ECE/CS 552: Midterm Review

Instructor: Mikko H Lipasti

Fall 2010
University of Wisconsin-Madison

Lecture notes based on notes by Mark Hill and John P. Shen
Updated by Mikko Lipasti

Computer Architecture

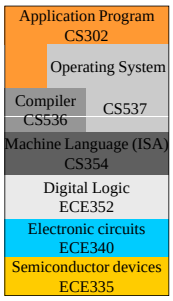
- Exercise in engineering tradeoff analysis
  - Find the fastest/cheapest/power-efficient/etc. solution
  - Optimization problem with 100s of variables
- All the variables are changing
  - At non-uniform rates
  - With inflection points
  - Only one guarantee: Today's right answer will be wrong tomorrow
- Two high-level effects:
  - Technology push
  - Application Pull

Abstraction

- Difference between interface and implementation
  - Interface: WHAT something does
  - Implementation: HOW it does so

What's the Big Deal?

- Tower of abstraction
- Complex interfaces implemented by layers below
- Abstraction hides detail
- Hundreds of engineers build one product
- Complexity unmanageable otherwise



Performance vs. Design Time

- Time to market is critically important
- E.g., a new design may take 3 years
  - It will be 3 times faster
  - But if technology improves 50%/year
  - In 3 years $1.5^3 = 3.38$
  - So the new design is worse!
    (unless it also employs new technology)

Bottom Line

- Designers must know BOTH software and hardware
- Both contribute to layers of abstraction
- IC costs and performance
- Compilers and Operating Systems

Performance

- Time and performance: Machine A n times faster than Machine B
  - Iff Time(B)/Time(A) = n
- Iron Law: Performance = Time/program =

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

(code size) (CPI) (cycle time)

Performance cont'd

- Other Metrics: MIPS and MFLOPS
  - Beware of peak and omitted details
- Benchmarks: SPEC2000 (95 in text)
- Summarize performance:
  - AM for time
  - HM for rate
  - GM for ratio
- Amdahl's Law:

$$Speedup = \frac{1}{1 - f + \frac{f}{s}}$$

Ch 2 Summary

- Basics
- Registers and ALU ops
- Memory and load/store
- Branches and jumps
- Addressing Modes

Summary: Instruction Formats

R:	opcode	rs	rt	rd	shamt	function
	6	5	5	5	5	6
I:	opcode	rs	rt	address/immediate		
	6	5	5	16		
J:	opcode	addr				
	6	26				

- Instruction decode:
  - Read instruction bits
  - Activate control signals

Conclusions

- Simple and regular
  - Constant length instructions, fields in same place
- Small and fast
  - Small number of operands in registers
- Compromises inevitable
  - Pipelining should not be hindered
- Make common case fast!
- Backwards compatibility!

Basic Arithmetic and the ALU

- Number representations: 2's complement, unsigned
- Addition/Subtraction
- Add/Sub ALU
  - Full adder, ripple carry, subtraction
- Carry-lookahead addition
- Logical operations
  - and, or, xor, nor, shifts
- Overflow

Unsigned Integers

- f(b31..b0) = b31 x 2^{31} + … + b1 x 2^1 + b0 x 2^0
- Treat as normal binary number
  E.g. 0…01101010101
  = 1 x 2^7 + 1 x 2^6 + 0 x 2^5 + 1 x 2^4 + 1 x 2^3 + 0 x 2^1 + 1 x 2^0
  = 128 + 64 + 16 + 4 + 1 = 213
- Max f(111…11) = 2^{32} – 1 = 4,294,967,295
- Min f(000…00) = 0
- Range [0,2^{32}-1] => # values (2^{32}-1) – 0 + 1 = 2^{32}

Signed Integers

- 2's complement
  f(b31 … b1 b0) = -b31 x 2^{31} + … b1 x 2^1 + b0 x 2^0
- Max f(0111…11) = 2^{31} – 1 = 2147483647
- Min f(100…00) = -2^{31} = -2147483648 (asymmetric)
- Range[-2^{31},2^{31}-1] => # values(2^{31}-1 - -2^{31} _ 1) = 2^{32}
- E.g. –6
  – 000…0110 => 111…1001 + 1 => 111…1010

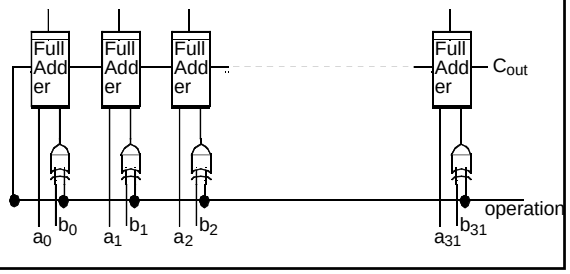
Full Adder

- Full adder (a,b,c_{in}) => (c_{out}, s)
- c_{out} = two or more of (a, b, c_{in})
- s = exactly one or three of (a,b,c_{in})

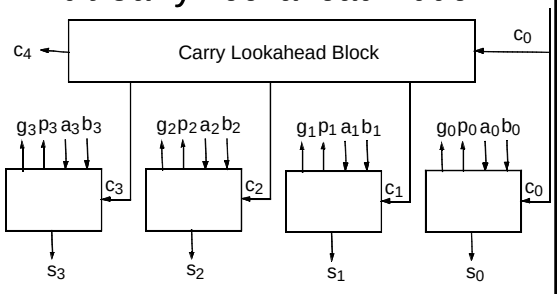
a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Combined Ripple-carry Adder/Subtractor

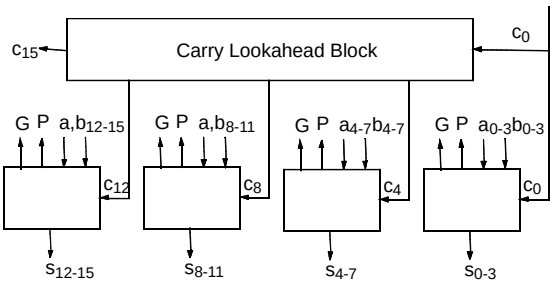
- Control = 1 => subtract
- XOR B with control and set c_{in0} to control



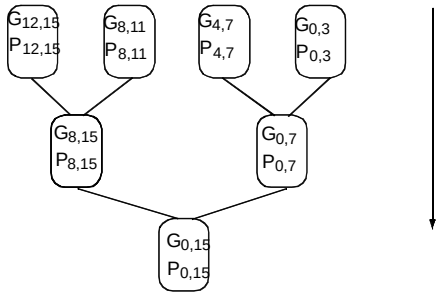
4-bit Carry Lookahead Adder



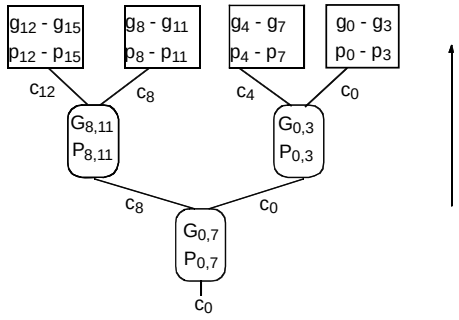
Hierarchical Carry Lookahead for 16 bits



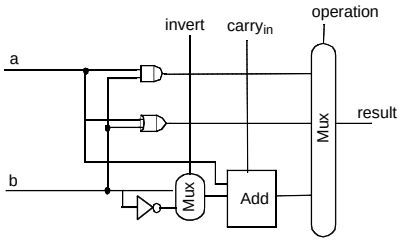
CLA: Compute G's and P's



CLA: Compute Carries



All Together



Addition Overflow

- 2 + 3 = 5 > 4: 010 + 011 = 101 =? –3 < 0
  - X is f(2)
- -1 + -4: 111 + 100 = 011 > 0
  - Y is ~f(2)

Overflow = f(2) * ~(a2)*~(b2) + ~f(2) * a(2) * b(2)

Subtraction Overflow

- No overflow on a-b if signs are the same
- Neg – pos => neg ;; overflow otherwise
- Pos – neg => pos ;; overflow otherwise

Overflow = f(2) * ~(a2)*(b2) + ~f(2) * a(2) * ~b(2)

What to do on Overflow?

- Ignore ! (C language semantics)
  - What about Java? (try/catch?)
- Flag – condition code
- Sticky flag – e.g. for floating point
  - Otherwise gets in the way of fast hardware
- Trap – possibly maskable
  - MIPS has e.g. add that traps, addu that does not

Ch. 3 Summary

- Binary representations, signed/unsigned
- Arithmetic
  - Full adder, ripple-carry, carry lookahead
  - Carry-select, Carry-save
  - Overflow, negative
  - More (multiply/divide/FP) later
- Logical
  - Shift, and, or

Ch. 4 Processor Implementation

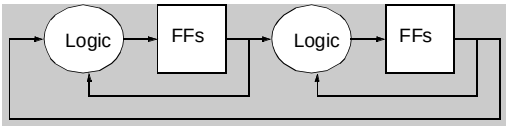
- Heart of 552 – key to project
  - Sequential logic design review (brief)
  - Clock methodology (FSD)
  - Datapath – 1 CPI
    - Single instruction, 2's complement, unsigned
  - Control
  - Multiple cycle implementation (information only)
  - Microprogramming (information only)
  - Exceptions

Clocking Methology

- Motivation
  - Design data and control without considering clock
  - Use Fully Synchronous Design (FSD)
    - Just a convention to simplify design process
    - Restricts design freedom
    - Eliminates complexity, can guarantee timing correctness
    - Not really feasible in real designs
    - Even in 554 you will violate FSD

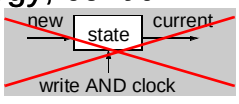
Our Methodology

- Only flip-flops
- All on the same edge (e.g. falling)
- All with same clock
  - No need to draw clock signals
- All logic finishes in one cycle

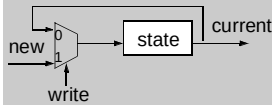


Our Methodology, cont'd

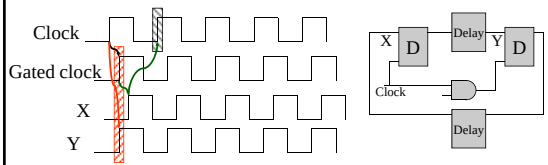
- No clock gating!
  - Book has bad examples



- Correct design:

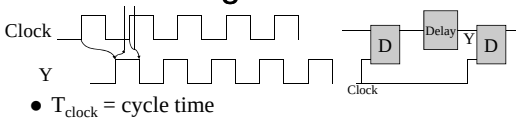


Delayed Clocks (Gating)



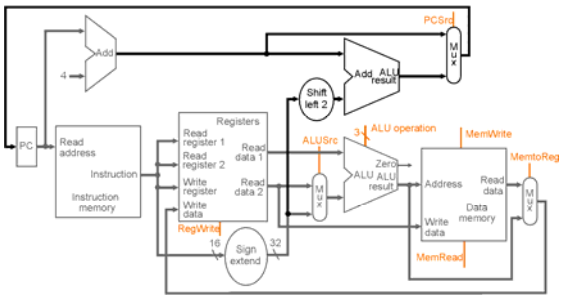
- Problem:
  - Some flip-flops receive gated clock late
  - Data signal may violate setup & hold req't

FSD Clocking Rules

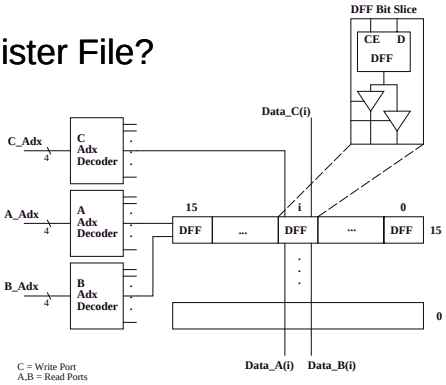


- T_{clock} = cycle time
- T_{setup} = FF setup time requirement
- T_{hold} = FF hold time requirement
- T_{FF} = FF combinational delay
- T_{comb} = Combinational delay
- FSD Rules:
  - $T_{clock} > T_{FF} + T_{comb} + T_{setup}$
  - $T_{FF} + T_{comb} > T_{hold}$

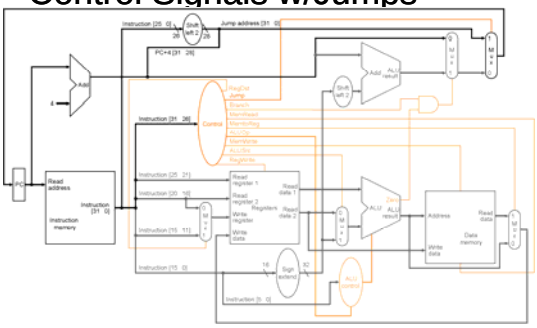
All Together



Register File?



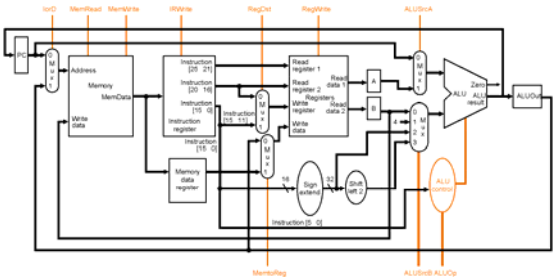
Control Signals w/Jumps



Multi-cycle Implementation

- Clock cycle = max(i-mem,reg-read+reg-write, ALU, d-mem)
- Reuse combination logic on different cycles
  - One memory
  - One ALU without other adders
- But
  - Control is more complex (later)
  - Need new registers to save values (e.g. IR)
    - Used again on later cycles
    - Logic that computes signals is reused

Multi-cycle Ctrl Signals



Multi-cycle Steps

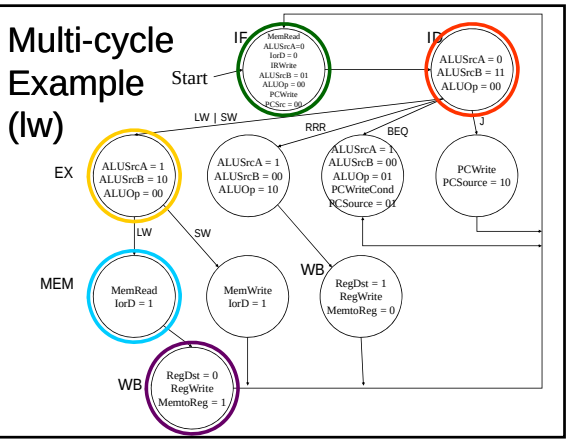
Step	Description	Sample Actions
IF	Fetch	IR=MEM[PC] PC=PC+4
ID	Decode	A=RF(IR[25:21]) B=RF(IR[20:16]) Target=PC+SE(IR[15:0] << 2)

EX	Execute	ALUout = A + SE(IR[15:0]) # lw/sw
		ALUout = A op B # rrr
		if (A==B) PC = target # beq

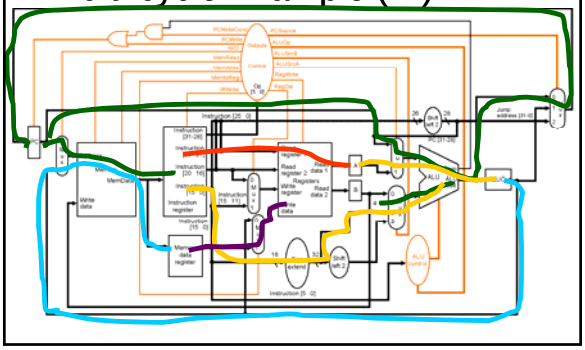
Mem	Memory	MEM[ALUout] = B # sw
		MDR = MEM[ALUout] #lw
		RF(IR[15:11]) = ALUout # rrr

WB	Writeback	Reg(IR[20:16]) = MDR # lw

Multi-cycle Example (lw)



Multi-cycle Example (lw)



Microprogramming

- Alternative way of specifying control
- FSM
  - State – bubble
  - Control signals in bubble
  - Next state given by signals on arc
  - Not a great language for specifying complex events
- Instead, treat as a programming problem

Microprogramming

- Datapath remains the same
- Control is specified differently but does the same
- Each cycle a microprogram field specifies required control signals

Label	Alu	Src1	Src2	Reg	Memory	Pcwrite	Next?
Fetch	Add	Pc	4	Read pc	Alu	Alu	+1
	Add	Pc	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
Lw2					Read alu		+1
				Write mdr			fetch

Exceptions: Big Picture

- Two types:
  - Interrupt (asynchronous) or
  - Trap (synchronous)
- Hardware handles initial reaction
- Then invokes a software exception handler
  - By convention, at e.g. 0xC00
  - O/S kernel provides code at the handler address

Exceptions: Hardware

- Sets state that identifies cause of exception
  - MIPS: in exception_code field of Cause register
- Changes to kernel mode for dangerous work ahead
- Disables interrupts
  - MIPS: recorded in status register
- Saves current PC (MIPS: exception PC)
- Jumps to specific address (MIPS: 0x80000080)
  - Like a surprise JAL – so can't clobber $31

Exceptions: Software

- Exception handler:
  - MIPS: .ktext at 0x80000080
- Set flag to detect incorrect entry
  - Nested exception while in handler
- Save some registers
- Find exception type
  - E.g. I/O interrupt or syscall
- Jump to specific exception handler

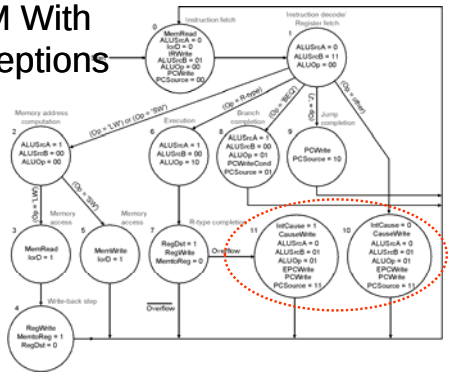
Exceptions: Software, cont'd

- Handle specific exception
- Jump to clean-up to resume user program
- Restore registers
- Reset flag that detects incorrect entry
- Atomically
  - Restore previous mode
  - Enable interrupts
  - Jump back to program (using EPC)

Implementing Exceptions

- We worry only about hardware, not s/w
- IntCause
  - 0 undefined instruction
  - 1 arithmetic overflow
- Changes to the datapath
- New states in control FSM

FSM With Exceptions



Review

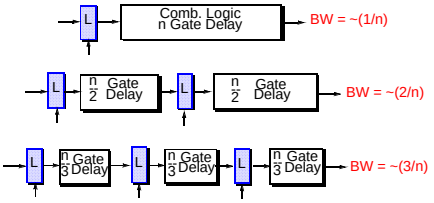
Type	Control	Datapath	Time (CPI, cycle time)
Single-cycle	Comb + end update	No reuse	1 cycle, (imem + reg + ALU + dmem)
Multi-cycle	Comb + FSM update	Reuse	[3,5] cycles, Max(imem, reg, ALU, dmem)
We want?	?	?	~1 cycle, Max(imem, reg, ALU, dmem

- We will use pipelining to achieve last row

Pipelining (4.5-4.9)

- Summary
  - Big Picture
  - Datapath
  - Control
  - Data Hazards
    - Stalls
    - Forwarding
  - Control Hazards
  - Exceptions

Ideal Pipelining



- Bandwidth increases linearly with pipeline depth
- Latency increases by latch delays

Ideal Pipelining

Cycle: Instr:	1	2	3	4	5	6	7	8	9	10	11	12	13
i	F	D	X	M	W								
i+1		F	D	X	M	W							
i+2			F	D	X	M	W						
i+3				F	D	X	M	W					
i+4					F	D	X	M	W				

Pipelining Idealisms

- Uniform subcomputations
  - Can pipeline into stages with equal delay
- Identical computations
  - Can fill pipeline with identical work
- Independent computations
  - No relationships between work units
- Are these practical?
  - No, but can get close enough to get significant speedup

Complications

- Datapath
  - Five (or more) instructions in flight
- Control
  - Must correspond to multiple instructions
- Instructions may have
  - data and control flow *dependences*
  - I.e. units of work are not independent
    - One may have to stall and wait for another

Program Data Dependences

- True dependence (RAW) $D(i) \cap R(j) \neq \phi$
  - j cannot execute until i produces its result
- Anti-dependence (WAR) $R(i) \cap D(j) \neq \phi$
  - j cannot write its result until i has read its sources
- Output dependence (WAW) $D(i) \cap D(j) \neq \phi$
  - j cannot write its result until i has written its result

Control Dependences

- Conditional branches
  - Branch must execute to determine which instruction to fetch next
  - Instructions following a conditional branch are control dependent on the branch instruction

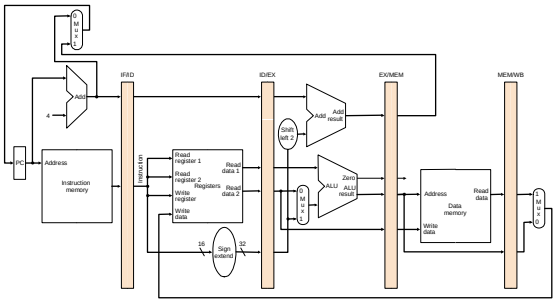
Resolution of Pipeline Hazards

- Pipeline hazards
  - Potential violations of program dependences
  - Must ensure program dependences are not violated
- Hazard resolution
  - Static: compiler/programmer guarantees correctness
  - Dynamic: hardware performs checks at runtime
- Pipeline interlock
  - Hardware mechanism for dynamic hazard resolution
  - Must detect and enforce dependences at runtime

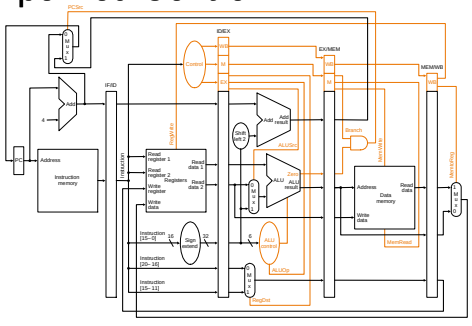
Pipeline Hazards

- Necessary conditions:
  - WAR: write stage earlier than read stage
    - Is this possible in IF-RD-EX-MEM-WB ?
  - WAW: write stage earlier than write stage
    - Is this possible in IF-RD-EX-MEM-WB ?
  - RAW: read stage earlier than write stage
    - Is this possible in IF-RD-EX-MEM-WB?
- If conditions not met, no need to resolve
- Check for both register and memory

Pipelined Datapath



Pipelined Control



Pipelined Control

- Controlled by different instructions
- Decode instructions and pass the signals down the pipe
- Control sequencing is embedded in the pipeline

Data Hazards

- Must first detect hazards

ID/EX.WriteRegister = IF/ID.ReadRegister1

ID/EX.WriteRegister = IF/ID.ReadRegister2

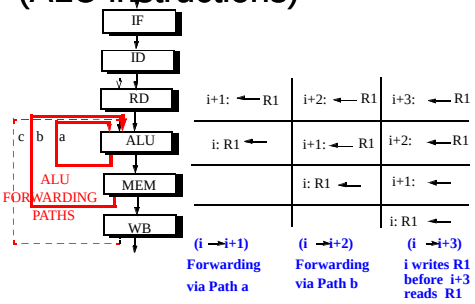
EX/MEM.WriteRegister = IF/ID.ReadRegister1

EX/MEM.WriteRegister = IF/ID.ReadRegister2

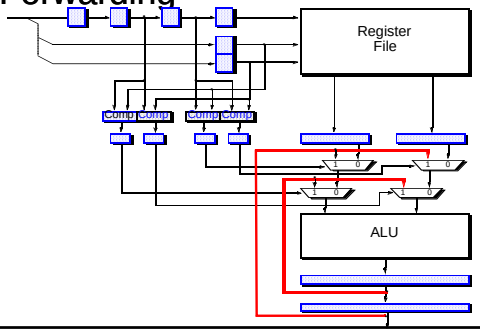
MEM/WB.WriteRegister = IF/ID.ReadRegister1

MEM/WB.WriteRegister = IF/ID.ReadRegister2

Forwarding Paths (ALU instructions)



Implementation of ALU Forwarding



Control Flow Hazards

- What to do?
  - Always stall
  - Easy to implement
  - Performs poorly
  - 1/6th instructions are branches, each branch takes 3 cycles
  - CPI = 1 + 3 x 1/6 = 1.5 (lower bound)

Control Flow Hazards

- Predict branch not taken
- Send sequential instructions down pipeline
- Kill instructions later if incorrect
- Must stop memory accesses and RF writes
  - Including loads (why?)
- Late flush of instructions on misprediction
  - Complex
  - Global signal (wire delay)

Exceptions

- Even worse: in one cycle
  - I/O interrupt
  - User trap to OS (EX)
  - Illegal instruction (ID)
  - Arithmetic overflow
  - Hardware error
  - Etc.
- Interrupt priorities must be supported

Review

- Big Picture
- Datapath
- Control
  - Data hazards
    - Stalls
    - Forwarding or bypassing
  - Control flow hazards
    - Branch prediction
- Exceptions

IBM RISC Experience [Agerwala and Cocke 1987]

- Internal IBM study: Limits of a scalar pipeline?
- Memory Bandwidth
  - Fetch 1 instr/cycle from I-cache
  - 40% of instructions are load/store (D-cache)
- Code characteristics (dynamic)
  - Loads – 25%
  - Stores 15%
  - ALU/RR – 40%
  - Branches – 20%
    - 1/3 unconditional (always taken
    - 1/3 conditional taken, 1/3 conditional not taken

Simplify Branches

- Assume 90% can be PC-relative
  - No register indirect, no register access
  - Separate adder (like MIPS R3000)
  - Branch penalty reduced
- Total CPI: 1 + 0.063 + 0.085 = 1.15 CPI = 0.87 IPC

15% Overhead from program dependences

PC-relative	Schedulable	Penalty
Yes (90%)	Yes (50%)	0 cycle
Yes (90%)	No (50%)	1 cycle
No (10%)	Yes (50%)	1 cycle
No (10%)	No (50%)	2 cycles

Processor Performance

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

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

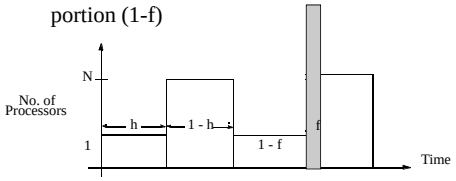
(code size) (CPI) (cycle time)

- In the 1980's (decade of pipelining):
  - CPI: 5.0 => 1.15
- In the 1990's (decade of superscalar):
  - CPI: 1.15 => 0.5 (best case)

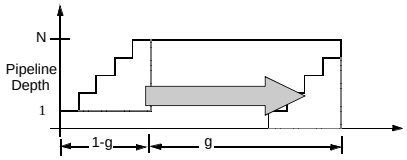
Revisit Amdahl's Law

- Sequential bottleneck
- Even if v is infinite
  - Performance limited by nonvectorizable portion (1-f)

$$\lim_{v \to \infty} \frac{1}{1 - f + \frac{f}{v}} = \frac{1}{1 - f}$$

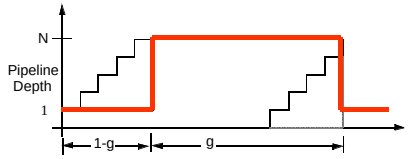


Pipelined Performance Model



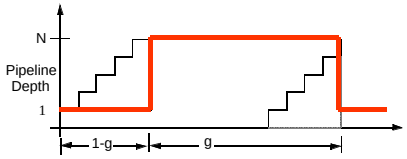
- g = fraction of time pipeline is filled
- 1-g = fraction of time pipeline is not filled (stalled)

Pipelined Performance Model



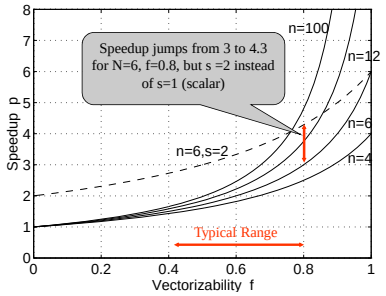
- g = fraction of time pipeline is filled
- 1-g = fraction of time pipeline is not filled (stalled)

Pipelined Performance Model



- Tyranny of Amdahl's Law [Bob Colwell]
  - When g is even slightly below 100%, a big performance hit will result
  - Stalled cycles are the key adversary and must be minimized as much as possible

Motivation for Superscalar [Agerwala and Cocke]



Superscalar Proposal

- Moderate tyranny of Amdahl's Law
  - Ease sequential bottleneck
  - More generally applicable
  - Robust (less sensitive to f)
  - Revised Amdahl's Law:

$$Speedup = \frac{1}{\frac{(1-f)}{s} + \frac{f}{v}}$$

Limits on Instruction Level Parallelism (ILP)

Weiss and Smith [1984]	1.58
Sohi and Vajapeyam [1987]	1.81
Tjaden and Flynn [1970]	1.86 (Flynn's bottleneck)
Tjaden and Flynn [1973]	1.96
Uht [1986]	2.00
Smith et al. [1989]	2.00
Jouppi and Wall [1988]	2.40
Johnson [1991]	2.50
Acosta et al. [1986]	2.79
Wedig [1982]	3.00
Butler et al. [1991]	5.8
Melvin and Patt [1991]	6
Wall [1991]	7 (Jouppi disagreed)
Kuck et al. [1972]	8
Riseman and Foster [1972]	51 (no control dependences)
Nicolau and Fisher [1984]	90 (Fisher's optimism)

Superscalar Proposal

- Go beyond single instruction pipeline, achieve IPC > 1
- Dispatch multiple instructions per cycle
- Provide more generally applicable form of concurrency (not just vectors)
- Geared for sequential code that is hard to parallelize otherwise
- Exploit fine-grained or instruction-level parallelism (ILP)

Classifying ILP Machines [Jouppi, DECWRL 1991]

- Scalar pipelined
- Superpipelined
- Superscalar
- VLIW
- Superpipelined superscalar

Review Summary

- Ch. 1: Intro & performance
- Ch. 2: Instruction Sets
- Ch. 3: Arithmetic I
- Ch. 4: Data path, control, pipelining
- Details
  - Fri. 10/29 2:25-3:30 (1 hour) in EH2317
  - Closed books/notes/homeworks
  - One page handwritten cheatsheet for quick reference
  - A mix of short answer, design, analysis problems