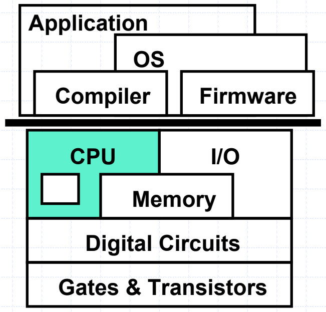
CIS 501
Introduction to Computer Architecture

Unit 6: Pipelining

This Unit: Pipelining

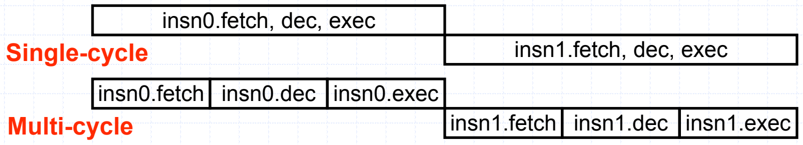


- Basic Pipelining
  - Single, in-order issue
  - Clock rate vs. IPC
- Data Hazards
  - Hardware: stalling and bypassing
  - Software: pipeline scheduling
- Control Hazards
  - Branch prediction
- Precise state

Readings

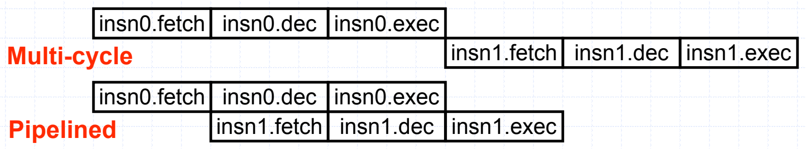
- H+P
  - Appendix A.1 - A.6
- Background slides
  - http://.../~amir/cse371/lecture_notes/pipeline.pdf

Quick Review



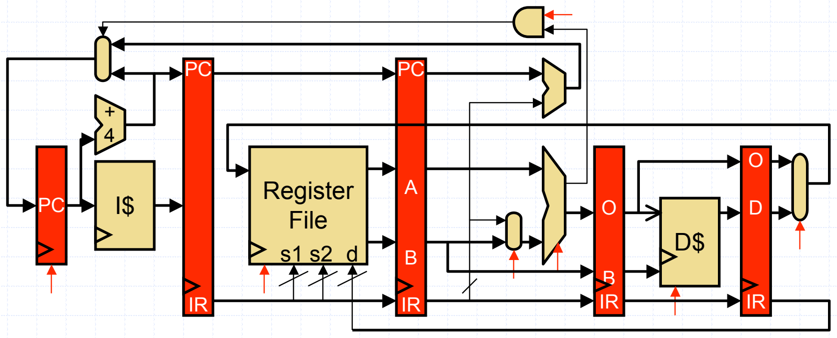
- Basic **datapath**: fetch, decode, execute
- **Single-cycle control**: hardwired
  - + Low CPI (1)
  - – Long clock period (to accommodate slowest instruction)
- **Multi-cycle control**: micro-programmed
  - + Short clock period
  - – High CPI
- Can we have both low CPI and short clock period?
  - Not if datapath executes only one instruction at a time
  - No good way to make a single instruction go faster

Pipelining



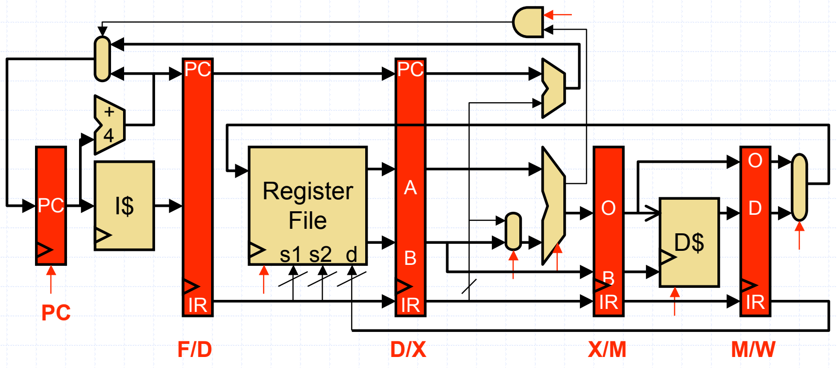
- Important performance technique
  - **Improves instruction throughput rather instruction latency**
- Begin with multi-cycle design
  - When instruction advances from stage 1 to 2
  - Allow next instruction to enter stage 1
  - Form of parallelism: "insn-stage parallelism"
  - Individual instruction takes the same number of stages
  - + **But instructions enter and leave at a much faster rate**
- Automotive assembly line analogy

5 Stage Pipelined Datapath



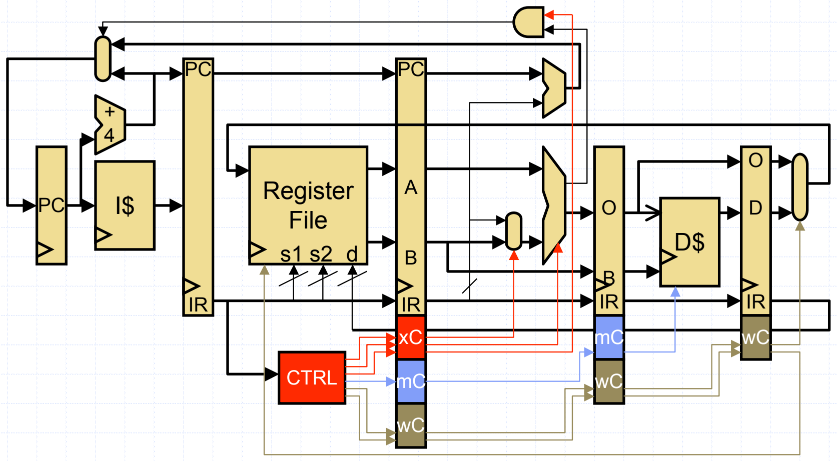
- Temporary values (PC,IR,A,B,O,D) re-latched every stage
  - Why? 5 insns may be in pipeline at once, they share a single PC?
  - Notice, PC not latched after ALU stage (why not?)

Pipeline Terminology



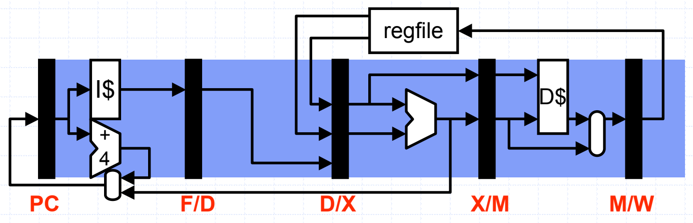
- Five stage: **F**etch, **D**ecode, e**X**ecute, **M**emory, **W**riteback
  - Nothing magical about the number 5 (Pentium 4 has 22 stages)
- Latches (pipeline registers) named by stages they separate
  - **PC**, **F/D**, **D/X**, **X/M**, **M/W**

Pipeline Control



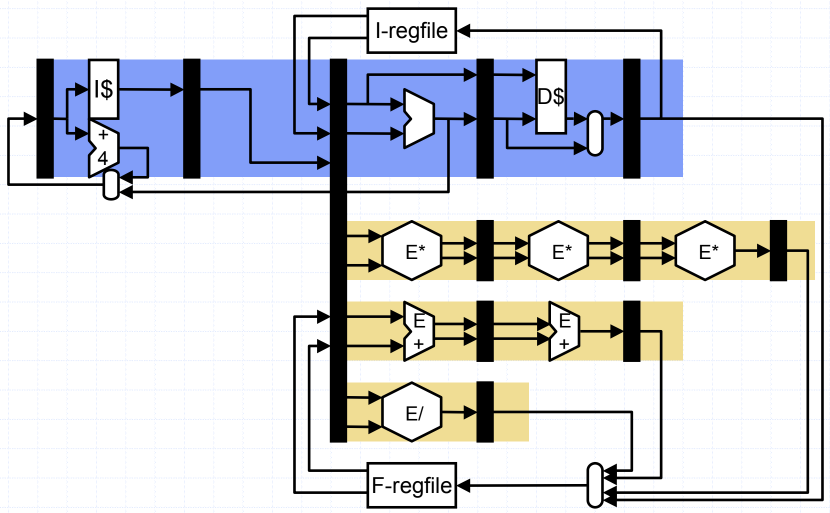
- One single-cycle controller, but pipeline the control signals

Abstract Pipeline



- This is an **integer pipeline**
  - Execution stages are X,M,W
- Usually also one or more **floating-point (FP) pipelines**
  - Separate FP register file
  - One "pipeline" per functional unit: E+, E*, E/
    - "Pipeline": functional unit need not be pipelined (e.g, E/)
  - Execution stages are E+,E+,W (no M)

Floating Point Pipelines



Pipeline Diagram

	1	2	3	4	5	6	7	8	9
`add r3,r2,r1`	F	D	X	M	W				
`ld r4,0(r5)`		F	D	X	M	W			
`st r6,4(r7)`			F	D	X	M	W		

- **Pipeline diagram**
  - Cycles across, insns down
  - Convention: **X** means `ld r4,0(r5)` finishes execute stage and writes into X/M latch at end of cycle 4
- Reverse stream analogy
  - "Downstream": earlier stages, younger insns
  - "Upstream": later stages, older insns
  - Reverse? instruction stream fixed, pipeline flows over it
    - Architects see instruction stream as fixed by program/compiler

Pipeline Performance Calculation

- Back of the envelope calculation
  - Branch: 20%, load: 20%, store: 10%, other: 50%
- Single-cycle
  - Clock period = 50ns, CPI = 1
  - Performance = 50ns/insn
- Pipelined
  - Clock period = **12ns**
  - CPI = **1** (each insn takes 5 cycles, but 1 completes each cycle)
  - Performance = **12ns/insn**

Principles of Pipelining

- Let: insn execution require **N** stages, each takes t_n time
  - $\mathbf{L_1}$ (1-insn latency) = $\sum t_n$
  - **T** (throughput) = $1/L_1$
  - $\mathbf{L_M}$ (M-insn latency, where M>>1) = $M*L_1$
- Now: N-stage pipeline
  - $L_{1+P} = L_1$
  - $T_{+P} = 1/\max(t_n) \leq \mathbf{N}/L_1$
    - If t_n are equal (i.e., $\max(t_n) = L_1/N$), throughput = N/L_1
  - $L_{M+P} = M*\max(t_n) \geq M*L_1/\mathbf{N}$
  - S_{+P} (speedup) = $[M*L_1 / (\geq M*L_1/N)]$ = **≤ N**
- Q: for arbitrarily high speedup, use arbitrarily high N?

No, Part I: Pipeline Overhead

- Let: **O** be extra delay per pipeline stage
  - Latch overhead: pipeline latches take time
  - Clock/data skew
- Now: N-stage pipeline with overhead
  - Assume $\max(t_n) = L_1/N$
  - $L_{1+P+O} = L_1 + N*O$
  - $T_{+P+O} = 1/(L_1/N + O) = 1/(1/T + O) \leq T$, $\mathbf{\leq 1/O}$
  - $L_{M+P+O} = M*L_1/N + M*O = L_{M+P} + M*O$
  - $S_{+P+O} = [M*L_1 / (M*L_1/N + M*O)] = \leq N = S_{+P}$, $\mathbf{\leq L_1/O}$
- **O** limits throughput and speedup → useful N

No, Part II: Hazards

- **Dependence**: relationship that serializes two insns
  - **Structural**: two insns want to use same structure, one must wait
  - **Data**: two insns use same storage location
  - **Control**: one instruction affects whether another executes at all
- **Hazard**: dependence and both insns in pipeline together
  - Possibility for getting order wrong
  - Often fixed with **stalls**: insn stays in same stage for multiple cycles
- Let: **H** be average number of hazard stall cycles per instruction
  - $L_{1+P+H} = L_{1+P}$ (no hazards for one instruction)
  - $T_{+P+H} = [N/(N+H)]*N/L_1 = [N/(N+H)] * T_{+P}$
  - $L_{M+P+H} = M* L_1/N * [(N+H)/N] = [(N+H)/N] * L_{M+P}$
  - $S_{+P+H} = M*L_1 / M*L_1/N*[(N+H)/N] = [N/(N+H)]*S_{+P}$
- **H** also limit throughput, speedup → useful N
  - N↑→ H↑ (more insns "in flight" → dependences become hazards)
  - Exact H depends on program, requires detailed simulation

Clock Rate vs. IPC

- Deeper pipeline (bigger N)
  - + frequency↑
  - – IPC↓
  - Ultimate metric is *IPC * frequency*
    - But people buy *frequency*, not *IPC * frequency*
- Trend has been for deeper pipelines
  - Intel example:
    - 486: 5 stages (50+ gate delays / clock)
    - Pentium: 7 stages
    - Pentium II/III: 12 stages
    - Pentium 4: 22 stages (10 gate delays / clock)
    - 800 MHz Pentium III was faster than 1 GHz Pentium4
    - Next Intel core: fewer pipeline stages than Pentium 4

Optimizing Pipeline Depth

- Parameterize clock cycle in terms of gate delays
  - G gate delays to process (fetch, decode, execute) a single insn
  - O gate delays overhead per stage
  - X average stall per instruction per stage
    - Simplistic: real X function much, much more complex
- Compute optimal N (pipeline stages) given G,O,X
  - IPC = 1 / (1 + X * N)
  - f = 1 / (G / N + O)
  - Example: G = 80, O = 1, X = 0.16,

N	IPC = 1/(1+0.16*N)	freq=1/(80/N+1)	IPC*freq
5	**0.56**	0.059	0.033
10	0.38	0.110	**0.042**
20	0.33	**0.166**	0.040

Managing a Pipeline

- Proper flow requires two pipeline operations
  - Mess with latch write-enable and clear signals to achieve
- Operation I: **stall**
  - Effect: stops some insns in their current stages
  - Use: make younger insns wait for older ones to complete
  - Implementation: de-assert write-enable
- Operation II: **flush**
  - Effect: removes insns from current stages
  - Use: see later
  - Implementation: assert clear signals
- Both stall and flush must be propagated to younger insns

Structural Hazards

	1	2	3	4	5	6	7	8	9
`ld r2,0(r1)`	F	D	X	**M**	W				
`add r1,r3,r4`		F	D	X	M	W			
`sub r1,r3,r5`			F	D	X	M	W		
`st r6,0(r1)`				**F**	D	X	M	W	

- **Structural hazard**: resource needed twice in one cycle
  - Example: shared I/D$

Fixing Structural Hazards

	1	2	3	4	5	6	7	8	9
`ld r2,0(r1)`	F	D	X	M	W				
`add r1,r3,r4`		F	D	X	M	W			
`sub r1,r3,r5`			F	D	X	M	W		
`st r6,0(r1)`				**s***	F	D	X	M	W

- Can fix structural hazards by stalling
  - **s*** = structural stall
  - Q: which one to stall: `ld` or `st`?
    - Always safe to stall younger instruction (here `st`)
      - Fetch stall logic: (D/X.op == `ld` || D/X.op == `st`)
    - But not always the best thing to do performance wise (?)

  + Low cost, simple

  – Decreases IPC
  - Upshot: better to avoid by design, then to fix

Avoiding Structural Hazards

- Replicate the contended resource
  - + No IPC degradation
  - – Increased area, power, latency (interconnect delay?)
  - For cheap, divisible, or highly contended resources (e.g, I$/D$)
- Pipeline the contended resource
  - + No IPC degradation, low area, power overheads
  - – Sometimes tricky to implement (e.g., for RAMs)
  - For multi-cycle resources (e.g., multiplier)
- Design ISA/pipeline to reduce structural hazards (RISC)
  - Each insn uses a resource at most once (same insn hazards)
  - Always in same pipe stage (hazards between two of same insn)
    - Reason why integer operations forced to go through M stage
  - And always for one cycle

Data Hazards

- Real insn sequences pass values via registers/memory
  - Three kinds of **data dependences** (where's the fourth?)

`add r2,r3➔r1`
`sub r1,r4➔r2`
`or r6,r3➔r1`
	`add r2,r3➔r1`
	`sub r5,r4➔r2`
	`or r6,r3➔r1`
		`add r2,r3➔r1`
		`sub r1,r4➔r2`
		`or r6,r3➔r1`

Read-after-write (RAW) True-dependence	Write-after-read (WAR) Anti-dependence	Write-after-write (WAW) Output-dependence

  - Only one dependence between any two insns (RAW has priority)

- **Data hazards**: function of data dependences and pipeline
  - Potential for executing dependent insns in wrong order
  - Require both insns to be in pipeline ("in flight") simultaneously

Dependences and Loops

- Data dependences in loops
  - **Intra-loop**: within same iteration
  - **Inter-loop**: across iterations
  - Example: DAXPY (**D**ouble precision **A X P**lus **Y**)

```
for (i=0;i<100;i++)
  Z[i]=A*X[i]+Y[i];

0: ldf f2,X(r1)
1: mulf f2,f0,f4
2: ldf f6,Y(r1)
3: addf f4,f6,f8
4: stf f8,Z(r1)
5: addi r1,8,r1
6: cmplti r1,800,r2
7: beq r2,Loop
```

- RAW intra: 0→1(`f2`), 1→3(`f4`), 2→3(`f6`), 3→4(`f8`), 5→6(`r1`), 6→7(`r2`)
- RAW inter: 5→0(`r1`), 5→2(`r1`), 5→4(`r1`), 5→5(`r1`)
- WAR intra: 0→5(`r1`), 2→5(`r1`), 4→5(`r1`)
- WAR inter: 1→0(`f2`), 3→1(`f4`), 3→2(`f6`), 4→3(`f8`), 6→5(`r1`), 7→6(`r2`)
- WAW intra: none
- WAW inter: 0→0(`f2`), 1→1(`f4`), 2→2(`f6`), 3→3(`f8`), 6→6(`r2`)

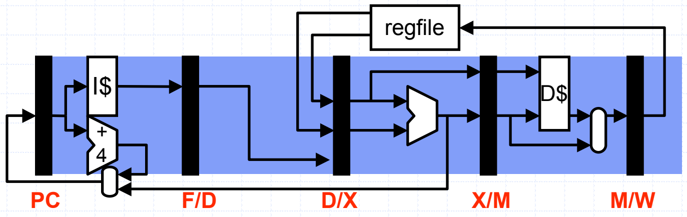
RAW

- **Read-after-write (RAW)**

```
add r2,r3➔r1
sub r1,r4➔r2
or r6,r3➔r1
```

  - Problem: swap would mean `sub` uses wrong value for `r1`
  - **True**: value flows through this dependence
    - Using different output register for `add` doesn't help

RAW: Detect and Stall



- **Stall logic**: detect and stall reader in D
  (F/D.rs1 & (F/D.rs1==D/X.rd | F/D.rs1==X/M.rd | F/D.rs1==M/W.rd)) |
  (F/D.rs2 & (F/D.rs2==D/X.rd | F/D.rs2==X/M.rd | F/D.rs2==M/W.rd))
  - Re-evaluated every cycle until no longer true
  + Low cost, simple
  – IPC degradation, dependences are the common case

Two Stall Timings (without bypassing)

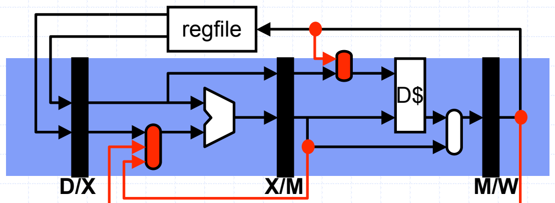
- Depend on how D and W stages share regfile
  - Each gets regfile for half a cycle
  – 1st half D reads, 2nd half W writes 3 cycle stall
  - **d*** = data stall, **p*** = propagated stall

	1	2	3	4	5	6	7	8	9	10
add r2,r3→r1	F	D	X	M	W					
sub r1,r4→r2		F	d*	d*	d*	D	X	M	W	
add r5,r6→r7			p*	p*	p*	F	D	X	M	W

  + 1st half W writes, 2nd half D reads 2 cycle stall
  - How does the stall logic change here?

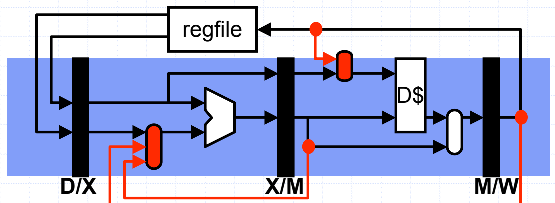
	1	2	3	4	5	6	7	8	9	10
add r2,r3→r1	F	D	X	M	W					
sub r1,r4→r2		F	d*	d*	D	X	M	W		
add r5,r6→r7			p*	p*	F	D	X	M	W	

Reducing RAW Stalls with Bypassing



- Why wait until W stage? Data available after X or M stage
  - **Bypass** (aka **forward**) data directly to input of X or M
    - **MX**: from beginning of M (X output) to input of X
    - **WX**: from beginning of W (M output) to input of X
    - **WM**: from beginning of W (M output) to data input of M
    - Two each of MX, WX (figure shows 1) + WM = **full bypassing**
  + Reduces stalls in a big way
  – Additional wires and muxes may increase clock cycle

Bypass Logic



- Bypass logic: similar to but separate from stall logic
  - Stall logic controls latches, bypass logic controls mux inputs
  - Complement one another: can't bypass → must stall
  - ALU input mux bypass logic
    - (D/X.rs2 & X/M.rd==D/X.rs2) → 2 // check first
    - (D/X.rs2 & M/W.rd==D/X.rs2) → 1 // check second
    - (D/X.rs2) → 0 // check last

Pipeline Diagrams with Bypassing

- If bypass exists, "from"/"to" stages execute in same cycle
  - Example: full bypassing, use MX bypass

	1	2	3	4	5	6	7	8	9	10
`add r2,r3→r1`	F	D	X	**M**	W					
`sub r1,r4→r2`		F	D	**X**	M	W				

  - Example: full bypassing, use WX bypass

	1	2	3	4	5	6	7	8	9	10
`add r2,r3→r1`	F	D	X	M	**W**					
`ld [r7]→r5`		F	D	X	M	W				
`sub r1,r4→r2`			F	D	**X**	M	W			

  - Example: WM bypass

	1	2	3	4	5	6	7	8	9	10
`add r2,r3→r1`	F	D	X	M	**W**					
`?`		F	D	X	**M**	W				

    - Can you think of a code example that uses the WM bypass?

Load-Use Stalls

- Even with full bypassing, stall logic is unavoidable
  - **Load-use stall**
    - Load value not ready at beginning of M → can't use MX bypass
    - Use WX bypass

	1	2	3	4	5	6	7	8	9	10
`ld [r3+4]→r1`	F	D	X	M	**W**					
`sub r1,r4→r2`		F	D	**d***	**X**	M	W			

- Aside: with WX bypassing, stall logic can be in D or X

	1	2	3	4	5	6	7	8	9	10
`ld [r3+4]→r1`	F	D	X	M	**W**					
`sub r1,r4→r2`		F	**d***	D	**X**	M	W			

- Aside II: how does stall/bypass logic handle cache misses?

Compiler Scheduling

- Compiler can schedule (move) insns to reduce stalls
  - **Basic pipeline scheduling**: eliminate back-to-back load-use pairs
  - Example code sequence: `a = b + c; d = f – e;`
  - MIPS Notation:
    - "ld r2,4(sp)" is "ld [sp+4]→r2" "st r1, 0(sp)" is "st r1→[sp+0]"

```
Before                                   After

ld r2,4(sp)                              ld r2,4(sp)
ld r3,8(sp)                              ld r3,8(sp)
add r3,r2,r1     //stall                 ld r5,16(sp)
st r1,0(sp)                              add r3,r2,r1     //no stall
ld r5,16(sp)                             ld r6,20(sp)
ld r6,20(sp)                             st r1,0(sp)
sub r5,r6,r4     //stall                 sub r5,r6,r4     //no stall
st r4,12(sp)                             st r4,12(sp)
```

Compiler Scheduling Requires

- **Large scheduling scope**
  - Independent instruction to put between load-use pairs
  - + Original example: large scope, two independent computations
  - – This example: small scope, one computation

```
Before                                   After

ld r2,4(sp)                              ld r2,4(sp)
ld r3,8(sp)                              ld r3,8(sp)
add r3,r2,r1     //stall                 add r3,r2,r1     //stall
st r1,0(sp)                              st r1,0(sp)
```

Compiler Scheduling Requires

- **Enough registers**
  - To hold additional "live" values
  - Example code contains 7 different values (including `sp`)
  - Before: max 3 values live at any time → 3 registers enough
  - After: max 4 values live → 3 registers not enough → WAR violations

```
Original                          Wrong!

ld r2,4(sp)                       ld r2,4(sp)
ld r1,8(sp)                       ld r1,8(sp)
add r1,r2,r1    //stall           ld r2,16(sp)
st r1,0(sp)                       add r1,r2,r1   //WAR
ld r2,16(sp)                      ld r1,20(sp)
ld r1,20(sp)                      st r1,0(sp)    //WAR
sub r2,r1,r1    //stall           sub r2,r1,r1
st r1,12(sp)                      st r1,12(sp)
```

Compiler Scheduling Requires

- **Alias analysis**
  - Ability to tell whether load/store reference same memory locations
    - Effectively, whether load/store can be rearranged
  - Example code: easy, all loads/stores use same base register (`sp`)
  - New example: can compiler tell that `r8` = `sp`?

```
Before                            Wrong(?)

ld r2,4(sp)                       ld r2,4(sp)
ld r3,8(sp)                       ld r3,8(sp)
add r3,r2,r1    //stall           ld r5,0(r8)
st r1,0(sp)                       add r3,r2,r1
ld r5,0(r8)                       ld r6,4(r8)
ld r6,4(r8)                       st r1,0(sp)
sub r5,r6,r4    //stall           sub r5,r6,r4
st r4,8(r8)                       st r4,8(r8)
```

WAW Hazards

- **Write-after-write (WAW)**
  ```
  add r2,r3,r1
  sub r1,r4,r2
  or r6,r3,r1
  ```
- Compiler effects
  - Scheduling problem: reordering would leave wrong value in `r1`
    - Later instruction reading `r1` would get wrong value
  - **Artificial**: no value flows through dependence
    - Eliminate using different output register name for `or`
- Pipeline effects
  - Doesn't affect in-order pipeline with single-cycle operations
    - One reason for making ALU operations go through M stage
  - Can happen with multi-cycle operations (e.g., FP or cache misses)

Handling WAW Hazards

	1	2	3	4	5	6	7	8	9	10
`div f0,f1→f2`	F	D	E/	E/	E/	E/	E/	**W**		
`stf f2→[r1]`		F	D	d*	d*	d*	X	M	W	
`addf f0,f1→f2`			F	D	E+	E+	**W**			

- What to do?
  - Option I: stall younger instruction (`addf`) at writeback
    - + Intuitive, simple
    - – Lower performance, cascading W structural hazards
  - Option II: cancel older instruction (`divf`) writeback
    - + No performance loss
    - – What if `divf` or `stf` cause an exception (e.g., /0, page fault)?

Handling Interrupts/Exceptions

- How are interrupts/exceptions handled in a pipeline?
  - **Interrupt**: external, e.g., timer, I/O device requests
  - **Exception**: internal, e.g., /0, page fault, illegal instruction
  - We care about **restartable** interrupts (e.g. `stf` page fault)

	1	2	3	4	5	6	7	8	9	10
`divf f0,f1→f2`	F	D	E/	E/	E/	E/	E/	W		
`stf f2→[r1]`		F	D	d*	d*	d*	X	**M**	W	
`addf f0,f1→f2`			F	D	E+	E+	W			

- VonNeumann says
  - "Insn execution should appear sequential and atomic"
    - Insn X should complete before instruction X+1 should begin
    - + Doesn't physically have to be this way (e.g., pipeline)
    - But be ready to restore to this state at a moments notice
  - Called **precise state** or **precise interrupts**

Handling Interrupts

	1	2	3	4	5	6	7	8	9	10
`divf f0,f1→f2`	F	D	E/	E/	E/	E/	E/	W		
`stf f2→[r1]`		F	D	d*	d*	d*	X	**M**	W	
`addf f0,f1→f2`			F	D	E+	E+	W			

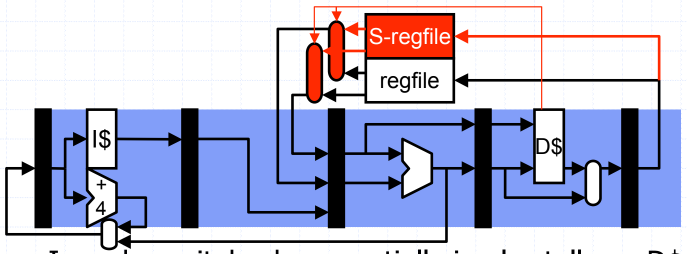
- In this situation
  - Make it appear as if `divf` finished and `stf`, `addf` haven't started
  - Allow `divf` to writeback
  - **Flush** `stf` and `addf` (so that's what a flush is for)
    - But `addf` has already written back
    - – Keep an "undo" register file? Complicated
    - – Force in-order writebacks? Slow
  - Invoke exception handler
  - Restart `stf`

More Interrupt Nastiness

	1	2	3	4	5	6	7	8	9	10
`divf f0,f1→f2`	F	D	E/	E/	E/	E/	E/	W		
`stf f2→[r1]`		F	D	d*	d*	d*	X	**M**	W	
`divf f0,f4→f2`			F	D	**E/**	E/	E/	E/	E/	W

- What about two simultaneous in-flight interrupts
  - Example: `stf` page fault, `divf` /0
  - Interrupts must be handled in program order (`stf` first)
    - Handler for `stf` must see program as if `divf` hasn't started
  - Must defer interrupts until writeback **and** force in-order writeback
  - Kind of a bogus example, /0 is non-restartable
- In general: interrupts are really nasty
  - Some processors (Alpha) only implement precise integer interrupts
  - Easier because fewer WAW scenarios
  - Most floating-point interrupts are non-restartable anyway

Research: Runahead Execution



- In-order writebacks essentially imply stalls on D$ misses
  - Can save power … or use idle time for performance
- **Runahead execution** [Dundas+]
  - Shadow regfile kept in sync with main regfile (write to both)
  - D$ miss: continue executing using shadow regfile (disable stores)
  - D$ miss returns: flush pipe and restart with stalled PC
  - + Acts like a smart prefetch engine
  - + Performs better as cache t_{miss} grows (relative to clock period)

WAR Hazards

- **Write-after-read (WAR)**
  ```
  add r2,r3,r1
  sub r5,r4,r2
  or r6,r3,r1
  ```
- Compiler effects
  - Scheduling problem: reordering would mean `add` uses wrong value for `r2`
  - **Artificial**: solve using different output register name for `sub`
- Pipeline effects
  - Can't happen in simple in-order pipeline
  - Can happen with out-of-order execution (after mid-term)

Memory Data Hazards

- So far, have seen/dealt with register dependences
  - Dependences also exist through memory

`st r2→[r1]`	`st r2→[r1]`	`st r2→[r1]`
`ld [r1]→r4`	`ld [r1]→r4`	`ld [r1]→r4`
`st r5→[r1]`	`st r5→[r1]`	`st r5→[r1]`
Read-after-write (RAW)	Write-after-read (WAR)	Write-after-write (WAW)

  - But in an in-order pipeline like ours, they do not become hazards
  - Memory read and write happen at the same stage
    - Register read happens three stages earlier than register write
  - In general: memory dependences more difficult than register

	1	2	3	4	5	6	7	8	9	10
`st r2→[r1]`	F	D	X	**M**	W					
`ld [r1]→r4`		F	D	X	**M**	W				

Control Hazards

- **Control hazards**
  - Must fetch post branch insns before branch outcome is known
  - Default: assume "**not-taken**" (at fetch, can't tell it's a branch)
  - Control hazards indicated with **c*** (or not at all)
  - Taken branch penalty is 2 cycles

	1	2	3	4	5	6	7	8	9
`addi r1,1→r3`	F	D	X	M	W				
`bnez r3,targ`		F	D	X	M	W			
`st r6→[r7+4]`			**c***	**c***	F	D	X	M	W

- Back of the envelope calculation
  - **Branch: 20%**, other: 80%, **75% of branches are taken**
  - $CPI_{BASE} = 1$
  - $CPI_{BASE+BRANCH} = 1 + 0.20*0.75*2 = 1.3$
  - – **Branches cause 30% slowdown**

ISA Branch Techniques

- **Fast branch**: resolves at D, not X
  - Test must be comparison to zero or equality, no time for ALU
  - + New taken branch penalty is 1
  - – Additional comparison insns (e.g., `cmplt`, `slt`) for complex tests
  - – Must bypass into decode now, too
- **Delayed branch**: branch that takes effect one insn later
  - Insert insns that are independent of branch into "branch delay slot"
  - Preferably from before branch (always helps then)
  - But from after branch OK too
    - As long as no undoable effects (e.g., a store)
  - Upshot: short-sighted feature (MIPS regrets it)
    - – Not a big win in today's pipelines
    - – Complicates interrupt handling

Big Idea: Speculation

- **Speculation**
  - "Engagement in risky transactions on the chance of profit"
- **Speculative execution**
  - Execute before all parameters known with certainty
- **Correct speculation**
  + Avoid stall, improve performance
- **Incorrect speculation (mis-speculation)**
  – Must abort/flush/squash incorrect instructions
  – Must undo incorrect changes (recover pre-speculation state)

The "game": **[$\%_{correct}$ * gain] – [(1–$\%_{correct}$) * penalty]**

Control Hazards: Control Speculation

- Deal with control hazards with **control speculation**
  - Unknown parameter: are these the correct insns to execute next?
- Mechanics
  - Guess branch target, start fetching at guessed position
  - Execute branch to verify (check) guess
    - Correct speculation? keep going
    - Mis-speculation? Flush mis-speculated insns
  - Don't write registers or memory until prediction verified
- Speculation game for in-order 5 stage pipeline
  - Gain = 2 cycles
  - Penalty = 0 cycles
    - No penalty → mis-speculation no worse than stalling
  - $\%_{correct}$ = **branch prediction**
    - Static (compiler) OK, **dynamic** (hardware) much better

Control Speculation and Recovery

Correct:

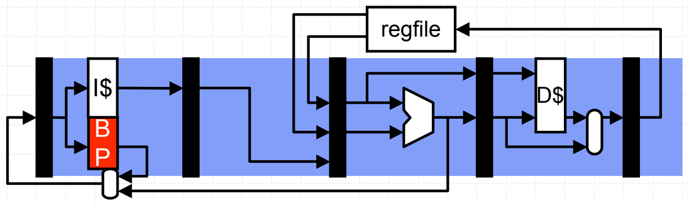
		1	2	3	4	5	6	7	8	9
Correct:	`addi r1,1→r3`	F	D	X	M	W				
	`bnez r3,targ`		F	D	X	M	W			
	`st r6→[r7+4]`			**F**	**D**	X	M	W		
	`targ:add r4,r5→r4`				**F**	D	X	M	W	

speculative

- **Mis-speculation recovery**: what to do on wrong guess
  - Not too painful in an in-order pipeline
  - Branch resolves in X
  + Younger insns (in F, D) haven't changed permanent state
  - **Flush** insns currently in F/D and D/X (i.e., replace with `nops`)

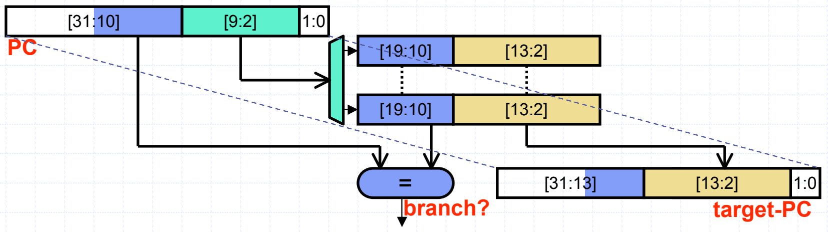
		1	2	3	4	5	6	7	8	9
Recovery:	`addi r1,1→r3`	F	D	X	M	W				
	`bnez r3,targ`		F	D	**X**	M	W			
	~~`st r6→[r7+4]`~~			**F**	**D**	--	--	--		
	~~`targ:add r4,r5→r4`~~				**F**	--	--	--	--	
	`targ:add r4,r5→r4`					**F**	D	X	M	W

Dynamic Branch Prediction



- BP part I: **target predictor**
  - Applies to all control transfers
  - Supplies target PC, tells if insn is a branch prior to decode
  + Easy
- BP part II: **direction predictor**
  - Applies to conditional branches only
  - Predicts taken/not-taken
  – Harder

Branch Target Buffer



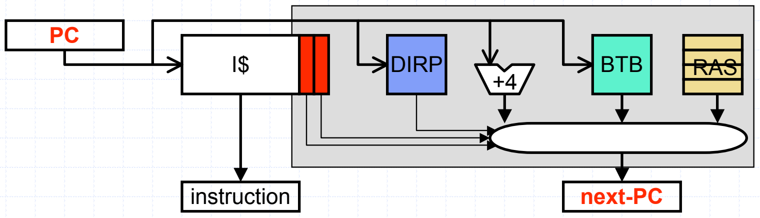
- **Branch target buffer (BTB)**
  - A small cache: address = PC, data = target-PC
    - Hit? This is a control insn and it's going to target-PC (if "taken")
    - Miss? Not a control insn, or one I have never seen before
  - Partial data/tags: full tag not necessary, target-PC is just a guess
    - **Aliasing**: tag match, but not actual match (OK for BTB)
  - Pentium4 BTB: 2K entries, 4-way set-associative

Why Does a BTB Work?

- Because control insn targets are stable
  - **Direct** means constant target, **indirect** means register target
  - + Direct conditional branches? Check
  - + Direct calls? Check
  - + Direct unconditional jumps? Check

  - + Indirect conditional branches? Not that useful→not widely supported
  - Indirect calls? Two idioms
    - + Dynamically linked functions (DLLs)? Check
    - + Dynamically dispatched (virtual) functions? Pretty much check
  - Indirect unconditional jumps? Two idioms
    - – Switches? Not really, but these are rare
    - – Returns? Nope, but…

Return Address Stack (RAS)



- Return addresses are easy to predict without a BTB
  - Hardware **return address stack (RAS)** tracks call sequence
  - Calls push PC+4 onto RAS
  - Prediction for returns is RAS[TOS]
  - Q: how can you tell if an insn is a return before decoding it?
    - RAS is not a cache
  - A: attach **pre-decode bits** to I$
    - Written after first time insn executes
    - Two useful bits: return?, conditional-branch?

Branch Direction Prediction

- **Direction predictor (DIRP)**
  - Map conditional-branch PC to taken/not-taken (T/N) decision
  - Seemingly innocuous, but quite difficult
  - Individual conditional branches often unbiased or weakly biased
    - 90%+ one way or the other considered **"biased"**

Branch History Table (BHT)

- **Branch history table (BHT)**: simplest direction predictor
  - PC indexes table of bits (0 = N, 1 = T), no tags
  - Essentially: branch will go same way it went last time
  - Problem: consider **inner loop branch** below (***** = mis-prediction)

```
for (i=0;i<100;i++)
   for (j=0;j<3;j++)
      // whatever
```

State/prediction	**N***	**T**	**T**	**T***	**N***	**T**	**T**	**T***	**N***	**T**	**T**	**T***
Outcome	T	T	T	N	T	T	T	N	T	T	T	N

– Two "built-in" mis-predictions per inner loop iteration
– Branch predictor "changes its mind too quickly"

Two-Bit Saturating Counters (2bc)

- **Two-bit saturating counters (2bc)** [Smith]
  - Replace each single-bit prediction
    - (0,1,2,3) = (N,n,t,T)
  - Force DIRP to mis-predict twice before "changing its mind"

State/prediction	**N***	**n***	**t**	**T***	**t**	**T**	**T**	**T***	**t**	**T**	**T**	**T***
Outcome	T	T	T	N	T	T	T	N	T	T	T	N

+ Fixes this pathology (which is not contrived, by the way)

Correlated Predictor

- **Correlated (two-level) predictor** [Patt]
  - Exploits observation that branch outcomes are correlated
  - Maintains separate prediction per (PC, BHR)
    - **Branch history register (BHR)**: recent branch outcomes
  - Simple working example: assume program has one branch
    - BHT: one 1-bit DIRP entry
    - BHT+**2BHR**: **4** 1-bit DIRP entries

State/prediction	BHR=NN	**N***	T	T	T	T	T	T	T	T	T	T	T
"active pattern"	BHR=NT	N	**N***	T	T	T	**T**	T	T	T	**T**	T	T
	BHR=TN	N	N	N	N	**N***	T	T	T	**T**	T	T	T
	BHR=TT	N	N	**N***	**T***	N	N	**N***	**T***	N	N	**N***	**T***
Outcome		T	T	T	N	T	T	T	N	T	T	T	N

– We didn't make anything better, what's the problem?

Correlated Predictor

- What happened?
  - BHR wasn't long enough to capture the pattern
  - Try again: BHT+**3BHR**: **8** 1-bit DIRP entries

State/prediction	BHR=NNN	**N***	T	T	T	T	T	T	T	T	T	T	T
	BHR=NNT	N	**N***	T	T	T	T	T	T	T	T	T	T
	BHR=NTN	N	N	N	N	N	N	N	N	N	N	N	N
"active pattern"	BHR=NTT	N	N	**N***	T	T	T	**T**	T	T	T	**T**	T
	BHR=TNN	N	N	N	N	N	N	N	N	N	N	N	N
	BHR=TNT	N	N	N	N	N	**N***	T	T	T	**T**	T	T
	BHR=TTN	N	N	N	N	**N***	T	T	T	**T**	T	T	T
	BHR=TTT	N	N	N	**N**	N	N	N	**N**	N	N	N	N
Outcome		T	T	T	N	T	T	T	N	T	T	T	N

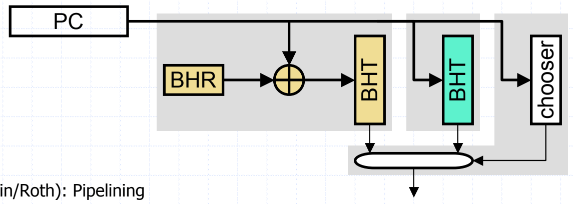
+ No mis-predictions after predictor learns all the relevant patterns

Correlated Predictor

- Design choice I: one **global** BHR or one per PC (**local**)?
  - Each one captures different kinds of patterns
  - Global is better, captures local patterns for tight loop branches
- Design choice II: how many history bits (BHR size)?
  - Tricky one
  - + Given unlimited resources, longer BHRs are better, but…
  - – BHT utilization decreases
    - – Many history patterns are never seen
    - – Many branches are history independent (don't care)
    - PC ^ BHR allows multiple PCs to dynamically share BHT
    - BHR length < $\log_2$(BHT size)
  - – Predictor takes longer to train
  - Typical length: 8–12

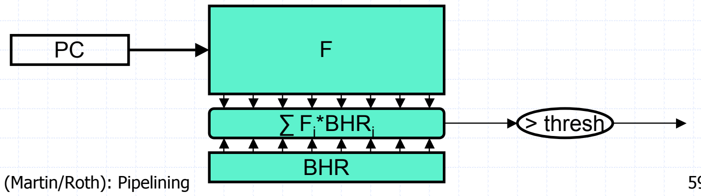
Hybrid Predictor

- **Hybrid (tournament) predictor** [McFarling]
  - Attacks correlated predictor BHT utilization problem
  - Idea: combine two predictors
    - **Simple BHT** predicts history independent branches
    - **Correlated predictor** predicts only branches that need history
    - **Chooser** assigns branches to one predictor or the other
    - Branches start in simple BHT, move mis-prediction threshold
  - + Correlated predictor can be made smaller, handles fewer branches
  - + 90–95% accuracy



Research: Perceptron Predictor

- **Perceptron predictor** [Jimenez]
  - Attacks BHR size problem using machine learning approach
  - BHT replaced by table of function coefficients F_i (signed)
  - Predict taken if $\sum(BHR_i*F_i)$> threshold
  - + Table size #PC*|BHR|*|F| (can use long BHR: ~60 bits)
    - – Equivalent correlated predictor would be #PC*$2^{|BHR|}$
  - How does it learn? Update F_i when branch is taken
    - BHR_i == 1 ? F_i++ : F_i– –;
    - "don't care" F_i bits stay near 0, important F_i bits saturate
  - + Hybrid BHT/perceptron accuracy: 95–98%



Branch Prediction Performance

- Same parameters
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%
  - 75% of branches are taken
- Dynamic branch prediction
  - Branches predicted with 95% accuracy
  - CPI = 1 + 0.20*0.05*2 = **1.02**

Pipeline Performance Summary

- Base CPI is 1, but hazards increase it
- Nothing magical about a 5 stage pipeline
  - Pentium4 has 22 stage pipeline
- Increasing **pipeline depth**
  - + Increases clock frequency (that's why companies do it)
  - – But decreases IPC
  - Branch mis-prediction penalty becomes longer
    - More stages between fetch and whenever branch computes
  - Non-bypassed data hazard stalls become longer
    - More stages between register read and write
  - At some point, CPI losses offset clock gains, question is when?

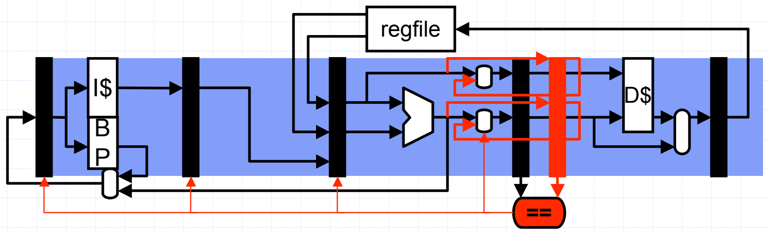
Dynamic Pipeline Power

- Remember control-speculation game
  - [2 cycles * $\%_{correct}$] – [**0 cycles** * (1–$\%_{correct}$)]
  - No penalty → mis-speculation no worse than stalling
  - This is a performance-only view
  - From a power standpoint, mis-speculation is worse than stalling
- **Power control-speculation game**
  - [0 nJ * $\%_{correct}$] – [**X nJ** * (1–$\%_{correct}$)]
  - No benefit → correct speculation no better than stalling
    - Not exactly, increased execution time increases static power
  - How to balance the two?

Research: Speculation Gating

- **Speculation gating** [Manne+]
  - Extend branch predictor to give prediction + **confidence**
  - Speculate on high-confidence (mis-prediction unlikely) branches
  - Stall (save energy) on low-confidence branches
- **Confidence estimation**
  - What kind of hardware circuit estimates confidence?
  - Hard in absolute sense, but easy relative to given threshold
  - Counter-scheme similar to $\%_{miss}$ threshold for cache resizing
  - Example: assume 90% accuracy is high confidence
    - PC-indexed table of confidence-estimation counters
    - Correct prediction? table[PC]+=1 : table[PC]–=9;
    - Prediction for PC is confident if table[PC] > 0;

Research: Razor



- **Razor** [Uht, Ernst+]
  - Identify pipeline stages with narrow signal margins (e.g., **X**)
  - Add **"Razor"** X/M latch: relatches X/M input signals after safe delay
  - Compare X/M latch with "safe" razor X/M latch, different?
    - Flush F,D,X & M
    - Restart M using X/M razor latch, restart F using D/X latch
  - + Pipeline will not "break" → reduce V_{DD} until flush rate too high
  - + Alternatively: "over-clock" until flush rate too high

Summary

- Principles of pipelining
  - Effects of overhead and hazards
  - Pipeline diagrams
- Data hazards
  - Stalling and bypassing
- Control hazards
  - Branch prediction
- Power techniques
  - Dynamic power: speculation gating
  - Static and dynamic power: razor latches