Computer Architecture

Appendix C - Pipelining: Basic Concepts

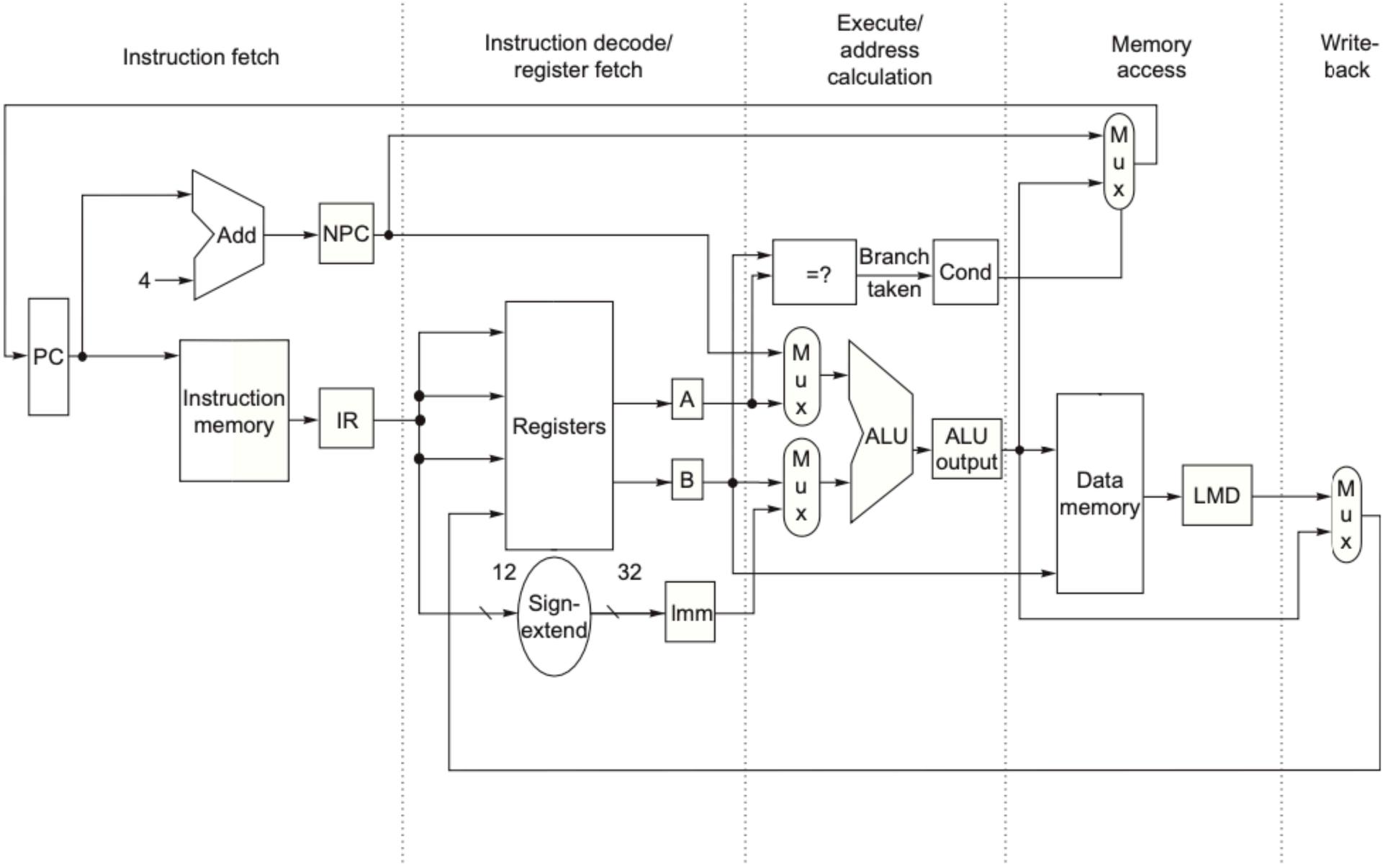


Figure C.18 The implementation of the RISC V data path allows every instruction to be executed in 4 or 5 clock

RISC-V Data Path

1. *Instruction fetch cycle* (IF):

```
IR ← Mem[PC];
NPC ← PC + 4;
```

Operation—Send out the PC and fetch the instruction from memory into the instruction register (IR); increment the PC by 4 to address the next sequential instruction. The IR is used to hold the instruction that will be needed on subsequent clock cycles; likewise, the register NPC is used to hold the next sequential PC.

RISC-V Data Path

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

```
A ← Regs[rs1];
B ← Regs[rs2];
Imm ← sign-extended immediate field of IR;
```

Operation—Decode the instruction and access the register file to read the registers (rs1 and rs2 are the register specifiers). The outputs of the general-purpose registers are read into two temporary registers (A and B) for use in later clock cycles. The lower 16 bits of the IR are also sign extended and stored into the temporary register Imm, for use in the next cycle.

RISC-V Data Path

3. *Execution/effective address cycle* (EX):

   The ALU operates on the operands prepared in the prior cycle, performing one of four functions depending on the RISC V instruction type:

   - Memory reference:

     `ALUOutput ← A + Imm;`

     Operation—The ALU adds the operands to form the effective address and places the result into the register ALUOutput.

   - Register-register ALU instruction:

     `ALUOutput ← A` *`func`* `B;`

     Operation—The ALU performs the operation specified by the function code (a combination of the func3 and func7 fields) on the value in register A and on the value in register B. The result is placed in the temporary register ALUOutput.

RISC-V Data Path

- Register-Immediate ALU instruction:

```
ALUOutput ← A op Imm;
```

Operation—The ALU performs the operation specified by the opcode on the value in register A and on the value in register Imm. The result is placed in the temporary register ALUOutput.

- Branch:

```
ALUOutput ← NPC + (Imm << 2);
Cond ← (A == B)
```

Operation—The ALU adds the NPC to the sign-extended immediate value in Imm, which is shifted left by 2 bits to create a word offset, to compute the address of the branch target. Register A, which has been read in the prior cycle, is checked to determine whether the branch is taken, by comparison with Register B, because we consider only branch equal.

RISC-V Data Path

4. *Memory access/branch completion cycle* (MEM):

The PC is updated for all instructions: `PC ← NPC;`

- Memory reference:

```
LMD ← Mem[ALUOutput] or
Mem[ALUOutput] ← B;
```

Operation—Access memory if needed. If the instruction is a load, data return from memory and are placed in the LMD (load memory data) register; if it is a store, then the data from the B register are written into memory. In either case, the address used is the one computed during the prior cycle and stored in the register ALUOutput.

- Branch:

```
if (cond) PC ← ALUOutput
```

Operation—If the instruction branches, the PC is replaced with the branch destination address in the register ALUOutput.

RISC-V Data Path

5. *Write-back cycle* (WB):

   - Register-register or Register-immediate ALU instruction:

     `Regs[rd] ← ALUOutput;`

   - Load instruction:

     `Regs[rd] ← LMD;`

   Operation—Write the result into the register file, whether it comes from the memory system (which is in LMD) or from the ALU (which is in ALUOutput) with rd designating the register.

RISC-V Pipeline w/Pipeline Registers

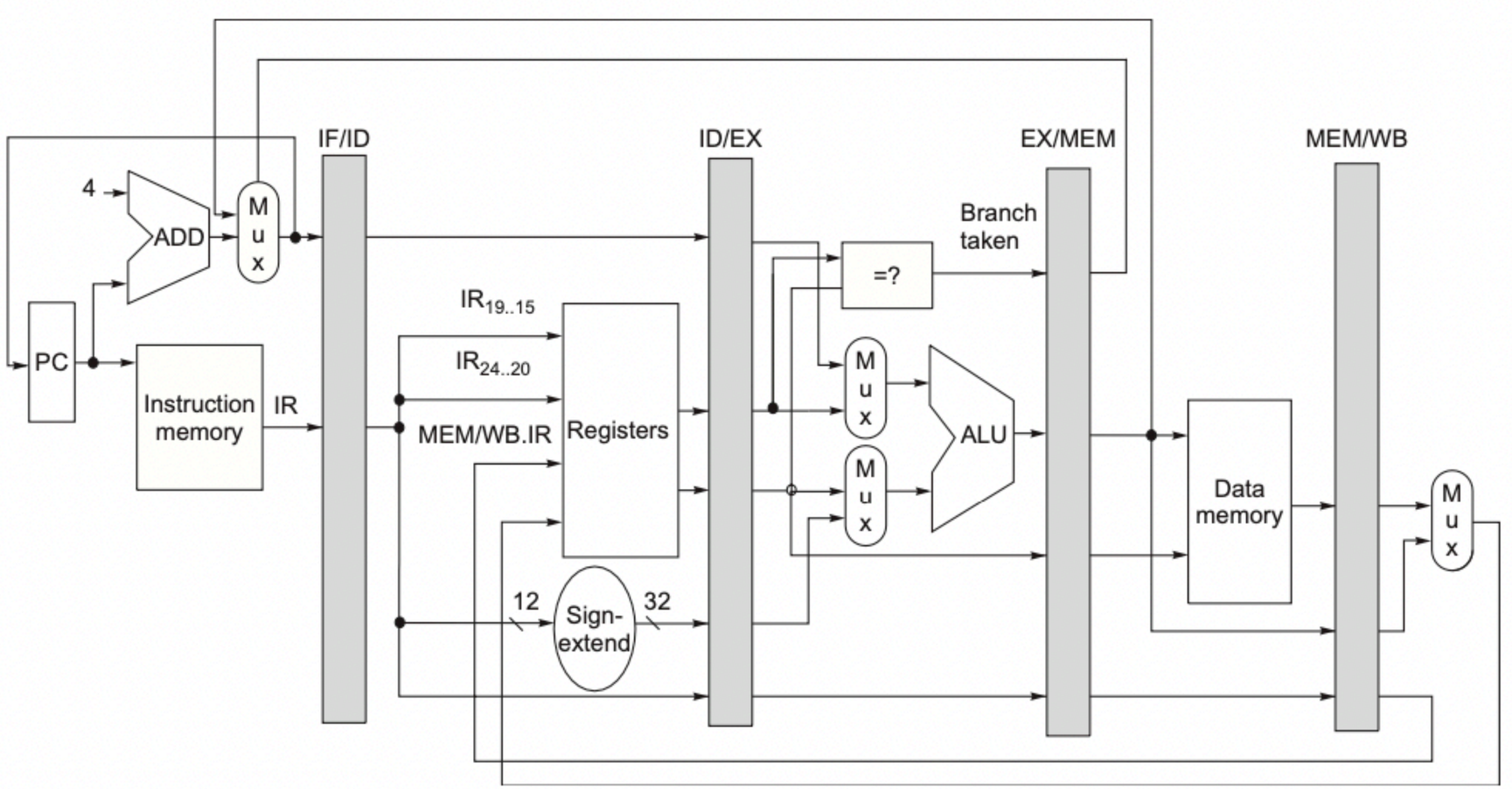


Figure C.19 The data path is pipelined by adding a set of registers, one between each pair of pipe stages. The

RISC-V Detailed Operation

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[rs1]]; ID/EX.B ← Regs[IF/ID.IR[rs2]]; 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 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 == ID/EX.B);
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;	For load only: Regs[MEM/WB.IR[rd]] ← MEM/WB.LMD;	

Figure C.20 Events on every pipe stage of the RISC V pipeline. Let's review the actions in the stages that are specific

RISC-V Pipeline: Abbreviated Diagram I

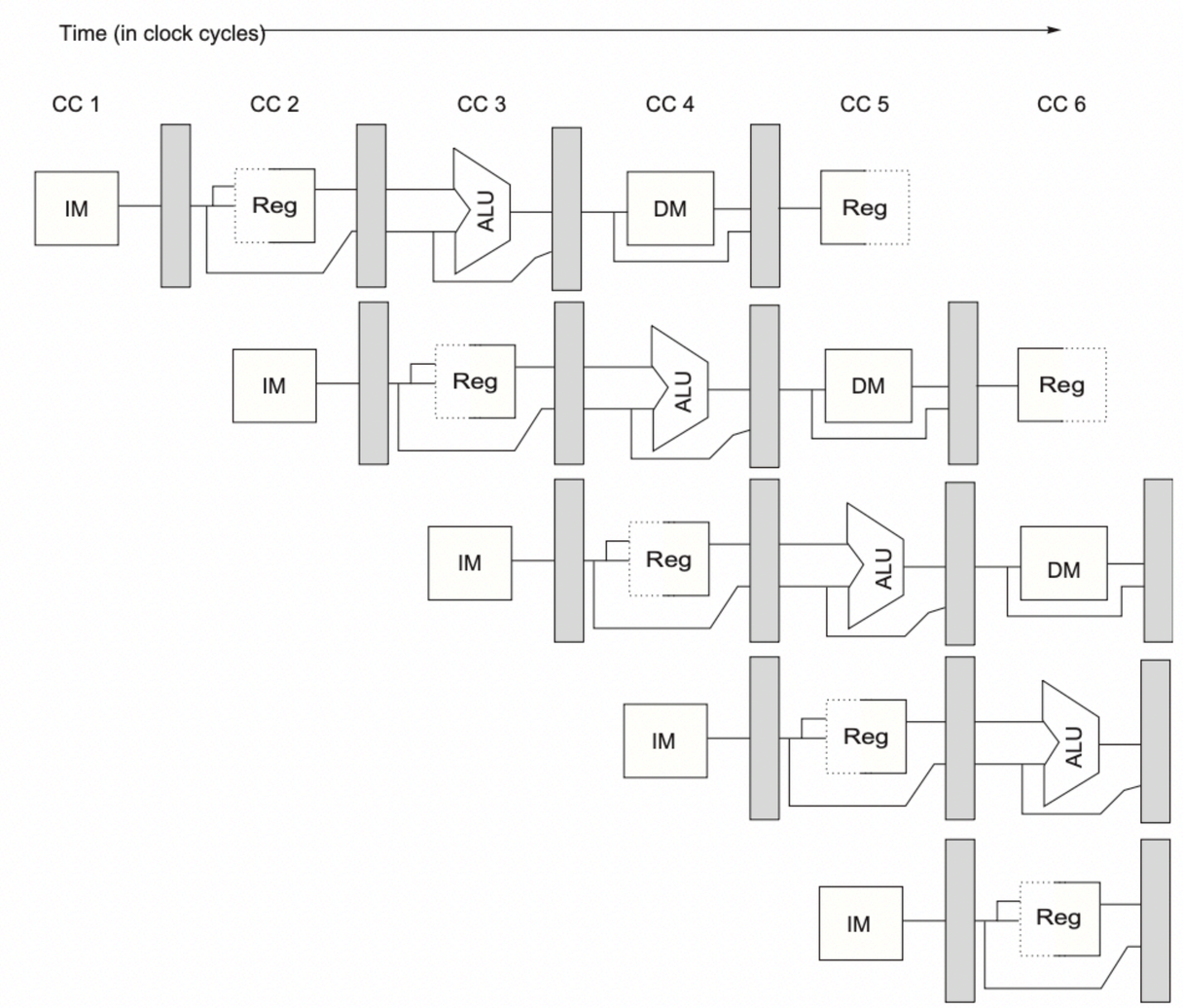


Figure C.3 A pipeline showing the pipeline registers between successive pipeline stages. Notice that the registers

RISC-V Pipeline: Abbreviated Diagram II

	Clock number								
Instruction 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 C.1 Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its five-cycle

RISC-V Performance Example

Example Consider the unpipelined processor in the previous section. Assume that it has a 4 GHz clock (or a 0.5 ns clock cycle) and that it uses four cycles for ALU operations and branches and five 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 0.1 ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

Answer The average instruction execution time on the unpipelined processor is

$$\begin{aligned}\text{Average instruction execution time} &= \text{Clock cycle} \times \text{Average CPI}\\ &= 0.5\,\text{ns} \times [(40\% + 20\%) \times 4 + 40\% \times 5]\\ &= 0.5\ \text{ns} \times 4.4\\ &= 2.2\ \text{ns}\end{aligned}$$

In the pipelined implementation, the clock must run at the speed of the slowest stage plus overhead, which will be 0.5 + 0.1 or 0.6 ns; this is the average instruction execution time. Thus, the speedup from pipelining is

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

$$= \frac{2.2\ \text{ns}}{0.6\ \text{ns}} = 3.7\ \text{times}$$

The 0.1 ns overhead essentially establishes a limit on the effectiveness of pipelining. If the overhead is not affected by changes in the clock cycle, Amdahl's Law tells us that the overhead limits the speedup.

Instruction-Level Parallelism Roadmap

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	C.2
Simple branch scheduling and prediction	Control hazard stalls	C.2
Basic compiler pipeline scheduling	Data hazard stalls	C.2, 3.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	C.7
Loop unrolling	Control hazard stalls	3.2
Advanced branch prediction	Control stalls	3.3
Dynamic scheduling with renaming	Stalls from data hazards, output dependences, and antidependences	3.4
Hardware speculation	Data hazard and control hazard stalls	3.6
Dynamic memory disambiguation	Data hazard stalls with memory	3.6
Issuing multiple instructions per cycle	Ideal CPI	3.7, 3.8
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls	H.2, H.3
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls	H.4, H.5

Figure 3.1 The major techniques examined in Appendix C, Chapter 3, and Appendix H are shown together with the component of the CPI equation that the technique affects.

Data Dependencies

For an instruction sequence *i, j, k*, an instruction *j* is *data-dependent* on instruction *i* if:

- Instruction *i* produces a result that may be used by instruction *j, or*
- Instruction *j* is data-dependent on instruction *k*, and instruction *k* is data-dependent on instruction *i*.

```
Loop:  fld     f0,0(x1)    //f0=array element
       fadd.d  f4,f0,f2    //add scalar in f2
       fsd     f4,0(x1)    //store result
       addi    x1,x1,-8    //decrement pointer 8 bytes
       bne     x1,x2,Loop  //branch x1≠x2
```

Also called *true dependencies*

Name Dependences

The second type of dependence is a *name dependence*. A name dependence occurs when two instructions use the same register or memory location, called a *name*, but there is no flow of data between the instructions associated with that name. There are two types of name dependences between an instruction *i* that *precedes* instruction *j* in program order:

1. 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. In the example on page 171, there is an antidependence between `fsd` and `addi` on register `x1`.
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*.

Control Dependencies

```
if p1 {
        S1;
};
if p2 {
        S2;
}
```

S1 is control-dependent on p1, and S2 is control-dependent on p2 but not on p1.

In general, two constraints are imposed by control dependences:

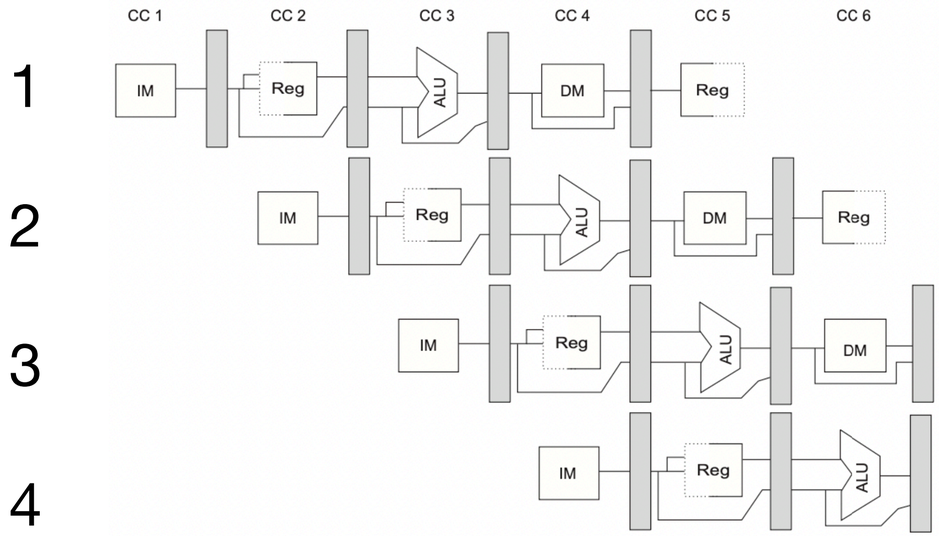
1. An instruction that is control-dependent on a branch cannot be moved *before* the branch so that its execution *is no longer controlled* by the branch. For example, we cannot take an instruction from the then portion of an if statement and move it before the if statement.
2. An instruction that is not control-dependent on a branch cannot be moved *after* the branch so that its execution *is controlled* by the branch. For example, we cannot take a statement before the if statement and move it into the then portion.

Pipeline Hazards

- Structural Hazards
- Data Hazards
- Control Hazards

Structural Hazards

Possible structural hazards:



- In CC4, memory is accessed by both inst 1 and inst 4
- In CC5, register file is accessed by both inst 1 and inst 4

Data Hazards

1. Read After Write (RAW) hazard: the most common, these occur when a read of register *x* by instruction *j* occurs before the write of register *x* by instruction *i*. If this hazard were not prevented instruction j would use the wrong value of *x*.
2. Write After Read (WAR) hazard: this hazard occurs when read of register *x* by instruction *i* occurs after a write of register *x* by instruction *j*. In this case, instruction *i* would use the wrong value of *x*. WAR hazards are impossible in the simple five stage, integrer pipeline, but they occur when instructions are reordered, as we will see when we discuss dynamically scheduled pipelines beginning on page C.65.
3. Write After Write (WAW) hazard: this hazard occurs when write of register *x* by instruction *i* occurs after a write of register *x* by instruction *j*. When this occurs, register *x* will have the wrong value going forward. WAR hazards are also impossible in the simple five stage, integrer pipeline, but they occur when instructions are reordered or when running times vary, as we will see later.

Data Hazard Example

Consider the pipelined execution of these instructions:

```
add     x1,x2,x3
sub     x4,x1,x5
and     x6,x1,x7
or      x8,x1,x9
xor     x10,x1,x11
```

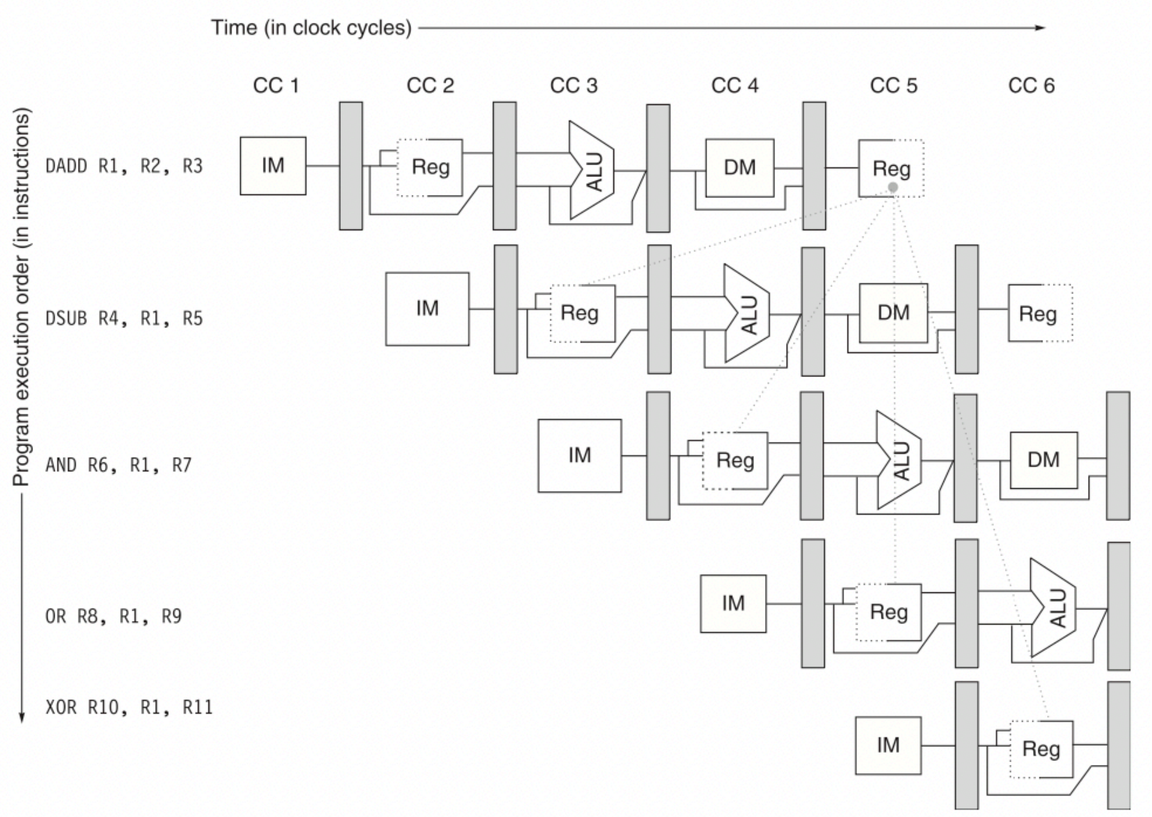
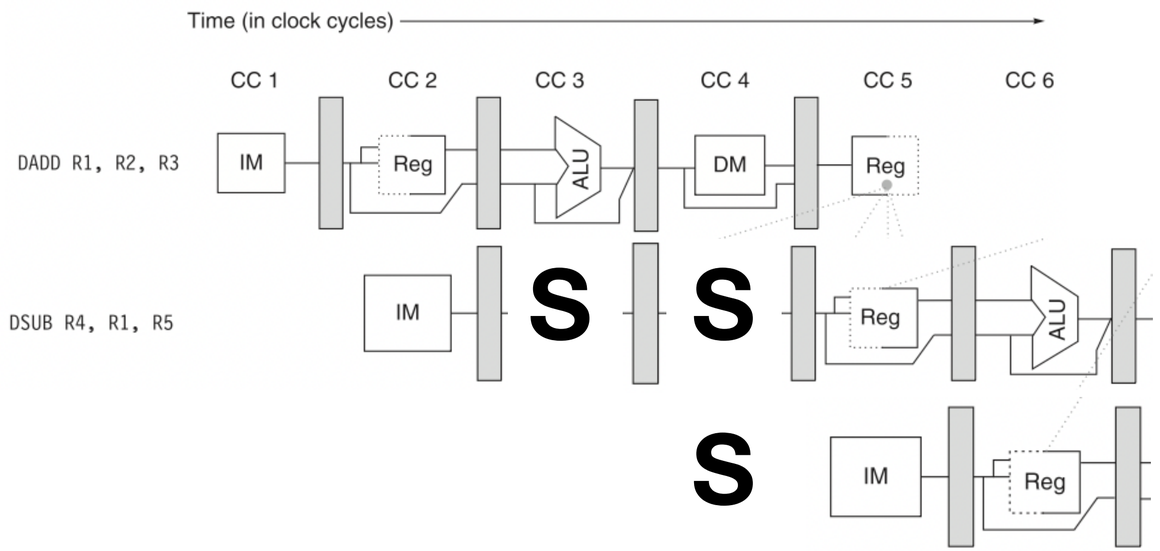


Figure C.4 The use of the result of the `add` instruction in the next three instructions causes a hazard, because the register is not written until after those instructions read it.

Data Hazard - one solution



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, starting with the formula from the previous section:

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

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

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

Pipelining can be thought of as decreasing the CPI or the clock cycle time. Because 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:

$$\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}$$

$$= 1 + \text{Pipelines stall clock cycles per instruction}$$

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

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

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.

Data Forwarding (or Bypassing)

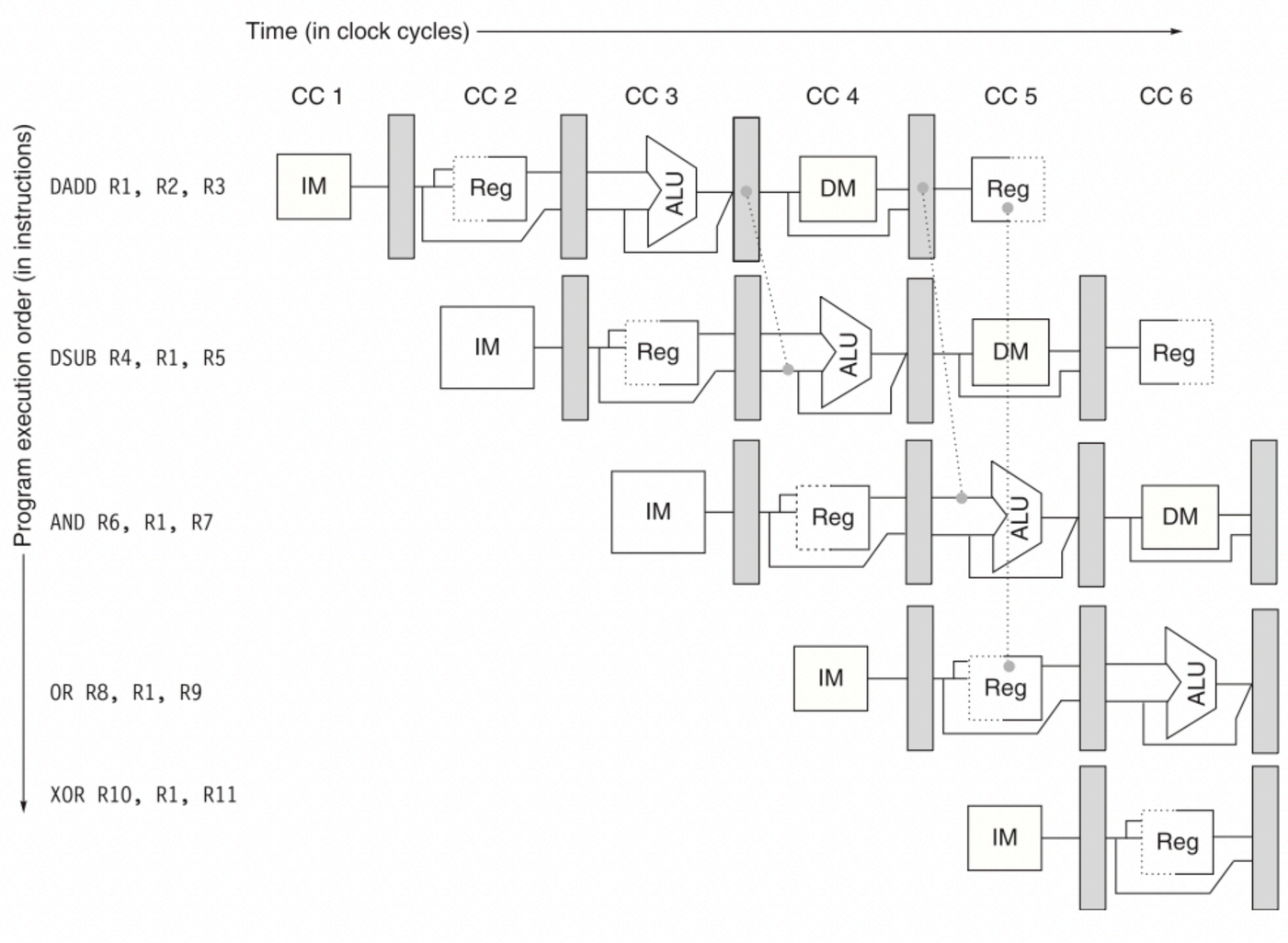


Figure C.5 A set of instructions that depends on the `add` result uses forwarding paths to avoid the data hazard.

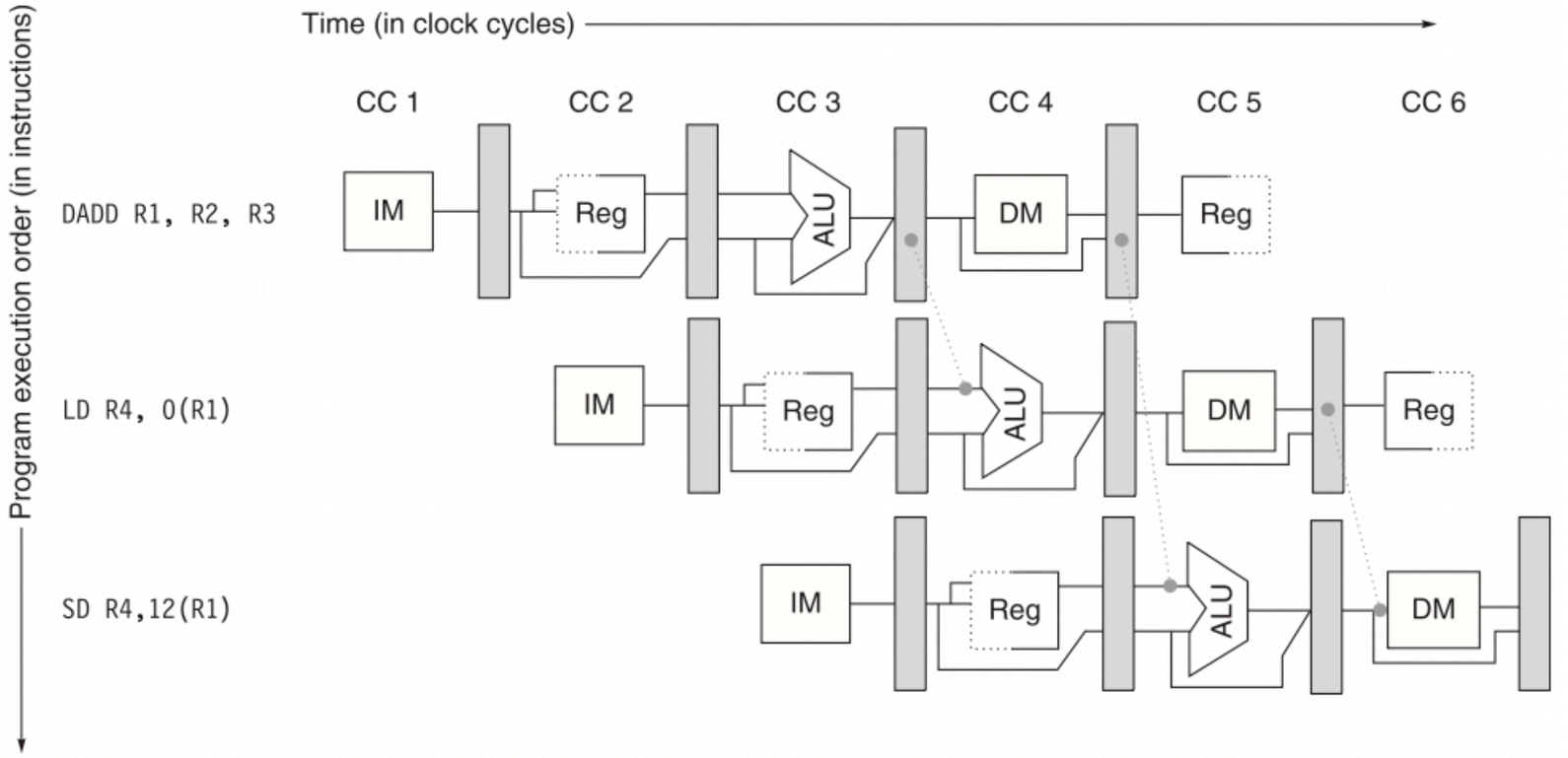


Figure C.6 Forwarding of operand required by stores during MEM. The result of the load is forwarded from the memory output to the memory input to be stored. In addition, the ALU output is forwarded to the ALU input for the address calculation of both the load and the store (this is no different than forwarding to another ALU operation). If the store depended on an immediately preceding ALU operation (not shown herein), the result would need to be forwarded to prevent a stall.

Early Branch decision

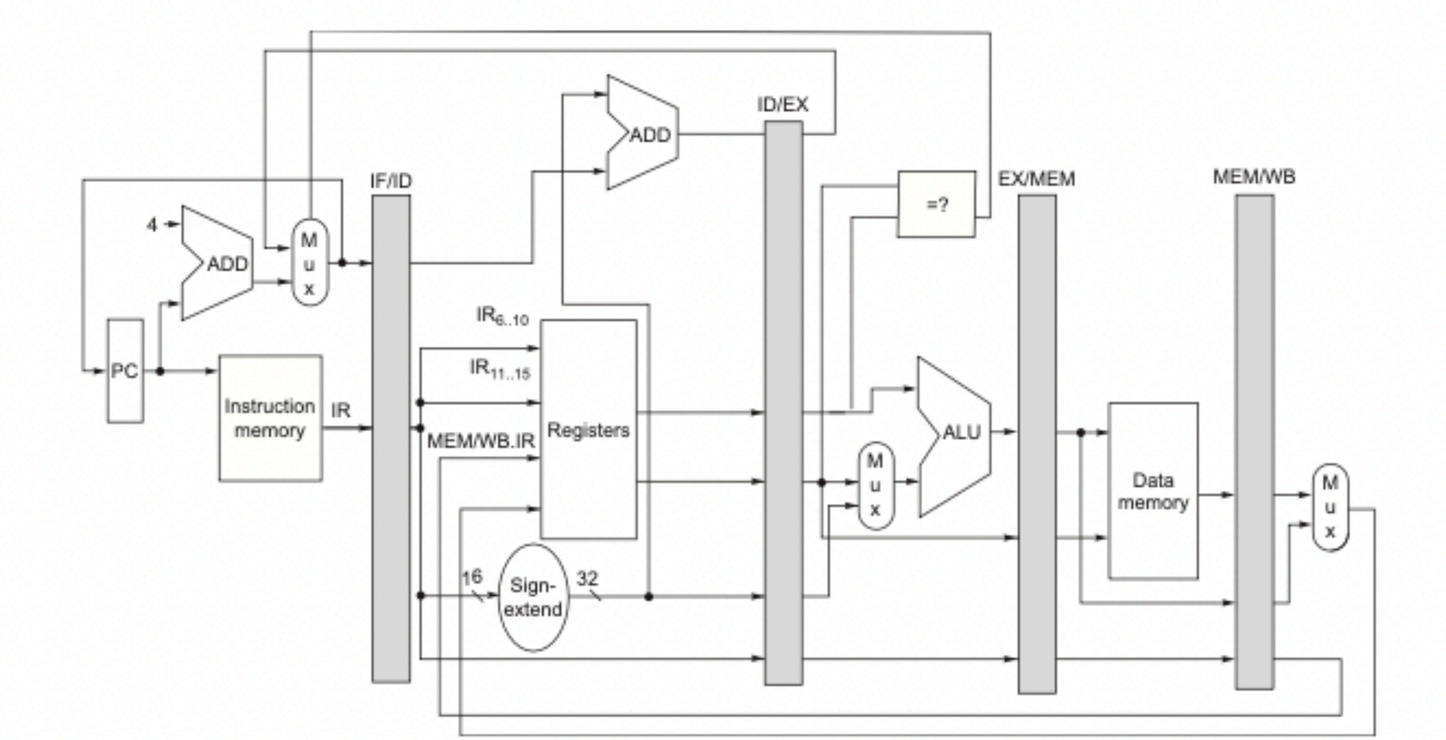


Figure C.25 To minimize the impact of deciding whether a conditional branch is taken, we compute the branch target address in ID while doing the conditional test and final selection of next PC in EX. As mentioned in

Data Hazards Requiring Stalls

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

```
ld      x1,0(x2)
sub     x4,x1,x5
and     x6,x1,x7
or      x8,x1,x9
```

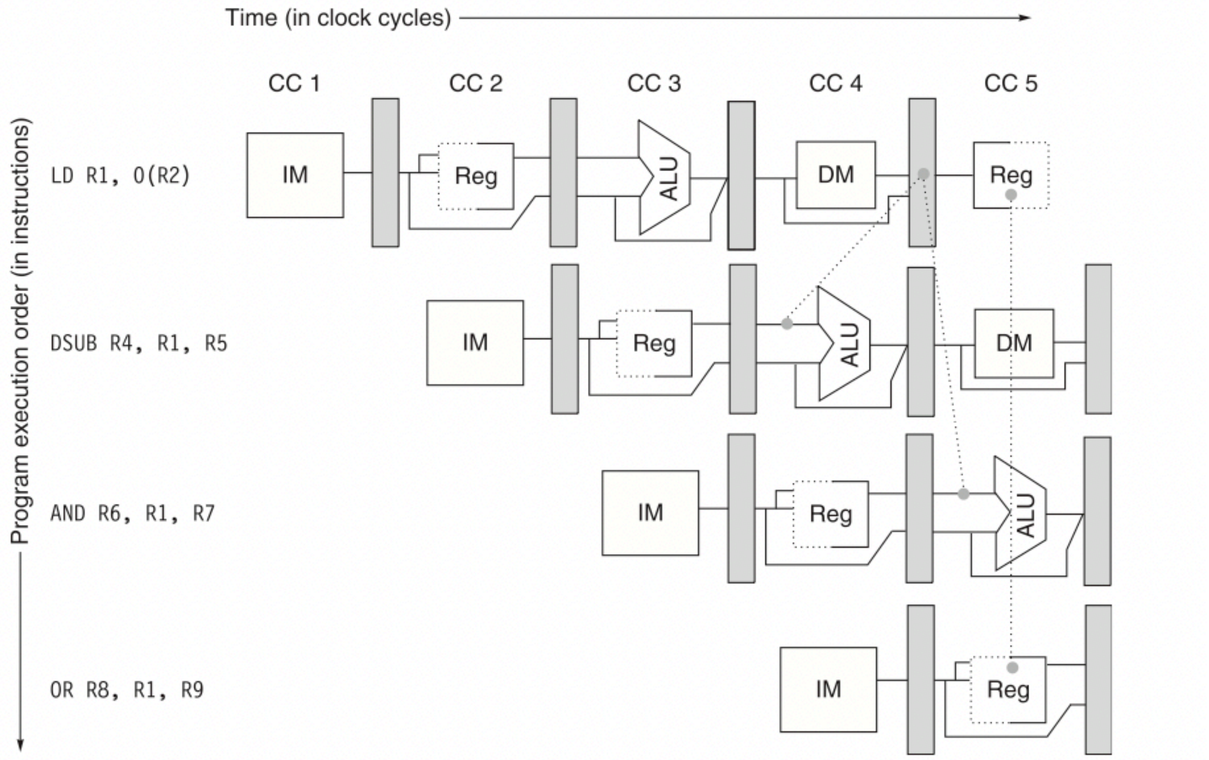


Figure C.7 The load instruction can bypass its results to the `and` and `or` instructions, but not to the `sub`, because that would mean forwarding the result in "negative time."

`ld x1,0(x2)`	IF	ID	EX	MEM	WB				
`sub x4,x1,x5`		IF	ID	EX	MEM	WB			
`and x6,x1,x7`			IF	ID	EX	MEM	WB		
`or x8,x1,x9`				IF	ID	EX	MEM	WB	
`ld x1,0(x2)`	IF	ID	EX	MEM	WB				
`sub x4,x1,x5`		IF	ID	Stall	EX	MEM	WB		
`and x6,x1,x7`			IF	Stall	ID	EX	MEM	WB	
`or x8,x1,x9`				Stall	IF	ID	EX	MEM	WB

Figure C.8 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 `sub`, which occurs at the same time. This problem is solved by inserting a stall, as shown in the bottom half.

Control (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 C.9 A branch causes a one-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.

Predict Not-Taken Scheme

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figure C.10 The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom). When the branch is untaken, determined during ID, we fetch the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall 1 clock cycle.

Branch Prediction

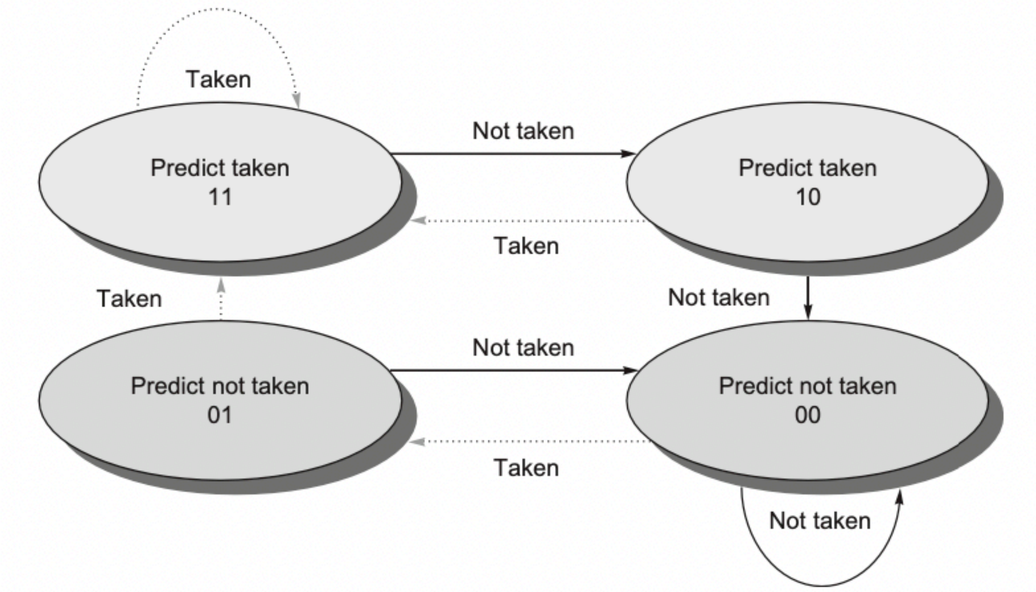


Figure C.15 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually a specialization of a more general scheme that has an n-bit saturating counter for each entry in the prediction buffer. With an n-bit counter, the counter can take on values between 0 and $2^n - 1$: when the counter is greater than or equal to one-half of its maximum value ($2^n - 1$), the branch is predicted as taken; otherwise, it is predicted as untaken. Studies of n-bit predictors have shown that the 2-bit predictors do almost as well, thus most systems rely on 2-bit branch predictors rather than the more general n-bit predictors.

Dynamic Branch Methods

- Branch Prediction Buffer
- Branch Target Buffer

Branch Delay Slot

Original intent of code

```
Loop:   fld      f0, 0(x1)      // f0=array element
        fadd.d   f4, f0, f2     // add scalar in f2
        fsd      f4, 0(x1)      // store result
        addi     x1, x1, -8     // decrement ptr 8 bytes
        bne      x1, x2, Loop   // branch if x1/=x2
```

Now, change architecture to handle branch delay slot

```
Loop:   fld      f0, 0(x1)      // f0=array element
        fadd.d   f4, f0, f2     // add scalar in f2
        addi     x1, x1, -8     // decrement ptr 8 bytes
        bne      x1, x2, Loop   // branch if x1/=x2
        fsd      f4, 8(x1)      // branch delay slot,
                                // still considered in loop
```

Implementing Forwarding

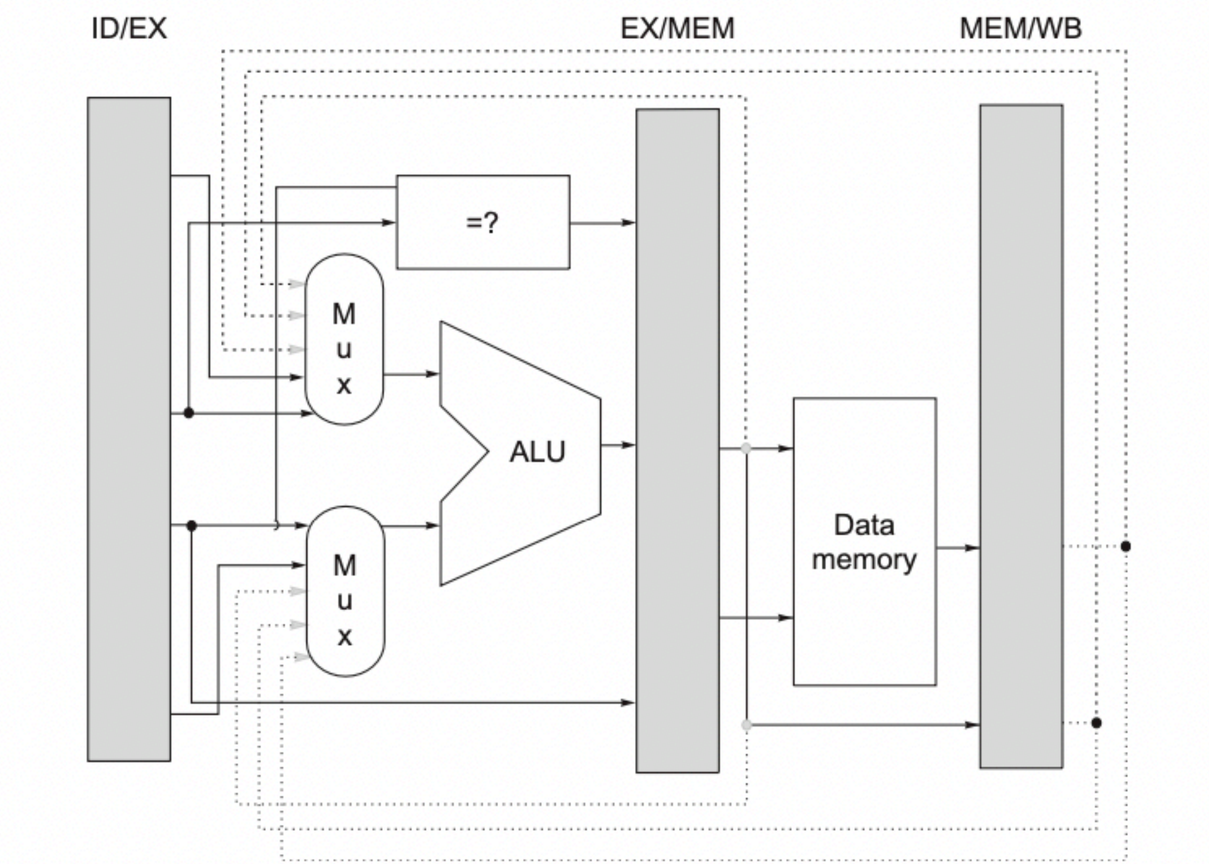


Figure C.24 Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs. The paths correspond to a bypass of: (1) the ALU output at the end of the EX, (2) the ALU output at the end of the MEM stage, and (3) the memory output at the end of the MEM stage.

Branch Delay Slot

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay 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

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction ($i+1$)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figure C.11 The behavior of a delayed branch is the same whether or not the branch is taken. The instructions in the delay slot (there was only one delay slot for most RISC architectures that incorporated them) are executed. If the branch is untaken, execution continues with the instruction after the branch delay instruction; if the branch is taken, execution continues at the branch target. When the instruction in the branch delay slot is also a branch, the meaning is unclear: if the branch is not taken, what should happen to the branch in the branch delay slot? Because of this confusion, architectures with delay branches often disallow putting a branch in the delay slot.

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 C.26 Five categories are used to define what actions are needed for the different exception types. Excep-

RISC-V Exceptions

ld	IF	ID	EX	MEM	WB	
add		IF	ID	EX	MEM	WB

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 C.27 Exceptions that may occur in the RISC V pipeline. Exceptions raised from instruction or data memory access account for six out of eight cases.

Multicycle Stages

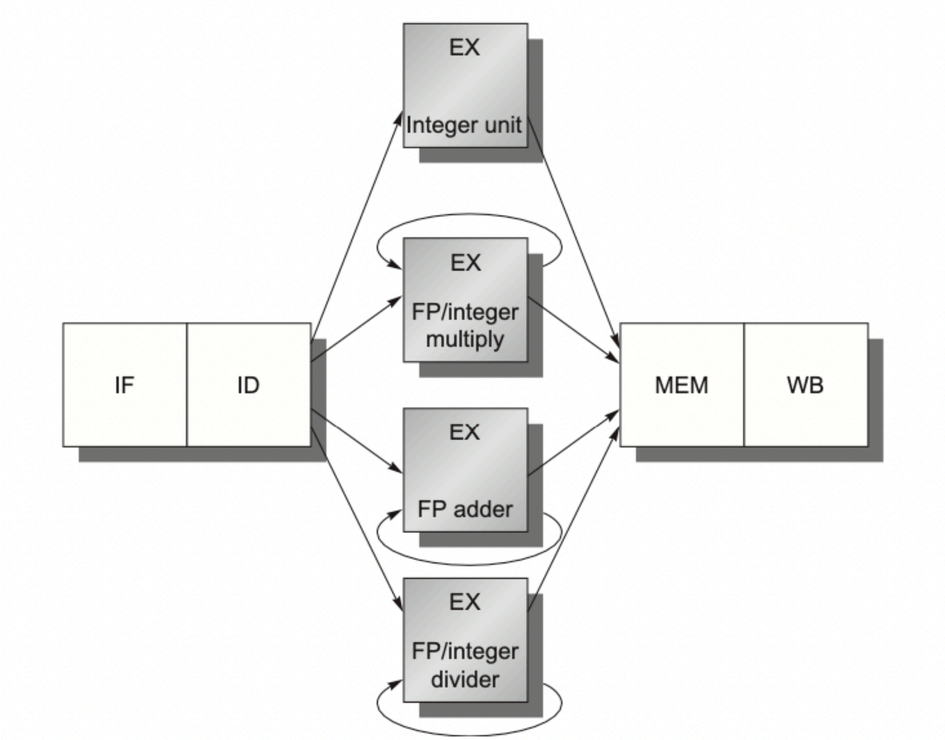


Figure C.28 The RISC V pipeline with three additional unpipelined, floating-point,

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 C.29 Latencies and initiation intervals for functional units.

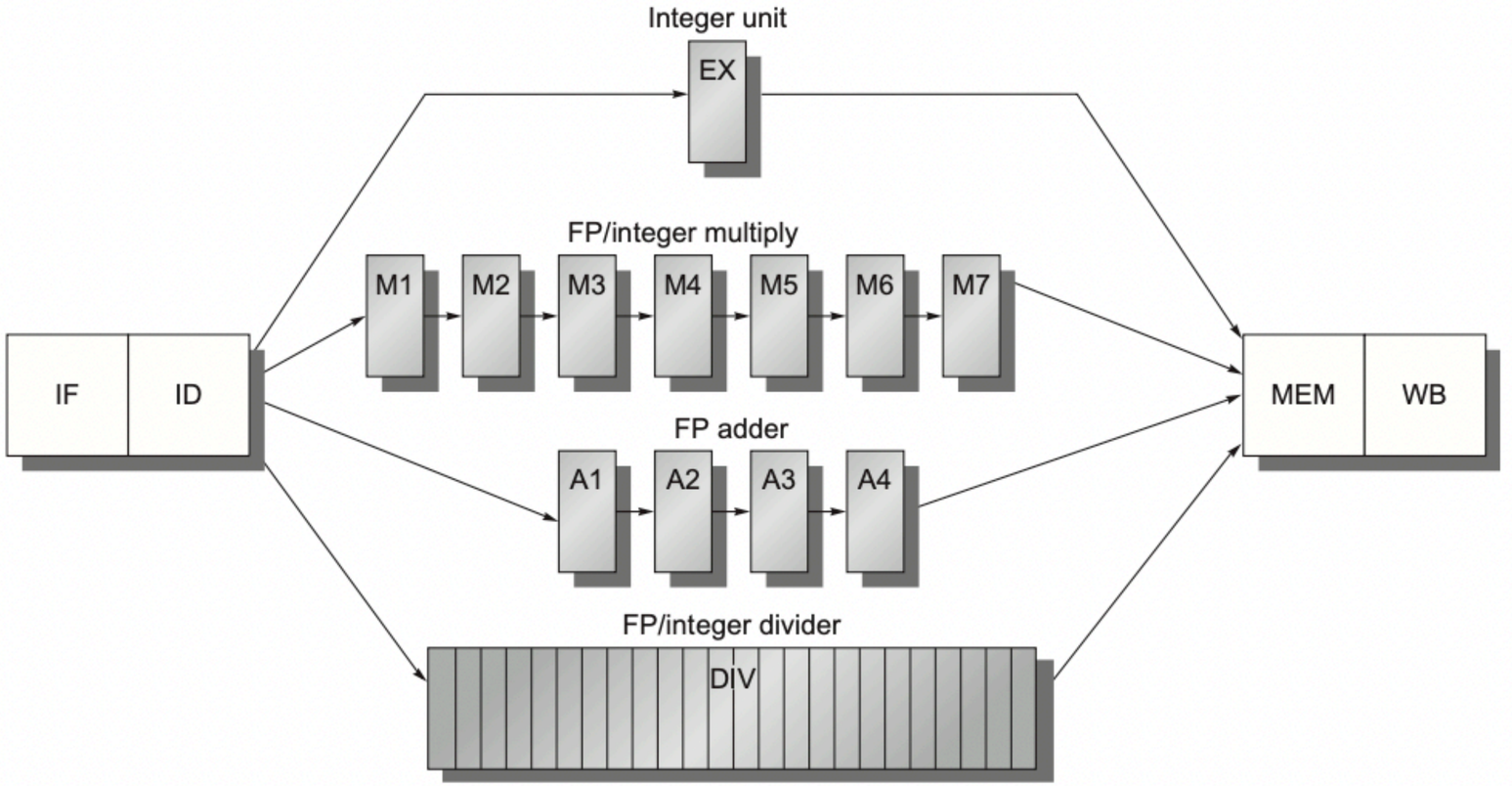


Figure C.30 A pipeline that supports multiple outstanding FP operations. The FP multiplier and adder are fully pipelined and have a depth of seven and four stages, respectively. The FP divider is not pipelined, but requires 24 clock cycles to complete. The latency in instructions between the issue of 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.

Hazards and Forwarding in Longer Latency Pipelines

There are a number of different aspects to the hazard detection and forwarding for a pipeline like that shown in Figure C.30.

1. Because the divide unit is not fully pipelined, structural hazards can occur. These will need to be detected and issuing instructions will need to be stalled.
2. Because the instructions have varying running times, the number of register writes required in a cycle can be larger than 1.
3. Write after write (WAW) hazards are possible, because instructions no longer reach WB in order. Note that write after read (WAR) hazards are not possible, because the register reads always occur in ID.
4. Instructions can complete in a different order than they were issued, causing problems with exceptions; we deal with this in the next subsection.
5. Because of longer latency of operations, stalls for RAW hazards will be more frequent.

	Clock cycle number																
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
`fld f4,0(x2)`	IF	ID	EX	MEM	WB												
`fmul.d f0,f4,f6`		IF	ID	Stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
`fadd.d f2,f0,f8`			IF	Stall	ID	Stall	Stall	Stall	Stall	Stall	Stall	A1	A2	A3	A4	MEM	WB
`fsd f2,0(x2)`					IF	Stall	Stall	Stall	Stall	Stall	Stall	ID	EX	Stall	Stall	Stall	MEM

Figure C.32 A typical FP code sequence showing the stalls arising from RAW hazards. The longer pipeline substantially raises the frequency of stalls versus the shallower integer pipeline. Each instruction in this sequence is dependent on the previous and proceeds as soon as data are available, which assumes the pipeline has full bypassing and forwarding. The `fsd` must be stalled an extra cycle so that its MEM does not conflict with the `fadd.d`. Extra hardware could easily handle this case.

	Clock cycle number										
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
`fmul.d f0,f4,f6`	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
`...`		IF	ID	EX	MEM	WB					
`...`			IF	ID	EX	MEM	WB				
`fadd.d f2,f4,f6`				IF	ID	A1	A2	A3	A4	MEM	WB
`...`					IF	ID	EX	MEM	WB		
`...`						IF	ID	EX	MEM	WB	
`fld f2,0(x2)`							IF	ID	EX	MEM	WB

Figure C.33 Three instructions want to perform a write-back to the FP register file simultaneously, as shown in clock cycle 11. This is *not* the worst case, because an earlier divide in the FP unit could also finish on the same clock. Note that although the `fmul.d`, `fadd.d`, and `fld` are in the MEM stage in clock cycle 10, only the `fld` actually uses the memory, so no structural hazard exists for MEM.

Deeper Pipelines

MIPS R4000

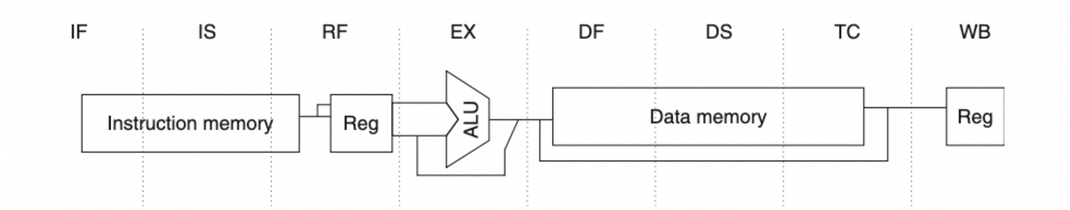


Figure C.36 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction 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, because we cannot write the data into the register until we know whether the cache access was a hit or not.

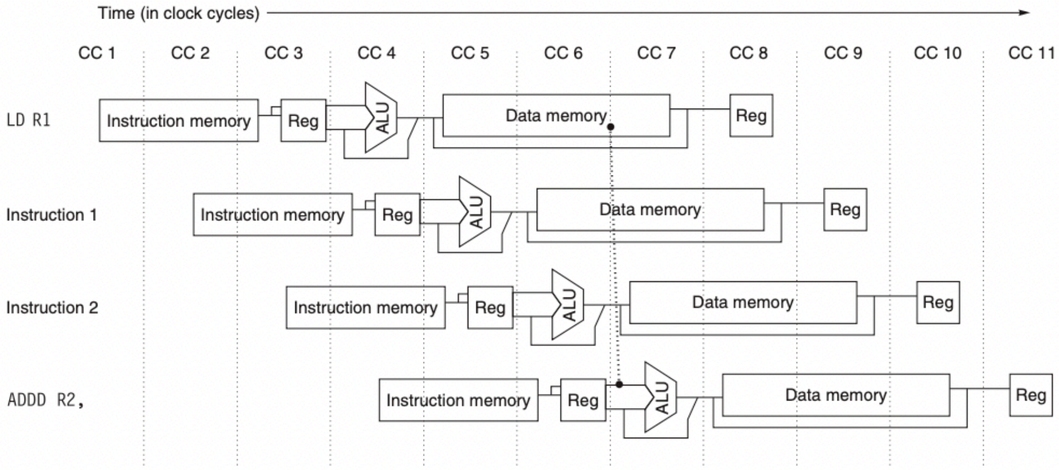


Figure C.37 The structure of the R4000 integer pipeline leads to a x1 load delay. A x1 delay is possible because the data value is available at the end of DS and can be bypassed. If the tag check in TC indicates a miss, the pipeline is backed up a cycle, when the correct data are available.

Floating Point

Stage	Functional unit	Description
A	FP adder	Mantissa add stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

Figure C.41 The eight stages used in the R4000 floating-point pipelines.

FP instruction	Latency	Initiation interval	Pipe stages
Add, subtract	4	3	U, S+A, A+R, R+S
Multiply	8	4	U, E+M, M, M, M, N, N+A, R
Divide	36	35	U, A, R, D^{28}, D+A, D+R, D+A, D+R, A, R
Square root	112	111	U, E, $(A+R)^{108}$, A, R
Negate	2	1	U, S
Absolute value	2	1	U, S
FP compare	3	2	U, A, R

Figure C.42 The latencies and initiation intervals for the FP operations initiation intervals for the FP operations

		Clock cycle												
Operation	**Issue/stall**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Multiply	Issue	U	E+M	M	M	M	N	**N+A**	**R**					
Add	Issue		U	S+A	A+R	R+S								
	Issue			U	S+A	A+R	R+S							
	Issue				U	S+A	A+R	R+S						
	Stall					U	S+A	**A+R**	**R+S**					
	Stall						U	**S+A**	**A+R**	R+S				
	Issue							U	S+A	A+R	R+S			
	Issue								U	S+A	A+R	R+S		

Figure C.43 An FP multiply issued at clock 0 is followed by a single FP add issued between clocks 1 and 7. The