

1 – Intro to Parallelism

Parallelism consists in breaking a *task* down into *stages* and executing the stages concurrently.

A parallel computation is correct if its results are equivalent to the results of a correct sequential execution.

2 – Mutual Interference

Parallelism fails when the order in which the stages are performed matters. In software this sensitivity to order is called a *dependency*.

Computational tasks are composed of *instructions* which operate on data. Two instructions are said to be *mutually interfering* when one operation modifies data which the other instruction reads or writes.

3 – Bernstein's Conditions

Consider two tasks, T_i and T_j , and the set of memory locations read a task are denoted $R(T)$ and the set of memory locations written to by a task are $W(T)$. Then T_i and T_j are non interfering if and only if they satisfy *Bernsteins Conditions* which are:

$$\emptyset = (R(T_i) \cap W(T_j)) \cup (W(T_i) \cap R(T_j)) \cup (W(T_i) \cap W(T_j)) \quad (1)$$

4 – Intro to Pipelining

Consider a *task* broken into *stages* of work. *Pipelining* is a form of parallelism in which overlapping tasks are concurrently in different stages.

Example: automobile assembly lines.

In architecture pipelining occurs at the instruction level, pipelining is used in software as well (but that is beyond the scope of the course).

5 – Pipelining Performance Goals

Consider a pipeline, with n stages s_1, s_2, \dots, s_n , with stage s_i taking time t_i to complete.

- The total time a task must spend in the pipeline is at least $\sum_{i=1}^n t_i$ since the task must visit each stage. This no better than sequential processing.
- However the pipeline can service tasks in parallel at each stage, so it can complete a task every $\max_{1 \leq i \leq n} t_i$ units of time.
- The througput is then $\frac{1}{\max_{1 \leq i \leq n} t_i}$.

This gives pipeline architects incentive to have all stages of the pipeline take about the same amount of time.

6 – Pipelining in DLX

An instruction in DLX is pipelined through the following stages, called *cycles*:

1. Instruction Fetch Cycle (IF) — Loads the instruction into the pipe
2. Instruction Decode/Register Fetch Cycle (ID) — Decode the instruction and fetch the register operands.
3. Execution/Effective Address Cycle (EA) — The ALU performs operation dependent work on the operands fetched in ID.
4. Memory Access/Branch Completion Cycle (MEM) — Does memory access for load/store instructions or updates the PC for a branch. Other instructions do not use this stage.
5. Write-back Cycle (WB) — Updates the register file with the results of the operation.

7 – The DLX Datapath

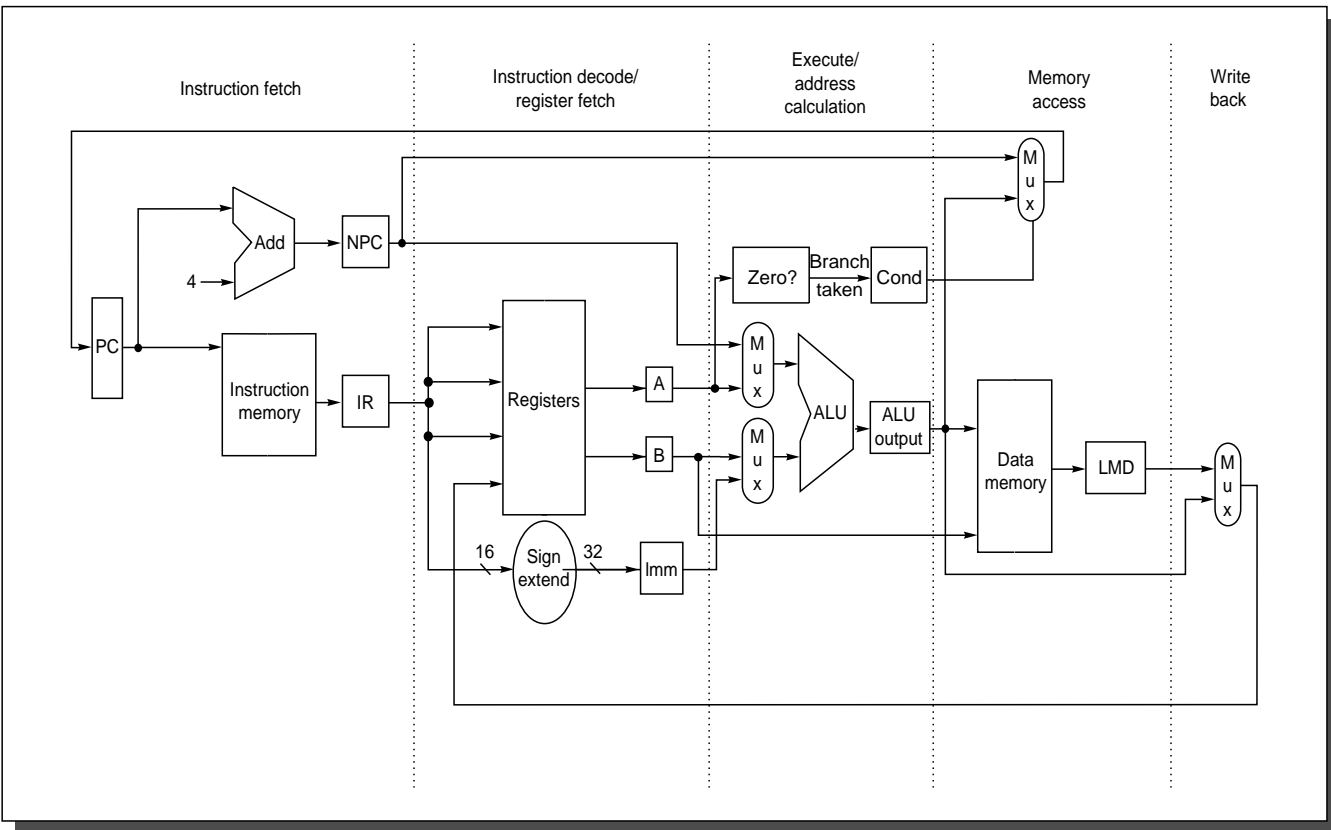


FIGURE 3.1 The implementation of the DLX datapath allows every instruction to be executed in four or five clock cycles.

8 – IF Cycle Semantics

The IF Cycle does the following:

1. $IR \leftarrow \text{Mem}[PC]$
2. $NPC \leftarrow PC + 4$

Where **IR** is the instruction register, and **NPC** holds the next sequential PC.

9 – ID Cycle Semantics

The ID Cycle does the following:

1. $A \leftarrow Regs[IR_{6..10}]$
2. $B \leftarrow Regs[IR_{11..15}]$
3. $Imm \leftarrow ((IR_{16})^{16} \#\# IR_{16..31})$

Note that this can be done due to *fixed field encoding*, and if an unnecessary register read occurs, the penalty is negligible.

10 – EX Cycle Semantics

The semantics of this cycle are dependent on the type of instruction (as determined in the ID cycle):

- Memory Reference:
 1. $ALUOutput \leftarrow A + Imm$
- Register-Register ALU instruction:
 1. $ALUOutput \leftarrow A \text{ op } B$
where op is the (binary) operator.
- Register-Immediate ALU instruction:
 1. $ALUOutput \leftarrow A \text{ op } Imm$
- Branch:
 1. $ALUOutput \leftarrow NPC \text{ op } Imm$
 2. $Cond \leftarrow (A \text{ op } 0)$

11 – MEM Cycle Semantics

Only the following instructions use the MEM Cycle:

- Load:
 1. $LMD \leftarrow MEM[ALUOutput]$
where LMD is the load memory data register.
- Store:
 1. $MEM[ALUOutput] \leftarrow B$
- Branch:
 1. if (cond) $PC \leftarrow ALUOutput$
else $PC \leftarrow NPC$

12 – WB Cycle Semantics

The WB cycle places the result of the operation into its spot in the register file:

- Register-Register ALU operations:
 1. $\text{Regs}[\text{IR}_{16..20}] \leftarrow \text{ALUOutput};$
- Register-Immediate ALU instruction:
 1. $\text{Regs}[\text{IR}_{11..15}] \leftarrow \text{ALUOutput};$
- Load Instruction:
 1. $\text{Regs}[\text{IR}_{11..15}] \leftarrow \text{LMD};$

13 – Overlap in Execution

Pipelining allows utilization of available parts of the datapath in overlapping instructions.

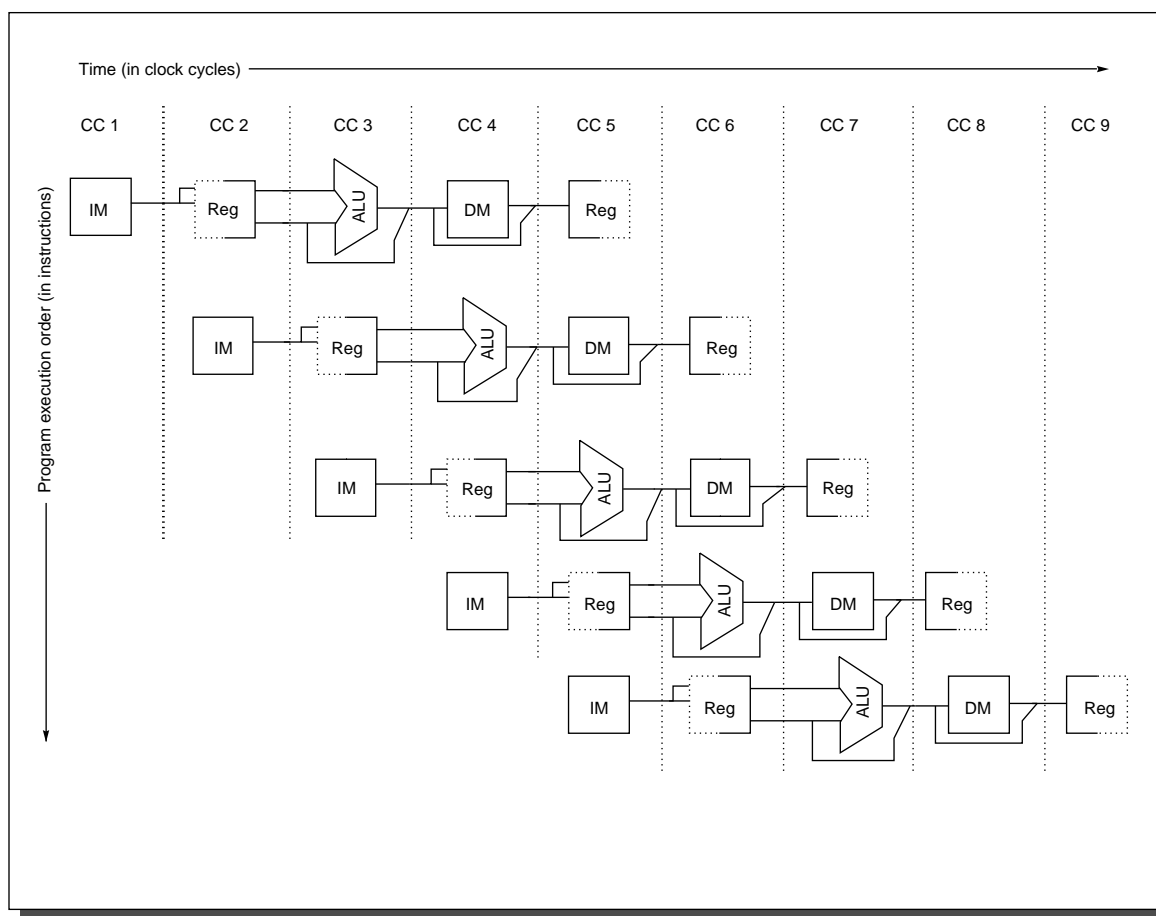


FIGURE 3.3 The pipeline can be thought of as a series of datapaths shifted in time.

14 – Registers “Glue” Stages Together

Internal (hidden to the user) registers are used to provide an interface between the stages of the pipe.

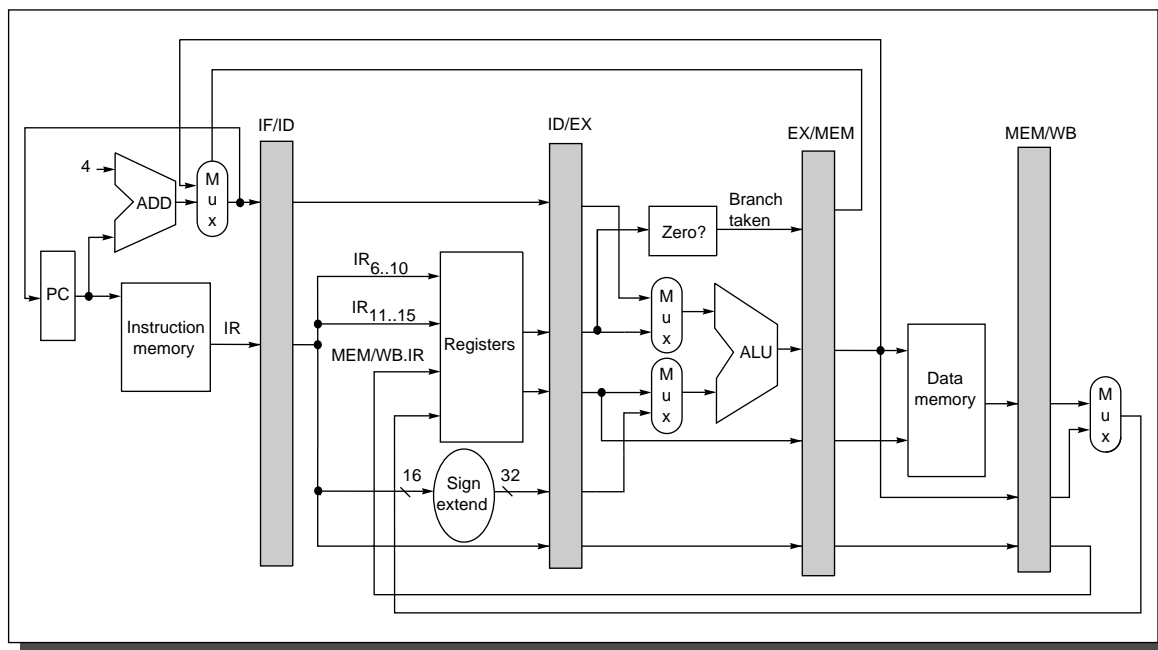


FIGURE 3.4 The datapath is pipelined by adding a set of registers, one between each pair of pipe stages.

15 – In Class Exercise

Let's compare a pipelined vs. a non pipelined implementation. Consider a machine having a 10-ns cpu clock cycle which uses 4 cycles for ALU operations and branches, but 5 cycles for memory operations. Now suppose further that the machine uses 45%, 15% and 40% respectively for each operation type. Suppose that pipelining adds 1 ns to each clock cycle (due to clock skew and setup). How much speedup does pipelining provide?

16 – Solution

Let $t_{\text{instruction}}$ be the mean time it takes to execute an instruction. Then assuming no pipelining:

$$\begin{aligned}t_{\text{instruction}} &= \text{Clock Cycle} \times \text{Mean CPI} \\ &= 10\text{ns} \times [(0.45 + 0.15) \times 4 + 0.4 \times 5] \\ &= 10\text{ns} \times 4.4 = 44\text{ns}\end{aligned}$$

Assuming pipelining, the speed of the slowest stage is 10ns with an additional 1ns overhead, so with maximum pipeline speedup assumed:

$$\begin{aligned}t_{\text{pipelined}} &= \text{Slowest stage} + \text{Overhead} \\ &= 10\text{ns} + 1\text{ns} = 11\text{ns} \\ \text{SpeedUp} &= \frac{t_{\text{instruction}}}{t_{\text{pipelined}}} = \frac{44\text{ns}}{11\text{ns}} = 4 \quad (2)\end{aligned}$$

17 – Hazards

Hazards are obstacles to efficient pipelining. Recall that mutual interference limits parallelism. In hardware, interference conditions are referred to as *data hazards*. The following hazards exist:

1. Structural Hazards — caused by resource limitations of the hardware.
2. Data Hazards — caused by data dependencies in prior instructions.
3. Control Hazards — caused by pipelining of flow of control instructions that change the PC.

Resolving hazards can require *stalling* the pipeline, causing later operations to delay until the hazard is cleared.

18 – Performance With Hazards 1 of 4

Recall the speedup from pipelining:

$$\begin{aligned} \text{Speedup} &= \frac{\text{mean instruction time unpipelined}}{\text{mean instruction time pipelined}} \\ &= \frac{\text{CPI unpipelined} \times \text{Clock Cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock Cycle pipelined}} \\ &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock Cycle unpipelined}}{\text{Clock Cycle pipelined}} \end{aligned}$$

19 – Performance With Hazards 2 of 4

Since the ideal CPI for a pipelined machine is typically 1:

$$\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Pipeline Stall Cycles per Instruction}$$

$$= 1 + \text{Pipeline Stall Cycles per Instruction}$$

$$\text{Speedup} = \frac{\text{CPI}_{\text{pipelined}}}{1 + \text{Pipeline Stall Cycles per Instruction}}$$

When all instructions take the same number of cycles, which is the *pipeline depth* (number of stages):

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline Stall Cycles per Instruction}}$$

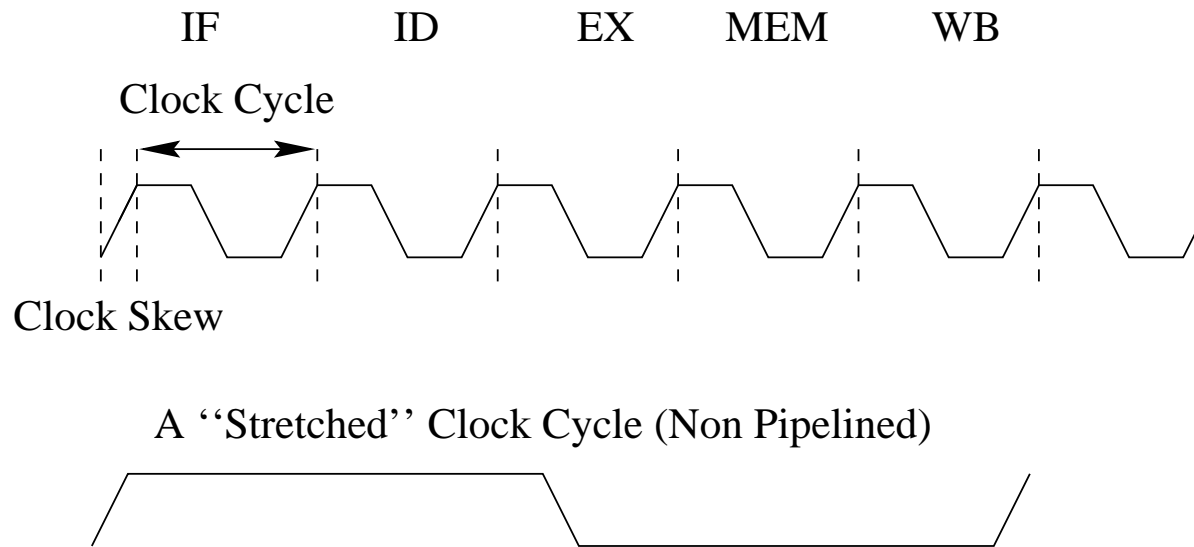
20 – Performance With Hazards 3 of 4

So putting it all together yields:

$$\begin{aligned}
 \text{Speedup} &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock Cycle unpipelined}}{\text{Clock Cycle pipelined}} \\
 &= \frac{1}{1 + \text{Pipeline Stall Cycles per Instruction}} \times \frac{\text{Clock Cycle unpipelined}}{\text{Clock Cycle pipelined}}
 \end{aligned}$$

21 – Performance With Hazards 4 of 4

There are two possibilities of how to handle the clock cycle time, one is to keep it the same frequency and have multicycle instructions for the CPU or to “stretch” the clock cycle as shown below:



22 – Performance With Hazards 4 of 4

When the pipe stages are perfectly balanced with negligible overhead and each stage has a uniform clock cycle time:

$$\text{Pipeline depth} = \frac{\text{Clock Cycle unpipelined}}{\text{Clock Cycle pipelined}}$$

$$\text{Speedup} = \frac{1}{1 + \text{Pipeline Stall Cycles per Instruction}} \times \text{Pipeline depth}$$

23 – Structural Hazards

If an insufficient number of functional units exist within a pipelined processor, a resource conflict called a *structural hazard* can occur.

Example: Simultaneous instruction fetch and memory access via loads and stores requires multiple memory ports.

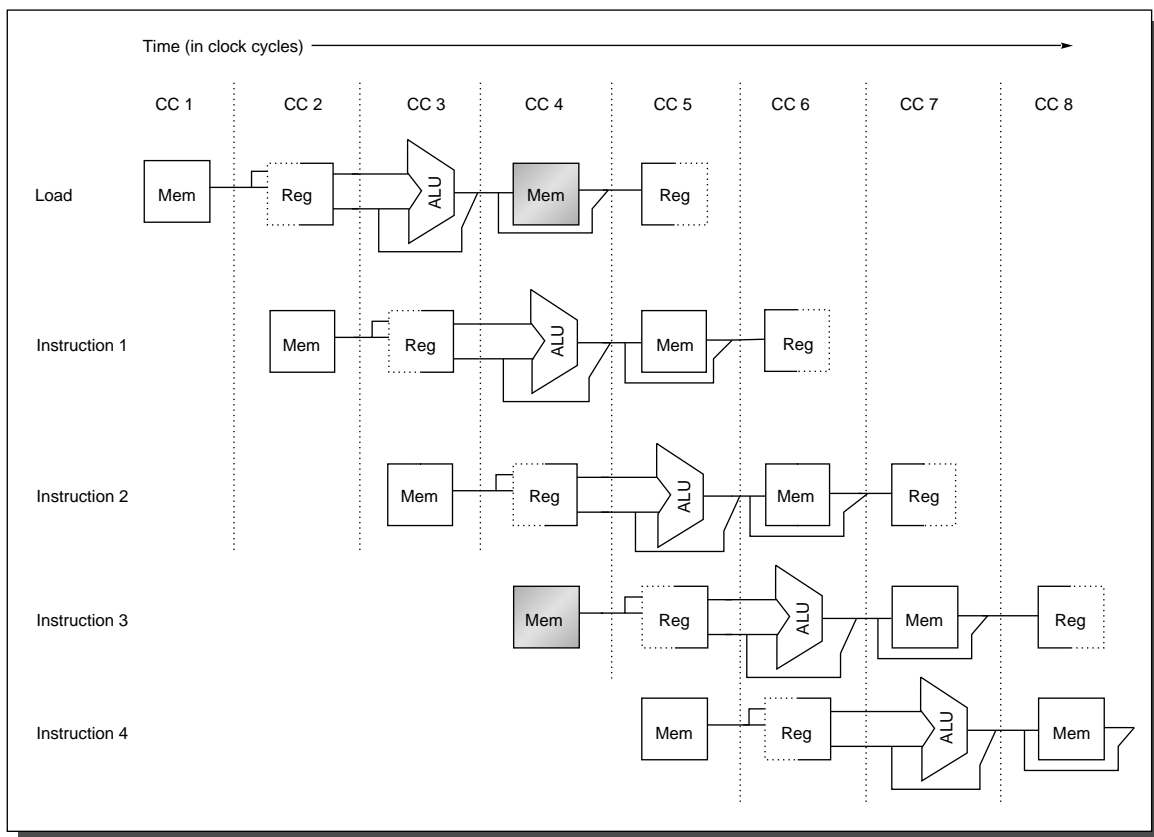


FIGURE 3.6 A machine with only one memory port will generate a conflict whenever a memory reference occurs.

24 – Stalls due to Structural Hazards

Pending instructions will be *stalled* in response to the resulting *pipeline bubble* in which no further instructions can advance.

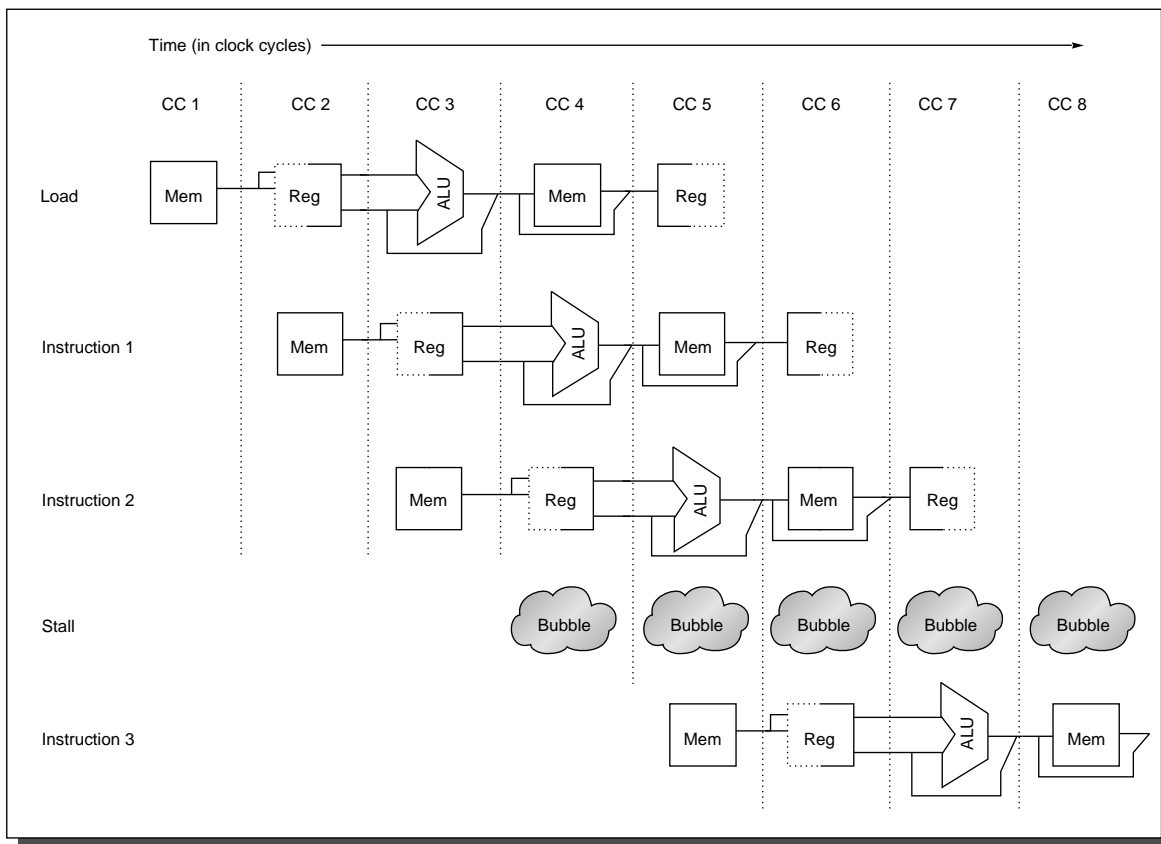


FIGURE 3.7 The structural hazard causes pipeline bubbles to be inserted.

25 – In Class Exercise

Suppose that the data references occur in 30% of the instructions, and that the ideal CPI without structural hazards is 1. Assume that the clock rate for a machine with the structural hazard is 1.1 times slower than the clock rate of a machine without the hazard. Treating other performance losses as negligible, what is the average instruction time of the machine with the structural hazard relative to the machine without the structural hazard?

26 – Solution

Just compute the mean instruction time on the two machines, and then divide for the speedup. Letting the machine which does not stall have the clock cycle time $\text{Clock Cycle Time}_{\text{ideal}}$:

$$\begin{aligned} \text{Mean Instruction Time} &= \text{CPI} \times \text{Clock Cycle Time} \\ &= (1 \times (1 - 0.3) + 2 \times 0.3) \times \\ &\quad \frac{\text{Clock Cycle Time}_{\text{ideal}}}{1.1} \\ &\approx 1.18 \times \text{Clock Cycle Time}_{\text{ideal}} \end{aligned} \tag{3}$$

The machine without the structural hazard is faster, however it might either be impractical to construct (due to limited transistor density).

27 – Data Hazards

Data hazards are induced by mutual interference of instructions which are concurrently in the pipeline. Below we see R1 is used in later instructions.

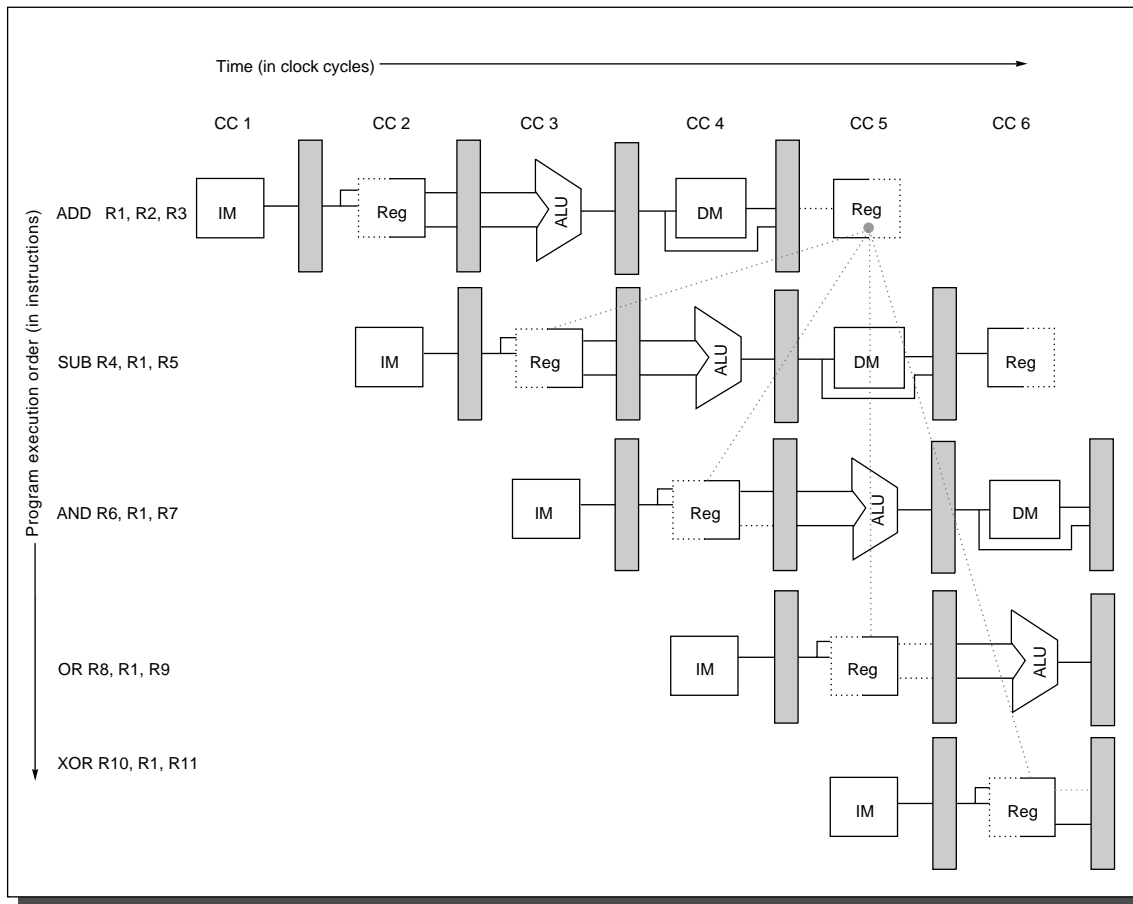


FIGURE 3.9 The use of the result of the `ADD` instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

28 – Forwarding Minimizes Data Hazards

Extra circuitry can be introduced to route results to where they are required.

1. The ALU result from the EX/MEM register is fed back to the ALU input latches.
2. If the forwarding hardware detects that the output of a prior ALU operation is input to the current ALU operation, the forwarded results are selected, otherwise the register file is read.

29 – Forwarding During ALU Operations

From our previous example we see R1 is forwarded, note that later the value happens to be in the MEM/WB registers used by the AND operation.

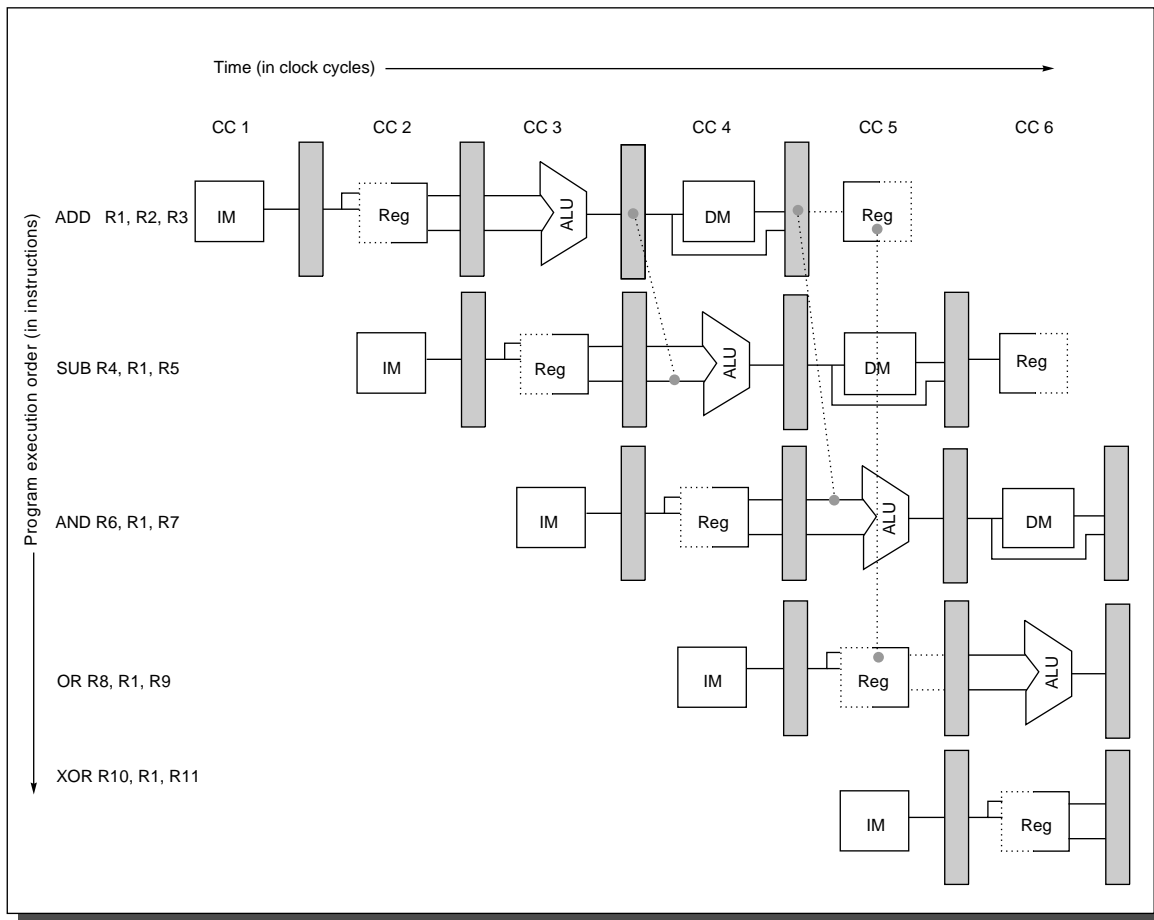


FIGURE 3.10 A set of instructions that depend on the `ADD` result use forwarding paths to avoid the data hazard.

30 – Forwarding During Loads and Stores

This technique can be generalized to other parts of the processor. Here we see R4 is used by a load following a store with the MEM/WB register being forwarded.

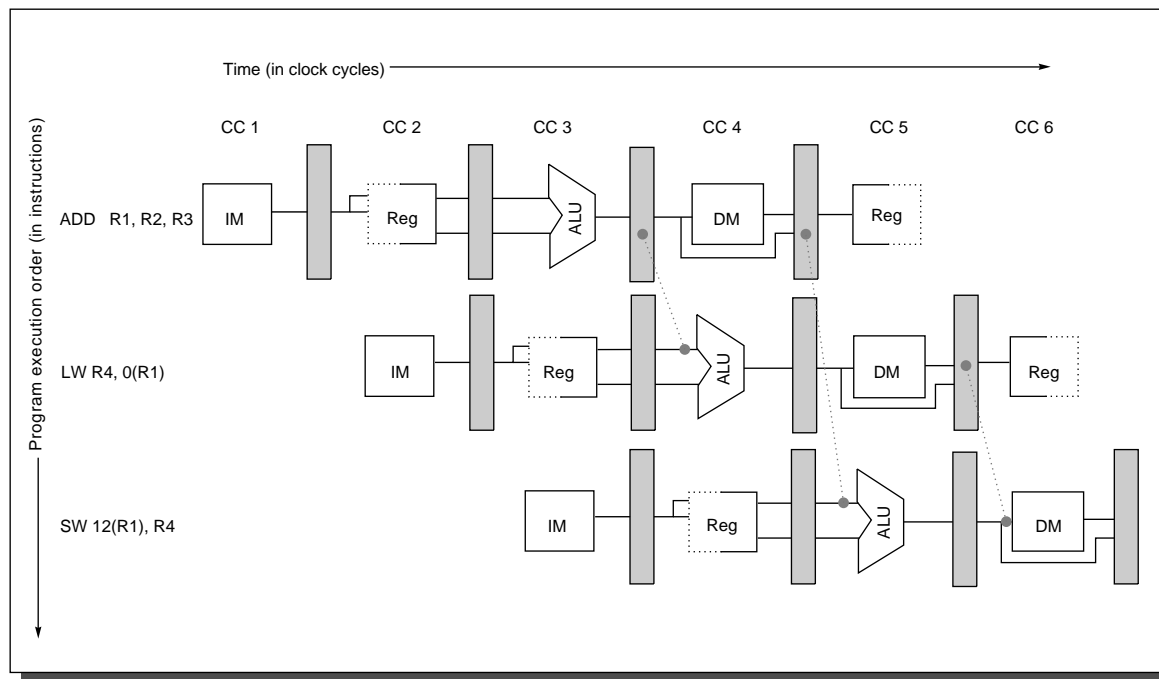


FIGURE 3.11 Stores require an operand during MEM, and forwarding of that operand is shown here.

31 – Types of Data Hazards

Data hazards reflect a dependence between instructions as we saw before in Bernstein's conditions. Consider instructions i and j , with i preceding j . The possible hazards are:

1. RAW (read after write) — j attempts to read the result of i prior to i writing it. Forwarding handles this common case.
2. WAW (write after write) — j writes an operand before i can complete its write. This can only occur if the architecture writes in more than one stage. DLX writes registers in only the WB stage and avoids this class of hazards.
3. WAR (write after read) — j writes an operand prior to i reading it. This is disallowed in DLX since reads occur in ID before writes in WB in this pipeline.

32 – Do Data Hazards Require Stalls?

Sadly, yes, since some the MEM/WB register is updated later in the pipeline than the EX/MEM register. Loads can therefore induce a necessary stall as seen below with register R1. Sometimes the compiler can reorder instructions to avoid the stall.

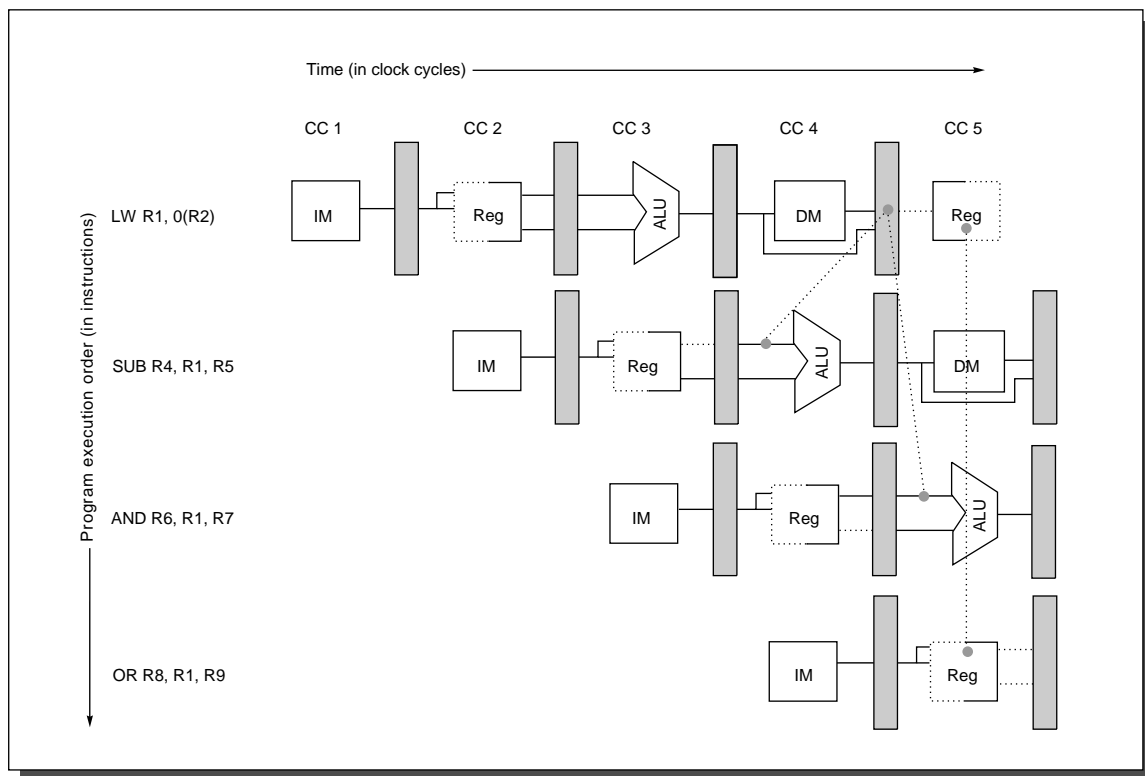


FIGURE 3.12 The load instruction can bypass its results to the AND and OR instructions, but not to the SUB, since that would mean forwarding the result in "negative time."

33 – How DLX handles Data Hazard Stalls

Loads causing a necessary stall use a *pipeline interlock* hardware to ensure synchronization.

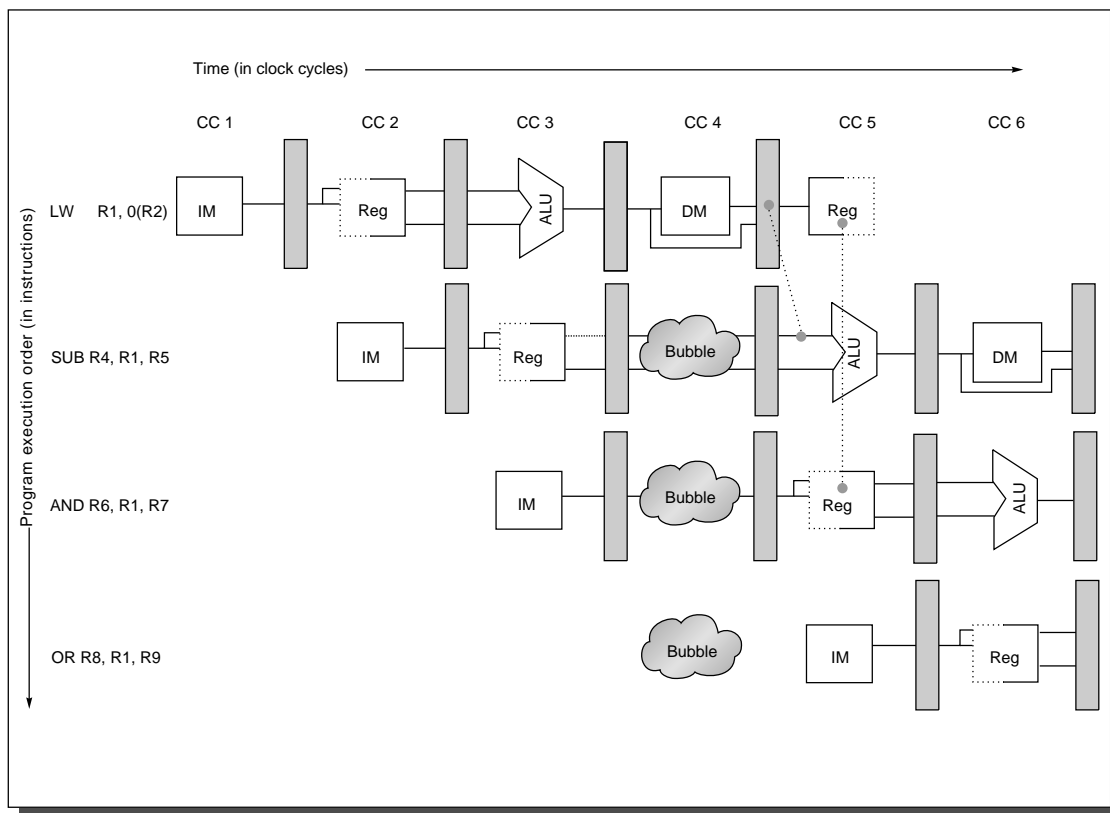


FIGURE 3.13 The load interlock causes a stall to be inserted at clock cycle 4, delaying the `SUB` instruction and those that follow by one cycle.

34 – Control Hazards

Recall that branch instructions conditionally change the PC to the target address if the branch is *taken* or if it is *not taken* the PC is sequentially advanced. Since the PC is not known, inappropriate instructions can be loaded into the pipeline.

One simple solution is to stall the pipeline as soon as a branch operation is detected (in the ID stage) until the PC is known (in the MEM stage).

This results in 3 instruction cycles being wasted as per Figure 3.21 in Hennessy and Patterson.

35 – In Class Exercise

Suppose that without branching, the pipelined CPI for DLX is 1. If 30% of the instructions are branches, what is the CPI resulting from the simple scheme above, where each branch induces a 3 clock cycle stall in the pipeline?

36 – Solution

$$\begin{aligned} \text{CPI} &= (1 - 0.3) \times \text{CPI}_{\text{no branch}} + \\ &\quad \text{Branch Stall Penalty} \times 0.3 \\ &= 0.7 \times 1 + 3 \times 0.3 = 1.6 \end{aligned} \quad (4)$$

However since instructions need to execute in whole clock cycles, we round up to $2 = \lceil 1.6 \rceil$ clocks. This means we can get only $1/2$ the speedup from ideal pipelining.

37 – Reducing the Branch Penalty

To reduce the number of clock cycles of a branch induced stall:

1. Determine if the branch is taken earlier in the pipeline.
2. Compute the taken PC earlier.

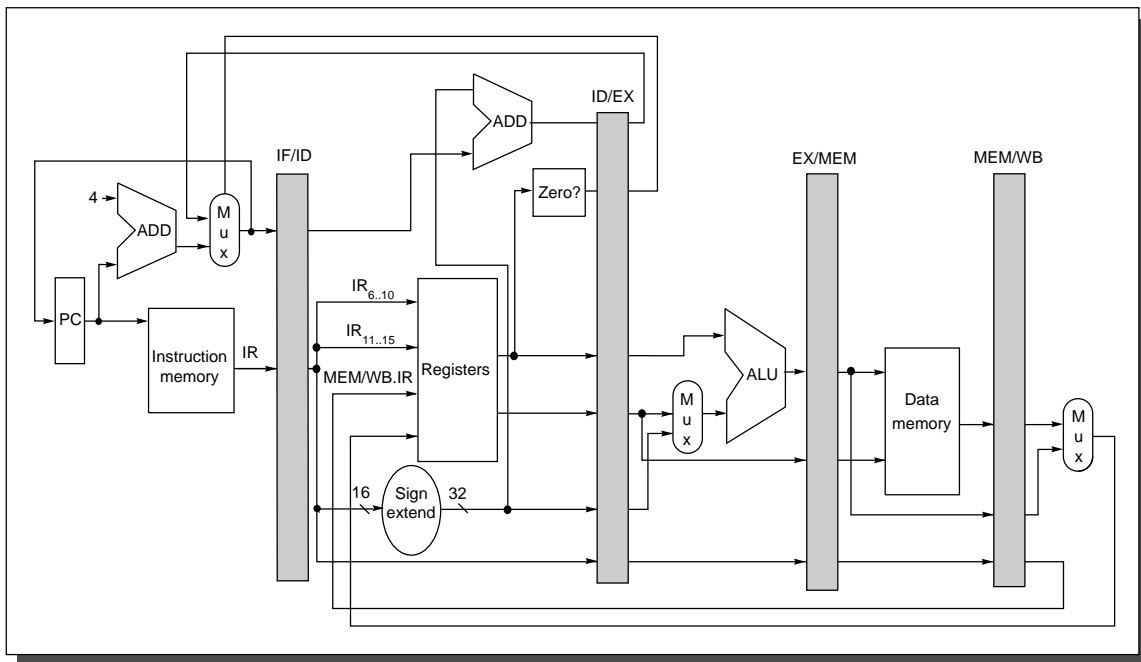


FIGURE 3.22 The stall from branch hazards can be reduced by moving the zero test and branch target calculation into the ID phase of the pipeline.

38 – Branch Behavior in Programs

Although branch behavior depends on the combination of a program and its inputs, most studies indicate that forward conditional branches are the most common case (since if/else statements are heavily used).

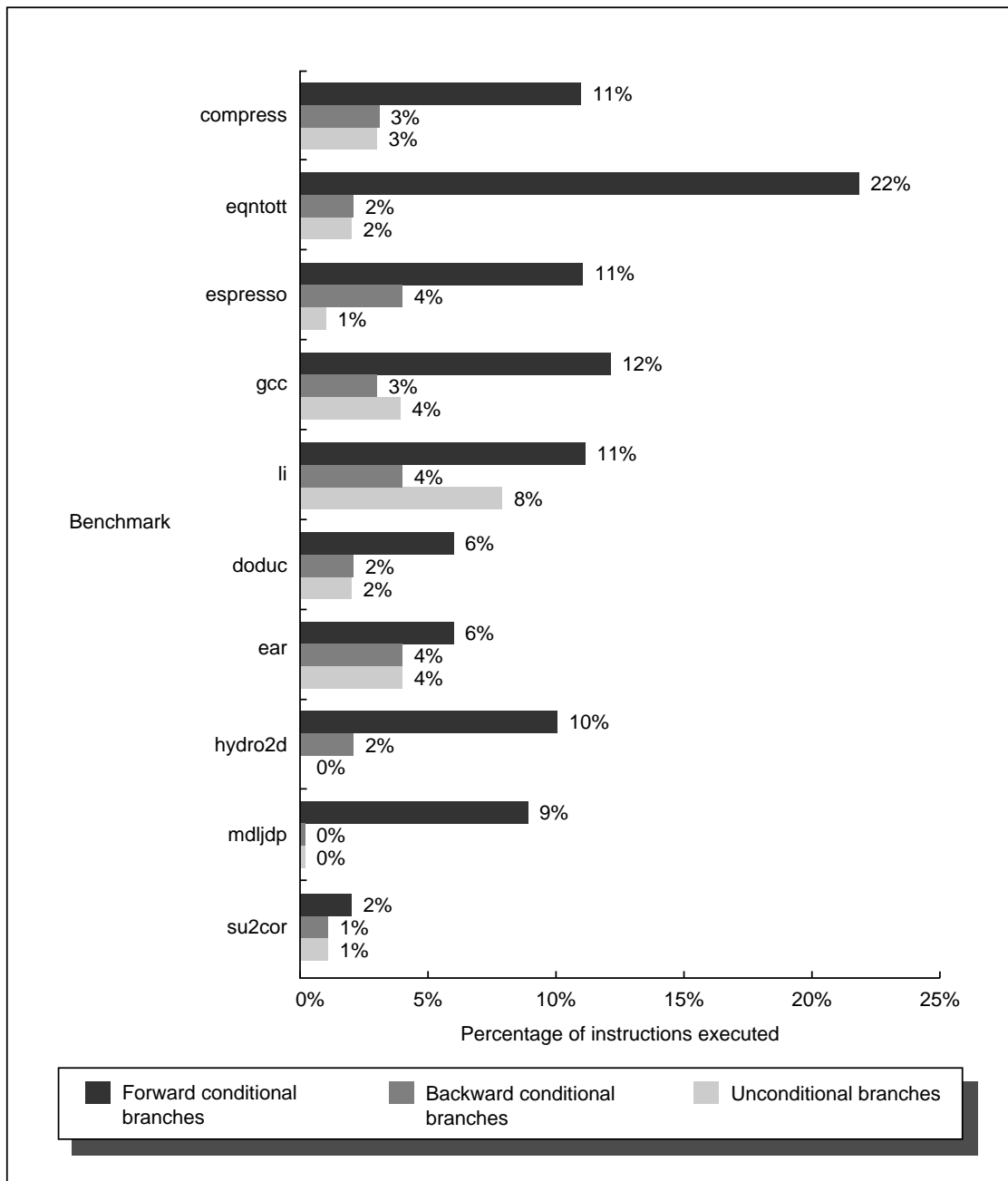


FIGURE 3.24 The frequency of instructions (branches, jumps, calls, and returns) that may change the PC.

39 – Branch Behavior in Programs

Both forward branches are more frequently taken, while backward branches tend to be not taken. This is especially true when loop unrolling optimization is done as seen below.

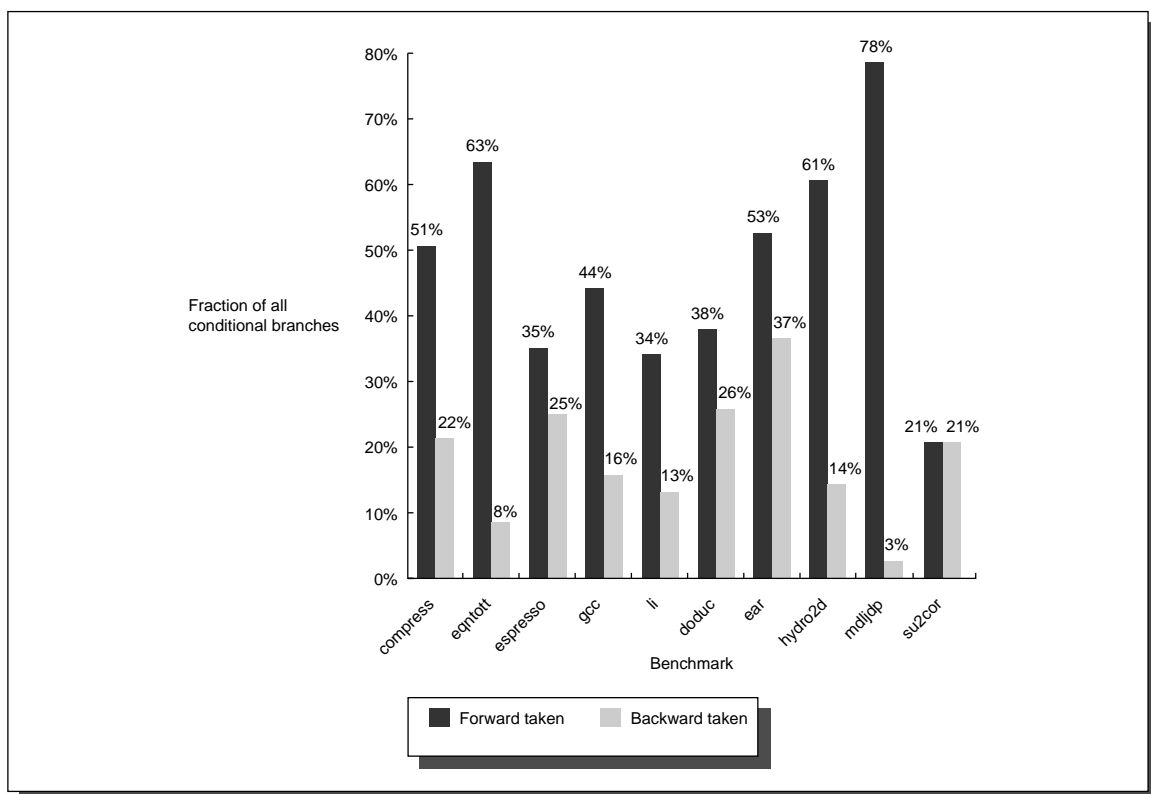


FIGURE 3.25 Together the forward and backward taken branches account for an average of 67% of all conditional branches.

40 – Reducing Pipeline Branch Penalties

The following schemes could be used:

1. Freezing or Flushing the Pipe — expensive but simple.
2. Predict-not-taken — speculatively prefetch and process instructions as if the branch was not taken and discard partial results if it was indeed taken (used by DLX).
3. Predict-taken — While desirable (especially for forward branches), the lateness at which the new PC is known in DLX means this is not a win.
4. Delayed Branch — Reorder instructions to permit the branch to be computed concurrently with preceding statements. (Done in DLX with assembler support).

41 – Using a Branch Delay Slot

To fill the delay slot an instruction is allowed to execute while the target of a branch is being computed.

Branch Not Taken	IF	ID	EX	MEM	WB				
Branch Delay		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

Branch Not Taken	IF	ID	EX	MEM	WB				
Branch Delay		IF	ID	EX	MEM	WB			
Branch Target			IF	ID	EX	MEM	WB		
Target + 1				IF	ID	EX	MEM	WB	
Target + 2					IF	ID	EX	MEM	WB

42 – Scheduling the Branch Delay Slot

Traditionally the assembler implicitly reorders the statements in machine code generation to accomodate this (unless the programmer specifically disables this).

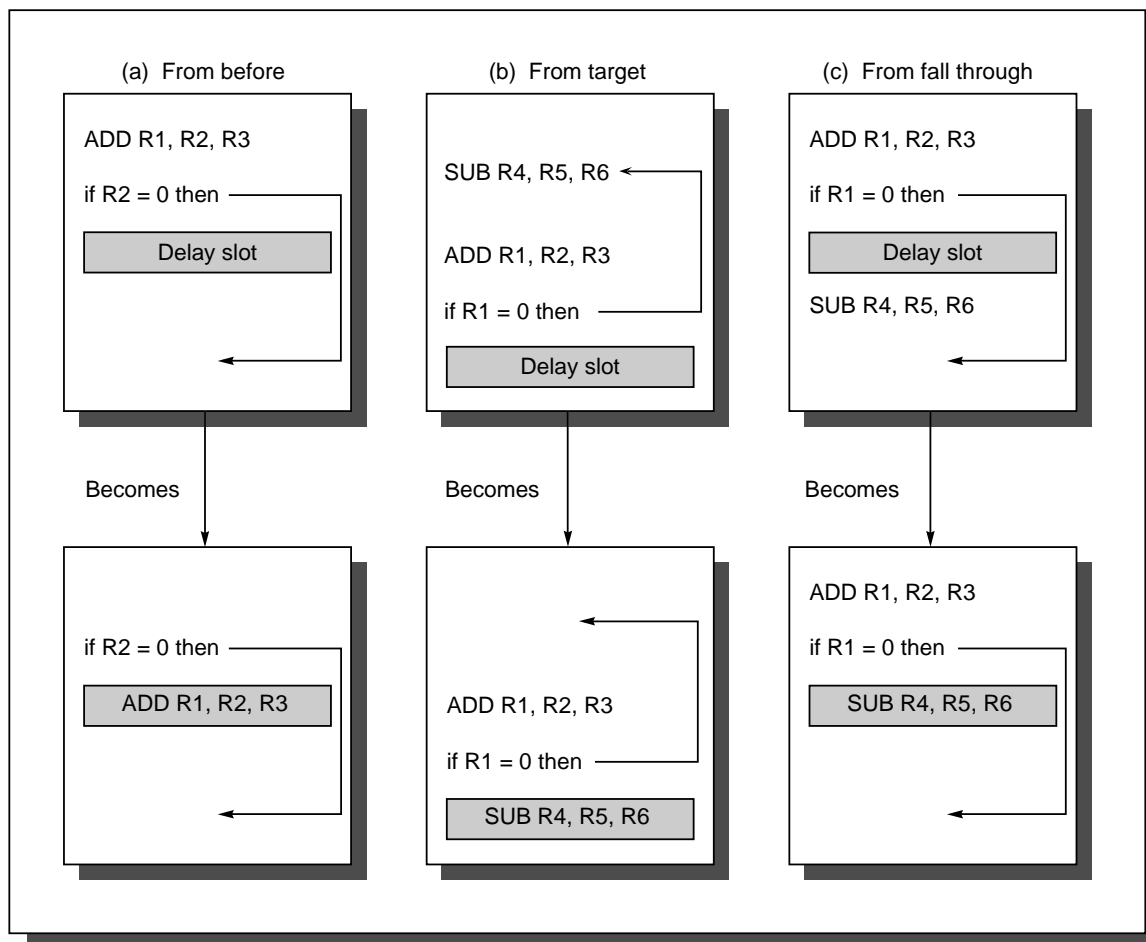


FIGURE 3.28 Scheduling the branch-delay slot.

43 – How to Choose a Branch Delay?

1. The statement prior to the branch is the best choice.
2. Otherwise, if the branch is taken with high probability (e.g. loop branches) then a statement from the target is a good candidate.
3. Otherwise, the branch is seldom taken, then a statement from the fall through becomes a good candidate

Scheduling Strategy	Requirements	When it Helps?
a) From Before Branch	Branch must not depend on instruction	always
b) From Target	Rescheduled instruction must be O.K. to execute if branch not taken. May duplicate instructions.	When branch is taken. Can expand code.
c) From fall through	Rescheduled instruction must be O.K. to execute if branch is taken.	When branch is not taken

44 – Canceling Branch Delays

Picking a branch delay may be hard if:

1. Restrictions on which instructions are suited to fill the delay slot (due to program structure).
2. Predicting if a branch is taken or not taken is hard.

Canceling a branch consists of:

1. Speculatively predicting that the branch will be taken.
2. Optimistically entering the target instruction of the branch at the predicted location in the pipeline.
3. Converting it to a no-op if the destination of the branch falls through.

45 – How Canceling a Branch can help

Canceling relaxes the requirement that the delay slot instruction be O.K. to execute if the branch does not go in the predicted direction. However, a wrong guess does get a 1 cycle penalty.

Branch Not Taken	IF	ID	EX	MEM	WB				
Branch Delay		IF	ID	idle	idle	idle			
Instruction $i + 1$			IF	ID	EX	MEM	WB		
Instruction $i + 2$				IF	ID	EX	MEM	WB	
Instruction $i + 3$					IF	ID	EX	MEM	WB

Branch Not Taken	IF	ID	EX	MEM	WB				
Branch Delay		IF	ID	EX	MEM	WB			
Branch Target + 1			IF	ID	EX	MEM	WB		
Branch Target + 2				IF	ID	EX	MEM	WB	
Branch Target + 3					IF	ID	EX	MEM	WB

46 – Notes on Branch Cancellation

Branches cannot fill delay slots due to semantic problems which result.

Most machines supporting cancelling branches have the following instructions:

1. A non-cancelling branch
2. A cancel if not taken branch (*branch likely*)

This relaxes the requirements for filling the delay slot from the branch target, but does not help with filling from fall through.

Recent increases in pipeline depth and the resulting longer branch delays combined with hardware branch prediction schemes has resulted in a demphasis on compiler support for branch prediction.

47 – Performance of Branch Schemes

When the ideal CPI is 1, the pipeline speedup with branch penalties is:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline Stalls due to branches}}$$

Pipeline Stalls due to branches = Branch frequency \times Branch Penalty

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch Penalty}}$$

Both unconditional and conditional branches contribute to the branch frequency and the branch penalty. However conditional branches are the dominant form due to high frequency. Deeper pipelines induce a larger branch penalty.

48 – In Class Exercise 1 of 2

Consider the MIPS R4000 pipeline has a total of eight stages. The R4000 requires three cycles to determine the branch target address and an additional cycle to evaluate the branch condition (assuming no stalls no the registers in the conditional comparison). The branch penalties are:

Branch Scheme	Penalty Unconditional	Penalty Untaken	Penalty Taken
Flush Pipeline	2	3	3
Predict Taken	2	3	2
Predict Untaken	2	0	3

49 – In Class Exercise 2 of 2

How many additional cycles per instruction are needed due to branch penalties (treating other penalties as negligible) for the gcc benchmark in Figure 3.24 of the book, where the frequencies are:

Branch Type	Frequency of opcode	Percent Taken
Forward Conditional Branches	12%	44%
Backward Conditional Branches	3%	16%
Unconditional Branches	4%	100%

You should try the Predict Untaken policy first, try the others if you have time.

50 – Solution 1 of 2

We will show the predict untaken case solved and leave the others as an exercise for the student (with results shown here). Letting P represent pipeline stall cycles from branches per instruction, we get:

$$P = \text{Unconditional Branch Induced Stalls} + \\ \text{Conditional Taken Branch Induced Stalls} + \\ \text{Conditional Untaken Branch Induced Stalls}$$

For each possible way of inducing a stall:

$$\text{Stalls Induced} = \text{Frequency of Stall} \times \text{Penalty of Stall}$$

51 – Solution 2 of 2

The frequency of unconditional branches is 0.04 from the table and the penalty is 2. Letting f_t denote the frequency of taking conditional branches and f_u be the frequency of untaken conditional branches in the instruction stream:

$$f_t = (0.12 \times 0.44) + (0.03 \times 0.16) = 0.0576 \approx 0.05$$

$$f_u = (0.12 \times (1 - 0.44)) + (0.03 \times (1 - 0.16)) = 0.0924 \approx 0.09$$

Substituting back in we get:

$$P = (0.04 \times 2) + (0.09 \times 0) + (0.05 \times 3) = 0.23$$

52 – Static Branch Prediction

If we know a bit about program execution we can predict if branches are taken or untaken at compile time. This technique is called *static branch prediction*.

1. Examine General Program Behavior
 - (a) Most branches are taken, so predict taken.
 - (b) Since loops tend to iterate, predict backward branches taken and forward branches untaken.
2. Profiler Based Branching — use a profiler to improve branch prediction. This is superior because a particular branch may have a high probability of being taken, while another branch may have a high probability of not being taken.

53 – Profiler Based Static Branch Prediction

We see that the misprediction rates for profiler based predictors tends to be small.

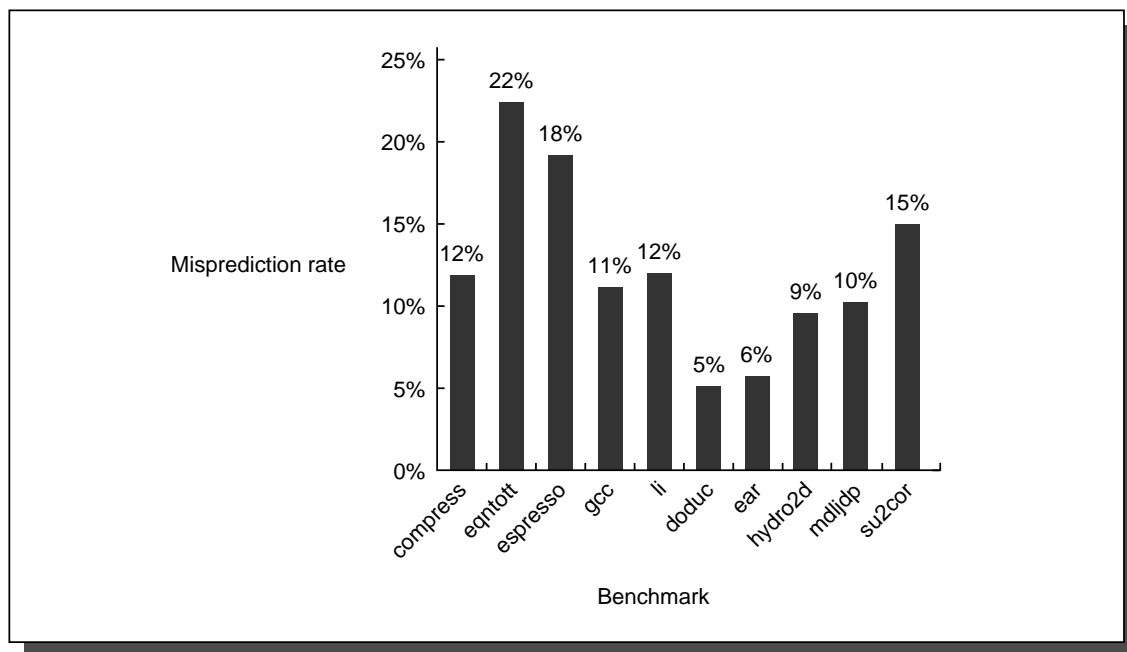


FIGURE 3.36 Misprediction rate for a profile-based predictor varies widely but is generally better for the FP programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%.

54 – Static Branch Prediction Performance

Profiler based prediction strategy tends to be more efficient than a predict taken strategy.

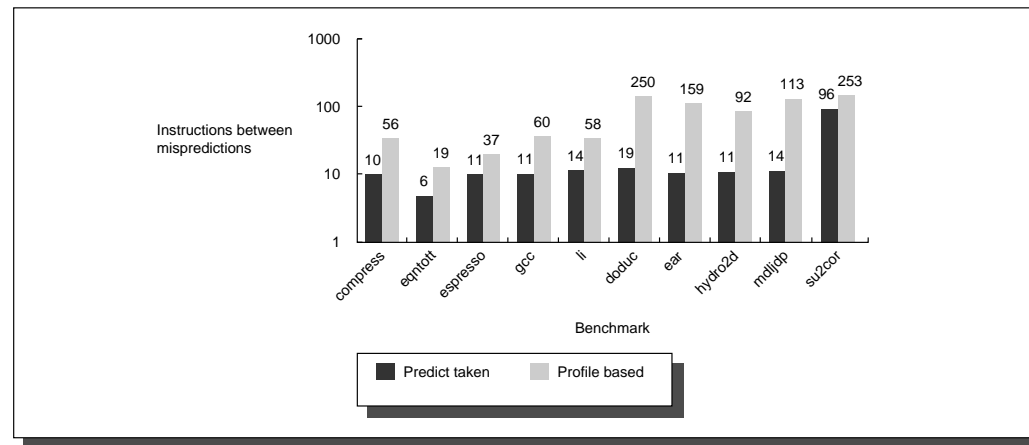


FIGURE 3.37 Accuracy of a predict-taken strategy and a profile-based predictor as measured by the number of instructions executed between mispredicted branches and shown on a log scale.

55 – Exceptions and Pipelining

Exceptions make pipelining hard to implement, some common exception causes are:

1. I/O device request
2. System Call
3. Tracing Program Execution
4. Integer arithmetic overflow/underflow
5. FP arithmetic error
6. Page Fault
7. Data misalignment in memory
8. Invalid Instruction
9. Hardware failure
10. Power failure

People sometimes use the words interrupt or fault (but the usage is not standardized).

56 – Exception Handling Issues

How an exception should be handled depends on the cause. Some issues in handling exceptions include:

1. Is is *synchronous* (i.e. triggered by the program) or *asynchronous* (i.e. triggered by an external event)?

Asynchronous exceptions can usually be processed after instruction completion (and tend to be easier).

2. User requested vs. coerced — coerced are harder because they are unpredictable.
3. User maskable vs. nonmaskable — *masking* an exception disables hardware response to it.
4. Within vs. Between instructions — Exceptions within instructions are synchronous and tend to be harder to implement (due to instruction restart).
5. Resume vs. Terminate — It is easier to implement exceptions terminating program

execution, since resuming execution means restarting stalled instructions.

57 – Stopping and Restarting Execution

When partially evaluated instructions are stalled pending exception handling, the instruction must be *restarted* to run properly.

The most difficult exceptions:

1. Occur within instructions (i.e. in DLX in the EX or MEM stages).
2. Are restartable.

e.g. a “page fault”.

58 – Exception Handling Issues

This means the pipeline must be safely shut down and its state saved during the exception handling by:

1. Force a trap instruction into the pipeline on the next IF.
2. Until the trap is taken, disable all writes for faulting instructions. This is done by putting 0's in the latches accessed by faulting instructions (but not for their predecessors).
3. After the exception handler receives control, it saves the PC of the faulting instruction. This is used to return control from the exception handler. If the instruction is a delayed branch, the PC's of the instructions filling the delay slot also need to be preserved.

59 – Restarting After Shutdown

Special instructions are issued to restart the pipeline after the exception by reloading the PCs and restarting the instruction stream.

Precise Exceptions permit instructions prior to the fault to complete, and those after it to be started from scratch.

Precise exceptions tend to be slower because of pipeline reloading, so some (newer) machines have two modes, one supporting precise exceptions the other not.

Precise exceptions are required for:

1. Debugging
2. Page Faults
3. IEEE FP arithmetic traps.

60 – DLX Exceptions

In a pipelined system, multiple faults can occur during the same clock cycle due to overlapped execution.

Pipeline Stage	Exception Types
IF	Page Fault on Instruction Misaligned Memory Access Memory Protection Violation
ID	Illegal Opcode
EX	Arithmetic Exception
MEM	Page Fault on Data Misaligned Memory Access Memory Protection Violation
WB	None

61 – Instruction Set Problems

A *committed* instruction which is guaranteed to complete. In DLX instructions reaching the MEM (or WB) stage are committed.

Autoincrement addressing modes for registers (e.g. the VAX) make restarting an instruction hard (since more state needs to be saved and restored).

Block memory operations (e.g. Pentium and VAX) may induce a large number of faults during the instruction. The Pentium uses registers as working storage, so preserving them suffices. On the VAX, memory is used so much more state information needs to be saved.

Machines with a status word (e.g. Pentium) must reset to the status word to its proper values. This could mean replaying instructions.

62 – Handling Multicycle Stages

Our prior DLX model assumed every pipeline instruction takes a single cycle. This is not realistic (e.g. for floating point) operations. (The loops indicate more than one stage is required).

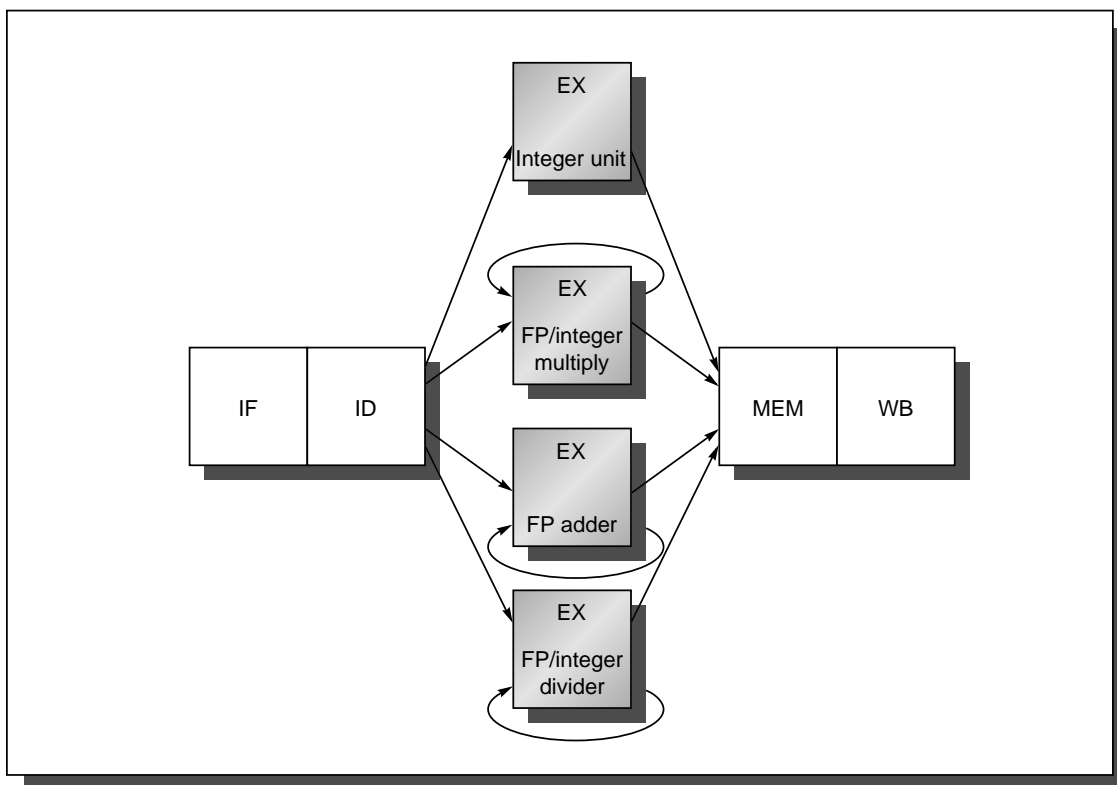


FIGURE 3.42 The DLX pipeline with three additional unpipelined, floating-point, functional units.

63 – Functional Units in DLX

(For now) assume that DLX has 4 functional units (FU):

1. The integer FU does loads/stores, ALU operations, and branches.
2. The FP and integer multiplier.
3. The FP adder does add, subtract, and conversion.
4. FP and integer divider (the slowest unit).

64 – Functional Unit Performance

Latency refers how many cycles must separate an instruction computing a result and the instruction using the result.

Initiation Interval (II) refers to how long the interval remains utilized once an instruction has entered it.

FU	Latency	II
Integer ALU	0	1
Data Memory	1	1
FP Adder	3	1
Multiplier (FP and Integer)	6	1
Divider (FP sqrt)	24	24

65 – Overlapping Pipelines

Some of the FUs are pipelined, and if a pipeline not stalled, another operation can enter it. The divider is unpipelined and needs 25 cycles.

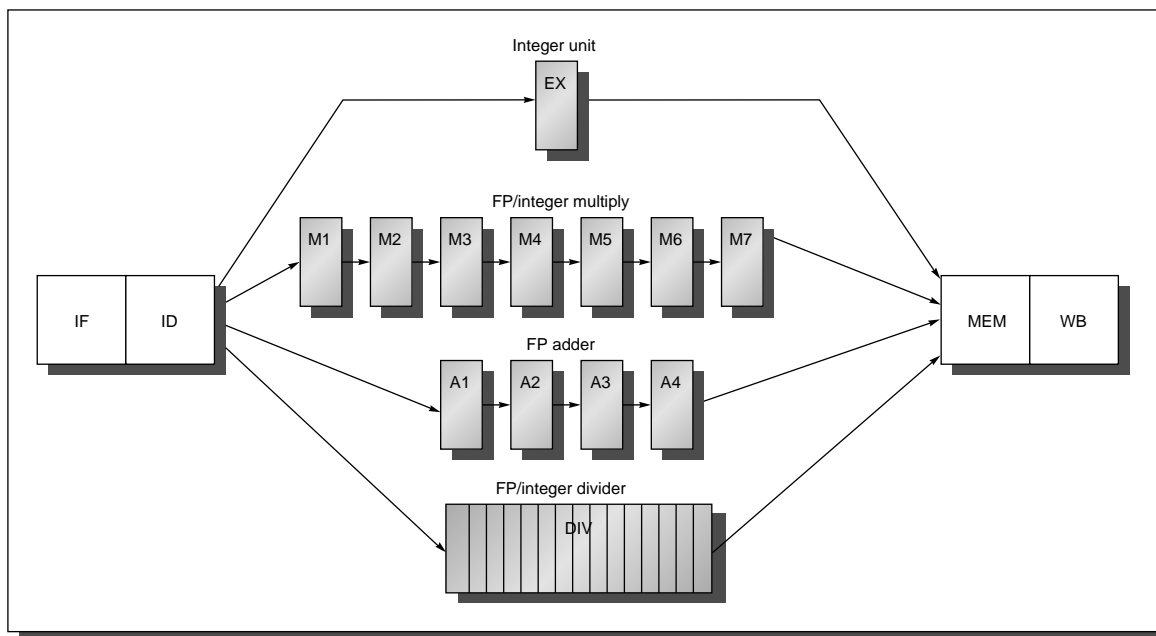


FIGURE 3.44 A pipeline that supports multiple outstanding FP operations.

66 – Handling Hazards

The pipeline interlock stalls instructions in the ID stage (which is simpler than the alternative of halting them at entry into MEM).

Additional hazard handling is required:

1. Check for structural hazards — Wait until the required FU is not busy and ensure availability of the register write port.
2. Check for a RAW data hazard — Wait until source registers are not listed as pending destinations of a pipelined operation.
3. Check for a WAW data hazard — Determine if an instruction in the A1,...,A4, D, or M1,...,M7 has the same destination register as this instruction.

67 – Maintaining Precise Exceptions

Maintaining precise exceptions becomes hard because instructions can complete out of order.

To ensure precise exceptions either:

1. Additional special instructions must be issued to check for exceptions, or a special processor mode must be set.
2. Buffer the results of an operation until all operations issued earlier complete. Two approaches to this involve keeping one of the following:
 - (a) *History file* of the previous values of registers, when an exception occurs, (if needed) roll them back prior to the state of some out of order completions.
 - (b) *Future File* — Store results of early completions in the future file, and when the late completion is finished, update the user visible register file.
3. Allow Exceptions to become imprecise but

keep enough data such that trap-handling routines can create a precise sequence for the exception.

4. Hybrid — Use special instructions to indicate if all instructions before the issuing instruction are guaranteed to complete without an exception. Otherwise stalls may be needed to ensure precise exceptions.

68 – The Moral of the story

We learned some lessons along the way:

1. Variable length instructions and running time can imbalance the pipeline and back it up. Also hazard detection and maintaining precise exceptions becomes hard.
2. Sophisticated Address modes make restarts and hazard detection hard. Multiple memory access instructions make it tough to pipeline smoothly.
3. Architectures supporting self modifying code (e.g. Intel 80x86) can make pipelining and cache management tough.
4. Implicitly set condition codes make it more difficult to determine if a branch has been decided, and complicate scheduling branch delays.

References