, respectively, or 46% and 59% of the corresponding latency. Structural hazards for divides are rare, since the divide frequency is low.

Outline

- Introduction
- The Major Hurdle of Pipelining—Pipeline Hazards
- How Is Pipelining Implemented?
- What Makes Pipelining Hard to Implement?
- Extending the MIPS Pipeline to Handle Multicycle Operations
- Putting It All Together: The MIPS R4000 Pipeline
- Crosscutting Issues
- Fallacies and Pitfalls

Superpipelining

- The R4000 implements MIPS64 but uses a deeper pipeline than that of our five-stage design both for integer and FP programs.
- This deeper pipeline allows it to achieve higher clock rates by decomposing the five-stage integer pipeline into eight stages.
- Because cache access is particularly time critical, the extra pipeline stages come from decomposing the memory access.
- This type of deeper pipelining is sometimes called *superpipelining*.

The eight-stage pipeline of R4000

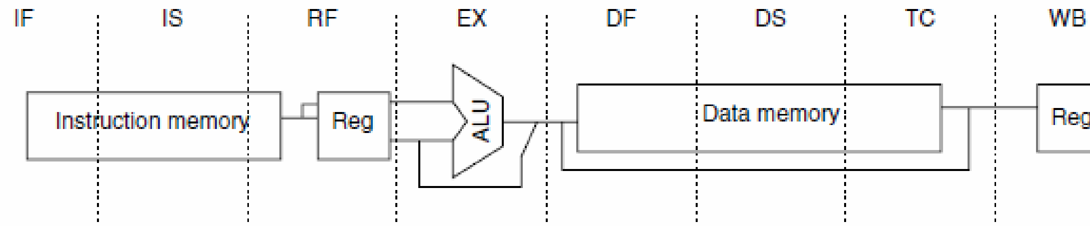


Figure A.37 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction is actually available at the end of IS, but the tag check is done in RF, while the registers are fetched. Thus, we show the instruction memory as operating through RF. The TC stage is needed for data memory access, since we cannot write the data into the register until we know whether the cache access was a hit or not.

IF—First half of instruction fetch; PC selection actually happens here, together with initiation of instruction cache access.
IS—Second half of instruction fetch, complete instruction cache access.
RF—Instruction decode and register fetch, hazard checking, and also instruction cache hit detection.
EX—Execution, which includes effective address calculation, ALU operation, and branch-target computation and condition evaluation.
DF—Data fetch, first half of data cache access.
DS—Second half of data fetch, completion of data cache access.
TC—Tag check, determine whether the data cache access hit.
WB—Write back for loads and register-register operations.

R4000 Integer Pipeline

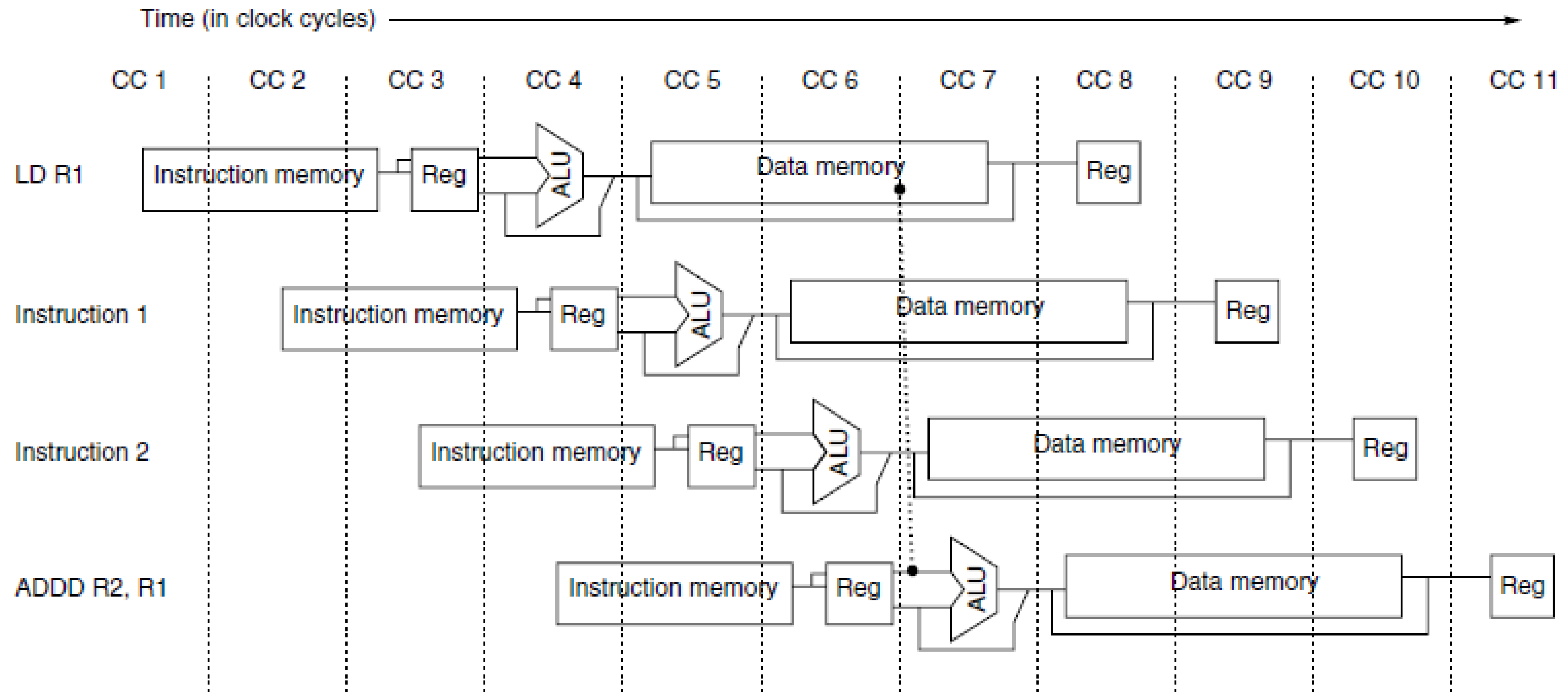


Figure A.38 The structure of the R4000 integer pipeline leads to a 2-cycle load delay. A 2-cycle delay is possible because the data value is available at the end of DS and can be bypassed. If the tag check in TC indicates a miss, the pipeline is backed up a cycle, when the correct data are available.

R4000 Integer Pipeline

Instruction number	Clock number 1	2	3	4	5	6	7	8	9
LD R1,...	IF	IS	RF	EX	DF	DS	TC	WB	
DADD R2,R1,...		IF	IS	RF	stall	stall	EX	DF	DS
DSUB R3,R1,...			IF	IS	stall	stall	RF	EX	DF
OR R4,R1,...				IF	stall	stall	IS	RF	EX

Figure A.39 A load instruction followed by an immediate use results in a 2-cycle stall. Normal forwarding paths can be used after two cycles, so the DADD and DSUB get the value by forwarding after the stall. The OR instruction gets the value from the register file. Since the two instructions after the load could be independent and hence not stall, the bypass can be to instructions that are 3 or 4 cycles after the load.

Performance of the R4000 Pipeline

- We examine here the stalls that occur for the SPEC92 benchmarks when running on the R4000 pipeline structure.
- There are four major causes of pipeline stalls or losses:
  1. *Load stalls* — Delays arising from the use of a load result 1 or 2 cycles after the load.
  2. *Branch stalls* — 2-cycle stall on every taken branch plus unfilled or canceled branch delay slots
  3. *FP result stalls* — Stalls because of RAW hazards for an FP operand
  4. *FP structural stalls* — Delays because of conflicts for functional units in the FP pipeline

The pipeline CPI for 10 of the SPEC92 benchmarks

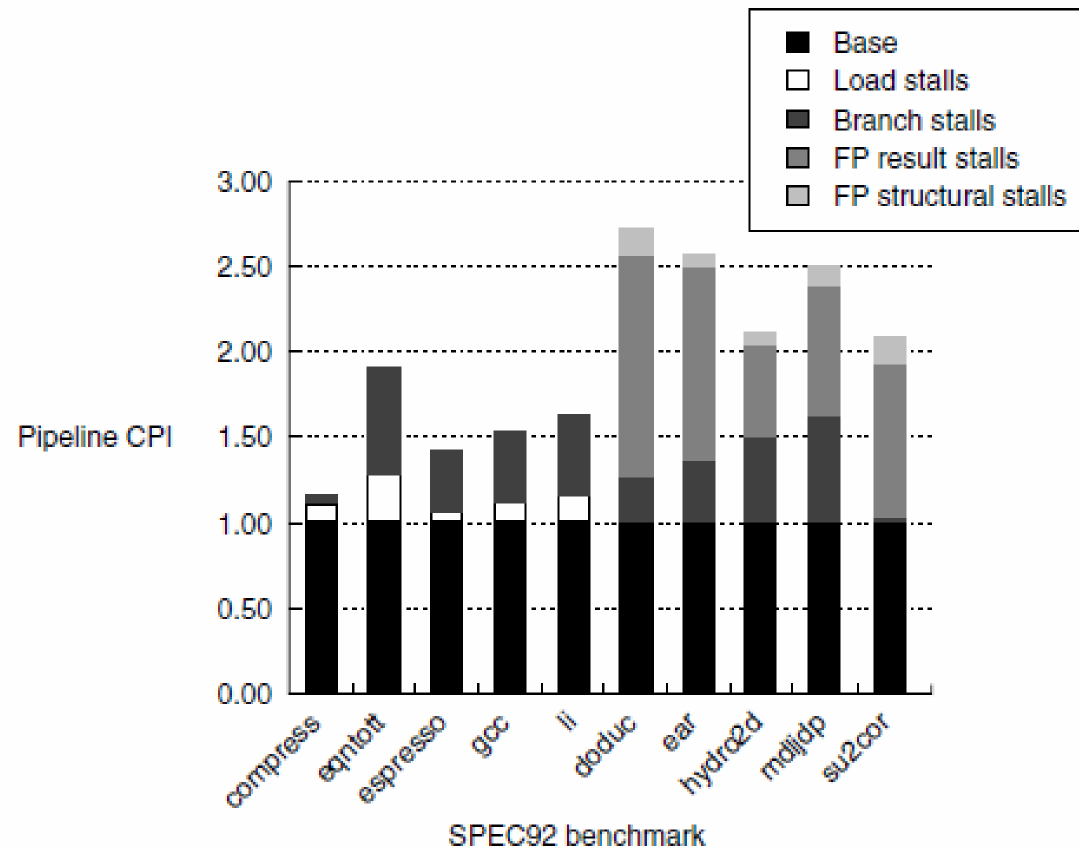


Figure A.48 The pipeline CPI for 10 of the SPEC92 benchmarks, assuming a perfect cache. The pipeline CPI varies from 1.2 to 2.8. The leftmost five programs are integer programs, and branch delays are the major CPI contributor for these. The rightmost five programs are FP, and FP result stalls are the major contributor for these. Figure A.49 shows the numbers used to construct this plot.

The total pipeline CPI

Benchmark	Pipeline CPI	Load stalls	Branch stalls	FP result stalls	FP structural stalls
compress	1.20	0.14	0.06	0.00	0.00
eqntott	1.88	0.27	0.61	0.00	0.00
espresso	1.42	0.07	0.35	0.00	0.00
gcc	1.56	0.13	0.43	0.00	0.00
li	1.64	0.18	0.46	0.00	0.00
Integer average	1.54	0.16	0.38	0.00	0.00
doduc	2.84	0.01	0.22	1.39	0.22
mdljdp2	2.66	0.01	0.31	1.20	0.15
ear	2.17	0.00	0.46	0.59	0.12
hydro2d	2.53	0.00	0.62	0.75	0.17
su2cor	2.18	0.02	0.07	0.84	0.26
FP average	2.48	0.01	0.33	0.95	0.18
Overall average	2.00	0.10	0.36	0.46	0.09

Figure A.49 The total pipeline CPI and the contributions of the four major sources of stalls are shown. The major contributors are FP result stalls (both for branches and for FP inputs) and branch stalls, with loads and FP structural stalls adding less.

Performance observations

- From the data, we can see the penalty of the deeper pipelining.
  - The R4000's pipeline has much longer branch delays than the classic five-stage pipeline.
  - The longer branch delay substantially increases the cycles spent on branches, especially for the integer programs with a higher branch frequency.
- An interesting effect for the FP programs is that the latency of the FP functional units leads to more result stalls than the structural hazards, which arise both from the initiation interval limitations and from conflicts for functional units from different FP instructions.
  - Thus, reducing the latency of FP operations should be the first target, rather than more pipelining or replication of the functional units.

Outline

- Introduction
- The Major Hurdle of Pipelining—Pipeline Hazards
- How Is Pipelining Implemented?
- What Makes Pipelining Hard to Implement?
- Extending the MIPS Pipeline to Handle Multicycle Operations
- Putting It All Together: The MIPS R4000 Pipeline
- Crosscutting Issues
- Fallacies and Pitfalls

RISC Instruction Sets and Efficiency of Pipelining

- We have already discussed the advantages of instruction set simplicity in building pipelines.
- Simple instruction sets offer another advantage:

  They make it easier to schedule code to achieve efficiency of execution in a pipeline.
- To see this, consider a simple example: Suppose we need to add two values in memory and store the result back to memory.
- In some sophisticated instruction sets this will take only a single instruction; in others it will take two or three.
- A typical RISC architecture would require four instructions (two loads, an add, and a store).
  - These instructions cannot be scheduled sequentially in most pipelines without intervening stalls.

RISC Instruction Sets and Efficiency of Pipelining

- With a RISC instruction set, the individual operations are separate instructions and may be individually scheduled either by the compiler or by using dynamic hardware scheduling techniques (which we discuss next)
- These efficiency advantages, coupled with the greater ease of implementation, appear to be so significant that almost all recent pipelined implementations of complex instruction sets actually translate their complex instructions into simple RISC-like operations, and then schedule and pipeline those operations.
  - During ILP lecture (Chapter 2) we will show that both the Pentium III and Pentium 4 use this approach.

Static scheduling approach

- Simple pipelines fetch an instruction and issue it, unless there is a data dependence between an instruction already in the pipeline and the fetched instruction that cannot be hidden with bypassing or forwarding.
- Forwarding logic reduces the effective pipeline latency so that certain dependences do not result in hazards.
- If there is an unavoidable hazard, then the hazard detection hardware stalls the pipeline (starting with the instruction that uses the result).
  - No new instructions are fetched or issued until the dependence is cleared.
  - To overcome these performance losses, the compiler can attempt to schedule instructions to avoid the hazard; this approach is called *compiler* or *static scheduling*.

In-order instruction issue

- Several early processors used another approach, called *dynamic scheduling*, whereby the hardware rearranges the instruction execution to reduce the stalls.
- Here we explain a simpler introduction to dynamic scheduling by explaining the scoreboarding technique of the CDC 6600.
- All the techniques discussed so far use in-order instruction issue, which means that if an instruction is stalled in the pipeline, no later instructions can proceed.
  - With in-order issue, if two instructions have a hazard between them, the pipeline will stall, even if there are later instructions that are independent and would not stall.

Out-of-order execution

- In the MIPS pipeline developed earlier, both structural and data hazards were checked during instruction decode (ID):
  - When an instruction could execute properly, it was issued from ID.
- To allow an instruction to begin execution as soon as its operands are available, even if a predecessor is stalled, we must separate the issue process into two parts:
  - checking the structural hazards and
  - waiting for the absence of a data hazard.
- We decode and issue instructions in order. However, we want the instructions to begin execution as soon as their data operands are available.
- Thus, the pipeline will do *out-of-order execution,* which implies *out-of order completion*.

Implementing out-of-order execution

- To implement out-of-order execution, we must split the ID pipe stage into two stages:
  1. *Issue*—Decode instructions, check for structural hazards.
  2. *Read operands*—Wait until no data hazards, then read operands.

Implementing out-of-order execution

- The IF stage proceeds the issue stage, and the EX stage follows the read operands stage, just as in the MIPS pipeline.
- As in the MIPS floating-point pipeline, execution may take *multiple cycles*, depending on the operation.
- Thus, we may need to distinguish when an instruction *begins execution* and when it *completes execution;* between the two times, the instruction is *in execution*.
  - This allows multiple instructions to be in execution at the same time.
- In addition to these changes to the pipeline structure, we will also change the functional unit design by varying the number of units, the latency of operations, and the functional unit pipelining, so as to better explore these more advanced pipelining techniques.

Dynamic Scheduling with a Scoreboard

- In a dynamically scheduled pipeline, all instructions pass through the issue stage in order (in-order issue);
  - *However, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order*.
- *Scoreboarding* is a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependences; it is named after the CDC 6600 scoreboard, which developed this capability.

Data Hazards (Chap.2)

- Consider two instructions *i* and *j,* with *i* preceding *j* in program order. The possible data hazards are:
- RAW *(read after write)* — *j* tries to read a source before *i* writes it, so *j* incorrectly gets the *old* value. This hazard is the most common type and corresponds to a true data dependence. Program order must be preserved to ensure that *j* receives the value from *i*.
- WAW *(write after write)* — *j* tries to write an operand before it is written by *i*. The writes end up being performed in the wrong order, leaving the value written by *i* rather than the value written by *j* in the destination.
  - This hazard corresponds to an output dependence. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

Data Hazards (Chap.2)

- WAR *(write after read)* — *j* tries to write a destination before it is read by *i*, so *i* incorrectly gets the *new* value.
- WAR hazards cannot occur in most static issue pipelines — even deeper pipelines or floating-point pipelines — because all reads are early (in ID) and all writes are late (in WB).
- A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline *and* other instructions that read a source late in the pipeline, or when instructions are reordered => our case here.

Antidependence (Chap.2)

- An *antidependence* between instruction *i* and instruction *j* occurs when instruction *j* writes a register or memory location that instruction *i* reads.
- The original ordering must be preserved to ensure that *i* reads the correct value.

Output dependence (Chap.2)

- An *output dependence* occurs when instruction *i* and instruction *j* write the same register or memory location.
- The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction *j*.

WAR Hazards in out-of-order execution

- It is important to observe that WAR hazards, which did not exist in the MIPS floating-point or integer pipelines, may arise when instructions execute out of order.
- For example, consider the following code sequence:

```
DIV.D     F0,F2,F4
ADD.D     F10,F0,F8
SUB.D     F8,F8,F14
```

- There is an antidependence between the ADD.D and the SUB.D: If the pipeline executes the SUB.D before the ADD.D, it will violate the antidependence, yielding incorrect execution.
- Likewise, to avoid violating output dependences, WAW hazards (e.g., as would occur if the destination of the SUB.D were F10) must also be detected.
- As we will see, both these hazards are avoided in a scoreboard by stalling the later instruction involved in the antidependence.

Scoreboard

- The goal of a scoreboard is to maintain an execution rate of one instruction per clock cycle (when there are no structural hazards) by executing an instruction as early as possible.

  Thus, when the next instruction to execute is stalled, other instructions can be issued and executed if they do not depend on any active or stalled instruction.
- The scoreboard takes full responsibility for instruction issue and execution, including all hazard detection.
- Taking advantage of out-of-order execution requires multiple instructions to be in their EX stage simultaneously.
- This can be achieved with multiple functional units, with pipelined functional units, or with both.
- Since these two capabilities — pipelined functional units and multiple functional units — are essentially equivalent for the purposes of pipeline control, we will assume the processor has multiple functional units.

Scoreboard structure

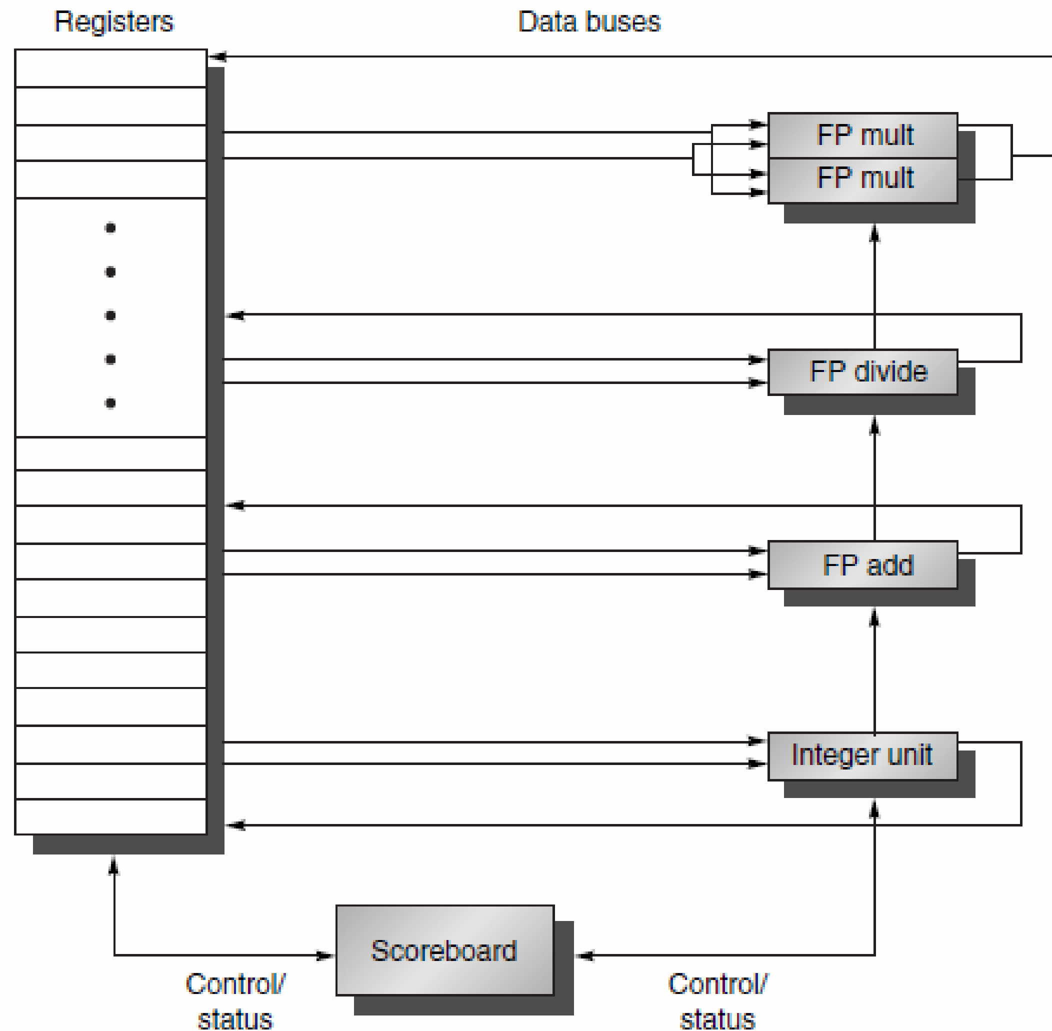


Figure A.50 **The basic structure of a MIPS processor with a scoreboard.** The scoreboard's function is to control instruction execution (vertical control lines). All data flows between the register file and the functional units over the buses (the horizontal lines, called trunks in the CDC 6600). There are two FP multipliers, an FP divider, an FP adder, and an integer unit. One set of buses (two inputs and one output) serves a group of functional units. The details of the scoreboard are shown in Figures A.51–A.54.

Scoreboard in action

- Every instruction goes through the scoreboard, where a record of the data dependences is constructed;
  - This step corresponds to instruction issue and replaces part of the ID step in the MIPS pipeline.
- The scoreboard then determines when the instruction can read its operands and begin execution.
- If the scoreboard decides the instruction cannot execute immediately, it monitors every change in the hardware and decides when the instruction *can* execute.
- The scoreboard also controls when an instruction can write its result into the destination register.
- Thus, all hazard detection and resolution is centralized in the scoreboard.

Costs and benefits of scoreboards

- The costs and benefits of *scoreboarding* are interesting considerations.
- The CDC 6600 designers measured a performance improvement of 1.7 for FORTRAN programs and 2.5 for hand-coded assembly language.
- However, this was measured in the days before software pipeline scheduling, semiconductor main memory, and caches (which lower memory access time).
- The scoreboard on the CDC 6600 had about as much logic as one of the functional units, which is surprisingly low.
- The main cost was in the large number of buses—about four times as many as would be required if the CPU only executed instructions in order (or if it only initiated one instruction per execute cycle).

Limiting Factors for scoreboards

- In eliminating stalls, a scoreboard is limited by several factors:

1. *The amount of parallelism available among the instructions* — This determines whether independent instructions can be found to execute.
   - If each instruction depends on its predecessor, no dynamic scheduling scheme can reduce stalls.
   - If the instructions in the pipeline simultaneously must be chosen from the same basic block (as was true in the 6600), this limit is likely to be quite severe.

Limiting Factors for scoreboards

2. *The number of scoreboard entries* — This determines how far ahead the pipeline can look for independent instructions.
   - The set of instructions examined as candidates for potential execution is called the *window*. The size of the scoreboard determines the size of the window.

3. *The number and types of functional units* — This determines the importance of structural hazards, which can increase when dynamic scheduling is used.

4. *The presence of antidependences and output dependences* — These lead to WAR and WAW stalls.

Improvements of scoreboards

- Next time (Chap.2 and Chap.3): Techniques that attack the problem of exposing and better utilizing available ILP.
- The second and third factors can be attacked by:
  - increasing the size of the scoreboard and the number of functional units;
    - However, these changes have cost implications and may also affect cycle time.
- WAW and WAR hazards become more important in dynamically scheduled processors because the pipeline exposes more name dependences.

Outline

- Introduction
- The Major Hurdle of Pipelining—Pipeline Hazards
- How Is Pipelining Implemented?
- What Makes Pipelining Hard to Implement?
- Extending the MIPS Pipeline to Handle Multicycle Operations
- Putting It All Together: The MIPS R4000 Pipeline
- Crosscutting Issues
- Fallacies and Pitfalls

Pitfall: Unexpected execution sequences may cause unexpected hazards

- At first glance, WAW hazards look like they should never occur in a code sequence because no compiler would ever generate two writes to the same register without an intervening read. But they can occur when the sequence is unexpected.

Pitfall: Extensive pipelining can impact other aspects of a design, leading to overall worse cost-performance.

- The best example of this phenomenon comes from two implementations of the VAX, the 8600 and the 8700.
- When the 8600 was initially delivered, it had a cycle time of 80 ns.
- Subsequently, a redesigned version, called the 8650, with a 55 ns clock was introduced.
- The 8700 has a much simpler pipeline that operates at the microinstruction level, yielding a smaller CPU with a faster clock cycle of 45 ns.
- The overall outcome is that the 8650 has a CPI advantage of about 20%, but the 8700 has a clock rate that is about 20% faster.
- Thus, the 8700 achieves the same performance with much less hardware.

Pitfall: Evaluating dynamic or static scheduling on the basis of unoptimized code.

- Unoptimized code — containing redundant loads, stores, and other operations that might be eliminated by an optimizer— is much easier to schedule than "tight" optimized code.
- Of course, the optimized program is much faster, since it has fewer instructions.
- To fairly evaluate a compile time scheduler or run time dynamic scheduling, you must use optimized code, since in the real system you will derive good performance from other optimizations in addition to scheduling.

End of Lecture 2

- Readings
  - Book: Appendix A