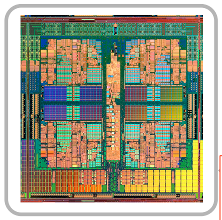
Lecture 17
Short Pipelining Review

Suggested Readings

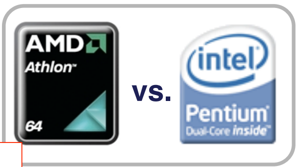
- Readings
  - H&P: Chapter 4.5-4.7
    - (Over the next 3-4 lectures)

Processor components

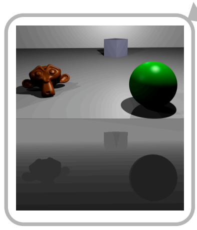
Multicore processors and programming



Processor comparison

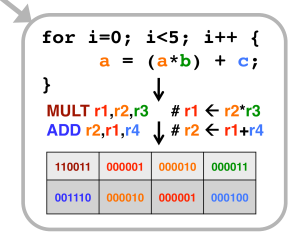


Goal:
Describe the fundamental components required in a single core of a modern microprocessor as well as how they interact with each other, with main memory, and with external storage media.



Writing more efficient code

The right HW for the right application



```
for i=0; i<5; i++ {
        a = (a*b) + c;
}
MULT r1,r2,r3     # r1 ← r2*r3
ADD r2,r1,r4      # r2 ← r1+r4
```

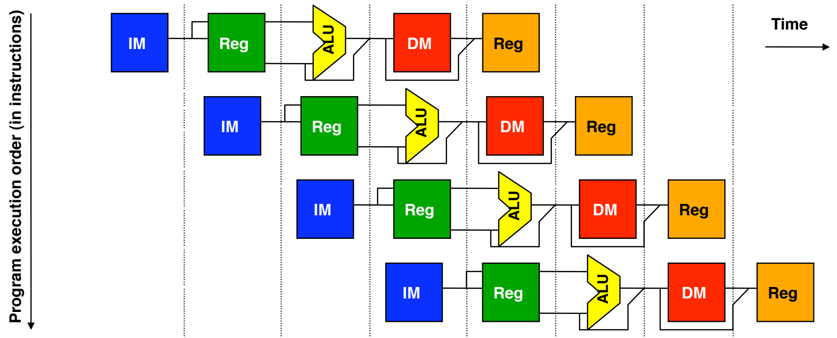
110011	000001	000010	000011
001110	000010	000001	000100

HLL code translation

Recap: Pipelining improves throughput

Clock Number

Inst. #	1	2	3	4	5	6	7	8
Inst. i	IF	ID	EX	MEM	WB			
Inst. i+1		IF	ID	EX	MEM	WB		
Inst. i+2			IF	ID	EX	MEM	WB	
Inst. i+3				IF	ID	EX	MEM	WB



Recap: pipeline math

- If times for all S stages are equal to T:
  - Time for one initiation to complete still ST
  - Time between 2 initiates = T not ST
  - Initiations per second = 1/T

Time for N initiations to complete:	NT + (S-1)T
Throughput:	Time per initiation = T + (S-1)T/N → T!

- Pipelining: Overlap multiple executions of same sequence
  - Improves THROUGHPUT, not the time to perform a single operation

Recap: Stalls and performance

- Stalls impede progress of a pipeline and result in deviation from 1 instruction executing/clock cycle
- Pipelining can be viewed to:
  - Decrease CPI or clock cycle time for instruction
  - Let's see what affect stalls have on CPI…
- CPI pipelined =
  - Ideal CPI + Pipeline stall cycles per instruction
  - 1 + Pipeline stall cycles per instruction
- Ignoring overhead and assuming stages are balanced:

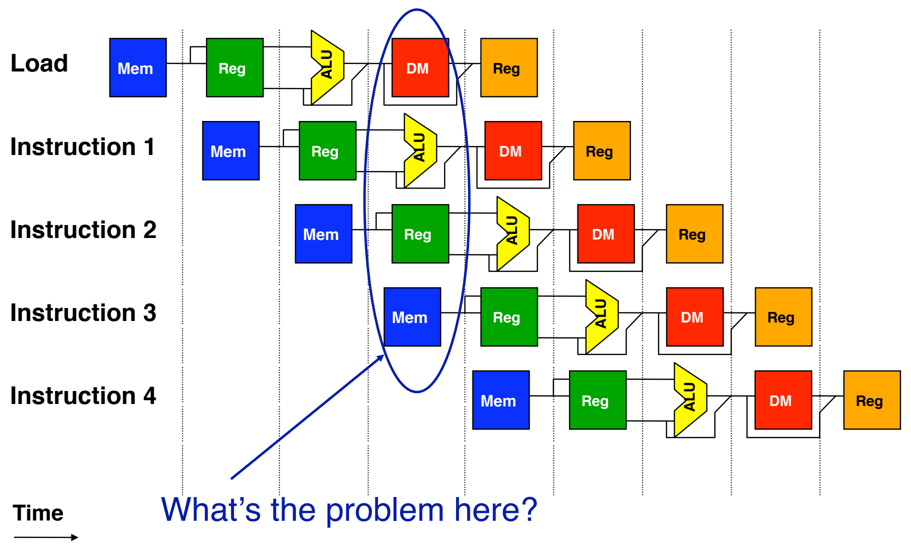
$$Speedup = \frac{CPI\ unpipelined}{1 + pipeline\ stall\ cycles\ per\ instruction}$$

- If no stalls, speedup equal to # of pipeline stages in ideal case

Recap: Structural hazards

- 1 way to avoid structural hazards is to duplicate resources
  - i.e.: An ALU to perform an arithmetic operation and an adder to increment PC
- If not all possible combinations of instructions can be executed, structural hazards occur
- Most common instances of structural hazards:
  - When a functional unit not fully pipelined
  - When some resource not duplicated enough
- Pipelines stall result of hazards, CPI increased from the usual "1"

Recap: Structural hazard example



Recap: Data hazards

- These exist because of pipelining
- Why do they exist???
  - Pipelining changes order or read/write accesses to operands
  - Order differs from order seen by sequentially executing instructions on un-pipelined machine
- Consider this example:
  - ADD R1, R2, R3
  - SUB R4, R1, R5
  - AND R6, R1, R7
  - OR R8, R1, R9
  - XOR R10, R1, R11

All instructions after ADD use result of ADD

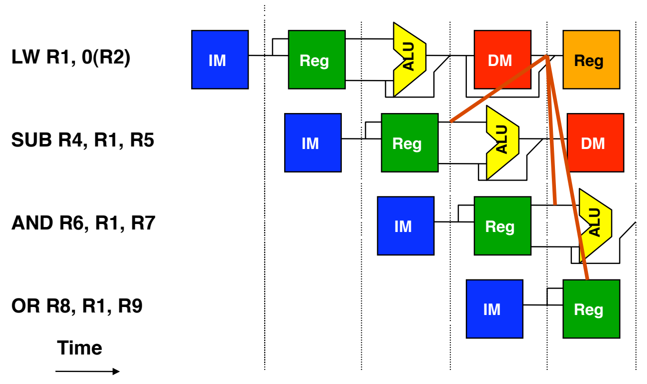
ADD writes the register in WB but SUB needs it in ID.

<u>This is a data hazard</u>

Recap: Forwarding & data hazards

- Problem illustrated on previous slide can actually be solved relatively easily – with <u>forwarding</u>
- In this example, result of the ADD instruction not <u>really</u> needed until after ADD actually produces it
- Can we move the result from EX/MEM register to the beginning of ALU (where SUB needs it)?
  - Yes! Hence this slide!
- Generally speaking:
  - Forwarding occurs when a result is passed directly to functional unit that requires it.
  - Result goes from output of one unit to input of another

Recap: Forwarding doesn't always work

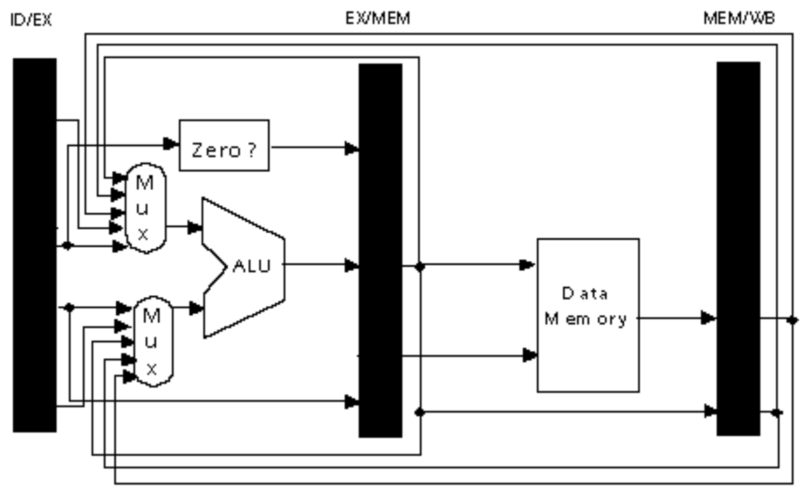


Load has a latency that forwarding can't solve.

Pipeline must stall until hazard cleared (starting with instruction that wants to use data until source produces it).

Can't get data to subtract b/c result needed at beginning of CC #4, but not produced until end of CC #4.

Recap: HW change for forwarding



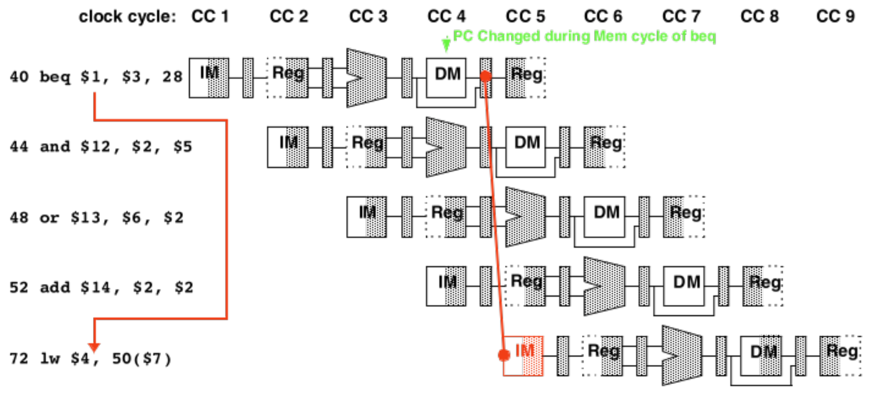
Recap: Hazards vs. dependencies

- dependence: fixed property of instruction stream
  - (i.e., program)
- hazard: property of program and processor organization
  - implies potential for executing things in wrong order
    - potential only exists if instructions can be simultaneously "in-flight"
    - property of dynamic distance between instructions vs. pipeline depth
- For example, can have RAW dependence with or without hazard
  - depends on pipeline

Recap: Branch/Control Hazards

- So far, we've limited discussion of hazards to:
  - Arithmetic/logic operations
  - Data transfers
- Also need to consider hazards involving branches:
  - Example:
    - 40: beq $1, $3, $28 # ($28 gives address 72)
    - 44: and $12, $2, $5
    - 48: or $13, $6, $2
    - 52: add $14, $2, $2
    - 72: lw $4, 50($7)
- How long will it take before the branch decision takes effect?
  - What happens in the meantime?

Recap: How branches impact a pipeline



- If branch condition true, must skip 44, 48, 52
  - But, these have already started down the pipeline
  - They will complete unless we do something about it
- How do we deal with this?
  - We'll consider 2 possibilities

Recap: Or assume branch is *not taken*

- On average, branches are taken ½ the time
  - If branch not taken…
    - Continue normal processing
  - Else, if branch is taken…
    - Need to flush improper instruction from pipeline
- Cuts overall time for branch processing in ½

Recap: Branch penalty impact

- Assume 16% of all instructions are branches
  - 4% unconditional branches: 3 cycle penalty
  - 12% conditional: 50% taken
- For a sequence of N instructions (assume N is large)
  - N cycles to initiate each
  - 3 * 0.04 * N delays due to unconditional branches
  - 0.5 * 3 * 0.12 * N delays due to conditional taken
  - Also, an extra 4 cycles for pipeline to empty
- Total:
  - 1.3*N + 4 total cycles (or 1.3 cycles/instruction) (CPI)
    - 30% Performance Hit!!! (Bad thing)

Lecture 17
Wrap-up Discussion of Hazards

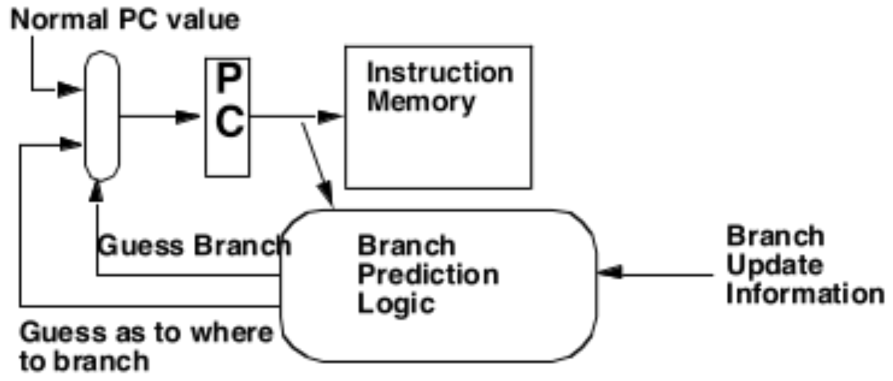
Branch Penalty Impact

- Some solutions:
  - In ISA: branches always execute next 1 or 2 instructions
    - Instruction so executed said to be in delay slot
    - See SPARC ISA
    - (example – loop counter update)
  - In organization: move comparator to ID stage and decide in the ID stage
    - Reduces branch delay by 2 cycles
    - Increases the cycle time

Branch Prediction

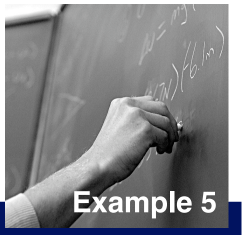
- Prior solutions are "ugly"
- Better (& more common): guess in IF stage
  - Technique is called "branch predicting"; needs 2 parts:
    - "Predictor" to guess where/if instruction will branch (and to where)
    - "Recovery Mechanism": i.e. a way to fix your mistake
  - Prior strategy:
    - Predictor: always guess branch never taken
    - Recovery: flush instructions if branch taken
  - Alternative: accumulate info. in IF stage as to…
    - Whether or not for any particular PC value a branch was taken next
    - To where it is taken
    - How to update with information from later stages

A Branch Predictor

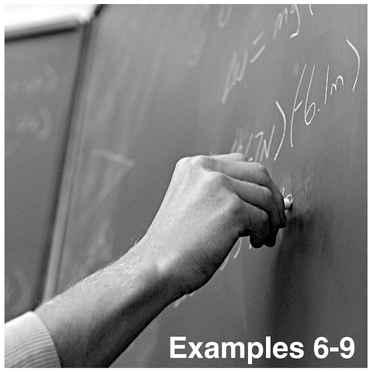


Computing Performance

- Program assumptions:
  - 23% loads and in ½ of cases, next instruction uses load value
  - 13% stores
  - 19% conditional branches
  - 2% unconditional branches
  - 43% other
- Machine Assumptions:
  - 5 stage pipe with all forwarding
    - Only penalty is 1 cycle on use of load value immediately after a load)
    - Jumps are totally resolved in ID stage for a 1 cycle branch penalty
    - 75% branch prediction accuracy
    - 1 cycle delay on misprediction


Example 5

Examples…

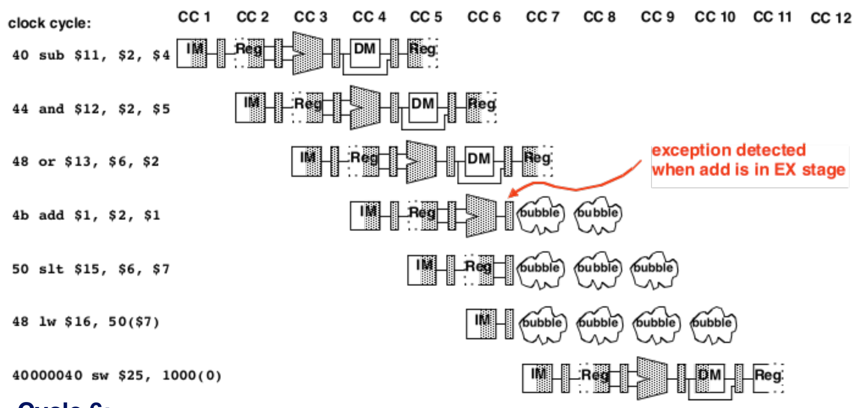

Examples 6-9

Exception Hazards

• 40_{hex}:	sub	\$11, \$2, \$4	
• 44_{hex}:	and	\$12, \$2, \$5	
• 48_{hex}:	or	\$13, \$6, \$2	
• $4b_{hex}$:	add	\$1, \$2, \$1	(overflow in EX stage)
• 50_{hex}:	slt	\$15, \$6, \$7	(already in ID stage)
• 54_{hex}:	lw	\$16, 50(\$7)	(already in IF stage)
• …			
• 40000040_{hex}:	sw	\$25, 1000(\$0)	exception handler
• 40000044_{hex}:	sw	\$26, 1004(\$0)	

- Need to transfer control to exception handler ASAP
  - Don't want invalid data to contaminate registers or memory
  - Need to flush instructions already in the pipeline
  - Start fetching instructions from 40000040_{hex}
  - Save addr. following offending instruction (50_{hex}) in TrapPC (EPC)
  - Don't clobber \$1 – use for debugging

Flushing pipeline after exception

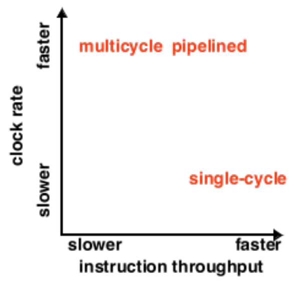


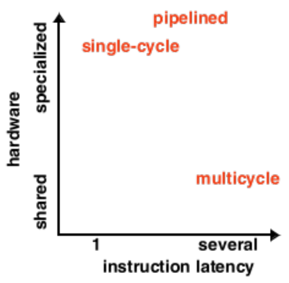
- Cycle 6:
  - Exception detected, flush signals generated, bubbles injected
- Cycle 7
  - 3 bubbles appear in ID, EX, MEM stages
  - PC gets 40000040_{hex}, TrapPC gets 50_{hex}

Discussion

- How does instruction set design impact pipelining?
- Does increasing the depth of pipelining always increase performance?

Comparative Performance





- Throughput: instructions per clock cycle = 1/cpi
  - Pipeline has fast throughput and fast clock rate
- Latency: inherent execution time, in cycles
  - High latency for pipelining causes problems
    - Increased time to resolve hazards

Summary

- Performance:
  - Execution time *or* throughput
  - Amdahl's law
- Multi-bus/multi-unit circuits
  - one long clock cycle or N shorter cycles
- Pipelining
  - overlap independent tasks
- Pipelining in processors
  - "hazards" limit opportunities for overlap