CS152 – Computer Architecture and Engineering

Lecture 11 – Pipeline Control

2003-09-29

Dave Patterson

(www.cs.berkeley.edu/~patterson)

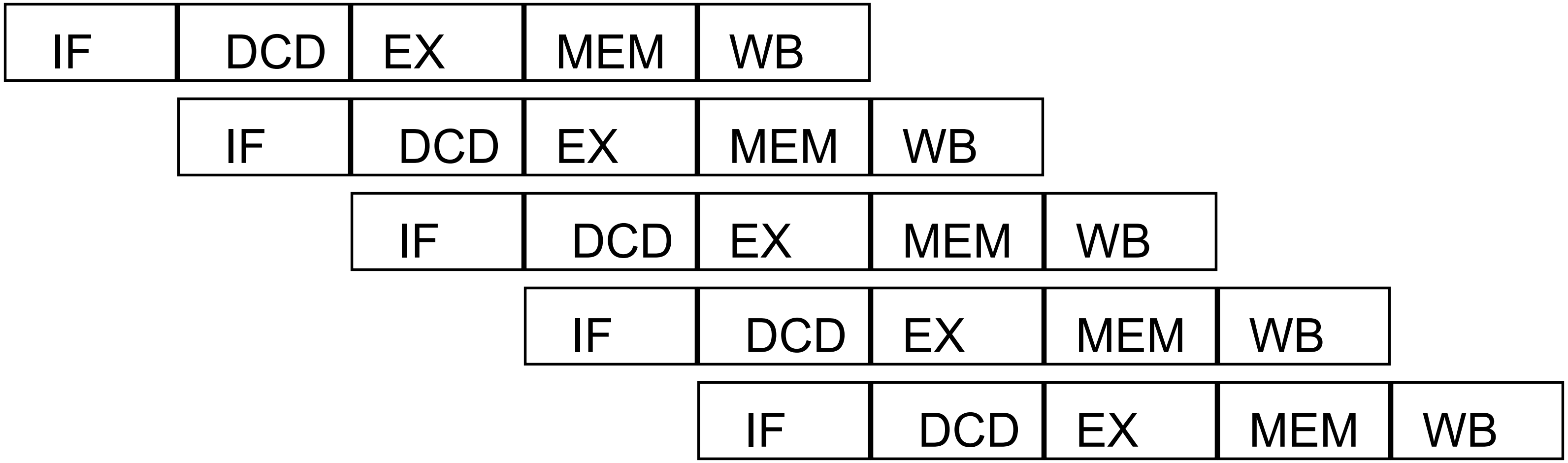
www-inst.eecs.berkeley.edu/~cs152/

Review: Pipelining

- Reduce CPI by overlapping many instructions
  - Average throughput of approximately 1 CPI with fast clock
- Utilize capabilities of the Datapath
  - start next instruction while working on the current one
  - limited by length of longest stage (plus fill/flush)
  - detect and resolve hazards
- What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- What makes it hard?
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction

Recap: Ideal Pipelining

Assume instructions are completely independent!



IF DCD EX MEM WB

IF DCD EX MEM WB

IF DCD EX MEM WB

IF DCD EX MEM WB

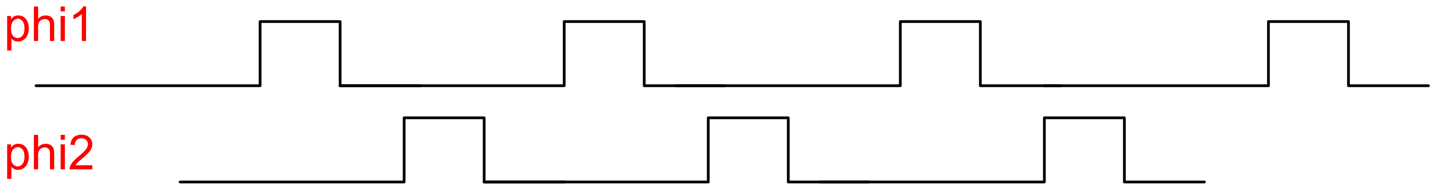
IF DCD EX MEM WB

Maximum Speedup ≤ Number of stages

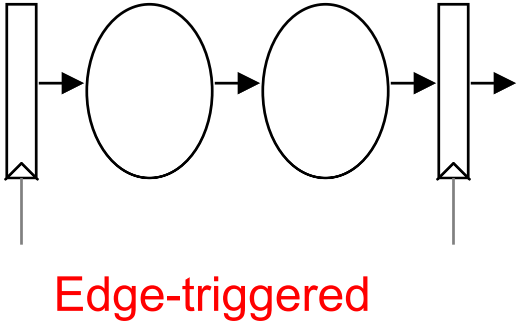
$$\text{Speedup} \leq \frac{\text{Time for unpipelined operation}}{\text{Time for longest stage}}$$

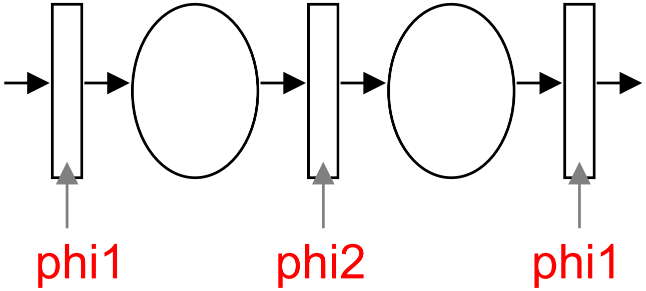
Example: 40ns data path, 5 stages, Longest stage is 10 ns, Speedup ≤ 4

FYI: MIPS R3000 clocking discipline

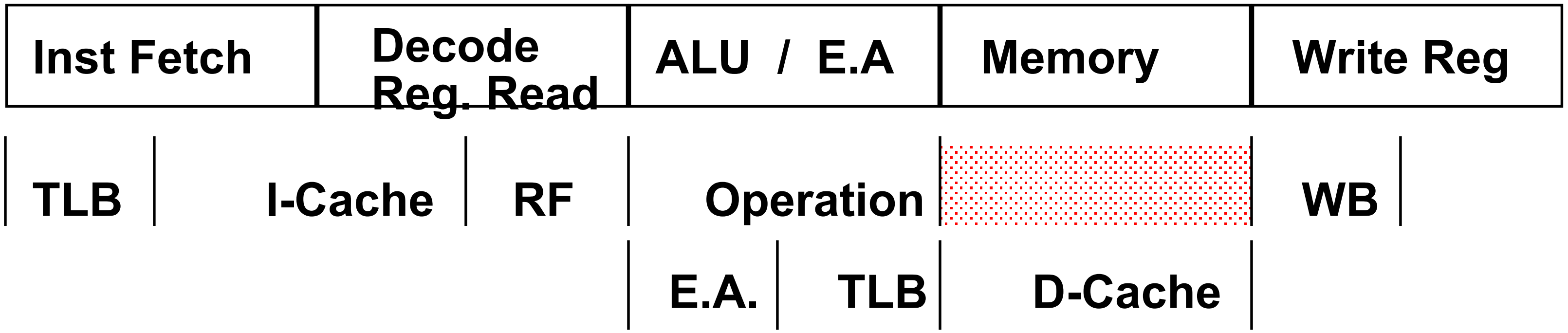


- 2-phase non-overlapping clocks
- Pipeline stage is two (level sensitive) latches





MIPS R3000 Instruction Pipeline



Inst Fetch | Decode Reg. Read | ALU / E.A | Memory | Write Reg

TLB | I-Cache | RF | Operation | WB

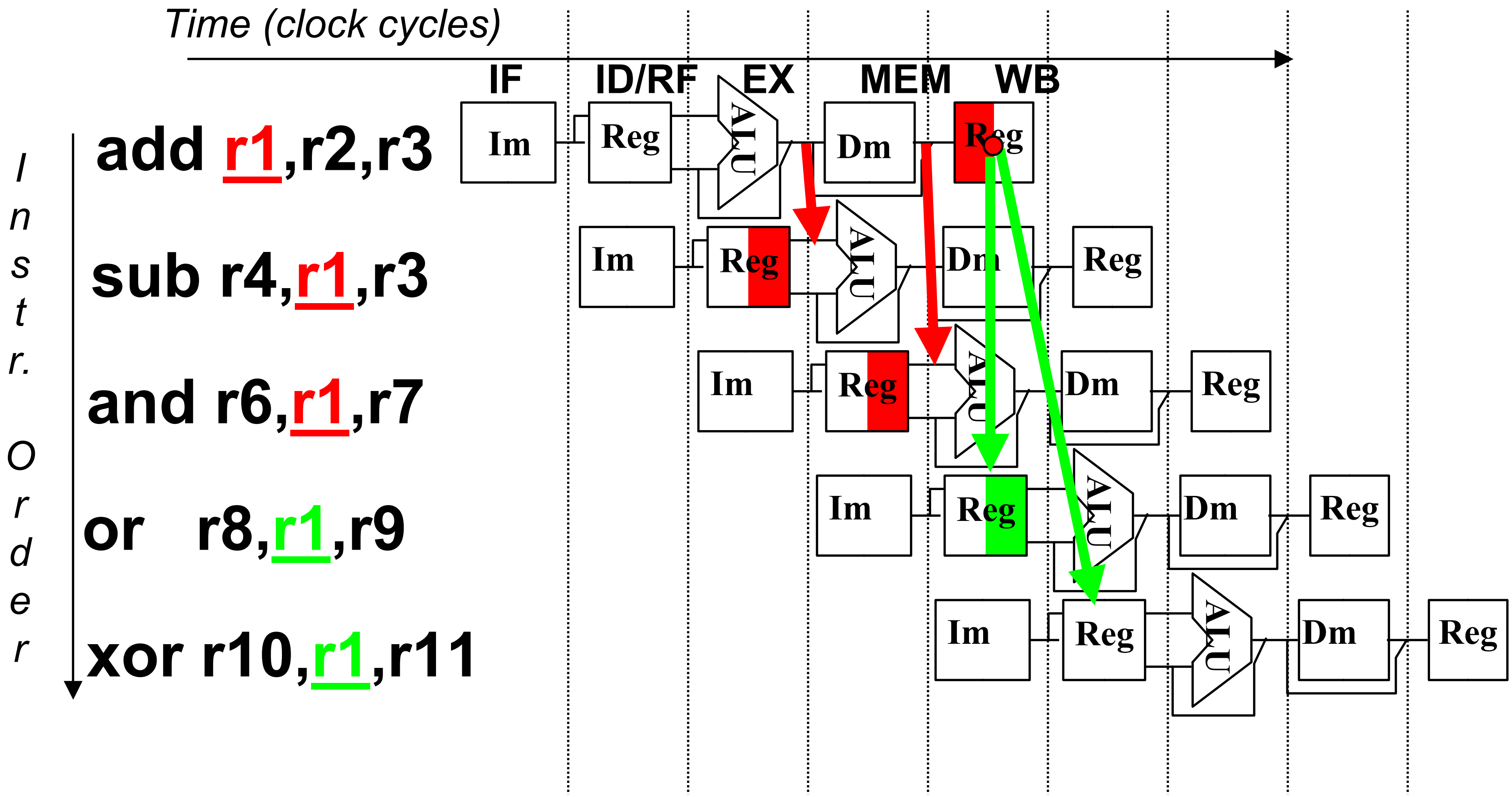
E.A. | TLB | D-Cache

Resource Usage

TLB					TLB										
	I-cache														
			RF					WB							
				ALU	ALU										
						D-Cache									

Write in phase 1, read in phase 2 => eliminates bypass from WB

Recall: Data Hazard on r1



Time (clock cycles)

IF ID/RF EX MEM WB

Instr. Order

add r1,r2,r3

sub r4,r1,r3

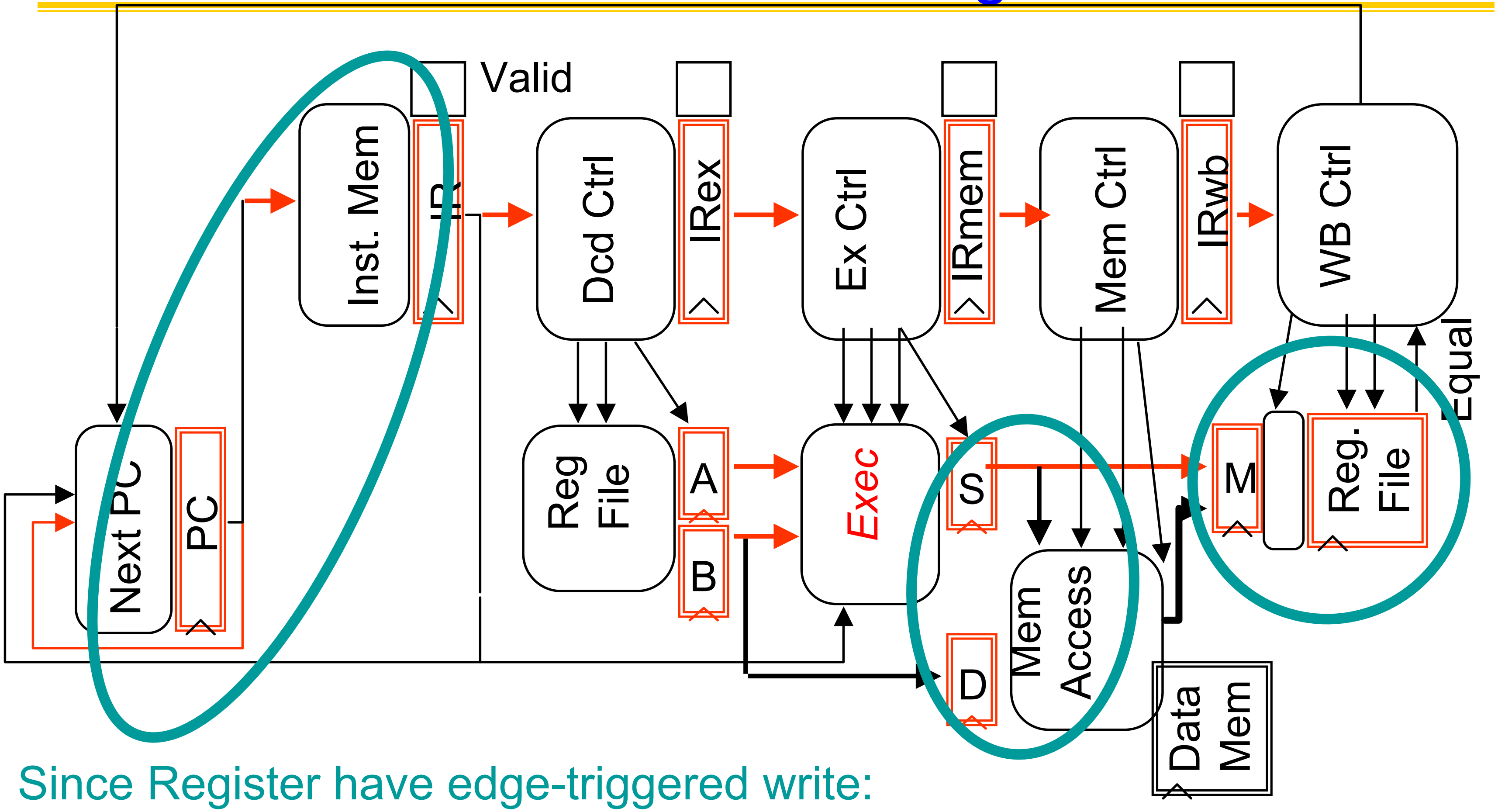
and r6,r1,r7

or r8,r1,r9

xor r10,r1,r11

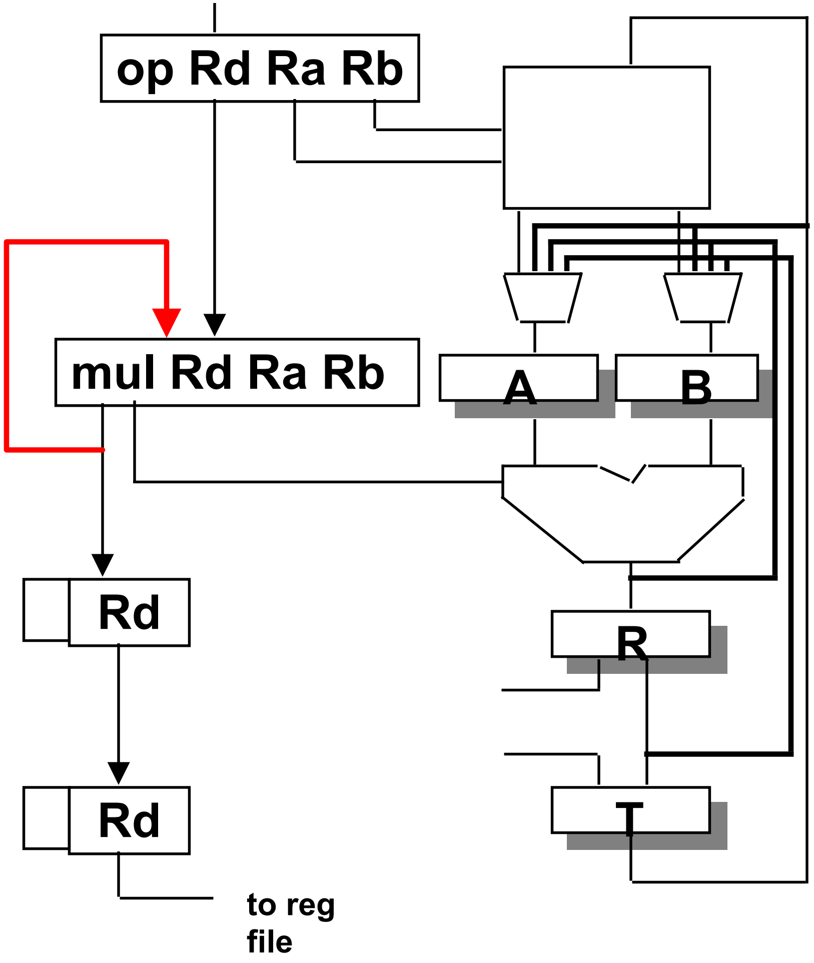
With MIPS R3000 pipeline, no need to forward from WB stage

Clarification about clock edges in lab4!



- Since Register have edge-triggered write:
  - Must have everything set up at *end* of memory stage
  - This means that "M" register here is not necessary!
- Also, Memories will be *synchronous*
  - Need to setup addresses and values in advance

MIPS R3000 Multicycle Operations



Use control word of local stage to step through multicycle operation

Stall all stages above multicycle operation in the pipeline

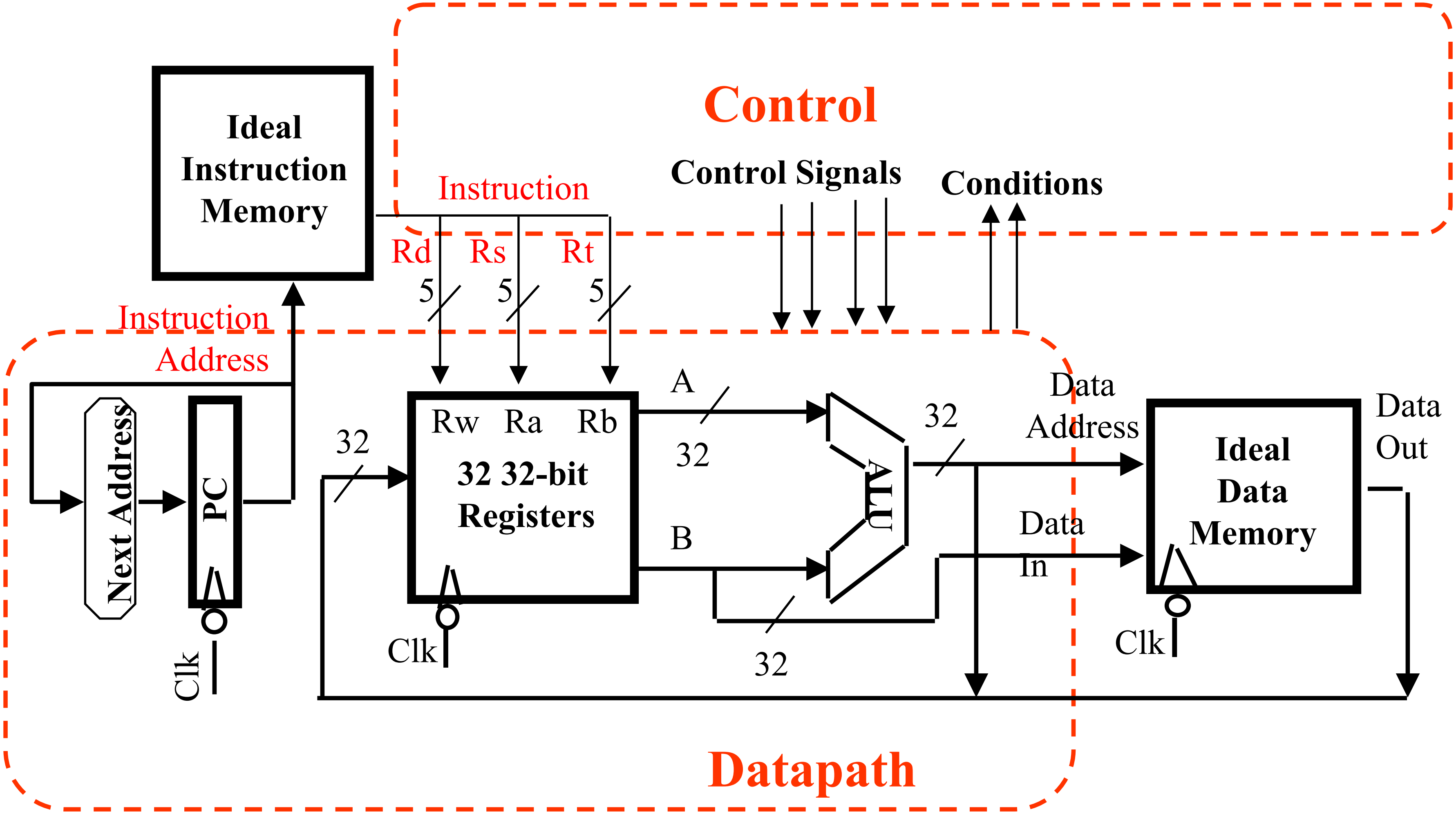
Drain (bubble) stages below it

Alternatively, launch multiply/divide to autonomous unit, only stall pipe if attempt to get result before ready

- This means stall mflo/mfhi in decode stage if multiply/divide still executing
- Extra credit in Lab 5 does this

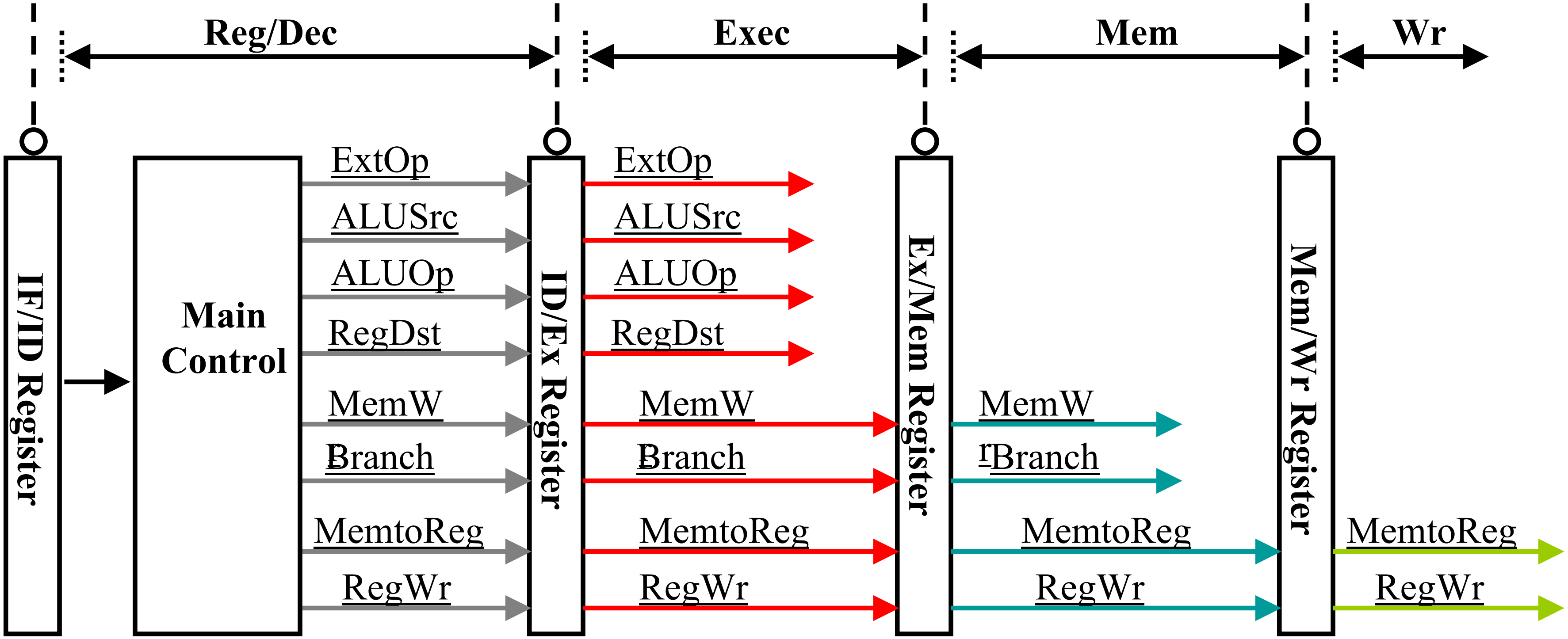
Ex: Multiply, Divide, Cache Miss

Recall: Single cycle control!

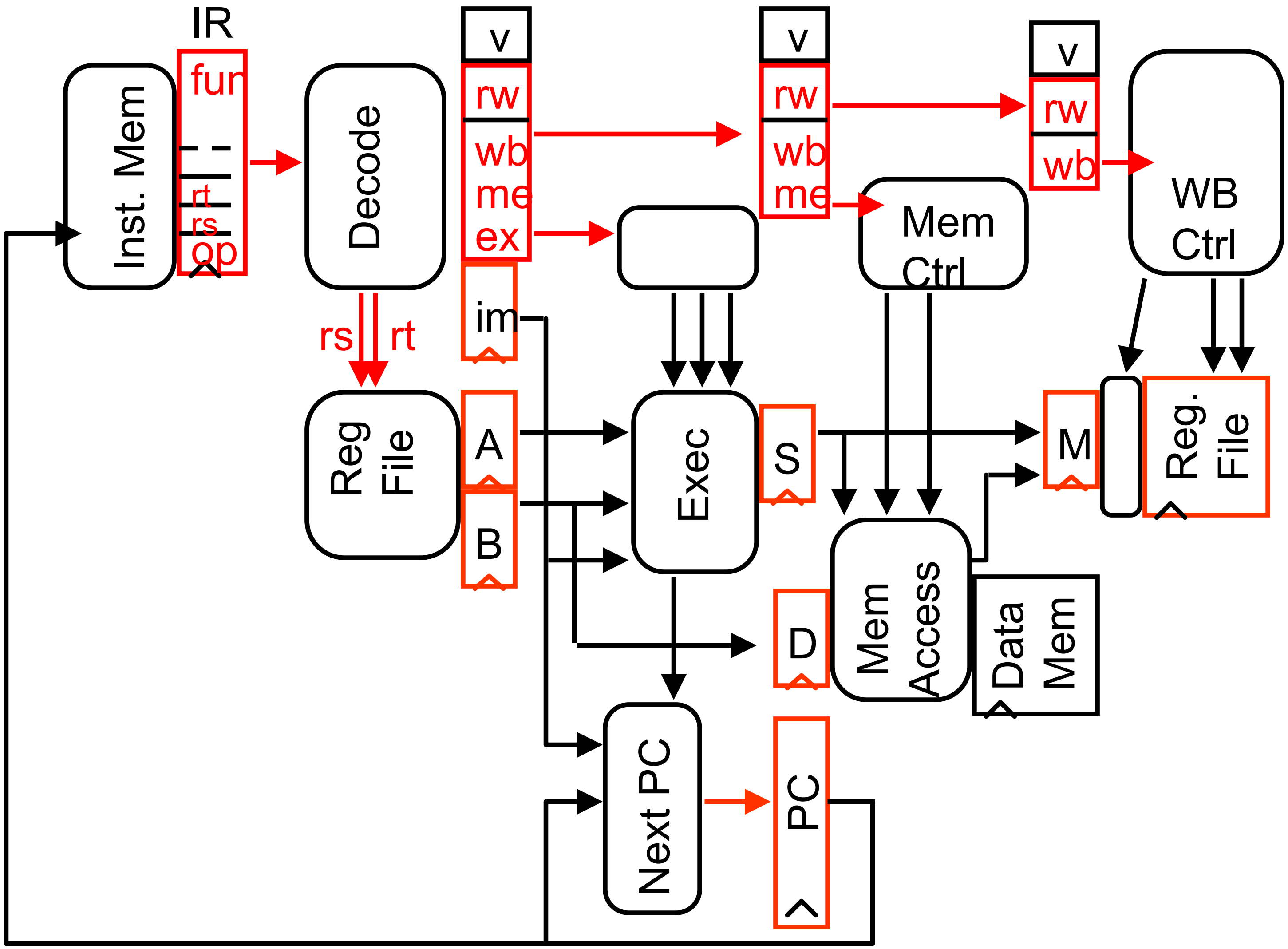


Data Stationary Control

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



Datapath + Data Stationary Control



Administrivia

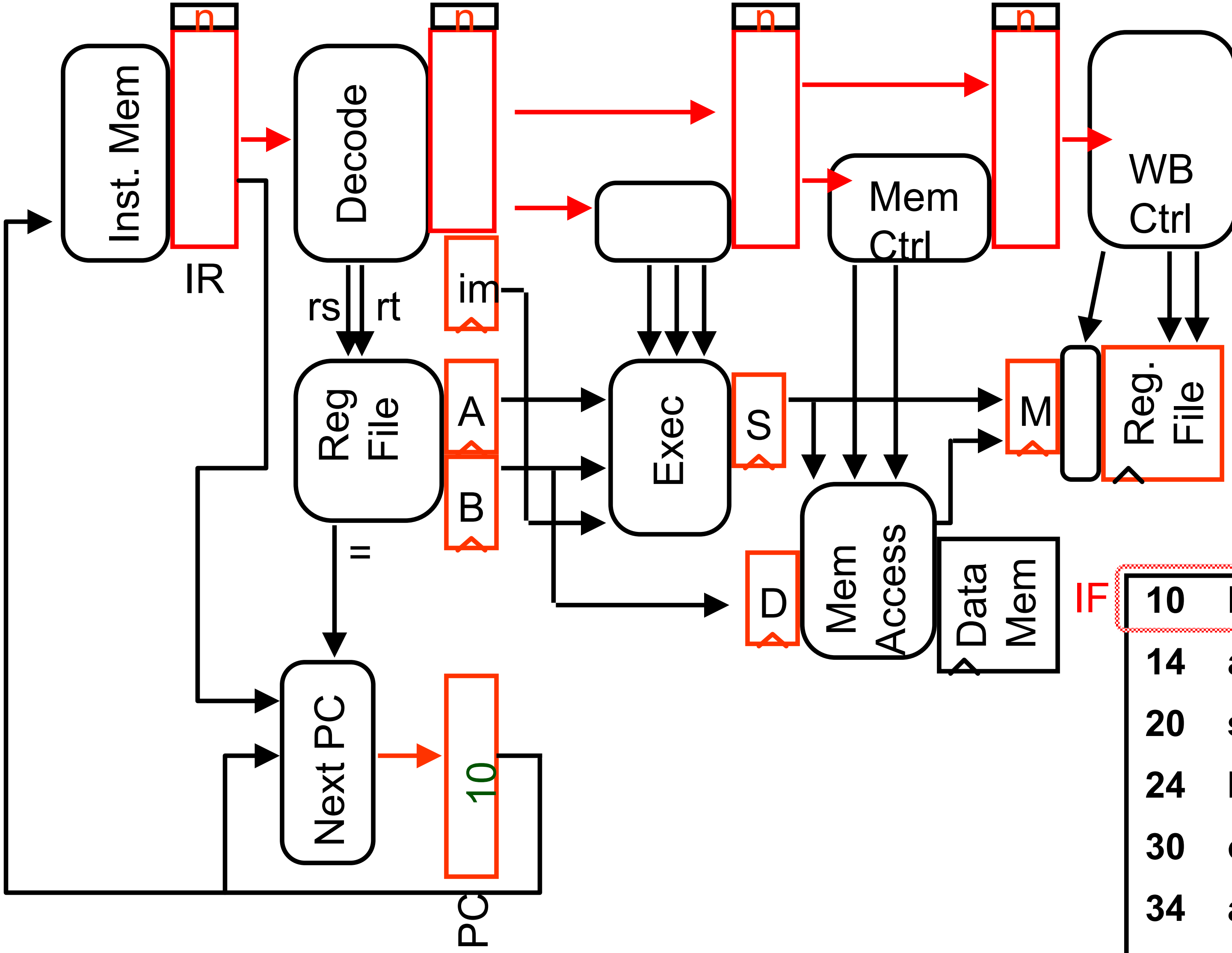
- Lab 4 Project document Thursday 9 PM paper or email
- Reading Chapter 6, sections 6.1 to 6.5
- Midterm Wed Oct 8 5:30 - 8:30 in 1 LeConte
  - Midterm review Sunday Oct 4, 5 PM in 306 Soda
  - Bring 1 page, handwritten notes, both sides
  - Meet at LaVal's Northside afterwards for Pizza
- Office hours
  - Mon 4 – 5:30 Jack, Tue 3:30-5 Kurt,
    Wed 3 – 4:30 John, Thu 3:30-5 Ben
  - Dave's office hours Tue 3:30 – 5

Let’s Try it Out

```
10     lw     r1, r2(35)
14     addI   r2, r2, 3
20     sub    r3, r4, r5
24     beq    r6, r7, 100
30     ori    r8, r9, 17
34     add    r10, r11, r12

100    and    r13, r14, 15
```

these addresses are octal

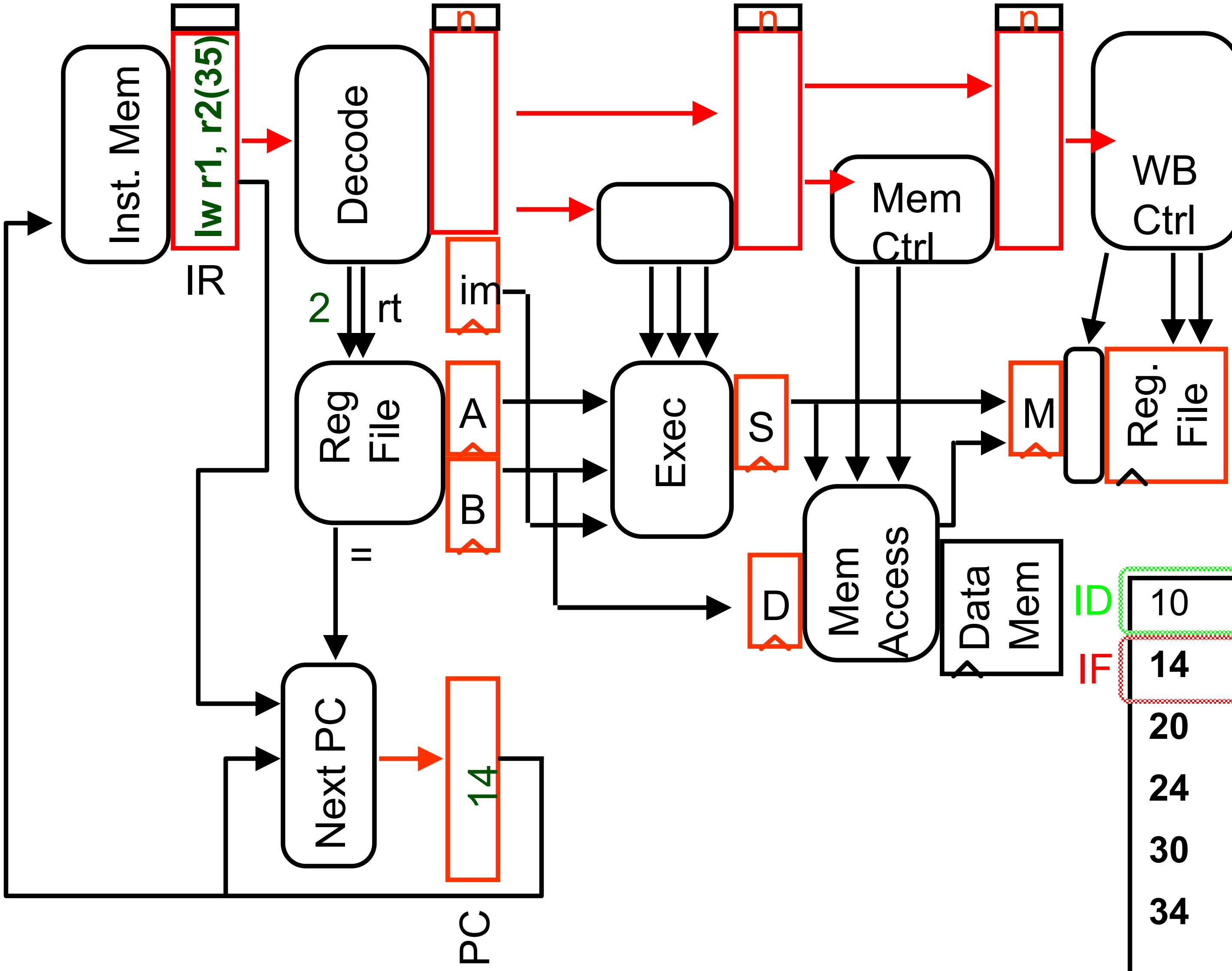
Start: Fetch 10



IF

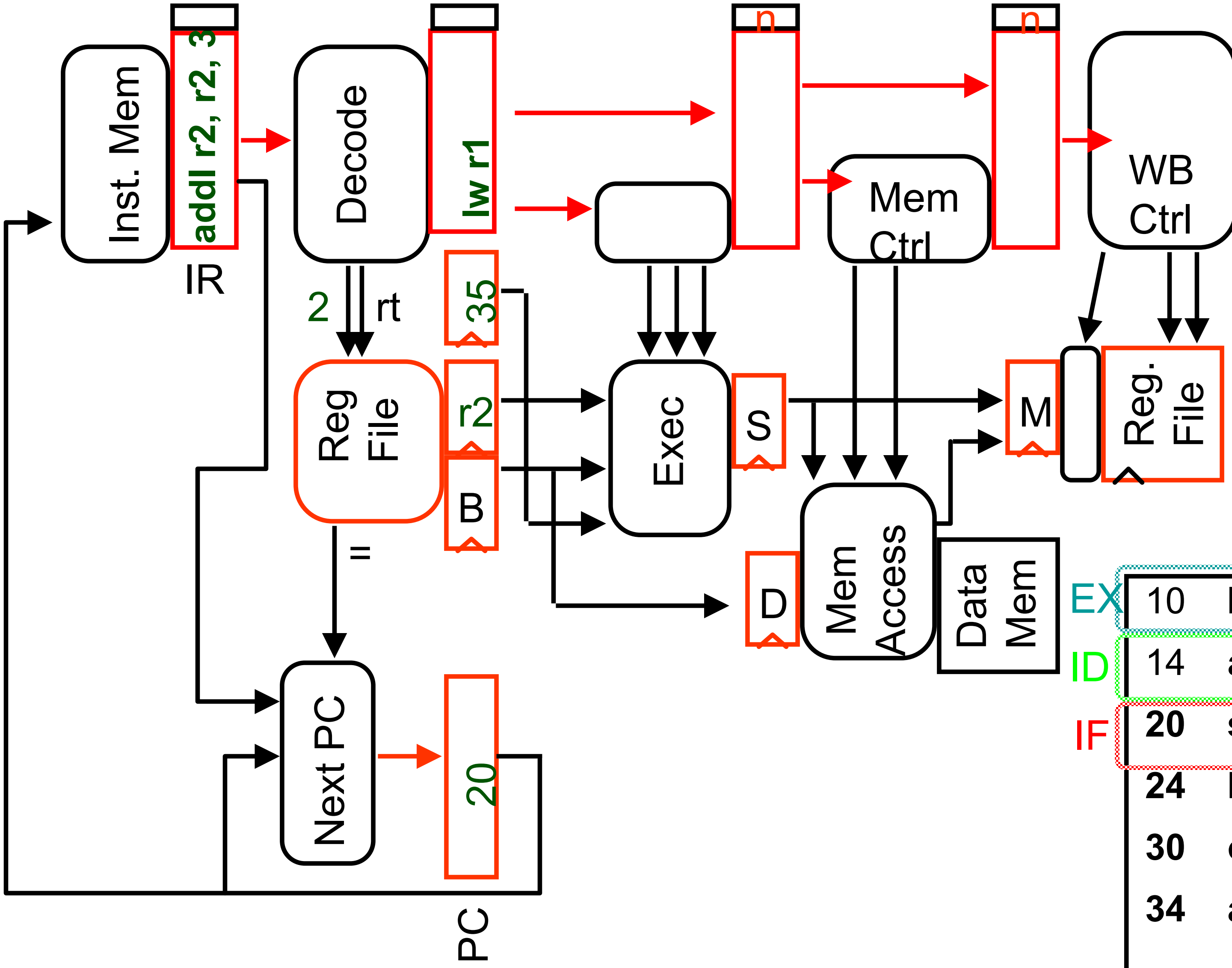
```
10   lw    r1, r2(35)
14   addI  r2, r2, 3
20   sub   r3, r4, r5
24   beq   r6, r7, 100
30   ori   r8, r9, 17
34   add   r10, r11, r12


100  and   r13, r14, 15
```

Fetch 14, Decode 10



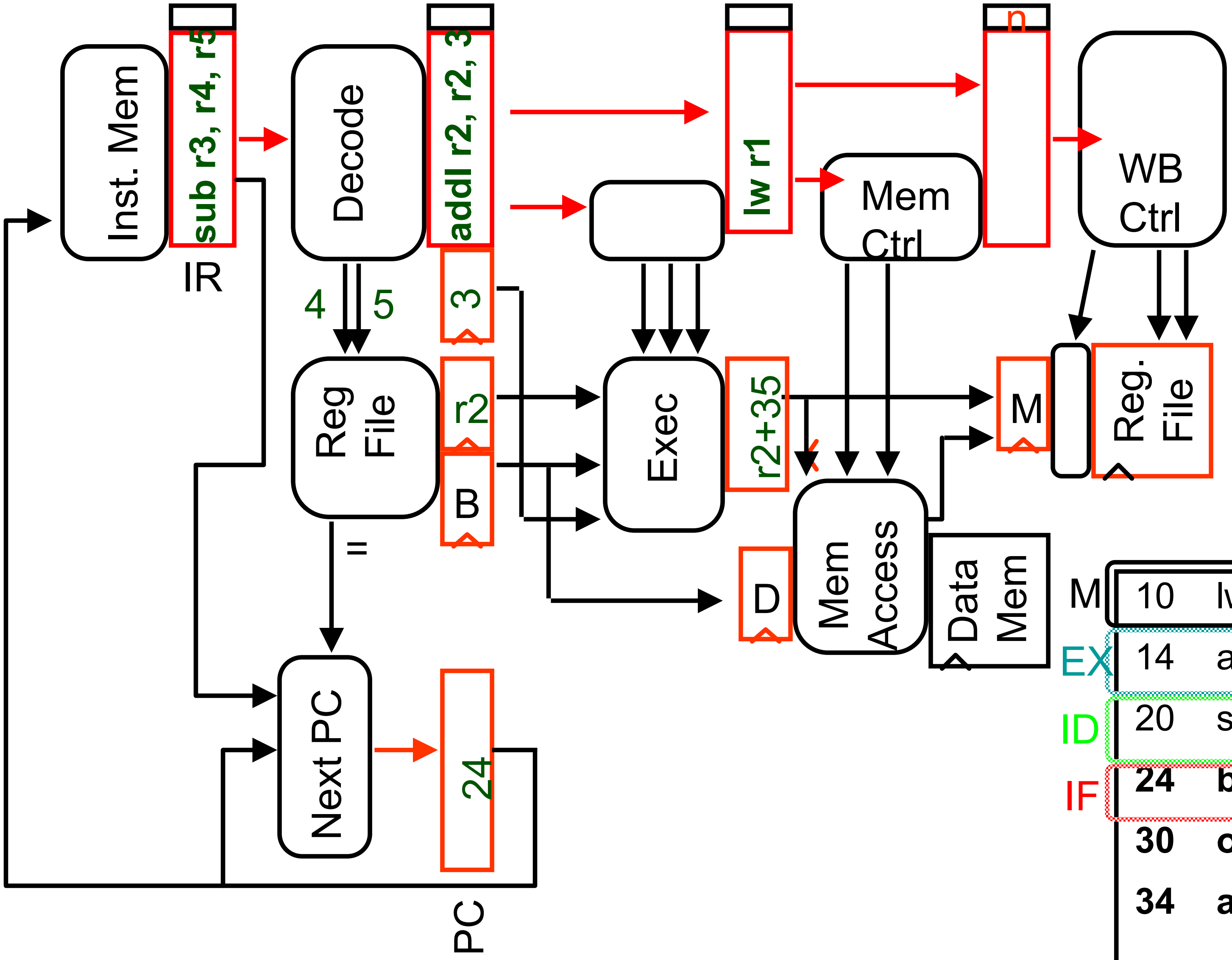
ID	10	lw	r1, r2(35)
IF	**14**	**addI**	**r2, r2, 3**
	20	**sub**	**r3, r4, r5**
	24	**beq**	**r6, r7, 100**
	30	**ori**	**r8, r9, 17**
	34	**add**	**r10, r11, r12**
	100	**and**	**r13, r14, 15**

Fetch 20, Decode 14, Exec 10



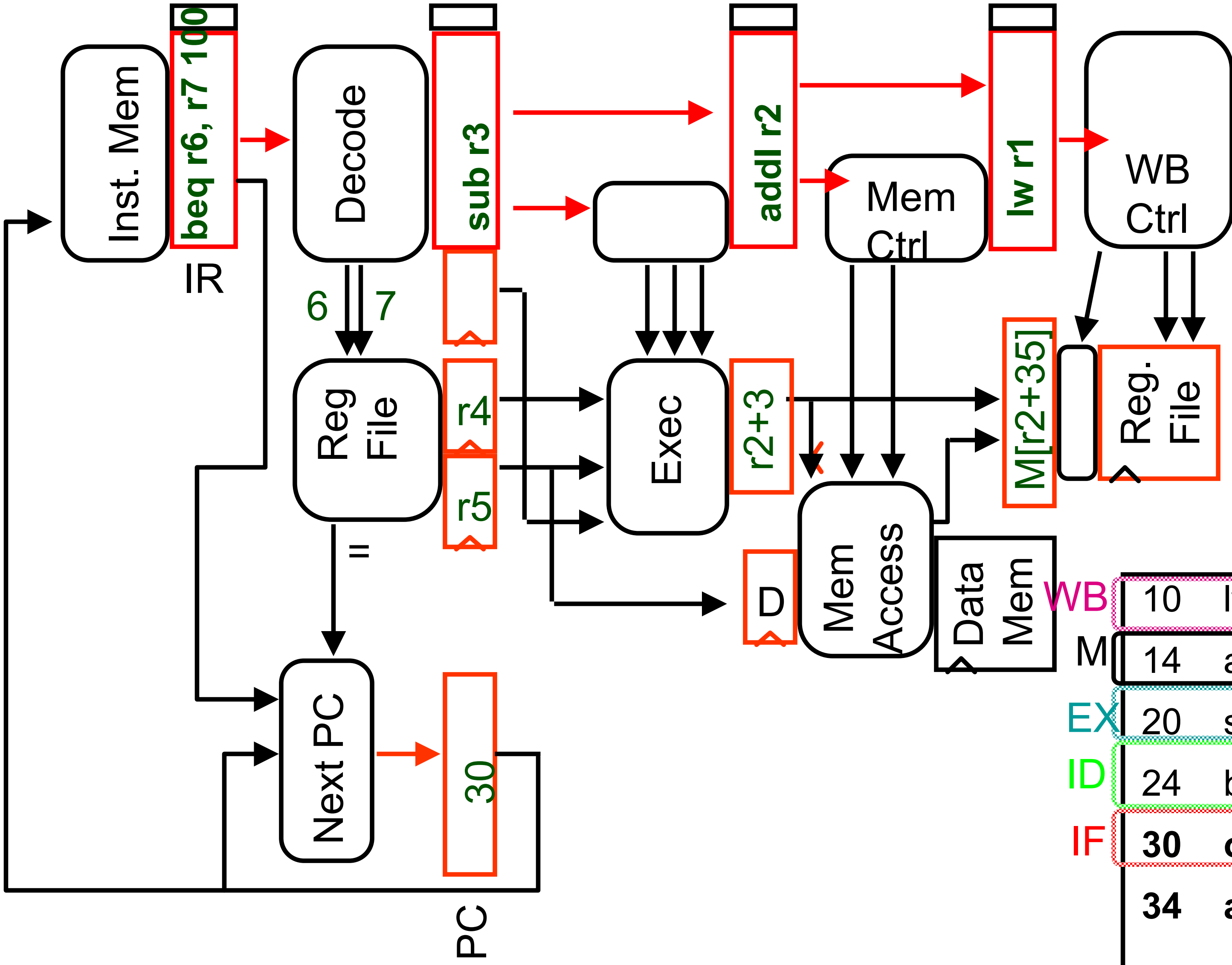
```
EX  10   lw     r1, r2(35)
ID  14   addI   r2, r2, 3
IF  20   sub    r3, r4, r5
    24   beq    r6, r7, 100
    30   ori    r8, r9, 17
    34   add    r10, r11, r12

    100  and    r13, r14, 15
```

Fetch 24, Decode 20, Exec 14, Mem 10



M	10	lw	r1, r2(35)
EX	14	addI	r2, r2, 3
ID	20	sub	r3, r4, r5
IF	**24**	**beq**	**r6, r7, 100**
	30	**ori**	**r8, r9, 17**
	34	**add**	**r10, r11, r12**
	100	**and**	**r13, r14, 15**

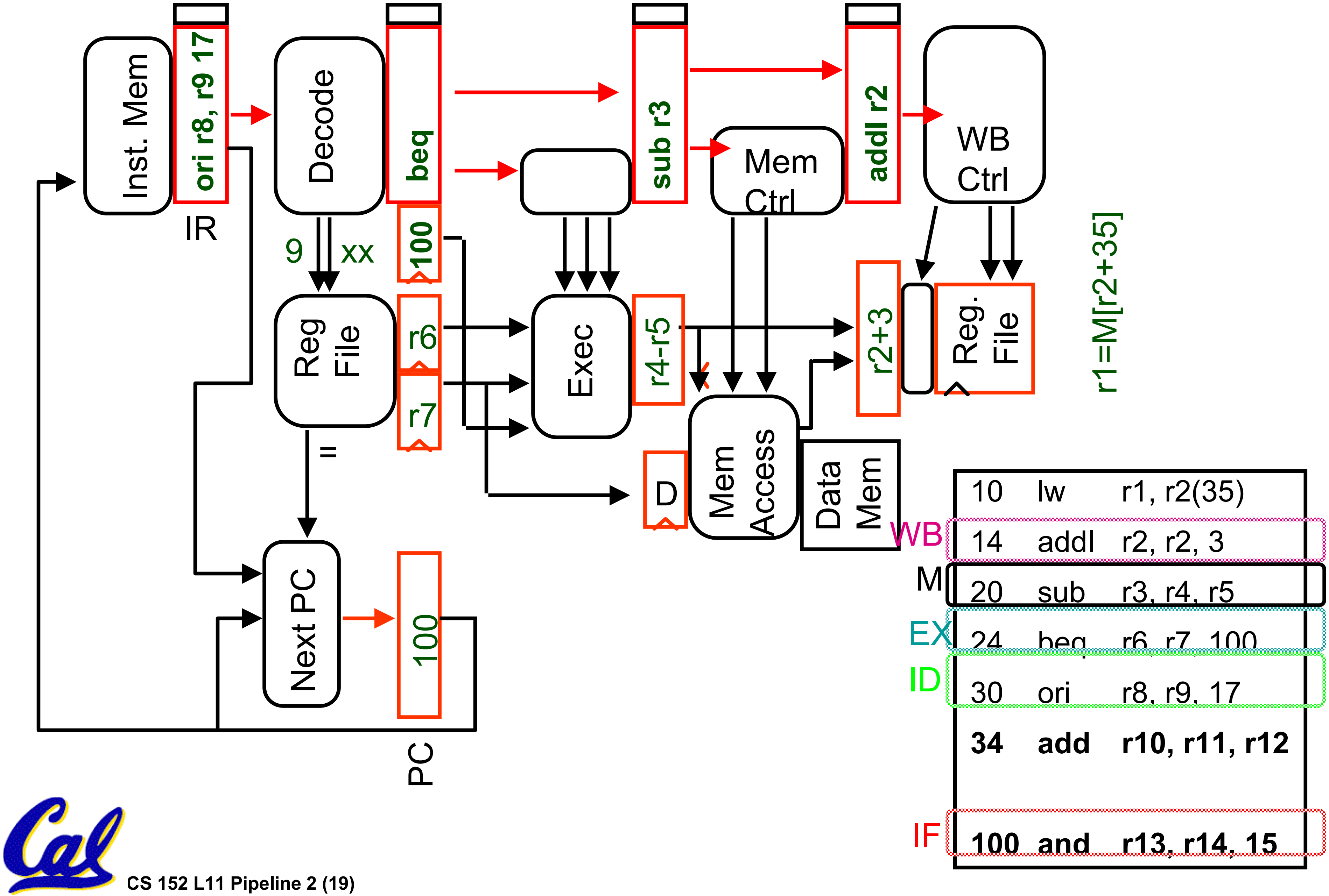
Fetch 30, Dcd 24, Ex 20, Mem 14, WB 10



WB	10	lw	r1, r2(35)
M	14	addI	r2, r2, 3
EX	20	sub	r3, r4, r5
ID	24	beq	r6, r7, 100
IF	**30**	**ori**	**r8, r9, 17**
	34	**add**	**r10, r11, r12**
	100	**and**	**r13, r14, 15**

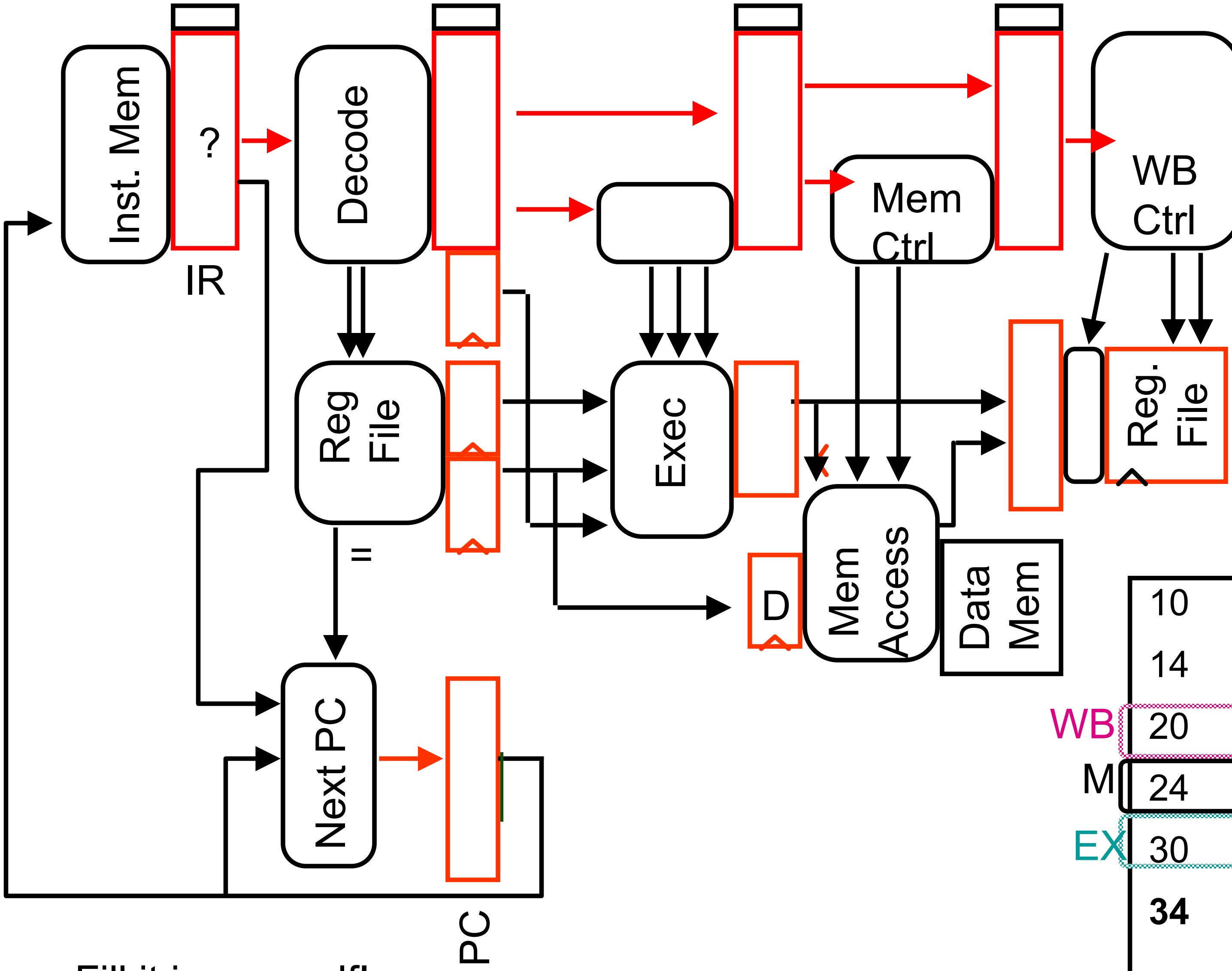
Note Delayed Branch: always execute `ori` after `beq`

Fetch 100, Dcd 30, Ex 24, Mem 20, WB 14



```
   10   lw    r1, r2(35)
WB 14   addI  r2, r2, 3
M  20   sub   r3, r4, r5
EX 24   beq   r6, r7, 100
ID 30   ori   r8, r9, 17
   34   add   r10, r11, r12

IF 100  and   r13, r14, 15
```

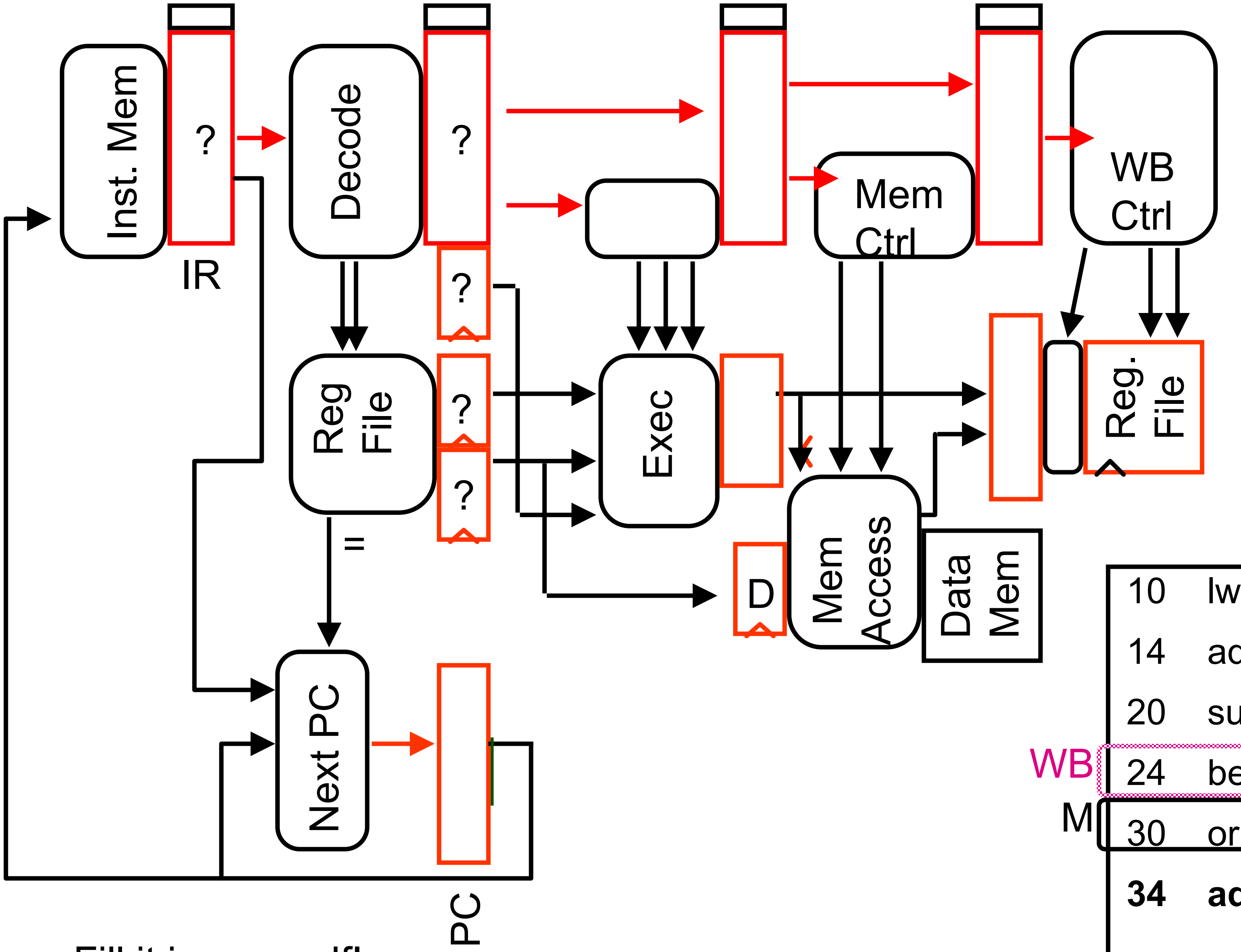
Fetch 104, Dcd 100, Ex 30, Mem 24, WB 20



	10	lw	r1, r2(35)
	14	addI	r2, r2, 3
WB	20	sub	r3, r4, r5
M	24	beq	r6, r7, 100
EX	30	ori	r8, r9, 17
	34	**add**	**r10, r11, r12**
ID	**100**	**and**	**r13, r14, 15**

Fill it in yourself!

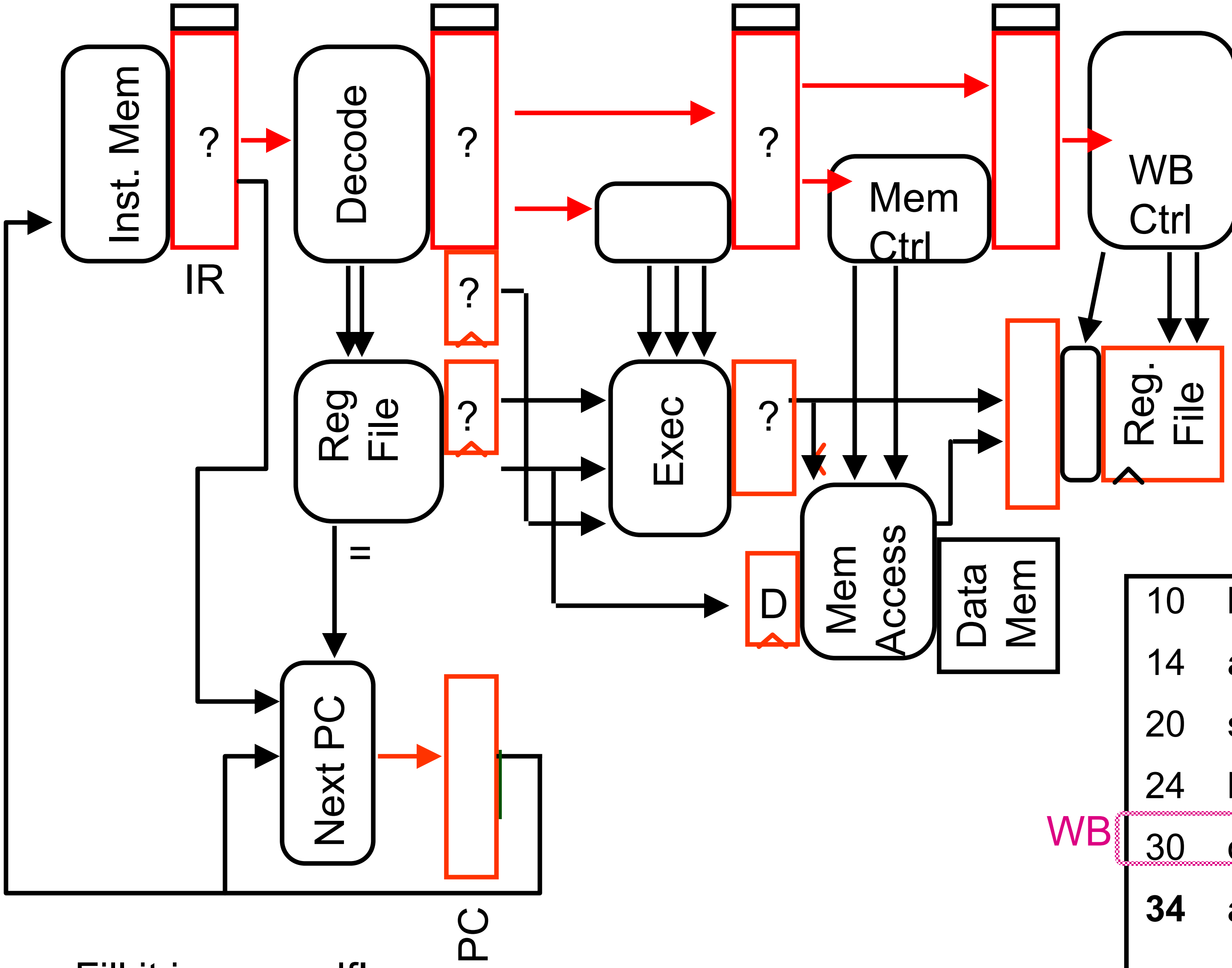
Fetch 110, Dcd 104, Ex 100, Mem 30, WB 24



	10	lw	r1, r2(35)
	14	addI	r2, r2, 3
	20	sub	r3, r4, r5
WB	24	beq	r6, r7, 100
M	30	ori	r8, r9, 17
	34	**add**	**r10, r11, r12**
EX	**100**	**and**	**r13, r14, 15**

Fill it in yourself!

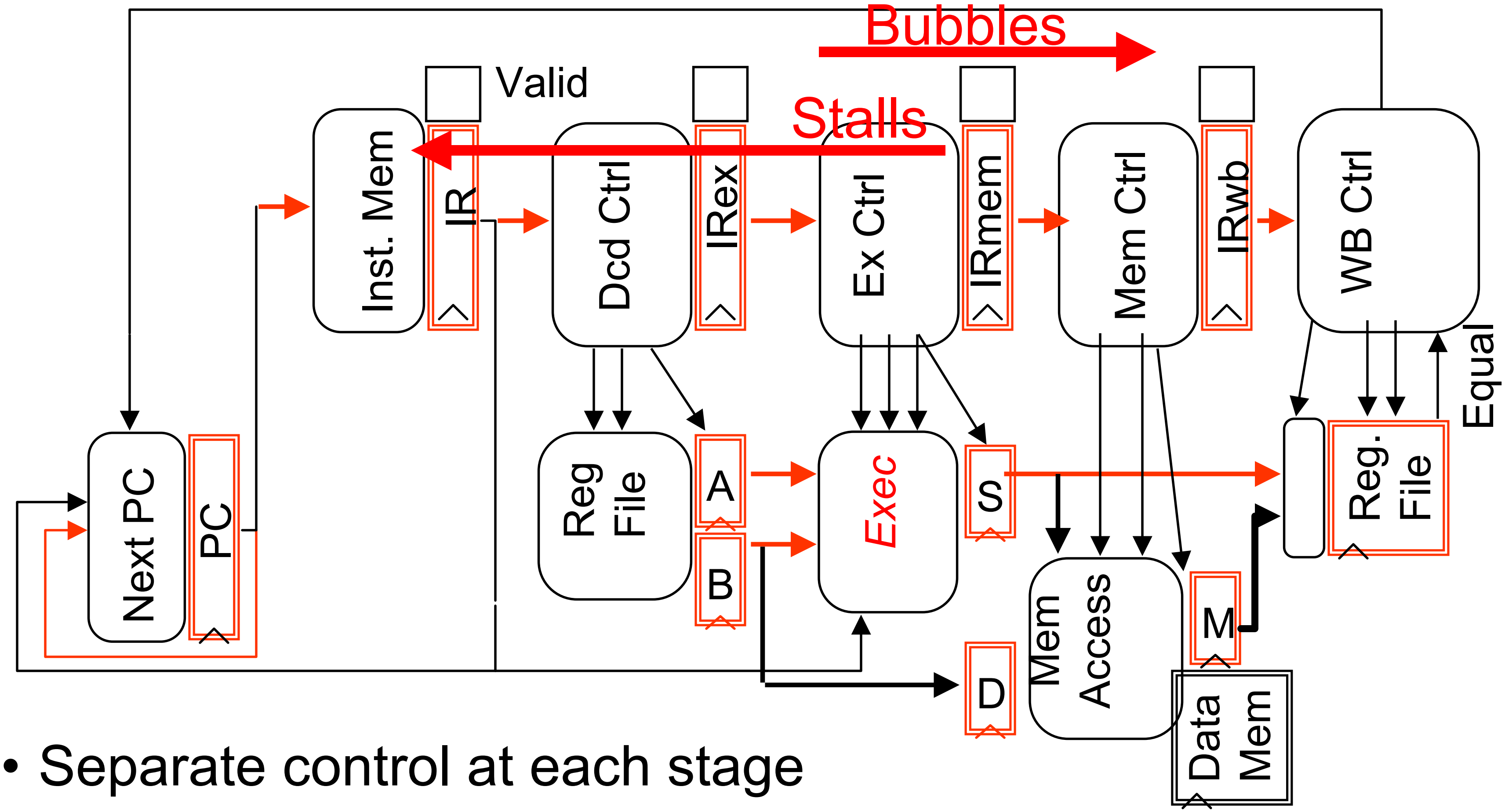
Fetch 114, Dcd 110, Ex 104, Mem 100, WB 30



```
    10   lw     r1, r2(35)
    14   addI   r2, r2, 3
    20   sub    r3, r4, r5
    24   beq    r6, r7, 100
WB  30   ori    r8, r9, 17
    34   add    r10, r11, r12

M   100  and    r13, r14, 15
```

Fill it in yourself!

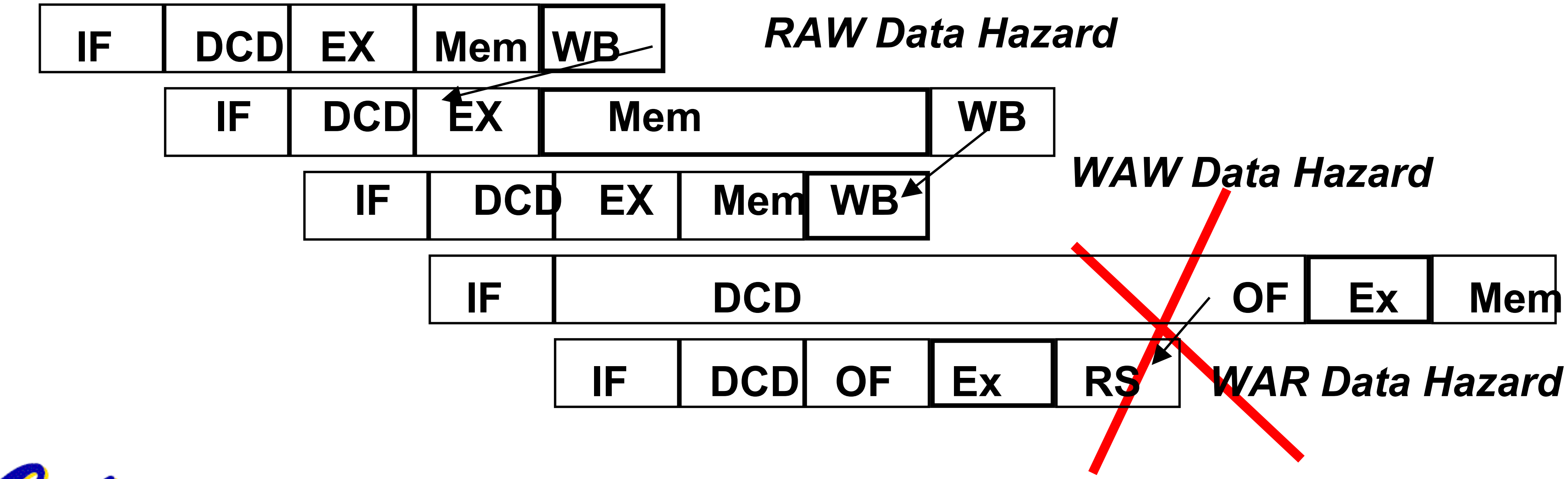
Pipelined Processor



- Separate control at each stage
- Stalls propagate backwards to freeze previous stages
- Bubbles in pipeline introduced by placing “Noops” into local stage, stall previous stages.

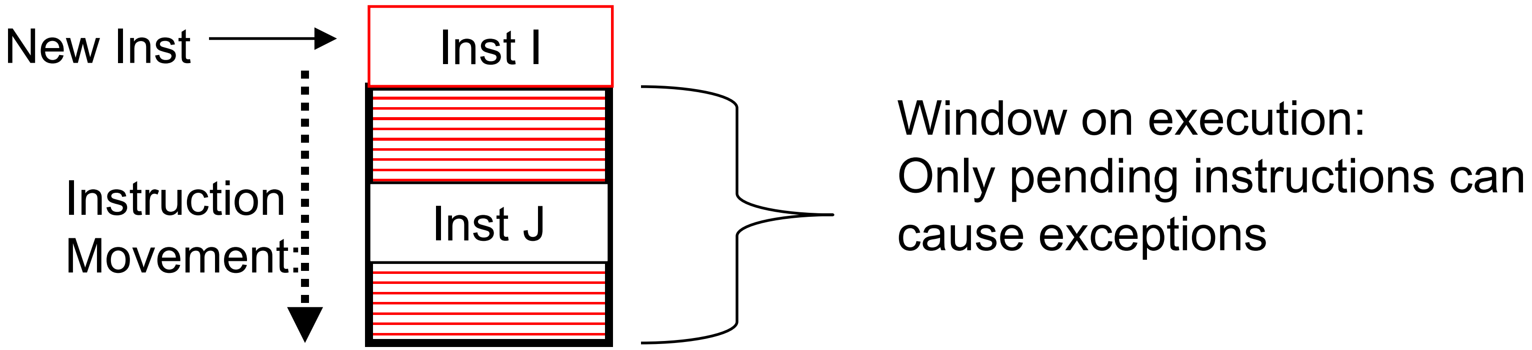
Recap: Data Hazards

- Avoid some "by design"
    - eliminate WAR by always fetching operands early (DCD) in pipe
    - eleminate WAW by doing all WBs in order (last stage, static)
- Detect and resolve remaining ones
    - stall or forward (if possible)



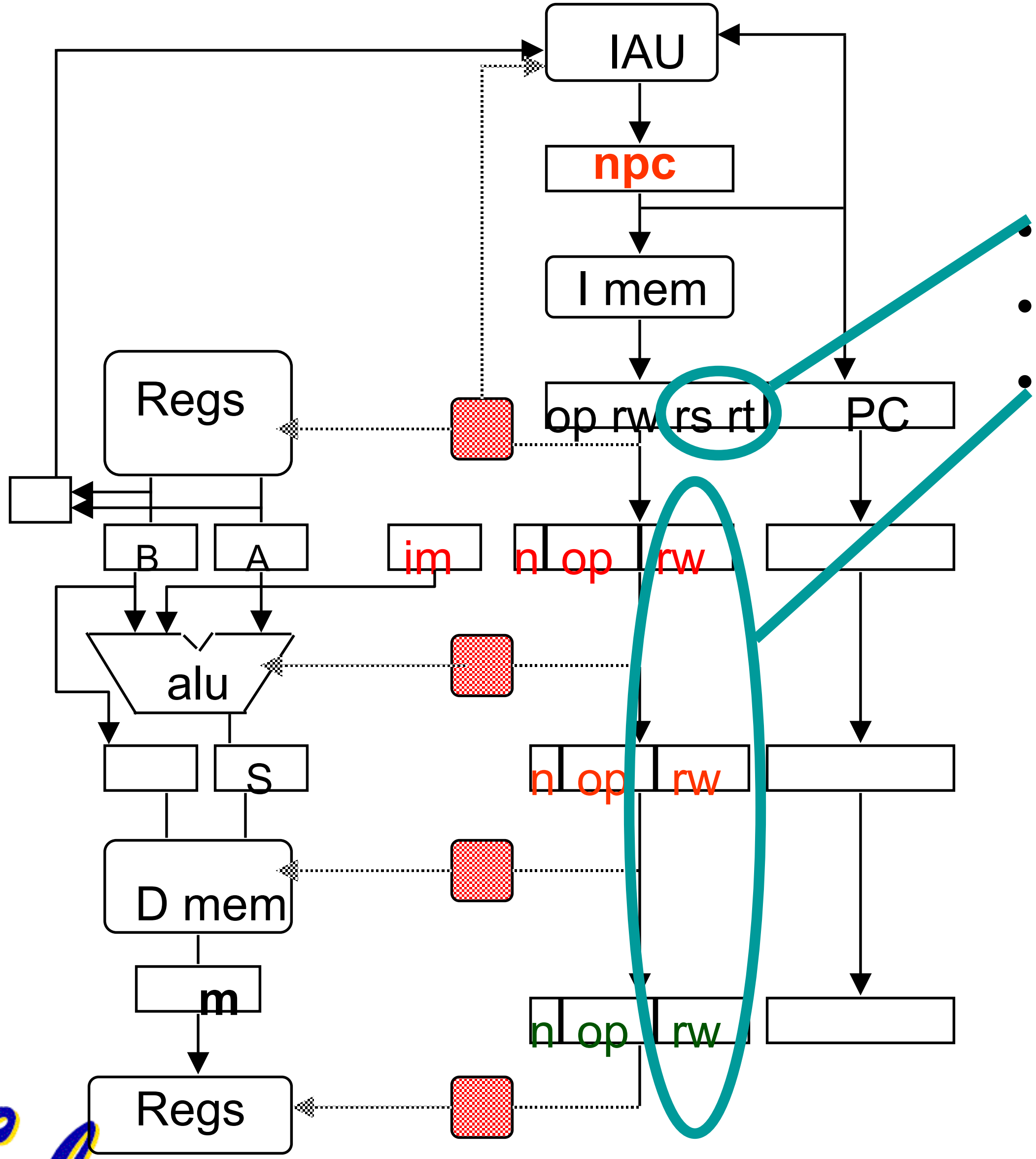
Hazard Detection

- Suppose instruction *i* is about to be issued and a predecessor instruction *j* is in the instruction pipeline.



- A RAW hazard exists on register ρ if $\rho \in \text{Rregs}(\ i\ ) \cap \text{Wregs}(\ j\ )$
  - Keep a record of pending writes (for inst's in the pipe) and compare with operand regs of current instruction.
  - When instruction issues, reserve its result register.
  - When on operation completes, remove its write reservation.
- A WAW hazard exists on register ρ if $\rho \in \text{Wregs}(\ i\ ) \cap \text{Wregs}(\ j\ )$
- A WAR hazard exists on register ρ if $\rho \in \text{Wregs}(\ i\ ) \cap \text{Rregs}(\ j\ )$

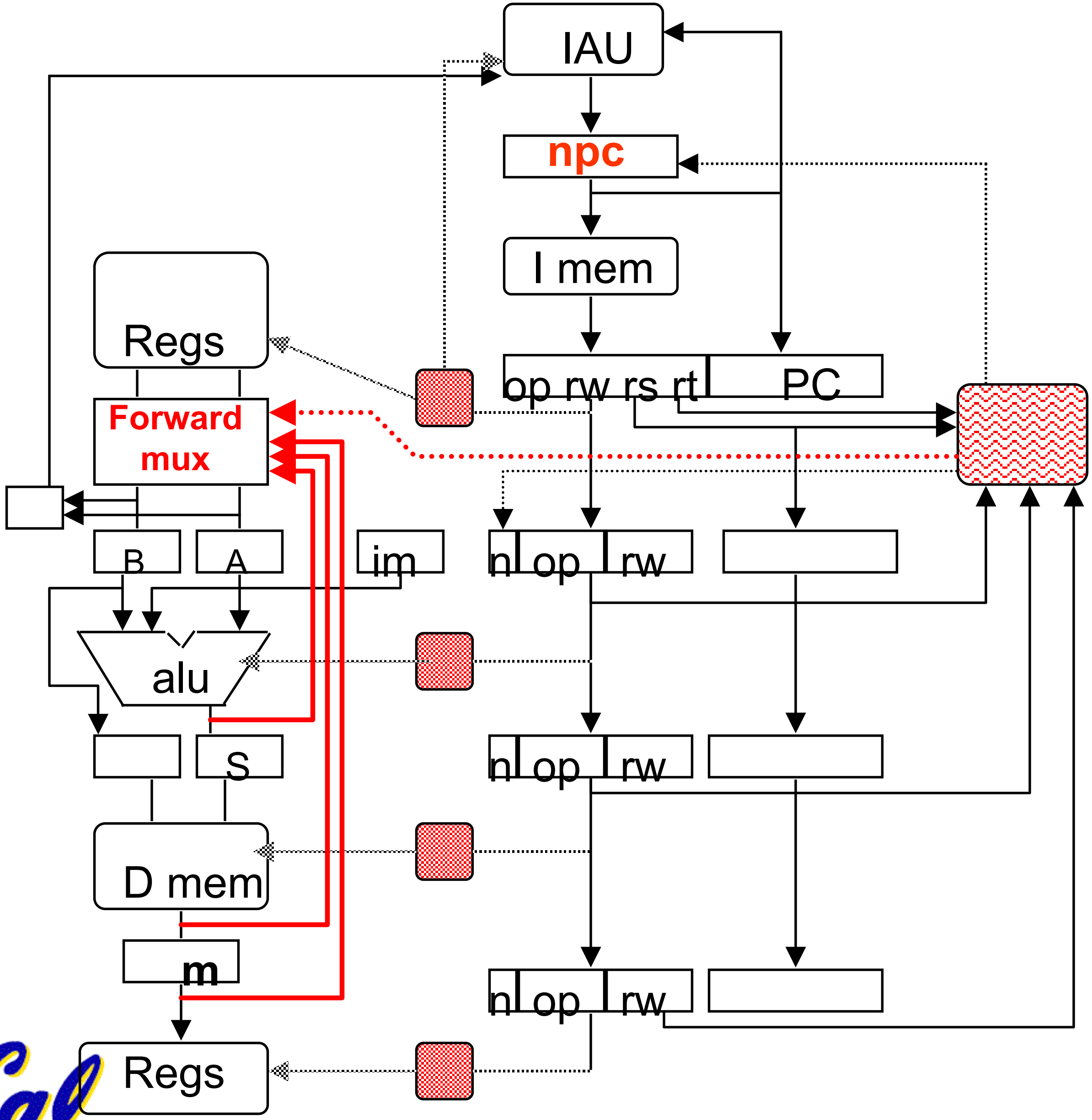
Record of Pending Writes In Pipeline Registers



- Current operand registers
- Pending writes
- hazard <=

$((rs == rw_{ex})$ & $regW_{ex})$
OR
$((rs == rw_{mem})$ & $regW_{me})$
OR
$((rs == rw_{wb})$ & $regW_{wb})$
OR
$((rt == rw_{ex})$ & $regW_{ex})$
OR
$((rt == rw_{mem})$ & $regW_{me})$
OR
$((rt == rw_{wb})$ & $regW_{wb})$

Resolve RAW by “forwarding” (or bypassing)

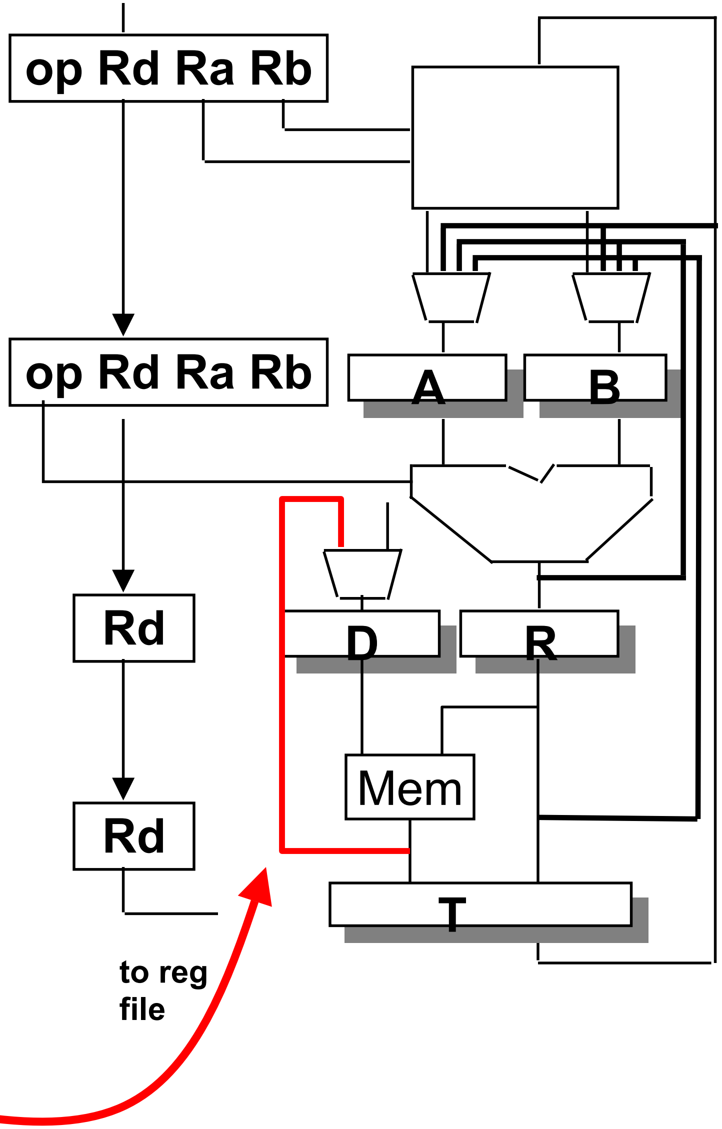


- Detect nearest valid write op operand register and forward into op latches, bypassing remainder of the pipe
- Increase muxes to add paths from pipeline registers
- Data Forwarding = Data Bypassing

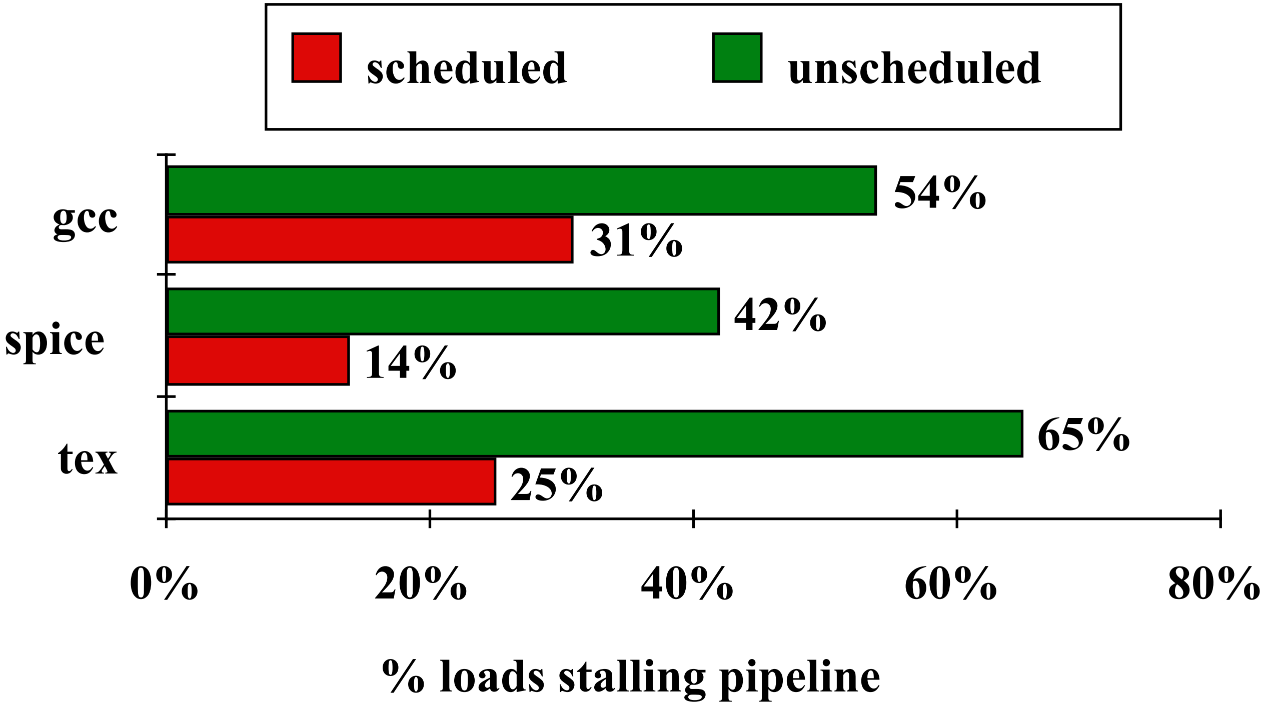
What about memory operations?

- If instructions are initiated in order and operations always occur in the same stage, there can be no hazards between memory operations!
- What does delaying WB on arithmetic operations cost?
  - cycles ?
  - hardware ?
- What about data dependence on loads?
  R1 <- R4 + R5
  R2 <- Mem[R2 + I]
  R3 <- R2 + R1
  ⇒ "Delayed Loads"
- Can recognize this in decode stage and introduce bubble while stalling fetch stage (hint for lab 5!)
- Tricky situation:
  R1 <- Mem[R2 + I]
  Mem[R3+34] <- R1

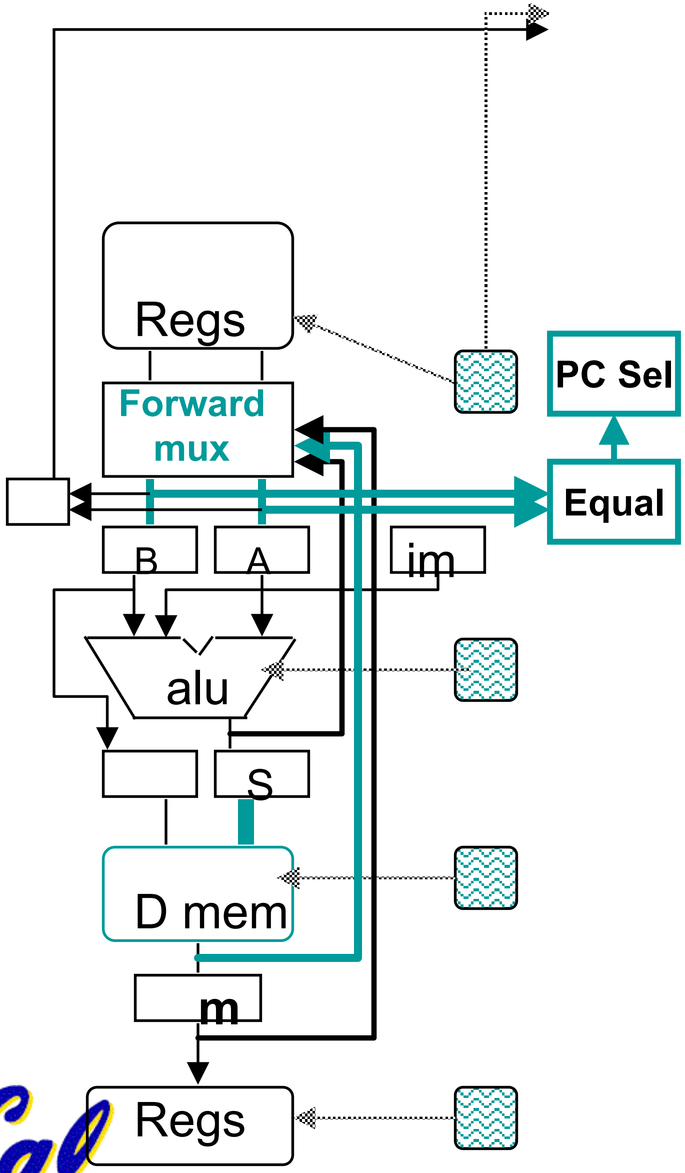
  Handle with bypass in memory stage!



Compiler Avoiding Load Stalls:



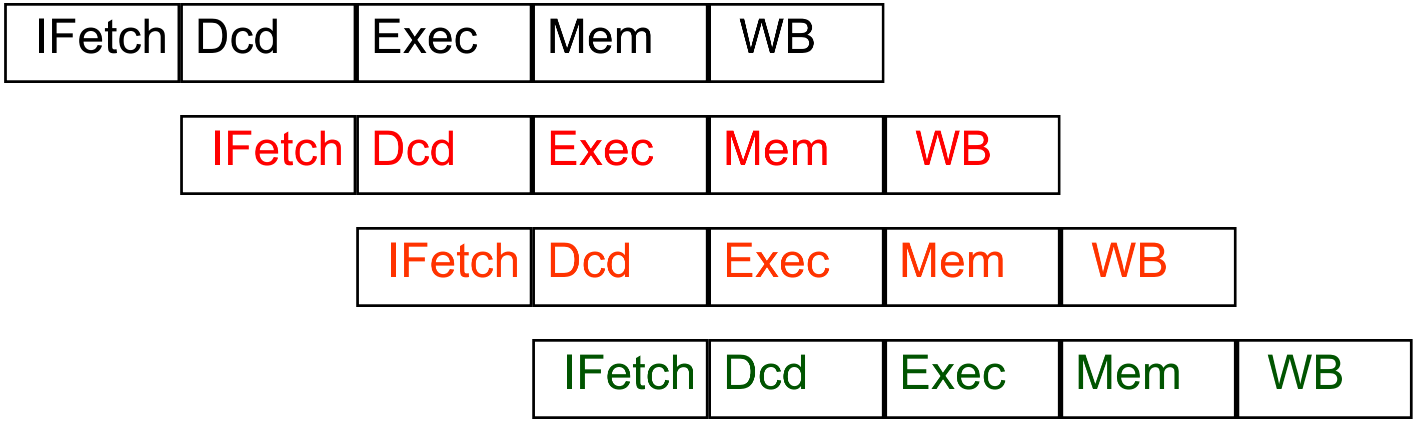
Question: Critical Path???



- Bypass path is invariably trouble
- Options?
  - Make logic really fast
  - Move forwarding after muxes
    - Problem: screws up branches that require forwarding!
    - Use same tricks as “carry-skip” adder to fix this?
    - This option may just push delay around….!

Is CPI = 1 for our pipeline?

- Remember that CPI is an “Average # cycles/inst



IFetch	Dcd	Exec	Mem	WB			
	IFetch	Dcd	Exec	Mem	WB		
		IFetch	Dcd	Exec	Mem	WB	
			IFetch	Dcd	Exec	Mem	WB

- CPI here is 1, since the average throughput is 1 instruction every cycle.
- What if there are stalls or multi-cycle execution?
- Usually CPI > 1. How close can we get to 1??

Recall: Compute CPI?

- Start with Base CPI
- Add stalls

$$CPI = CPI_{base} + CPI_{stall}$$

$$CPI_{stall} = STALL_{type-1} \times freq_{type-1} + STALL_{type-2} \times freq_{type-2}$$

- Suppose:
  - $CPI_{base}=1$
  - $Freq_{branch}=20\%$, $freq_{load}=30\%$
  - Suppose branches always cause 1 cycle stall
  - Loads cause a 100 cycle stall 1% of time
- Then: $CPI = 1 + (1\times0.20)+(100 \times 0.30\times0.01)=1.5$
- Multicycle? Could treat as:

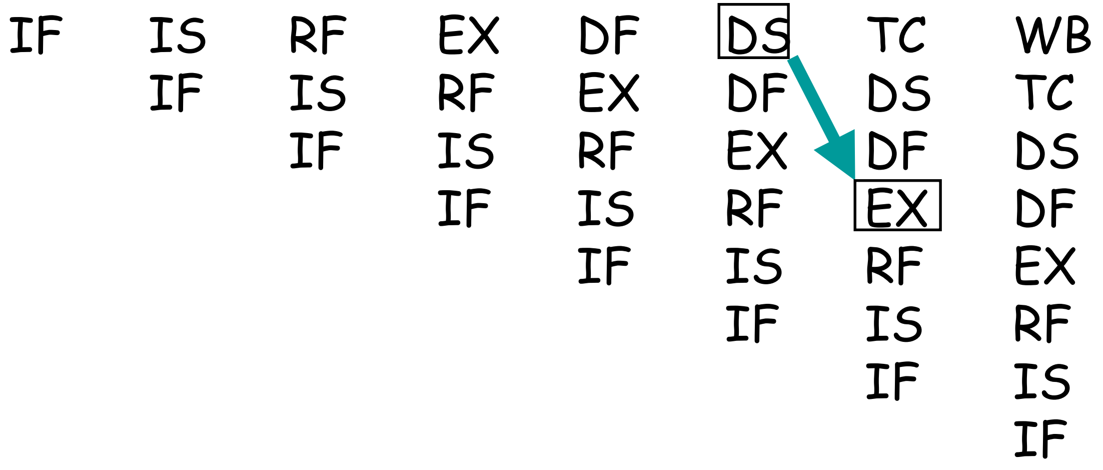
  $CPI_{stall}=(CYCLES-CPI_{base}) \times freq_{inst}$

Case Study: MIPS R4000 (200 MHz)

- 8 Stage Pipeline:
  - IF–first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
  - IS–second half of access to instruction cache.
  - RF–instruction decode and register fetch, hazard checking and also instruction cache hit detection.
  - EX–execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
  - DF–data fetch, first half of access to data cache.
  - DS–second half of access to data cache.
  - TC–tag check, determine whether the data cache access hit.
  - WB–write back for loads and register-register operations.
- 8 Stages:
  What is impact on Load delay? Branch delay? Why?

Case Study: MIPS R4000

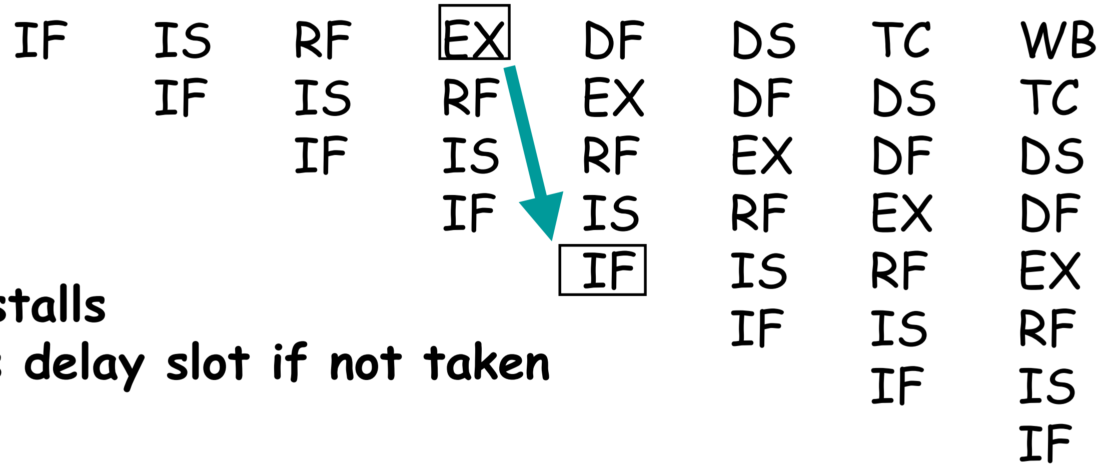
TWO Cycle
Load Latency



IF	IS	RF	EX	DF	DS	TC	WB
	IF	IS	RF	EX	DF	DS	TC
		IF	IS	RF	EX	DF	DS
			IF	IS	RF	EX	DF
				IF	IS	RF	EX
					IF	IS	RF
						IF	IS
							IF

THREE Cycle
Branch Latency
(conditions evaluated
during EX phase)

Delay slot plus two stalls
Branch likely cancels delay slot if not taken



IF	IS	RF	EX	DF	DS	TC	WB
	IF	IS	RF	EX	DF	DS	TC
		IF	IS	RF	EX	DF	DS
			IF	IS	RF	EX	DF
				IF	IS	RF	EX
					IF	IS	RF
						IF	IS
							IF

MIPS R4000 Floating Point

- FP Adder, FP Multiplier, FP Divider
- Last step of FP Multiplier/Divider uses FP Adder HW
- 8 kinds of stages in FP units:

Stage	*Functional unit*	*Description*
A	FP adder	Mantissa ADD stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

MIPS FP Pipe Stages

FP Instr	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*…*
Add, Subtract	U	S+A	A+R	R+S					
Multiply	U	E+M	M	M	M	N	N+A	R	
Divide	U	A	R	D^{28}	…	D+A	D+R, D+R, D+A, D+R, A, R		
Square root	U	E	(A+R)^{108}		…	A	R		
Negate	U	S							
Absolute value	U	S							
FP compare	U	A	R						

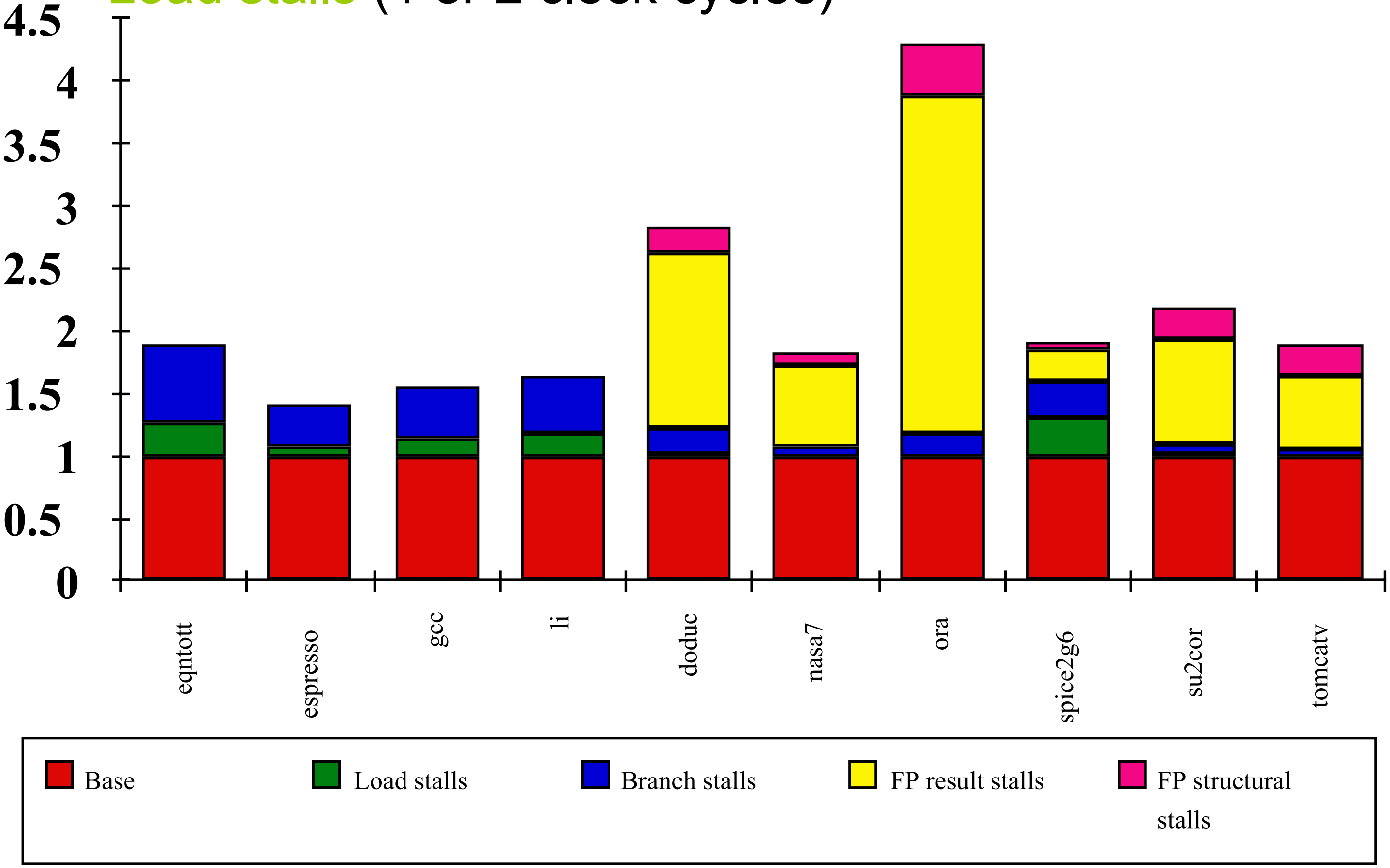
Stages:

M	*First stage of multiplier*
N	*Second stage of multiplier*
R	*Rounding stage*
S	*Operand shift stage*
U	*Unpack FP numbers*

A	***Mantissa ADD stage***
D	***Divide pipeline stage***
E	***Exception test stage***

R4000 Performance

- Not ideal CPI of 1:
  - FP structural stalls: Not enough FP hardware (parallelism)
  - FP result stalls: RAW data hazard (latency)
  - Branch stalls (2 cycles + unfilled slots)
  - Load stalls (1 or 2 clock cycles)



Summary

- What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction
- Data hazards must be handled carefully:
  - RAW data hazards handled by forwarding
  - WAW and WAR hazards don’t exist in 5-stage pipeline
- MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)
- More performance from deeper pipelines, parallelism