

Readings in Pipelining

H+P

- Appendix A (except for A.8)
- This will be mostly review for those who took ECE 152

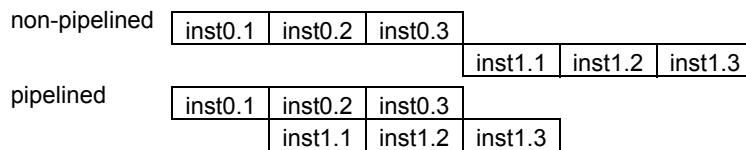
Basic Pipelining

- basic := single, in-order issue
 - **single issue** := one instruction at a time (per stage)
 - **in-order issue** := instructions (start to) execute in order
 - next unit: multiple issue
 - unit after that: out-of-order issue
- pipelining principles
 - tradeoff: clock rate vs. IPC
 - hazards: structural, data, control
- vanilla pipeline: single-cycle operations
 - structural hazards, RAW hazards, control hazards
- dealing with multi-cycle operations
 - more structural hazards, WAW hazards, precise state

Pipelining

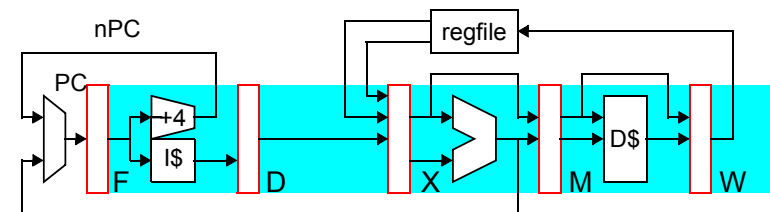
observe: instruction processing consists of N sequential stages

idea: overlap different instructions at different stages



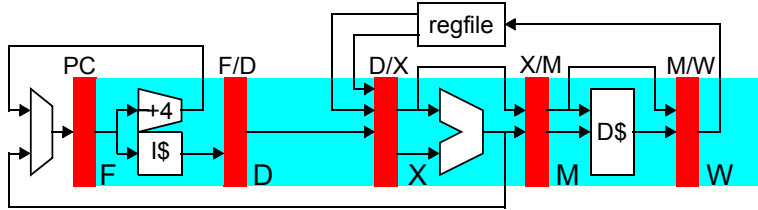
- + increase resource utilization: fewer stages sitting idle
- + increase completion rate (throughput): up to 1 in $1/N$ time
- almost every processor built since 1970 is pipelined
 - first pipelined processor: IBM Stretch [1962]

Without Pipelining



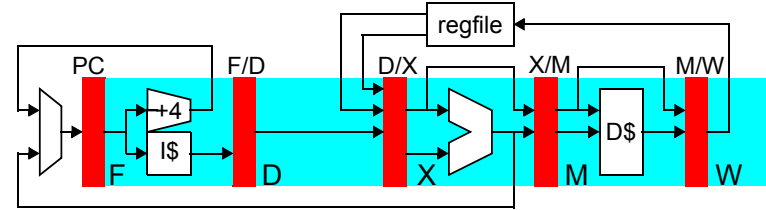
- 5 parts of instruction execution
 - fetch (F, IF): fetch instruction from I\$
 - decode (D, ID): decode instruction, read input registers
 - execute (X, EX): ALU, load/store address, branch outcome
 - memory access (M, MEM): load/store to D\$/DTLB
 - writeback (W, WB): write results (from ALU or Id) back to register file

Simple 5-Stage Pipeline



- 5 stages (pipeline depth is 5)
 - fetch (F, IF): fetch instruction from I\$
 - decode (D, ID): decode instruction, read input registers
 - execute (X, EX): ALU, load/store address, branch outcome
 - memory access (M, MEM): load/store to D\$/DTLB
 - writeback (W, WB): write results (from ALU or Id) back to register file
- stages divided by *pipeline registers/latches*

Pipeline Registers (Latches)



- contain info for controlling flow of instructions through pipe
 - PC: PC
 - F/D: PC, undecoded instruction
 - D/X: PC, opcode, regfile[rs1], regfile[rs2], immed, rd
 - X/M: opcode (why?), regfile[rs1], ALUOUT, rd
 - M/W: ALUOUT, MEMOUT, rd

Pipeline Diagram

	1	2	3	4	5	6	7	8	← cycles
inst0	F	D	X	M	W				
inst1		F	D	X	M	W			
inst2			F	D	X	M	W		
inst3				F	D	X	M	W	

Compared to non-pipelined case:

- Better throughput: an instruction finishes every cycle
- Same latency per instruction: each still takes 5 cycles

Principles of Pipelining

let: instruction execution require N stages, each takes t_n time

- un-pipelined processor
 - single-instruction latency $T = \sum t_n$
 - throughput = $1/T = 1/\sum t_n$
 - M -instruction latency = $M * T$ ($M \gg 1$)
- now: *N -stage pipeline*
 - single-instruction latency $T = \sum t_n$ (same as unpipelined)
 - throughput = $1/\max(t_n) \leq N/T$ ($\max(t_n)$ is the bottleneck)
 - if all t_n are equal (i.e., $\max(t_n) = T/N$), then throughput = N/T
 - M -instruction latency ($M \gg 1$) = $M * \max(t_n) \leq M * T/N$
 - speedup $\leq N$
- can we choose N to get arbitrary speedup?

Wrong (part I): Pipeline Overhead

V := overhead delay per pipe stage

- cause #1: latch overhead
 - pipeline registers take time
- cause #2: clock/data skew

so, for an N-stage pipeline with overheads

- single-instruction latency $T = \Sigma(V + t_n) = N*V + \Sigma t_n$
- throughput = $1/(\max(t_n) + V) \leq N/T$ (and $\leq 1/V$)
- M-instruction latency = $M*(\max(t_n) + V) \leq M*V + M*T/N$
- speedup = $T/(\max(t_n)) \leq N$

Overhead limits throughput, speedup & useful pipeline depth

Pipelining: Clock Rate vs. IPC

deeper pipeline (more stages, larger N)

- + increases clock rate
- decreases IPC (longer stalls for hazards - will see later)
- ultimate metric is **execution rate** = clock rate*IPC
 - (clock cycle / unit real time) * (instructions / clock cycle)
 - number of instructions is fixed, for purposes of this discussion
- how does pipeline overhead factor in?

to think about this, **parameterize the clock cycle**

- basic time unit is the **gate-delay** (time to go through a gate)
 - e.g., 80 gate-delays to process (fetch, decode,...) an instruction
 - let's look at an example ...

Wrong (part II): Hazards

hazards: conditions that lead to incorrect behavior if not fixed

- **structural**: two instructions use same h/w in same cycle
- **data**: two instructions use same data (register/memory)
- **control**: one instruction affects which instruction is next

• hazards \Rightarrow **stalls** (sometimes)

- stall: instruction stays in same stage for more than one cycle

• what if average stall per instruction = S stages?

- latency' $\Rightarrow T(N+S)/N = ((N+S)/N)*latency > latency$
- throughput' $\Rightarrow N^2/T(N+S) = (N/(N+S))*throughput < throughput$
- $M_latency' \Rightarrow M*T(N+S)/N^2 = ((N+S)/N)*M_latency > M_latency$
- speedup' $\Rightarrow N^2/(N+S) = (N/(N+S))*speedup < speedup$

Clock Rate vs. IPC Example

- G: gate-delays to process an instruction
- V: gate-delays of overhead per stage
- S: average cycle stall per instruction per pipe stage
 - overly simplistic model for stalls
- compute optimal N (depth) given G, V, S [Smith+Pleszkun]
 - $IPC = 1 / (1 + S*N)$
 - clock rate (in gate-delays) = $1/(\text{gate delays}/\text{stage}) = 1/(G/N + V)$
 - e.g., G = 80, S = 0.16, V = 1

N	IPC := $1/(1+0.16*N)$	clock := $1/(80/N+1)$	execution rate
10	0.38	0.11	0.042
20	0.24	0.20	0.048
30	0.17	0.27	0.046

Pipeline Depth Upshot

trend is for *deeper pipelines* (more stages)

- why? faster clock (higher frequency)
 - clock period = $f(\text{transistor latency, gate delays per pipe stage})$
 - **superpipelining**: add more stages to reduce gate-delays/pipe-stage
 - but increased frequency may not mean increased performance...
 - who cares? we can sell frequency!
- e.g., Intel IA-32 pipelines
 - 486: 5 stages (50+ gate-delays per clock period)
 - Pentium: 7 stages
 - Pentium II/III: 12 stages
 - Pentium 4: 22 stages (10 gate-delays per clock)
 - Gotcha! 800MHz Pentium III performs better than 1GHz Pentium 4

Managing the Pipeline

to resolve hazards, need fine pipe-stage control

- play with pipeline registers to control pipe flow
- trick #1: **the stall (or the bubble)**
 - effect: stops *SOME* instructions in current pipe-stages
 - use: make younger instructions wait for older ones to complete
 - implementation: de-assert write-enable signals to pipeline registers
- trick #2: **the flush**
 - effect: clears instructions out of current pipe-stages
 - use: undoes speculative work that was incorrect (see later)
 - implementation: assert clear signals on pipeline registers
- stalls & flushes must be propagated upstream (why?)
 - upstream: towards fetch (downstream = towards writeback)

Structural Hazards

two different instructions need same h/w resource in same cycle

- e.g., loads/stores use the same cache port as fetch
 - assume unified L1 cache (for this example)

	1	2	3	4	5	6	7	8	9	10	11	12	13
load	F	D	X	M	W								
inst2		F	D	X	M	W							
inst3			F	X	X	M	W						
inst3				F	D	X	M	W					

Fixing Structural Hazards

- fix structural hazard by stalling (**s*** = structural stall)
 - + low cost, simple
 - decreases IPC
 - used rarely
- Q: which one to stall, inst4 or load?
 - always safe to stall younger instruction (why?)...
 - ...but may not be the best thing to do performance-wise (why?)

	1	2	3	4	5	6	7	8	9	10	11	12	13
load	F	D	X	M	W								
inst2		F	D	X	M	W							
inst3			F	D	X	M	W						
inst4				s*	F	D	X	M	W				

Avoiding Structural Hazards

- option #1: replicate the contended resource
 - + good performance
 - increased area, slower (interconnect delay)?
 - use for cheap, divisible, or highly-contended resources (e.g., I\$/D\$)
- option #2: pipeline the contended resource
 - + good performance, low area
 - sometimes complex (e.g., RAM)
 - useful for multicycle resources
- option #3: design ISA/pipeline to reduce structural hazards
 - key 1: each instruction uses a given resource **at most once**
 - key 2: each instruction uses a given resource **in same pipeline stage**
 - key 3: each instruction uses a given resource **for one cycle**
 - this is why we force ALU operations to go thru MEM stage

Data Hazards

two different instructions use the same storage location

- we must preserve **the illusion of sequential execution**

```
add R1, R2, R3
sub R2, R4, R1
or R1, R6, R3
```

read-after-write (RAW)

true dependence
(real)

```
add R1, R2, R3
sub R2, R4, R1
or R1, R6, R3
```

write-after-read (WAR)

anti-dependence
(artificial)

```
add R1, R2, R3
sub R2, R4, R1
or R1, R6, R3
```

write-after-write (WAW)

output dependence
(artificial)

Q: What about read-after-read dependences? (RAR)

RAW

read-after-write (RAW) = true dependence (dataflow)

```
add R1, R2, R3
sub R2, R4, R1
or R1, R6, R3
```

- **problem:** `sub` reads `R1` before `add` has written it
 - Pipelining enables this overlapping to occur
 - But this violates sequential execution semantics!
 - Recall: user just sees ISA and expects sequential execution

RAW: Detect and Stall

detect RAW and stall instruction at ID before it reads registers

- mechanics? disable PC, F/D write
- RAW detection? *compare register names*
 - notation: $rs1(D)$:= source register #1 of instruction in D stage
 - compare $rs1(D)$ and $rs2(D)$ with $rd(D/X)$, $rd(X/M)$, $rd(M/W)$
 - stall (disable PC + F/D, clear D/X) on any match
- RAW detection? *register busy-bits*
 - set for $rd(D/X)$ when instruction passes ID
 - clear for $rd(M/W)$
 - stall if $rs1(D)$ or $rs2(D)$ are "busy"

+ low cost, simple

- low performance (many stalls)

Two Stall Timings

depend on how ID and WB stages share the register file

- each gets register file for half a cycle
- 1st half ID reads, 2nd half WB writes \Rightarrow 3 cycle stall

	1	2	3	4	5	6	7	8	9
add R1, R2, R3	F	D	X	M	W				
sub R2, R4, R1		F	d*	d*	d*	D	X	M	W
load R5, R6, R7			p*	p*	p*	F	D	X	M

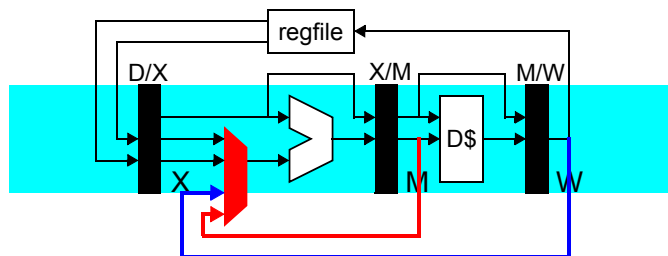
- 1st half WB writes, 2nd half ID reads \Rightarrow 2 cycle stall

	1	2	3	4	5	6	7	8	9
add R1, R2, R3	F	D	X	M	W				
sub R2, R4, R1		F	d*	d*	D	X	M	W	

Stall Signal Example (2nd Timing)

PC	F/D	D/X	X/M	M/W
load r6,0(r4)	add r4,r2,r1	load r2,0(r3)	add r5,r5,#4	call func
F	D	X	M	W
↑write disable	↑write disable	↑clear	c1: rs1(D) == rd(D/X) \Rightarrow stall	
PC	F/D	D/X	X/M	M/W
load r6,0(r4)	add r4,r2,r1	load r2,0(r3)	add r5,r5,#4	
F	D	X	M	W
↑write disable	↑write disable	↑clear	c2: rs1(D) == rd(X/M) \Rightarrow stall	
PC	F/D	D/X	X/M	M/W
load r6,0(r4)	add r4,r2,r1			load r2,0(r3)
F	D	X	M	W
				c3: rs1(D) == rd(X/M) \Rightarrow go
PC	F/D	D/X	X/M	M/W
sub r6,r6,#1	load r6,0(r4)	add r4,r2,r1		
F	D	X	M	W

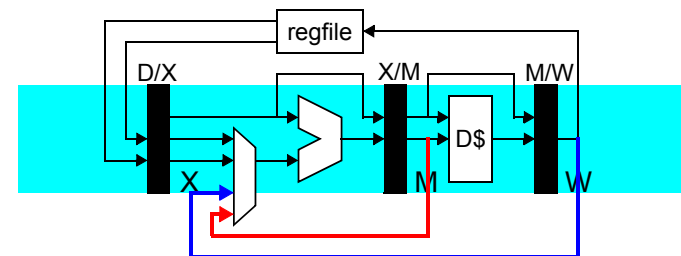
Reducing RAW Stalls: Bypassing



why wait until WB stage? data available at end of EX/MEM stage

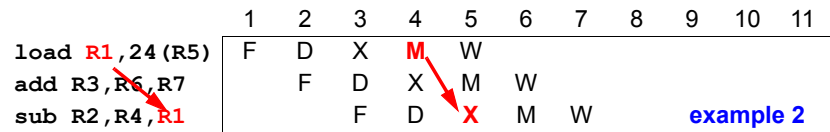
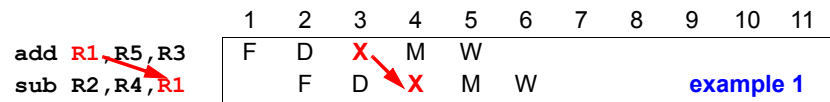
- bypass (aka "forward") data directly to input of EX
- + very effective at reducing/avoiding stalls
 - in practice, a large fraction of input operands are bypassed (why?)
- complex
- does not relieve you from having to perform WB

Implementing Bypassing

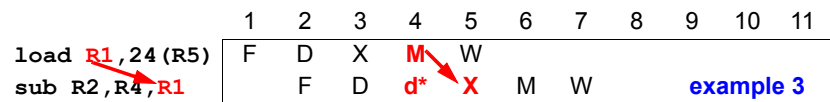


- first, detect bypass opportunity
 - tag compares in D/X latch
 - similar to but separate from stall logic in F/D latch
- then, control bypass MUX
 - if $rs2(X) == rd(X/M)$ then $ALUOUT(M)$
 - else if $rs2(X) == rd(M/W)$ then $ALUOUT(W)$

Pipeline Diagrams with Bypassing



- even with full bypassing, not all RAW stalls can be avoided
- example: load to ALU in consecutive cycles



Pipeline Scheduling

compiler schedules (moves) instructions to reduce stall

- eliminate back-to-back load-ALU scenarios
- example code sequence `a = b + c; d = e - f`

before

```
load R2, b
load R3, c
add R1, R2, R3 //stall
store R1, a
load R5, e
load R6, f
sub R4, R5, R6 //stall
store R4, d
```

after

```
load R2, b
load R3, c
load R5, e
add R1, R2, R3 //no stall
load R6, f
store R1, a
sub R4, R5, R6 //no stall
store R4, d
```

WAR: Write After Read

write-after-read (WAR) = artificial (name) dependence

```
add R1, R2, R3
sub R2, R4, R1
or R1, R6, R3
```

- **problem:** `add` could use wrong value for `R2`
- can't happen in vanilla pipeline (reads in ID, writes in WB)
 - can happen if: early writes (e.g., auto-increment) + late reads (??)
 - can happen if: out-of-order reads (e.g., out-of-order execution)
- **artificial:** using different output register for `sub` would solve
 - The dependence is on the **name** `R2`, but not on actual dataflow

WAW: Write After Write

write-after-write (WAW) = artificial (name) dependence

```
add R1, R2, R3
sub R2, R4, R1
or R1, R6, R3
```

- **problem:** reordering could leave wrong value in `R1`
 - later instruction that reads `R1` would get wrong value
- can't happen in vanilla pipeline (register writes are in order)
 - another reason for making ALU ops go through MEM stage
 - can happen: multi-cycle operations (e.g., FP, cache misses)
- **artificial:** using different output register for `or` would solve
 - Also a dependence on a name: `R1`

RAR: Read After Read

read-after-read (RAR)

```
add R1, R2, R3
sub R2, R4, R3
or R1, R6, R3
```

- **no problem**: R3 is correct even with reordering

Memory Data Hazards

have seen register hazards, can also have *memory hazards*

	RAW	WAR	WAW
	store R1, 0 (SP)	load R4, 0 (SP)	store R1, 0 (SP)
	load R4, 0 (SP)	store R1, 0 (SP)	store R4, 0 (SP)

	1	2	3	4	5	6	7	8	9
store R1, 0 (SP)	F	D	X	M	W				
load R1, 0 (SP)		F	D	X	M	W			

- in simple pipeline, memory hazards are easy

- in-order
- one at a time
- read & write in same stage

- in general, though, more difficult than register hazards

Hazards vs. Dependences

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 instrs vs. pipeline depth

For example, can have RAW dependence with or without hazard

- depends on pipeline

Control Hazards

when an instruction affects *which* instruction executes next

```
store R4, 0 (R5)
bne R2, R3, loop
sub R1, R6, R3
```

- naive solution: stall until outcome is available (end of EX)
 - + simple
 - low performance (2 cycles here, longer in general)
 - e.g. 15% branches * 2 cycle stall ⇒ 30% CPI increase!

	1	2	3	4	5	6	7	8	9
store R4, 0 (R5)	F	D	X	M	W				
bne R2, R3, loop		F	D	X	M	W			
??			c*	c*	F	D	X	M	W

Control Hazards: “Fast” Branches

fast branches: can be evaluated in ID (rather than EX)

+ reduce stall from 2 cycles to 1

	1	2	3	4	5	6	7	8	9
sw R4,0(R5)	F	D	X	M	W				
bne R2,R3,loop		F	D	X	M	W			
??			c*	F	D	X	M	W	

– requires more hardware

- dedicated ID adder for (PC + immediate) targets

– requires simple branch instructions

- no time to compare two registers (would need full ALU)
- comparisons with 0 are fast (beqz, bnez)

Control Hazards: Delayed Branches

delayed branch: execute next instruction whether taken or not

- instruction after branch said to be in “*delay slot*”
- old microcode trick stolen by RISC (MIPS)

store R4,0(R5)	bne R2,R3,loop	sub R1,R6,R6
bne R2,R3,loop	store R4,0(R5)	sub R1,R6,R6
sub R1,R6,R6		

	1	2	3	4	5	6	7	8	9
bne R2,R3,loop	F	D	X	M	W				
store R4,0(R5)		F	D	X	M	W			
sub R1,R6,R6			c*	F	D	X	M	W	

What To Put In Delay Slot?

- instruction from before branch
 - when? if branch and instruction are independent
 - helps? always
- instruction from target (taken) path
 - when? if safe to execute, but may have to duplicate code
 - helps? on taken branch, but may increase code size
- instruction from fall-through (not-taken) path
 - when? if safe to execute
 - helps? on not-taken branch
- upshot: short-sighted ISA feature
 - not a big win for today’s machines (why? consider pipeline depth)
 - complicates interrupt handling (later)

Control Hazards: Speculative Execution

idea: doing anything is better than waiting around doing nothing

- **speculative execution**
 - guess branch target \Rightarrow start executing at guessed position
 - execute branch \Rightarrow verify (check) guess
 - + minimize penalty if guess is right (to zero?)
 - wrong guess could be worse than not guessing
- **branch prediction:** guessing the branch
 - one of the “important” problems in computer architecture
 - very heavily researched area in last 15 years
 - static: prediction by compiler
 - dynamic: prediction by hardware
 - hybrid: compiler hints to hardware predictor

The Speculation Game

speculation: engagement in risky business transactions on the chance of quick or considerable profit

- **speculative execution (control speculation)**
 - execute before all parameters known with certainty
- + **correct speculation**
 - + avoid stall/get result early, performance improves
- **incorrect speculation (mis-speculation)**
 - must abort/squash incorrect instructions
 - must undo incorrect changes (recover pre-speculation state)
- **the speculation game:** profit > penalty
 - profit = speculation accuracy * correct-speculation gain
 - penalty = (1-speculation accuracy) * mis-speculation penalty

Speculative Execution Scenarios

	1	2	3	4	5
inst0/B	F	D	X	M	W
inst8		F	D	X	M
inst9			F	D	X
inst10				F	D

- **correct speculation**
 - cycle1: fetch branch, predict next (inst8)
 - c2, c3: fetch inst8, inst9
 - c3: execute/verify branch \Rightarrow correct
 - nothing needs to be fixed or changed

	1	2	3	4	5
inst0/B	F	D	X	M	W
inst1		F	D		
inst2			F		
inst8	verify/flush			F	D

- **incorrect speculation: mis-speculation**
 - c1: fetch branch, predict next (inst1)
 - c2, c3: fetch inst1, inst2
 - c3: execute/verify branch \Rightarrow wrong
 - c3: send correct target to IF (inst8)
 - c3: squash (abort) inst1, inst2 (flush F/D)
 - c4: fetch inst8

Static (Compiler) Branch Prediction

Some static prediction options

- predict always not-taken
 - + very simple, since we already know the target (PC+4)
 - most branches (~65%) are taken (why?)
- predict always taken
 - + better performance
 - more difficult, must know target before branch is decoded
- predict backward taken
 - most backward branches are taken
- predict specific opcodes taken
- use profiles to predict on per-static branch basis
 - pretty good

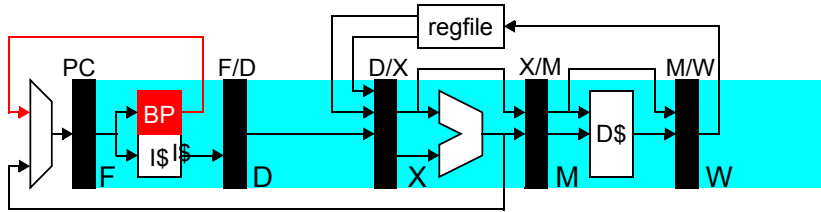
Comparison of Some Static Schemes

$$\text{CPI-penalty} = \%_{\text{branch}} * [(\%_{\text{T}} * \text{penalty}_{\text{T}}) + (\%_{\text{NT}} * \text{penalty}_{\text{NT}})]$$

- simple branch statistics
 - 14% PC-changing instructions (“branches”)
 - 65% of PC-changing instructions are “taken”

scheme	penalty _T	penalty _{NT}	CPI penalty
stall	2	2	0.28
fast branch	1	1	0.14
delayed branch	1.5	1.5	0.21
not-taken	2	0	0.18
taken	0	2	0.10

Dynamic Branch Prediction



hardware (BP) guesses whether and where a branch will go

```
0x64    bnez r1, #10
0x74    add r3, r2, r1
```

- start with branch PC (0x64) and produce
 - direction (Taken)
 - direction + target PC (0x74)
 - direction + target PC + target instruction (add r3, r2, r1)

Improvement: Two-bit Counters

example: 4-iteration inner loop branch

state/prediction	N	T	T	T	N	T	T	T	N	T	T	T
branch outcome	T	T	T	N	T	T	T	N	T	T	T	N
mis-prediction?	*			*	*			*	*			*

– problem: two mis-predictions per loop

- solution: 2-bit saturating counter to implement **hysteresis**
 - 4 states: strong/weak not-taken (N/n), strong/weak taken (T/t)
 - transitions: $N \leftrightarrow n \leftrightarrow t \leftrightarrow T$

state/prediction	n	t	T	T	t	T	T	T	t	T	T	T
branch outcome	T	T	T	N	T	T	T	N	T	T	T	N
mis-prediction?	*			*				*				*

+ only one mis-prediction per iteration

Branch History Table (BHT)

branch PC \Rightarrow prediction (T, NT)

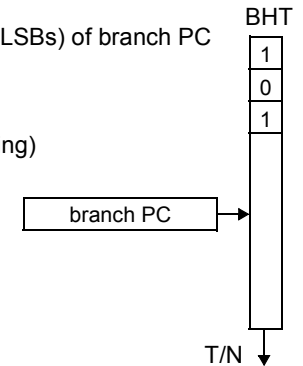
– need decoder/adder to compute target if taken

- **branch history table (BHT)**

- read prediction with least significant bits (LSBs) of branch PC
- change bit on misprediction
- + simple
- multiple PCs may map to same bit (aliasing)

- major improvements

- two-bit counters [Smith]
- correlating/two-level predictors [Patt]
- hybrid predictors [McFarling]



Improvement: Correlating Predictors

different branches may be **correlated**

- outcome of branch depends on outcome of other branches
 - makes intuitive sense (programs are written this way)
- e.g., if the first two conditions are true, then third is false

```
if (aa == 2) aa = 0;
if (bb == 2) bb = 0;
if (aa != bb) { . . . }
```

revelation: prediction = f(branch PC, recent branch outcomes)

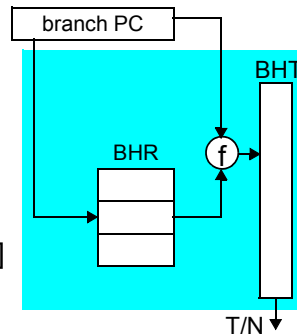
- revolution: BP accuracies increased dramatically
- lots of research in designing that function for best BP

Correlating (Two-Level) Predictors

- **branch history shift register (BHR)** holds recent outcomes
 - combination of PC and BHR accesses BHT
 - basically, multiple predictions per branch, choose based on history

design space

- number of BHRs
 - multiple BHRs (“local”, Intel)
 - 1 global BHR (“global”, everyone else)
- PC/BHR overlap
 - full, partial, none (concatenated?)
- popular design: Gshare [McFarling]
 - 1 global BHR, full overlap, $f = \text{XOR}$



Correlating Predictor Example

- example with alternating T,N (1-bit BHT, no correlation)

state/prediction	N	T	N	T	N	T	N	T	N	T	N	T
branch outcome	T	N	T	N	T	N	T	N	T	N	T	N
mis-prediction?	*	*	*	*	*	*	*	*	*	*	*	*

- add 1 1-bit BHR, concatenate with PC
 - effectively, two predictors per PC
 - top (BHR=N) bottom (BHR=T) **active entry**

state/prediction	N	T	T	T	T	T	T	T	T	T	T	T
branch outcome	N	N	N	N	N	N	N	N	N	N	N	N
mis-prediction?	*											

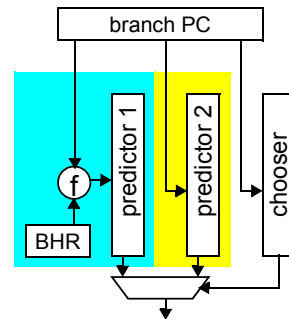
Hybrid/Competitive/Tournament Predictors

observation: different schemes work better for different branches

idea: multiple predictors, choose on per static-branch basis

mechanics

- two (or more) predictors
- chooser
 - if chosen predictor is wrong...
 - ...and other is right...
 - ...flip chooser
- popular design: Gselect [McFarling]
 - Gshare + 2-bit saturating counter



Branch Target Buffer (BTB)

branch PC \Rightarrow target PC

- target PC available at end of IF stage
 - + no bubble for correct predictions
- branch target buffer (BTB)
 - index: branch PC
 - data: target PC (+ T/NT?)
 - tags: branch PC (why are tags needed here and not in BHT?)
 - many more bits per entry than BHT
 - considerations: combine with l-cache? store not-taken branches?
- branch target cache (BTC)
 - data: target PC + target instruction(s)
 - enables “branch folding” optimization (branch removed from pipe)

Jump Prediction

exploit behavior of different kinds of jumps to improve prediction

- function returns
 - use *hardware return address stack (RAS)*
 - call pushes return address on top of RAS
 - for return, predict address at top of RAS and pop
 - trouble: must manage speculatively
- indirect jumps (switches, virtual functions)
 - more than one taken target per jump
 - path-based BTB [Driesen+Holzle]

Branch Issues

issue1: how do we know at IF which instructions are branches?

- BTB: don't need to "know"
 - check every instruction: BTB entry \Rightarrow instruction is a branch

issue2: BHR (RAS) depend on branch (call) history

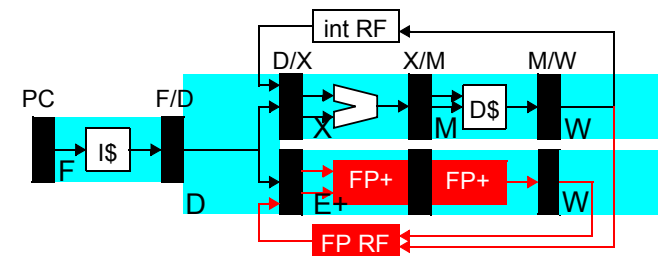
- when are these updated?
 - at WB is too late (if another branch is in-flight)
 - at IF (after prediction)
 - must be able to recover BHR (RAS) on mis-speculation (nasty)

Adding Multi-Cycle Operations

RISC tenet #1: "single-cycle operations"

- why was this such a big deal?
 - **fact:** not all operations complete in 1 cycle
 - FP add, int/FP multiply: 2–4 cycles, int/FP divide: 20–50 cycles
 - data cache misses: 10–150 cycles!
- slow clock cycle down to slowest operation?
 - can't without incurring huge performance loss
- solution: extend pipeline - add pipeline stages to EX

Extended Pipeline



- separate integer/FP, pipe register files
 - loads/stores in integer pipeline only (why?)
- additional, parallel functional units
 - E+: FP adder (2 cycles, pipelined)
 - E*: FP/integer multiplier (4 cycles, pipelined)
 - E/: FP/integer divider (20 cycles, not pipelined)

Multi-Cycle Example

	1	2	3	4	5	6	7	8	9	10
divf f0, f1, f2	F	D	E/	E/	E/	E/	W			
mulf f0, f3, f4		F	D	E*	E*	W				
addf f5, f6, f7			F	D	E+	E+	W			
subf f8, f6, f7				F	D	*	E+	E+	W	
mulf f9, f8, f7					F	D	*	*	E*	E*

- write-after-write (WAW) hazards
- register write port structural hazards
- functional unit structural hazards
- elongated read-after-write (RAW) hazards

Another Multi-Cycle Example

example: SAXPY (math kernel)

$z[i] = A * X[i] + Y[i]$ // single precision

	1	2	3	4	5	6	7	8	9	10
ldf f2, 0(r1)	F	D	X	M	W					
mulf f6, f0, f2		F	D	d*	E*	E*	E*	E*	W	
ldf f4, 0(r2)			F	p*	D	X	M	W		
addf f8, f6, f4				F	D	d*	d*	E+	E+	W
stf f8, 0(r3)					F	p*	p*	D	X	M
add r1, r1, #4								F	D	X
add r2, r2, #4									F	D
add r3, r3, #4										F

Register Write Port Structural Hazards

where are these resolved?

- multiple writeback ports?
 - not a good idea (why not?)
- in ID?
 - reserve writeback slot in ID (writeback reservation bits)
 - + simple, keeps stall logic localized to ID stage
 - won't work for cache misses (why not?)
- in MEM?
 - + works for cache misses, better utilization
 - two stall controls (F/D and M/W) must be synchronized
- in general: cache misses are hard
 - don't know in ID whether they will happen early enough (in ID)

WAW Hazards

how are these dealt with?

- stall younger instruction writeback?
 - + intuitive, simpler
 - lower performance (cascading writeback structural hazards)
- abort (don't do) older instruction writeback?
 - + no performance loss
 - but what if intermediate instruction causes an interrupt (next)

Dealing With Interrupts

interrupts (aka faults, exceptions, traps)

- e.g., arithmetic overflow, divide by zero, protection violation
- e.g., I/O device request, OS call, page fault

classifying interrupts

- terminal (fatal) vs. *restartable* (control returned to program)
- synchronous (internal) vs. asynchronous (external)
- user vs. coerced
- maskable (ignorable) vs. non-maskable
- between instructions vs. within instruction

Precise Interrupts

“unobserved system can exist in any intermediate state, upon observation system collapses to well-defined state”

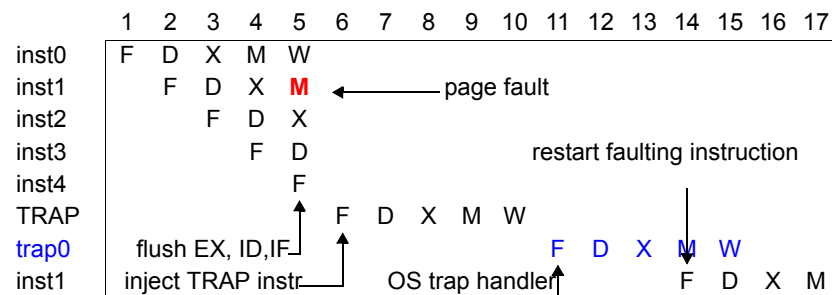
–2nd postulate of quantum mechanics

- system \Rightarrow processor, observation \Rightarrow interrupt

what is the “well-defined” state?

- von Neumann: “sequential, instruction atomic execution”
- precise state at interrupt
 - all instructions older than interrupt are complete
 - all instructions younger than interrupt haven’t started
- implies interrupts are taken in program order
- necessary for VM (why?), “highly recommended” by IEEE

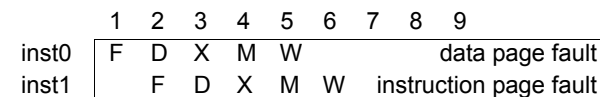
Interrupt Example: Data Page Fault



- squash (effects of) younger instructions
- inject fake TRAP instruction into IF
- from here, like a SYSCALL

More Interrupts

- interrupts can occur at different stages
 - IF, MEM: page fault, misaligned data, protection violation
 - ID: illegal/privileged instruction
 - EX: arithmetic exception



- too complicated to draw what goes on here
 - cycle2: instruction page fault, flush inst1, inject TRAP
 - c4: data page fault, flush inst0, inst1, TRAP
 - can get into an infinite loop here (with help of OS page placement)

Posted Interrupts

posted interrupts

- set interrupt bit when condition is raised
- check interrupt bit (potentially “take” interrupt) in WB
 - + interrupts are taken in order
 - longer latency, more complex

	1	2	3	4	5	6	7	8	9
inst0	F	D	X	M	W				
inst1		F	D	X	M	W			

data page fault

instruction page fault

- what happens now?
 - c2: set inst1 bit
 - c4: set inst0 bit
 - c5: take inst0 interrupt

Interrupts and Multi-Cycle Operations

	1	2	3	4	5	6	7	8	9	10	11
divf f0, f1, f2	F	D	E/	E/	E/	E/	W				
mulf f3, f4, f5			F	D	E*	E*	W				
addf f6, f7, f8				F	D	E+	E+	s*	W		

div by 0 (posted)

multi-cycle operations + precise state = trouble

- #1: how to undo early writes?
 - e.g., must make it seem as if `mulf` hasn't executed
 - undo writes: future file, history file -> ugly!
- #2: how to take interrupts in-order if WB is not in-order?
 - force in-order WB
 - slow

Interrupts Are Nasty

- odd bits of state must be precise (e.g., CC)
- delayed branches
 - what if instruction in delay slot takes an interrupt?
- modes with early-writes (e.g., auto-increment)
 - must undo write (e.g., future-file, history-file)
- some machines had precise interrupts only in integer pipe
 - sufficient for implementing VM
 - e.g., VAX/Alpha

Lucky for us, there's a nice, clean way to handle precise state

- We'll see how this is done in a couple of lectures ...

Summary

- principles of pipelining
 - pipeline depth: clock rate vs. number of stalls (CPI)
- hazards
 - structural
 - data (RAW, WAR, WAW)
 - control
- multi-cycle operations
 - structural hazards, WAW hazards
- interrupts
 - precise state

next up: dynamic ILP (chapter 3)