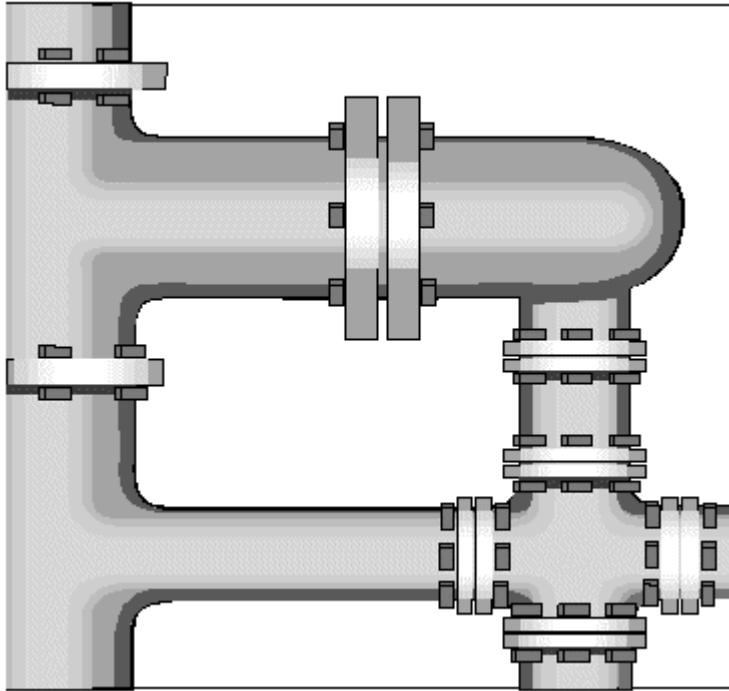


Pipelined Beta



Where are the registers?



Handouts: Lecture Slides

Increasing CPU Performance

$$\text{MIPS} = \frac{\text{Freq}}{\text{CPI}}$$

MIPS = Millions of Instructions/Second

Freq = Clock Frequency, MHz

CPI = Cycles per Instruction

To Increase MIPS:

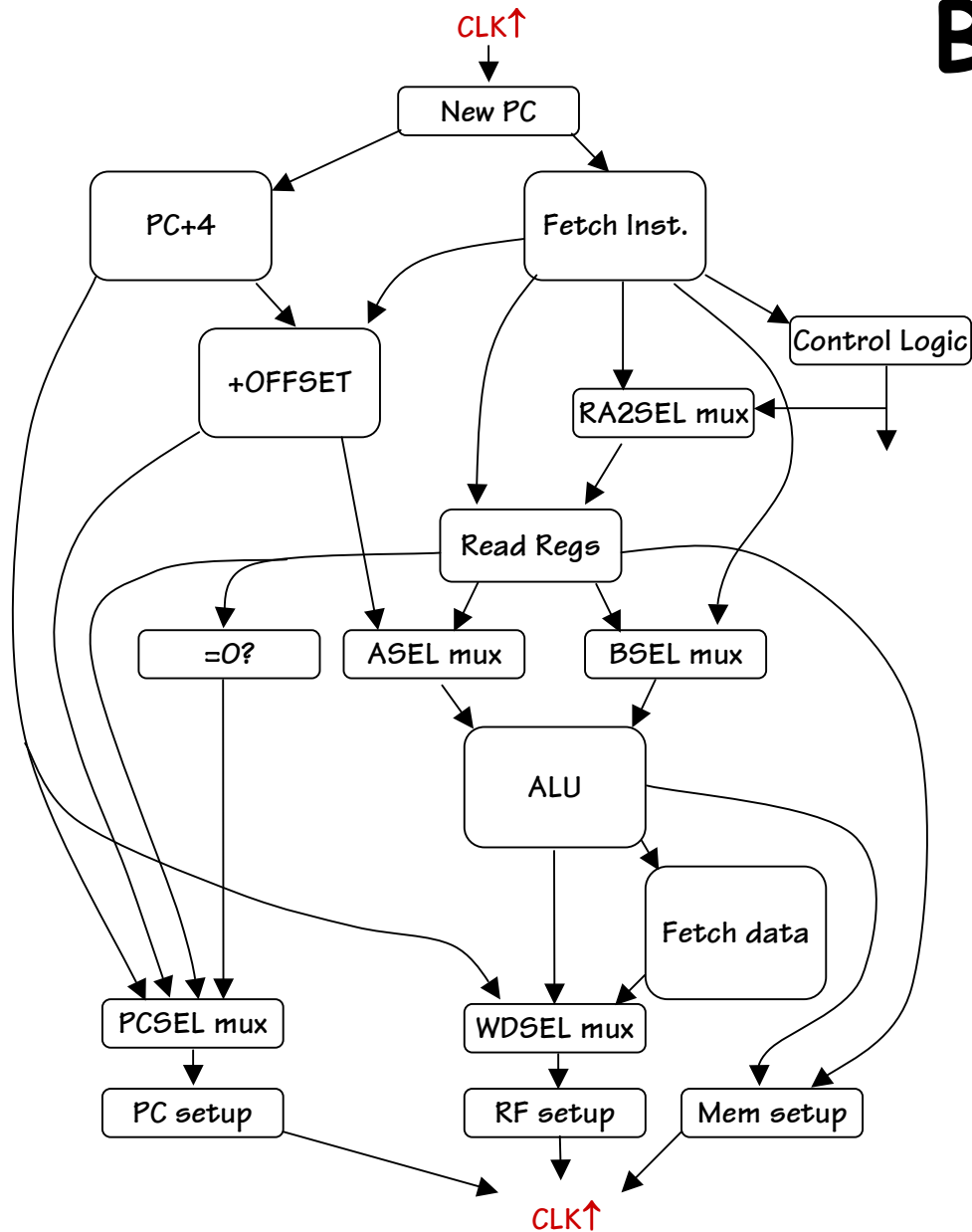
1. DECREASE CPI.

- RISC *simplicity* reduces CPI to 1.0.
- CPI *below 1.0?* Tough... you'll see multiple instruction issue machines in 6.823.

2. INCREASE Freq.

- Freq limited by delay along longest combinational path; hence
- **PIPELINING** is the key to improved performance through fast clocks.

Beta Timing



Wanted:
longest path

Complications:

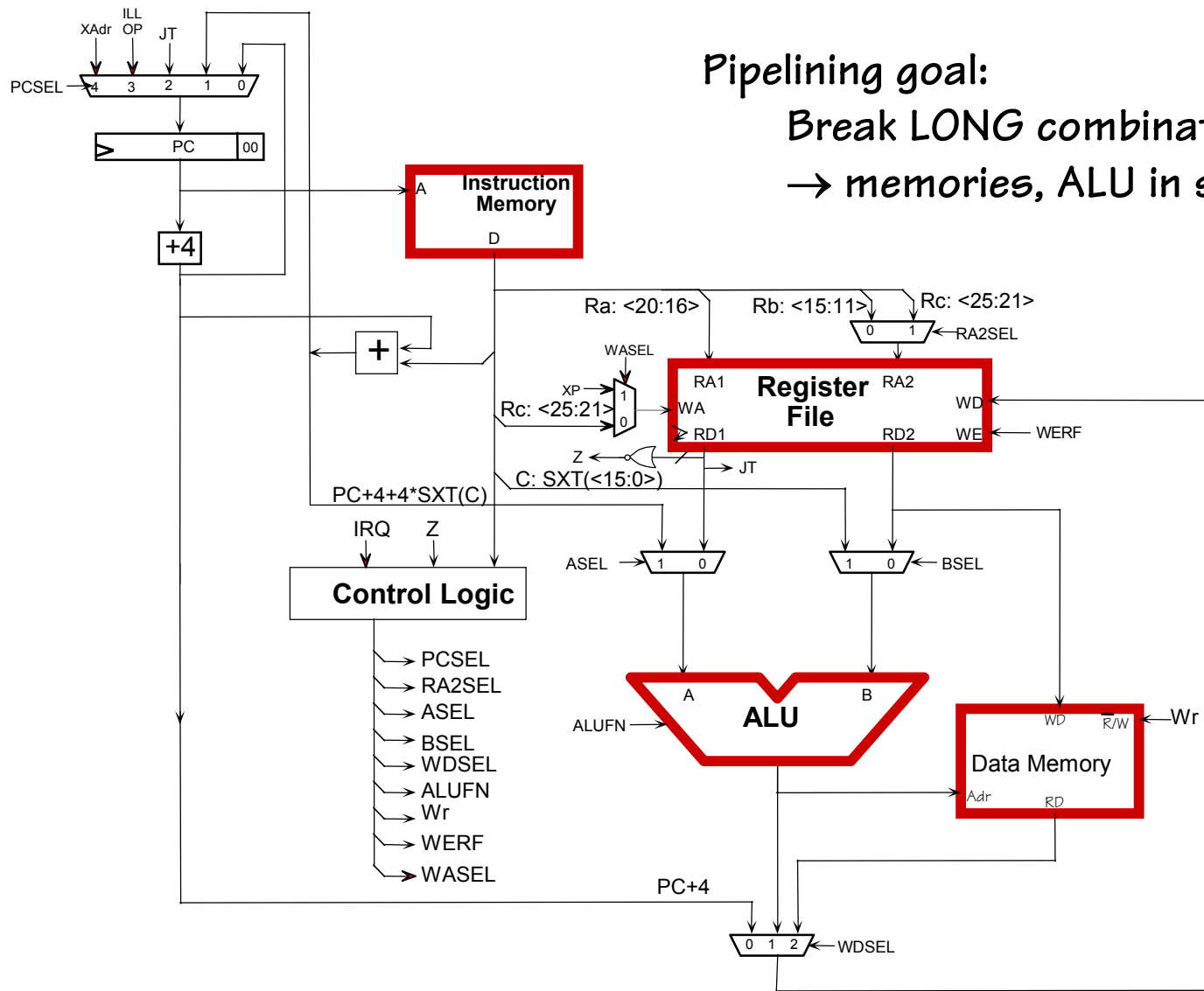
- some apparent paths aren't "possible"
- blobs have variable execution times (eg, ALU)
- time axis is not to scale (eg, $t_{PD, MEM}$ is very big!)

Where are the Bottlenecks?

Pipelining goal:

Break LONG combinational paths

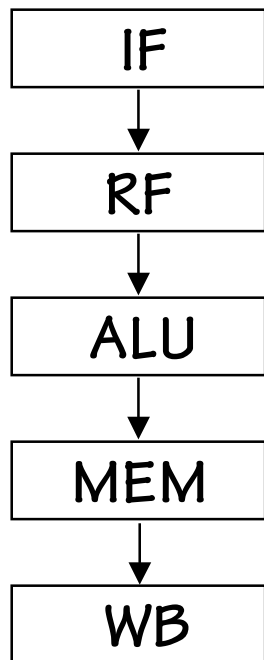
→ memories, ALU in separate stages



Ultimate Goal: 5-Stage Pipeline

GOAL: Maintain (nearly) 1.0 CPI, but increase clock speed to barely include slowest components (mems, regfile, ALU)

APPROACH: structure processor as 5-stage pipeline:



Instruction Fetch stage: Maintains PC, fetches one instruction per cycle and passes it to

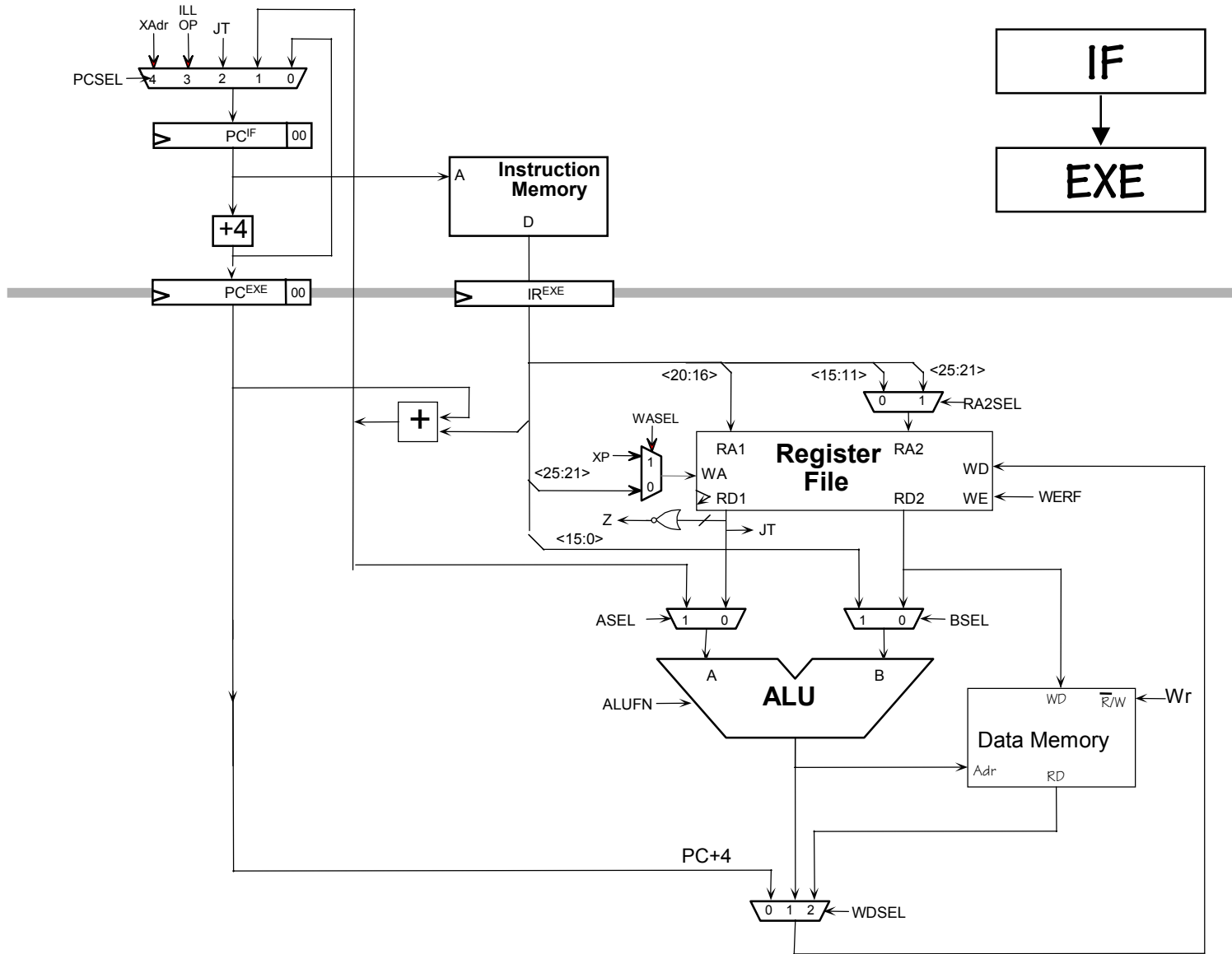
Register File stage: Reads source operands from register file, passes them to

ALU stage: Performs indicated operation, passes result to

Memory stage: If it's a LD, use ALU result as an address, pass mem data (or ALU result if not LD) to

Write-Back stage: writes result back into register file.

A Simple 2-Stage Pipeline



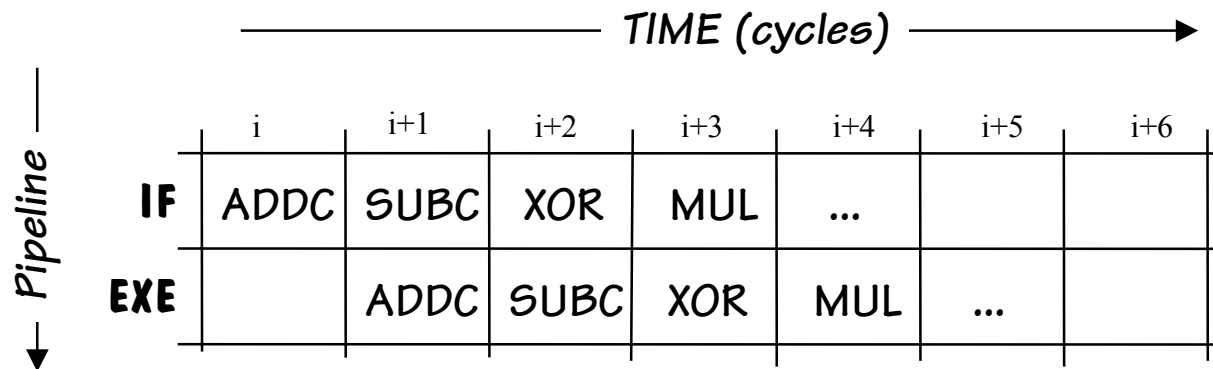
2-Stage Pipelined Beta Operation

Consider a sequence of instructions:

```

...
ADDC (r1, 1, r2)
SUBC (r1, 1, r3)
XOR (r1, r5, r1)
MUL (r2, r6, r0)
...
    
```

Executed on our 2-stage pipeline:




Pipeline Control Hazards

BUT consider instead:

```

LOOP:  ADD(r1, r3, r3)
        CMPLEC(r3, 100, r0)
        BT(r0, LOOP)
        XOR(r3, -1, r3)
        MUL(r1, r2, r2)
        ...
    
```

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADD	CMP	BT	XOR	...		
EXE		ADD	CMP	BT	?	...	


 This is the cycle where the branch decision is made... but we've already fetched the following instruction which should be executed *only* if branch is not taken!

Branch Delay Slots

PROBLEM: One (or more) following instructions have been pre-fetched by the time a branch is taken.

POSSIBLE SOLUTIONS:

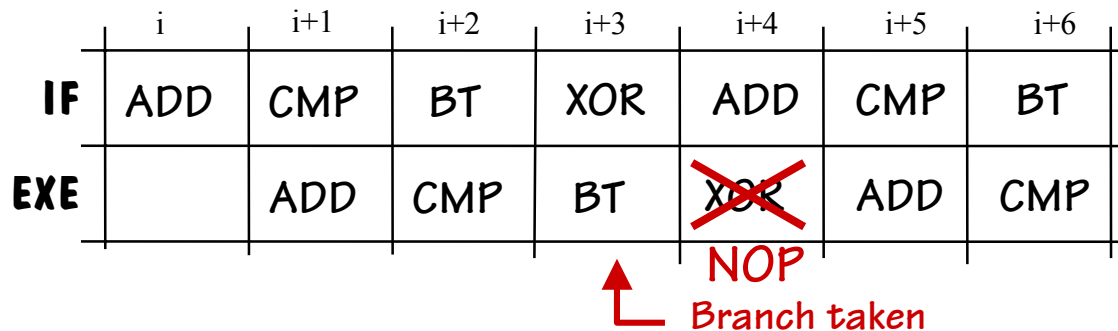
1. Make hardware “annul” instructions following branches which are taken, e.g., by disabling WERF and WR.
2. “Program around it”. Either
 - a) Follow each BR with a NOP instruction; or
 - b) Make compiler clever enough to move USEFUL instructions into the branch delay slots
 - i. Always execute instructions in delay slots
 - ii. Conditionally execute instructions in delay slots

Branch Alternative 1

Make the hardware annul instructions in the branch delay slots of a taken branch.

```

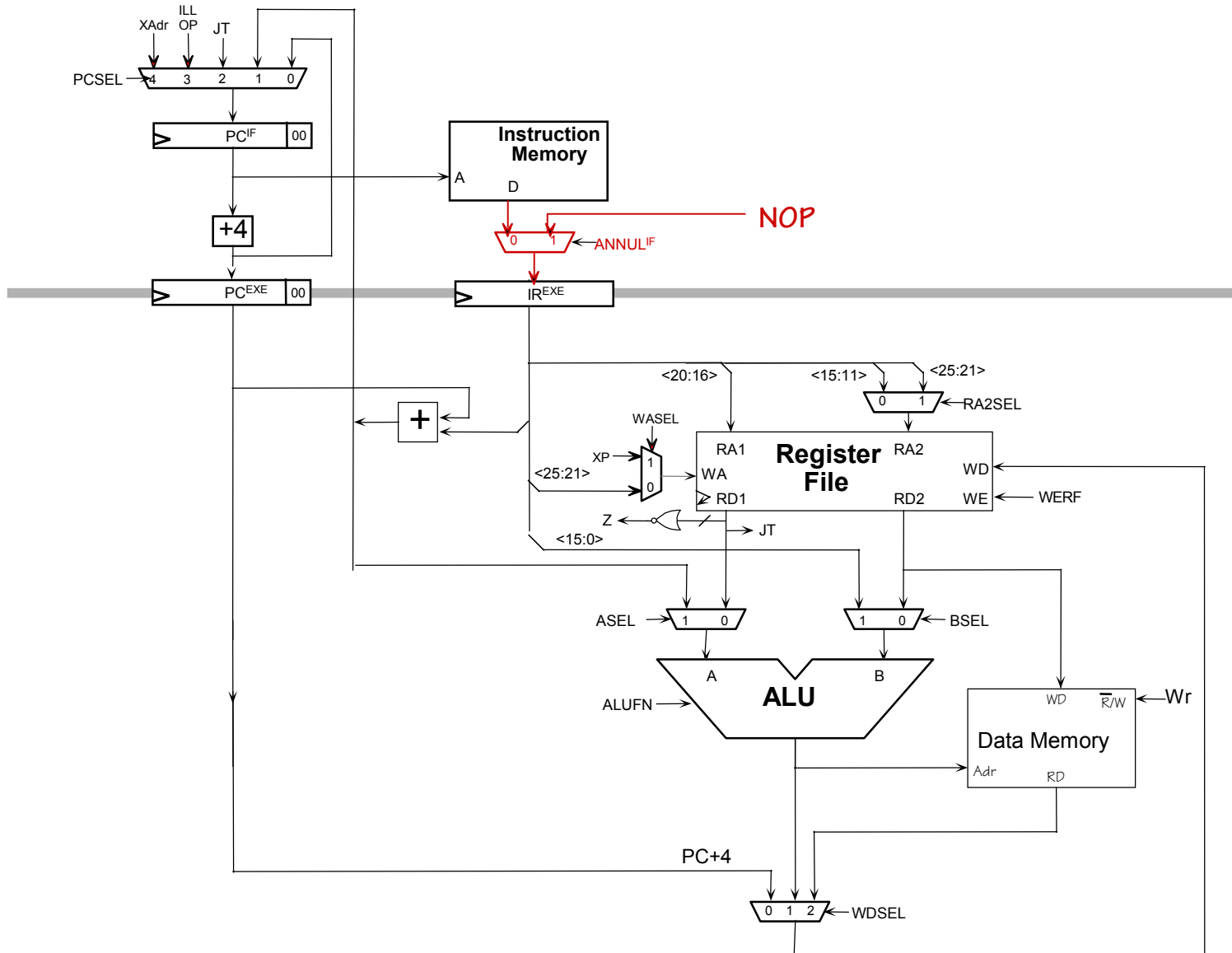
LOOP:  ADD(r1, r3, r3)
        CMPLEQ(r3, 100, r0)
        BT(r0, LOOP)
        XOR(r3, -1, r3)
        MUL(r1, r2, r2)
        ...
    
```



Pros: same program runs on both unpipelined and pipelined hardware

Cons: in SPEC benchmarks 14% of instructions are taken branches →
 12% of total cycles are annulled

Branch Annulment Hardware

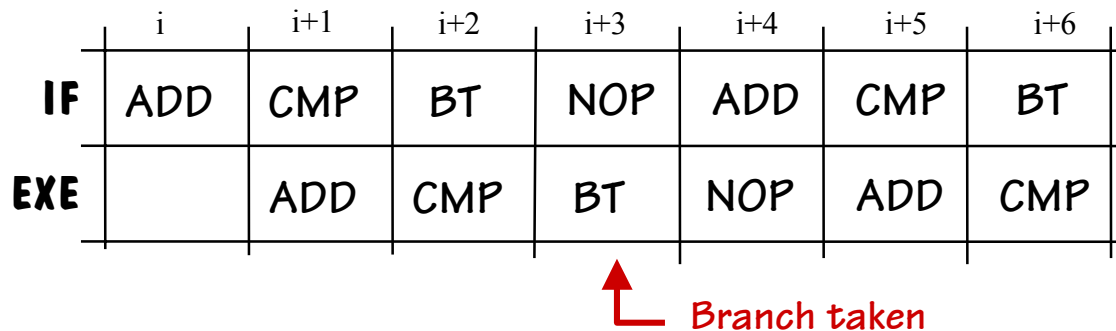


Branch Alternative 2a

Fill branch delay slots with NOP instructions (i.e., the software equivalent of alternative 1)

```

LOOP:  ADD(r1, r3, r3)
        CMPLE(r3, 100, r0)
        BT(r0, LOOP)
        NOP()
        XOR(r3, -1, r3)
        MUL(r1, r2, r2)
        ...
    
```



Pros: same as alternative 1

Cons: NOPs make code longer; 12% of cycles spent executing NOPs

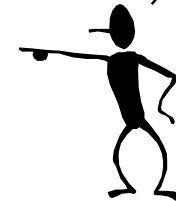
Branch Alternative 2b(i)

Put USEFUL instructions in the branch delay slots; remember they will be executed whether the branch is taken or not

```

LOOP:  ADD (r1, r3, r3)
LOOPx: CMPLEC (r3, 100, r0)
      BT (r0, LOOPx)
      ADD (r1, r3, r3)
      SUB (r1, r3, r3)
      XOR (r3, -1, r3)
      MUL (r1, r2, r2)
      ...
    
```

We need to add this silly instruction to UNDO the effects of that last ADD



	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADD	CMP	BT	ADD	CMP	BT	ADD
EXE		ADD	CMP	BT	ADD	CMP	BT

↑ Branch taken

Pros: only two “extra” instructions are executed (on last iteration)

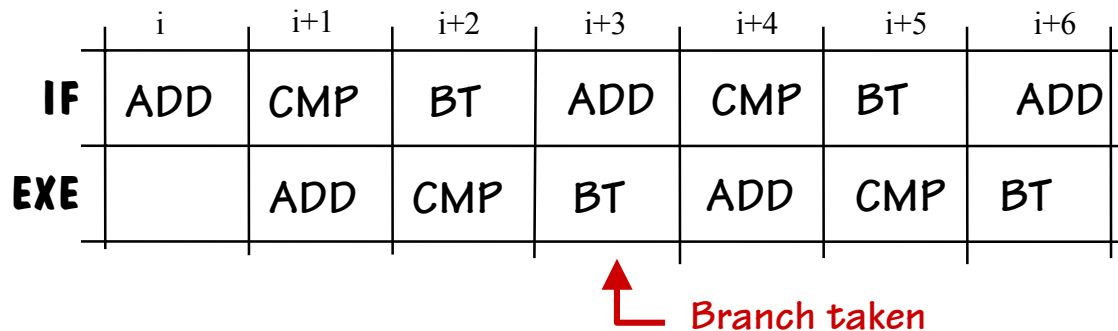
Cons: finding “useful” instructions that are *always* executed is difficult; clever rewrite may be required. Program executes differently on naïve unpipelined implementation.

Branch Alternative 2b(ii)

Put USEFUL instructions in the branch delay slots; annul them if branch *doesn't* behave as predicted

```

LOOP:  ADD(r1, r3, r3)
LOOPx: CMPLEC(r3, 100, r0)
      BT.taken(r0, LOOPx)
      ADD(r1, r3, r3)
      XOR(r3, -1, r3)
      MUL(r1, r2, r2)
      ...
    
```



Pros: only one instruction is annulled (on last iteration); about 70% of branch delay slots can be filled with useful instructions

Cons: Program executes differently on naïve unpipelined implementation; not really useful with more than one delay slot.

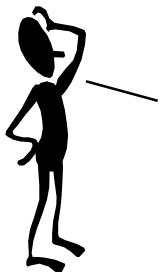
Architectural Issue: Branch Decision Timing

BETA approach:

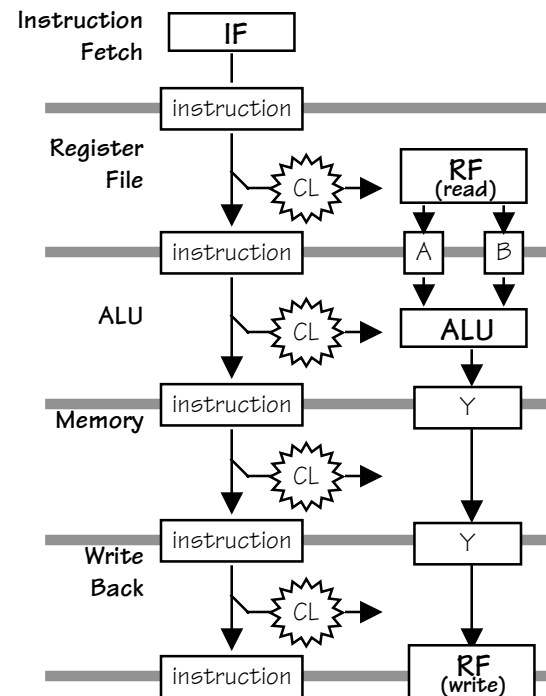
- SIMPLE branch condition logic ...
Test for $\text{Reg}[\text{Ra}] = 0!$
- ADVANTAGE: early decision,
single delay slot

ALTERNATIVES:

- Compare-and-branch...
(eg, if $\text{Reg}[\text{Ra}] > \text{Reg}[\text{Rb}]$)
- MORE powerful, but
- LATER decision (hence more
delay slots)

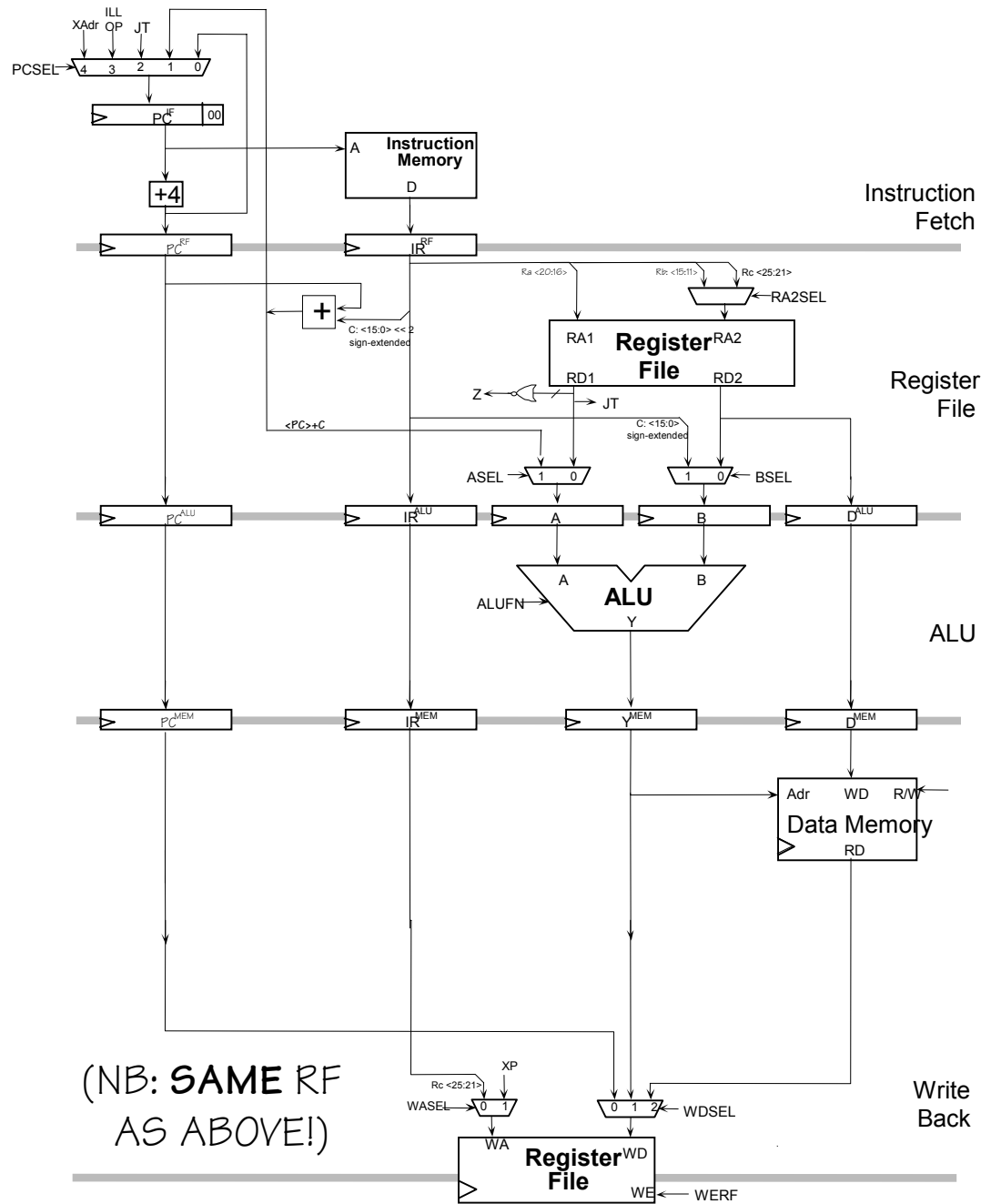


Wow! I guess those guys really were thinking when they made up all those instructions



Suppose decision were made in the ALU stage ... then there would be 2 branch delay slots (and instructions to annul!)

4-Stage Beta Pipeline



Treat register file as two separate devices:
 combinational READ,
 clocked WRITE at end of pipe.

What other information do we have to pass down pipeline?

PC (return addresses)
 instruction fields
 (decoding)

What sort of improvement should expect in cycle time?

4-Stage Beta Operation

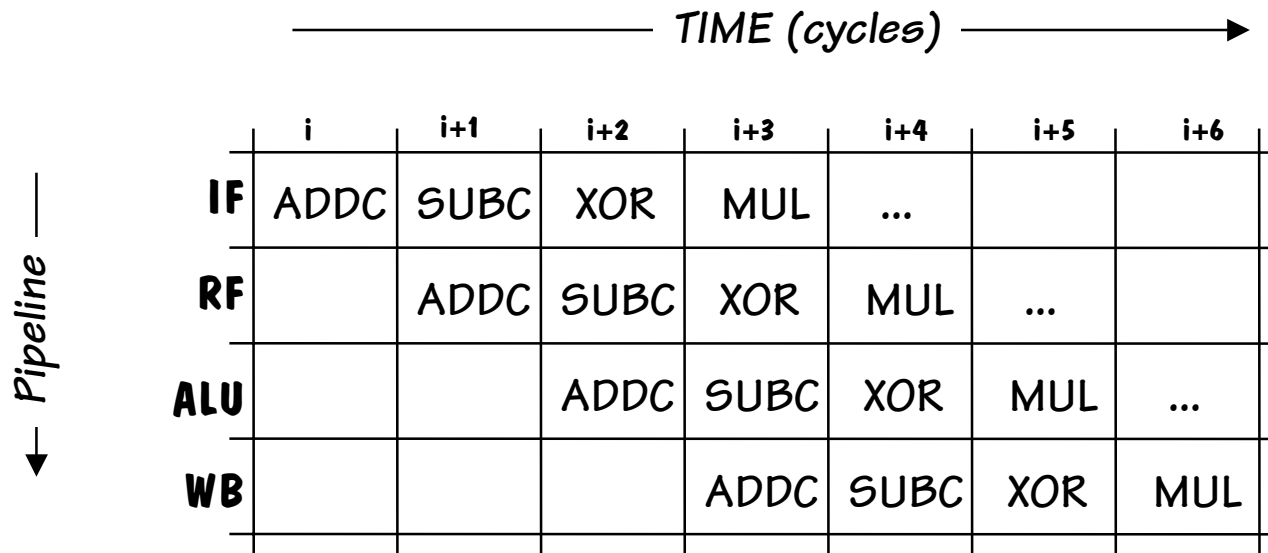
Consider a sequence
of instructions:

```

...
ADDC (r1, 1, r2)
SUBC (r1, 1, r3)
XOR (r1, r5, r1)
MUL (r2, r6, r0)
...

```

Executed on our 4-stage pipeline:



Pipeline "Data Hazard"

BUT consider instead:

ADD(r1, r2, r3)

CMPLEQ(r3, 100, r0)

MULC(r1, 100, r4)

SUB(r1, r2, r5)

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADD	CMP	MUL	SUB			
RF		ADD	CMP	MUL	SUB		
ALU			ADD	CMP	MUL	SUB	
WB				ADD	CMP	MUL	SUB

Oops! CMP is trying to read Reg[R3] during cycle i+2 but ADD doesn't write its result into Reg[R3] until the end of cycle i+3!

Data Hazard Solution 1

“Program around it”

... document weirdo semantics, declare it a software problem.

- Breaks sequential semantics!
- Costs code efficiency.

EXAMPLE: Rewrite

<code>ADD(r1, r2, r3)</code>		<code>ADD(r1, r2, r3)</code>
<code>CMPLEC(r3, 100, r0)</code>		<code>MULC(r1, 100, r4)</code>
<code>MULC(r1, 100, r4)</code>	as	<code>SUB(r1, r2, r5)</code>
<code>SUB(r1, r2, r5)</code>		<code>CMPLEC(r3, 100, r0)</code>

How often can we do this?

Data Hazard Solution 2

Stall the pipeline:

Freeze IF, RF stages for 2 cycles, inserting NOPs into ALU-stage instruction register

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADD	CMP	MUL	MUL	MUL	SUB	
RF		ADD	CMP	CMP	CMP	MUL	SUB
ALU			ADD	NOP	NOP	CMP	MUL
WB				ADD	NOP	NOP	CMP

Drawback: NOPs mean “wasted” cycles

Data Hazard Solution 3

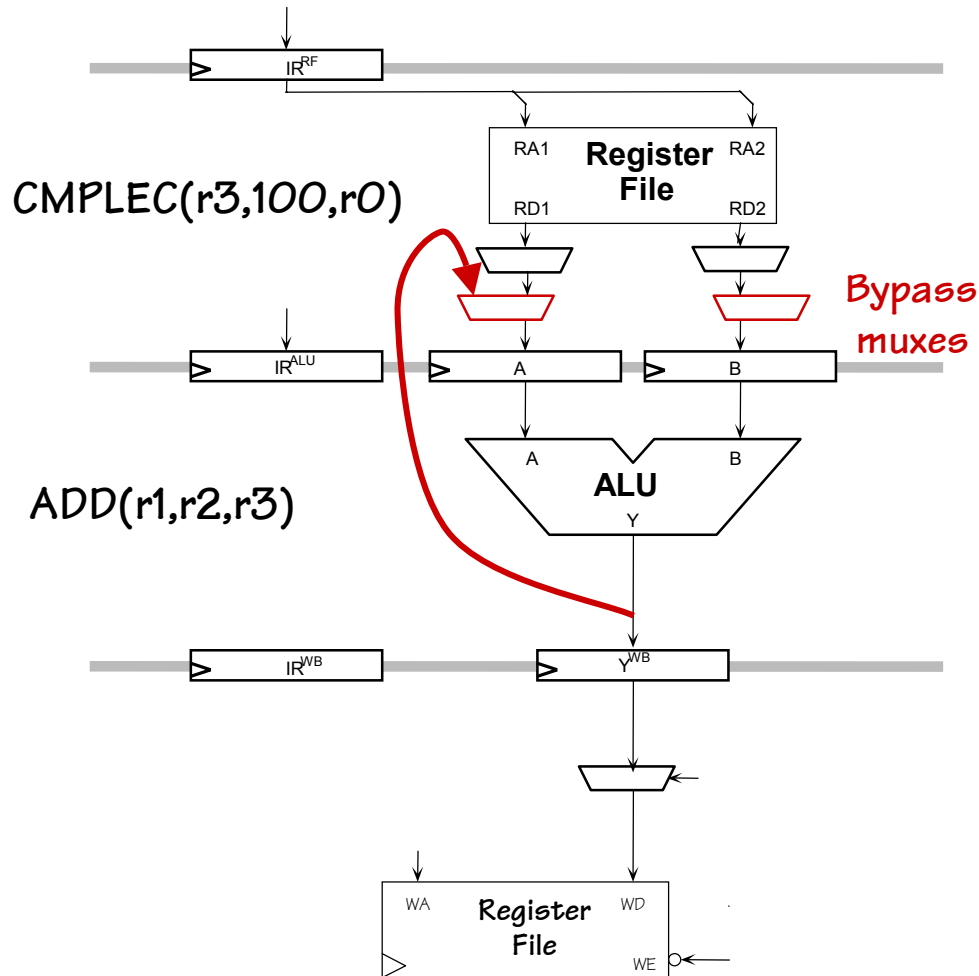
Bypass (aka forwarding) Paths:

Add extra data paths & control logic to re-route data in problem cases.

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADD	CMP	MUL	SUB			
RF		ADD	CMP	MUL	SUB		
ALU			ADD	CMP	MUL	SUB	
WB				ADD	CMP	MUL	SUB

Idea: the result from the ADD which will be written into the register file at the end of cycle $i+3$ is actually available at output of ALU during cycle $i+2$ – just in time for it to be used by CMP in the RF stage!

Bypass Paths (I)



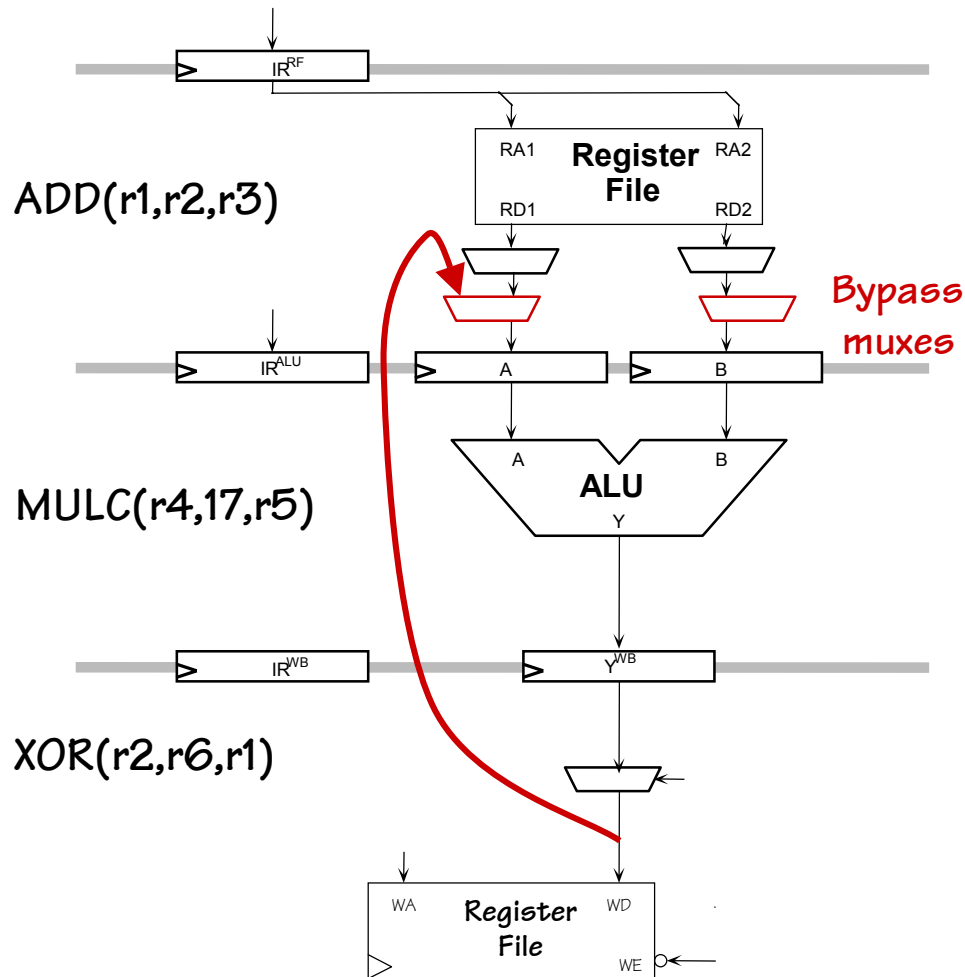
SELECT *this* BYPASS path if
 $OpCode^{RF} = \text{reads } Ra$
and $OpCode^{ALU} = OP, OPC$
and $Ra^{RF} = Rc^{ALU}$

*i.e., instructions which use
 ALU to compute result*

and $Ra^{RF} \neq R31$



Bypass Paths (II)



SELECT *this* BYPASS path if
 $OpCode^{RF} = reads\ Ra$
and $Ra^{RF} \neq R31$
and not using ALU bypass
and $WERF = 1$
and $Ra^{RF} = WA$

*But why can't we get
 it from the register file?
 It's being written this cycle!*



Next Time

