ECE 590.03
Computer Organization and Design

Pipelines

Admin

- Homework
  - Homework 6 out tonight
  - Due Friday (can use late days as needed)
  - Pencil and paper, no coding
- Reading:
  - Remainder of Chapter 4
- Recitation this week:
  - Review for final

Clock Period and CPI

- Single-cycle datapath
  - + Low CPI: 1
  - – Long clock period: to accommodate slowest insn

```
| insn0.fetch, dec, exec |
                         | insn1.fetch, dec, exec |
```

- Multi-cycle datapath
  - + Short clock period
  - – High CPI

```
| insn0.fetch | insn0.dec | insn0.exec |
                                       | insn1.fetch | insn1.dec | insn1.exec |
```

- Can we have both low CPI and short clock period?
  - – No good way to make a single insn go faster
  - + Insn latency doesn't matter anyway … insn throughput matters
  - Key: **exploit inter-insn parallelism**

Pipelining

- **Pipelining**: important performance technique
  - **Improves insn throughput rather than insn latency**
  - **Exploits parallelism at insn-stage level to do so**
  - Begin with multi-cycle design

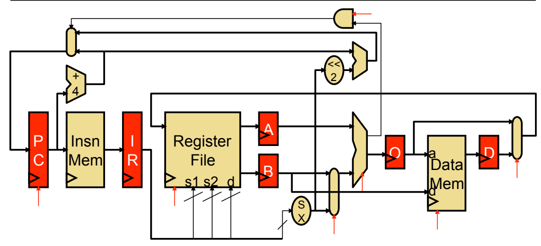
```
| insn0.fetch | insn0.dec | insn0.exec |
                                       | insn1.fetch | insn1.dec | insn1.exec |
```

  - When insn advances from stage 1 to 2, next insn enters stage 1

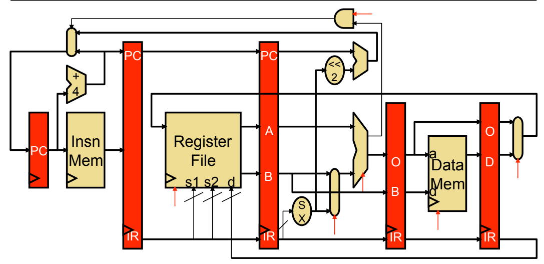
```
| insn0.fetch | insn0.dec   | insn0.exec |
              | insn1.fetch | insn1.dec  | insn1.exec |
```

  - Individual insns take same number of stages
  - + **But insns enter and leave at a much faster rate**
  - Physically breaks "atomic" VN loop ... but must maintain illusion
- Automotive assembly line analogy

5 Stage Multi-Cycle Datapath

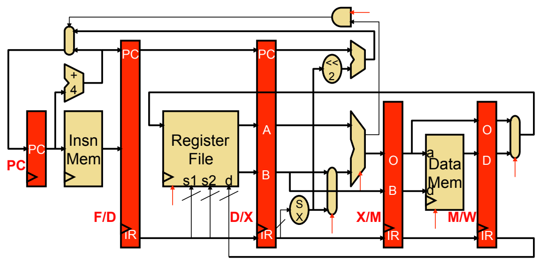


5 Stage Pipelined Datapath



- 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

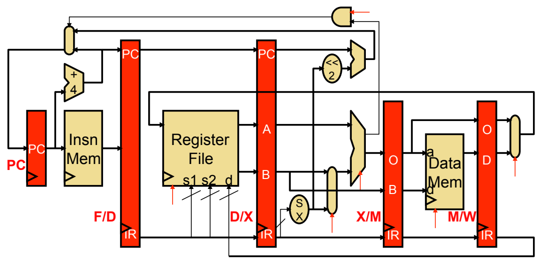


- Stages: **F**etch, **D**ecode, e**X**ecute, **M**emory, **W**riteback
- Latches (pipeline registers): **PC**, **F/D**, **D/X**, **X/M**, **M/W**

Some More Terminology

- **Scalar pipeline**: one insn per stage per cycle
  - Alternative: "superscalar" (take 552)
- **In-order pipeline**: insns enter execute stage in VN order
  - Alternative: "out-of-order" (take 552)
- **Pipeline depth**: number of pipeline stages
  - Nothing magical about five
  - Trend has been to deeper pipelines

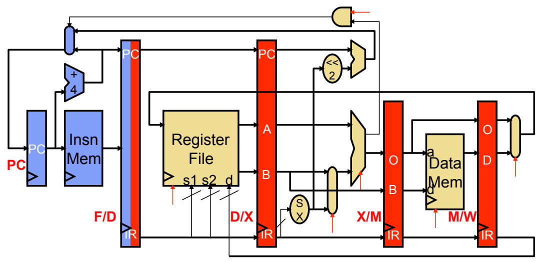
Pipeline Example: Cycle 1



`add $3,$2,$1`

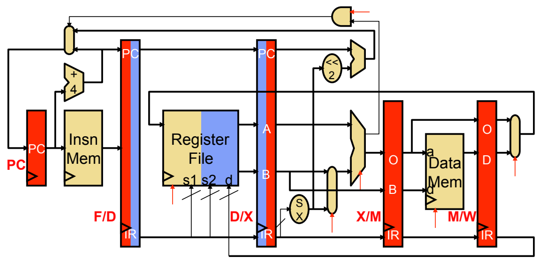
- 3 instructions

Pipeline Example: Cycle 2



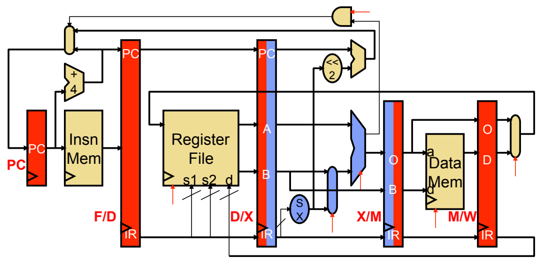
`lw $4,0($5)`   `add $3,$2,$1`

Pipeline Example: Cycle 3



`sw $6,4($7)`   `lw $4,0($5)`   `add $3,$2,$1`

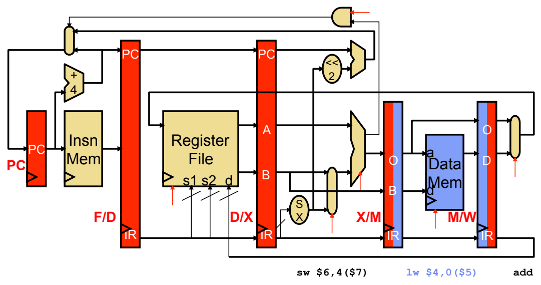
Pipeline Example: Cycle 4



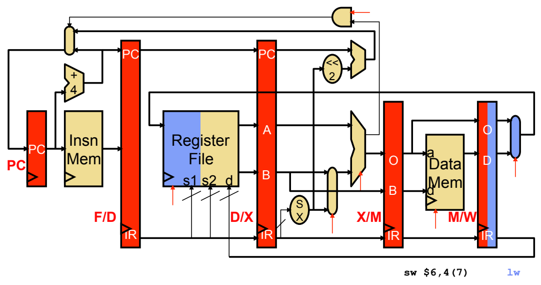
`sw $6,4($7)`   `lw $4,0($5)`   `add $3,$2,$1`

- 3 instructions

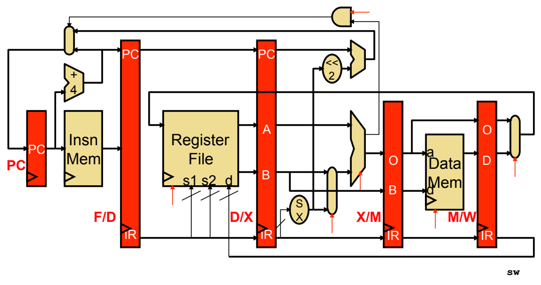
Pipeline Example: Cycle 5



Pipeline Example: Cycle 6



Pipeline Example: Cycle 7



Pipeline Diagram

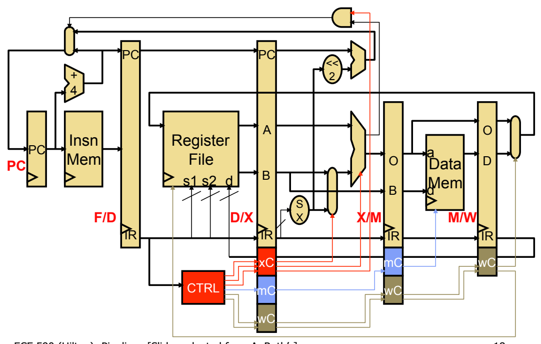
- **Pipeline diagram**: shorthand for what we just saw
  - Across: cycles
  - Down: insns
  - Convention: **X** means `lw $4,0($5)` finishes execute stage and writes into X/M latch at end of cycle 4

	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($5)`		F	D	**X**	M	W			
`sw $6,4($7)`			F	D	X	M	W		

What About Pipelined Control?

- Should it be like single-cycle control?
  - But individual insn signals must be staged
- Should it be like multi-cycle control?
  - But all stages are simultaneously active
- How many different controllers are we going to need?
  - One for each insn in pipeline?
- Solution: use simple single-cycle control, but pipeline it
  - Single controller

Pipelined Control



Pipeline Performance Calculation

- Single-cycle
  - Clock period = 50ns, CPI = 1
  - Performace = 50ns/insn
- Multi-cycle
  - Branch: 20% (3 cycles), load: 20% (5 cycles), other: 60% (4 cycles)
  - Clock period = **12ns**, CPI = (0.2*3+0.2*5+0.6*4) = 4
    - Remember: latching overhead makes it 12, not 10
  - Performance = 48ns/insn
- Pipelined
  - Clock period = **12ns**
  - CPI = **1.5** (on average insn completes every 1.5 cycles)
  - Performance = **18ns/insn**

Q1: Why Is Pipeline Clock Period …

- … > delay thru datapath / number of pipeline stages?
  - Latches (FFs) add delay
  - Pipeline stages have different delays, clock period is max delay
  - Both factors have implications for ideal number pipeline stages

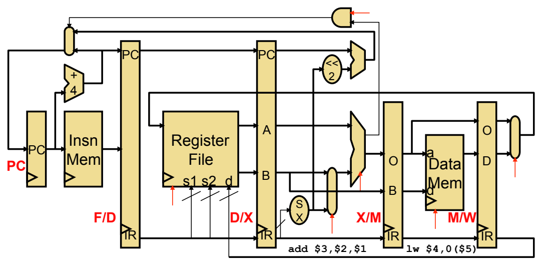
Q2: Why Is Pipeline CPI…

- … > 1?
  - CPI for scalar in-order pipeline is 1 **+ stall penalties**
  - Stalls used to resolve hazards
    - **Hazard**: condition that jeopardizes VN illusion
    - **Stall**: artificial pipeline delay introduced to restore VN illusion
- Calculating pipeline CPI
  - **Frequency of stall** * **stall cycles**
  - Penalties add (stalls generally don't overlap in in-order pipelines)
  - 1 + stall-freq$_1$*stall-cyc$_1$ + stall-freq$_2$*stall-cyc$_2$ + …
- Correctness/performance/MCCF
  - Long penalties OK if they happen rarely, e.g., 1 + 0.01 * 10 = 1.1
  - Stalls also have implications for ideal number of pipeline stages

Dependences and Hazards

- **Dependence**: relationship between two insns
  - **Data**: two insns use same storage location
  - **Control**: one insn affects whether another executes at all
  - Not a bad thing, programs would be boring without them
  - Enforced by making older insn go before younger one
    - Happens naturally in single-/multi-cycle designs
    - But not in a pipeline
- **Hazard**: dependence & possibility of wrong insn order
  - Effects of wrong insn order cannot be externally visible
    - **Stall**: for order by keeping younger insn in same stage
  - Hazards are a bad thing: stalls reduce performance

Why Does Every Insn Take 5 Cycles?

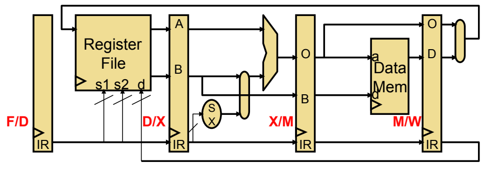


- Could /should we allow `add` to skip M and go to W? No
  - It wouldn't help: peak fetch still only 1 insn per cycle
  - **Structural hazards**: imagine `add` follows `lw`

Structural Hazards

- **Structural hazards**
  - Two insns trying to use same circuit at same time
    - E.g., structural hazard on regfile write port
- **To fix structural hazards**: proper ISA/pipeline design
  - Each insn uses every structure exactly once
  - For at most one cycle
  - Always at same stage relative to F

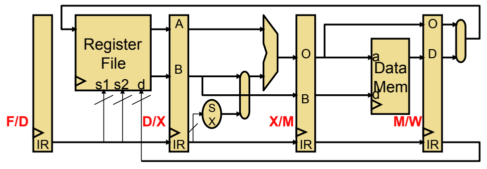
Data Hazards



`sw $6,0($7)` `lw $4,0($5)` `add $3,$2,$1`

- Let's forget about branches and the control for a while
- The three insn sequence we saw earlier executed fine…
  - But it wasn't a real program
  - Real programs have **data dependences**
    - They pass values via registers and memory

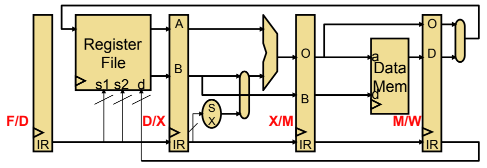
Data Hazards



`sw $3,0($7)` `addi $6,1,$3` `lw $4,0($3)` `add $3,$2,$1`

- Would this "program" execute correctly on this pipeline?
  - Which insns would execute with correct inputs?
  - `add` is writing its result into `$3` in current cycle
  – `lw` read `$3` 2 cycles ago → got wrong value
  – `addi` read `$3` 1 cycle ago → got wrong value
  - `sw` is reading `$3` this cycle → OK (regfile timing: write first half)

Memory Data Hazards



`lw $4,0($1)` `sw $5,0($1)`

- What about data hazards through memory? No
  - `lw` following `sw` to same address in next cycle, gets right value
  - Why? DMem read/write take place in same stage
- Data hazards through registers? Yes (previous slide)
  - Occur because register write is 3 stages after register read
  - Can only read a register value 3 cycles after writing it

Fixing Register Data Hazards

- Can only read register value 3 cycles after writing it

- One way to enforce this: make sure programs don't do it
  - Compiler puts two independent insns between write/read insn pair
    - If they aren't there already
  - Independent means: "do not interfere with register in question"
    - Do not write it: otherwise meaning of program changes
    - Do not read it: otherwise create new data hazard
  - **Code scheduling**: compiler moves around existing insns to do this
  - If none can be found, must use **nops**

  - This is called **software interlocks**
    - **MIPS**: **M**icroprocessor w/out **I**nterlocking **P**ipeline **S**tages

Software Interlock Example

```
add $3,$2,$1
lw $4,0($3)
sw $7,0($3)
add $6,$2,$8
addi $3,$5,4
```

- Can any of last three insns be scheduled between first two
  - `sw $7,0($3)`? No, creates hazard with `add $3,$2,$1`
  - `add $6,$2,$8`? OK
  - `addi $3,$5,4`? No, `lw` would read `$3` from it
  - Still need one more insn, use `nop`

```
add $3,$2,$1
add $6,$2,$8
nop
lw $4,0($3)
sw $7,0($3)
addi $3,$5,4
```

Software Interlock Performance

- Same deal
  - Branch: 20%, load: 20%, store: 10%, other: 50%

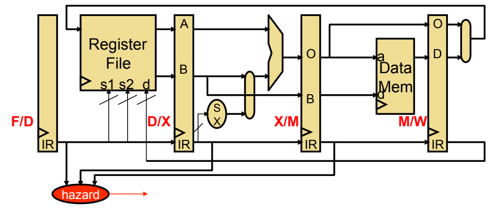
- Software interlocks
  - 20% of insns require insertion of 1 `nop`
  - 5% of insns require insertion of 2 `nops`

  - CPI is still 1 technically
  - But now there are more insns
  - #insns = 1 + 0.20*1 + 0.05*2 = **1.3**
  – **30% more insns (30% slowdown) due to data hazards**

Hardware Interlocks

- Problem with software interlocks? Not compatible
  - Where does **3** in "read register 3 cycles after writing" come from?
    - From structure (depth) of pipeline
  - What if next MIPS version uses a 7 stage pipeline?
    - Programs compiled assuming 5 stage pipeline will break
- A better (more compatible) way: **hardware interlocks**
  - Processor detects data hazards and fixes them
  - Two aspects to this
    - Detecting hazards
    - Fixing hazards

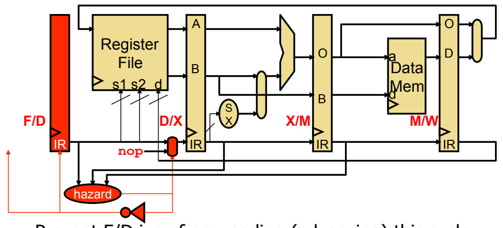
Detecting Data Hazards



- Compare F/D insn input register names with output register names of older insns in pipeline

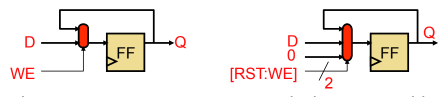
Hazard =
(F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||
(F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)

Fixing Data Hazards



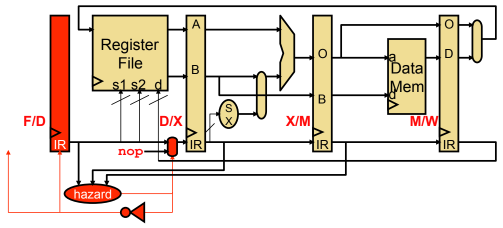
- Prevent F/D insn from reading (advancing) this cycle
  - Write `nop` into D/X.IR (effectively, insert `nop` in hardware)
  - Also reset (clear) the datapath control signals
  - Disable F/D latch and PC write enables (why?)
- Re-evaluate situation next cycle

Aside: Insert NOP/Reset Register



- Earlier: registers support separate clock, write enable
  - Useful for writes into register file
  - Also useful for implementing stalls
- Registers should also support **synchronous reset** (clear)
  - Useful for implementing stalls
  - Implement as additional hardwired 0 input to FF data mux
  - Resetting pipeline registers equivalent to inserting a NOP
    - If NOP is all zeros
    - If zero means "don't write" for all write-enable control signals
    - Design ISA/control signals to make sure this is the case

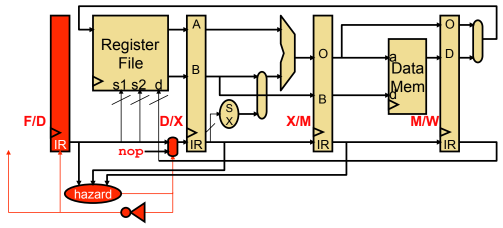
Hardware Interlock Example: cycle 1



`lw $4,0($3)` `add $3,$2,$1`

(F/D.IR.RS1 == D/X.IR.RD) || (**F/D.IR.RS2 == D/X.IR.RD**) ||
(F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)
= **1**

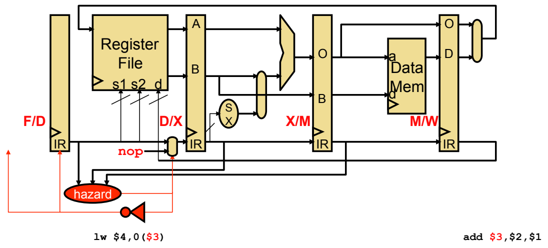
Hardware Interlock Example: cycle 2



`lw $4,0($3)` `add $3,$2,$1`

(F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||
(F/D.IR.RS1 == X/M.IR.RD) || (**F/D.IR.RS2 == X/M.IR.RD**)
= **1**

Hardware Interlock Example: cycle 3



(F/D.IR.RS1 == D/X.IR.RD) || (F/D.IR.RS2 == D/X.IR.RD) ||
(F/D.IR.RS1 == X/M.IR.RD) || (F/D.IR.RS2 == X/M.IR.RD)
= **0**

Pipeline Control Terminology

- Hardware interlock maneuver is called **stall** or **bubble**
- Mechanism is called **stall logic**
- Part of more general **pipeline control** mechanism
  - Controls advancement of insns through pipeline
- Distinguish from **pipelined datapath control**
  - Controls datapath at each stage
  - Pipeline control controls advancement of datapath control

Pipeline Diagram with Data Hazards

- Data hazard stall indicated with **d***
  - Stall propagates to younger insns

	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($3)`		F	d*	d*	D	X	M	W	
`sw $6,4($7)`					F	D	X	M	W

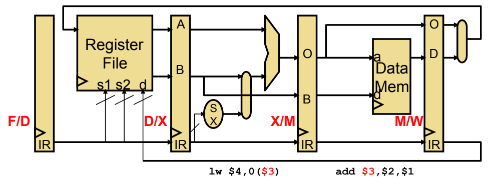
- This is not good (why?)

	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($3)`		F	d*	d*	D	X	M	W	
`sw $6,4($7)`			F	D	X	M	W		

Hardware Interlock Performance

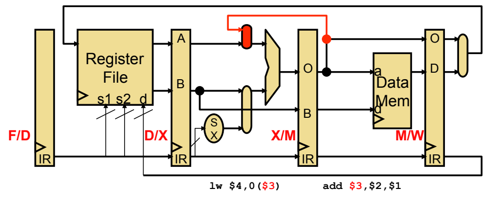
- Same deal
  - Branch: 20%, load: 20%, store: 10%, other: 50%
- Hardware interlocks: same as software interlocks
  - 20% of insns require 1 cycle stall (I.e., insertion of 1 `nop`)
  - 5% of insns require 2 cycle stall (I.e., insertion of 2 `nops`)
  - CPI = 1 * 0.20*1 + 0.05*2 = **1.3**
  - So, either CPI stays at 1 and #insns increases 30% (software)
  - Or, #insns stays at 1 (relative) and CPI increases 30% (hardware)
  - Same difference
- Anyway, we can do better

Observe



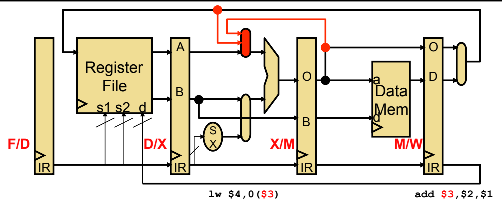
- Technically, this situation is broken
  - `lw $4,0($3)` has already read `$3` from regfile
  - `add $3,$2,$1` hasn't yet written `$3` to regfile
- But fundamentally, everything is OK
  - `lw $4,0($3)` hasn't actually used `$3` yet
  - `add $3,$2,$1` has already computed `$3`

Bypassing



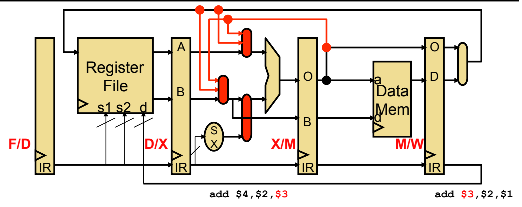
- **Bypassing**
  - Reading a value from an intermediate (μarchitectural) source
  - Not waiting until it is available from primary source
  - Here, we are bypassing the register file
  - Also called **forwarding**

WX Bypassing



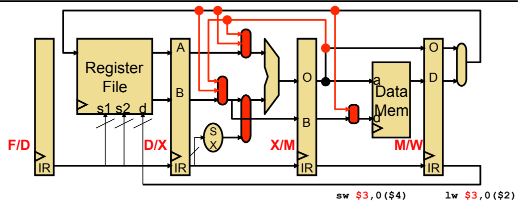
- What about this combination?
  - Add another bypass path and MUX input
  - First one was an **MX** bypass
  - This one is a **WX** bypass

ALUinB Bypassing



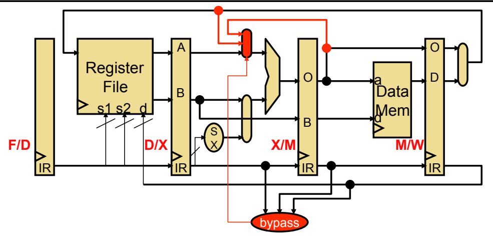
- Can also bypass to ALU input B

WM Bypassing?



- Does WM bypassing make sense?
  - Not to the address input (why not?)
  - But to the store data input, yes

Bypass Logic



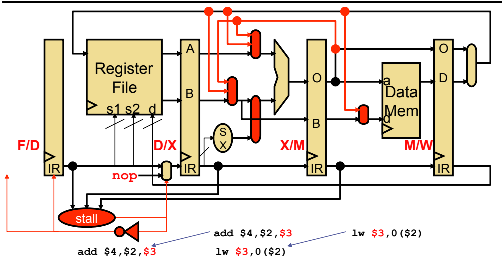
- Each MUX has its own, here it is for MUX ALUinA

  (D/X.IR.RS1 == X/M.IR.RD) => 0
  (D/X.IR.RS1 == M/W.IR.RD) => 1
  Else => 2

Bypass and Stall Logic

- Two separate things
  - Stall logic controls pipeline registers
  - Bypass logic controls MUXs
- But complementary
  - For a given data hazard: if can't bypass, must stall
- Slide #43 shows **full bypassing**: all bypasses possible
  - Is stall logic still necessary?

Yes, Load Output to ALU Input



Stall = (D/X.IR.OP == LOAD) &&
((F/D.IR.RS1 == D/X.IR.RD) ||
((F/D.IR.RS2 == D/X.IR.RD) && (F/D.IR.OP != STORE))

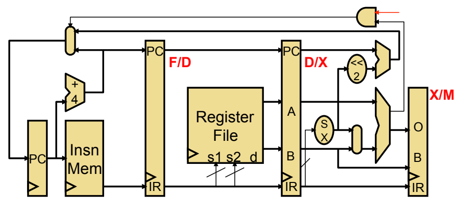
Pipeline Diagram With Bypassing

	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($3)`		F	D	X	M	W			
`addi $6,$4,1`			F	d*	D	X	M	W	

- Use compiler scheduling to reduce load-use stall frequency
  - Like software interlocks, but for performance not correctness

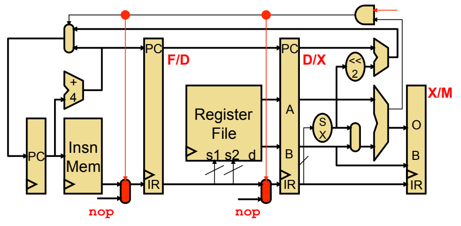
	1	2	3	4	5	6	7	8	9
`add $3,$2,$1`	F	D	X	M	W				
`lw $4,0($3)`		F	D	X	M	W			
`sub $8,$3,$1`			F	D	X	M	W		
`addi $6,$4,1`				F	D	X	M	W	

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)

Branch Recovery



- **Branch recovery**: what to do when branch is actually taken
  - Insns that will be written into F/D and D/X are wrong
  - **Flush them**, i.e., replace them with `nops`
  - + They haven't had written permanent state yet (regfile, DMem)

Branch Recovery Pipeline Diagram

		1	2	3	4	5	6	7	8	9
	`addi $3,$0,1`	F	D	X	M	W				
	`bnez $3,targ`		F	D	X	M	W			
	~~`sw $6,4($7)`~~			~~F~~	~~D~~					
`targ:`	~~`addi $8,$7,1`~~				~~F~~					
`targ:`	`addi $8,$7,1`					F	D	X	M	W

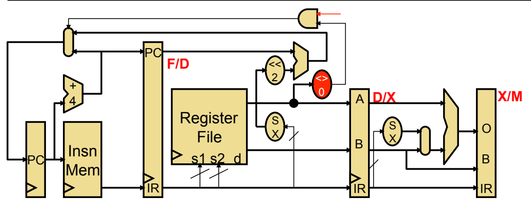
- Convention: don't fill in flushed insns
- Taken branch penalty is 2 cycles

		1	2	3	4	5	6	7	8	9
	`addi $3,$0,1`	F	D	X	M	W				
	`bnez $3,targ`		F	D	X	M	W			
`targ:`	`addi $8,$7,1`					F	D	X	M	W

Branch Performance

- Back of the envelope calculation
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%
  - **75% of branches are taken**
- CPI = 1 + **0.20*0.75*2** = 1.3
  - – **Branches cause 30% slowdown**
  - How do we reduce this penalty?

Fast Branch



- **Fast branch**: can decide at D, not X
  - Test must be comparison to zero or equality, no time for ALU
  - + New taken branch penalty is 1
  - – Additional insns (`slt`) for more complex tests, must bypass to D too
  - 25% of branches have complex tests that require extra insn
  - CPI = 1 + 0.20*0.75***1**(branch) + 0.20*0.25*1(extra insn) = **1.2**

Speculative Execution

- Speculation: "risky transactions on 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 insns
    – Must undo incorrect changes (recover pre-speculation state)
  - The "game": **$[\%_{correct} * \text{gain}] - [(1-\%_{correct}) * \text{penalty}]$**

- **Control speculation**: speculation aimed at control hazards
  - Unknown parameter: are these the correct insns to execute next?

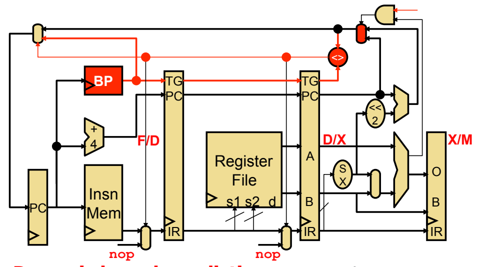
Control Speculation Mechanics

- Guess branch target, start fetching at guessed position
  - Doing nothing is implicitly guessing target is PC+4
  - Can actively guess other targets: **dynamic branch prediction**

- Execute branch to verify (check) guess
  - Correct speculation? keep going
  - Mis-speculation? Flush mis-speculated insns
    - Hopefully haven't modified permanent state (Regfile, DMem)
    + Happens naturally in in-order 5-stage pipeline

- "Game" for in-order 5 stage pipeline
  - $\%_{correct}$ = ?
  - Gain = 2 cycles
  + Penalty = 0 cycles → **mis-speculation no worse than stalling**

Dynamic Branch Prediction



- **Dynamic branch prediction**: guess outcome
  - Start fetching from guessed address
  - Flush on **mis-prediction** (notice new recovery circuit)

Branch Prediction: Short Summary

- Key principle of micro-architecture:
  - Programs do the same thing over and over (why?)

- Exploit for performance:
  - Learn what a program did before
  - Guess that it will do the same thing again

- Quick intro to details

Branch Prediction 10K feet

- Two (separate) tasks:
  - Predict taken/not taken
  - Predict taken target

Branch Prediction 10K feet

- Two (separate) tasks:
  - Predict taken/not taken
  - Predict taken target

- High level solution (both tasks):
  - SRAM "array" to remember most recent behaviors
  - Kind of like a cache, indexed by PC bits, but different
    - Typically no next level (but can have 2 levels)
    - Can skip tag, or use partial tag
      - Predictor: OK to be wrong (as long as we fix it)

Branch Target Buffer (BTB)

- Branch Target Buffer
  - SRAM array, holds recent taken targets
  - Example: 4K entries, direct mapped
  - Can be set-associative
  - Each entry holds partial PC (low order bits)
    - Assume high bits unchanged (why?)
    - Example: 16 bits

0	01F3
1	4242
2	1234
	…….
	…….
4097	4242

Branch Target Buffer (BTB)

- Branch Target Buffer
  - SRAM array, holds recent taken targets
  - Example: 4K entries, direct mapped
  - Can be set-associative
  - Each entry holds partial PC (low order bits)
    - Assume high bits unchanged (why?)
    - Example: 16 bits
- Prediction of taken target:
  - Use PC bits 2—13 to index BTB (why these bits?)
  - Replace PC bits 2—17 with value in BTB

0	01F3
1	4242
2	1234
	…….
	…….
4097	4242

Branch Target Buffer (BTB)

- Branch Target Buffer
  - SRAM array, holds recent taken targets
  - Example: 4K entries, direct mapped
  - Can be set-associative
  - Each entry holds partial PC (low order bits)
    - Assume high bits unchanged (why?)
    - Example: 16 bits
- Prediction of taken target:
  - Use PC bits 2—13 to index BTB (why these bits?)
  - Replace PC bits 2—17 with value in BTB
- Update (how do values get into predictor?)
  - At execute, if branch is taken write target into BTB
  - Use PC bits 2—13 to index for write also (same entry)

0	01F3
1	4242
2	1234
	…….
	…….
4097	4242

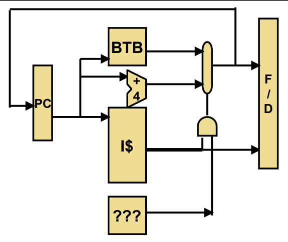
Target Prediction: BTB collisions

- PCs may collide in BTB
  - Example: 0x10000000 and 0x20000000 (both index 0)
  - Could use tags (or partial tags)
    - Better to just guess "not taken" than "taken to bogus target"
      - Why?

Target Prediction: BTB collisions

- PCs may collide in BTB
  - Example: 0x10000000 and 0x20000000 (both index 0)
  - Could use tags (or partial tags)
    - Better to just guess "not taken" than "taken to bogus target"
      - Why?
  - What if 0x10000000 is a branch, and 0x20000000 is not?
    - Pipeline may predict bogus next PC for non-branch
      - Fine as long as detected/fixed (extra checking)
      - Usually checked in decode if possible
    - Alternative: pre-decode bits
      - Add bits in I$ to say "is this a branch"
      - Know if not a branch while predicting
      - Bits set on I$ fill path (examine bits coming from L2)

Our branch predictor (so far)



- Missing piece (???): Direction predictor
  - Should we use the taken target (from BTB) or not?

Direction Prediction

- Need to predict "taken" (T) or "not taken" (N)
  - This is typically the hard part, by the way
- Simplest approach: just guess "same as last time"
  - Actually, kind of not bad:
    - Loops: almost always right (taken)
    - Error checks: almost always right (no error)
    - …etc..
- Implementation:
  - SRAM, indexed by PC bits
  - 1 bit per entry: 1 = taken, 0 = not taken
  - No tags.
  - Collisions? Meh—they happen

Direction Prediction: Example

- Consider:

```
for (int i = 0; I < 10000000; i++) {
 for (int j = 0; j < 6; j++) {
   //stuff
 }
}
Branches outcomes:
 TTTTTNTTTTTTNTTTTTTNTTTTTTNTTTTTTNTTTTTTNT…
```

Direction Prediction: Example

- Consider:

```
for (int i = 0; I < 10000000; i++) {
 for (int j = 0; j < 6; j++) {
   //stuff
 }
}
Branches outcomes:
 TTTTTNTTTTTTNTTTTTTNTTTTTTNTTTTTTNTTTTTTNT…
Predictions:
 NTTTTTTNTTTTTTNTTTTTTNTTTTTTNTTTTTTNTTTTTT…
```

Direction Prediction: Can we do better?

```
Branches outcomes:
 TTTTTNTTTTTTNTTTTTTNTTTTTTNTTTTTTNTTTTTTNT…
Predictions:
 NTTTTTTNTTTTTTNTTTTTTNTTTTTTNTTTTTTNTTTTTT…
```

- Problem:
  - A little too quick to react
  - One-off difference causes **two** mis-predictions
- Solution:
  - Slow down changes in prediction: 2-bit counters
  - T (11), t (10), n (00), N (01)
  - "Strongly" (T/N) and "weakly" (t/n) taken/not taken
  - Updates: taken-> increment, not taken -> decrement

Direction Prediction: Can we do better?

```
Branches outcomes:
 TTTTTNTTTTTTNTTTTTTNTTTTTTNTTTTTTNTTTTTTNT…
Predictions:
 NTTTTTTNTTTTTTNTTTTTTNTTTTTTNTTTTTTNTTTTTT…
 tTTTTTTtTTTTTTtTTTTTTtTTTTTTtTTTTTTtTTTTTT…
```

- Problem:
  - A little too quick to react
  - One-off difference causes **two** mis-predictions
- Solution:
  - Slow down changes in prediction: 2-bit counters
  - T (11), t (10), n (00), N (01)
  - "Strongly" (T/N) and "weakly" (t/n) taken/not taken
  - Updates: taken-> increment, not taken -> decrement

Can we do even better still?

- Our branches have a very regular pattern
  - 6Ts, then 1 N
  - We really should be able to get them all right… right?
- Real predictors use **history**
  - Take recent branch outcomes (NTTTTTT = 0111111)
  - XOR with PC to form table index
  - Same PC, different history -> different index -> different counter
  - Would predict previous example perfectly
- Also useful for correlation of branches
  - Nearby branches with related outcomes (why is this common?)

Direction Prediction: Continued..

- Real direction predictors more complex even still
  - Multiple tables with choosers (hybrid history schemes)
- Research ideas too
  - Late 90s/early 2000s: think up bpred idea, publish, repeat
- Big impediment to performance/hard to get well
- Also research ideas for how to get around it
  - Control Independence: predicting reconvergence point easier

Predicting returns

- Previous things don't work well on "return" instructions
  - jr $ra
  - Why not?

Predicting returns

- Previous things don't work well on "return" instructions
  - jr $ra
  - Why not?
  - Functions called from many places
    - Previous place to return to, not always current place to return to…
  - But should be predictable: why?

Predicting returns

- Previous things don't work well on "return" instructions
  - jr $ra
  - Why not?
  - Functions called from many places
    - Previous place to return to, not always current place to return to…
  - But should be predictable: why?
    - Matches up with jal's PC +4
    - In stack-like fashion
  - So….

Predicting returns

- Previous things don't work well on "return" instructions
  - jr $ra
  - Why not?
  - Functions called from many places
    - Previous place to return to, not always current place to return to…
  - But should be predictable: why?
    - Matches up with jal's PC +4
    - In stack-like fashion
  - So….
- "Return Address Stack" (aka "Link Stack")
  - Predictor tracks a stack of recent jals
  - Encounter a jr $ra? Pop stack for predicted target

Branch Prediction Performance

- Dynamic branch prediction
  - Simple predictor: branches predicted with 75% accuracy
  - CPI = 1 + 0.20***0.25***2 = **1.1**
  - More advanced predictor: 95% accuracy
  - CPI = 1 + 0.20***0.05***2 = **1.02**
- Branch mis-predictions still a big problem though
  - Pipelines are long: typical mis-prediction penalty is 10+ cycles
  - Pipelines have full bypassing: compiler schedules the rest
  - Pipelines are superscalar (later)

Pipelining And Exceptions

- Pipelining makes exceptions more complex
  - 5 insns in pipeline at once
  - Exception happens, how do you know which insn caused it?
    - Exceptions propagate along pipeline in latches
  - Two exceptions happen, how do you know which one to take first?
    - One belonging to oldest insn
  - When handling exception, have to flush younger insns
    - Piggy-back on branch mis-prediction machinery to do this
  - What about multi-cycle operations?

- Just FYI

Pipeline Depth

- No magic about 5 stages, trend had been to deeper pipelines
  - 486: 5 stages (50+ gate delays / clock)
  - Pentium: 7 stages
  - Pentium II/III: 12 stages
  - Pentium 4: 22 stages (~10 gate delays / clock) **"super-pipelining"**
  - Core1/2: 14 stages

- Increasing **pipeline depth**
  + Increases clock frequency (reduces period)
  – But decreases IPC (increases CPI)
  - Branch mis-prediction penalty becomes longer
  - Non-bypassed data hazard stalls become longer
  - At some point, CPI losses offset clock gains, question is when?
    - 1GHz Pentium 4 was slower than 800 MHz PentiumIII
  - What was the point? People by frequency, not frequency * IPC

Real pipelines…

- Real pipelines fancier than what we have seen
  - Superscalar: multiple instructions in a stage at once
  - Out-of-order: re-order instructions to reduce stalls
  - SMT: execute multiple threads at once on processor
    - Side by side, sharing pipeline resources
  - Multi-core: multiple pipelines on chip
    - Cache coherence: No stale data

- Learn more?
  - Take ECE 552 next Fall!