Computer Science 146
Computer Architecture

Fall 2019

Harvard University

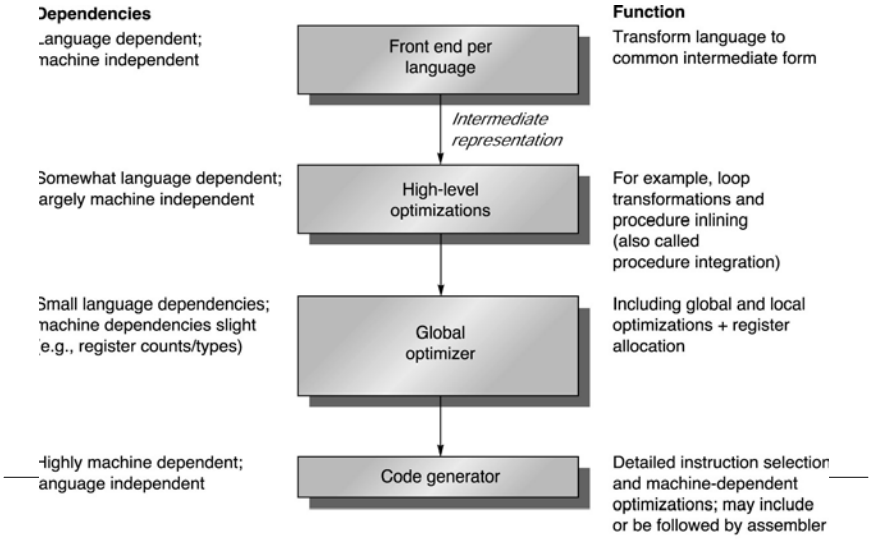
Instructor: Prof. David Brooks
dbrooks@eecs.harvard.edu

Lecture 4: Basic Implementation and Pipelining

Lecture Outline

- Finish ISAs (Compiler Impact)
- Basic Implementation Review
- Advantages of Pipelining
- "Easy" Challenges of Pipelining
  - Hazards (Structural, Data, Control)
- "Hard" Challenges of Pipelining
  - Maintaining Precise Exceptions

Compilers 101



Compiler Optimizations

- High-level optimizations
  - Done on source, may be source-to-source conversions
  - Examples – map data for cache efficiency, remove conditions, etc.
- Local Optimizations
  - Optimize code in small straight-line sections
- Global Optimizations
  - Extend local opts across branches and do loop optimizations (loop unrolling)
- Register Allocation
  - Assign temporary values to registers, insert spill code

Compilers and the ISA

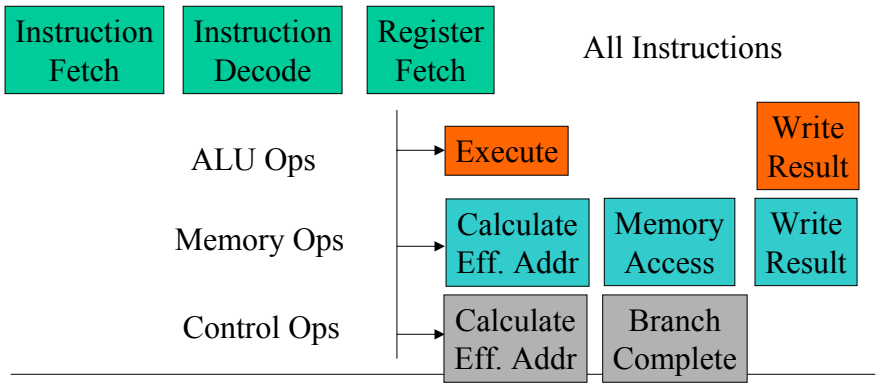
- Architects can help compiler writers
  - Providing regularity (already discussed)
  - Primitives, not solutions (HLL-support has not succeeded)
  - Simplify trade-offs among alternatives
  - Provide instructions that bind compile-time constants

Compiler support for MM ISAs

- Actually there is very little
- Surprising because vector-computers have good compiler support
- Problems
  - Short, architecture-limited vectors
  - Few registers and simple addressing modes
    - Vector machines support strided addressing and gather/scatters
  - Most programming languages don't support subwords
  - Results: Only some kernels tend to be handcoded

Implementation Review

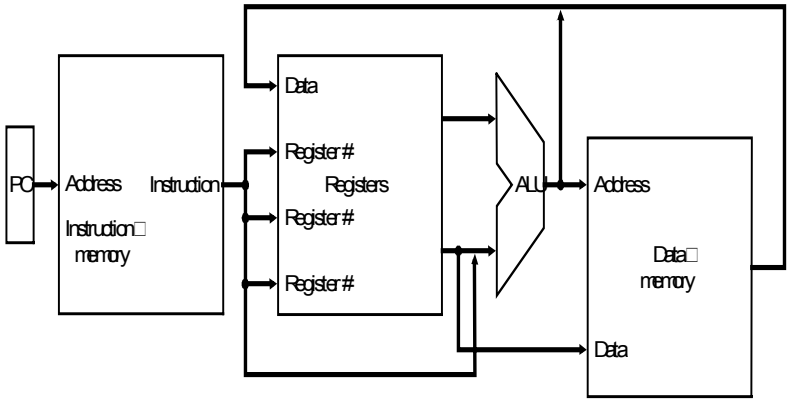
- First, let's think about how different instructions get executed



Instruction Fetch

- Send the Program Counter (PC) to memory
- Fetch the current instruction from memory
  - IR <= Mem[PC]
- Update the PC to the next sequential
  - PC <= PC + 4 (4-bytes per instruction)
- Optimizations
  - Instruction Caches, Instruction Prefetch
- Performance Affected by
  - Code density, Instruction size variability (CISC/RISC)

Abstract Implementation

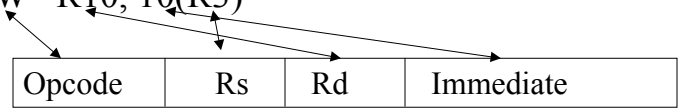


Instruction Decode/Reg Fetch

- Decide what type of instruction we have
  - ALU, Branch, Memory
  - Decode Opcode
- Get operands from Reg File
  - A <= Regs[$IR_{25..21}$]; B <= Regs[$IR_{20..16}$];
  - Imm <= SignExtend($IR_{15..0}$)
- Performance Affected by
  - Regularity in instruction format, instruction length

Calculate Effective Address: Memory Ops

- Calculate Memory address for data
- ALU_{output} <= A + Imm
- LW R10, 10(R3)



Calculate Effective Address: Branch/Jump Ops

- Calculate target for branch/jump operation
- BEQZ, BNEZ, J
  - ALU_{output} <= NPC + Imm; cond <= A op 0
  - "op" is a check against 0, equal, not-equal, etc.
  - J is an unconditional
- ALU_{output} <= A

Execution: ALU Ops

- Perform the computation
- Register-Register
  - ALU_{output} <= A op B
- Register-Immediate
  - ALU_{output} <= A op Imm
- No ops need to do effective address calc *and* perform an operation on data
- Why?

Memory Access

- Take effective address, perform Load or Store
- Load
  - LMD <= Mem[ALU_{output}]
- Store
  - Mem[ALU_{output}] <= B

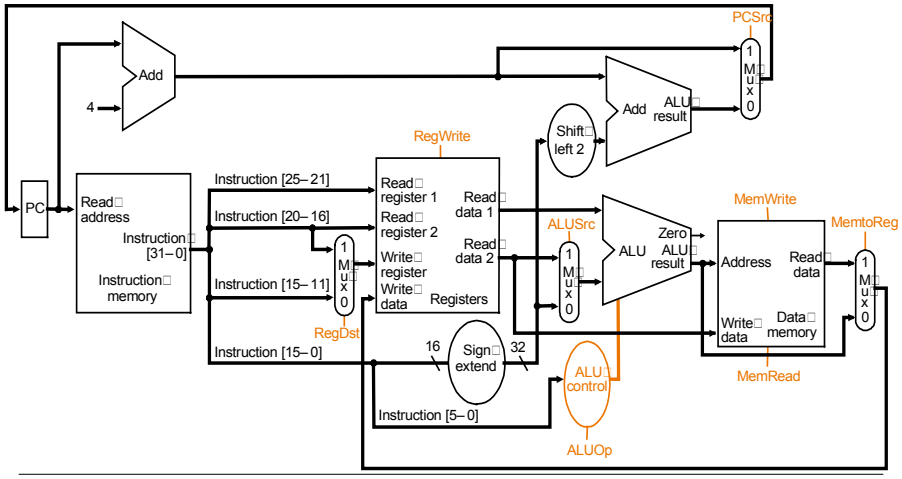
Mem Phase on Branches

- Set PC to the calculated effective address
- BEQZ, BNEZ
  - If (cond) PC <= ALU_{output} else PC <= NPC

Write-Back

- Send results back to register file
- Register-register ALU instructions
  - $Regs[IR_{15..11}] <= ALU_{output}$
- Register-Immediate ALU instruction
  - $Regs[IR_{20..16}] <= ALU_{output}$
- Load Instruction
  - $Regs[IR_{20..16}] <= LMD$
- Why does this have to be a separate step?

Final Implementation

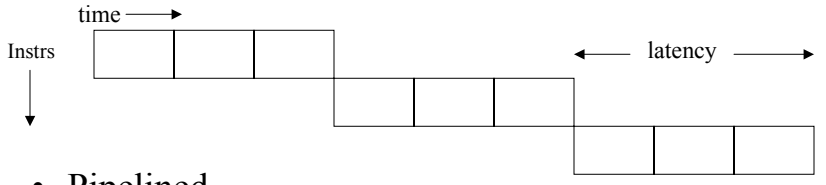


What is Pipelining?

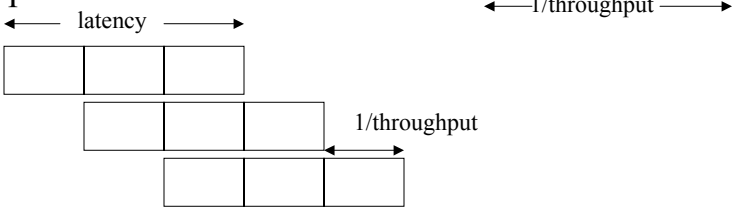
- Implementation where multiple instructions are simultaneously overlapped in execution
  - Instruction processing has N different stages
  - Overlap different instructions working on different stages
- Pipelining is not new
  - Ford's Model-T assembly line
  - Laundry – Washer/Dryer
  - IBM Stretch [1962]
  - Since the '70s nearly all computers have been pipelined

Pipelining Advantages

- Unpipelined



- Pipelined

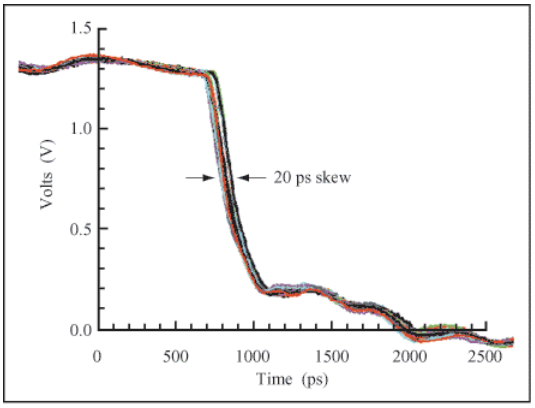


Ideal Pipelining Performance

- Assume instruction execution takes N stages
  - Each stage takes t_n time
  - Single Instruction latency, $T = \Sigma t_n$
  - Throughput = 1/T
  - M-Instruction Latency = M/T
- For an N-stage pipeline
  - Single Instruction latency, $T = \Sigma t_n$
  - Throughput = $1/\max(t_n) <= T/N$ (unless all t_n are equal)
  - M-instruction Latency = $M * \max(t_n) <= M*T/N$
- $CPI_{Ideal} = \frac{CPI_{withoutpipeline}}{\text{Pipeline Depth}}$

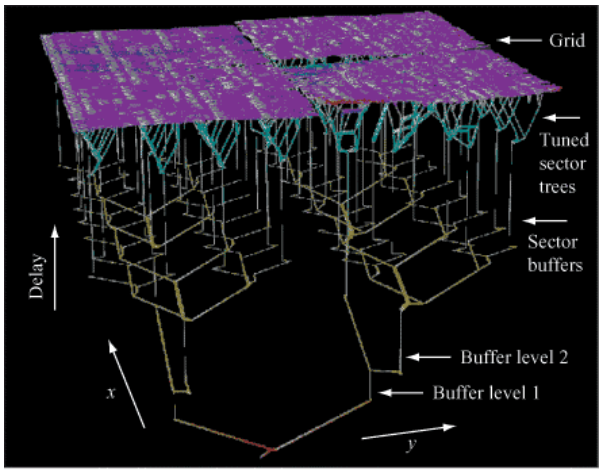
Why is this not the case?

- Two things we are missing
  - Pipelining overhead (latches, clock skew, jitter)
    - This eats into the maximum speedup
  - Hazards
- $CPI_{Real} = CPI_{Ideal} + CPI_{Stall}$

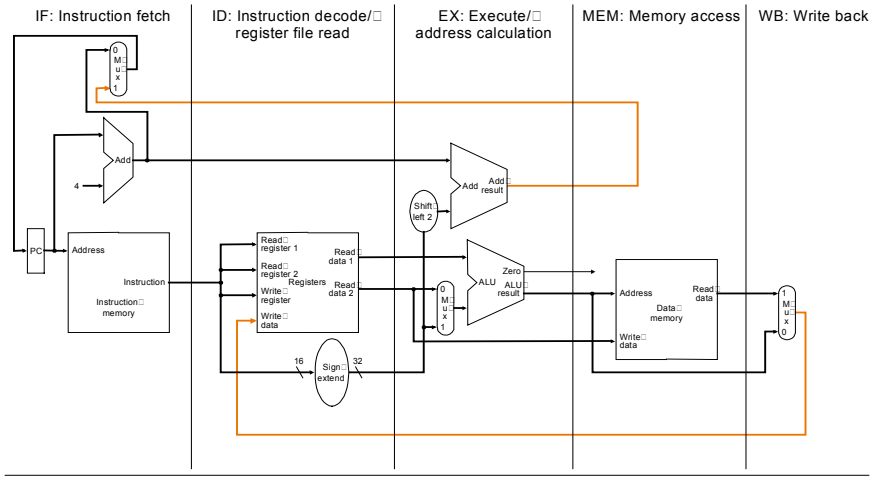


IBM POWER4 Clock Skew Measurement

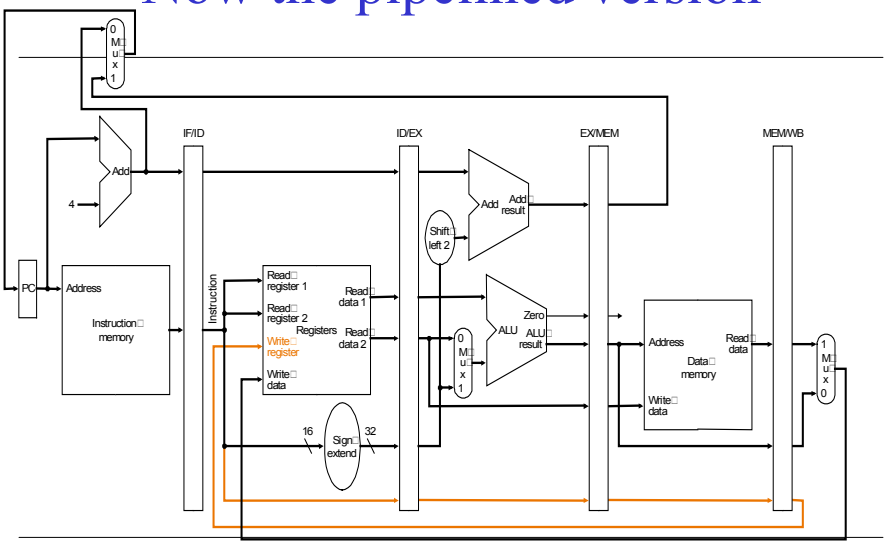
How are clocks distributed? POWER4 Clock Distribution Net



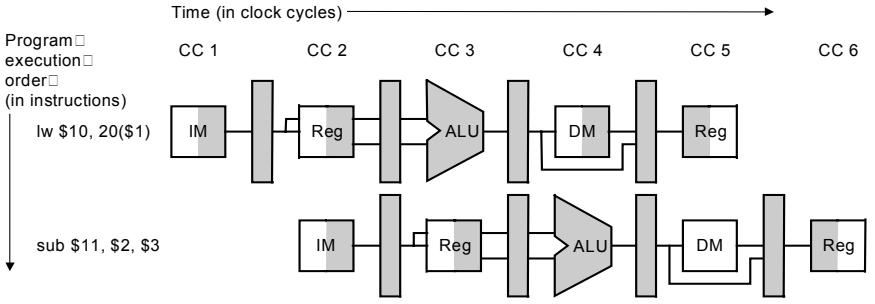
Recall from Earlier…



Now the pipelined version



Representation of Pipelines



lw R10, 20(R1)	IF	ID	EX	MEM	WB	
Sub R11, R2, R3		IF	ID	EX	MEM	WB

Pipeline Hazards

- Hazards
  - Situations that prevent the next instruction from executing in its designated clock cycle
- Structural Hazards
  - When two different instructions want to use the same hardware resource in the same cycle (resource conflict)
- Data Hazards
  - When an instruction depends on the result of a previous instruction that exposes overlapping of instructions
- Control Hazards
  - Pipelining of PC-modifying instructions (branch, jump, etc)

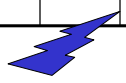
How to resolve hazards?

- Simple Solution: Stall the pipeline
  - Stops some instructions from executing
  - Make them wait for older instructions to complete
  - Simple implementation to "freeze" (de-assert write-enable signals on pipeline latches)
  - Inserts a "bubble" into the pipe
  - Must propagate upstream as well! Why?

Structural Hazards

- Two cases when this can occur
  - Resource used more than once in a cycle (Memory, ALU)
  - Resource is not fully pipelined (FP Unit)
- Imagine that our pipeline shares I- and D-memory

lw R10, 10(R1)	IF	ID	EX	MEM	WB			
sub R11, R2, R3		IF	ID	EX	MEM	WB		
add R12, R4, R5			IF	ID	EX	MEM	WB	
add R13, R6, R7				IF	ID	EX	MEM	WB



Structural Hazards Solutions

- Stall
  - Low Cost, Simple (+)
  - Increases CPI (-)
  - Try to use for rare events in high-performance CPUs
- Duplicate Resources
  - Decreases CPI (+)
  - Increases cost (area), possibly cycle time (-)
  - Use for cheap resources, frequent cases
    - Separate I-, D-caches, Separate ALU/PC adders, Reg File Ports

Structural Hazards Solutions

- Pipeline Resources
  - High performance (+)
  - Control is simpler than duplication (+)
  - Tough to pipeline some things (RAMs) (-)
  - Use when frequency makes it worthwhile
  - Ex. Fully pipelined FP add/multiplies critical for scientific
- Good news
  - Structural hazards don't occur as long as each instruction uses a resource
    - At most once
    - Always in the same pipeline stage
    - For one cycle
  - RISC ISAs are designed with this in mind, reduces structural hazards

Pipeline Stalls

- What could the performance impact of unified instruction/data memory be?

Loads ~15% of instructions, Stores ~10%

Prob (Ifetch + Dfetch) = .25

$CPI_{Real} = CPI_{Ideal} + CPI_{Stall} = 1.0 + .25 = 1.25$

Data Hazards

- Two operands from different instructions use the *same* storage location
- Must appear as if instructions are executed to completion one at a time
- Three types of Data Hazards
  - Read-After-Write (RAW)
    - True data-dependence (Most important)
  - Write-After-Read (WAR)
  - Write-After-Write (WAW)

RAW Example

Cycle	1	2	3	4	5	6	7	8
Add R3, R2, R1	IF	ID	EX	MEM	WB			
Add R4, R3, R5		IF	ID	EX	MEM	WB		
Add R6, R3, R5			IF	ID	EX	MEM	WB	
Add R7, R3, R5				IF	ID	EX	MEM	WB

- First Add writes to R3 in cycle 5
- Second Add reads R3 in cycle 3
- Third Add reads R3 in cycle 4
  - We would compute the wrong answer because R3 holds the "old" value

Solutions to RAW Hazards

- As usual, we have a couple of choices
- Stall whenever we have a RAW
  - Huge performance penalty, dependencies are common!
- Use Bypass/Forwarding to minimize the problem
  - Data is ready by end of EXE (Add) or MEM (Load)
  - Basic idea:
    - Add comparator for each combination of destination and source registers that can have RAW hazards (How many?)
    - Add muxes to datapath to select proper value instead of regfile
  - Only stall when absolutely necessary

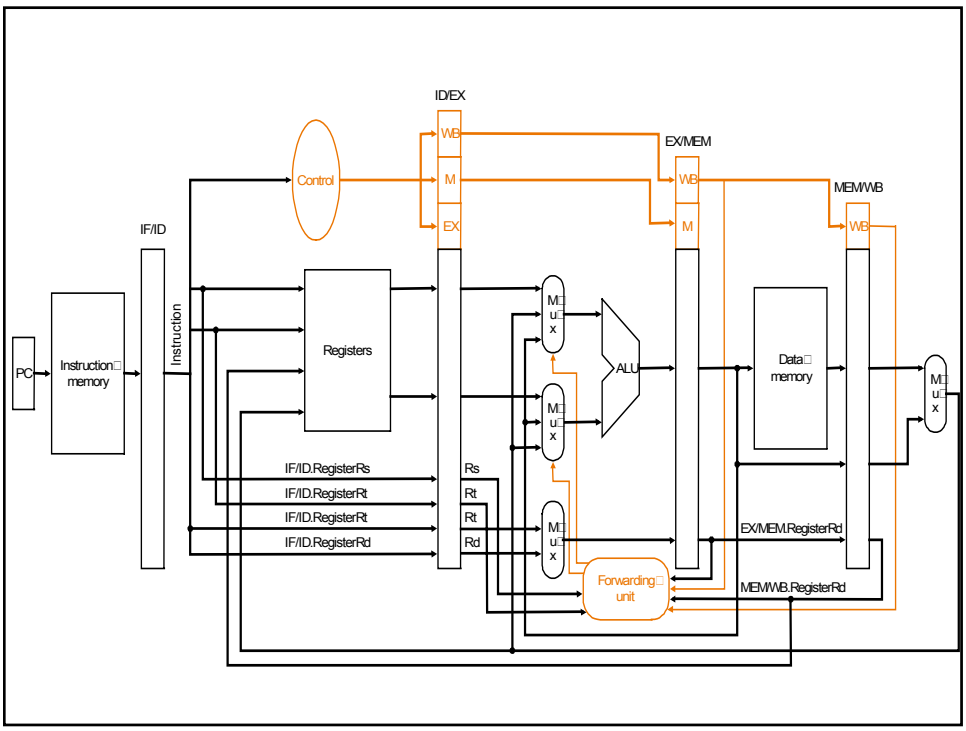
Solutions to RAW Hazards: Pipeline Interlocks

- Two part problem: Detect the RAW, forward/stall the pipe
  - Need to keep register ID's along with pipestages
  - Use comparators to check for hazards
- Operand 2 bypass ADD R1, R2, R3

  If (R3 == RD(MEM)) use ALUOUT(MEM)

  else (if R3 == RD(WB)) use ALUOUT (WB)

  else Use R3 from Register File



Forwarding, Bypassing

Cycle	1	2	3	4	5	6	7	8
Add R3, R2, R1	IF	ID	EX	MEM	WB			
Add R4, R3, R5		IF	ID	EX	MEM	WB		
Add R6, R3, R5			IF	ID	EX	MEM	WB	
Add R7, R3, R5				IF	ID	EX	MEM	WB

- Code is now “stall-free”
- Are there any cases where we must stall?

Load Use Hazards

Cycle	1	2	3	4	5	6	7	8
lw R3, 10(R1)	IF	ID	EX	MEM	WB			
Add R4, R3, R5		IF	ID	EX	MEM	WB		
Add R6, R3, R5			IF	ID	EX	MEM	WB	

- Unfortunately, we can’t forward “backward in time”

Cycle	1	2	3	4	5	6	7	8
lw R3, 10(R1)	IF	ID	EX	MEM	WB			
Add R4, R3, R5		IF	ID	stall	EX	MEM	WB	
Add R6, R3, R5			IF	stall	ID	EX	MEM	WB

Load Use Hazards

- Can the compiler help out?
  - Scheduling to avoid load followed by immediate use
- "Delayed Loads"
  - Define the pipeline slot after a load to be a "delay slot"
  - NO interlock hardware. Machine assumes the correct compiler
- Compiler attempts to schedule code to fill delay slots
- Limits to this approach:
  - Only can reorder between branches (5-6 instructions)
  - Order of loads/stores difficult to swap (alias problems)
  - Makes part of implementation *architecturally visible*

Instruction Scheduling Example

a = b +c;
d = e – f;

How many cycles for each?

No Scheduling Version	Scheduled Version
`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`

Other Data Hazards: WARs

- Write-After-Read (WAR) Hazards
  - Can't happen in our simple 5-stage pipeline because writes always follow reads
  - Preview: Late read, early write (auto-increment)
    - i DIV (R1), --, --
    - i+1 ADD --, R1+, --
  - Preview: Out-of-Order reads (OOO-execution)

Other Data Hazards: WAWs

- Write-After-Write (WAW) Hazards
  - Can't happen in our simple 5-stage pipeline because only one writeback stage (ALU ops go through MEM stage)
  - Preview: Slow operation followed by fast operation
    - i DIVF F0, --, --
    - i+1 BFPT --, --, --
    - i+2 ADDF F0, --, --
  - Also cache misses (they can return at odd times)
- What about RARs?

Control Hazards

Cycle	1	2	3	4	5	6	7	8
Branch Instr.	IF	ID	EX	MEM	WB			
Instr +1		IF	stall	stall	IF	ID	EX	MEM
Instr +2			stall	stall	stall	IF	ID	EX

- In base pipeline, branch outcome not known until MEM
- Simple solution – stall until outcome is known
- Length of control hazard is branch delay
  - In this simple case, it is 3 cycles (assume 10% cond. branches)
  - $CPI_{Real} = CPI_{Ideal} + CPI_{Stall} = 1.0 + 3 \text{ cycles} * .1 = 1.3$

Control Hazards: Solutions
Fast Branch Resolution

- Performance penalty could be more than 30%
  - Deeper pipelines, some code is very branch heavy
- Fast Branch Resolution
  - Adder in ID for PC + immediate targets
  - Only works for simple conditions (compare to 0)
  - Comparing two register values could be too slow

Cycle	1	2	3	4	5	6	7	8
Branch Instr.	IF	ID	EX	MEM	WB			
Instr +1		stall	IF	ID	EX	MEM	WB	
Instr +2			stall	IF	ID	EX	MEM	WB

Control Hazards: Branch Characteristics

- Integer Benchmarks: 14-16% instructions are conditional branches
- FP: 3-12%
- On Average:
  - 67% of conditional branches are "taken"
  - 60% of forward branches are taken
  - 85% of backward branches are taken
  - Why?

Control Hazards: Solutions

1. Stall Pipeline
   - Simple, No backing up, No Problems with Exceptions
2. Assume not taken
   - Speculation requires back-out logic:
     - What about exceptions, auto-increment, etc
   - Bets the "wrong way"
3. Assume taken
   - Doesn't help in simple pipeline! (don't know target)
4. Delay Branches
   - Can help a bit… we'll see pro's and con's soon

Control Hazards: Assume Not Taken

Cycle	1	2	3	4	5	6	7	8
Untaken Branch	IF	ID	EX	MEM	WB			
Instr +1		IF	ID	EX	MEM	WB		
Instr +2			IF	ID	EX	MEM	WB	

Looks good if we're right!

Cycle	1	2	3	4	5	6	7	8
Taken Branch	IF	ID	EX	MEM	WB			
Instr +1		IF	flush	flush	flush	flush		
Branch Target			IF	ID	EX	MEM	WB	
Branch Target +1				IF	ID	EX	MEM	WB

Control Hazards: Branch Delay Slots

- Find one instruction that will be executed no matter which way the branch goes
- Now we don't care which way the branch goes!
  - Harder than it sounds to find instructions
- What to put in the slot (80% of the time)
  - Instruction from before the branch (indep. of branch)
  - Instruction from taken or not-taken path
    - Always safe to execute? May need clean-up code (or nullifying branches)
    - Helps if you go the right way
- Slots don't help much with today's machines
  - Interrupts are more difficult (why? We'll see soon)

Now for the hard stuff!

- Precise Interrupts
  - What are interrupts?
  - Why do they have to be precise?

- Must have well-defined state at interrupt
  - All older instructions are complete
  - All younger instructions have not started
  - All interrupts are taken in program order

Interrupt Taxonomy

- Synchronous vs. Asychronous (HW error, I/O)
- User Request (exception?) vs. Coerced
- User maskable vs. Nonmaskable (Ignorable)
- Within vs. Between Instructions
- Resume vs. Terminate

The difficult exceptions are *resumable* interrupts *within* instructions

- Save the state, correct the cause, restore the state, continue execution

Interrupt Taxonomy

Exception Type	Sync vs. Async	User Request Vs. Coerced	User mask vs. nommask	Within vs. Between Insn	Resume vs. terminate
I/O Device Req.	Async	Coerced	Nonmask	Between	Resume
Invoke O/S	Sync	User	Nonmask	Between	Resume
Tracing Instructions	Sync	User	Maskable	Between	Resume
Breakpoint	Sync	User	Maskable	Between	Resume
Arithmetic Overflow	Sync	Coerced	Maskable	Within	Resume
Page Fault (not in main m)	Sync	Coerced	Nonmask	Within	Resume
Misaligned Memory	Sync	Coerced	Maskable	Within	Resume
Mem. Protection Violation	Sync	Coerced	Nonmask	Within	Resume
Using Undefined Insns	Sync	Coerced	Nonmask	Within	Terminate
Hardware/Power Failure	Async	Coerced	Nonmask	Within	Terminate

Interrupts on Instruction Phases

Exception Type	IF	ID	EXE	MEM	WB
Arithmetic Overflow			X		
Page Fault (not in main memory)	X			X	
Misaligned Memory	X			X	
Mem. Protection Violation	X			X	

- Exceptions can occur on many different phases
- However, exceptions are only handled in WB
- Why?

load IF ID EX MEM WB
add IF ID EX MEM WB

How to take an exception?

1. Force a trap instruction on the next IF
2. Squash younger instructions (Turn off all writes (register/memory) for faulting instruction and all instructions that follow it)
3. Save all processor state after trap begins
   - PC-chain, PSW, Condition Codes, trap condition
   - PC-chain is length of the branch delay plus 1
4. Perform the trap/exception code then restart where we left off

Summary of Exceptions

- Precise interrupts are a headache!
- All architected state must be precise
- Delayed branches
- Preview: Out-of-Order completion
  - What if something writes-back earlier than the exception?
- Some machines punt on the problem
  - Precise exceptions only for integer pipe
  - Special “precise mode” used for debugging (10x slower)

For next time

- Multi-cycle operations
  - More WAR, WAW nastiness
  - More precise interrupt nastiness
- SuperScalar/Dynamic Scheduling