CSE 530
Fundamentals of Computer Architecture
Spring 2021

Pipelined Datapaths

John (Jack) Sampson

Course material on CANVAS: canvas.psu.edu

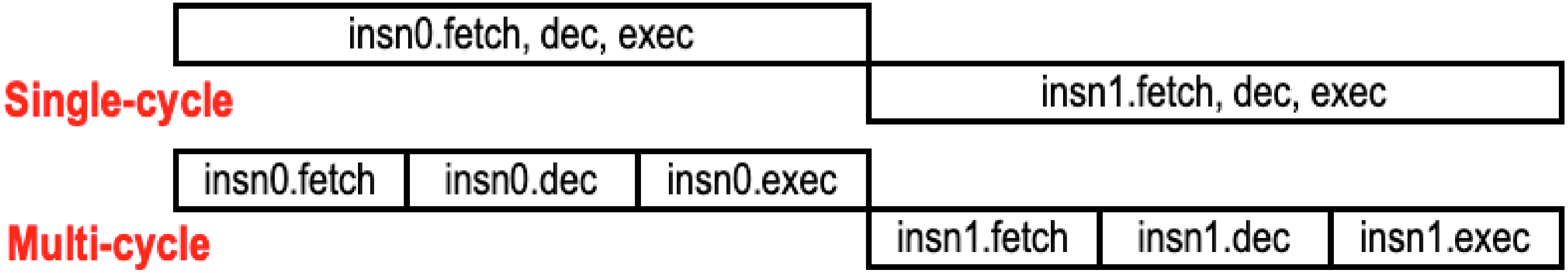
[Adapted in part from Mary Jane Irwin, V. Narayanan, A. Kolli @PSU, and includes materials originally developed by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, and Wenisch of Carnegie Mellon University, Purdue University, University of Michigan, University of Pennsylvania, and University of Wisconsin. Material flow organized in part around *Computer Architecture: A Quantitative Approach* by Hennessey and Patterson]

Scalar, in-order pipelined datapath design

Pipelining is *the first key* implementation technique to make fast (and, *usually*, resource-efficient) CPUs

- Principles of pipelining
  - Design goals
  - Pipeline diagrams
  - Effects of overhead and hazards
- Dealing with/overcoming data hazards
  - Stalling and bypassing
- Dealing with/overcoming control hazards
  - Branch prediction

Before there was pipelining…



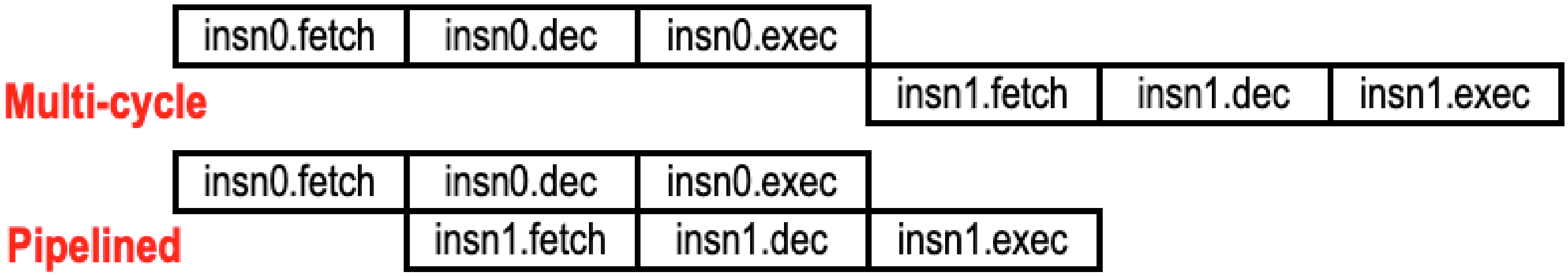
Basic **datapath**: fetch, decode, execute

- **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



Multi-cycle

insn0.fetch | insn0.dec | insn0.exec

insn1.fetch | insn1.dec | insn1.exec

Pipelined

insn0.fetch | insn0.dec | insn0.exec

insn1.fetch | insn1.dec | insn1.exec

- Important performance technique
  - **Improves throughput at the expense of latency**
    - *Why does latency go up?*

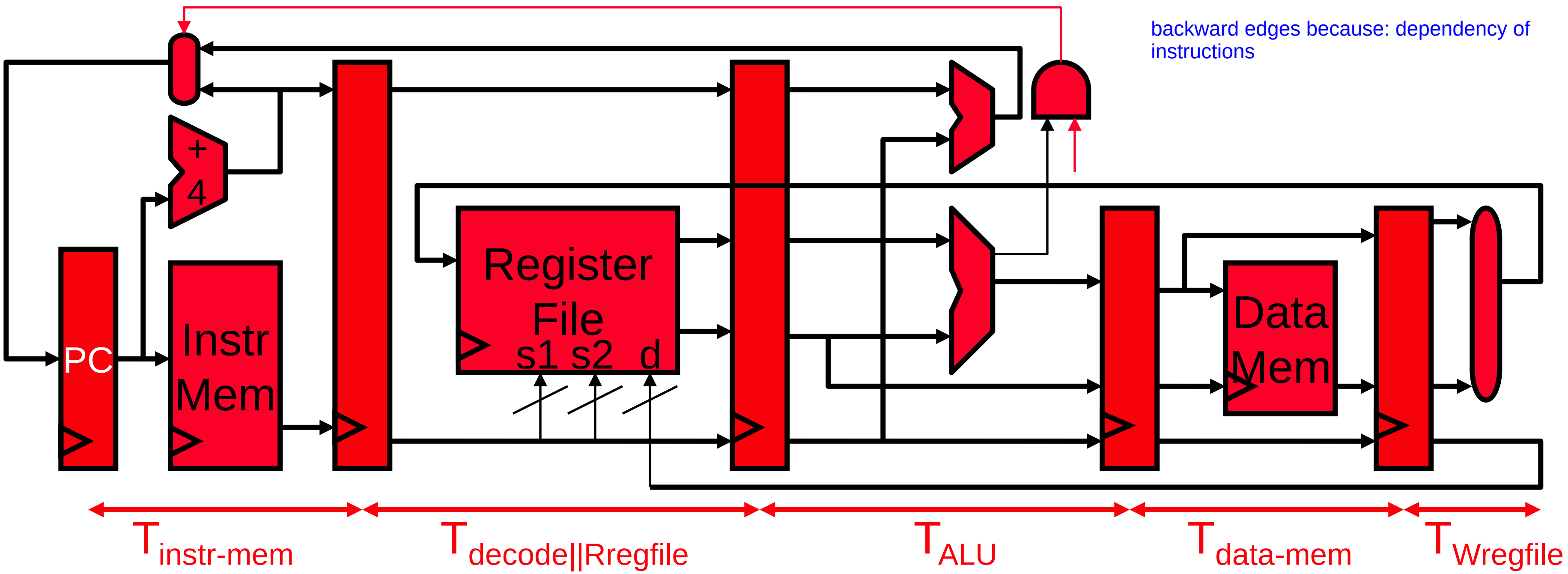
usually slightly increases the execution time of each instruction due to overhead in the pipeline control.

- Begin with multi-cycle design
  - When instruction advances from stage 1 to 2…

    … allow next instruction to enter stage 1
  - Each instruction still passes through all stages
  - + **But instructions enter and leave at a much faster rate**

instruction throughput much higher

Five stage RISC pipelined (in order) datapath



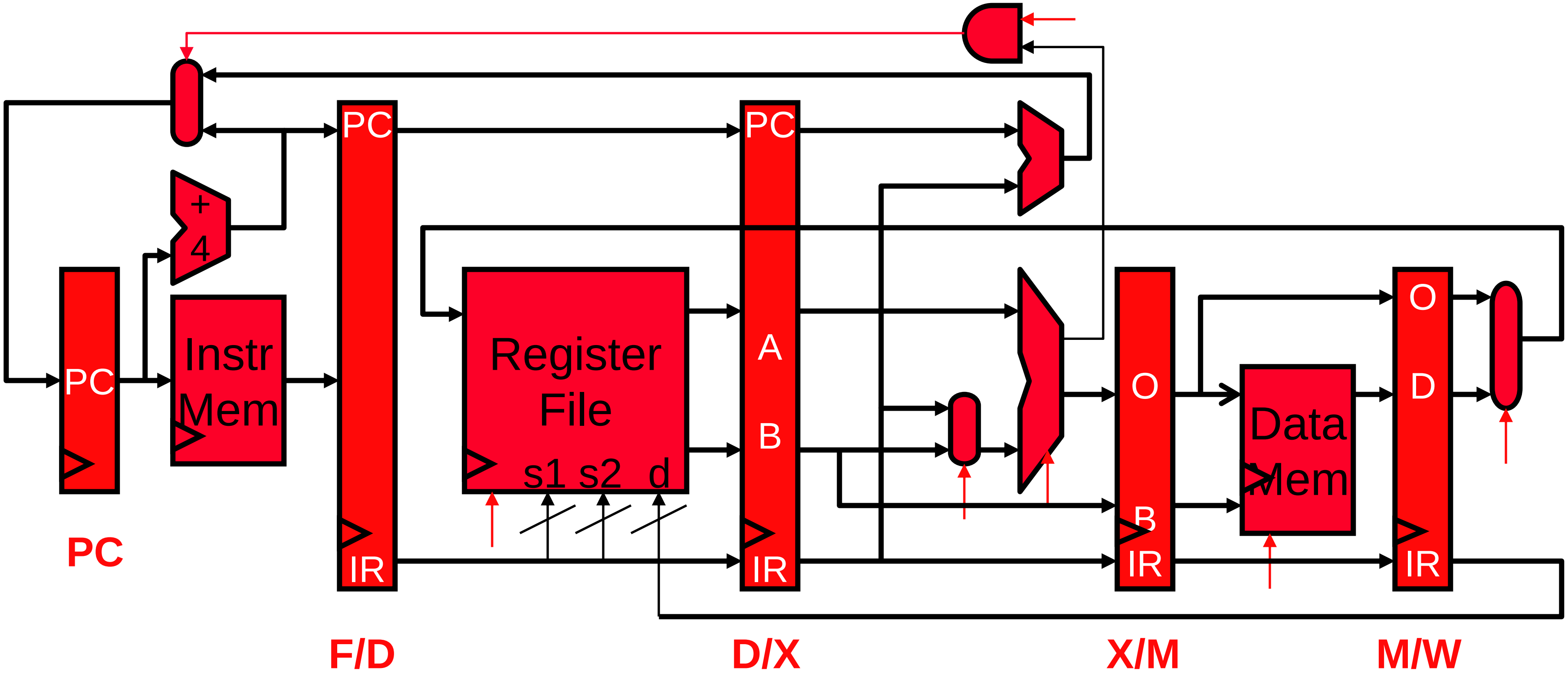
backward edges because: dependency of instructions

- **Pipelining**: cut datapath into N stages (here five)
  - One instr in each stage in each cycle
  - + Clock period = MAX($T_{\text{instr-mem}}$, T_{Rfegfile}, T_{ALU}, $T_{\text{data-mem}}$,T_{Wregfile})
  - + Base CPI = 1: an instr enters and leaves every cycle
    - – Actual CPI > 1: pipeline must often stall

which is why latency increases!

  - Instr latency improvement is not the goal, increased throughput is!

Pipeline terminology



- Five stages: **F**etch, **D**ecode, e**X**ecute, **M**emory, **W**riteback
- Latches (pipeline registers) named by stages they separate
  - Each instr's temp values (PC,IR,A,B,O,D) re-latched every stage
    - Notice, PC not latched after X (ALU) stage (not needed later)
    - Control signals themselves (in red, determined during D) are pipelined

Stage 1: Fetch

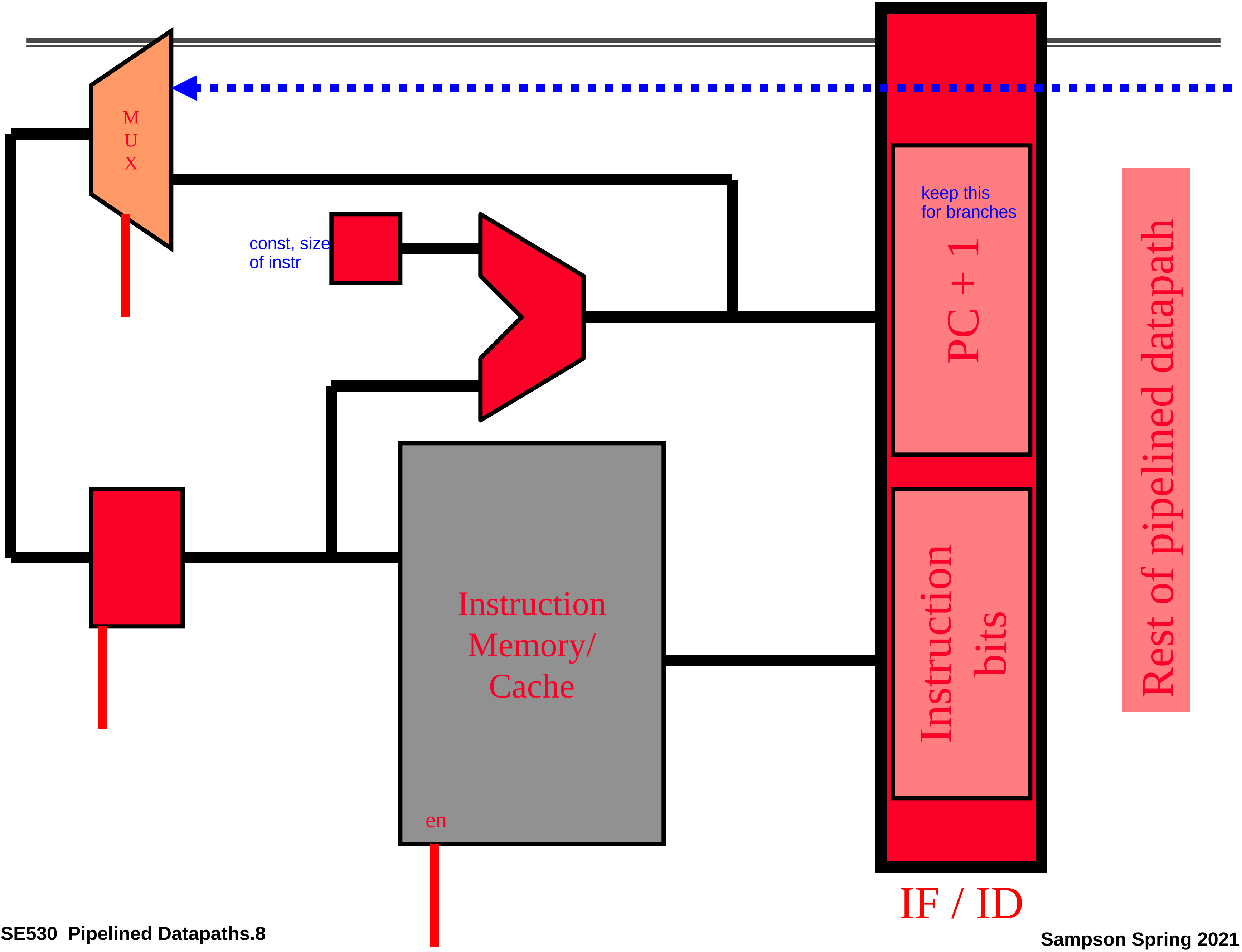
Fetch an instruction from memory every cycle.

- Use PC to index memory
- Increment PC (assume no branches for now)

Write state to the pipeline register (IF/ID)

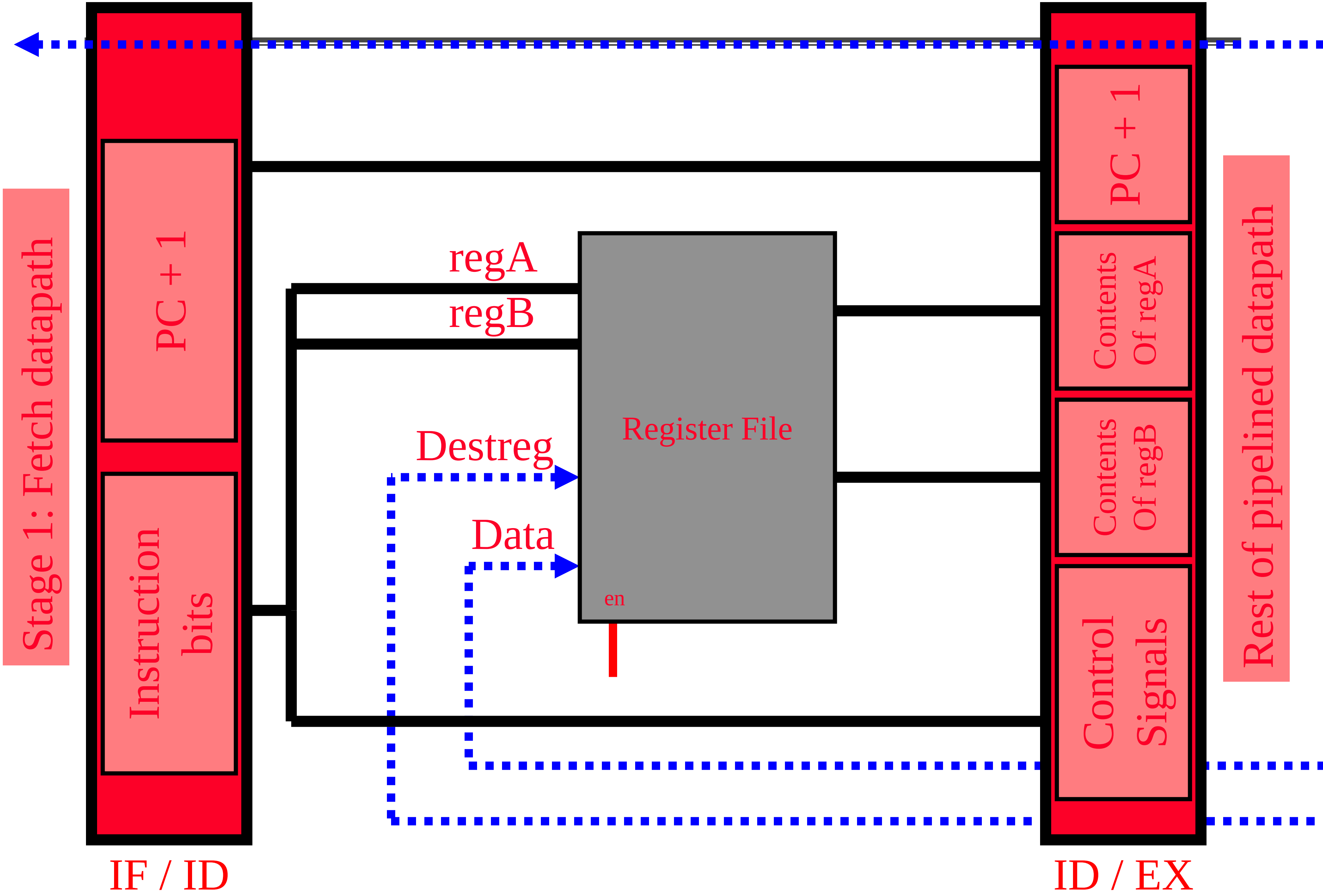
- The next stage will read this pipeline register.
- Note that pipeline register must be edge triggered

The register file in the pipelined processor writes on the falling edge of CLK so that it can write a result in the first half of a cycle and read that result in the second half of the cycle for use in a subsequent instruction.


M
U
X
const, size
of instr
keep this
for branches
PC + 1
Instruction
Memory/
Cache
en
Instruction
bits
Rest of pipelined datapath
IF / ID

Stage 2: Decode

- Decodes opcode bits
  - May set up control signals for later stages
- Read input operands from registers file
  - specified by regA and regB of instruction bits
- Write state to the pipeline register (ID/EX)
  - Opcode
  - Register contents
  - Offset & destination fields
  - PC+1 (even though decode didn’t use it)



Stage 1: Fetch datapath
PC + 1
Instruction bits
regA
regB
Register File
Destreg
Data
en
PC + 1
Contents Of regA
Contents Of regB
Control Signals
Rest of pipelined datapath
IF / ID
ID / EX

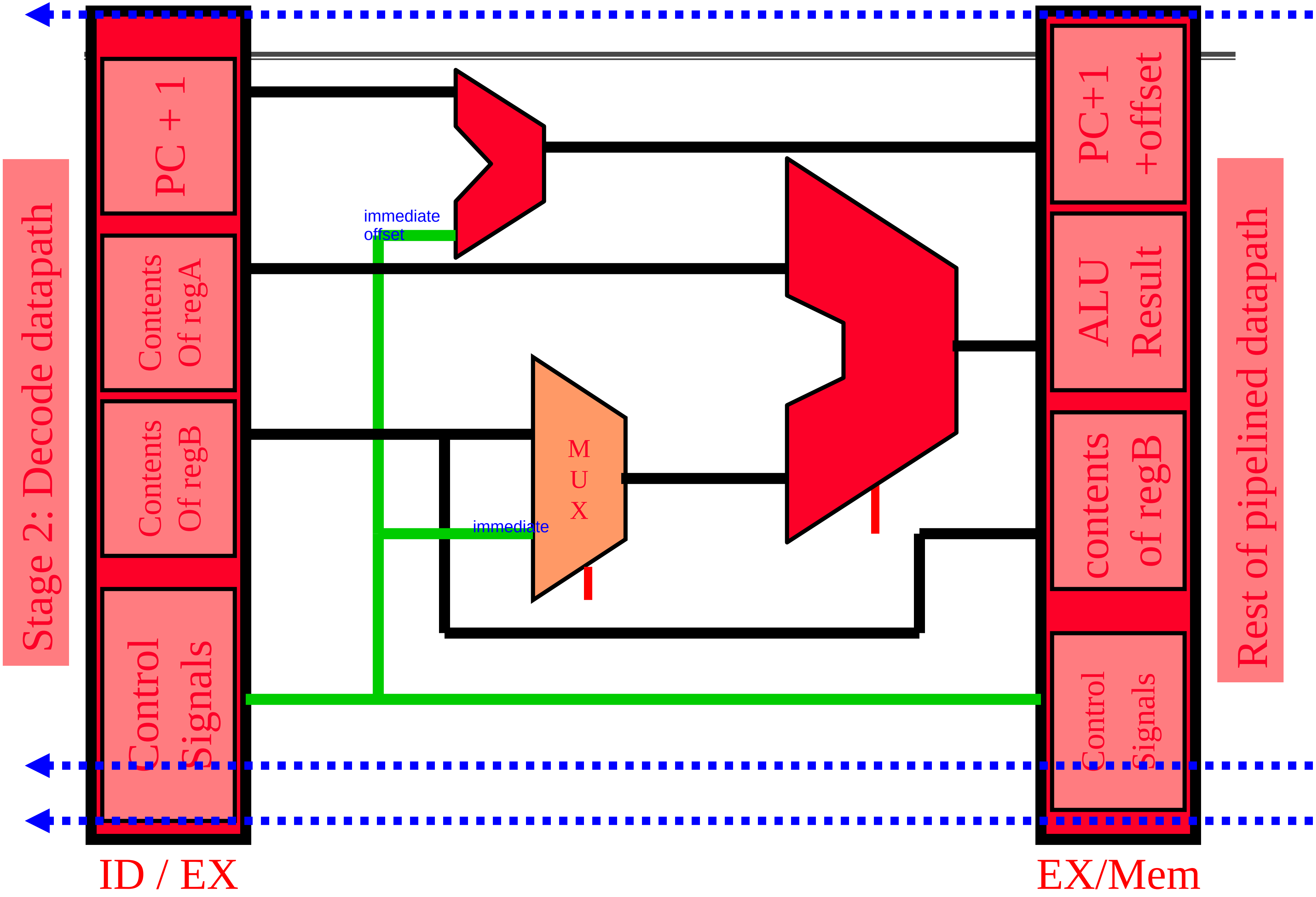
Stage 3: Execute

Perform ALU operation.

- Input operands can be:
  - Contents of regA or RegB
  - Offset field on the instruction for immediate ioerations
- Branches: calculate PC+1+offset

Write state to the pipeline register (EX/Mem)

- ALU result, contents of RegB and PC+1+offset
- Instruction bits for opcode and destReg specifiers



Stage 2: Decode datapath

PC + 1

Contents Of regA

Contents Of regB

Control Signals

immediate offset

immediate

M U X

PC+1 +offset

ALU Result

contents of regB

Control Signals

Rest of pipelined datapath

ID / EX

EX/Mem

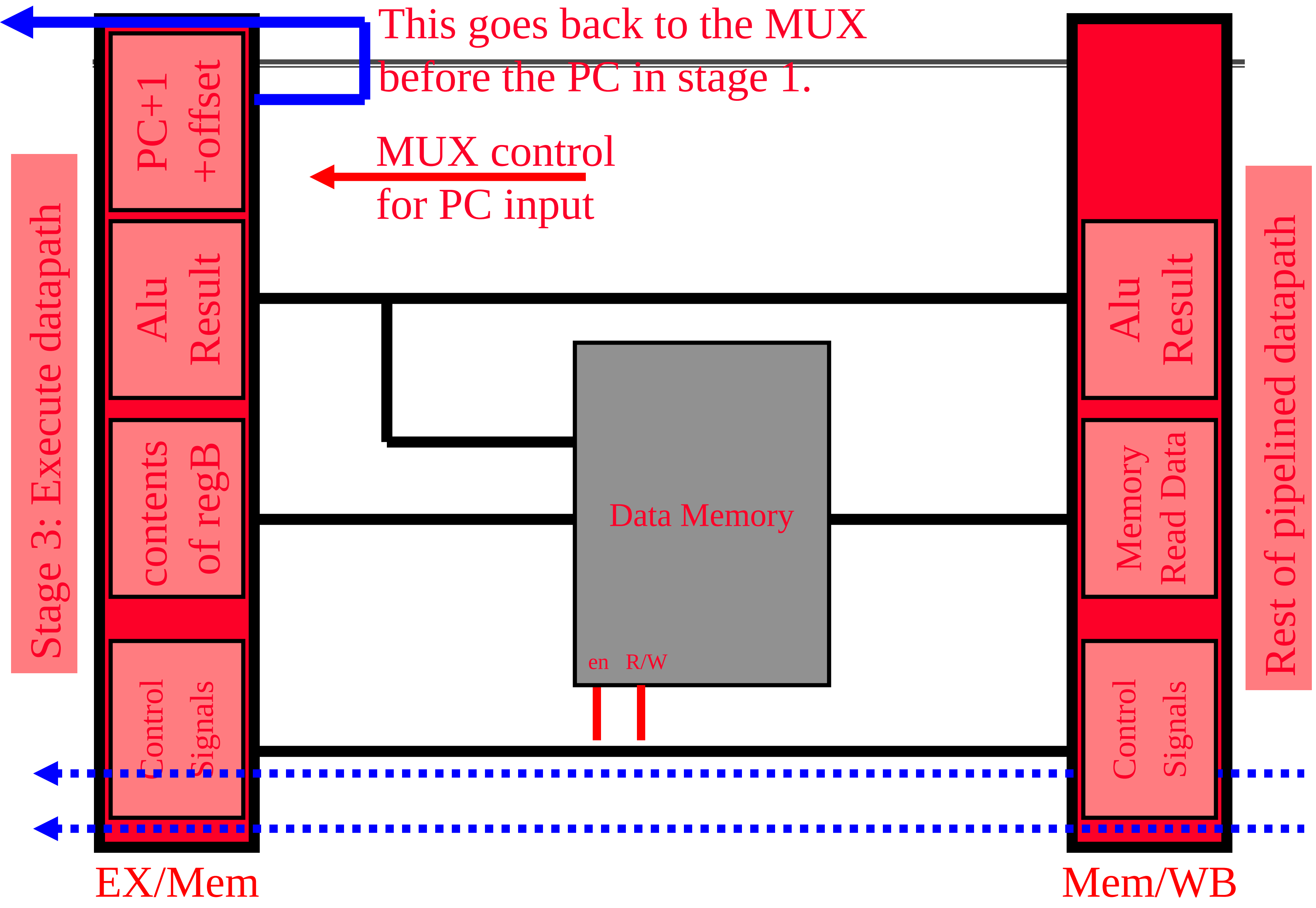
Stage 4: Memory Operation

Perform data cache access for memory ops

- ALU result contains address for **ld** and **st**
- Opcode bits control mem R/W and enable signals

Write state to the pipeline register (Mem/WB)

- ALU result and MemData
- Instruction bits for opcode and destReg specifiers

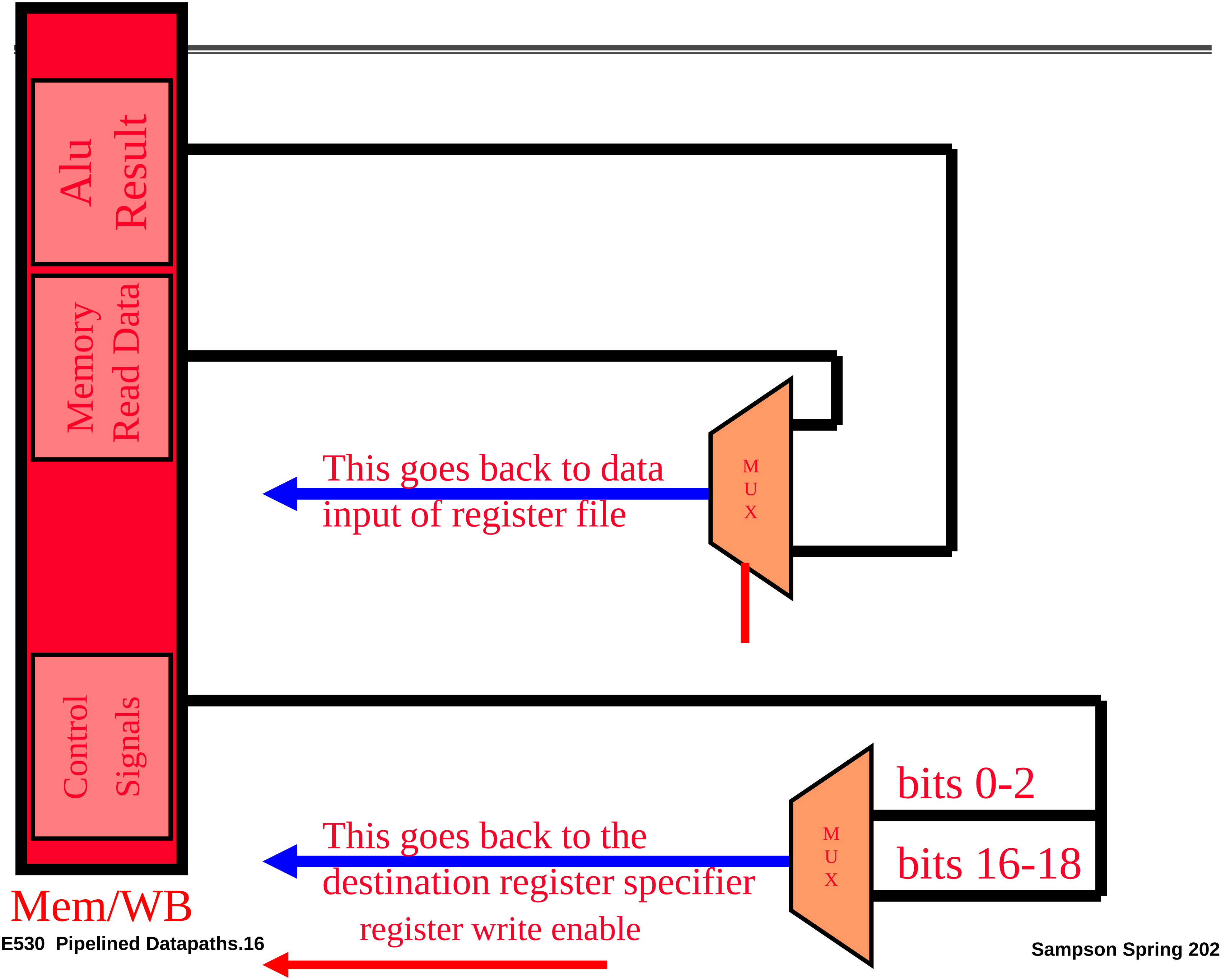

This goes back to the MUX
before the PC in stage 1.
MUX control
for PC input
PC+1
+offset
Alu
Result
contents
of regB
Control
Signals
Stage 3: Execute datapath
Data Memory
en
R/W
Alu
Result
Memory
Read Data
Control
Signals
Rest of pipelined datapath
EX/Mem
Mem/WB

Stage 5: Write back

Writing result to register file (if required)

- Write MemData to destReg for ld instruction
- Write ALU result to destReg for arithmetic instruction
- Opcode bits control register write enable signal

Stage 4: Memory datapath

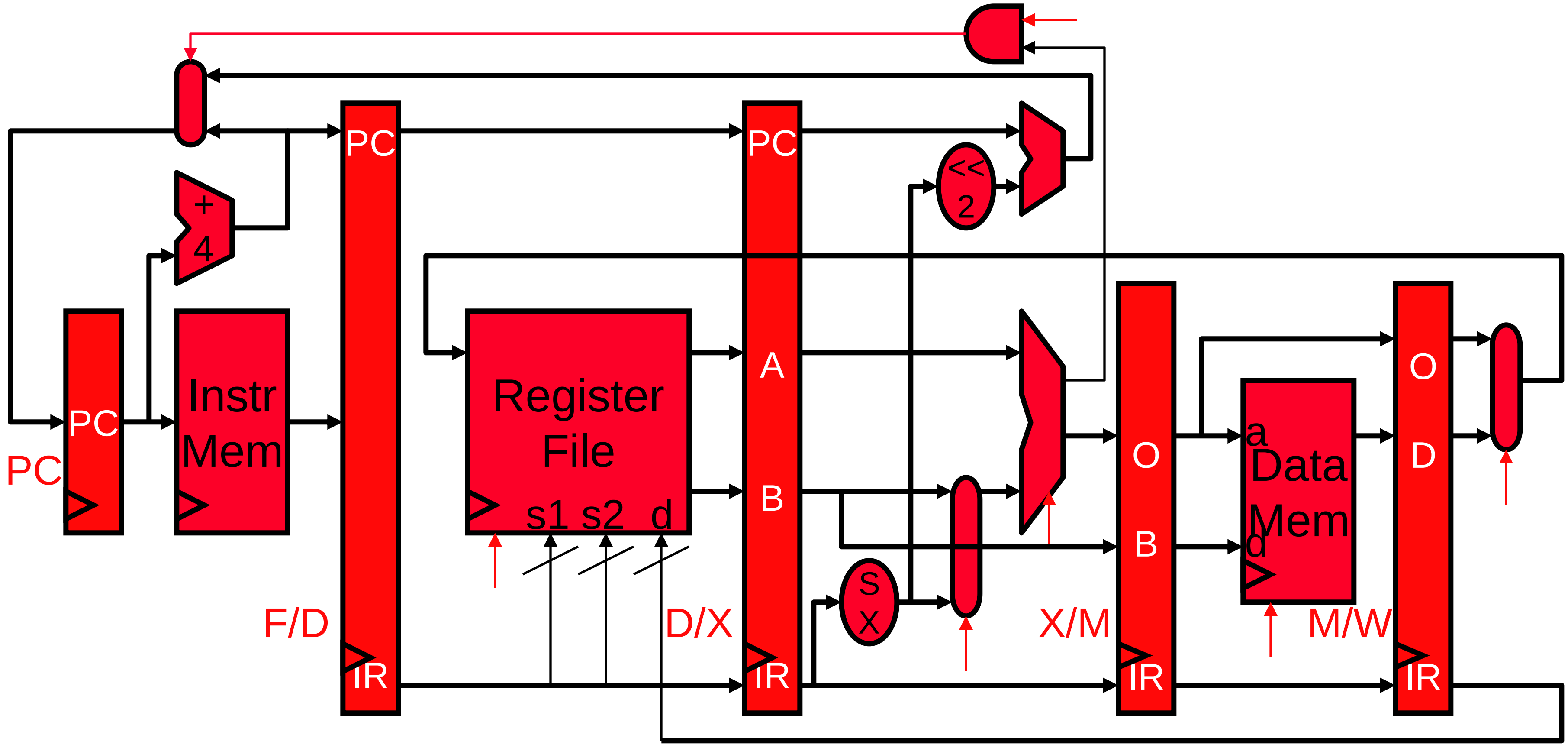


This goes back to data input of register file

This goes back to the destination register specifier

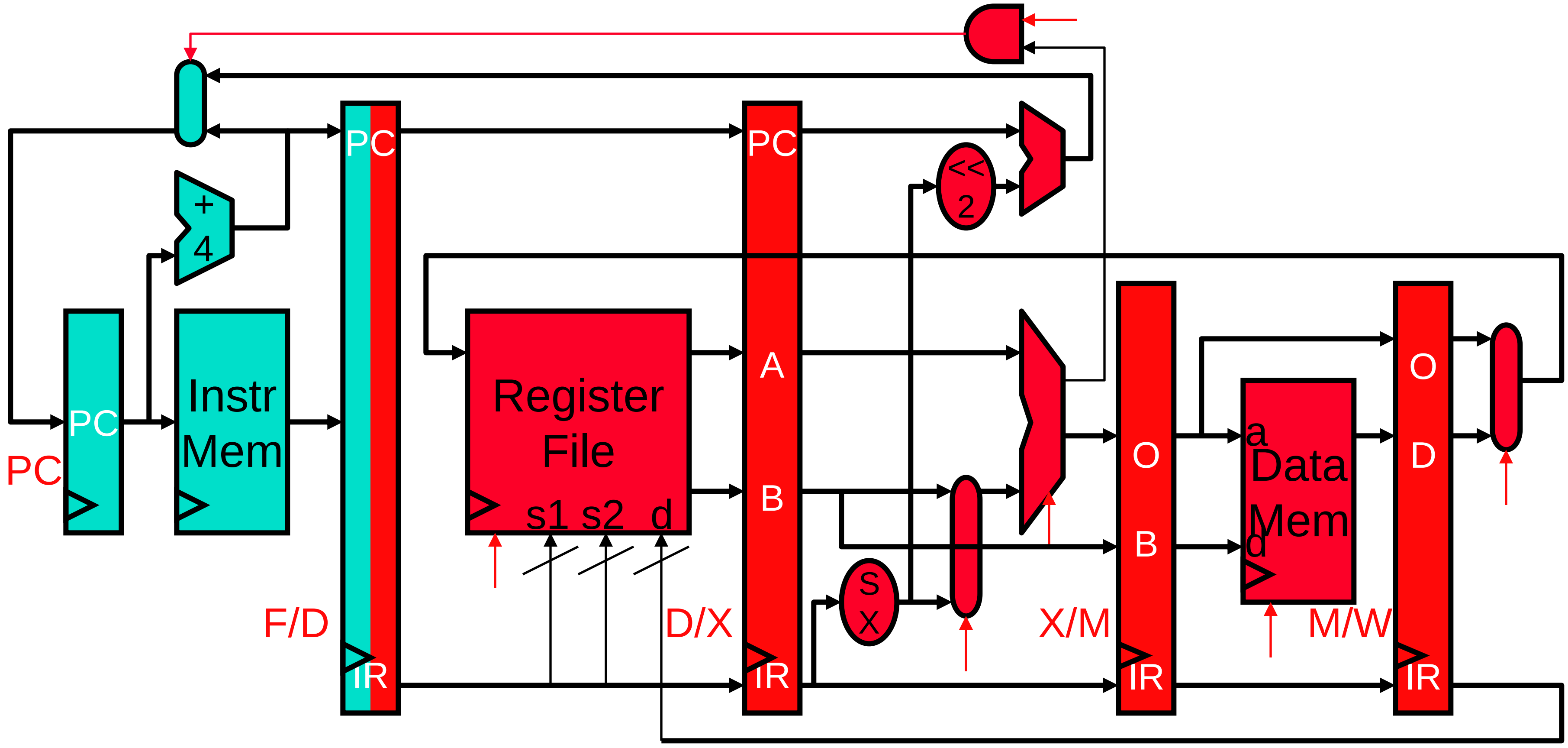
register write enable

Pipeline example: Cycle 1



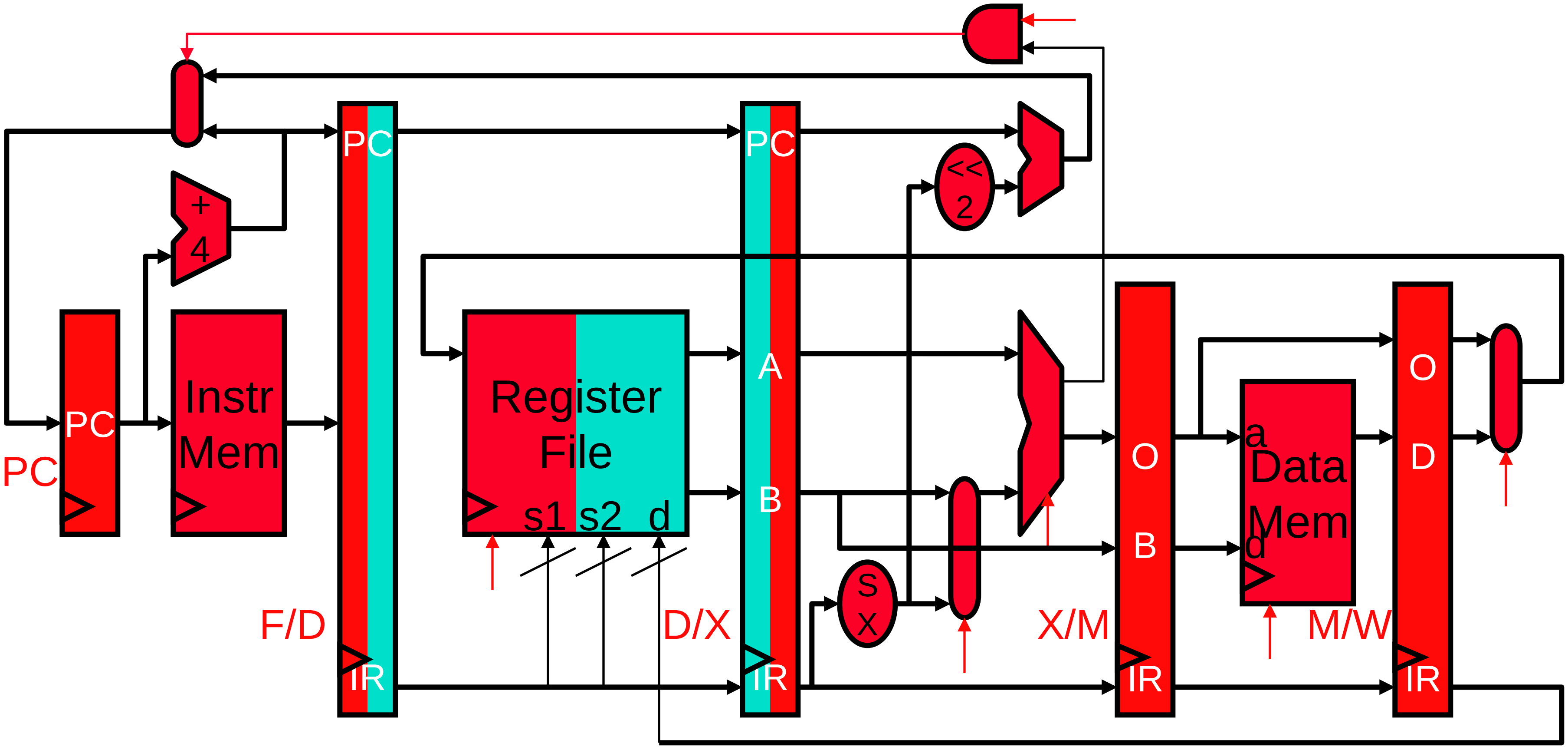
add $3,$2,$1

Pipeline example: Cycle 2



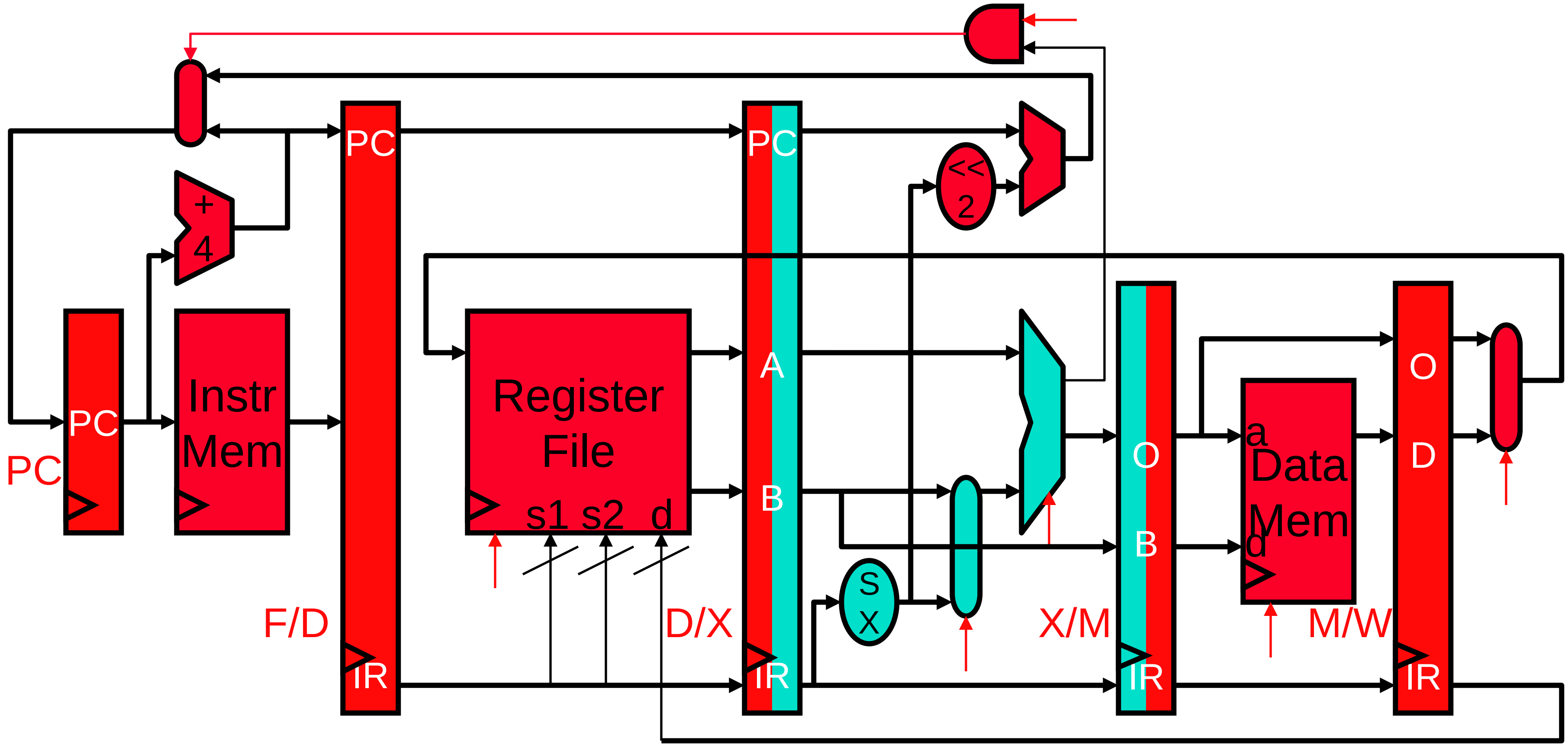
`lw $4,0($5)`   `add $3,$2,$1`

Pipeline example: Cycle 3



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

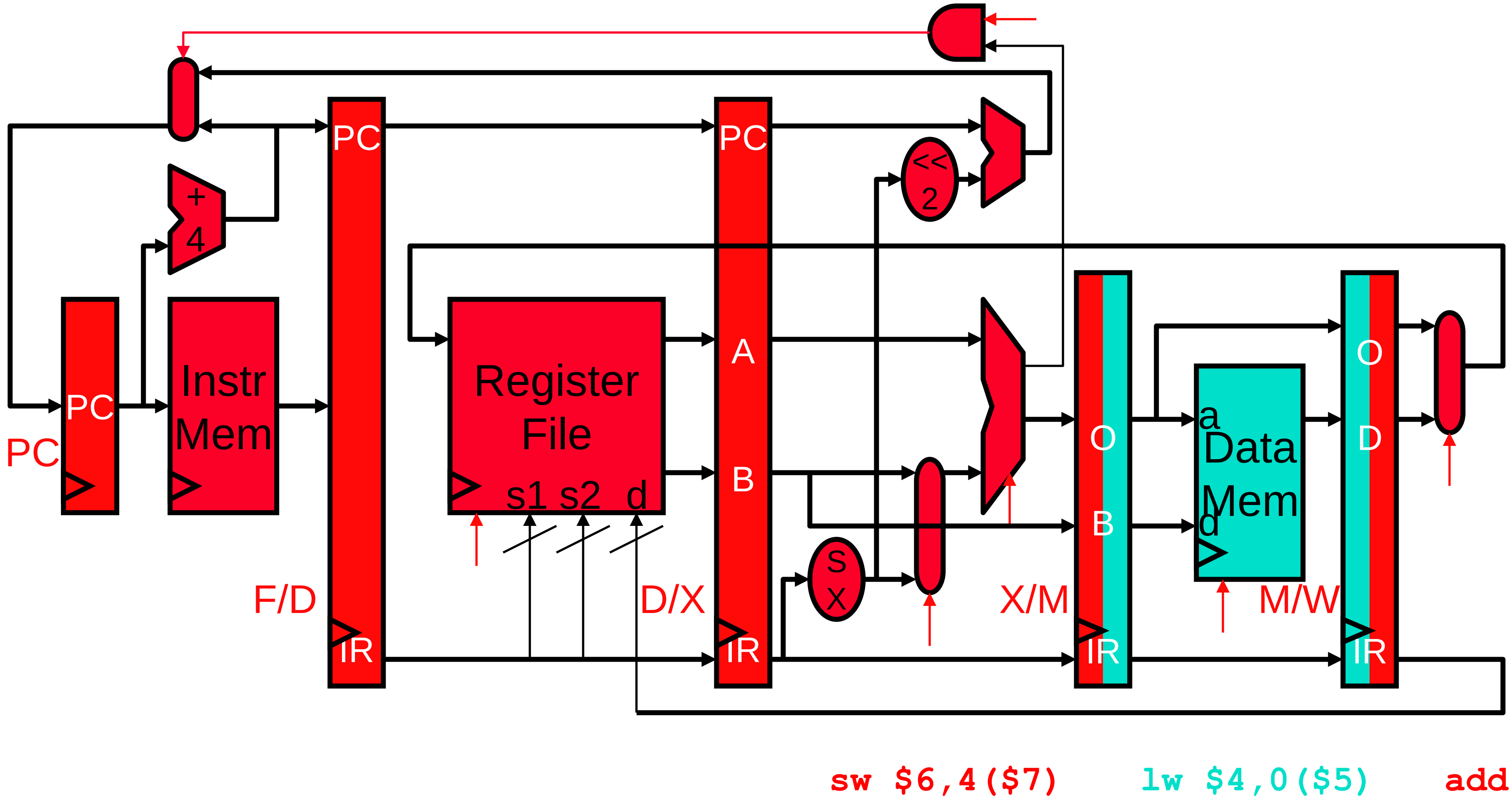
Pipeline example: Cycle 4



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

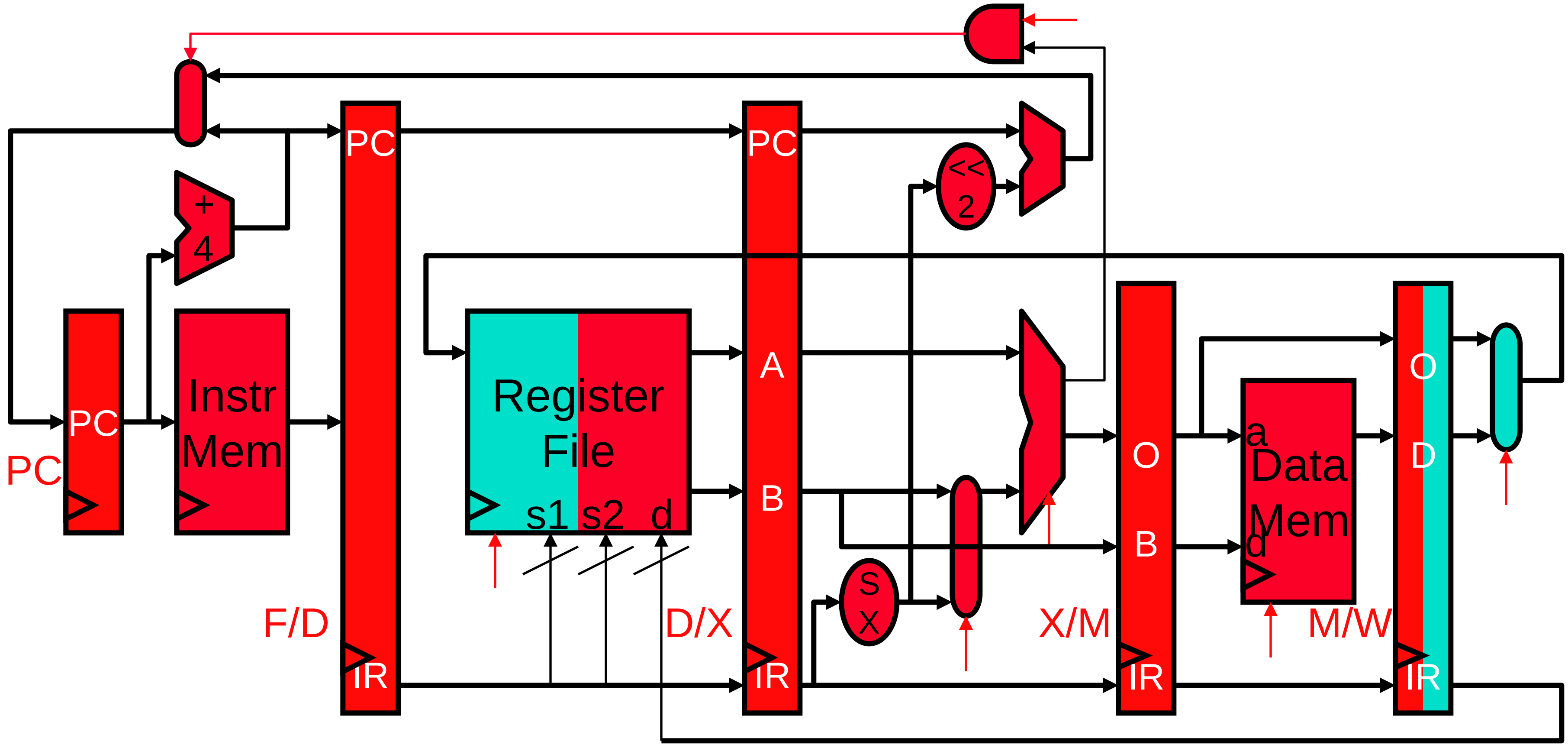
- Is the `add` instruction doing anything? If so, what?

Pipeline example: Cycle 5



- Is the `add` instruction doing anything now? If so, what?

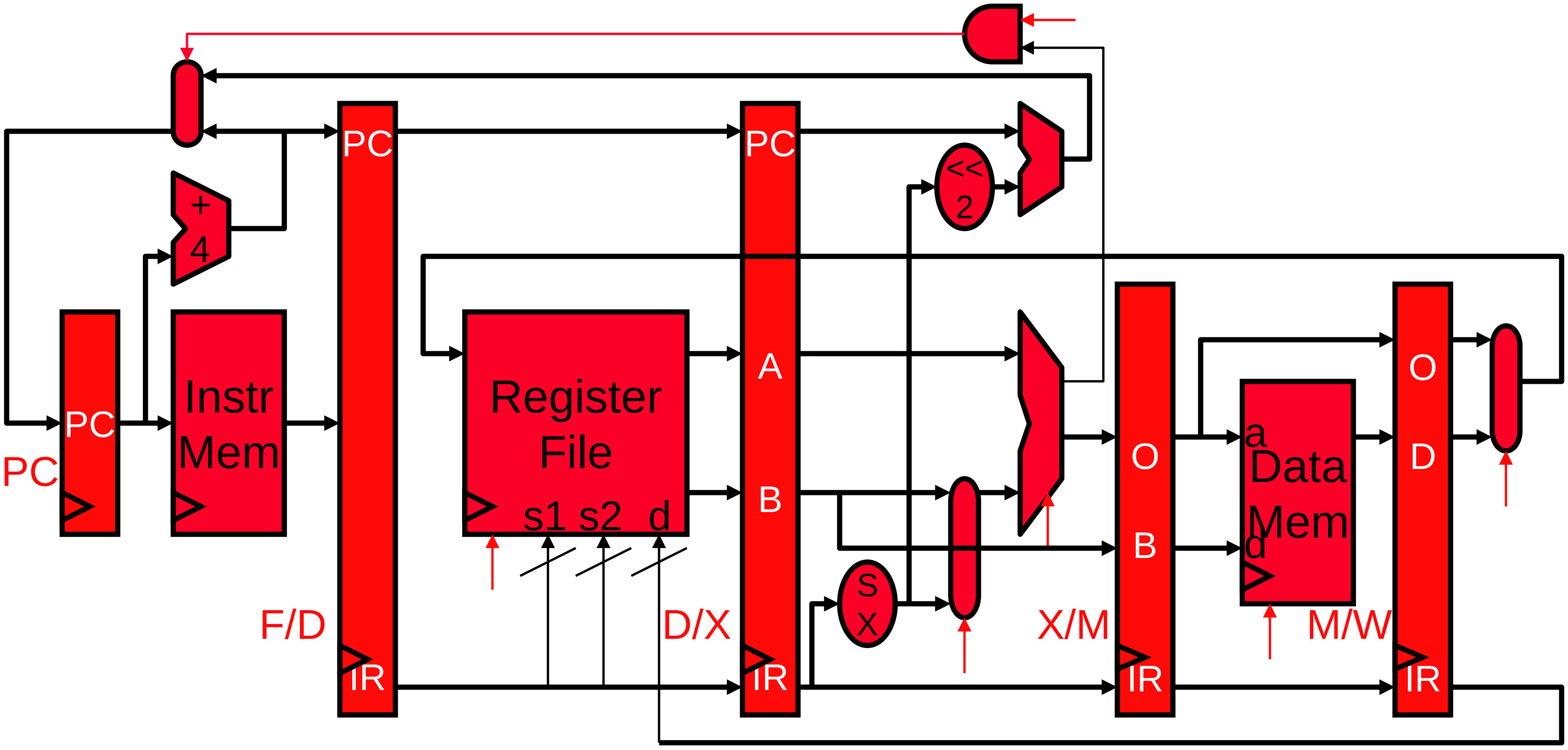
Pipeline example: Cycle 6



sw $6,4($7)   lw

- Is the `lw` instruction doing anything? If so, what?

Pipeline example: Cycle 7



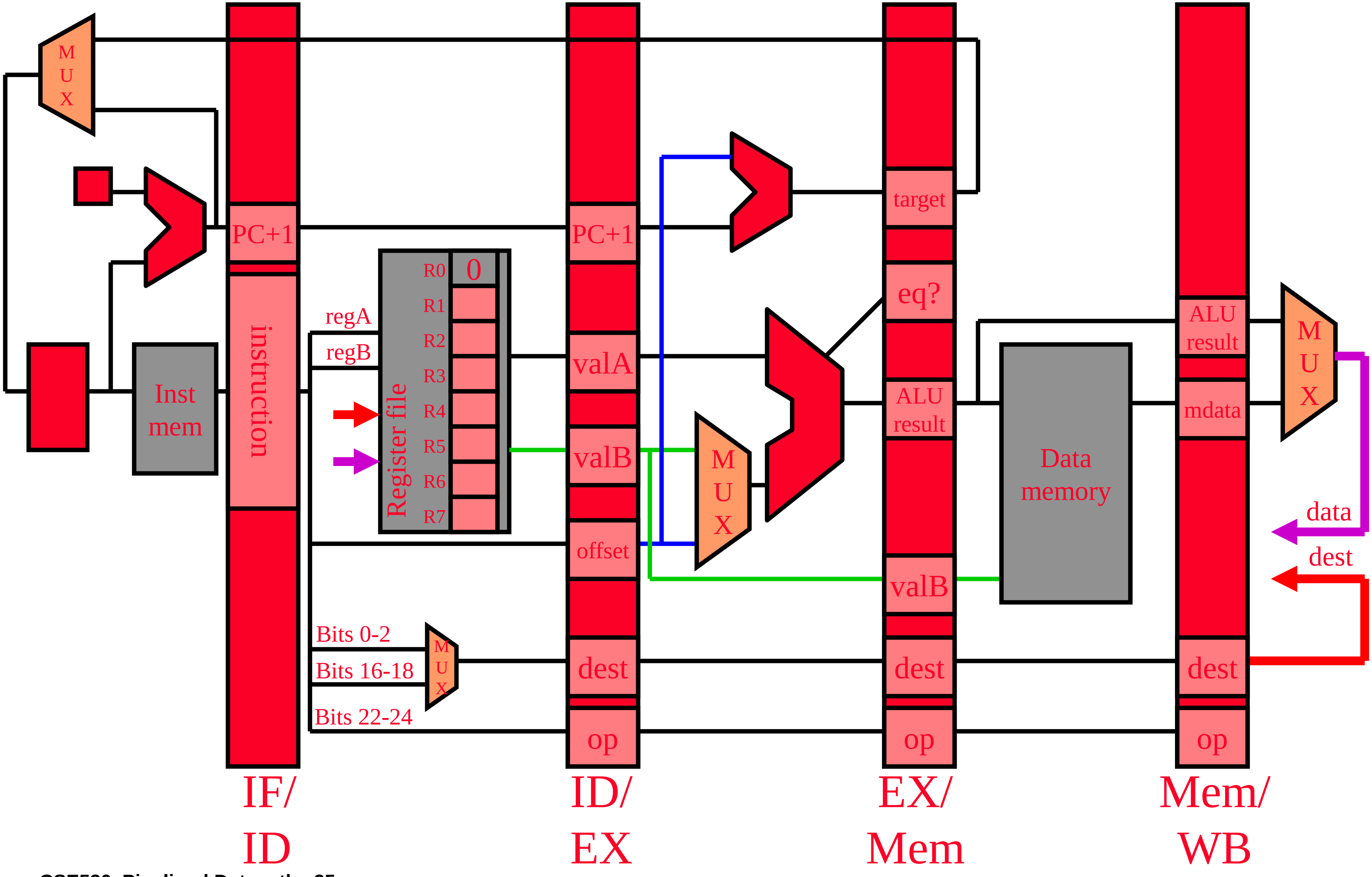
sw

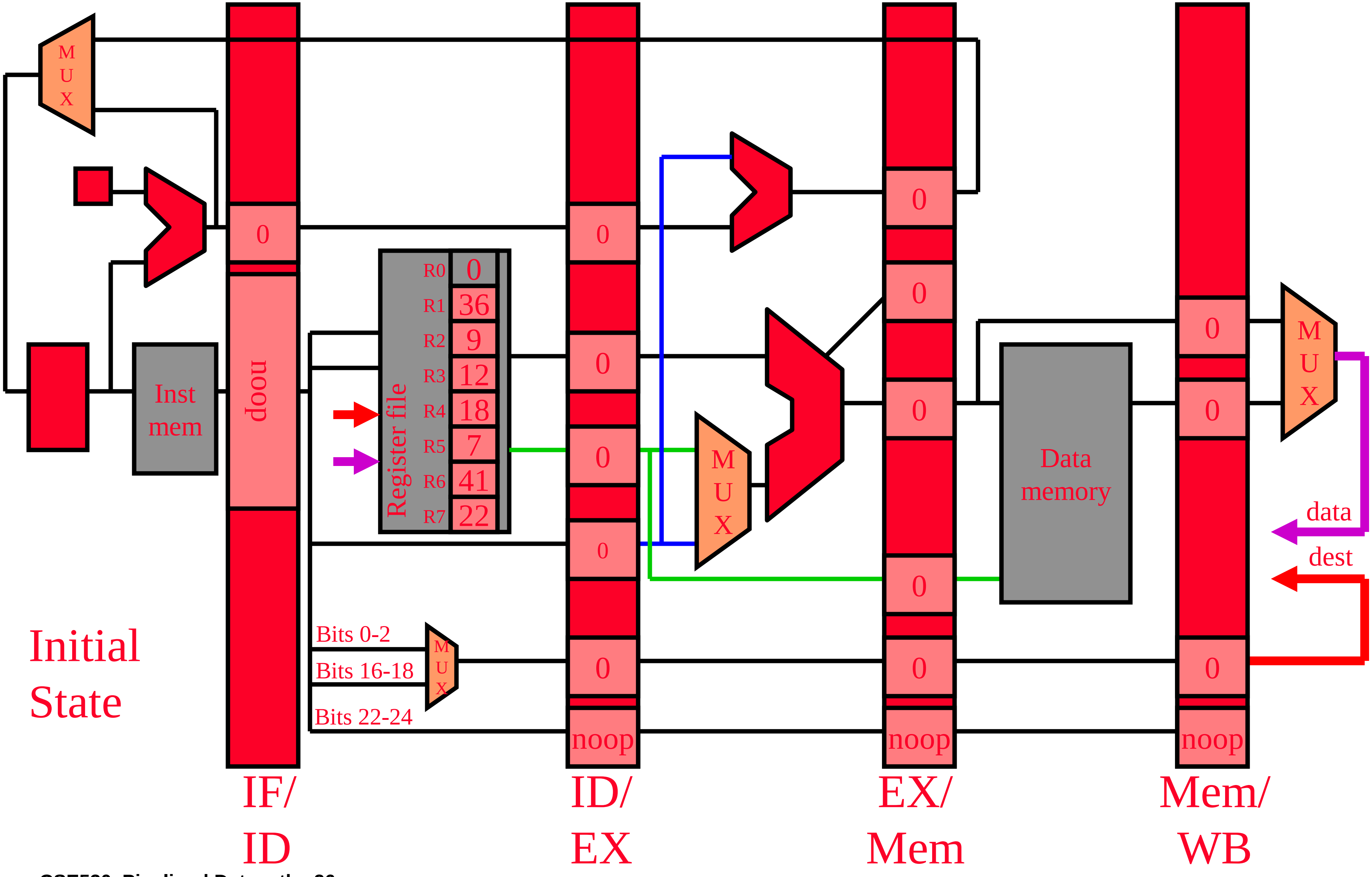
- Is the `sw` instruction doing anything? If so, what?

Sample Code (Simple)

Run the following code on a pipelined datapath:

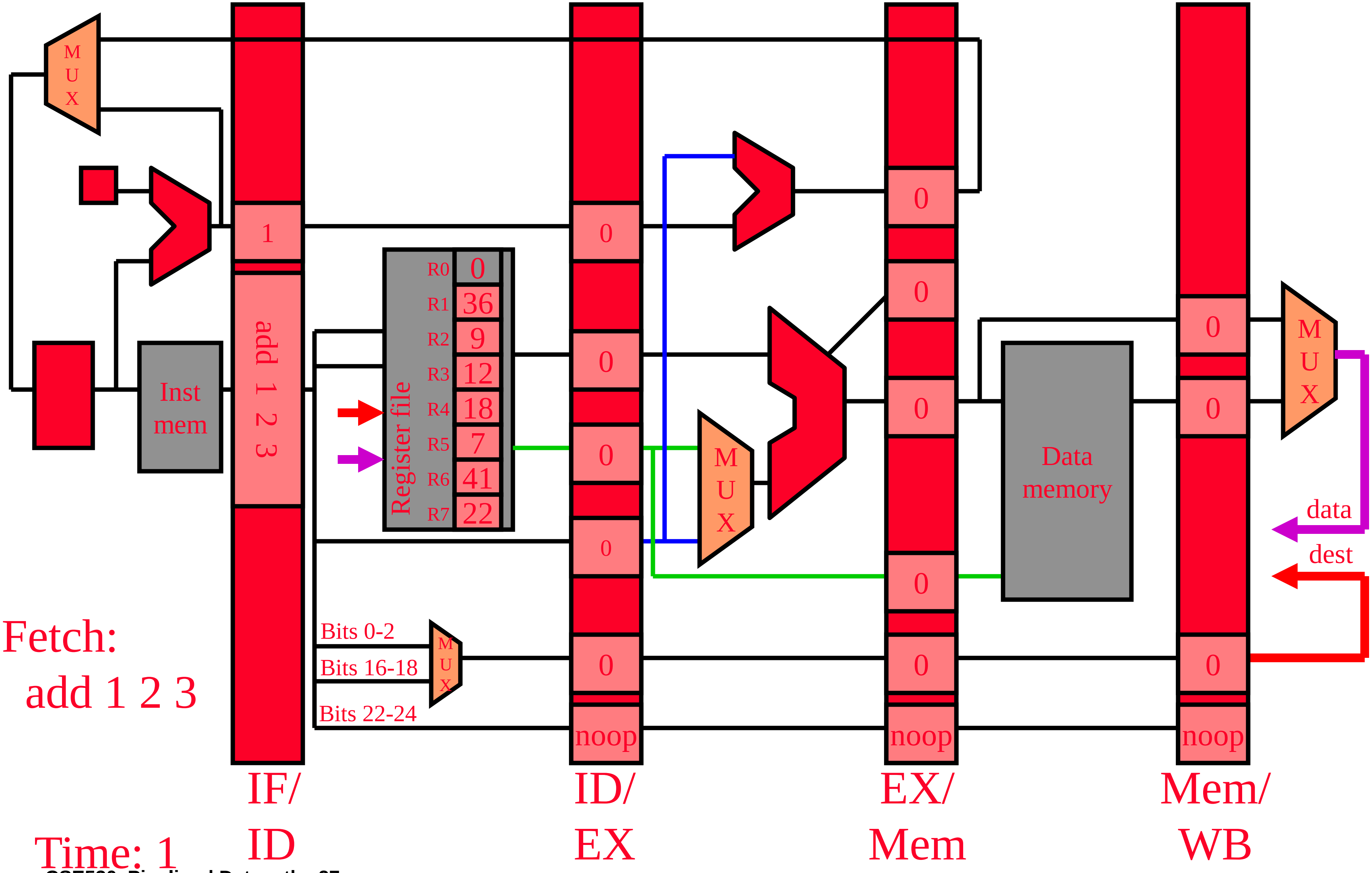
```
add     1   2   3      ;  reg 3 = reg 1 + reg 2
nand    4   5   6      ;  reg 6 = reg 4 !& reg 5
lw              2   4   20    ;  reg 4 =  Mem[reg2+20]
add     2   5   5      ;  reg 5 = reg 2 + reg 5
sw      3   7   10     ;  Mem[reg3+10] =reg 7
```


M
U
X
PC+1
Inst
mem
instruction
regA
regB
Register file
R0
R1
R2
R3
R4
R5
R6
R7
0
PC+1
valA
valB
offset
M
U
X
target
eq?
ALU
result
valB
Data
memory
ALU
result
mdata
M
U
X
data
dest
Bits 0-2
Bits 16-18
Bits 22-24
M
U
X
dest
dest
dest
op
op
op
IF/
ID
ID/
EX
EX/
Mem
Mem/
WB



Initial State

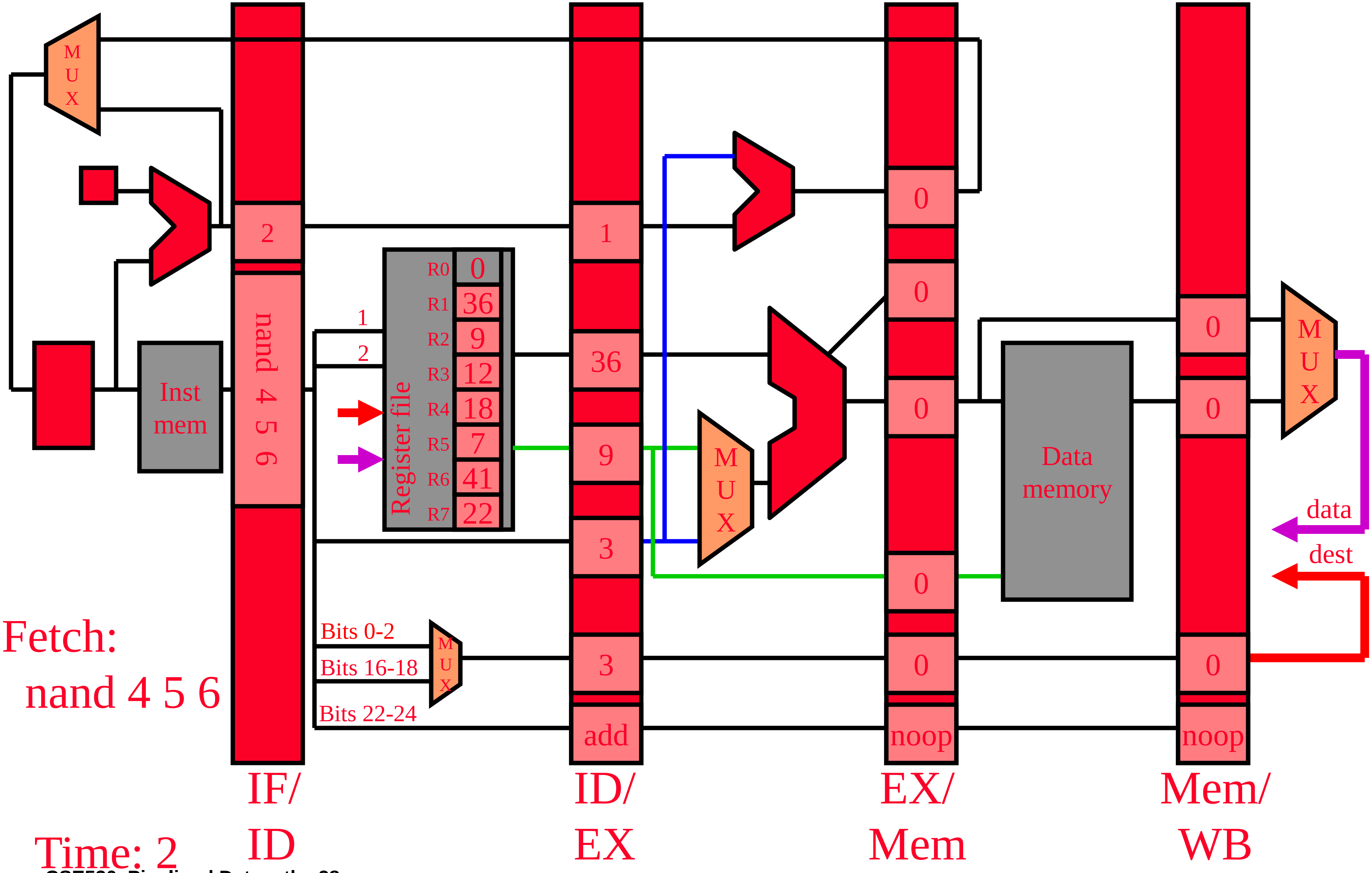
add 1 2 3



Fetch:
add 1 2 3

Time: 1

nand 4 5 6 add 1 2 3

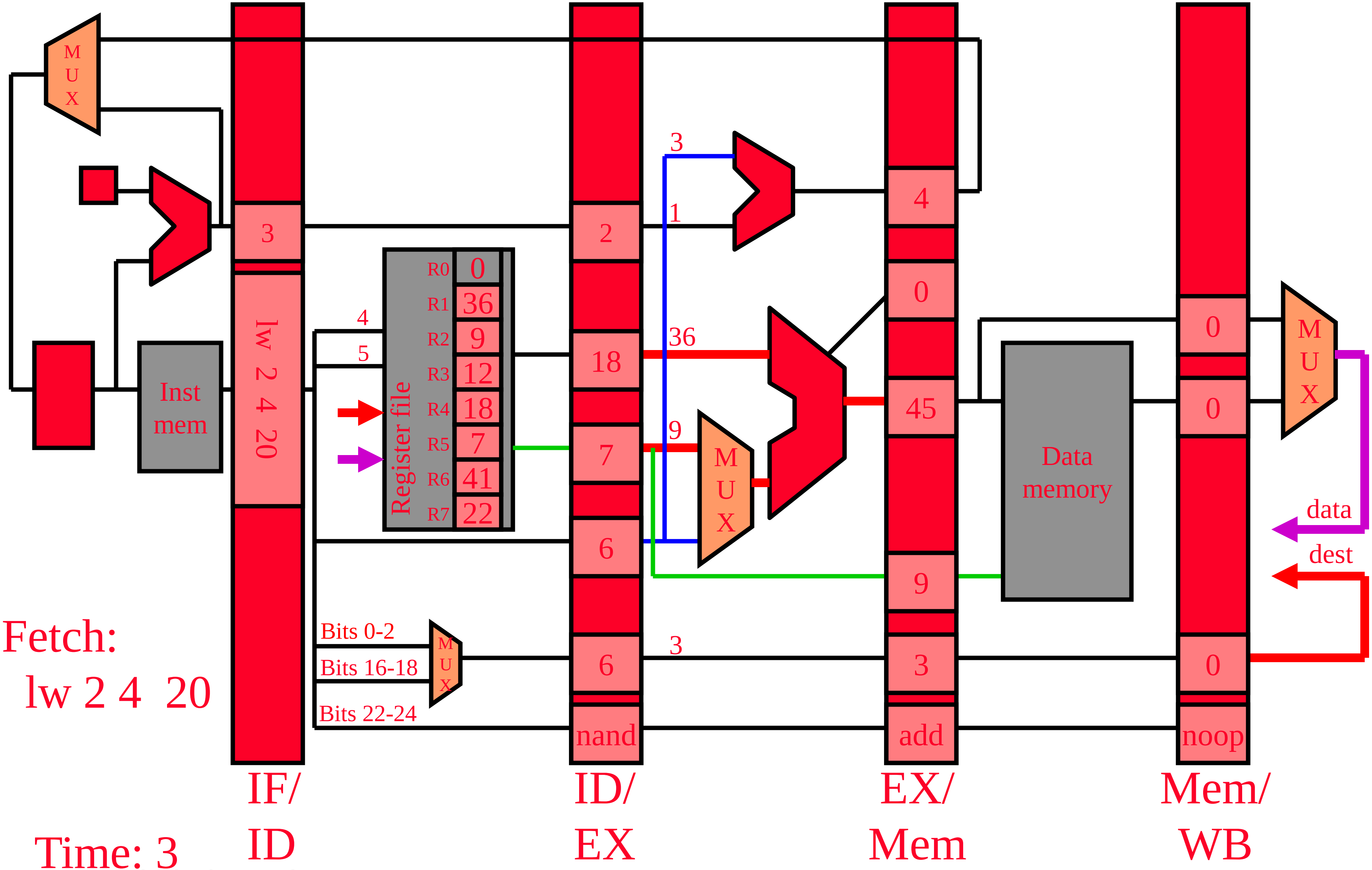


R0	0
R1	36
R2	9
R3	12
R4	18
R5	7
R6	41
R7	22

Fetch:
nand 4 5 6

Time: 2

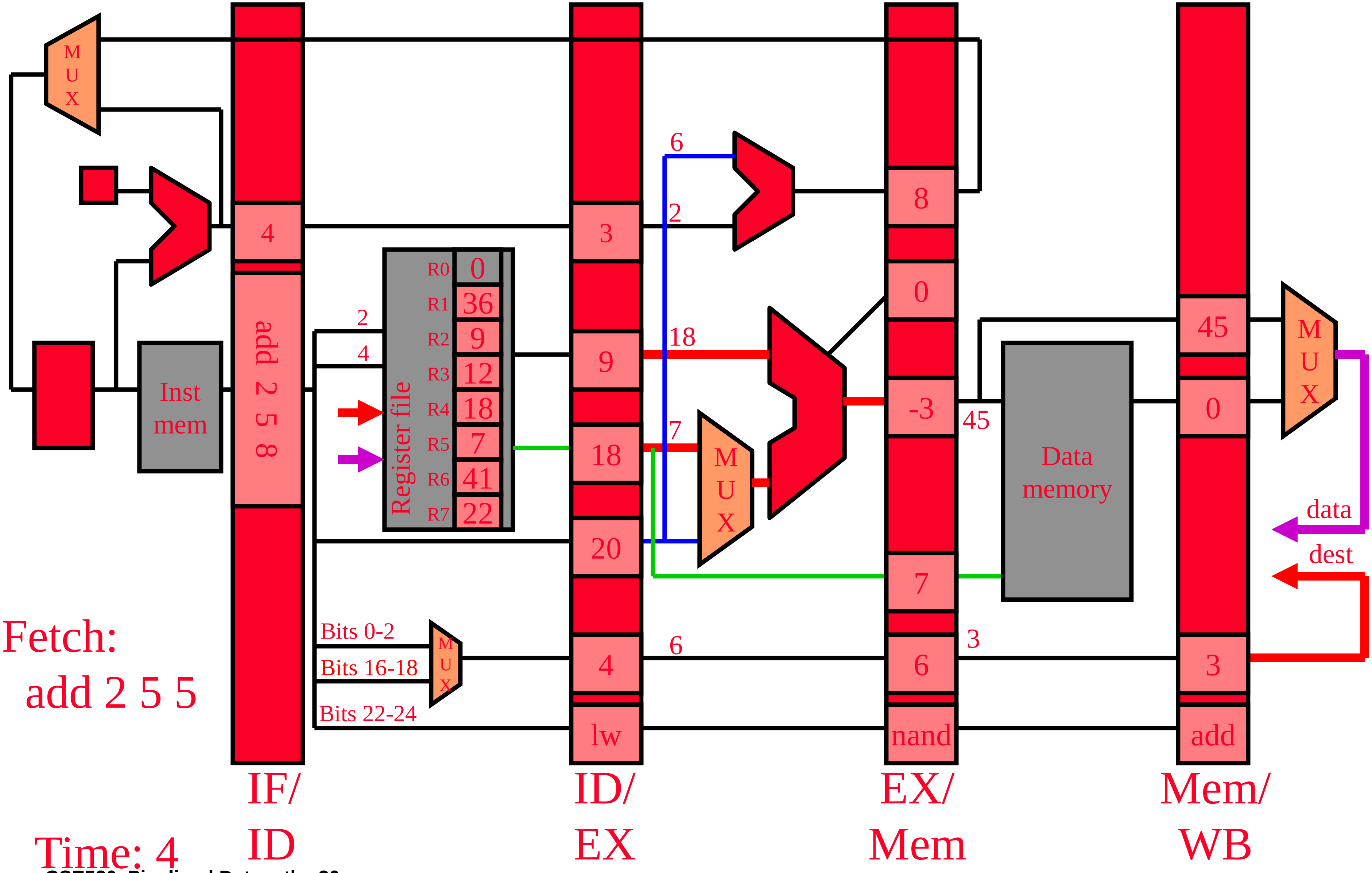
lw 2 4 20 nand 4 5 6 add 1 2 3



Fetch:
lw 2 4 20

Time: 3

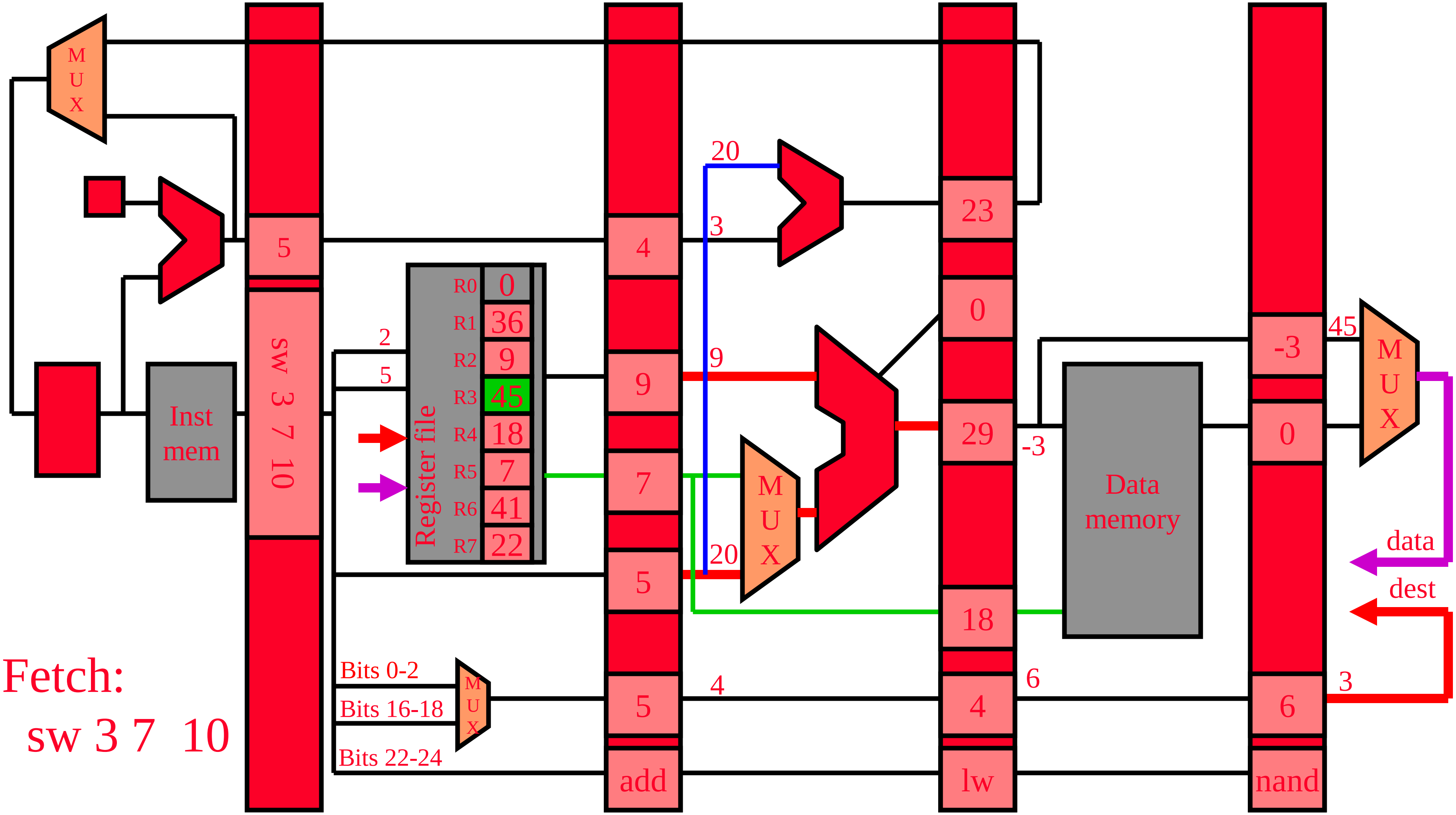
add 2 5 5 lw 2 4 20 nand 4 5 6 add 1 2 3



Fetch:
add 2 5 5

Time: 4

sw 3 7 10 add 2 5 5 lw 2 4 20 nand 4 5 6 add



IF/
ID

ID/
EX

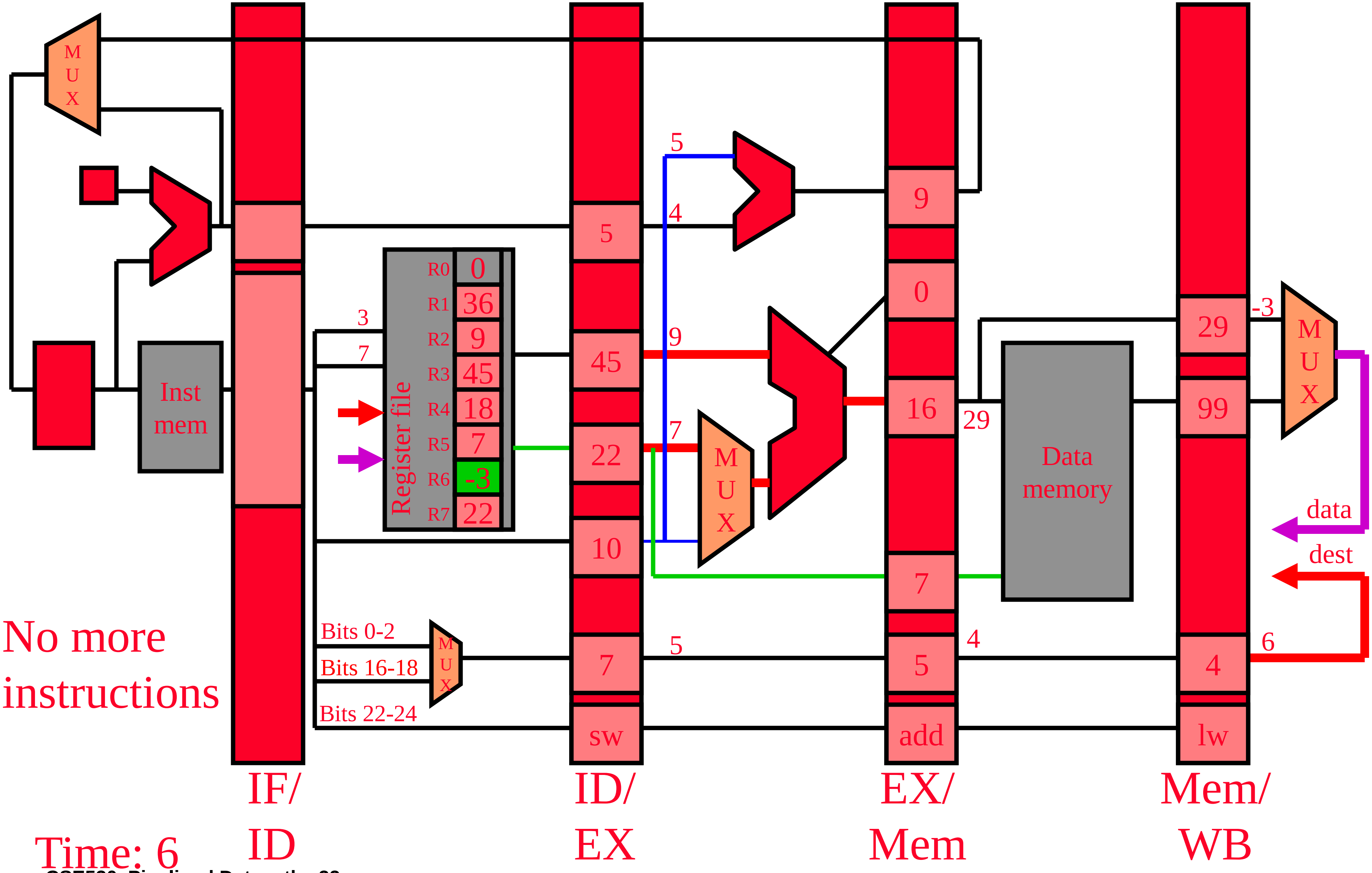
EX/
Mem

Mem/
WB

Fetch:
sw 3 7 10

Time: 5

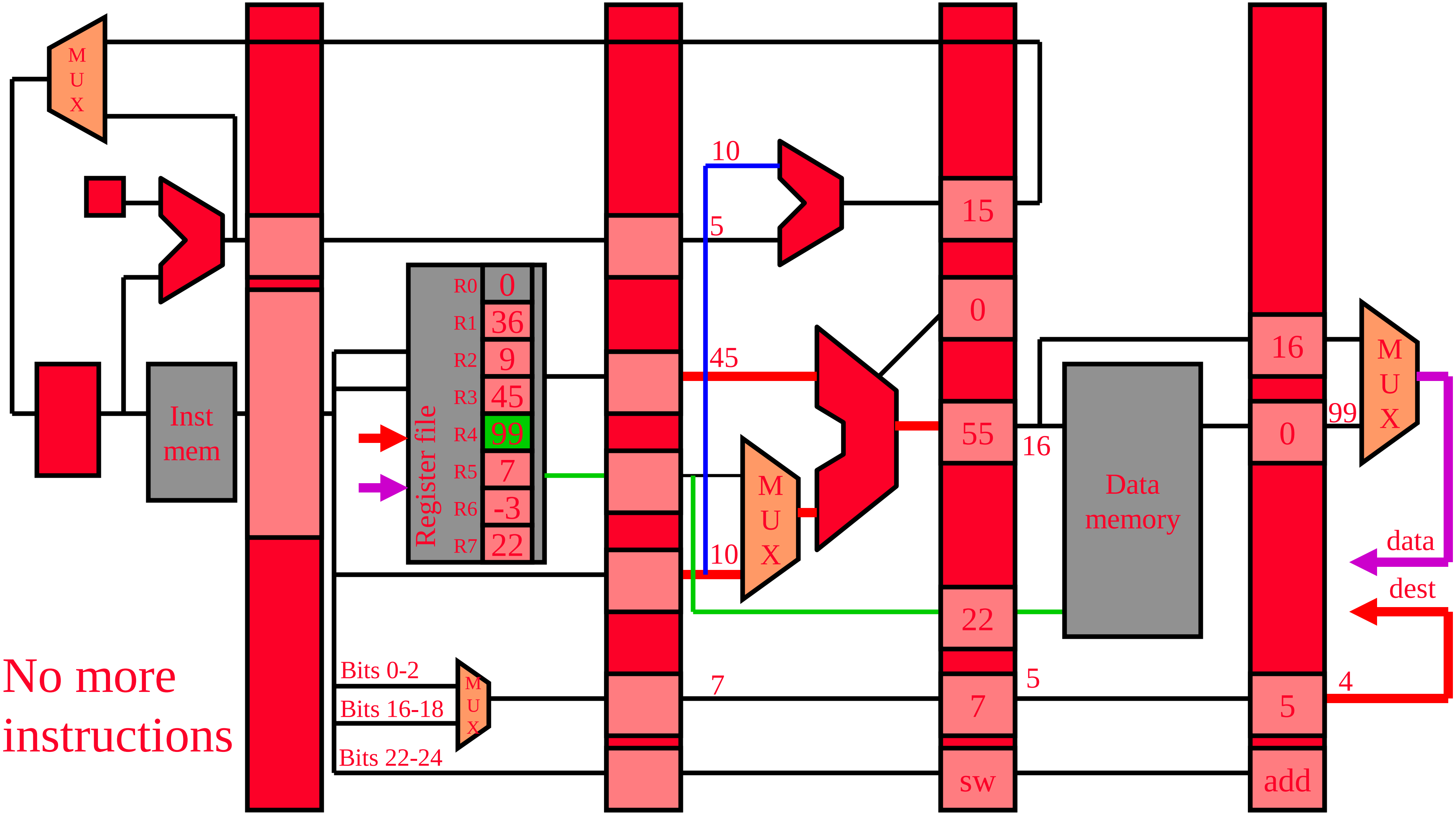
sw 3 7 10 add 2 5 5 lw 2 4 20 nand



No more instructions

Time: 6

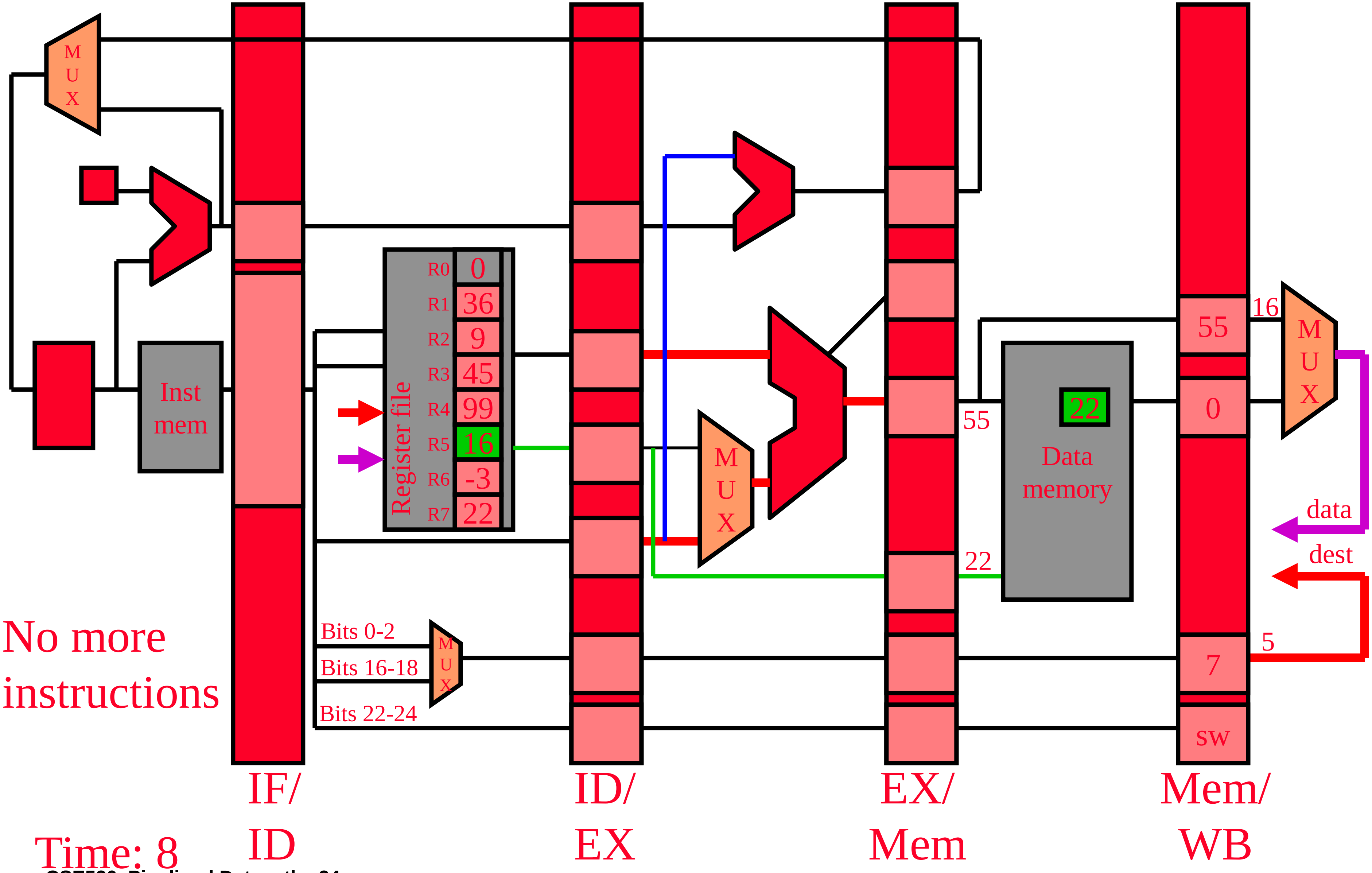
sw 3 7 10 add 2 5 5 lw



No more instructions

Time: 7

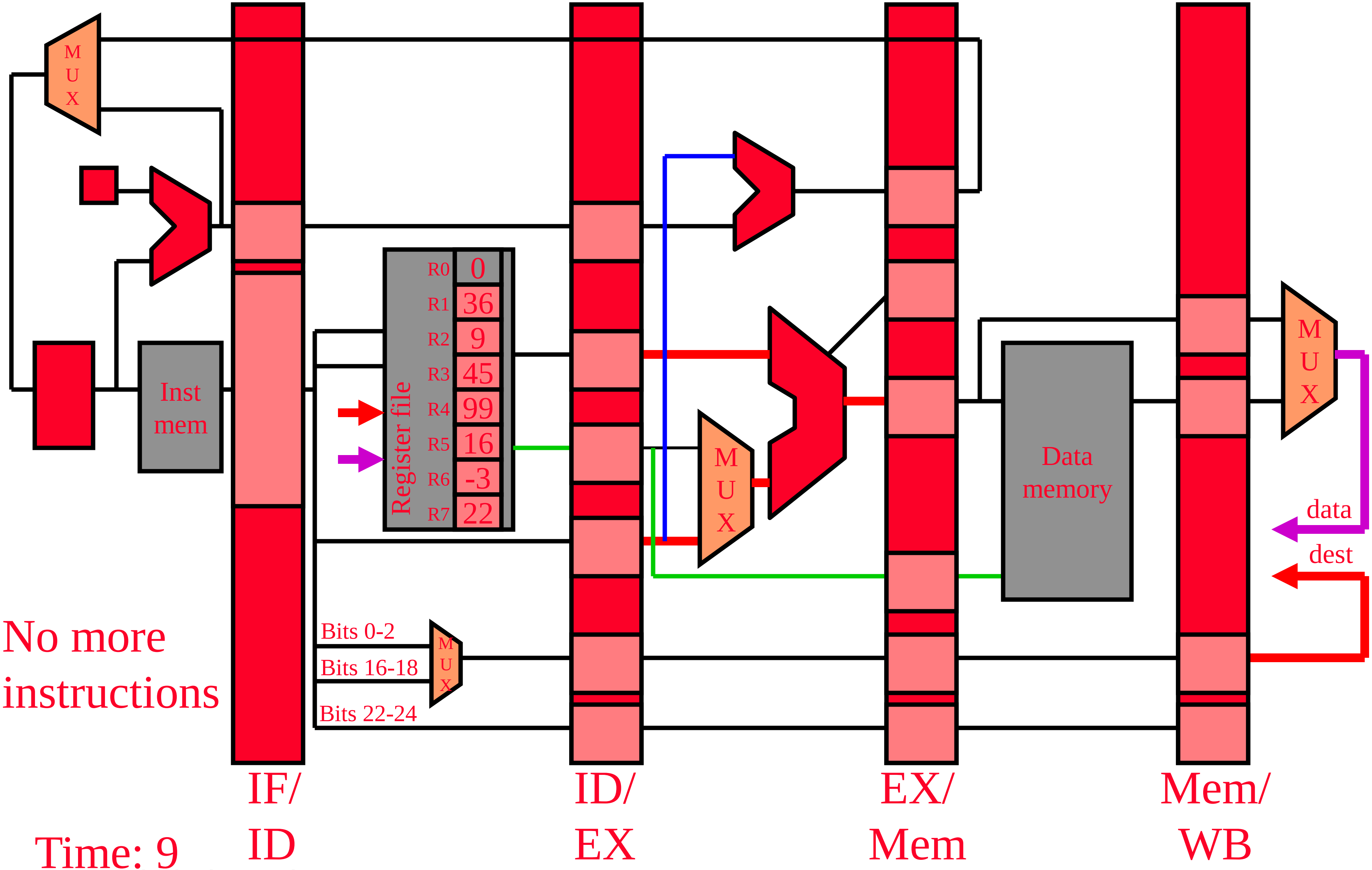
sw 3 7 10 add



No more instructions

Time: 8

SW



No more instructions

Time: 9

Pipeline diagram

- **Pipeline diagram**: shorthand for what we just saw
  - Across: cycles
  - Down: instr’s
  - 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		

Why is the pipeline CPI…

- … > 1?
  - CPI for scalar in-order pipeline is 1 **+ stall penalties**
  - Stalls are used to resolve hazards
    - **Hazard**: condition that jeopardizes sequential illusion
    - **Stall**: pipeline delay introduced to restore sequential 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/make common case fast
  - Long penalties OK if they happen rarely
    - e.g., 1 + 0.01 * 10 = 1.1
  - Stalls also have implications for determining the ideal number of pipeline stages

Data Dependences, Pipeline Hazards, and Bypassing

Dependences and hazards

- **Dependence**: relationship between two instr's
  - **Data**: two instr's use same storage location
  - **Control**: one instr affects whether another executes at all
  - The dependence relationship can always be enforced by making older instruction complete before younger ones
- **Hazard**: dependence & possibility of wrong instr order
  1. Structural hazard – resource conflicts
  2. Data hazards – when an instr depends on the results of an older instr still in the pipeline
     - Effects of wrong instr order cannot be externally visible
     - **Stall**: by keeping younger instr in same pipeline stage (no new instr's are fetched during stalls)
  3. Control hazards – from branches and other instructions that change the PC
  - Hazards are a **bad** thing: stalls reduce performance (increase CPI)

Structural hazards

Structural hazards

- Two instr's trying to use same circuit at same time
  - E.g., if had one unified L1 cache holding both instr's and data

unified ID Cache increases structural hazards

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

  - Redesign cache to allow 2 accesses per cycle (slow, expensive)
  - Stall pipeline (negatively impacts CPI)
  - Separate instruction/data memories (two L1 caches)

To fix structural hazards

- Each instr uses every structure exactly once, for at most one cycle, always at same stage relative to F (fetch)

Tolerate structure hazards

- Add stall logic to stall pipeline when hazards do occur

Data hazards - dependent data operations

- The three instr sequence we saw earlier executed fine… the instr's executed independent operations

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

  - But most programs have **true data dependences** – i.e., they pass values via registers and memory

- Would this program execute correctly on a pipeline?

```
add $3,$2,$1
add $6,$5,$3
```

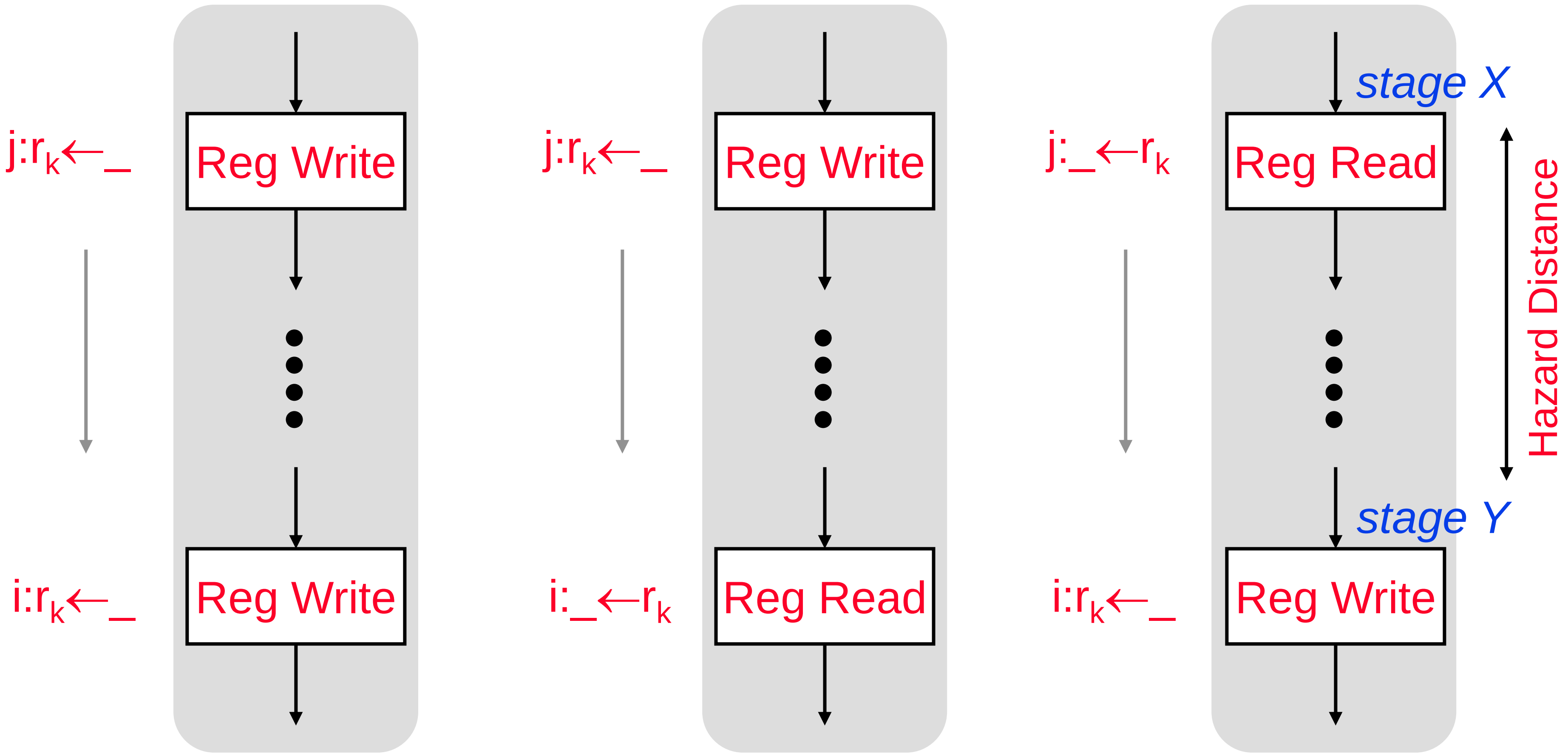
RAW data hazard (read after write)

- What about this program?

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

Necessary Conditions for Data Hazards

NOT IMPORTANT TO FORMALIZE



WAW Hazard

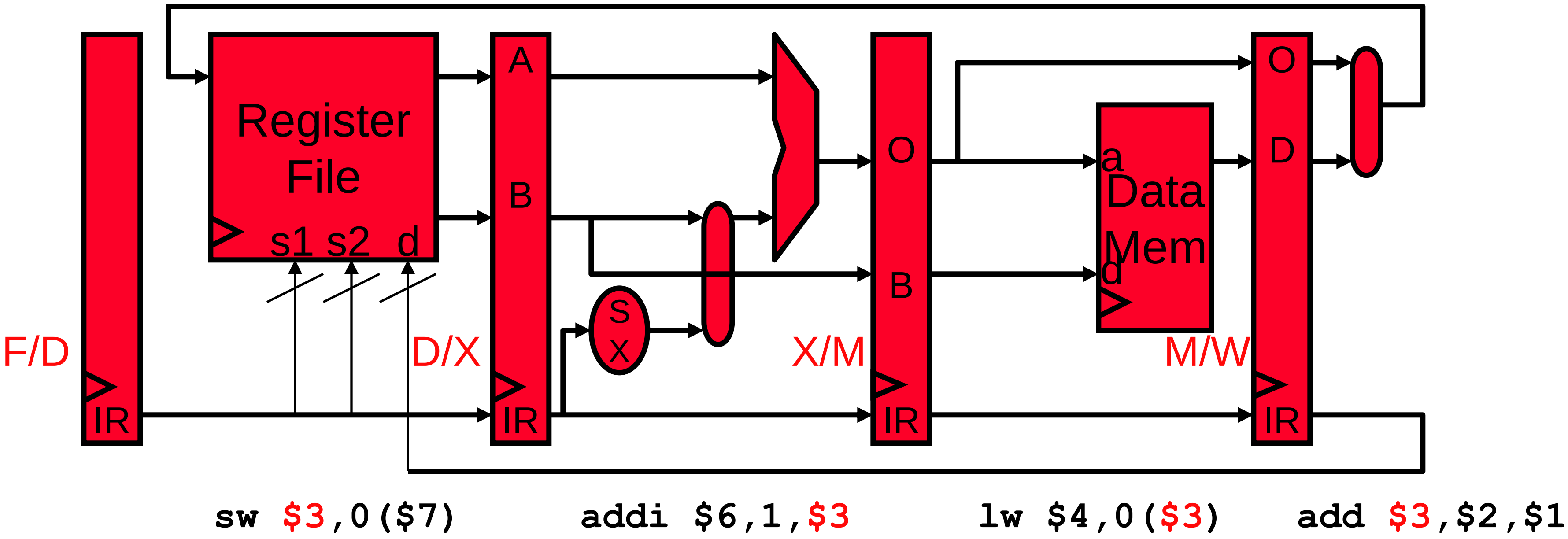
WAR Hazard

RAW Hazard

$dist(i,j) \leq dist(X,Y) \Rightarrow$ Hazard!!

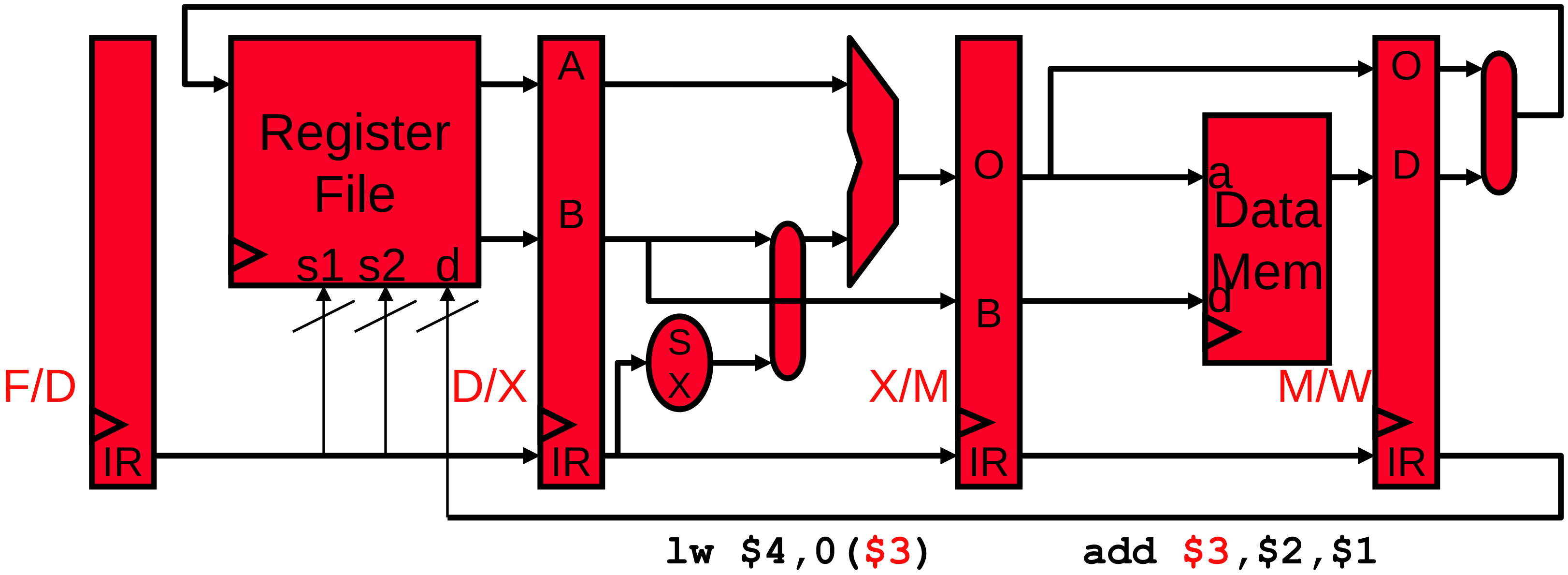
$dist(i,j) > dist(X,Y) \Rightarrow$ Safe

Data (RAW) hazard example



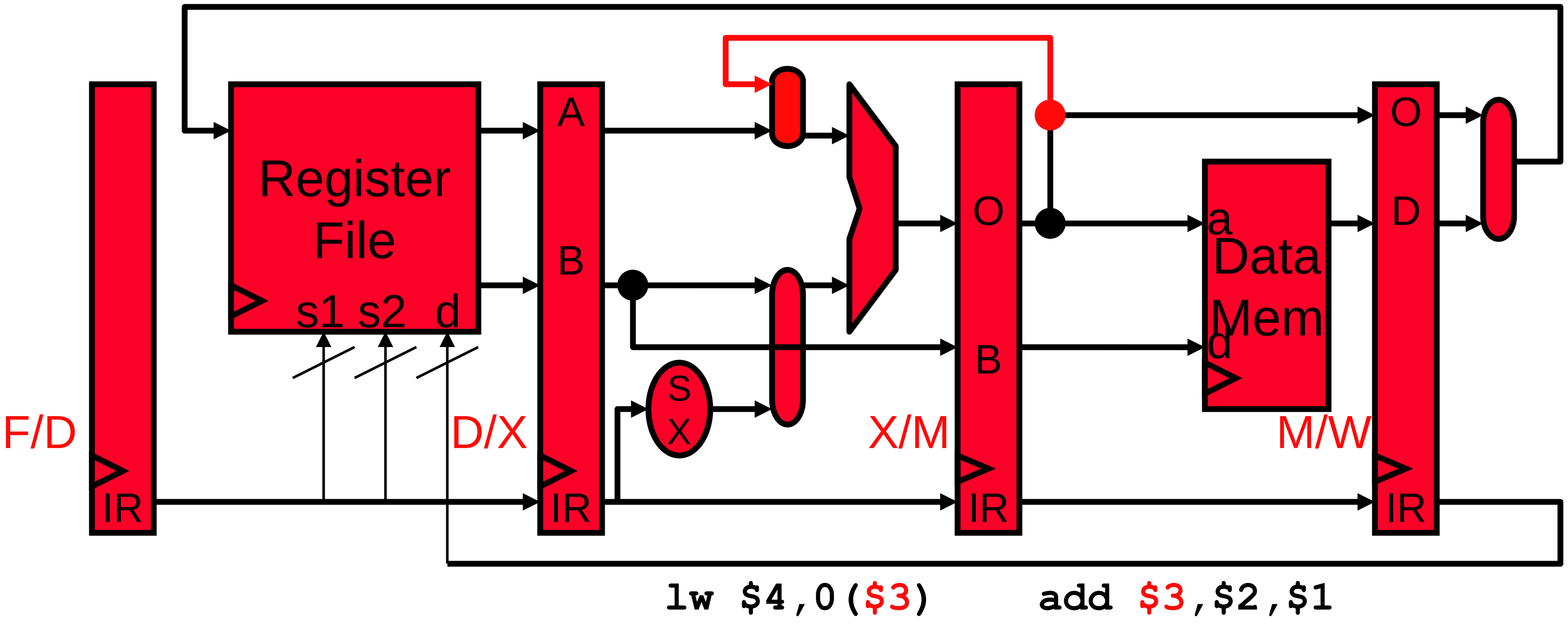
- Would this "program" execute correctly on this pipeline?
  - Which instr's would execute with correct inputs? (Start with `add`)
  - **`add`** is writing its result into **$3** in current cycle
    - **`lw`** read **$3** two cycles ago → got wrong value
    - **`addi`** read **$3** one cycle ago → got wrong value
  - **`sw`** is reading **$3** this cycle → maybe (depending on regfile design)

Observation!



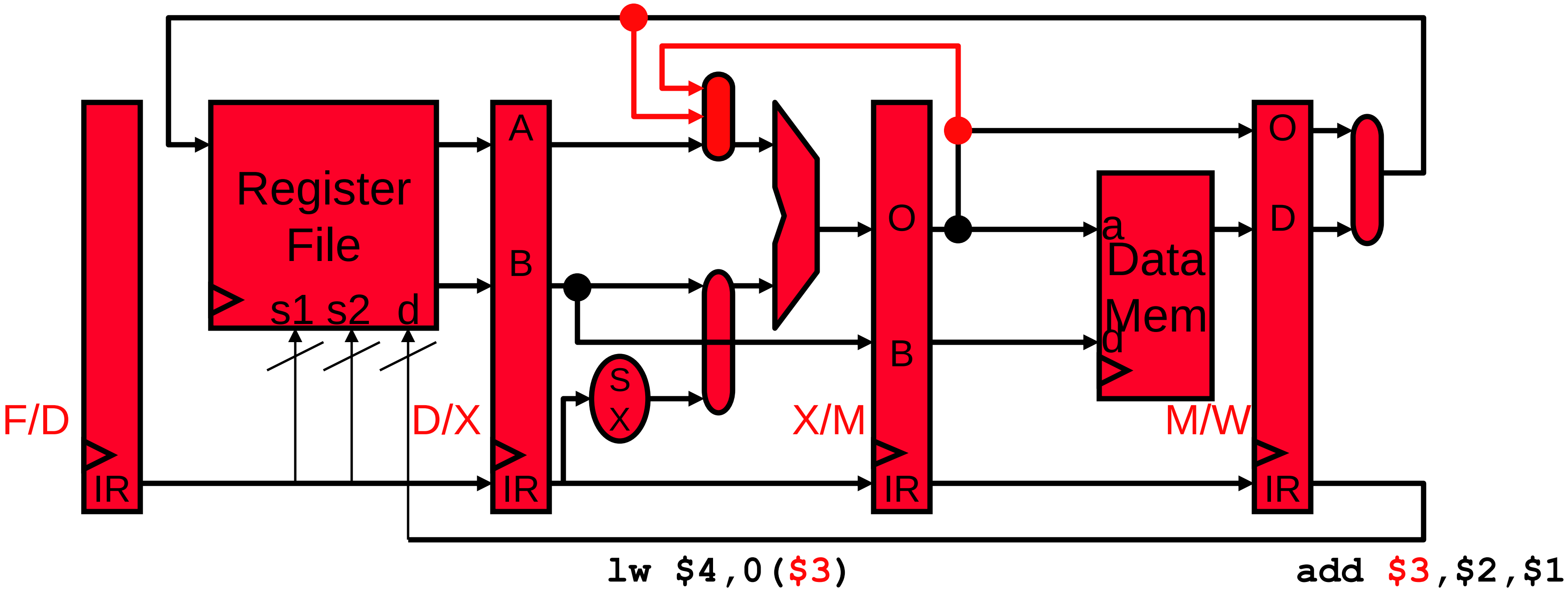
- Technically, this situation is broken
  - **`lw $4,0($3)`** has already read **$3** from regfile
  - **`add $3,$2,$1`** hasn't yet written **$3** to regfile
- But fundamentally, everything is OK
  - **`lw $4,0($3)`** hasn't actually used **$3** yet
  - **`add $3,$2,$1`** has already computed **$3**

Reducing data hazards with bypassing



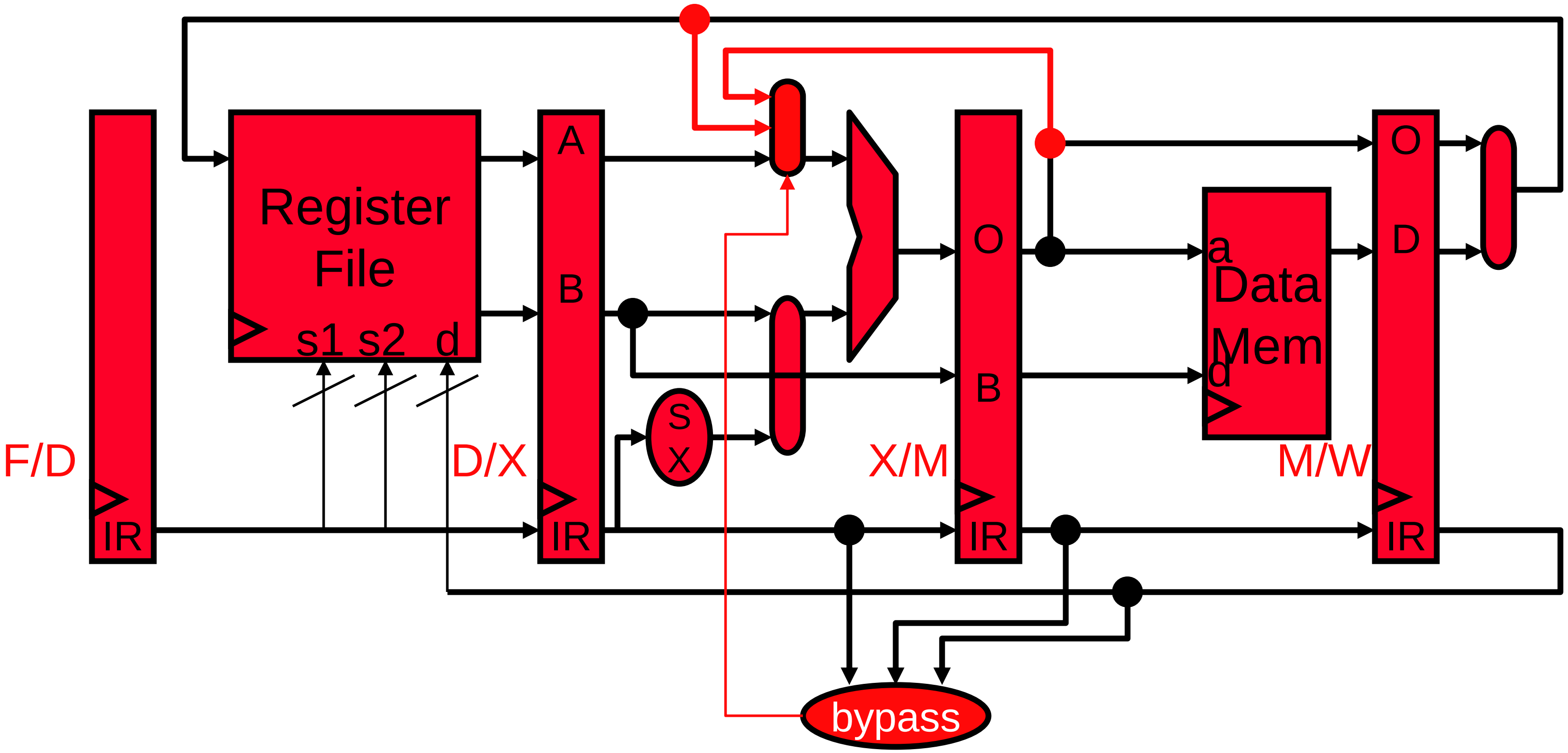
- **(MX) Bypassing**
  - Reading a value from an intermediate (μarchitectural) source
    - Don't wait until it is available from primary source (i.e., regfile)
  - Here, we are bypassing the register file
  - Also called **forwarding**

WX bypassing



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

MX/WX bypass control logic



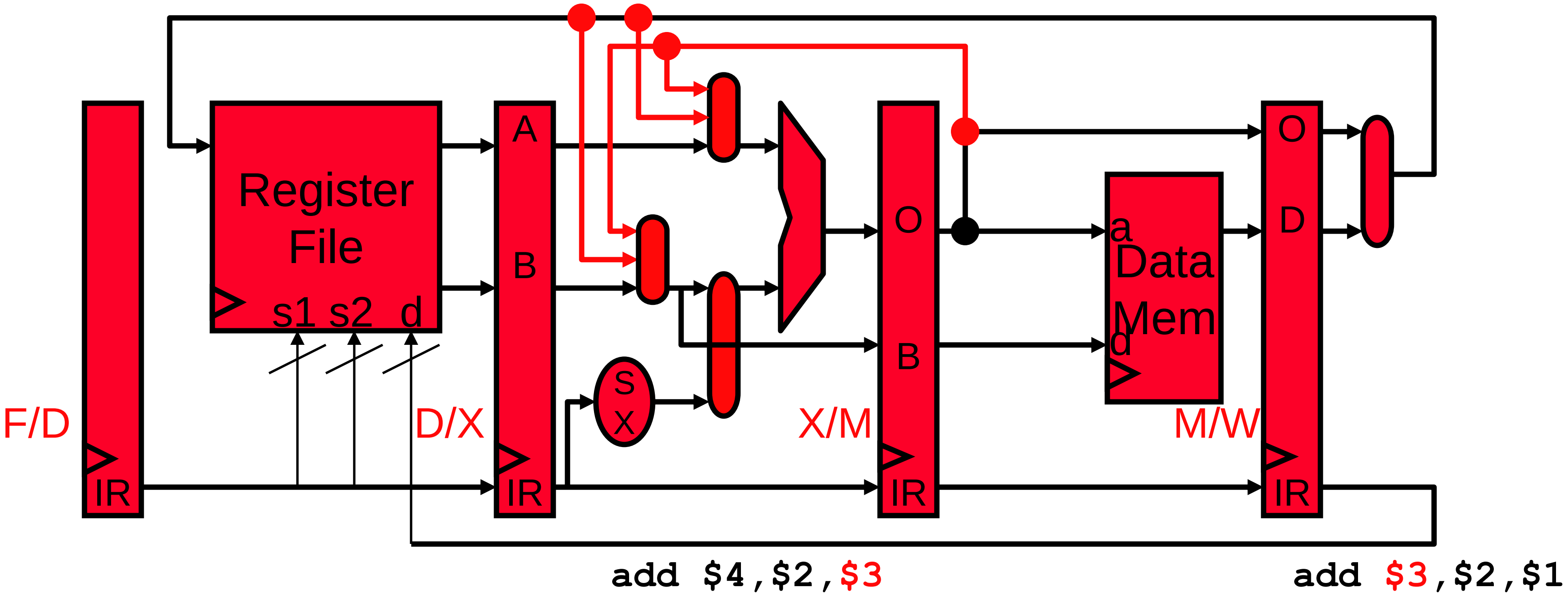
- Each bypass MUX has its own control, here it is for MUX ALUinA

  (D/X.IR.RegSource1 == X/M.IR.RegDest) => 0

  (D/X.IR.RegSource1 == M/W.IR.RegDest) => 1

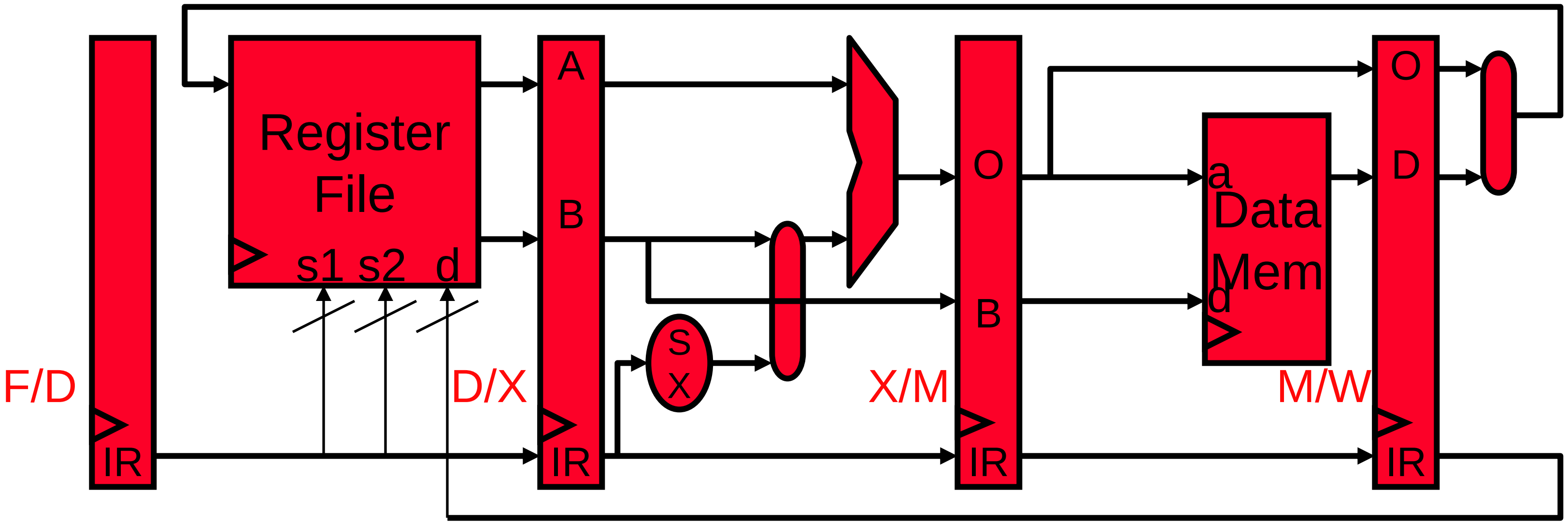
  Else => 2

ALUinB bypassing



- Can also bypass to ALUinput B

Memory data hazards

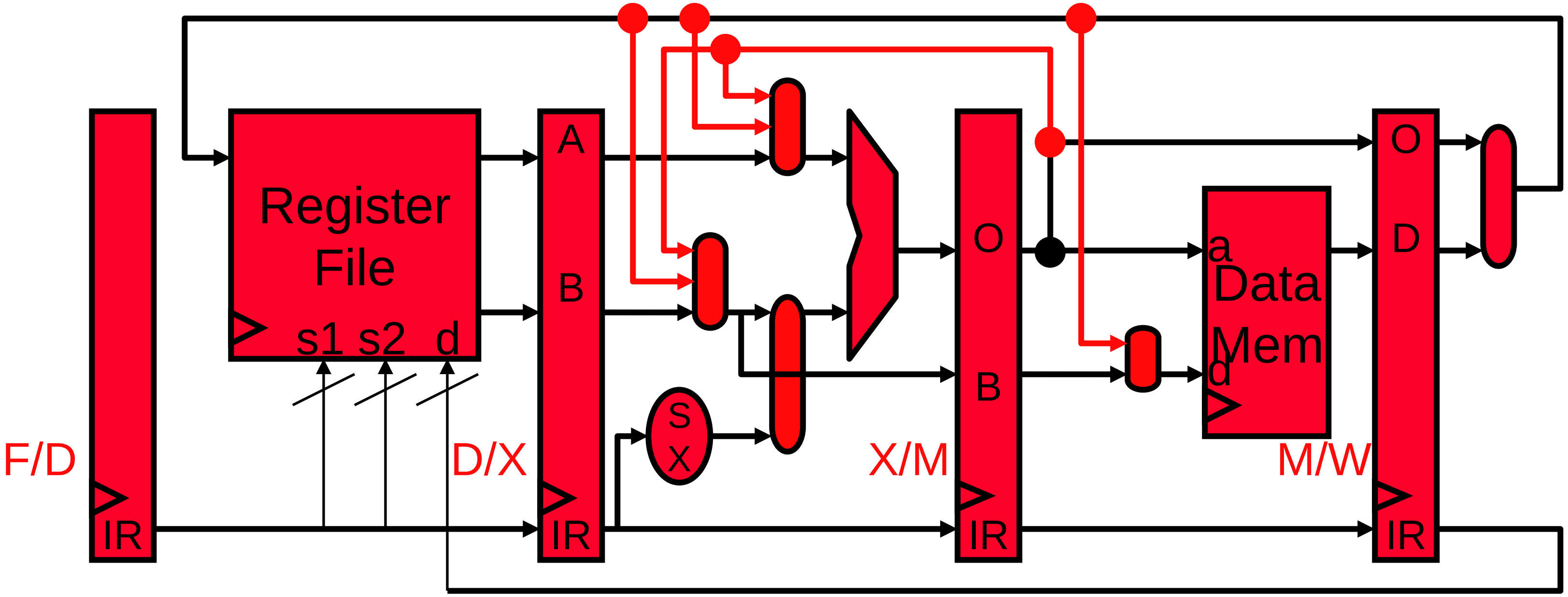


`lw $4,0($1)` `sw $5,0($1)`

- Are memory data hazards a problem for this pipeline? No
  - **`lw`** following **`sw`** to same address in next cycle, gets right value
    - Why? Data mem write/read always take place in same stage
- But how about

```
lw $3,0($2)
sw $3,0($4)
```

WM bypassing



with deeper pipelining, these MUXes will grow

```
sw $3,0($4)     lw $3,0($2)
```

with more complicated MUXes, cycle time will keep growing

- Do we need WM bypassing?
  - Not to the address input (why not?)
  - But to the store data input, yes

Pipeline diagrams with bypassing

- If bypass exists, “from”/“to” stages execute in same cycle
  - Bypassing occurs **from** the data field in the pipeline register of the stage doing the forwarding **to** the functional unit in the stage needing that data
  - Example: full bypassing, use MX bypass

	1	2	3	4	5	6	7	8	9	10
`add $2,$3➔$1`	F	D	X	**M**	W					
`sub $1,$4➔$2`		F	D	**X**	M	W				

- Example: full bypassing, use WX bypass

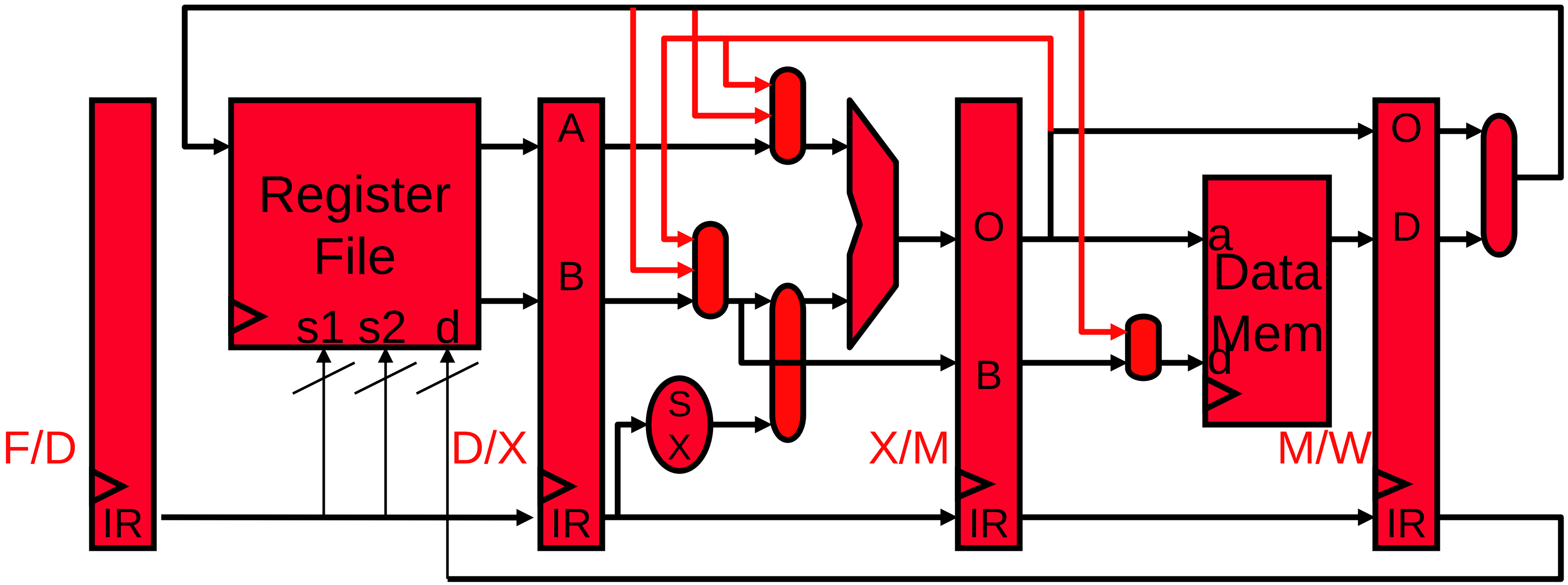
	1	2	3	4	5	6	7	8	9	10
`add $2,$3➔$1`	F	D	X	M	**W**					
`ld [$7]➔$5`		F	D	X	M	W				
`sub $1,$4➔$2`			F	D	**X**	M	W			

- Example: full bypassing, use WM bypass

What would the ? instr have to be?

	1	2	3	4	5	6	7	8	9	10
`add $2,$3➔$1`	F	D	X	M	**W**					
`?`		F	D	X	**M**	W				

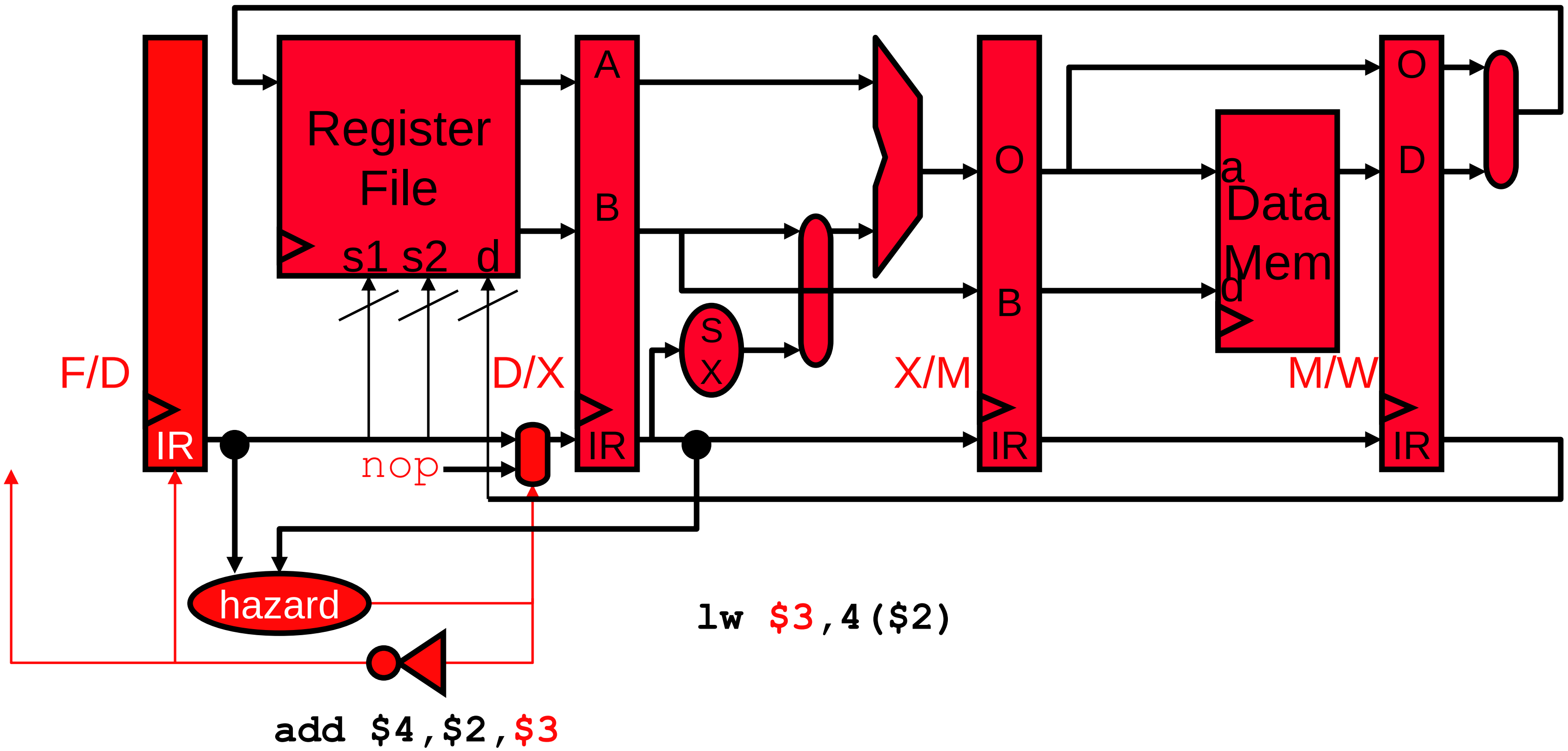
Have we prevented all the data hazards?



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

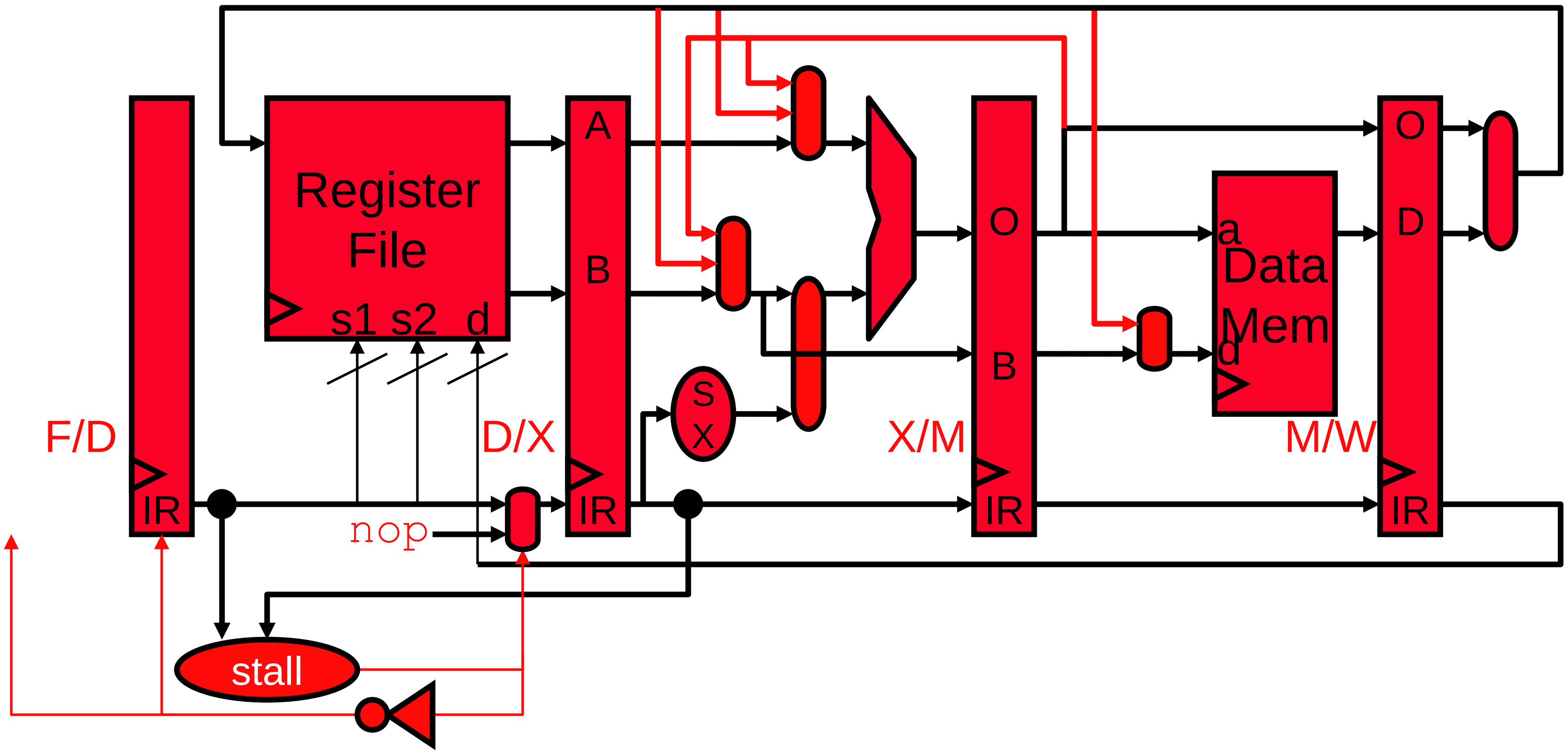
- No. Consider a “load” followed by a dependent “add” instr
  - Bypassing alone isn’t sufficient!
- Hardware solution: detect this situation and inject a stall cycle
- Software solution: ensure compiler doesn’t generate such code

Stalling to avoid data hazards



- Prevent F/D instr (`add`) from advancing this cycle
  - Write **`nop`** into D/X.IR
    - Reset (clear) the D/X control signals
  - Disable F/D latch and PC write enables (why?)

Stalling control on load-use dependences



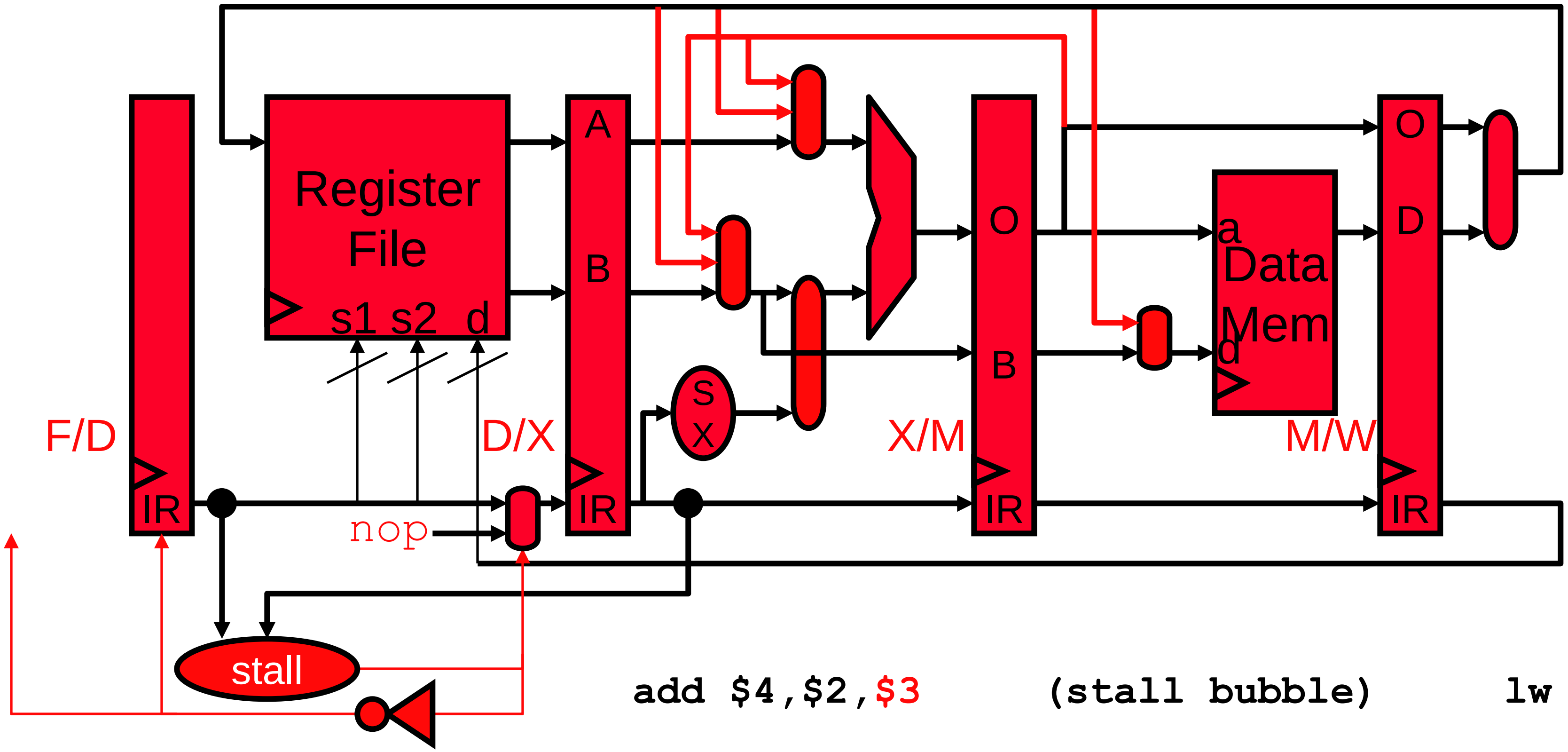
`add $4,$2,$3` `(stall bubble)` `lw $3,4($2)`

Stall = (D/X.IR.Operation == LOAD) &&

((F/D.IR.RegSrc1 == D/X.IR.RegDest) ||

(F/D.IR.RegSrc2 == D/X.IR.RegDest) && (F/D.IR.OP != STORE))

Stalling on load-use dependences



`add $4,$2,$3` `(stall bubble)` `lw $3,…`

- Now can bypass data ($3) from M/W to ALUinput B

Performance impact of load-use penalty

- Assume
  - Branch: 20%, load: 20%, store: 10%, other: 50%
  - 50% of loads are followed by dependent instruction
    - require 1 cycle stall (i.e., insertion of 1 **`nop`**)

- Calculated CPI
  - CPI = 1 + (1 * 20% * 50%) = **1.1**

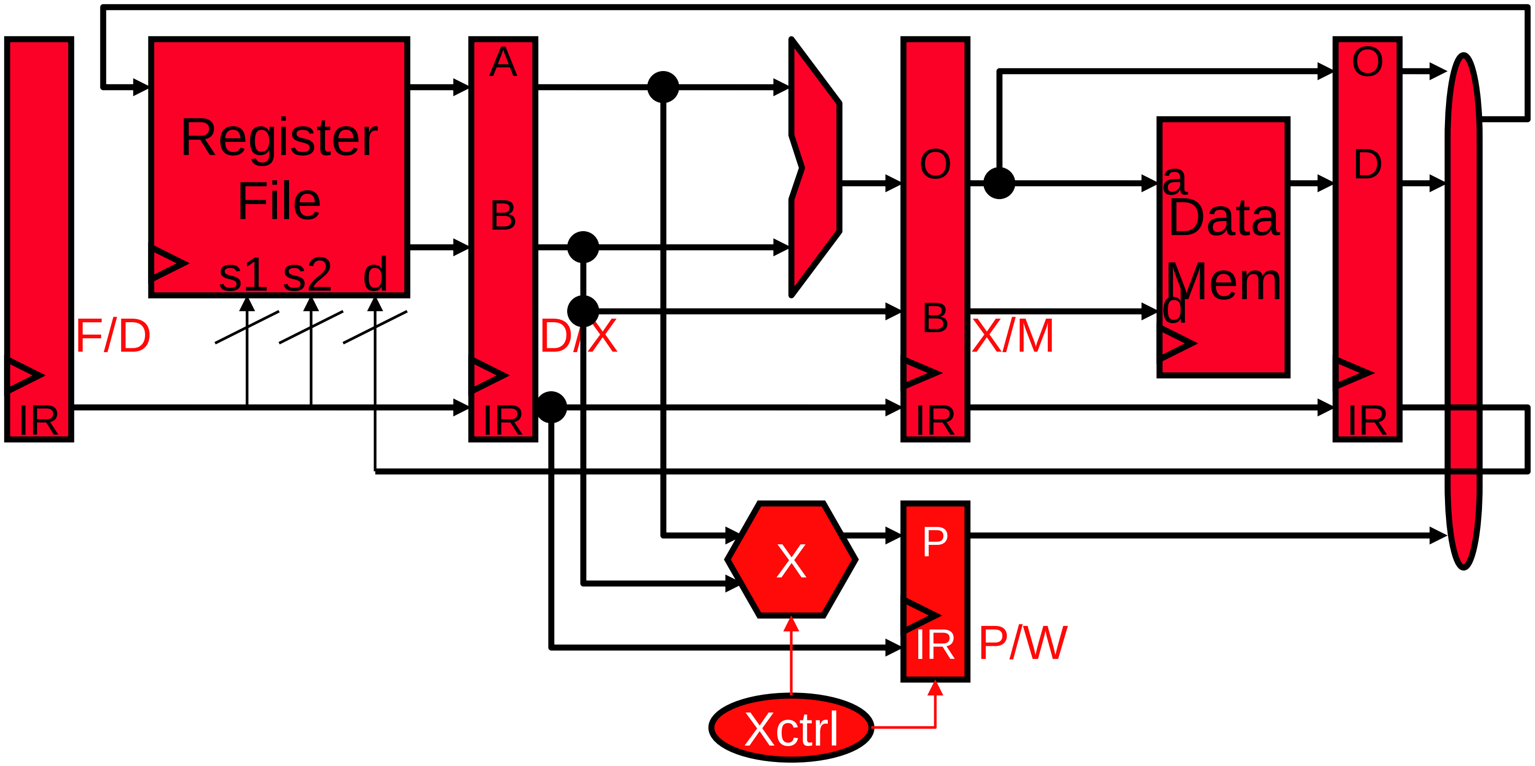
Reducing load-use stall frequency

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

- Use compiler scheduling to reduce load-use stall frequency
  - More on compiler scheduling later

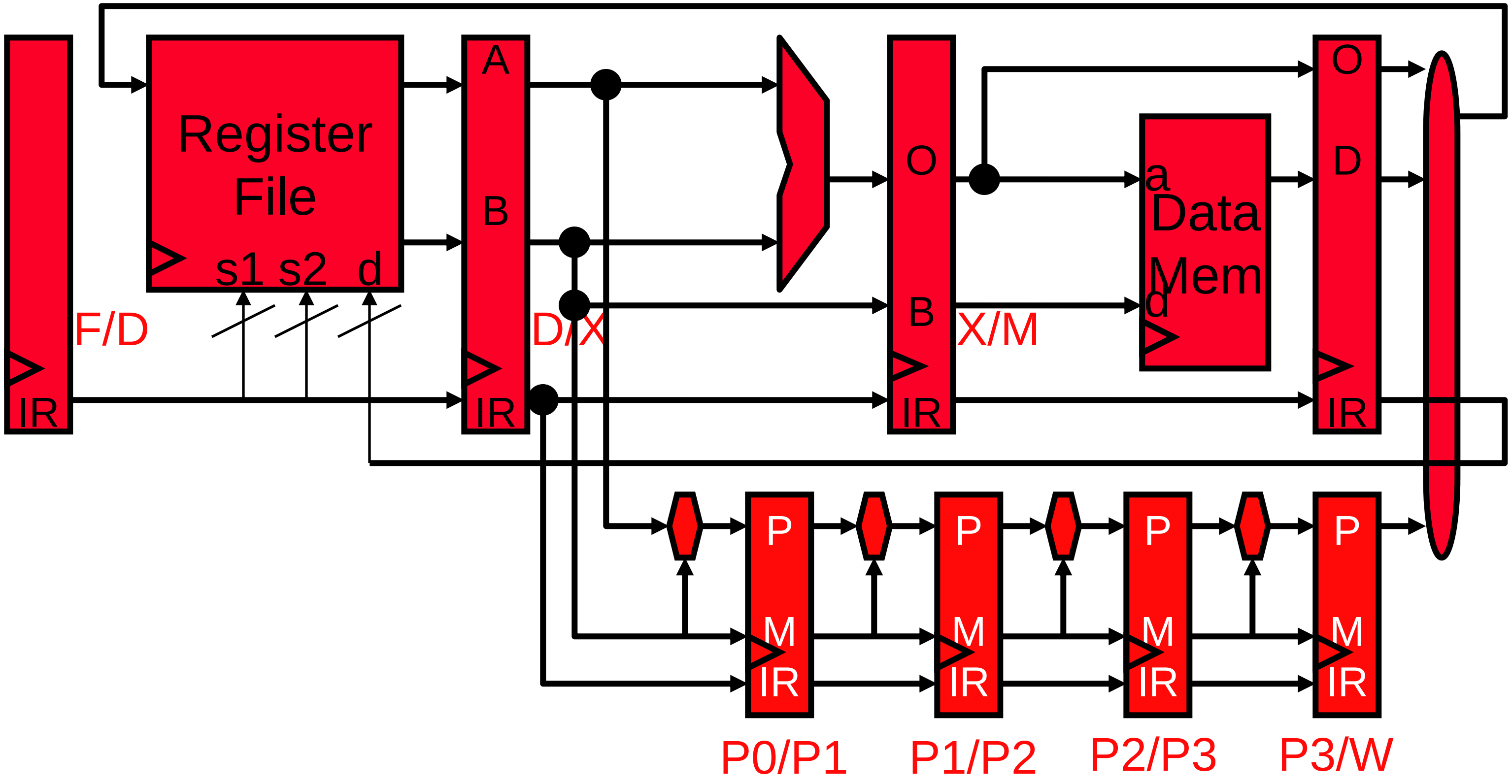
	1	2	3	4	5	6	7	8	9
add $3,$2,$1	F	D	X	M	W				
lw $4,4($3)		F	D	X	M	W			
sub $8,$3,$1			F	D	X	M	W		
addi $6,$4,1				F	D	X	M	W	

Pipelining and multi-cycle operations



- What if you wanted to add a multi-cycle operation?
  - E.g., a 4-cycle multiply
  - **P/W**: a separate output latch connects to W stage
  - Controlled by pipeline control finite state machine (FSM)

A pipelined multiplier



- Multiplier itself is often pipelined, what does this mean?
    - Product/multiplicand register/ALUs/latches replicated
    - Can start different multiply operations in consecutive cycles

Pipeline diagram with pipelined multiplier

	1	2	3	4	5	6	7	8	9
`mul  $4,$3,$5`	F	D	P0	P1	P2	P3	W		
`addi $6,$4,1`		F	**d**	**d**	**d**	**D**	X	M	W

- What about…
  - Two instructions trying to write regfile in same cycle?
  - Structural hazard! Must also prevent (or build a two write port register file).

	1	2	3	4	5	6	7	8	9
`mul  $4,$3,$5`	F	D	P0	P1	P2	P3	W		
`addi $6,$1,1`		F	D	X	M	**W**			
`add  $5,$6,$10`			F	D	X	M	**W**		

Not In Order!

2 Instructions, 1 Stage!

More multiplier nasties

- What about…
  - Mis-ordered writes to the same register (**WAW hazards**)
    - Software thinks **`add`** gets **`$4`** from **`addi`**, actually gets it from **`mul`**

	1	2	3	4	5	6	7	8	9
`mul  $4,$3,$5`	F	D	P0	P1	P2	P3	W		
`addi $4,$1,1`		F	D	X	M	**W**	**WAW!**		
…									
…									
`add  $10,$4,$6`					F	D	X	M	W

- Common? Not for a 4-cycle multiply with 5-stage pipeline
  - More common with deeper pipelines
  - In any case, we must ensure the correct outcome

Corrected pipeline diagram

- With the correct stall logic
  - Prevent mis-ordered writes to the same register
  - Why two cycles of delay?

	1	2	3	4	5	6	7	8	9
`mul  $4,$3,$5`	F	D	P0	P1	P2	P3	W		
`addi $4,$1,1`		F	**d**	**d**	D	X	M	**W**	
…									
…									
`add  $10,$4,$6`					F	D	X	M	W

- **Multi-cycle operations complicate pipeline logic**

Pipelined Functional Units

- ❑ Almost all multi-cycle functional units are pipelined
  - □ Each operation takes N cycles
  - □ But can start initiate a new (independent) operation every cycle
  - □ Requires internal latching and some hardware replication
  - + A cheaper way to add bandwidth than multiple non-pipelined units

	1	2	3	4	5	6	7	8	9	10	11
`mulf f0,f1,f2`	F	D	E*	E*	E*	E*	W				
`mulf f3,f4,f5`		F	D	E*	E*	E*	E*	W			

- ● One exception: int/FP divide: difficult to pipeline and not worth it

	1	2	3	4	5	6	7	8	9	10	11
`divf f0,f1,f2`	F	D	E/	E/	E/	E/	W				
`divf f3,f4,f5`		F	D	s*	s*	s*	E/	E/	E/	E/	W

- ● s* = structural hazard, two instr’s need same structure
  - ISAs and pipelines designed to have few of these
  - Canonical example: all instr’s forced to go through M stage

Optimal Pipeline Depth

Why is the pipeline clock period …

- … > (delay thru datapath) / (number of pipeline stages)?

  - A few reasons:
    - Pipeline stages have different delays (the goal of balancing the delay of each pipeline stage is difficult to achieve), clock period is determined by the delay of the slowest pipeline stage
    - Pipeline registers add delay (setup and propagation times)
    - Extra "bypassing" logic adds delay
    - Clock skew and jitter
  - These factors have implications for the ideal number pipeline stages
    - Diminishing clock frequency gains for longer (deeper) pipelines

Trends in pipeline depth

- Trend had been to deeper (to a point) 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"**
  - Core 2: 14, Nehalem: 20, Kaby Lake: 14 (14-20 in between)
- Increasing **pipeline depth**
  - + Increases clock frequency (reduces clock period)
    - But doubling the # of stages reduces the clock period by less than 2x
  - – Decreases IPC (increases CPI)
    - Branch mis-prediction penalty becomes longer
    - Non-bypassed data hazard stalls (e.g., load-use) become longer
  - At some point, causes performance to decrease, but when?
    - 1GHz Pentium 4 was slower than 800 MHz PentiumIII
  - "Optimal" pipeline depth is program and technology specific

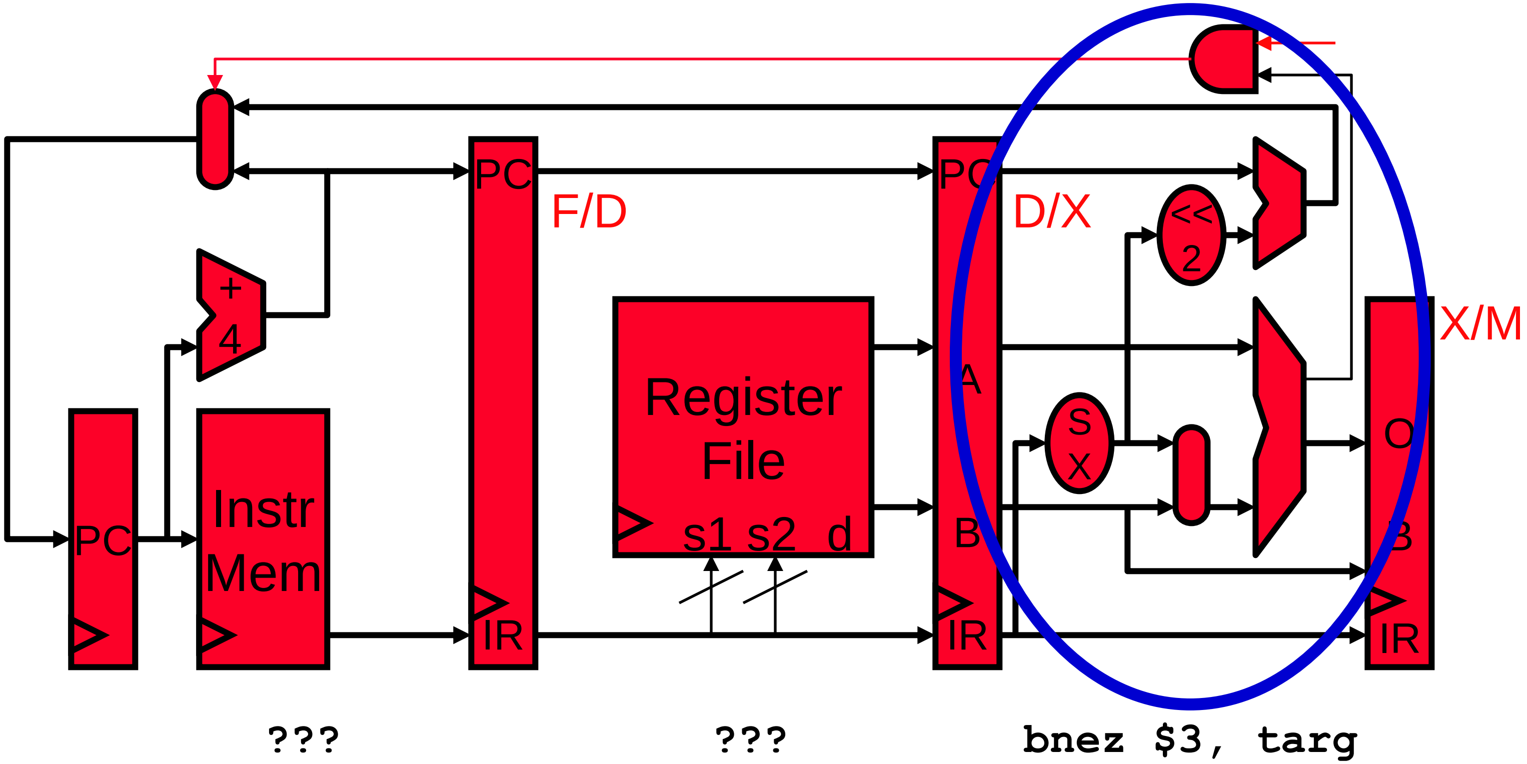
Control Dependences and Branch Prediction

Big idea: 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 instr’s
    - – Must undo incorrect changes (recover to pre-speculation state)
  - ● The “game”: **[$\%_{correct}$ * gain] – [(1–$\%_{correct}$) * penalty]**

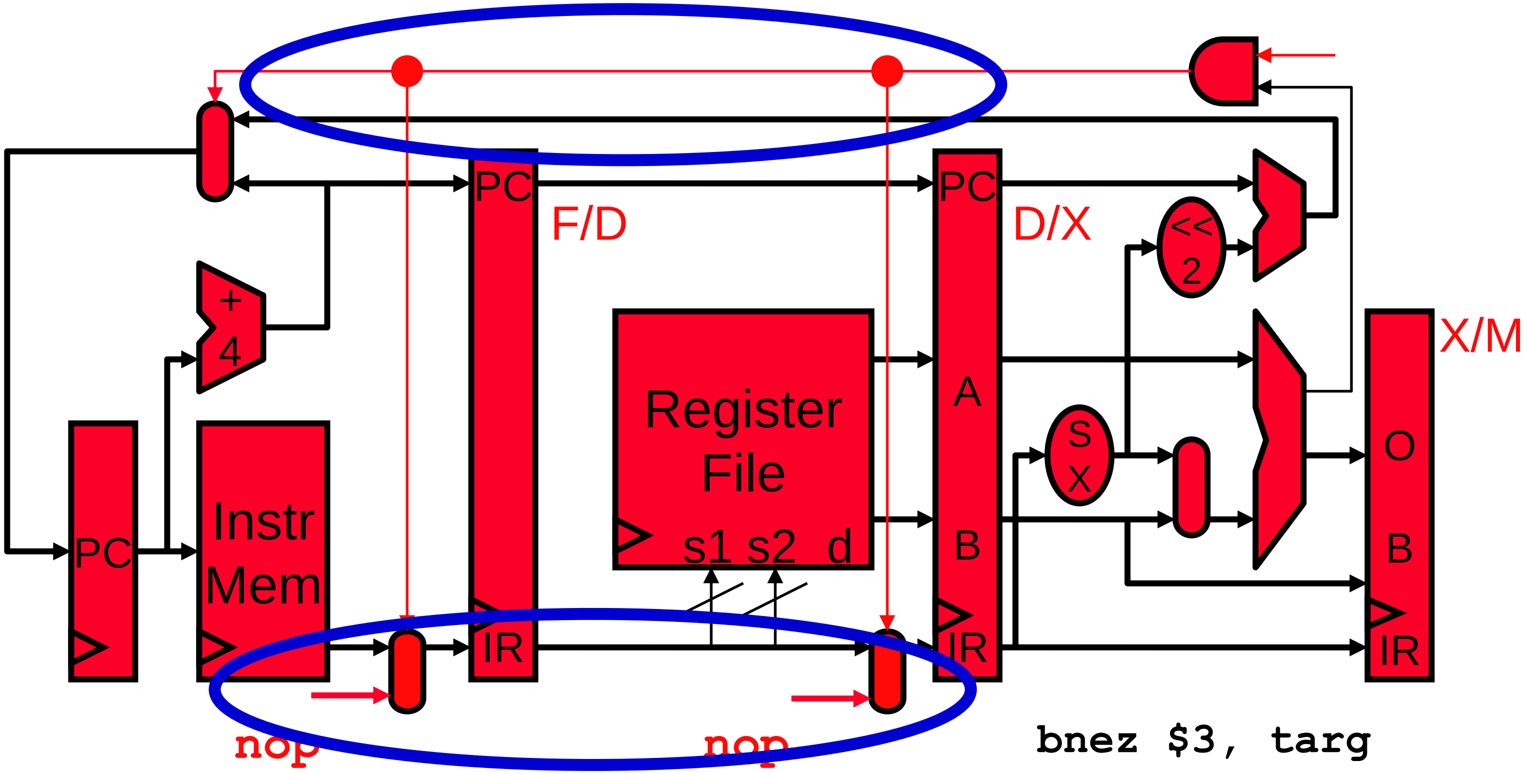
- ❑ **Control speculation**: speculation aimed at control hazards
  - Unknown parameter: are these the correct instr’s to execute next?

One control speculation option



- Instead of just stalling to wait for branch outcome (an automatic two-cycle penalty)
  - **Fetch past branch instr's before the branch outcome is known**
    - Default: assume "**not-taken**" (at fetch, can't tell it's a branch)

Mis-speculation branch recovery



- **Branch recovery**: what to do when branch is actually taken
  - Instr’s that will be written into F/D and D/X are wrong
    - **Flush them**, i.e., replace them with `nops`
  - + They haven’t written permanent state yet (i.e., regfile, DMem)
  - – Two cycle penalty for taken branches

Control speculation and recovery

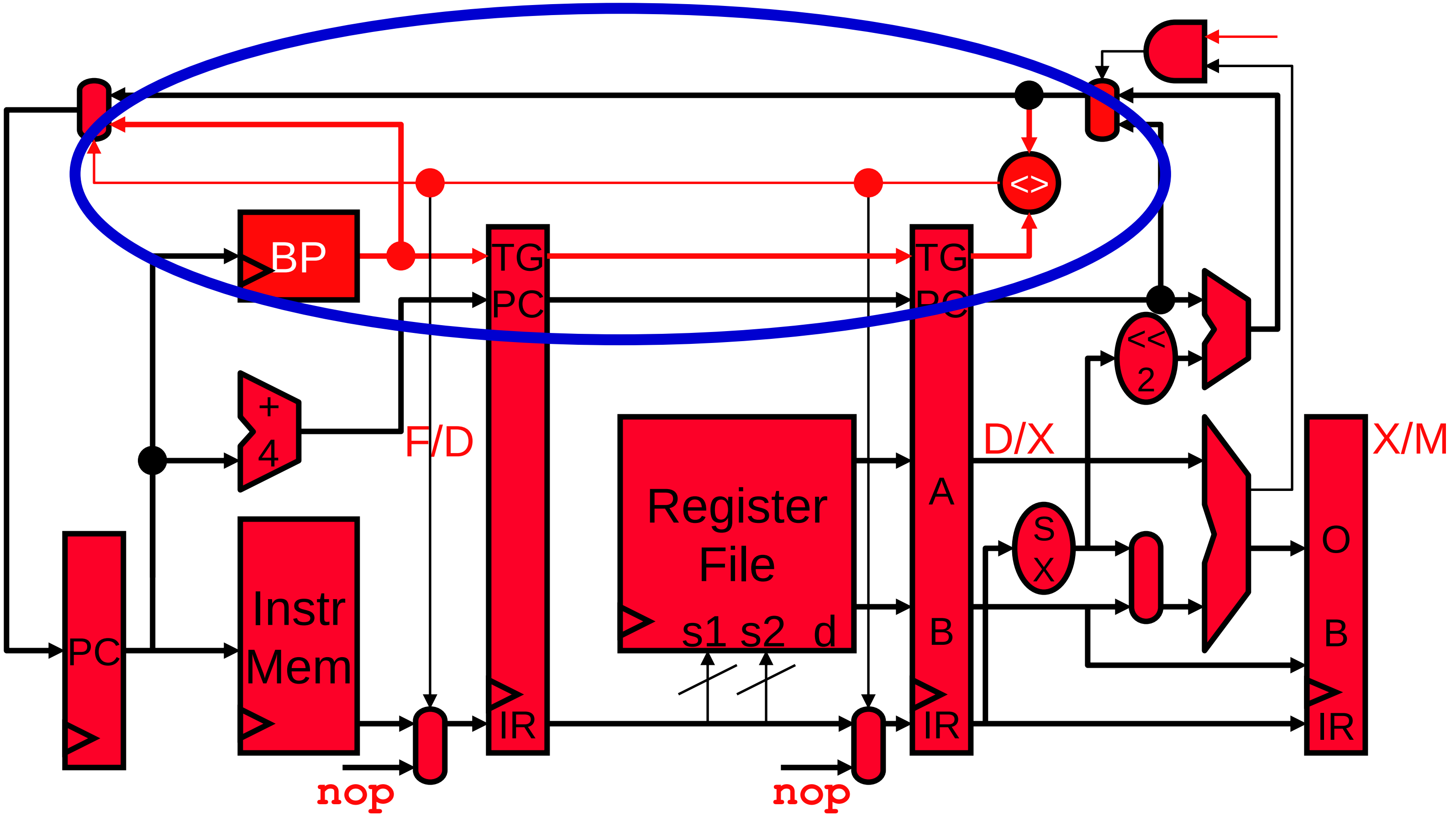
		1	2	3	4	5	6	7	8	9
Correct:	addi $1,1➔$3	F	D	X	M	W				
	bnez $3,targ		F	D	X	M	W			
	sw $6➔[$7+4]			**F**	**D**	X	M	W		
	targ:add $4,$5➔$4				**F**	D	X	M	W	

speculative

- **Mis-speculation recovery**: what to do on wrong guess
  - Not too painful in an in-order pipeline
  - Remember, our branches resolve in X

		1	2	3	4	5	6	7	8	9
	addi $1,1➔$3	F	D	X	M	W				
	bnez $3,targ		F	D	**X** C	M	W			
Recovery:	~~sw $6➔[$7+4]~~			**F**	**D**	--	--	--		
	~~targ:add $4,$5➔$4~~				**F**	--	--	--	--	
	targ:add $4,$5➔$4					**F**	D	X	M	W

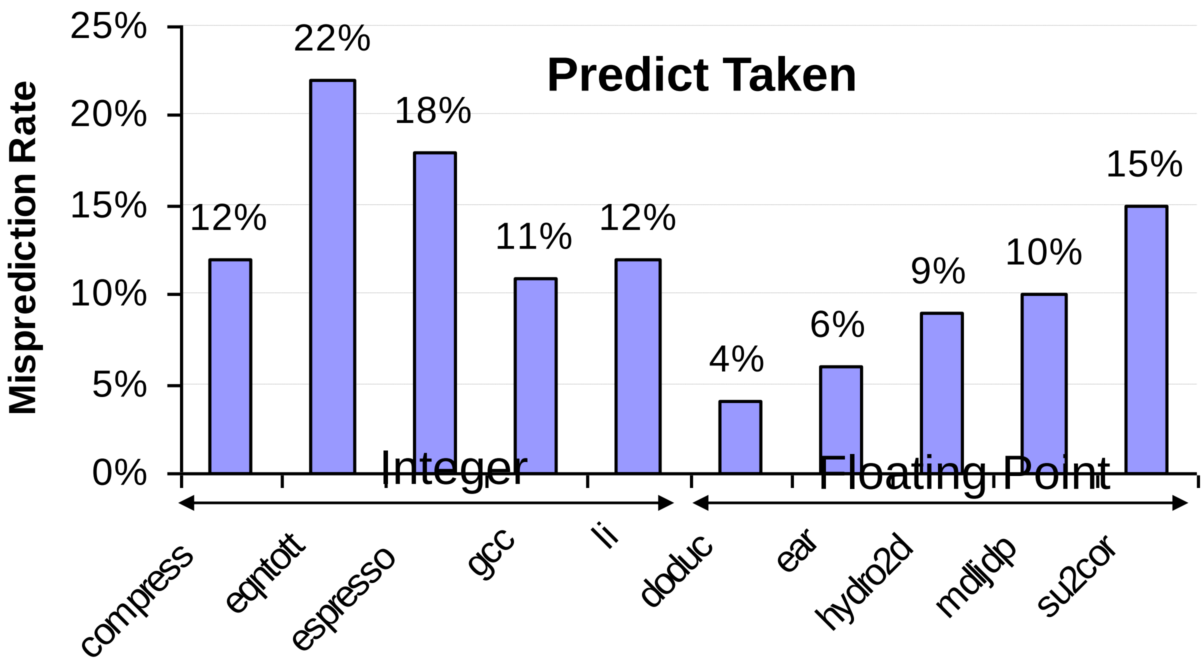
Static branch prediction



- **Static branch prediction**: always guess the same
  - Start fetching from guessed address (not taken **or** taken)
  - Flush on **mis-prediction** and restart pipeline at correct instr

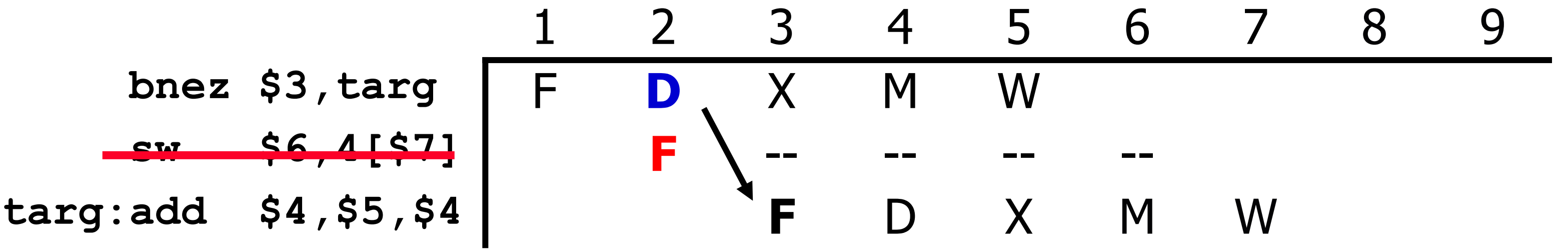
Static branch prediction performance

- Back of the envelope calculation
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%; static **predict not taken** strategy
  - Say, **75% of branches are taken**
- CPI = 1 + 20% * 75% * 2 = 1 + **0.20 * 0.75 * 2** = 1.3
  - **Branches cause 30% slowdown**
  - Even worse with deeper pipelines
- How do we reduce this penalty?



Moving branch decision earlier in the pipeline

- **Fast branch**: targets control-hazard penalty
  - Basically, have branch instr’s that can resolve in D, not X
    - - Test must be comparison to zero or equality, **no time for ALU op**
  - + New taken branch penalty is 1 (as compared to 2 before)
    - + On taken branch, must flush one instr and “bypass” from the decode stage
  - – Must now have additional comparison instr’s (e.g., **`cmplt`**, **`slt`**) to support complex tests

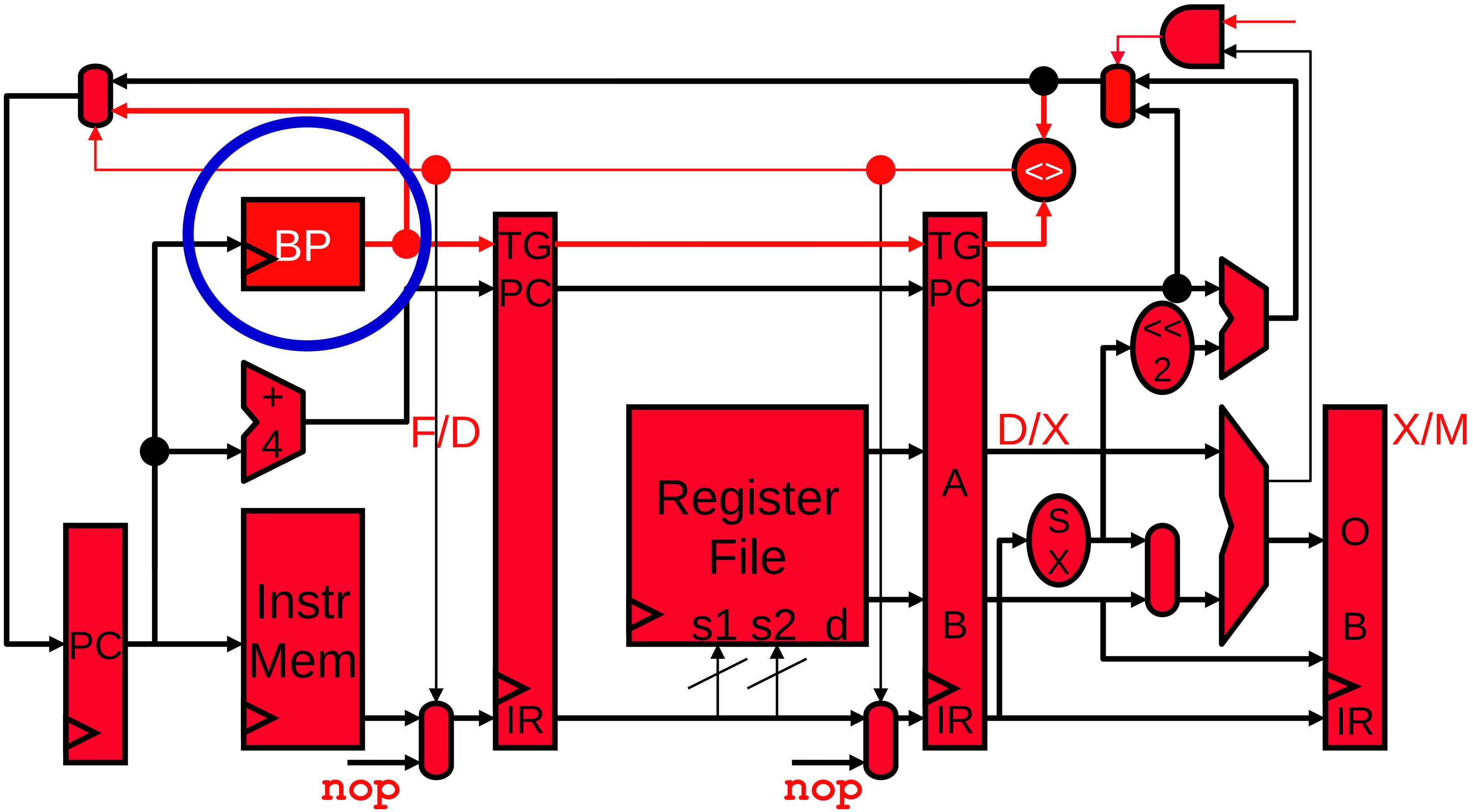


	1	2	3	4	5	6	7	8	9
`bnez $3,targ`	F	**D**	X	M	W				
~~**`sw $6,4[$7]`**~~		**F**	--	--	--	--			
`targ:add $4,$5,$4`			**F**	D	X	M	W		

Fast branch performance

- Assume: Branch: 20%, 75% of branches are taken
  - CPI = 1 + 20% * 75% * **1** = 1 + **0.20*0.75*1** = **1.15**
    - **15% slowdown** (better than the 30% from before)
- But wait, fast branches assume only simple comparisons
  - Fine for MIPS
  - But not fine for ISAs with "branch if \$1 > \$2" operations
- In such cases, say 25% of branches require an extra instruction
  - CPI = 1 + (20% * 75% * 1) + 20%*25%*1(extra instr) = **1.2**
- Example of ISA and micro-architecture interaction
  - Type of branch instructions
  - Another option: "Delayed branch" or "branch delay slot"
  - What about condition codes?

Dynamic branch prediction

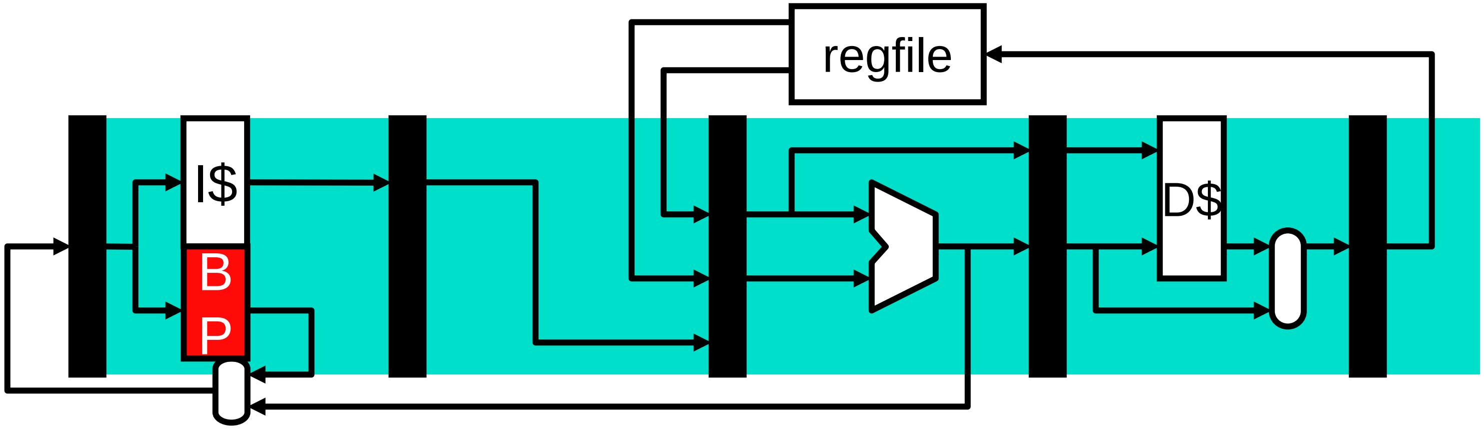


- **Dynamic branch prediction**: hardware guesses outcome dynamically based on collected profile information
  - Start fetching from guessed address
  - Flush on **mis-prediction** and restart pipeline at correct instr

Branch prediction performance

- Parameters
  - **Branch: 20%**, load: 20%, store: 10%, other: 50%; 2 cycle stall penalty on misprediction
- Static branch prediction (predict taken)
  - Average misprediction rate for SPEC is 34% (ranges from 59% to 9%)
  - CPI = 1 + 20% * 34% * 2 = **1.136**
- Dynamic (profile based) branch prediction
  - Assume branches can be predicted with 91% accuracy
  - CPI = 1 + 20% * 9% * 2 = **1.036**
  - How about more advanced predictors with 95% accuracy?
  - CPI = 1 + 20% * 5% * 2 = **1.02**
  - What if we could achieve 98% accuracy?
  - CPI = 1 + 20% * 2% * 2 = **1.008**

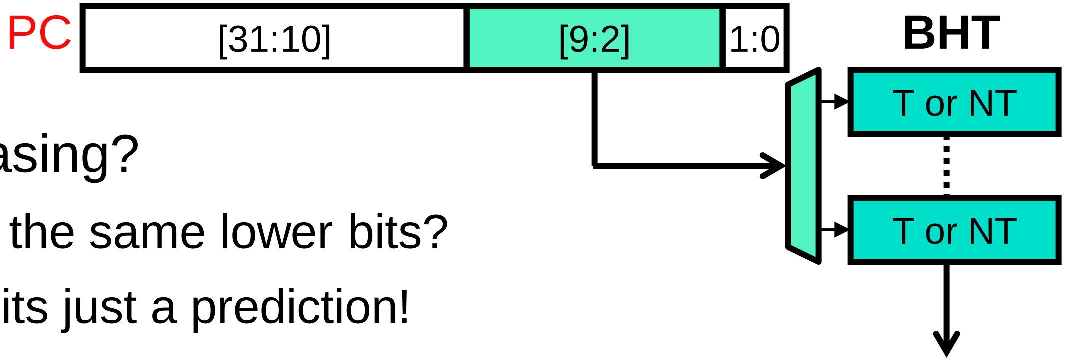
Dynamic branch prediction components



- Step #1: is it a branch?
  - Easy after decode, but ...
- Step #2: is the branch taken or not taken?
  - **Direction predictor** (applies to conditional branches only)
  - Predicts taken/not-taken
- Step #3: if the branch is taken, where does it go?
  - Easy after decode, but …

Step #2: Branch direction prediction

- Record the past in a hardware structure – a "direction predictor"
  - Map conditional-branch PC to taken/not-taken (T/NT) decision
  - Individual conditional branches often biased or weakly biased
    - 90%+ one way or the other considered **"biased"**
    - Why? Loop back edges, checking for uncommon conditions
- **Branch history table (BHT)**: simplest direction predictor
  - PC indexes table of bits (0 = NT (Not Taken), 1 = T), no tags
  - Essentially: branch will go same way it went last time (so have to update BHT)



  - What about aliasing?
    - Two PC with the same lower bits?
    - No problem, its just a prediction!

BHT (single bit prediction) shortcomings

- Problem: consider **inner loop branch** below (* = mis-prediction, starting state of N)

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

- conditional branch at the bottom of the loop code, so taken except when exiting the loop

State/prediction	**N***											
Branch outcome	T	T	T	N	T	T	T	N	T	T	T	N

- Two "built-in" mis-predictions per inner loop iteration
  - Since for-loop branches are at the bottom of the loop, it mispredicts when it enters the loop and when it exits the loop
- Branch predictor "changes its mind too quickly"

Two-bit saturating counters (2bc)

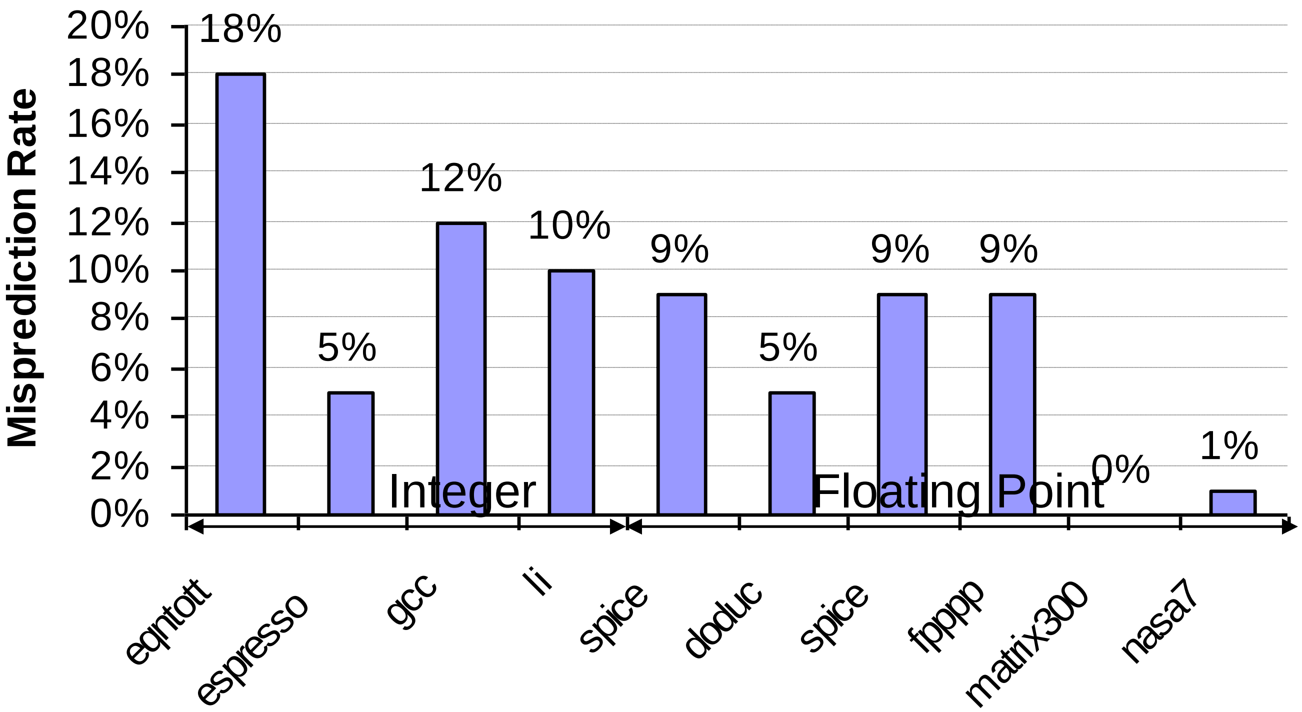
- **Two-bit saturating counters (2bc)** [Smith, 1981]
  - Replace each single-bit prediction with a 2-bit predictor
    - (0,1,2,3) = (N,n,t,T)
  - Adds “hysteresis”
    - Force predictor to mis-predict twice before “changing its mind”

State/prediction	**N***	**n***	**t**	**T***	**t**	**T**	**T**	**T***	**t**	**T**	**T**	**T***
Branch outcome	T	T	T	N	T	T	T	N	T	T	T	N

- One mispredict each loop execution (rather than two)
  - + Fixes this pathology (which is not contrived, by the way)
  - - Can we do even better?
- Note that this is a **different** FSM 2bc than the one in the book

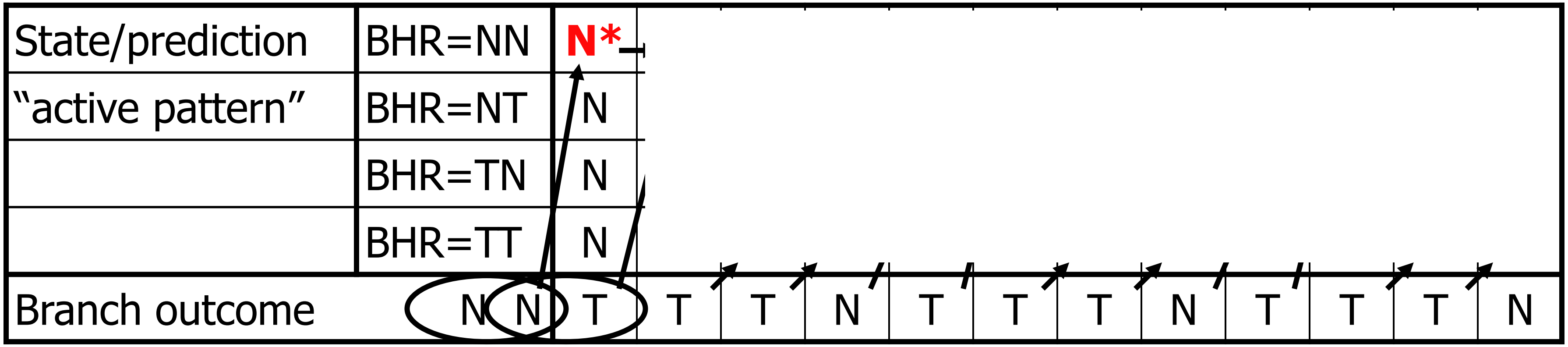
2bc performance

- Mispredicts because either
  - Wrong guess for that branch
  - Got branch history of wrong branch when indexing the table
- For a 4096 entry table (SPEC89)



Correlated branch predictor

- **Correlated (two-level) predictor** [Yeh&Patt, 1992]
  - Exploits observation that branch outcomes are correlated
  - Maintains separate prediction tables (BHT) per (PC, BHR)
    - **Branch history register (BHR)**: recent branch outcomes for that branch
  - Simple working example: assume program has **one** branch
    - BHT: one 1-bit entries
    - BHT+**2-bit BHR**: 2^2 = **4** 1-bit direction prediction entries



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

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

Correlated branch predictor “fix”

- What happened?
  - BHR wasn’t long enough to capture the “active pattern” (see yellow filled BHR labels)
  - Try again: BHT+**3-bit BHR**: 2^3 = **8** 1-bit direction prediction entries

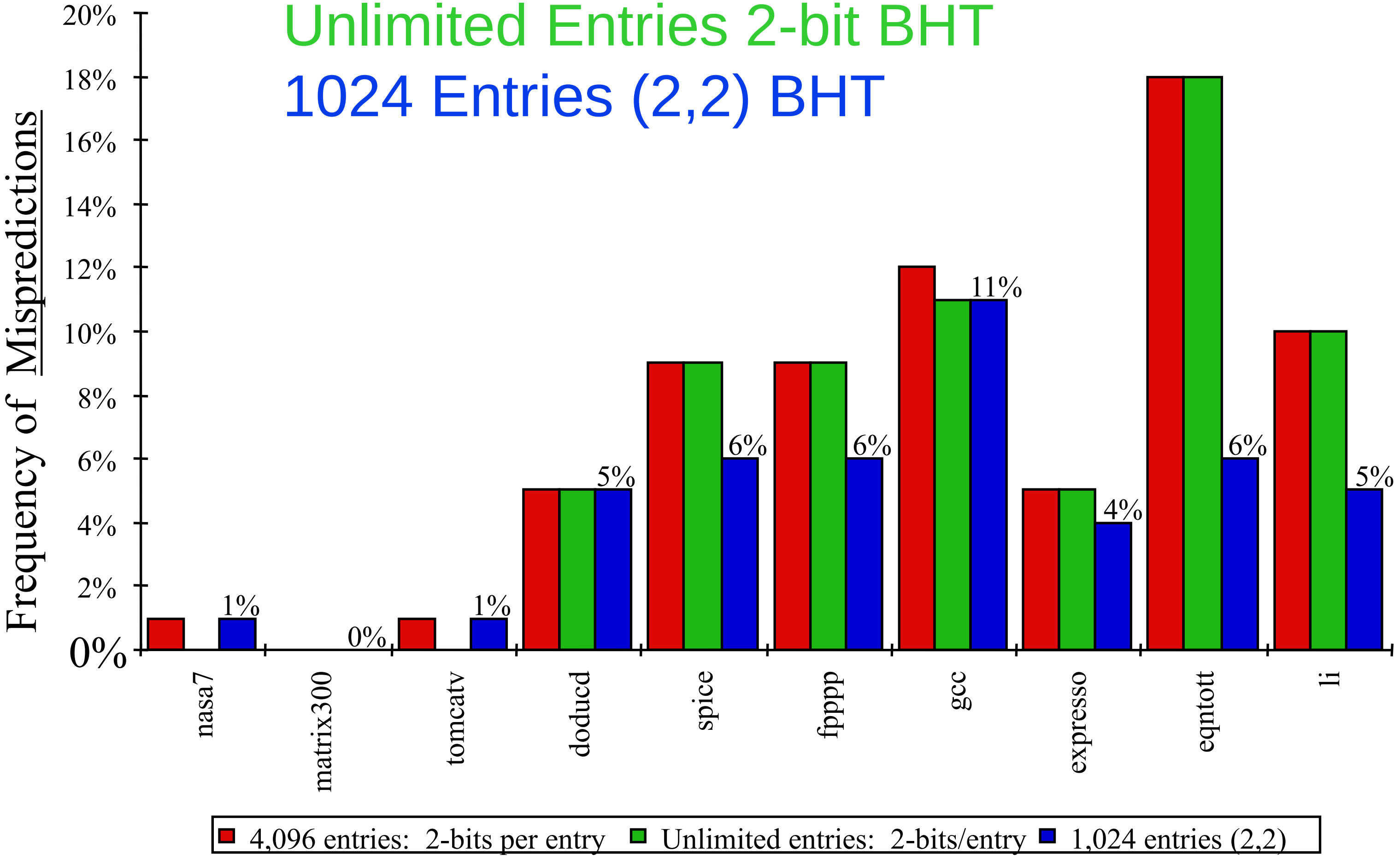
State/prediction	BHR=NNN	N*	T	T	T	T	T	T	T	T	T	T	T
	BHR=NNT	N	N*	T	T	T	T	T	T	T	T	T	T
	BHR=NTN	N	N	N	N	N	N	N	N	N	N	N	N
	BHR=NTT	N	N	N*	T	T	T	**T**	T	T	T	**T**	T
	BHR=TNN	N	N	N	N	N	N	N	N	N	N	N	N
	BHR=TNT	N	N	N	N	N	N*	T	T	T	**T**	T	T
	BHR=TTN	N	N	N	N	N*	T	T	T	**T**	T	T	T
	BHR=TTT	N	N	N	**N**	N	N	N	**N**	N	N	N	**N**
Branch outcome	N N N	T	T	T	N	T	T	T	N	T	T	T	N

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

Correlated branch predictor tradeoffs

- Design choice I: one **global** BHR or one per PC (**local**)?
  - Each one captures different kinds of patterns
  - Global takes less hardware (one register) & can capture local patterns for tight loop branches
  - Local takes more hardware (usually as a cache) but better prediction rates
- Design choice II: how many history bits (BHR size)?
  - + Given unlimited resources, longer BHRs are better, but…
  - – BHT utilization decreases
    - – Many history patterns are never seen
    - – Many branches are history independent (i.e., don't cares)
  - – Predictor takes longer to train
  - Typical length: 8 to 12 (so 2^8 to 2^{12} entries in the BHT)
- Design choice III: how many bits per BHT entry
  - Easy one - usually 1 (last-time) or 2 (saturating counter)

Comparison of 2bc and correlated prediction



4096 Entries 2bc (0,2) BHT

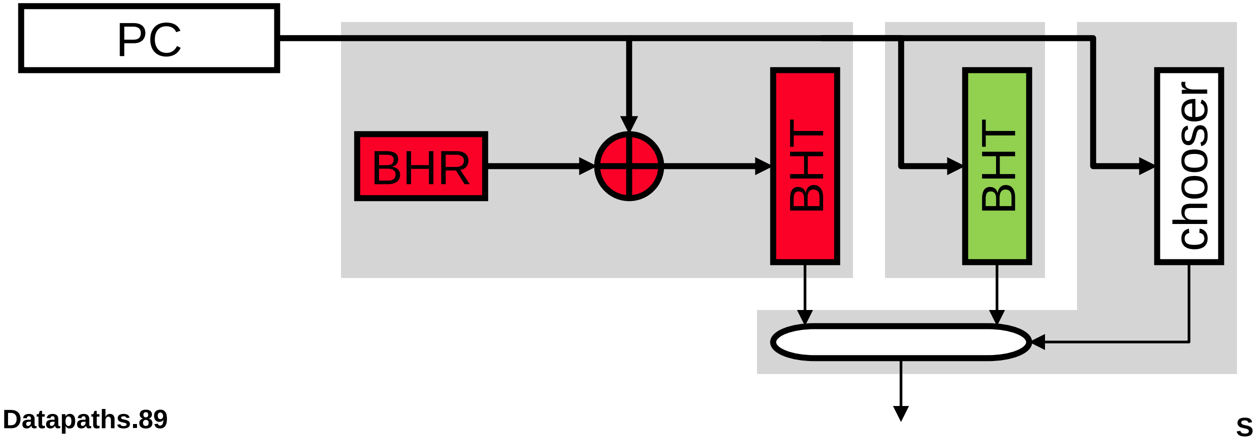
Unlimited Entries 2-bit BHT

1024 Entries (2,2) BHT

no history in first two

Aside: Hybrid predictor

- **Hybrid (tournament) predictor** [McFarling]
  - Attacks correlated predictor BHT capacity 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 on mis-prediction threshold
  - + Correlated predictor can be made **smaller**, handles fewer branches
  - + 90–95% accuracy



Step #1: When to perform branch prediction?

- During Decode
  - Look at instruction opcode to determine branch instructions
  - Can calculate next PC from instruction (for PC-relative branches)
  - One cycle “mis-fetch” penalty **even if branch predictor is correct**

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

- During Fetch
  - How do we do that ?
    - We are already doing the branch prediction during the F cycle
  - If we can do that (and the prediction is correct) then there is no branch penalty !

Step #3: Branch Target Buffer (BTB)

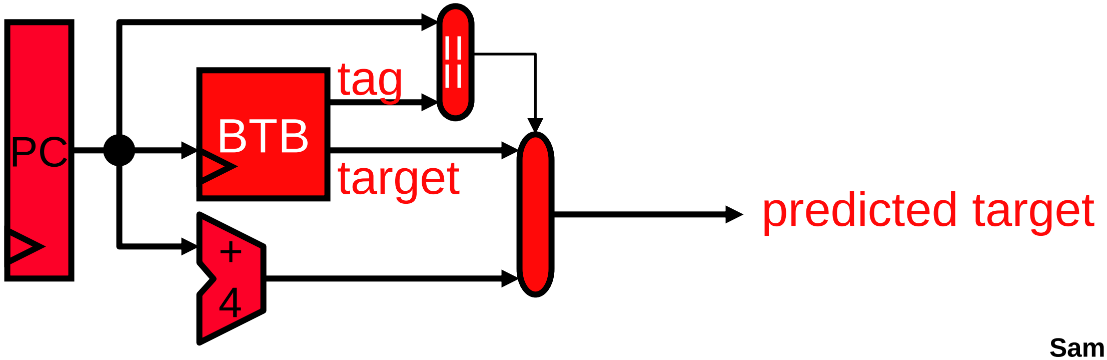
- As before: learn from past, predict the future
  - Record the past branch targets in a hardware structure

- **Branch target buffer (BTB)**:
  - “guess” the future PC based on past behavior
  - “Last time the branch X was taken, it went to address Y”
    - “So, in the future, if address X is fetched, fetch address Y next”

- Operation
  - Like a cache: address = PC, data = target-PC
  - Access during Fetch *in parallel* with instruction memory
    - predicted-target = BTB[PC]
  - Updated during X whenever target != predicted-target
    - BTB[PC] = target
  - Aliasing? No problem. As before, this is only a prediction

BTB, continued

- At Fetch, how does instr know it's a branch & should read BTB?   It doesn't have to…
  - **…**all instr's access BTB in parallel with Imem Fetch
- Key idea: **also use BTB to predict which instr's are branches**
  - Implement by "tagging" each entry with its corresponding PC
  - Update BTB on every taken branch instr, record target PC:
    - BTB[PC].tag = PC, BTB[PC].target = target of branch
  - All instr's access during Fetch *in parallel* with Imem
    - Check for tag match, signifies instr at that PC is a branch
    - Predicted PC = (BTB[PC].tag == PC) ? BTB[PC].target : PC+4

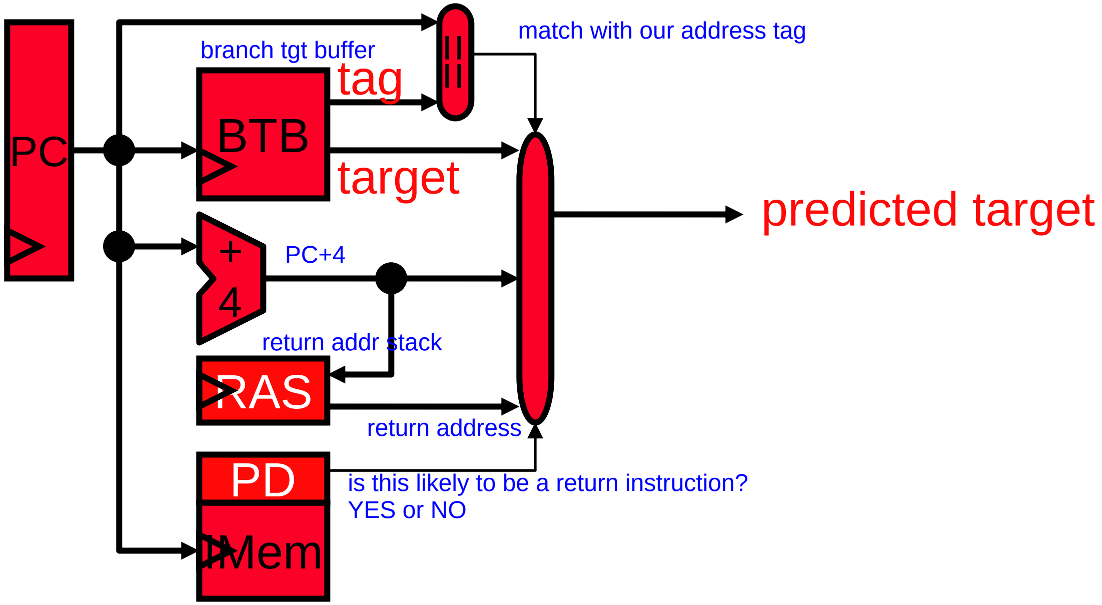


Why does a BTB work?

- Because most control instr's use **direct targets**
  - Target encoded in instr itself → same "taken" target every time

- What about **indirect targets**?
  - Target held in a register → can be different each time
  - Indirect conditional jumps are not widely supported
  - Two indirect call idioms
    - + Dynamically linked functions (DLLs): target always the same
    - - Dynamically dispatched (virtual) functions: hard but uncommon
  - Also two indirect unconditional jump idioms
    - - Switches: hard but uncommon
    - – Function returns: hard and common but…

Return Address Stack (RAS)

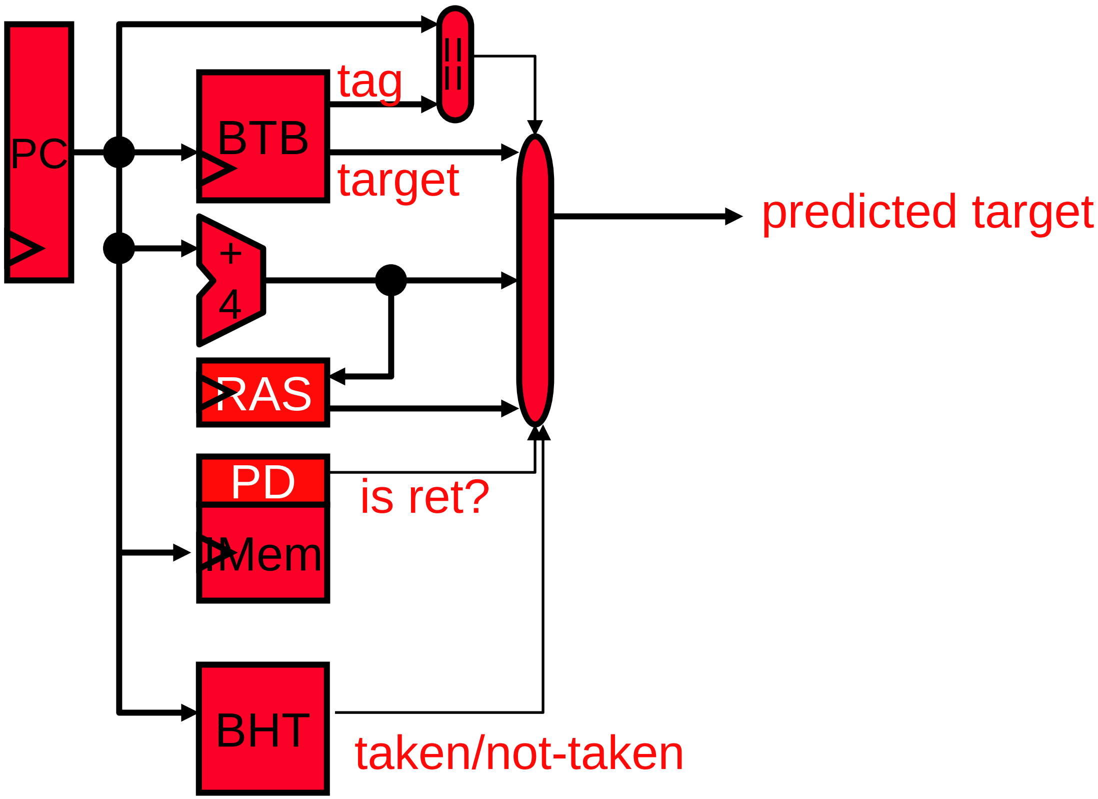


❑ Return address stack (RAS)

- Call instruction? RAS[TOS++] = PC+4
- Return instruction? Predicted-target = RAS[--TOS]
- Q: how can you tell if an instr is a call/return before decoding it?
  - Accessing RAS on every instr BTB-style doesn’t work
- Answer: **pre-decode bits** in Imem, written when first executed
  - Can also be used to signify branches

Putting it all together

- BTB & branch direction predictor during Fetch



- If branch prediction correct, no taken branch penalty

Why is all this still so important?

- For deep (high performance) pipelines, typical mis-prediction penalties are 10+ cycles
    - Big impact on CPI

- For in-order superscalar pipelines misprediction can degrade IPC (as we will see in the next lecture)
- For out-of-order (superscalar) pipelines, many instructions past the branch are being executed (but not committed until the branch outcome is determined) and if the branch was mis-predicted all of that work has to be thrown away (wasting performance and energy)
    - Big impact on IPC

Avoiding branches via ISA: Predication

- Conventional control
  - Conditionally executed instr's also conditionally fetched

	1	2	3	4	5	6	7	8	9
`beq $3,targ`	F	D	X	M	W				
`sub $6,1,$5`		F	D	--	--	--	flushed: wrong path		
`targ:add $4,$5,$4`			F	--	--	--	--	flushed: why?	
`targ:add $4,$5,$4`				F	D	X	M	W	

- If **beq** mis-predicts, both **sub** and **add** must be flushed
- Waste: **add** is independent of mis-prediction

- **Predication**: not prediction, predic**a**tion
  - ISA support for conditionally-executed unconditionally-fetched instr's
  - If **beq** mis-predicts, annul **sub** in place, preserve **add**
    - Example is if-then, but if-then-else can be predicated too
  - How is this done? How does **add** get correct value for **r5**

Full predication

- **Full predication**
  - Every instr can be annulled, annulment controlled by…
  - Predicate registers: additional register in each instr (e.g., IA64)

	1	2	3	4	5	6	7	8	9	
`setp.eq $3,p3`	F	D	X	M	W					
`sub.p $6,1,$5,p3`		F	D	X	--	--				annulled
`targ:add $4,$5,$4`			F	D	X	M	W			

  - Predicate codes: condition bits in each instr (e.g., ARM)

	1	2	3	4	5	6	7	8	9	
`setcc $3`	F	D	X	M	W					
`sub.nz $6,1,$5`		F	D	X	--	--				annulled
`targ:add $4,$5,$4`			F	D	X	M	W			

  - Only ALU instr shown (`sub`), but this applies to all instr's, even stores
  - Branches replaced with "set-predicate" instr's

Conditional register moves (CMOVs)

Conditional (register) moves

- Construct appearance of full predication from one primitive

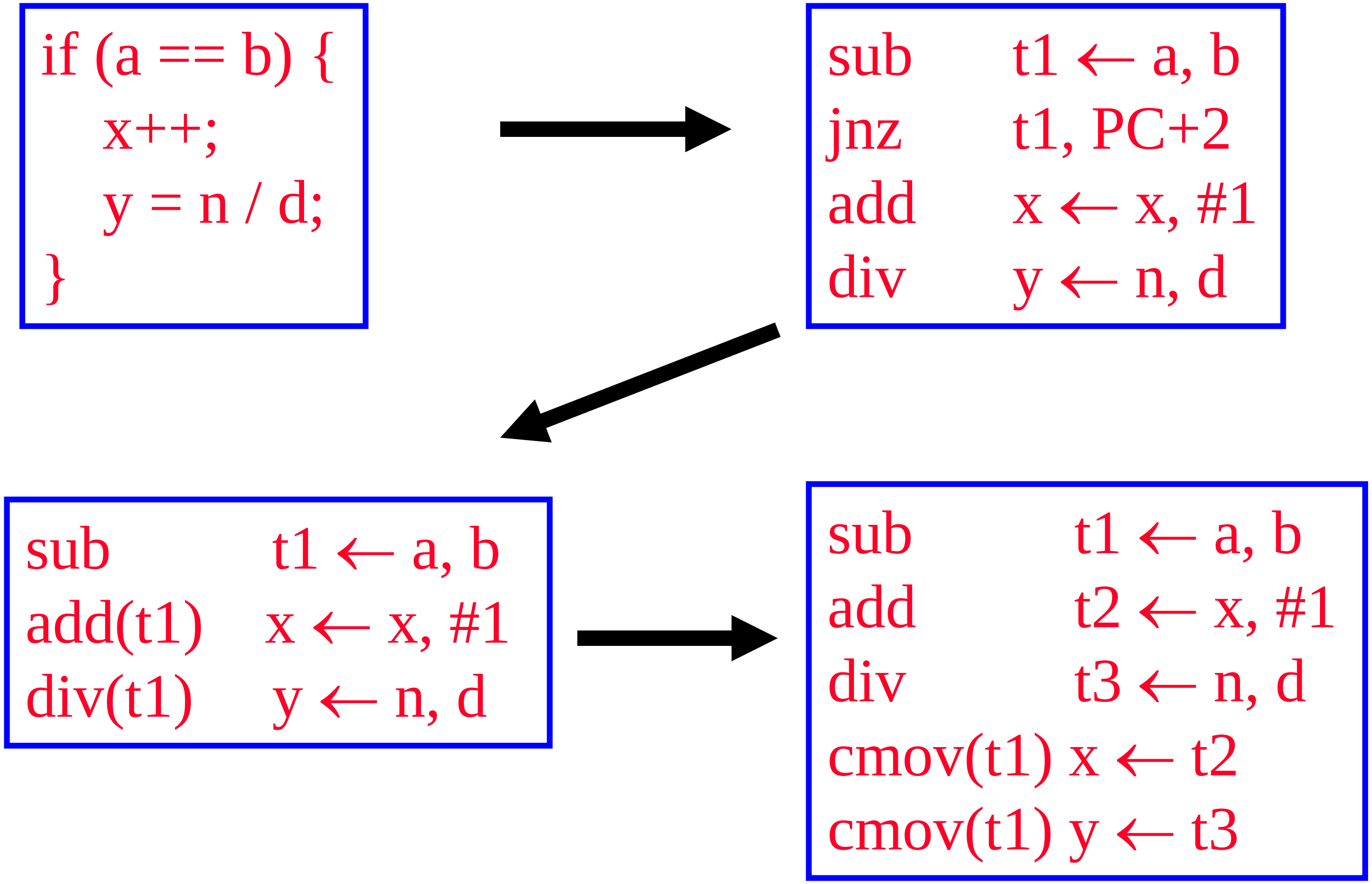
```
cmoveq $1,$2,$3          // if ($1==0) $3=$2;
```

  - May require some code duplication to achieve desired effect
  - Painful, potentially impossible for some instr sequences
  - Requires more registers
- Only good way of retro-fitting predication onto ISA (e.g., IA32, Alpha)

	1	2	3	4	5	6	7	8	9
`sub $6,1,$9`		D	X	M	W				
`cmovne $3,$9,$5`		F	D	X	M	W			
`targ:add $4,$5,$4`			F	D	X	M	W		

SKIPPED IN CLASS

If-conversion



```
if (a == b) {
    x++;
    y = n / d;
}
```

```
sub     t1 ← a, b
jnz     t1, PC+2
add     x ← x, #1
div     y ← n, d
```

```
sub         t1 ← a, b
add(t1)     x ← x, #1
div(t1)     y ← n, d
```

```
sub         t1 ← a, b
add         t2 ← x, #1
div         t3 ← n, d
cmov(t1) x ← t2
cmov(t1) y ← t3
```

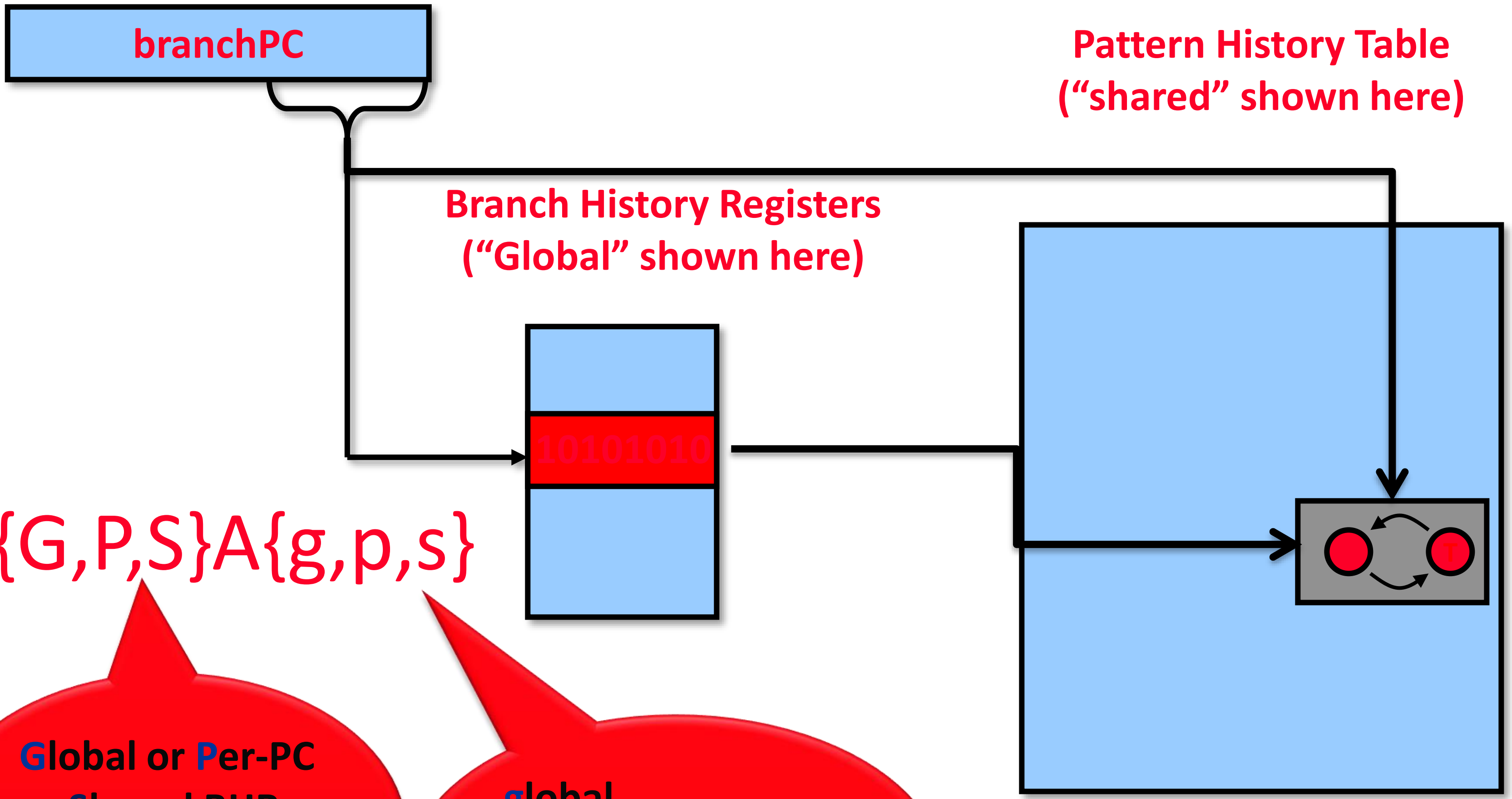
Predication performance

- Cost/benefit analysis
  - Benefit: predication avoids branches
    - Thus avoiding mis-predictions
    - Also reduces pressure on predictor table (few branches to track)
  - Cost: extra (annulled) instructions
- As branch predictors are highly accurate…
  - Might not help:
    - 5-stage pipeline, two instruction on each path of if-then-else
    - No performance gain, likely slower if branch predictable
  - Or even hurt!
  - But can help:
    - Deeper pipelines, hard-to-predict branches, and few added insn
- Thus, prediction is useful, but not a panacea

Discussion Prompts for F5

- “Two-level adaptive branch prediction” by Yeh and Patt
  - Q1: Yeh and Patt mention several static schemes in the Intro not discussed in class (e.g., based on the opcode, or the direction of the branch). Can they be implemented in the Fetch cycle?
  - Q2: Figure 2 shows 5 FSMs for setting the BHT bits, 2 were discussed in class and 1 presented in the book. Which are they? Which did they determine performs the best?
  - Q3: We only covered their GAg scheme in class (we assumed only one branch was encountered). Briefly describe the other two 2-level predictor variations, their pros and cons. Which scheme was the least cost if 97% accuracy is the goal?

Taxonomy & Nomenclature [Patt]

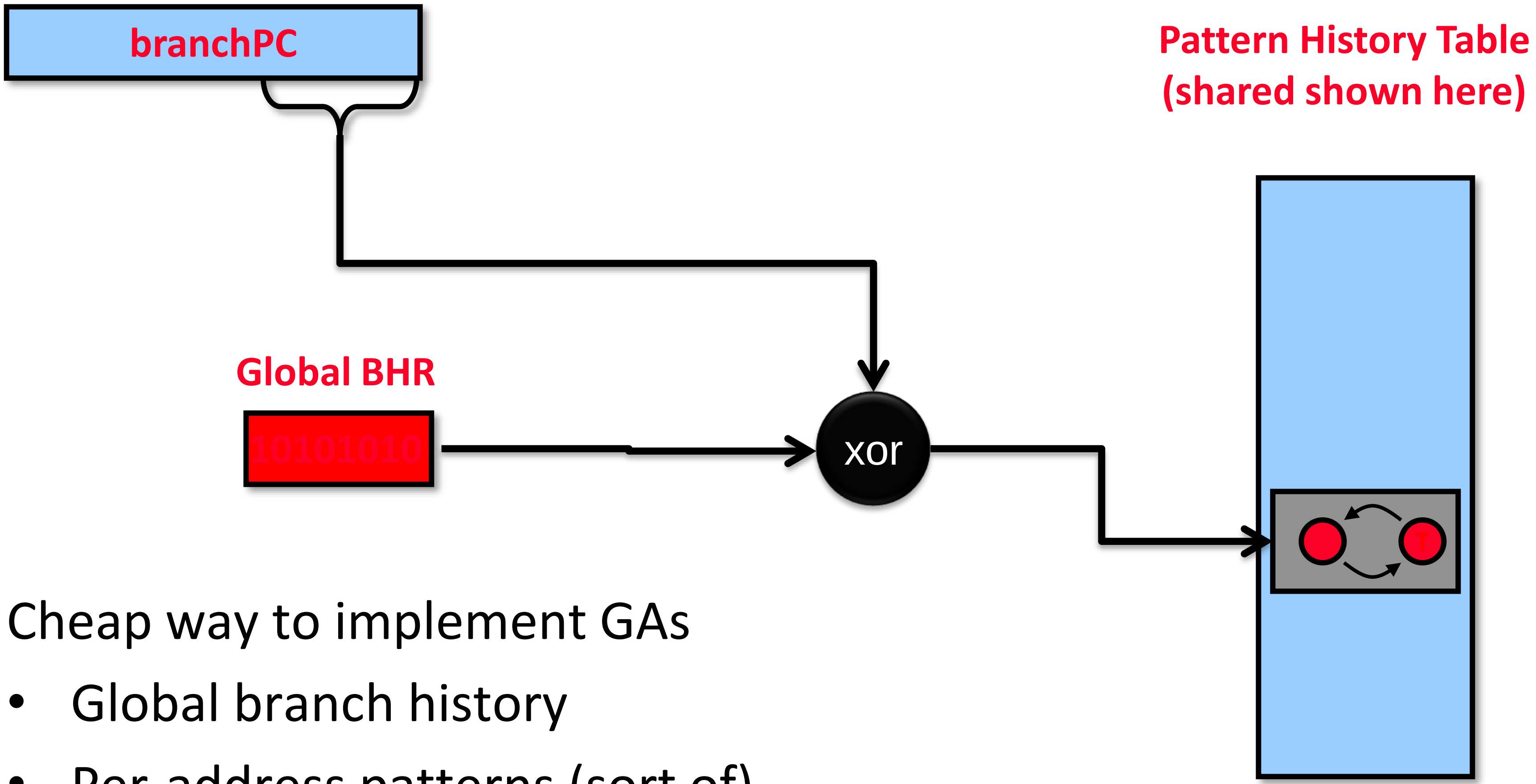


{G,P,S}A{g,p,s}

Global or Per-PC or Shared BHR

global, per-PC or shared PHT

Gshare [McFarling]



Cheap way to implement GAs

- Global branch history
- Per-address patterns (sort of)
- Try to make the PHT big enough to avoid collisions