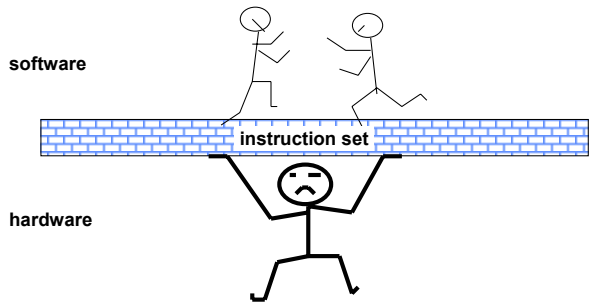
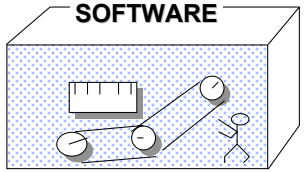
Lecture 2: Review of Pipelines

The Instruction Set: a Critical Interface



Instruction Set Architecture

... the attributes of a [computing] system as seen by the programmer, *i.e.* the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.
– Amdahl, Blaaw, and Brooks, 1964



- -- Organization of Programmable Storage
- -- Data Types & Data Structures: Encodings & Representations
- -- Instruction Formats
- -- Instruction (or Operation Code) Set
- -- Modes of Addressing and Accessing Data Items and Instructions
- -- Exceptional Conditions

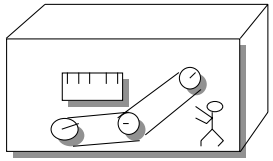
Organization

- Capabilities & Performance Characteristics of Principal Functional Units
  - (e.g., Registers, ALU, Shifters, Logic Units, ...)
- Ways in which these components are interconnected
- Information flows between components
- Logic and means by which such information flow is controlled.
- Choreography of FUs to realize the ISA
- Register Transfer Level (RTL) Description

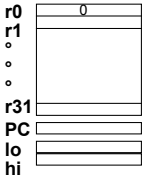
Logic Designer's View

ISA Level

FUs & Interconnect



Review: MIPS R3000 (core)



Programmable storage

2^32 x bytes

31 x 32-bit GPRs (R0=0)

32 x 32-bit FP regs (paired DP)

HI, LO, PC

Data types ?

Format ?

Addressing Modes?

Arithmetic logical

Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU,

AddI, AddIU, SLTI, SLTIU, AndI, OrI, XorI, *LUI*

SLL, SRL, SRA, SLLV, SRLV, SRAV

Memory Access

LB, LBU, LH, LHU, LW, LWL,LWR

SB, SH, SW, SWL, SWR

Control

32-bit instructions on word boundary

J, JAL, JR, JALR

BEq, BNE, BLEZ,BGTZ,BLTZ,BGEZ,BLTZAL,BGEZAL

Review: Basic ISA Classes

Accumulator:

1 address	add A	acc ← acc + mem[A]
1+x address	addx A	acc ← acc + mem[A + x]

Stack:

0 address	add	tos ← tos + next

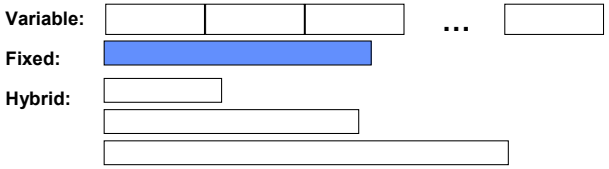
General Purpose Register:

2 address	add A B	EA(A) ← EA(A) + EA(B)
3 address	add A B C	EA(A) ← EA(B) + EA(C)

Load/Store:

3 address	add Ra Rb Rc	Ra ← Rb + Rc
	load Ra Rb	Ra ← mem[Rb]
	store Ra Rb	mem[Rb] ← Ra

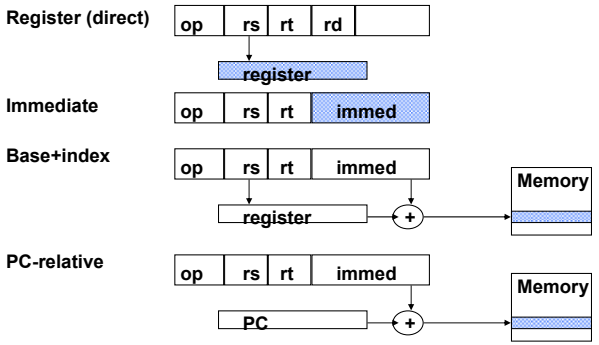
Instruction Formats



- Addressing modes
  - –each operand requires addess specifier => variable format
- code size => variable length instructions
- performance => fixed length instructions
  - –simple decoding, predictable operations
- With load/store instruction arch, only one memory address and few addressing modes
- => simple format, address mode given by opcode

MIPS Addressing Modes & Formats

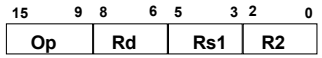
- Simple addressing modes
- All instructions 32 bits wide



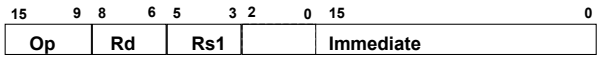
- Register Indirect?

Cray-1: the original RISC

Register-Register



Load, Store and Branch



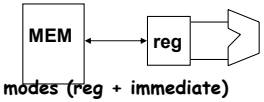
VAX-11: the canonical CISC

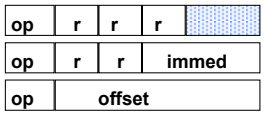
Variable format, 2 and 3 address instruction

- Rich set of orthogonal address modes
  - immediate, offset, indexed, autoinc/dec, indirect, indirect+offset
  - applied to any operand
- Simple and complex instructions
  - synchronization instructions
  - data structure operations (queues)
  - polynomial evaluation

Review: Load/Store Architectures

- 3 address GPR
- Register to register arithmetic
- Load and store with simple addressing modes (reg + immediate)
- Simple conditionals
  - compare ops + branch z
  - compare&branch
  - condition code + branch on condition
- Simple fixed-format encoding



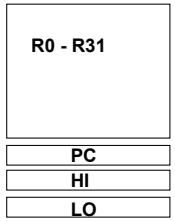


- Substantial increase in instructions
- Decrease in data BW (due to many registers)
- Even more significant decrease in CPI (pipelining)
- Cycle time, Real estate, Design time, Design complexity

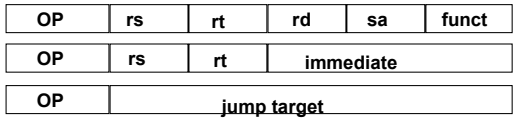
MIPS R3000 ISA (Summary)

- Instruction Categories
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
    - » coprocessor
  - Memory Management
  - Special

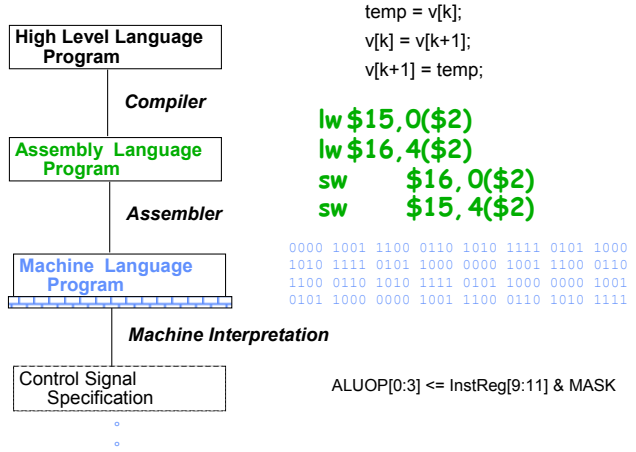
Registers



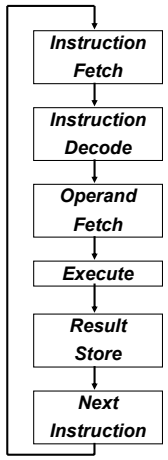
3 Instruction Formats: all 32 bits wide



Levels of Representation (61C Review)

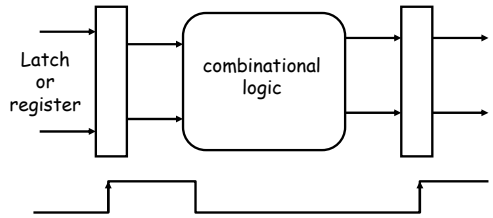


Execution Cycle



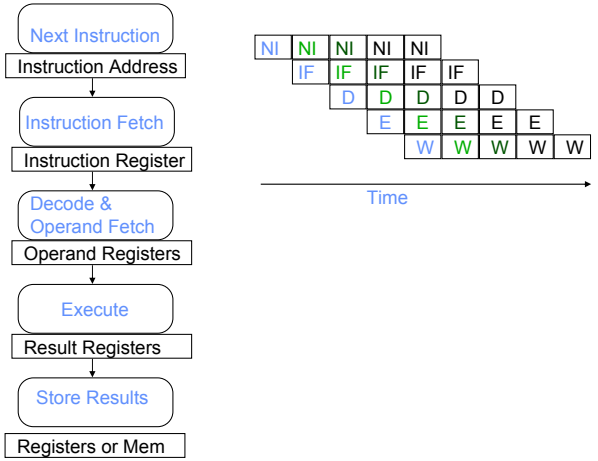
- **Obtain instruction from program storage**
- **Determine required actions and instruction size**
- **Locate and obtain operand data**
- **Compute result value or status**
- **Deposit results in storage for later use**
- **Determine successor instruction**

What's a Clock Cycle?



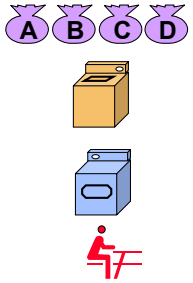
- **Old days: 10 levels of gates**
- **Today: determined by numerous time-of-flight issues + gate delays**
  - **clock propagation, wire lengths, drivers**

Fast, Pipelined Instruction Interpretation

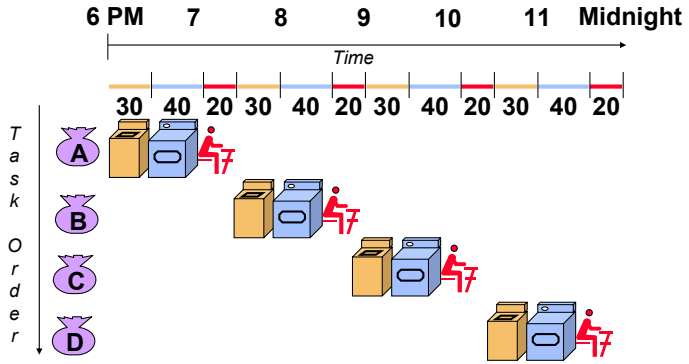


Pipelining: It's Natural!

- 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

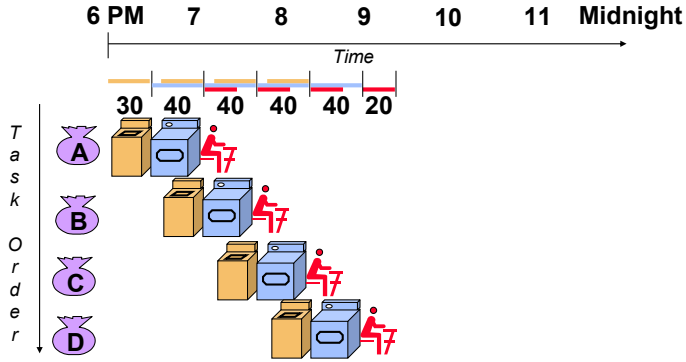


Sequential Laundry



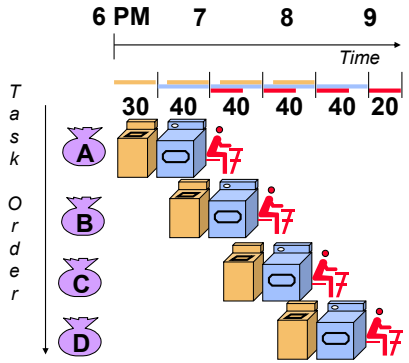
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

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

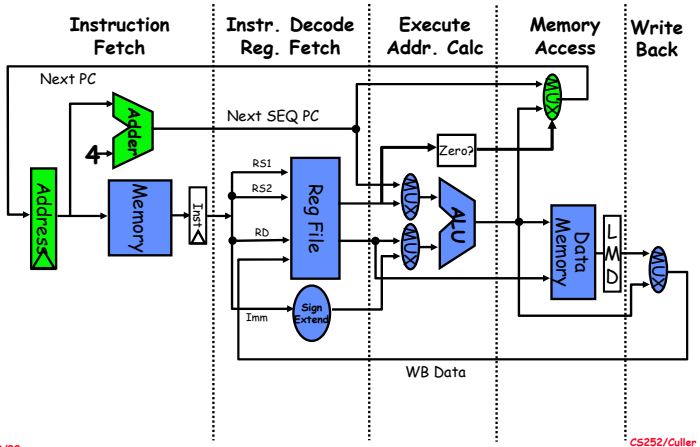
Computer Pipelines

- 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 MIPS Datapath

Figure 3.1, Page 130, CA:AQA 2e



Steps 1 & 2

- IF - instruction fetch step

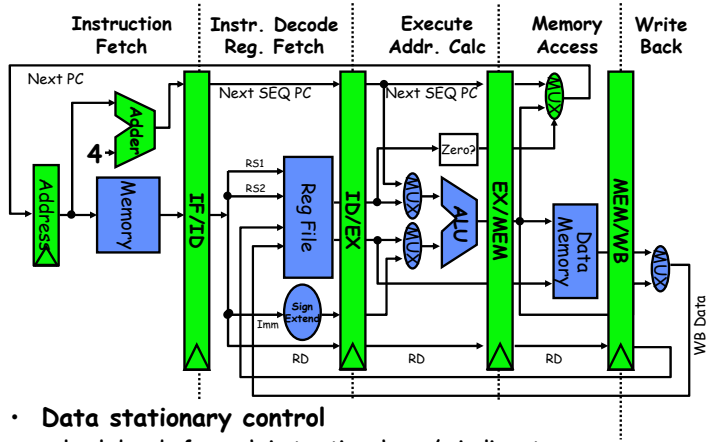
  IR <-- Mem[PC]: fetch the next instruction from memory

  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

5 Steps of MIPS Datapath

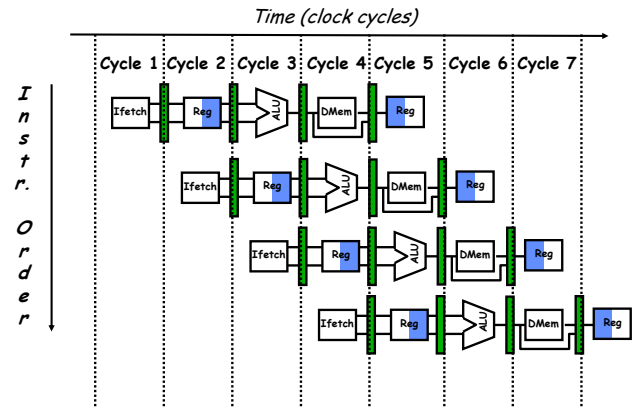
Figure 3.4, Page 134 , CA:AQA 2e



- Data stationary control
  - local decode for each instruction phase / pipeline stage

Visualizing Pipelining

Figure 3.3, Page 133 , CA:AQA 2e

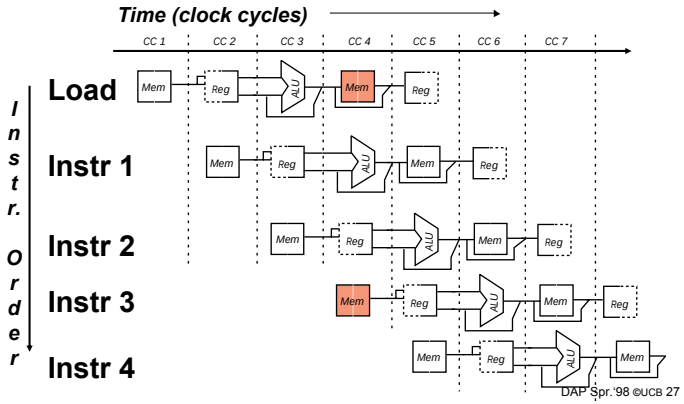


Its Not That Easy for Computers

- 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



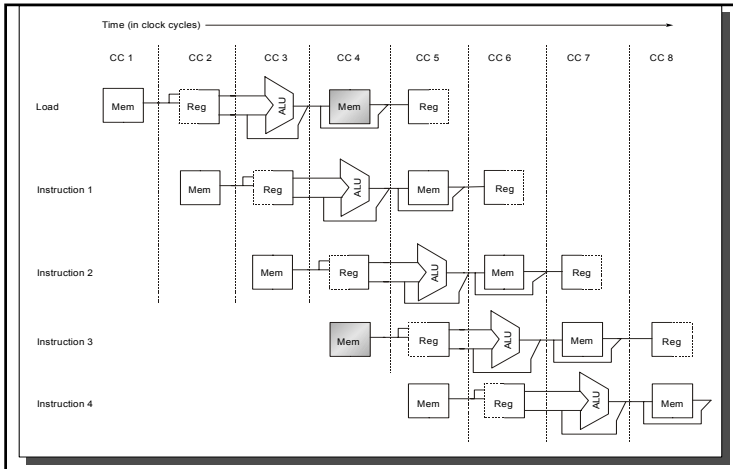
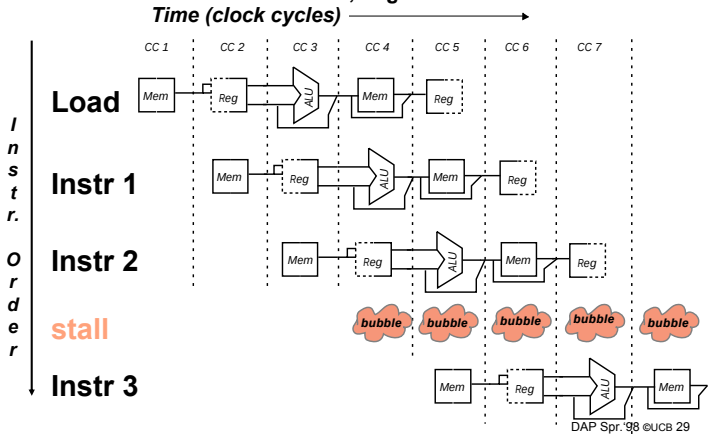


FIGURE 3.6 A machine with only one memory port will generate a conflict whenever a memory reference occurs.

One Memory Port/Structural Hazards

Figure 3.7, Page 143



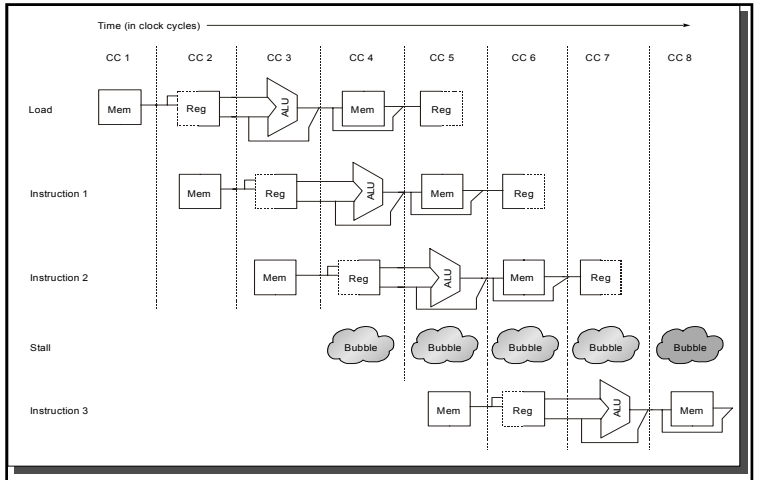


FIGURE 3.7 The structural hazard causes pipeline bubbles to be inserted.

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

$$\text{SpeedUp}_A = \text{Pipeline Depth}/(1 + 0) \times (\text{clock}_{unpipe}/\text{clock}_{pipe}) = \text{Pipeline Depth}$$

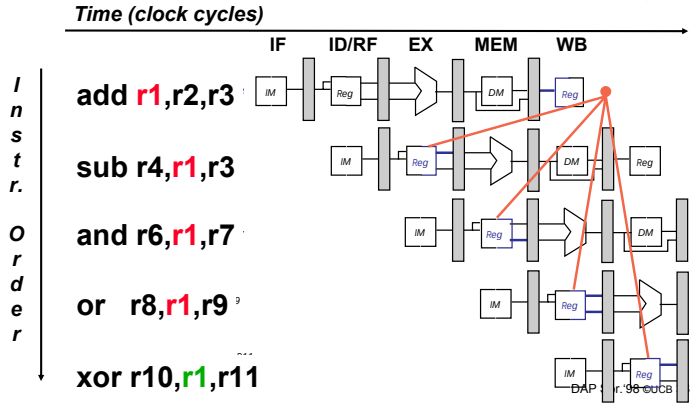
$$\text{SpeedUp}_B = \text{Pipeline Depth}/(1 + 0.4 \times 1) \times (\text{clock}_{unpipe}/(\text{clock}_{unpipe}/1.05)) = (\text{Pipeline Depth}/1.4) \times 1.05 = 0.75 \times \text{Pipeline Depth}$$

$\text{SpeedUp}_A / \text{SpeedUp}_B$ = Pipeline Depth/(0.75 x Pipeline Depth) = 1.33

- Machine A is 1.33 times faster

Data Hazard on R1

Figure 3.9, page 147



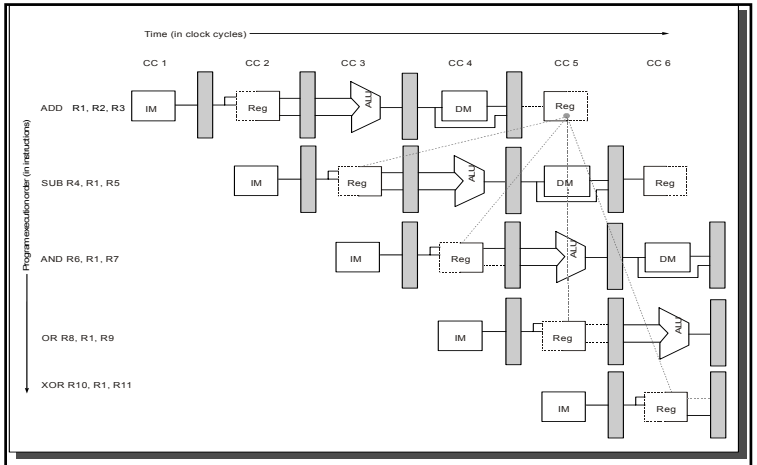


FIGURE 3.9 The use of the result of the ADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

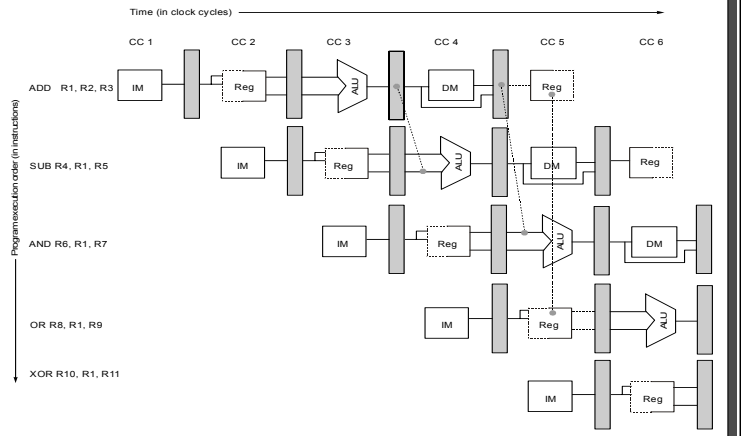


FIGURE 3.10 A set of instructions that depend on the ADD result use forwarding paths to avoid the data hazard.

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

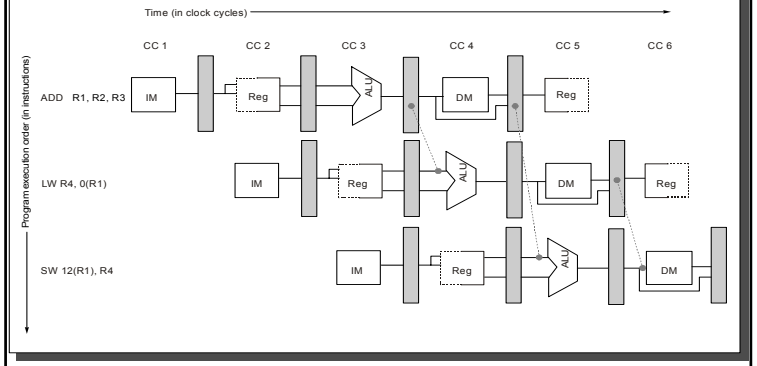


FIGURE 3.11 Stores require an operand during MEM, and forwarding of that operand is shown here.

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

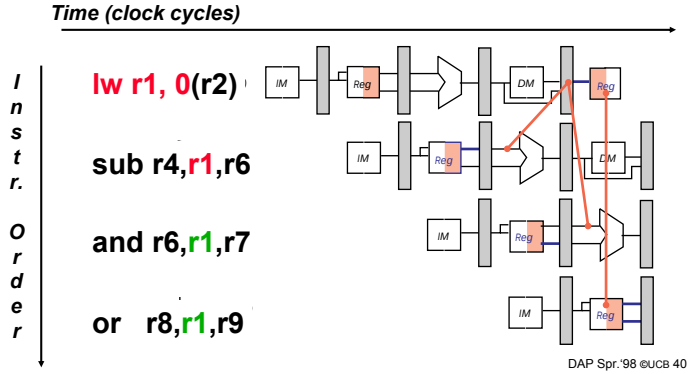
Three Generic Data Hazards

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



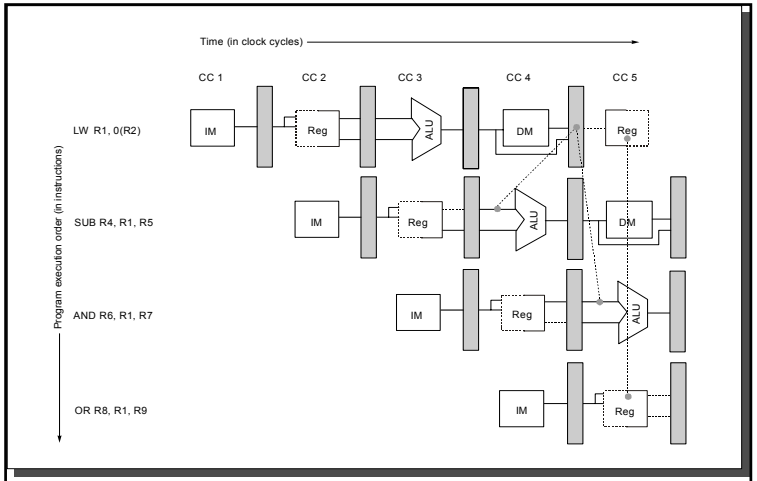


FIGURE 3.12 The load instruction can bypass its results to the AND and OR instructions, but not to the SUB, since that would mean forwarding the result in "negative time."

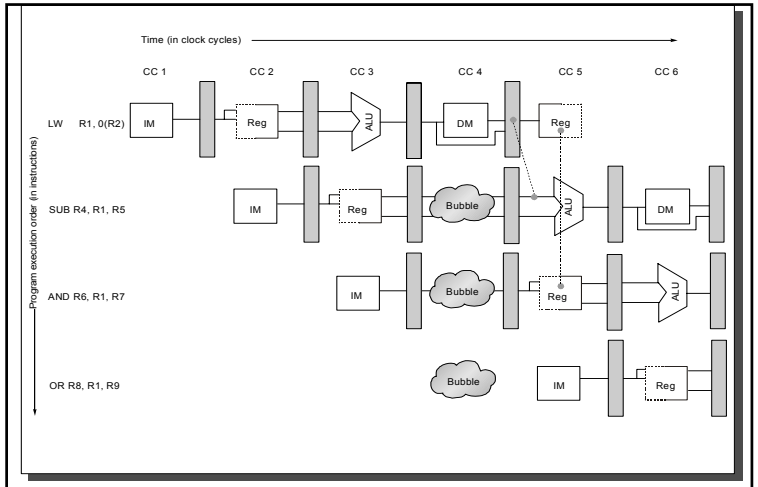


FIGURE 3.13 The load interlock causes a stall to be inserted at clock cycle 4, delaying the SUB instruction and those that follow by one cycle.

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;

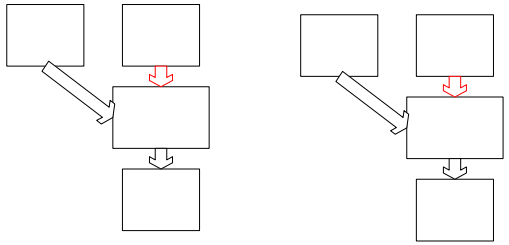
d = e – f;

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

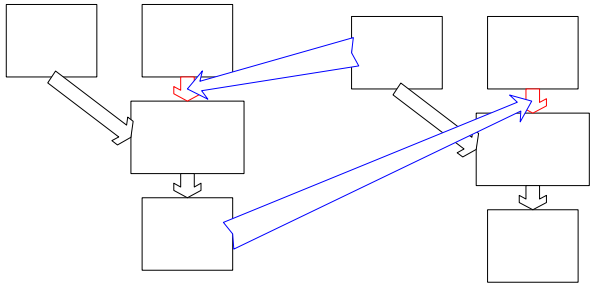
Slow code:

```
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
```

Data Flow Graph



Scheduling using DFG



Software Scheduling to Avoid Load Hazards

Try producing fast code for

a = b + c;

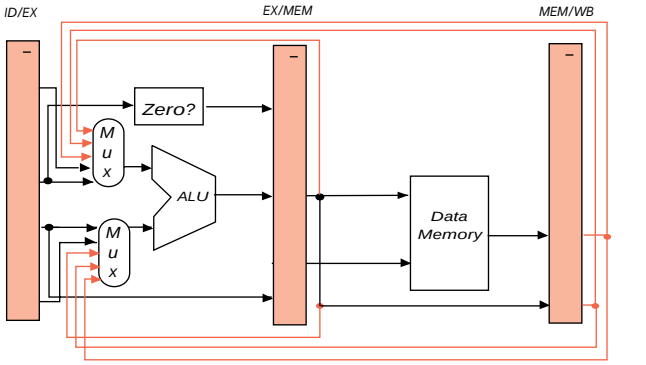
d = e – f;

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

Slow code:		Fast code:	
LW	Rb,b	LW	Rb,b
LW	Rc,c	LW	Rc,c
ADD	Ra,Rb,Rc	LW	Re,e
SW	a,Ra	ADD	Ra,Rb,Rc
LW	Re,e	LW	Rf,f
LW	Rf,f	SW	a,Ra
SUB	Rd,Re,Rf	SUB	Rd,Re,Rf
SW	d,Rd	SW	d,Rd

HW Change for Forwarding

Figure 3.20, Page 161



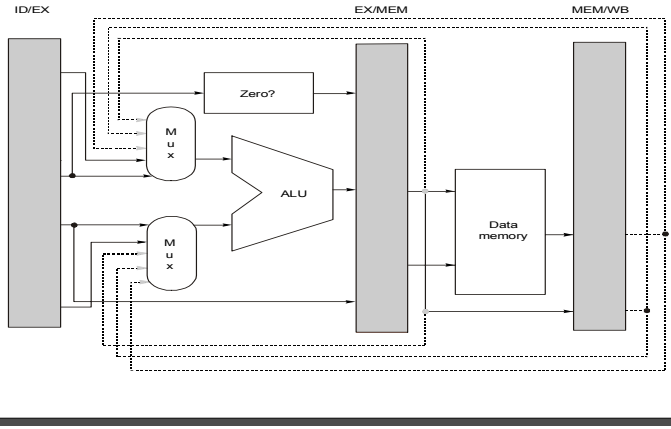


FIGURE 3.20 Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs.

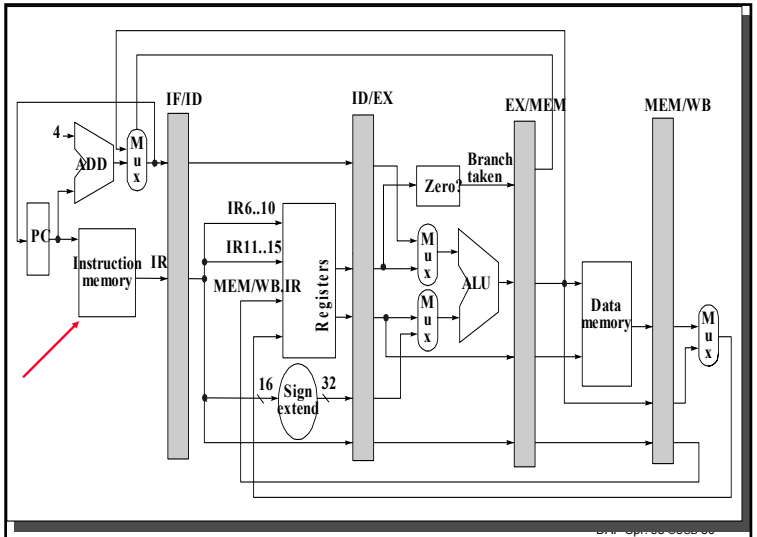
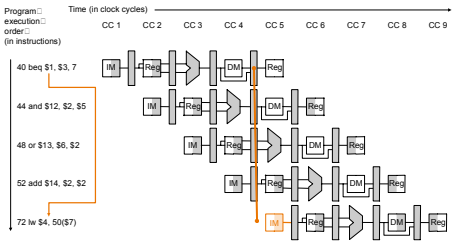


FIGURE 3.4 The datapath is pipelined by adding a set of registers, one between each pair of pipe stages.

Branch Hazards

- When we decide to branch, other instructions are in the pipeline!



- We are predicting "branch not taken"
  - need to add hardware for flushing instructions if we are wrong

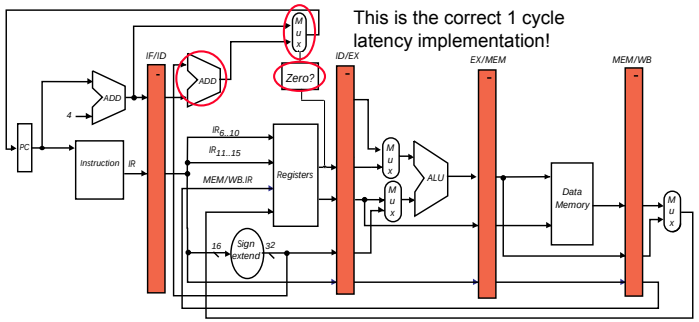
Control Hazard on Branches Three Stage Stall

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



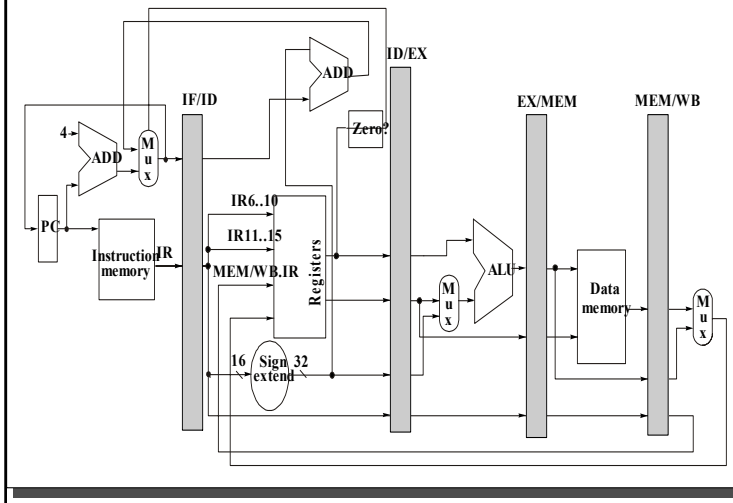


FIGURE 3.22 The stall from branch hazards can be reduced by moving the zero test and branch target calculation into the ID phase of the pipeline.

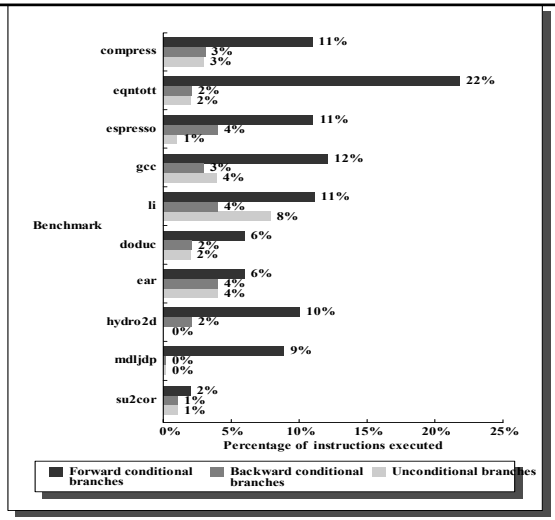


FIGURE 3.24 The frequency of instructions (branches, jumps, calls, and returns) that may change the PC.

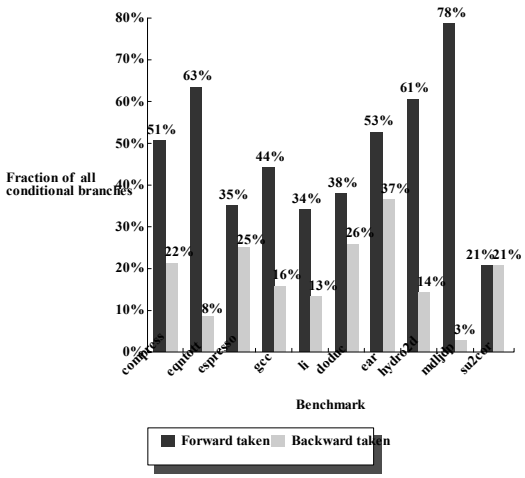


FIGURE 3.25 Together the forward and backward taken branches account for an average of 67% of all conditional branches.

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

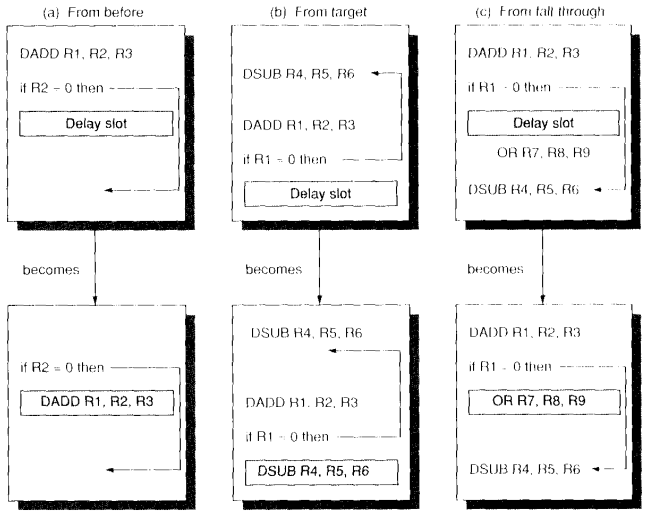
```
branch instruction
  sequential successor1
  sequential successor2
  ........
  sequential successorn
branch target if taken
```

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**