

Empty Book Template

Ryan Hou

Invalid Date

Table of contents

Preface	4
Resources	5
1 Introduction	6
1.1 Perspective	6
1.2 High Level Ideas	6
1.3 Understanding the Execution Core	6
1.4 Classes of Parallelism & Parallel Architectures	6
2 Performance, Power, ISA	8
2.1 Parallelism	9
2.1.1 Amdahl's Law	9
2.2 Performance	10
2.2.1 Averaging Metrics	11
2.2.2 Iron Law of Processor Performance	12
2.2.3 Performance - Summary	13
2.3 Instruction Set Architectures	13
2.3.1 RISC vs CISC	14
2.4 Power	14
2.5 Voltage Scaling	16
2.6 CPI, IPC	16
3 In-Order CPU, Pipelining	17
3.1 Fetch	17
3.2 Decode	18
3.3 Execute	19
3.4 Memory Operation	20
3.5 Writeback	21
3.6 Timing	22
3.7 Dependencies, Hazards	23
3.7.1 Structural Hazards	24
3.7.2 Data Hazards	24
3.7.3 Control Hazards	31
3.8 Summary	33
3.9 References	34

14 Summary	45
References	46

Preface

Notes on computer architecture.

Resources

Some relevant resources:

- [Resource Name](#)

Textbooks:

- [Book 1](#)

1 Introduction

1.1 Perspective

i Note 1: Definition - Some definition

Term is defined as blah blah blah...

This note does ...

1.2 High Level Ideas

1.3 Understanding the Execution Core

1. In-order chapter introduces traditional pipelined design in a 5-stage pipelin
2. Dynamic Scheduling: Scoreboard (OoO Basics Chapter)
3. Register Renaming: Tomasulo's Algorithm (OoO Basics Chapter)
4. Precise Interrupts with Reorder Buffer: P6 & MIPS R10K -style examples (P6 R10K Chapter)

1.4 Classes of Parallelism & Parallel Architectures

Reference: Patterson & Hennessy Chp. 1.2

Two Kinds of Parallelisms in Applications:

1. Data-Level Parallelism
 1. Multiple data items can be operated on at the same time
2. Task-Level Parallelism
 1. Multiple tasks can be processed independently at the same time

Hardware can exploit these application parallelisms in four ways:

1. Instruction-Level Parallelism
2. Vector Architectures
3. Thread-Level Parallelism
4. Request-Level Parallelism

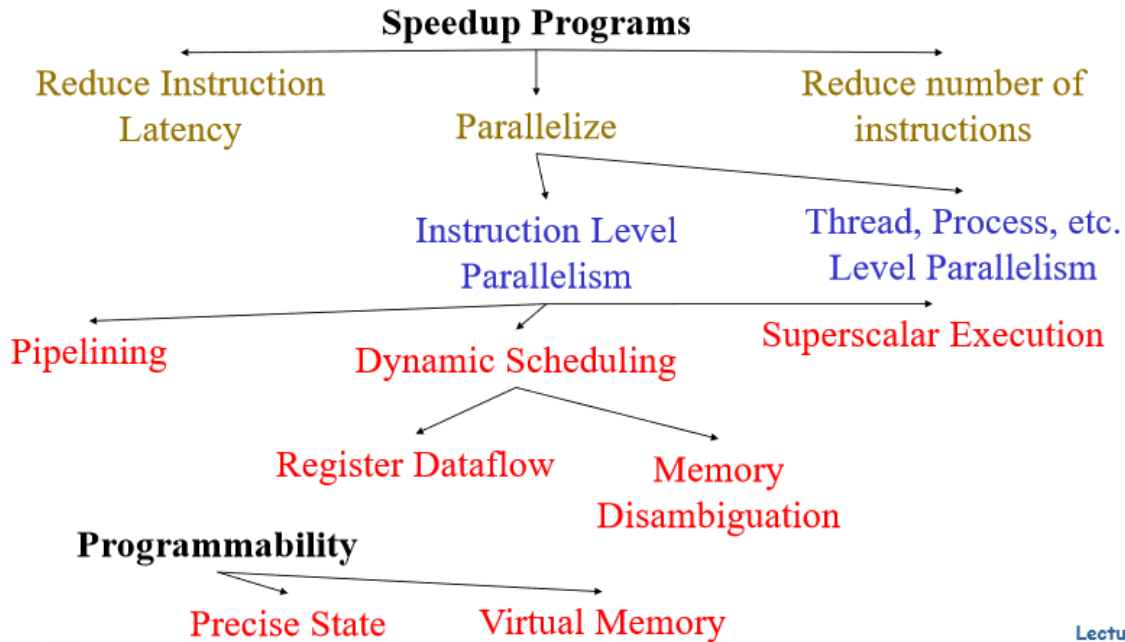
Four Categories of Computing Architectures:

1. Single Instruction stream, Single Data stream (SISD)
 1. A uniprocessor - can exploit ILP such as superscalar and speculative execution
2. Single Instruction stream, Multiple Data stream (SIMD)
 1. Same instruction executed by many processors on different data streams
 2. Exploits DLP
 3. Each processor has its own data memory, but have a single instruction memory and control processor
 4. NVIDIA: "SIMT"
3. Multiple Instruction streams, Single Data stream (MISD)
 1. Doesn't really exist in commercial use
4. Multiple Instruction streams, Multiple Data stream (MIMD)
 1. Each processor has its own instruction stream and data stream
 2. More flexible than SIMD but due to overheads this parallelism is expensive (limits degree of parallelism achievable)

2 Performance, Power, ISA

How do we speed up tasks? Three options

- Do less things
 - Program more efficiently
 - Compilers - optimize code
- Do things faster
 - Logic optimizations
 - Circuits - design faster and more efficient circuits
 - Architecture
 - * Memoization - (save results of previous comp. in memory to avoid recomputing)
 - * Locality - predict/speculate, cache
- Do more things at the same time
 - Architecture - Parallelization!
 - * Pipelining
 - * Speculative execution
 - * Dynamic scheduling
 - * Register Renaming
 - * Branch Prediction
 - * Superscalar
 - * Multiprocessing / Multithreading
 - * VLIW



EECS 470

Lecture 1
Slide 27

2.1 Parallelism

2.1.1 Amdahl's Law

Suppose an enhancement speeds up a fraction f of a task by a factor of S :

$$time_{new} = time_{orig} \cdot \left((1 - f) + \frac{f}{S} \right)$$

Amdahl's Law:

$$S_{overall} = \frac{1}{(1 - f) + \frac{f}{S}}$$

Parallelism - the amount of independent sub-tasks available

Work= T_1 - time to complete a computation on a sequential system

Critical Path= T_∞ - time to complete the same computation on an infinitely-parallel system

Average Parallelism

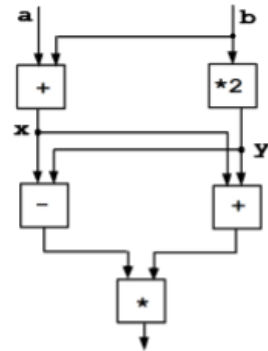
$$P_{\text{avg}} = \frac{T_1}{T_\infty}$$

For a p wide system

$$T_p \geq \max\left\{\frac{T_1}{p}, T_\infty\right\}$$

$$P_{\text{avg}} \gg p \Rightarrow T_p \approx \frac{T_1}{p}$$

```
x = a + b;
y = b * 2
z = (x-y) * (x+y)
```



Lec

2.2 Performance

Two key performance metrics:

- **Latency (execution time):** time to finish a fixed task
- **Throughput (bandwidth):** number of tasks finished in fixed time

- Latency / throughput means there is some ambiguity when describing something as "faster"
- Processor A is X times faster than processor B if
 - ❑ $\text{Latency}(A) = \text{Latency}(B) / X$
 - ❑ $\text{Throughput}(A) = \text{Throughput}(B) * X$
- Processor A is X% faster than processor B if
 - ❑ $\text{Latency}(A) = \text{Latency}(B) / (1+X/100)$
 - ❑ $\text{Throughput}(A) = \text{Throughput}(B) * (1+X/100)$
- Car/bus example
 - ❑ Latency? Car is 3 times (and 200%) faster than bus
 - ❑ Throughput? Bus is 4 times (and 300%) faster than car

On HW#1, we say one processor is X% faster on a given benchmark. Which definition is relevant?

2.2.1 Averaging Metrics

Latency can be added, but not throughput!

- $\text{Latency}(A+B) = \text{Latency}(A) + \text{Latency}(B)$
- $\text{Throughput}(A+B) \neq \text{Throughput}(A) + \text{Throughput}(B)$

Adding throughput:

$$\text{Throughput}(A + B) = \frac{1}{\frac{1}{\text{Throughput}(A)} + \frac{1}{\text{Throughput}(B)}}$$

Averaging Techniques:

- Three averaging techniques:

- **Arithmetic** : $(1/N) * \sum_{P=1..N} \text{Latency}(P)$
 - **For times**: units proportional to time (e.g., latency)
- **Harmonic** : $N / \sum_{P=1..N} 1/\text{Throughput}(P)$
 - **For rates**: units inversely proportional to time (e.g., throughput)
 - (Unless time is fixed)
- **Geometric** : $\sqrt[N]{\prod_{P=1..N} \text{Speedup}(P)}$
 - **For ratios**: unitless quantities (e.g., speedups)

EECS 470

2.2.2 Iron Law of Processor Performance

$$\text{Processor Performance} = \frac{\text{Time}}{\text{Program}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(code size) (CPI) (cycle time)

Another way of looking at it:

- seconds / instruction = (cycles / instructions) * (seconds / cycle)

2.2.3 Performance - Summary

Performance - Key Points

Amdahl's law

$$S_{\text{overall}} = \frac{1}{(1-f) + \frac{f}{S}}$$

Iron law

$$\frac{\text{Time}_{\text{Program}}}{\text{Program}} = \frac{\text{Instructions}_{\text{Program}}}{\text{Program}} \times \frac{\text{Cycles}_{\text{Instruction}}}{\text{Instruction}} \times \frac{\text{Time}_{\text{Cycle}}}{\text{Cycle}}$$

Averaging Techniques

Arithmetic
Time

$$\frac{1}{n} \sum_{i=1}^n \text{Time}_i$$

Harmonic
Rates

$$\frac{n}{\sum_{i=1}^n \frac{1}{\text{Rate}_i}}$$

Geometric
Ratios

$$\sqrt[n]{\prod_{i=1}^n \text{Ratio}_i}$$

EECS 470

2.3 Instruction Set Architectures

ISA is the “contract” between software and hardware:

- **Functional definition** of operations, modes, and storage locations supported by hardware
- **Precise description** of how to invoke and access them

2.3.1 RISC vs CISC

- Recall “Iron” law:
 - $(\text{instructions/program}) * (\text{cycles/instruction}) * (\text{seconds/cycle})$
- **CISC** (Complex Instruction Set Computing)
 - Improve “instructions/program” with “complex” instructions
 - Easy for assembly-level programmers, good code density
- **RISC** (Reduced Instruction Set Computing)
 - Improve “cycles/instruction” with many single-cycle instructions
 - Increases “instruction/program”, but hopefully not as much
 - Help from smart compiler
 - Perhaps improve clock cycle time (seconds/cycle)
 - via aggressive implementation allowed by simpler instructions

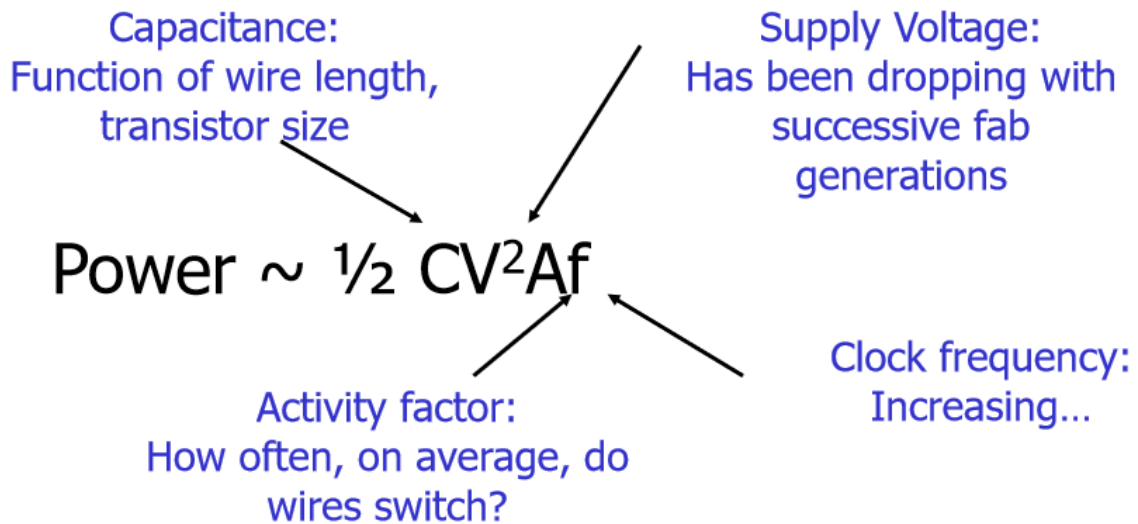
ECS 470

Back in the day it was important to minimize number of instructions due to limited memory. This has become much less of a concern nowadays.

2.4 Power

- Dynamic Power: Switching power
 - Capacitive and short-circuit
 - Capacitive power: charging/discharging transistors from 0 to 1 and 1 to 0
 - Short-circuit power: power due to brief short-circuit current during the transitions
 - Data dependent - a function of switching activity
- Static Power: leakage power
 - Steady, per-cycle energy cost
- Dynamic power dominates but static power increasing in importance (due to large transistor count and increasing leakage in tech scaling and)

Capacitive Power dissipation



- Power (Watts)
 - Determines battery life in hours
 - Sets packaging limits (due to thermals, etc)
- Energy (Joules)
 - Rate at which energy is consumed over time
 - Energy = Power \times Delay
 - * Joules = Watts \times Seconds

2.5 Voltage Scaling

- Scenario: 80W, 1 BIPS, 1.5V, 1GHz
 - Cache Optimization:
 - IPC decreases by 10%, reduces power by 20% => Final Processor: 900 MIPS, 64W
- What if we just adjust frequency/voltage on processor?
 - How to reduce power by 20%?
 - $P = CV^2F = CV^3 \Rightarrow$ Drop voltage by 7% (and also Freq) $\Rightarrow .93 * .93 * .93 = .8x$
 - So for equal power (64W)
 - Cache Optimization = 900MIPS
 - Simple Voltage/Frequency Scaling = 930MIPS

Power has a cubic relationship with voltage! Why? Voltage linearly correlated to frequency (higher VDD enables faster switching)

2.6 CPI, IPC

Cycles Per Instruction

- Lower the better
- IPC is its inverse
- Summary for different arch:
 - Ideal in-order pipeline (no stalling): 1 (ignoring cycles to load up and clear the pipeline)
 -

3 In-Order CPU, Pipelining

Pipelining

- Improves throughput at the expense of latency
 - Why latency goes up?
 - * More HW added to each stage (eg pipeline reg)
 - * Clocked at speed of slowest stage
- Higher throughput as instructions finish at faster rate, and more instructions are executed in parallel

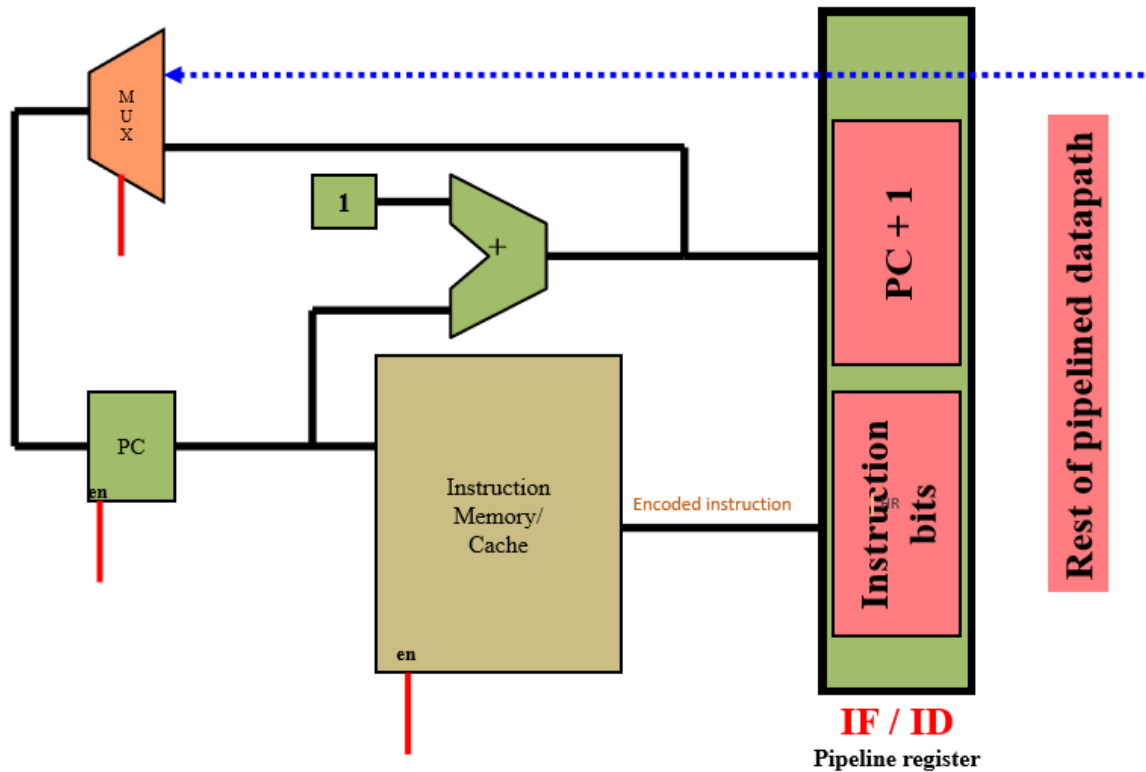
We now discuss an in-order 5-stage pipeline CPU architecture. The 5 stages are:

1. Fetch
2. Decode
3. Execute
4. Memory
5. Writeback

In the diagrams, red wires represent control signals

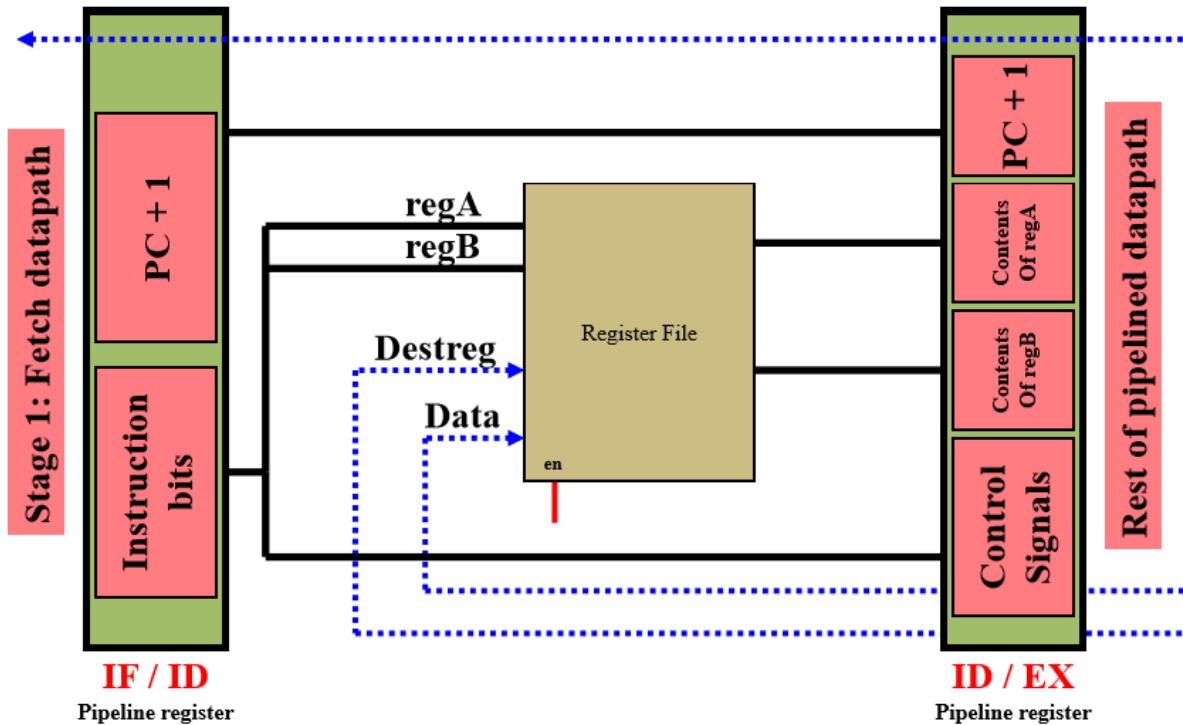
3.1 Fetch

- Fetch an instruction from memory every cycle
 - Use PC to index into memory
 - * $PC + 1$ or $PC + N$ (for N byte words)
 - Increment PC after fetching (assume no branches for now)
- Write the results to the pipeline register IF/ID



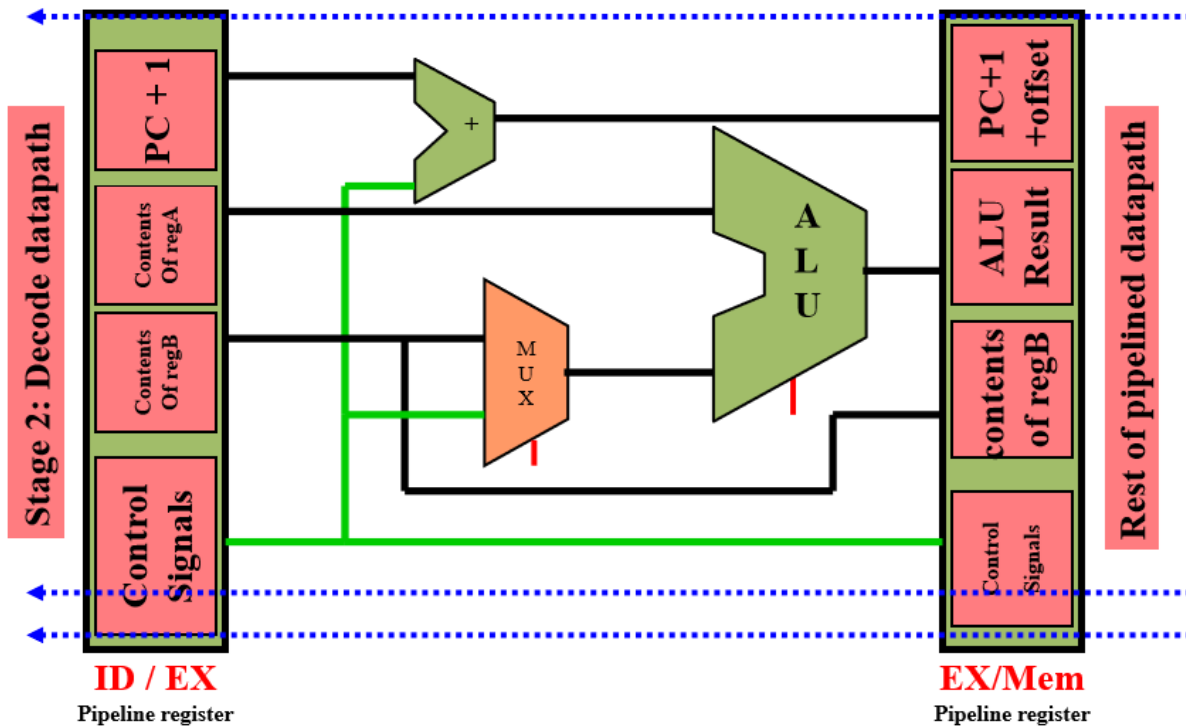
3.2 Decode

- Decodes the opcodes to know what operation it is
- Read input operands from the Register File
 - Operands specified by regA and regB of instruction
- Write state to the pipeline register ID/EX
 - Opcode
 - Register contents
 - Offset & destination fields
 - PC + 1 (or PC + 4)



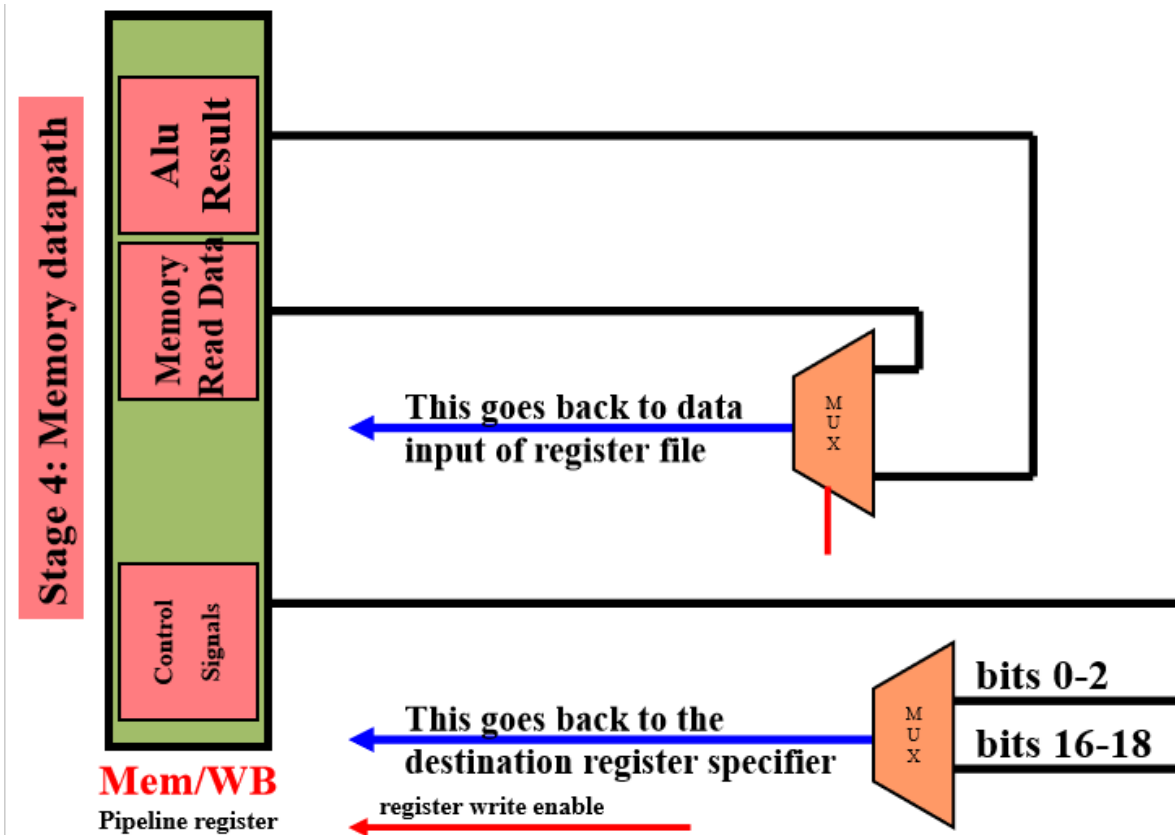
3.3 Execute

- Perform ALU Operation
 - Inputs can be regA or regB or offset fields on the instruction
 - Or for branch instructions calculate the $PC+1+offset$
- Write state to pipeline register EX/MEM
 - ALU results, contents of RegB and $PC+1+offset$
 - Instruction bits for opcode and destReg specifiers



3.4 Memory Operation

- Perform data cache access for memory operations (load/store)
 - ALU already gave us results for the address of load/store
 - Opcode bits control the read/write and enable signals to memory
- Write state to pipeline register MEM/WB
 - ALU Result and MemData
 - Instruction Bits for opcode and destReg specifiers
- Massively Simplifying assumption: mem operations take 1 cycle



3.6 Timing

	Time: 1	2	3	4	5	6	7	8	9
add	fetch	decode	execute	memory	writeback				
nand		fetch	decode	execute	memory	writeback			
lw			fetch	decode	execute	memory	writeback		
add				fetch	decode	execute	memory	writeback	
sw					fetch	decode	execute	memory	writeback

3.7 Dependencies, Hazards

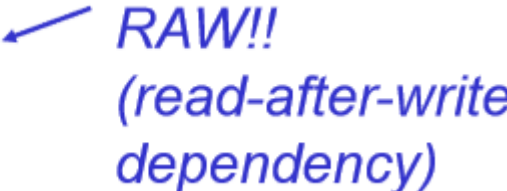
Why don't we have arbitrarily deep pipelines?

- Instruction pipelines are not ideal - i.e. instructions have dependencies between each other, too deep of a pipeline will cause a lot of stalls to resolve dependencies
- When pipelines are flushed, deep pipelines incur great cost since it takes many more cycles to fill back up

Hazards - situations that prevent the next instruction in the stream from executing in its designated clock cycle. Three classes of hazards:

1. **Structural Hazards** - resource conflicts in hardware
2. **Data hazards** - instruction depends on result of another instruction
3. **Control hazards** - comes from the pipelining of branches and other control flow

add 1 2 3
nand 3 4 5



*RAW!!
(read-after-write
dependency)*

RAW - Read After Write - a later instruction's input(s) relies on a previous instruction's result

Data dependency \neq hazards - they often lead to hazards but not necessarily

Pipeline Hazards:

- Potential violations of program dependencies
- Hazard resolution:
 - Static method: resolve at compile time in software (by compiler or programmer)
 - Dynamic method: resolve in hardware at run time
- Pipeline interlock:
 - Hardware mechanisms for dynamic hazard resolution
 - Must detect and enforce dependences at run time

3.7.1 Structural Hazards

In an in-order pipeline processor, structures such as the cache/memory can have limited ports to read/write from/to, thus overlapping usage of the memory can result in hazards

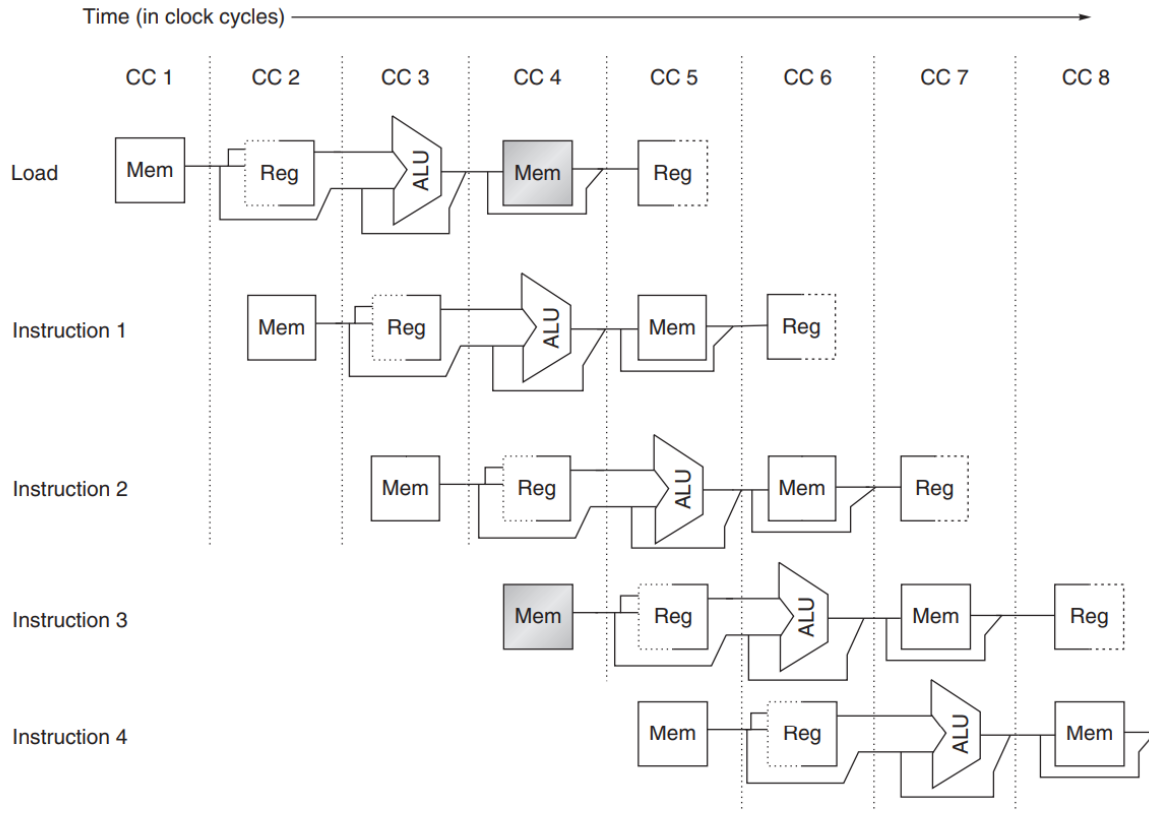


Figure C.4 A processor with only one memory port will generate a conflict whenever a memory reference occurs. In this example the load instruction uses the memory for a data access at the same time instruction 3 wants to fetch an instruction from memory.

3.7.2 Data Hazards

Techniques to handle data hazards:

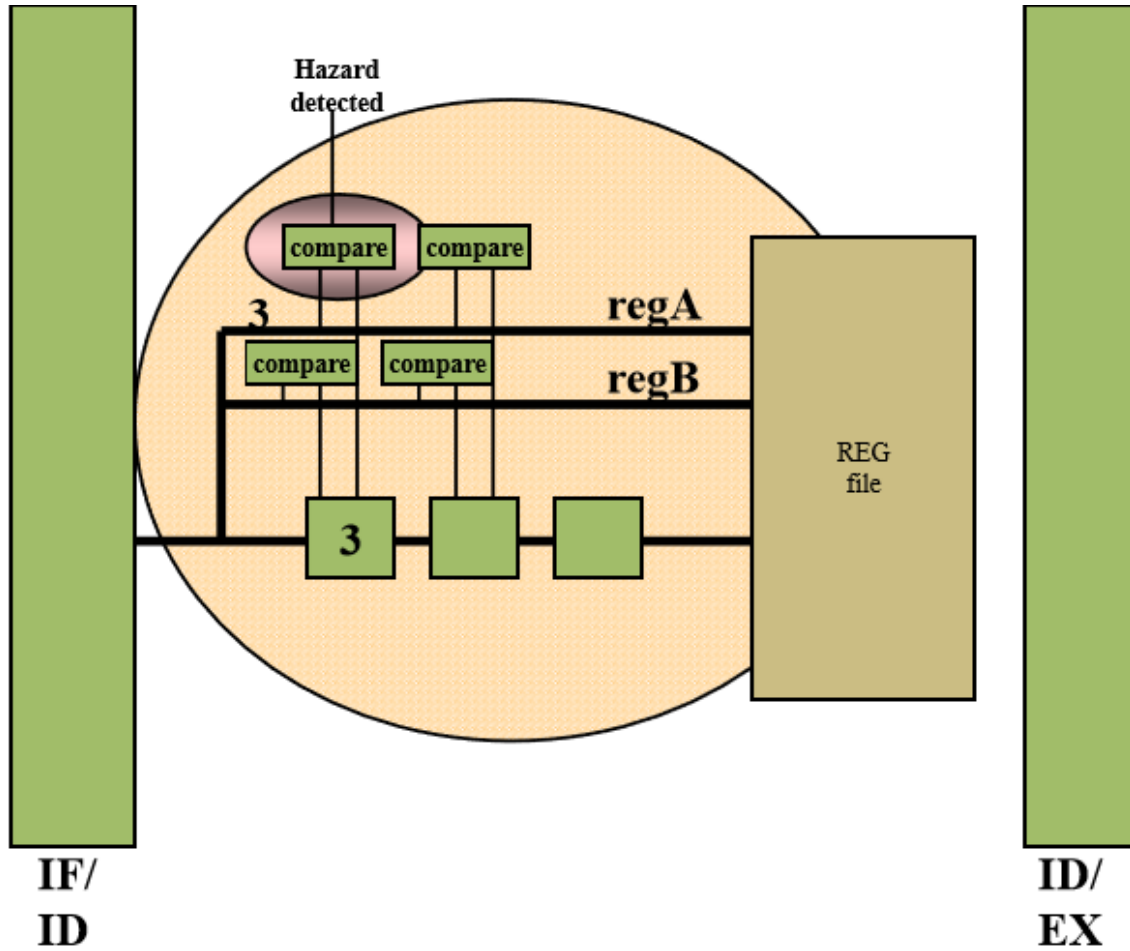
- Avoidance (static)
- Detect and Stall (dynamic)
- Detect and Forward (dynamic)

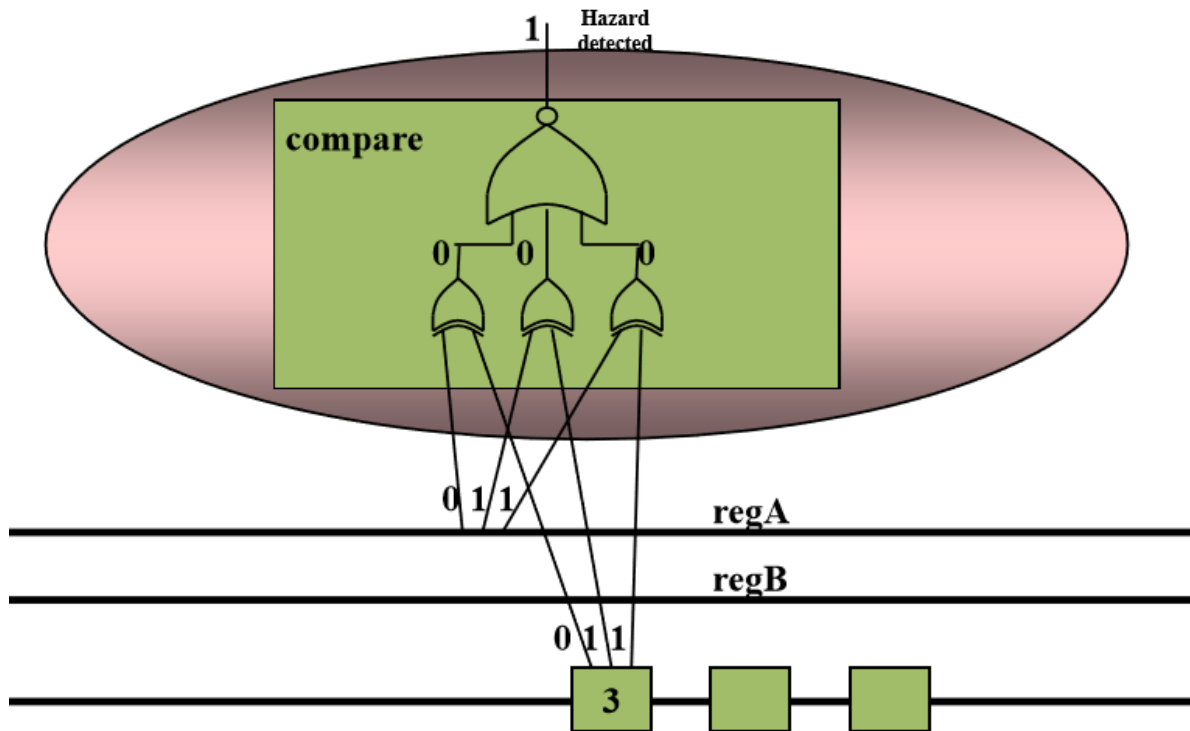
3.7.2.1 Avoidance

3.7.2.2 Detecting Data Hazard (In-Order Pipeline)

A RAW hazard can be detected by:

- Checking if regA & regB are the same as the destReg of the two instructions immediately before it
 - Why two instructions? See examples below





Consider the following example:

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

- All instructions after DADD reads from DADD's destReg (R1)
- DSUB will read the wrong R1
- AND will read the wrong R1
- OR will read the correct R1
 - Assuming ID reads from RF in the second half of the cycle:
 - DADD will write to RF in the first half of cycle and OR will perform RF read on second half. No forwarding needed
- XOR will read the correct R1

- Register read occurs the cycle AFTER DADD writesback

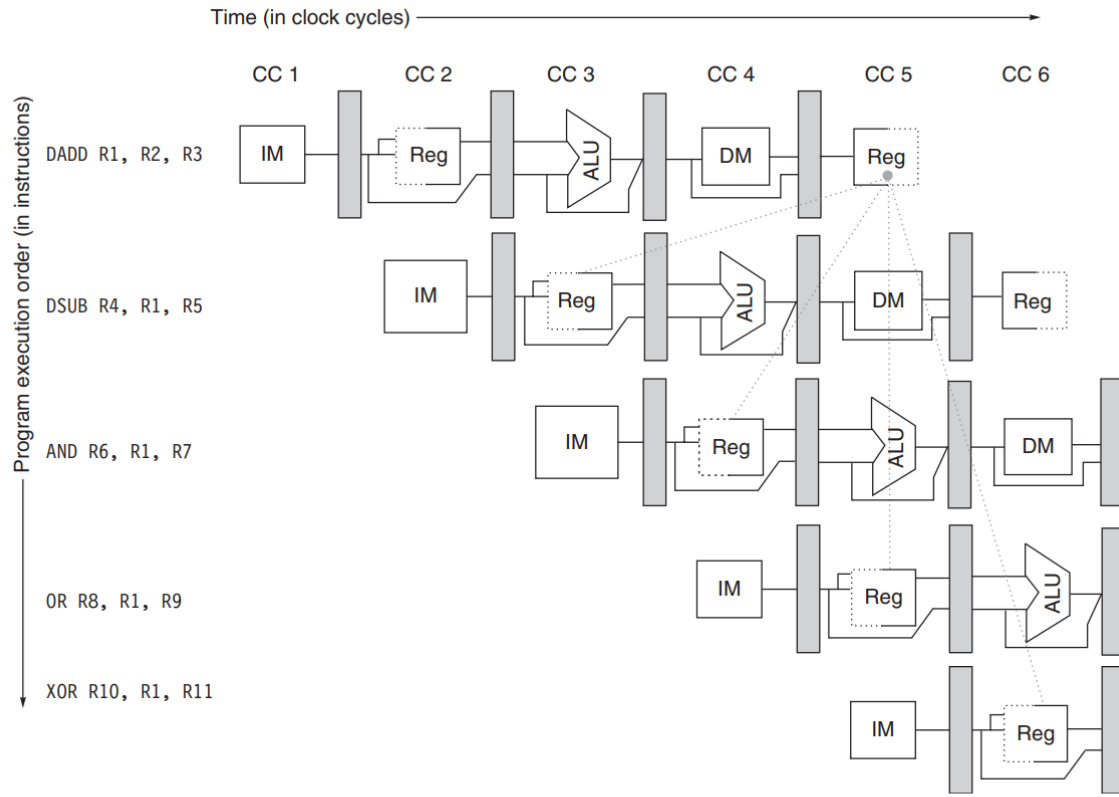


Figure C.6 The use of the result of the DADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

3.7.2.3 Detect & Stall

Every time a hazard is detected, we stall:

- Disable PC and do not advance pipeline register for IF/ID
- Clear ID/EX register
- Pass NOOP to Execute stage

Problems:

- CPI increases on every hazard!
- Unnecessary stalling (not “true” dependence)

3.7.2.4 Detect & Forward

After detecting hazard, forward the register's result

- Add data paths for all possible sources
- Add MUX in front of ALU to select source (based on detection)

Forwarding eliminates data hazards involving arithmetic instructions

More specifically, forwarding between arithmetic instructions works as follows:

1. ALU results from BOTH the EX/MEM and MEM/WB pipeline registers are fed back to the ALU inputs
 2. If data hazard is detected that the previous ALU instructions' destReg is a sourceReg of the current ALU instruction, MUX selects forwarded result
1. Note that if the instruction DSUB is stalled, forwarding will not be activated

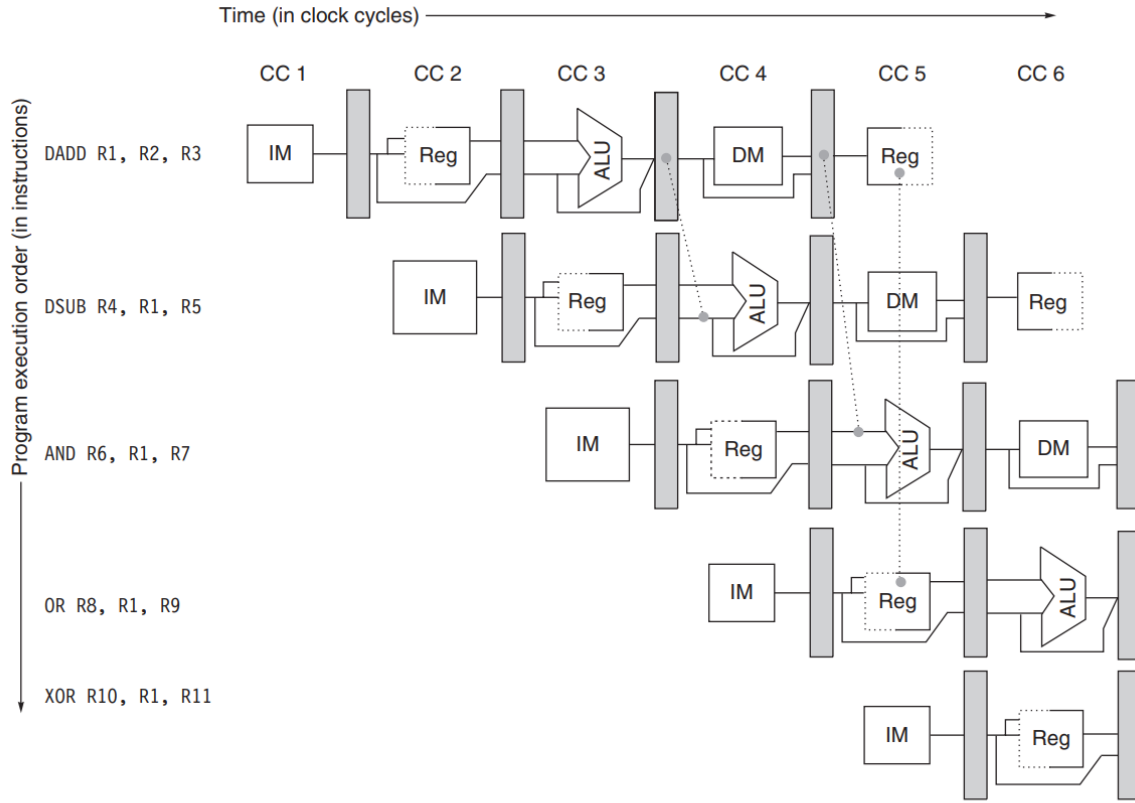


Figure C.7 A set of instructions that depends on the DADD result uses forwarding paths to avoid the data hazard. The inputs for the DSUB and AND instructions forward from the pipeline registers to the first ALU input. The OR receives its result by forwarding through the register file, which is easily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

In general, we can forward results directly to a functional unit that needs it

Example Instruction Sequence:

```
DADD R1,R2,R3
LD   R4,0(R1)
SD   R4,12(R1)
```

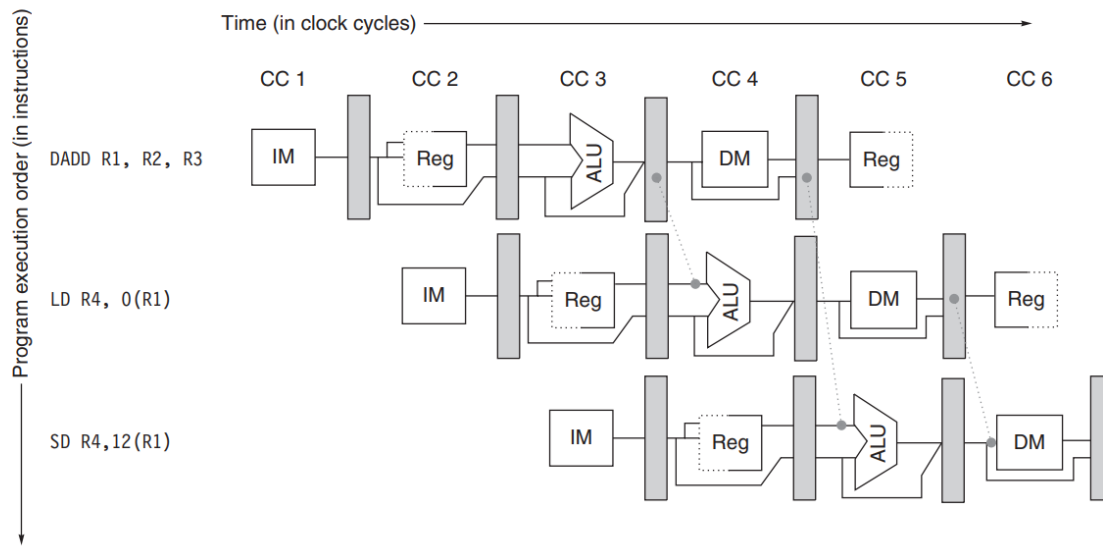


Figure C.8 Forwarding of operand required by stores during MEM. The result of the load is forwarded from the memory output to the memory input to be stored. In addition, the ALU output is forwarded to the ALU input for the address calculation of both the load and the store (this is no different than forwarding to another ALU operation). If the store depended on an immediately preceding ALU operation (not shown above), the result would need to be forwarded to prevent a stall.

In the above example, the forwarding path added on top of the previously mentioned paths is from WB/MEM register to input of MEM

Problems:

- Each possible hazard requires different forwarding paths
- “bypassing logic” is often a critical path in wide-issue machines
 - i.e. superscalar machines
 - number of forwarding paths grow quadratically with machine width

3.7.2.5 Data Hazards Requiring Stalls

Not all data hazards can be handled by forwarding/bypassing. Consider the following:

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

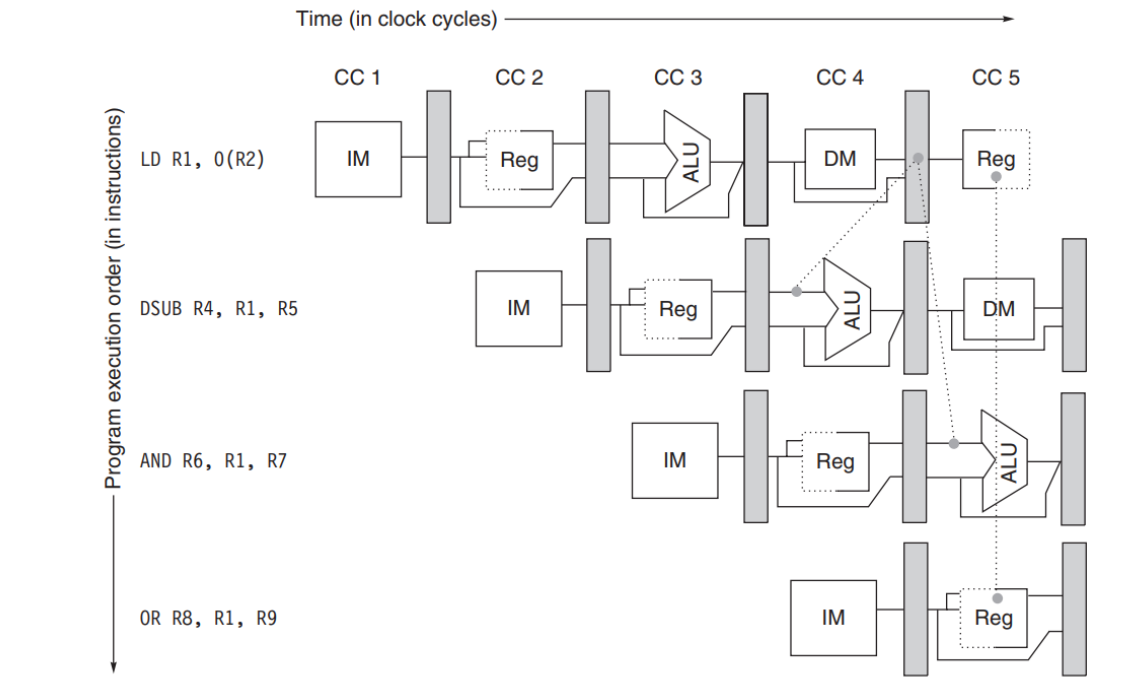


Figure C.9 The load instruction can bypass its results to the AND and OR instructions, but not to the DSUB, since that would mean forwarding the result in “negative time.”

Need to stall when an instruction is immediately after the load and the sourceReg is the load's destReg

To solve this, a **pipeline interlock** must be added to stall the dependent instruction by a cycle.

3.7.3 Control Hazards

Techniques to handle control hazards

- Avoidance (static)
- Detect and Stall (dynamic)
- Speculate and Squash (dynamic)

3.7.3.1 Avoidance

3.7.3.2 Detect and Stall

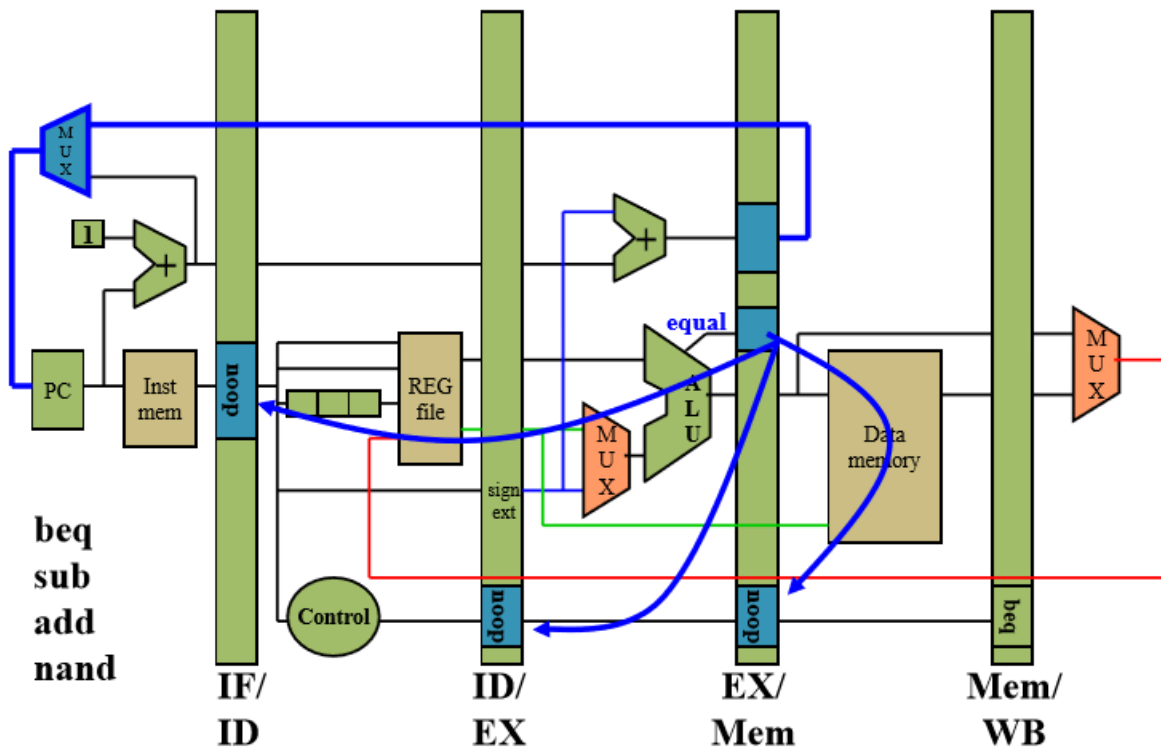
Detect when an opcode is a branch/jump. Then stall by inserting noops into the execute stage until the branch target is resolved.

- Branches result in a 1-2 cycle stall, depending on the ISA (if branch target is always known after ID, then 1 cycle stall; if branch target is known after EX, then 2 cycles stall)

3.7.3.3 Speculate and Squash

Speculate that a branch is Not Taken (i.e. $PC + 1$). If we see that we didn't speculate correctly, then we squash:

- Overwrite opcodes in fetch, decode, execute with NOOP (squash whatever is already in the pipeline)
- Pass correct target to fetch



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.12 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.

No penalties if the branches are always not taken.

3.8 Summary

- Hazards in in-order pipeline:
 - Structural Hazard - Memory port contention
 - Data Hazard - RAW Dependency
 - Control Hazard

Data Hazards - Can Forward:

- RAW dependence within 2 instructions:
- Load to RegA then immediately after storing from RegA to mem

Data Hazard - Forwarding Paths:

- For immediate RAW dependence: EX/MEM pipeline reg -> ALU input
- For 2 instruction RAW dependence: MEM/WB pipeline reg -> ALU input
- For immediate RAW (load-store) dependence: MEM/WB pipeline reg -> MEM input

Data Hazards - Need to stall:

- RAW dependence on a LOAD immediately before it. Stall for 1 cycle

Control Hazard - Detect and Stall:

- Branches result in a 1-2 cycle stall, depending on the ISA (if branch target is always known after ID, then 1 cycle stall; if branch target is known after EX, then 2 cycles stall)

3.9 References

- Patterson & Hennesy - Appendix C
- [CS4617 - L9](#)

4

5

6

7

8

9

10

11

12

13

14 Summary

In summary...

References