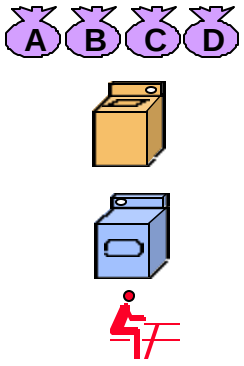
Lecture 2: Review of Pipelines

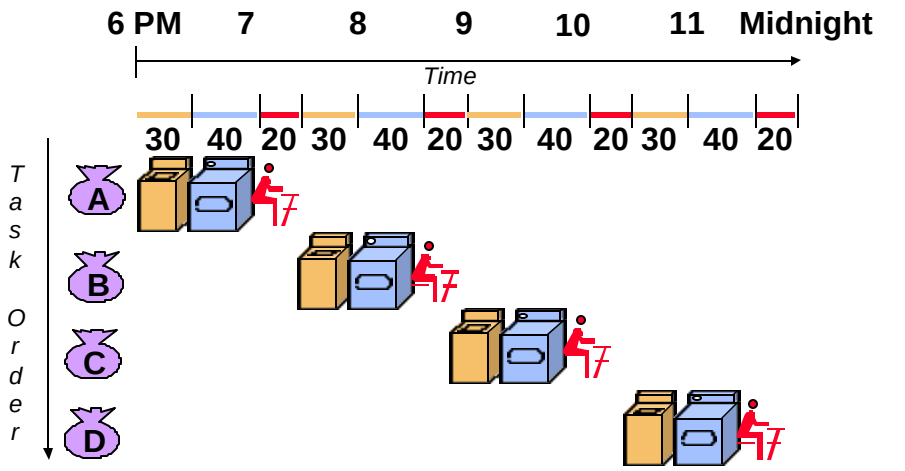
Prof. David A. Patterson
Modifié par M. Aboulhamid

Pipelining: Its Natural!

- **Laundry Example**
- **Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold**
- **Washer takes 30 minutes**
- **Dryer takes 40 minutes**
- **"Folder" takes 20 minutes**

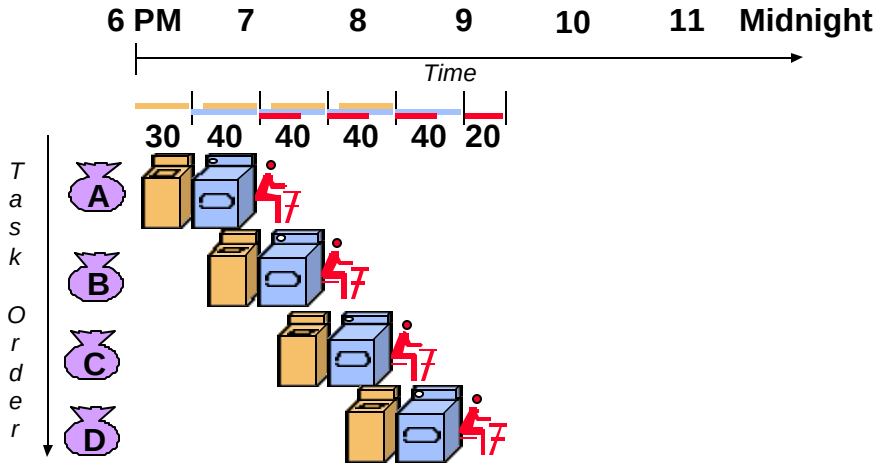


Sequential Laundry



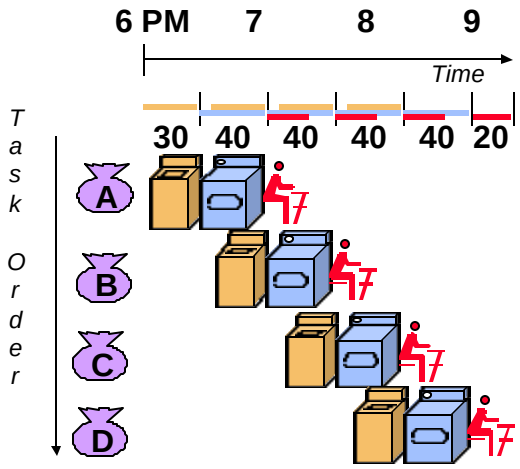
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry
Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

Computer Pipelines

- Execute billions of instructions, so throughput is what matters
- DLX desirable features: all instructions same length, registers located in same place in instruction format, memory operands only in loads or stores

+ N'est pas visible au programmeur

5 Steps of DLX Datapath

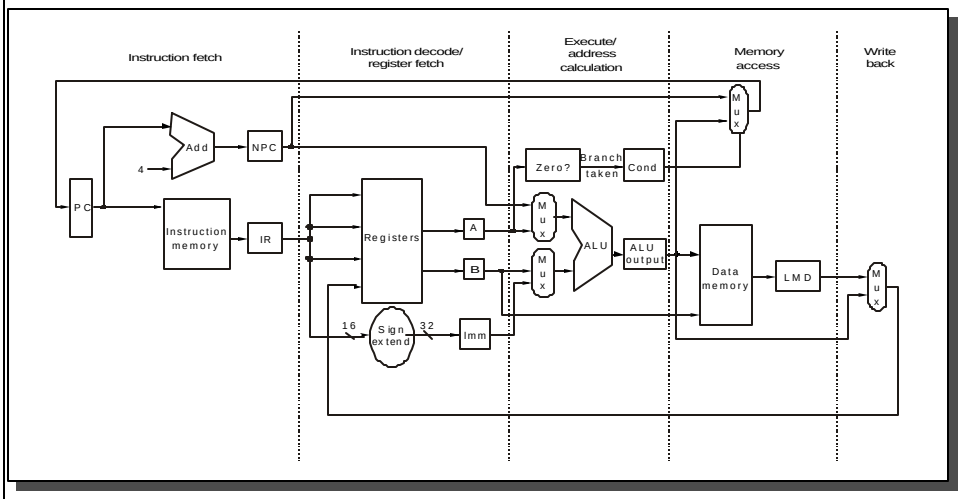


FIGURE 3.1 The implementation of the DLX datapath allows every instruction to be executed in four or five clock cycles.

Steps 1 & 2

- **IF - instruction fetch step**

  IR <-- Mem[PC]: fetch the next instruction from memory

  NPC <-- PC + 4 : compute the new PC
- **• done in parallel with opcode decode**
- **ID - instruction decode and register fetch step**
  - **A <-- Regs[IR 6.. 10]**
  - **B <-- Regs[IR 11.. 16]**
- **• Possible since register specifiers are encoded in *fixed fields***
- **• We may fetch register contents that we don't use but OK since**
- **the operands will be ready if the opcode is of the type that does use**
- **them**
- **• Also calculate the sign extended immediate in case that's the**
- **value that the opcode needs**

Pipelined DLX Datapath

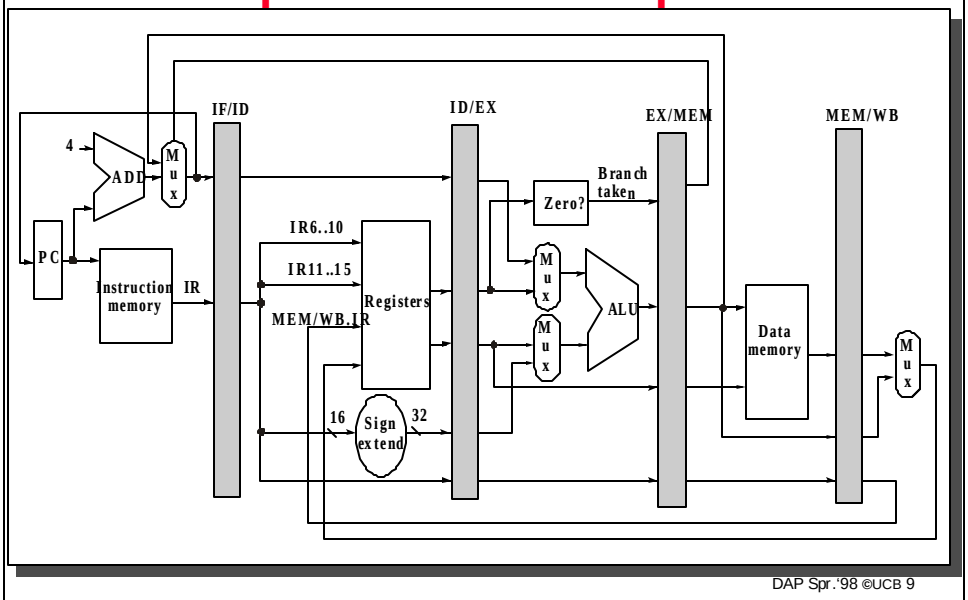


FIGURE 3.4 The datapath is pipelined by adding a set of registers, one between each pair of pipe stages.

Visualizing Pipelining

Figure 3.3, Page 133

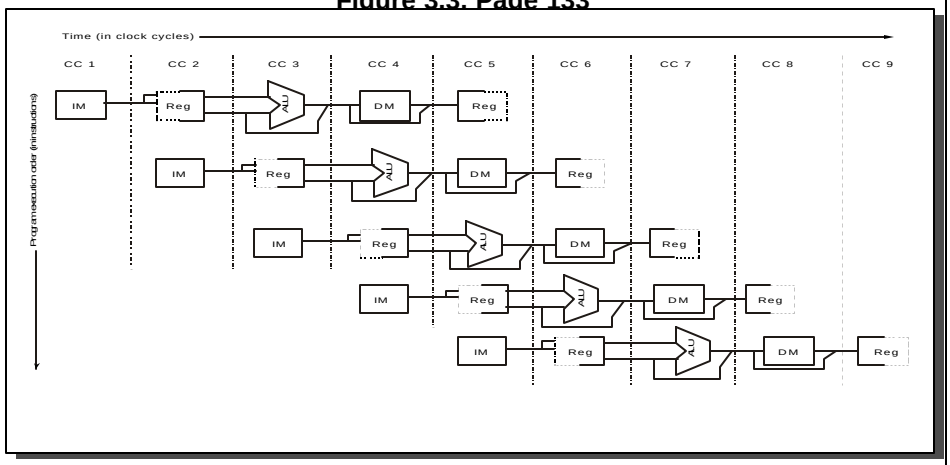


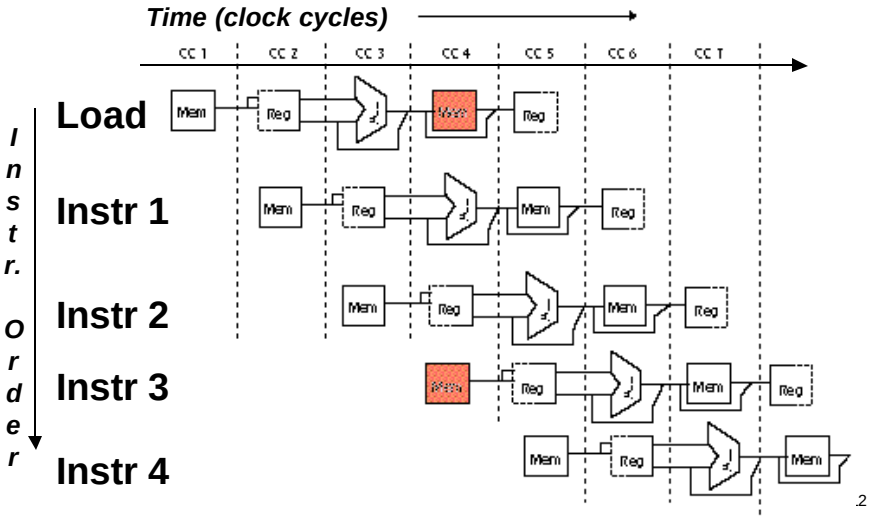
FIGURE 3.3 The pipeline can be thought of as a series of datapaths shifted in time.

Its Not That Easy for Computers

- **Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle**
  - **Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)**
  - **Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)**
  - **Control hazards: Pipelining of branches & other instructions that change the PC**
  - **Common solution is to stall the pipeline until the hazard is resolved, inserting one or more "bubbles" in the pipeline**

One Memory Port/Structural Hazards

Figure 3.6, Page 142



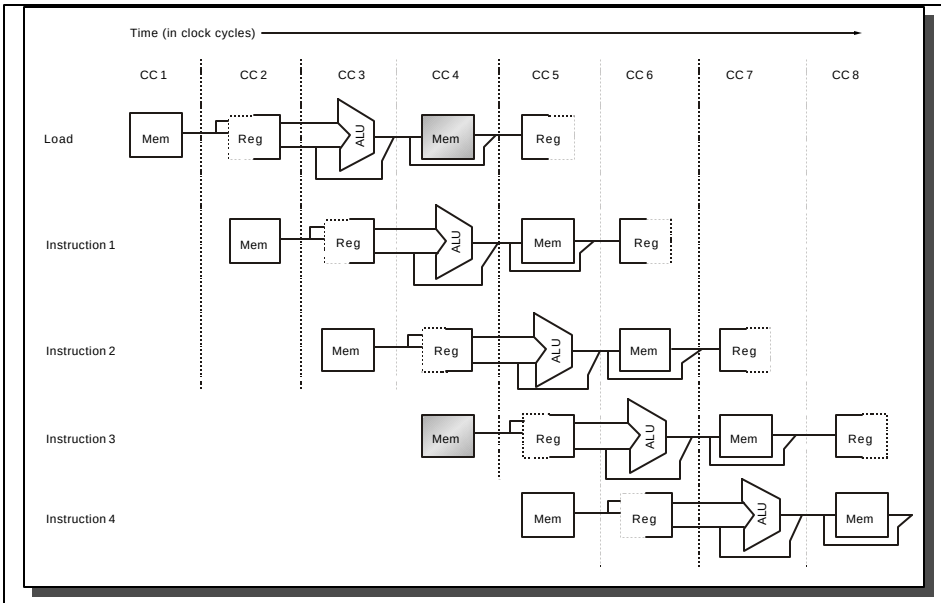
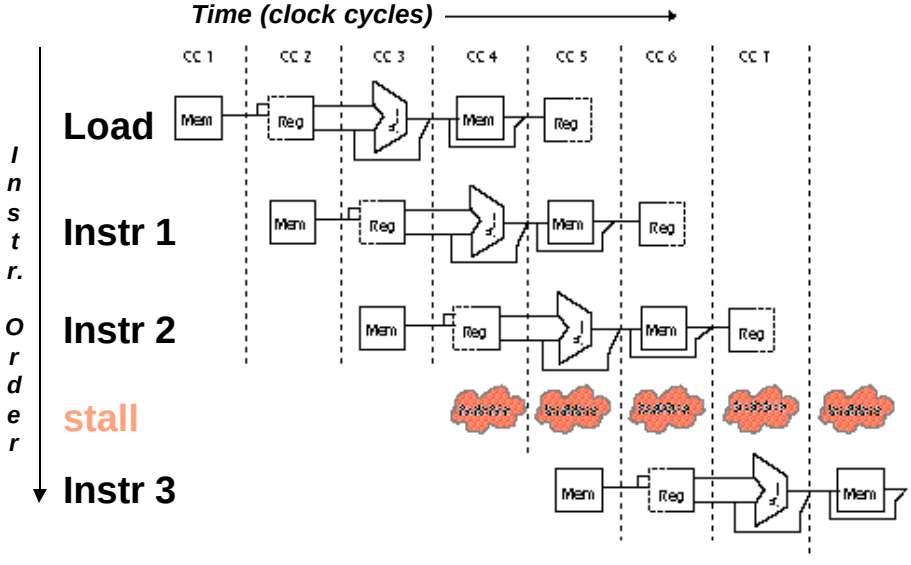


FIGURE 3.6 A machine with only one memory port will generate a conflict whenever a memory reference occurs.

DAP Spr.'98 ©UCB 13

One Memory Port/Structural Hazards

Figure 3.7, Page 143



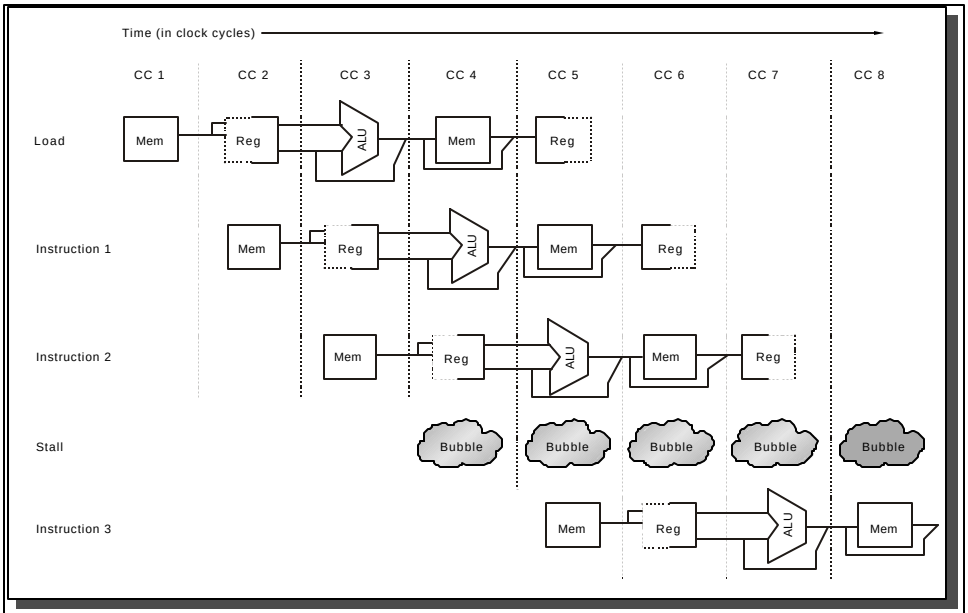


FIGURE 3.7 The structural hazard causes pipeline bubbles to be inserted.

Speed Up Equation for Pipelining

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

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle}_{unpipelined}}{\text{Clock Cycle}_{pipelined}}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle}_{unpipelined}}{\text{Clock Cycle}_{pipelined}}$$

Example: Dual-port vs. Single-port

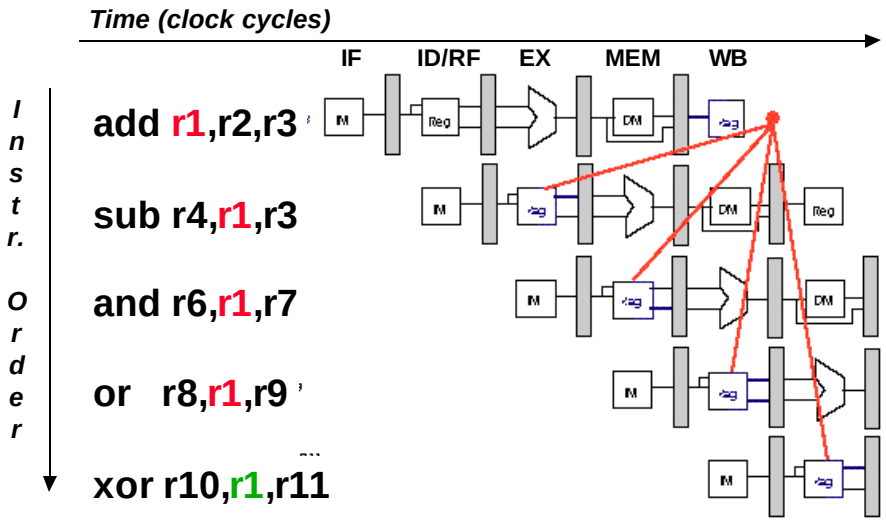
- Machine A: Dual ported memory
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

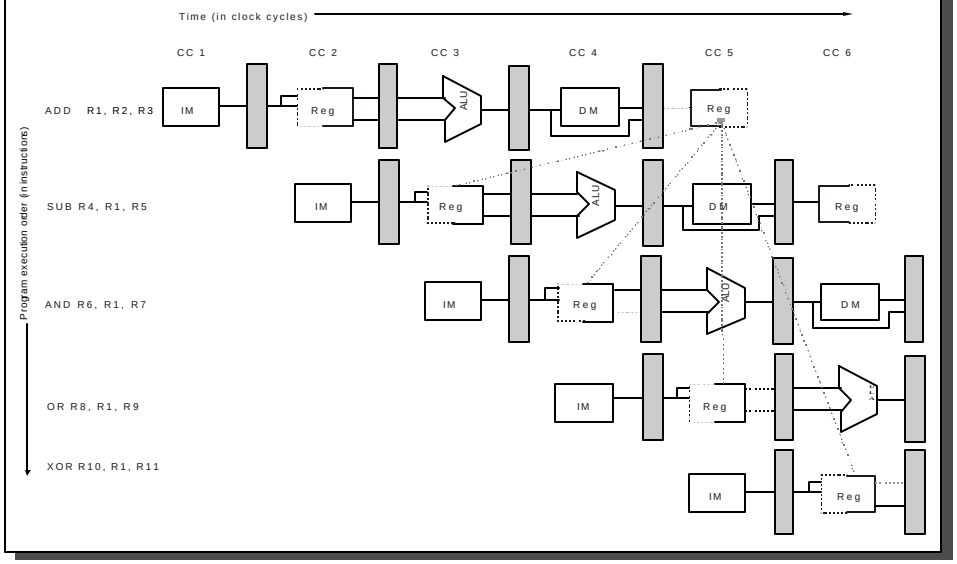
$$SpeedUp_A = Pipeline\ Depth/(1 + 0) \times (clock_{unpipe}/clock_{pipe})$$
$$= Pipeline\ Depth$$
$$SpeedUp_B = Pipeline\ Depth/(1 + 0.4 \times 1) \times (clock_{unpipe}/(clock_{unpipe} / 1.05)$$
$$= (Pipeline\ Depth/1.4) \times 1.05$$
$$= 0.75 \times Pipeline\ Depth$$
$$SpeedUp_A / SpeedUp_B = Pipeline\ Depth/(0.75 \times Pipeline\ Depth) = 1.33$$

- Machine A is 1.33 times faster

Data Hazard on R1

Figure 3.9, page 147





register is not written until after those instructions read it.

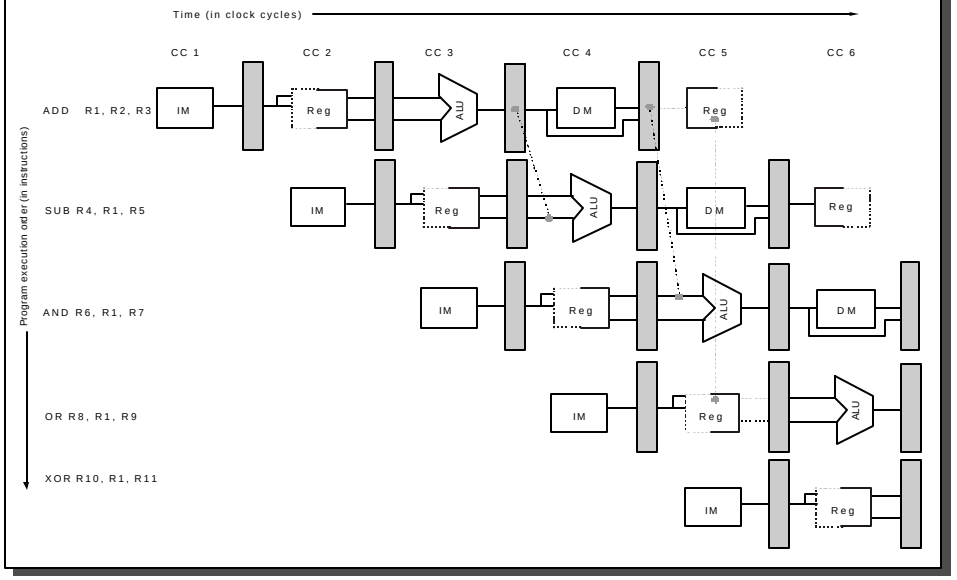


FIGURE 3.10 A set of instructions that depend on the ADD **result use forwarding paths to avoid the data hazard.**

Three Generic Data Hazards

$Instr_I$ followed by $Instr_J$

- **Read After Write (RAW)**
  $Instr_J$ tries to read operand before $Instr_I$ writes it

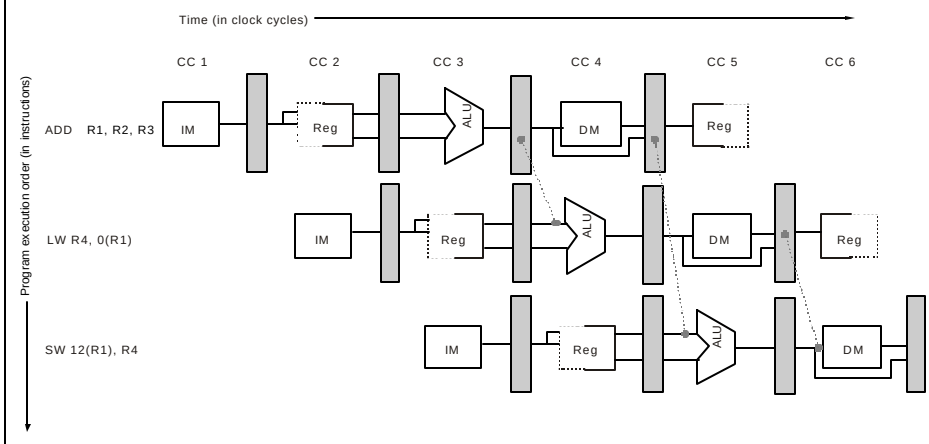


FIGURE 3.11 Stores require an operand during MEM, and forwarding of that operand is shown here.

Three Generic Data Hazards

$Instr_I$ followed by $Instr_J$

- **Write After Read (WAR)**
  $Instr_J$ tries to write operand *before* $Instr_I$ reads i
  - **Gets wrong operand**
- **Can't happen in DLX 5 stage pipeline because:**
  - **All instructions take 5 stages, and**
  - **Reads are always in stage 2, and**
  - **Writes are always in stage 5**

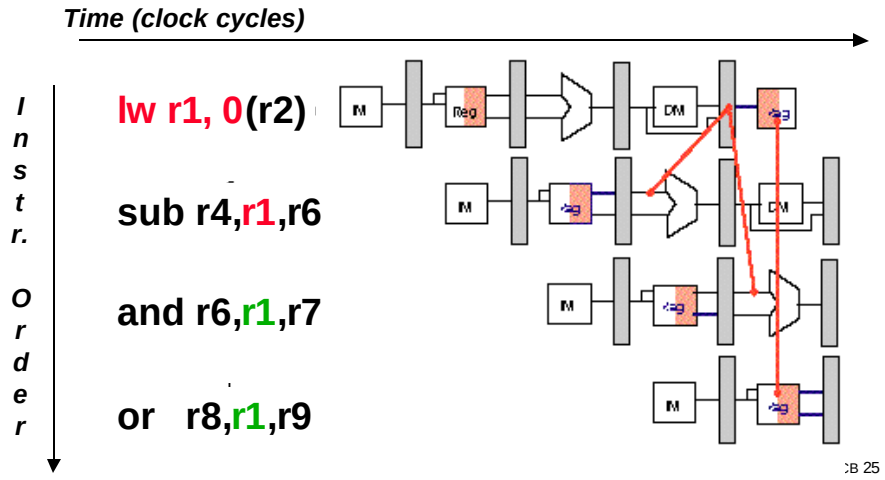
Three Generic Data Hazards

$Instr_I$ followed by $Instr_J$

- **Write After Write (WAW)**
  $Instr_J$ tries to write operand *before* $Instr_I$ writes it
  - **Leaves wrong result ($Instr_I$ not $Instr_J$)**
- **Can't happen in DLX 5 stage pipeline because:**
  - **All instructions take 5 stages, and**
  - **Writes are always in stage 5**
- **Will see WAR and WAW in later more complicated pipes**

Data Hazard Even with Forwarding

Figure 3.12, Page 153



CB 25

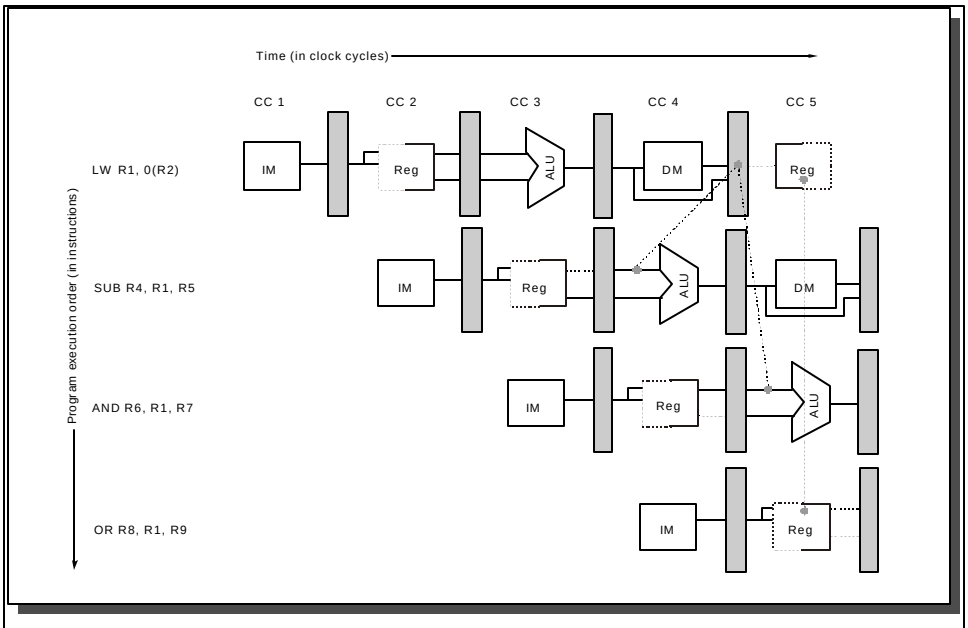
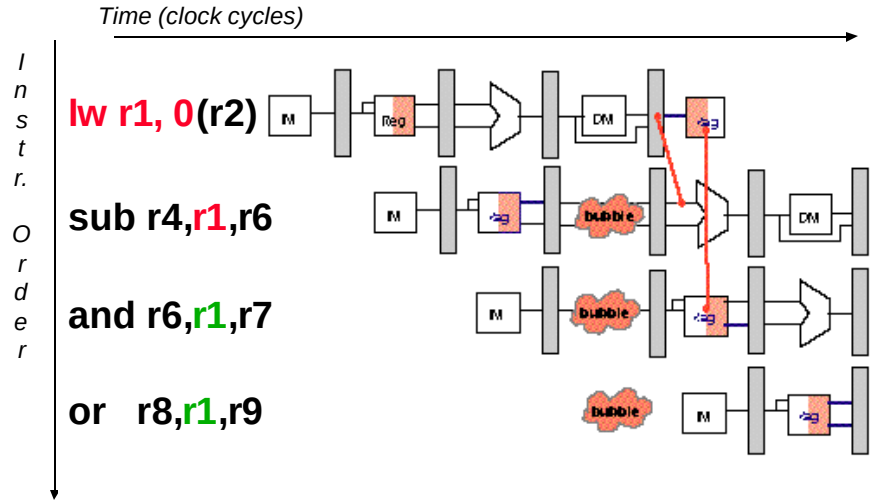


FIGURE 3.12 The load instruction can bypass its results to the AND and OR instructions, but not to the SUB, since that would mean forwarding the result in "negative time."

DAP Spr.'98 ©UCB 26

Data Hazard Even with Forwarding

Figure 3.13, Page 154



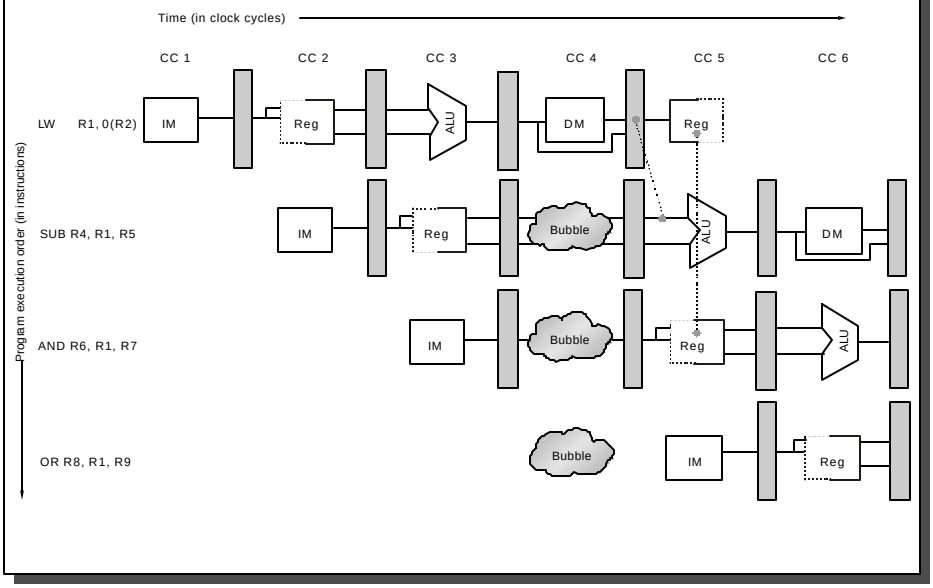


FIGURE 3.13 The load interlock causes a stall to be inserted at clock cycle 4, delaying the SUB instruction and those that follow by one cycle.

A = B + C

lw rb,b	IF	ID	EX	MEM	WB			
lw rc, c		IF	ID	EX	MEM	WB		
add ra,rb,rc			IF	ID	Cale	EX	MEM	WB
sw a, ra				IF	Cale	ID	EX	MEM	WB

Software Scheduling to Avoid Load Hazards

Try producing fast code for

a = b + c;

d = e – f;

assuming a, b, c, d ,e, and f in memory.

Slow code:

```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

Fast code:

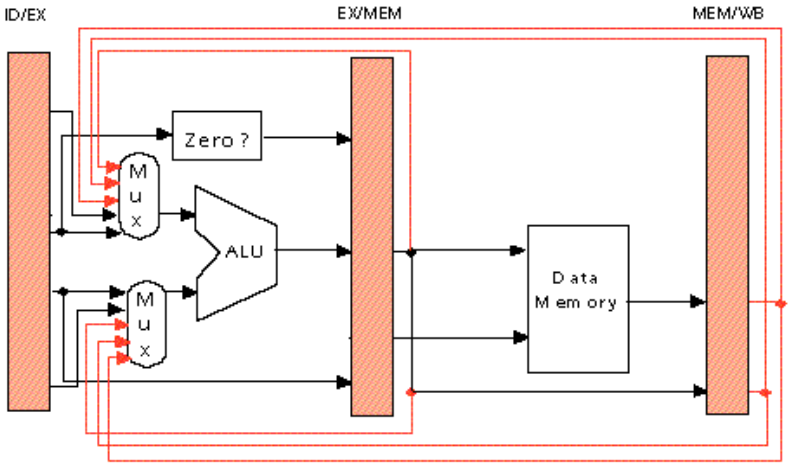
```
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
```

A = B + C; D = E + F

lw rb,b	IF	ID	EX	MEM	WB				
lw rc, c		IF	ID	EX	MEM	WB			
Lw re, e			IF	ID	EX	MEM	WB		
add ra,rb,rc				IF	ID	EX	MEM	WB	
Lw rf, f					IF	ID	EX	MEM	WB
Sw a, ra						IF	ID	EX	MEM

HW Change for Forwarding

Figure 3.20, Page 161



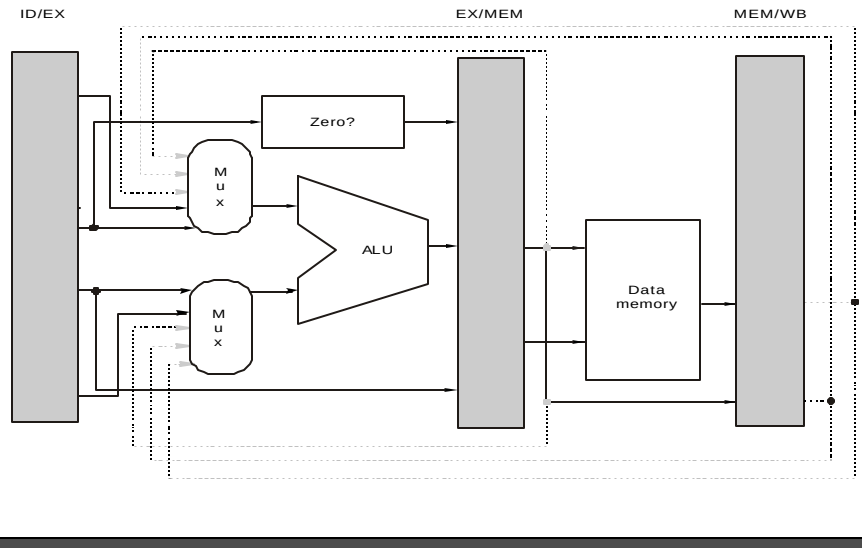
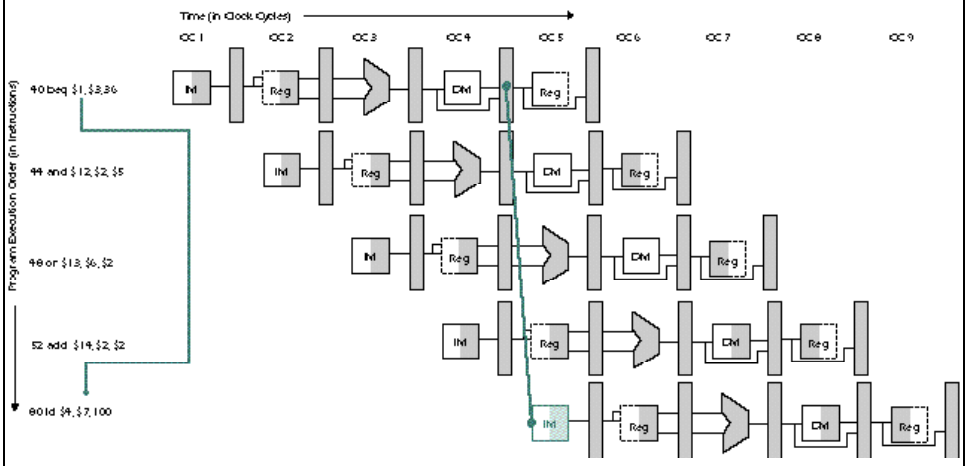


FIGURE 3.20 Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs.

Control Hazard on Branches Three Stage Stall



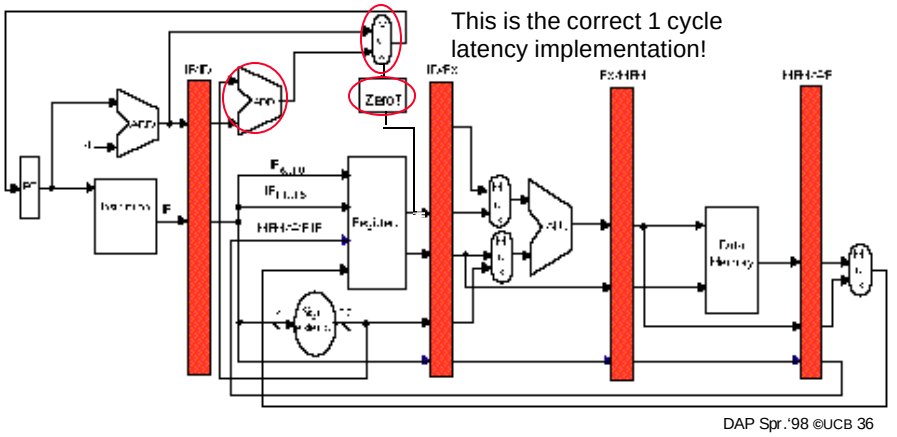
Branch Stall Impact

- If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- DLX branch tests if register = 0 or not 0
- DLX Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Pipelined DLX Datapath

Figure 3.22, page 163

Instruction Fetch	Instr. Decode Reg. Fetch	Execute Addr. Calc.	Memory Access	Write Back



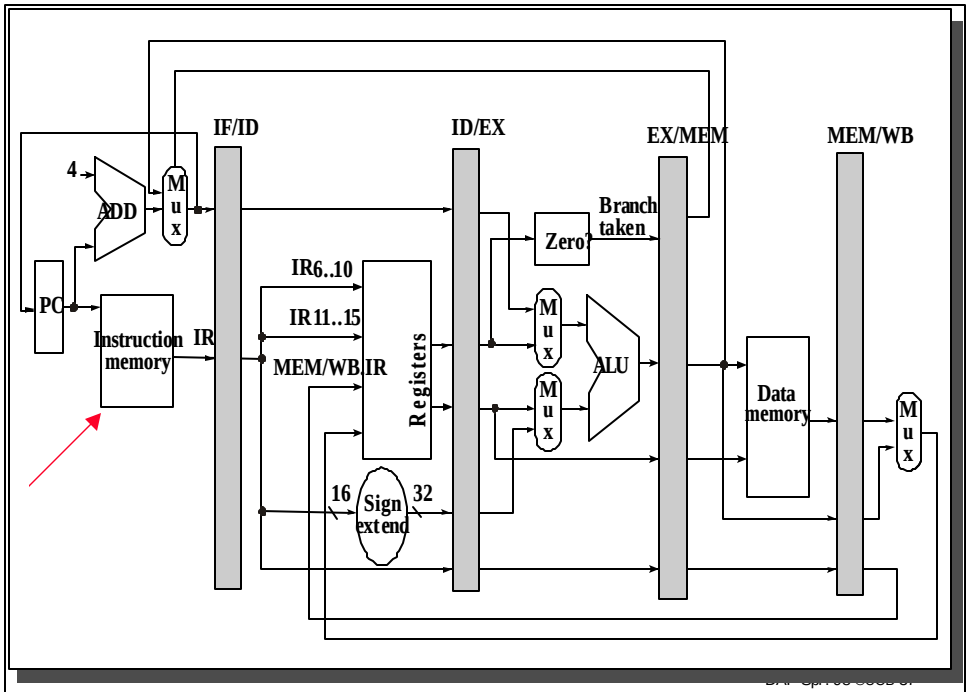


FIGURE 3.4 The datapath is pipelined by adding a set of registers, one between each pair of pipe stages.

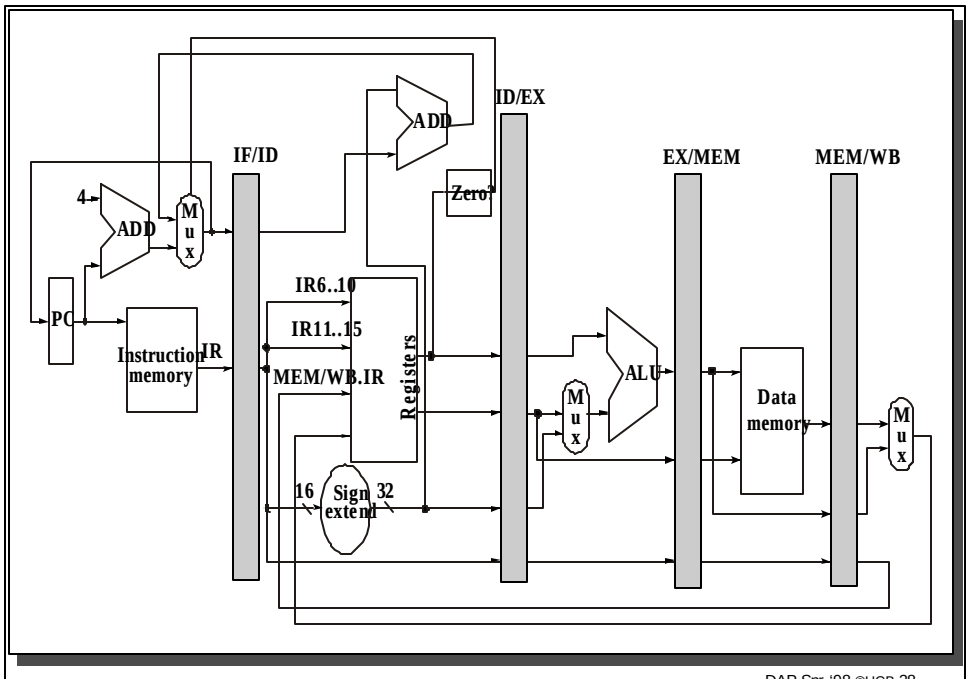


FIGURE 3.22 The stall from branch hazards can be reduced by moving the zero test and branch target calculation into the ID phase of the pipeline.

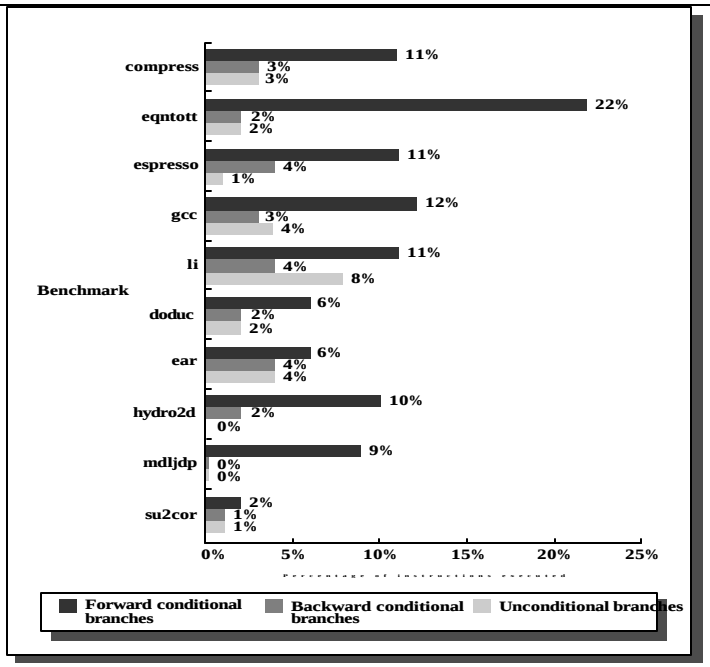


FIGURE 3.24 The frequency of instructions (branches, jumps, calls, and returns) that may change the PC.

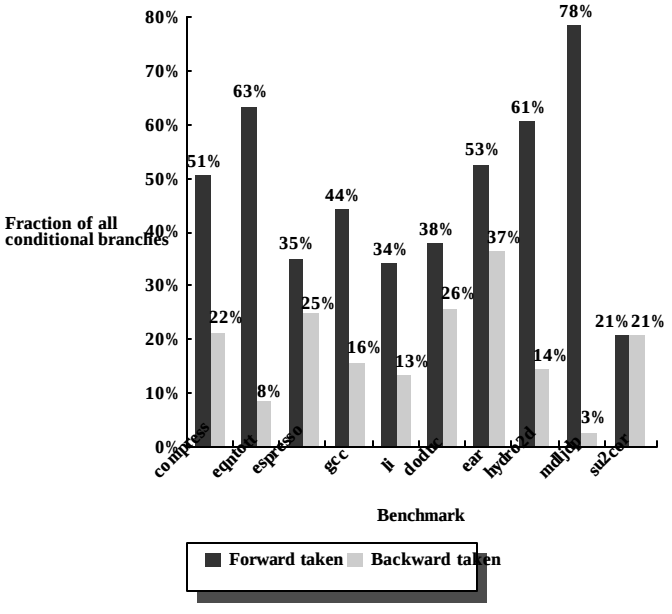


FIGURE 3.25 Together the forward and backward taken branches account for an average of 67% of all conditional branches.

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% DLX branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

- 53% DLX branches taken on average
- But haven't calculated branch target address in DLX
  - » DLX still incurs 1 cycle branch penalty
  - » Other machines: branch target known before outcome

Four Branch Hazard Alternatives

#4: Delayed Branch

- Define branch to take place AFTER a following instruction

```
branch instruction
  sequential successor1
  sequential successor2
  ........
  sequential successorn
branch target if taken
```

Branch delay of length *n*

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- DLX uses this

Delayed Branch

- **Where to get instructions to fill branch delay slot?**
  - **Before branch instruction**
  - **From the target address: only valuable when branch taken**
  - **From fall through: only valuable when branch not taken**
  - **Cancelling branches allow more slots to be filled**
- **Compiler effectiveness for single branch delay slot:**
  - **Fills about 60% of branch delay slots**
  - **About 80% of instructions executed in branch delay slots useful in computation**
  - **About 50% (60% x 80%) of slots usefully filled**
- **Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)**

Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. unpipelined*	*speedup v. stall*
Stall pipeline	**3**	**1.42**	**3.5**	**1.0**
Predict taken	**1**	**1.14**	**4.4**	**1.26**
Predict not taken	**1**	**1.09**	**4.5**	**1.29**
Delayed branch	**0.5**	**1.07**	**4.6**	**1.31**

Conditional & Unconditional = 14%, 65% change PC

Pipelining Introduction Summary

- **Just overlap tasks, and easy if tasks are independent**
- **Speed Up → Pipeline Depth; if ideal CPI is 1, then:**

$$\text{Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle Unpipelined}}{\text{Clock Cycle Pipelined}}$$

- **Hazards limit performance on computers:**
  - **Structural: need more HW resources**
  - **Data (RAW,WAR,WAW): need forwarding, compiler scheduling**
  - **Control: delayed branch, prediction**