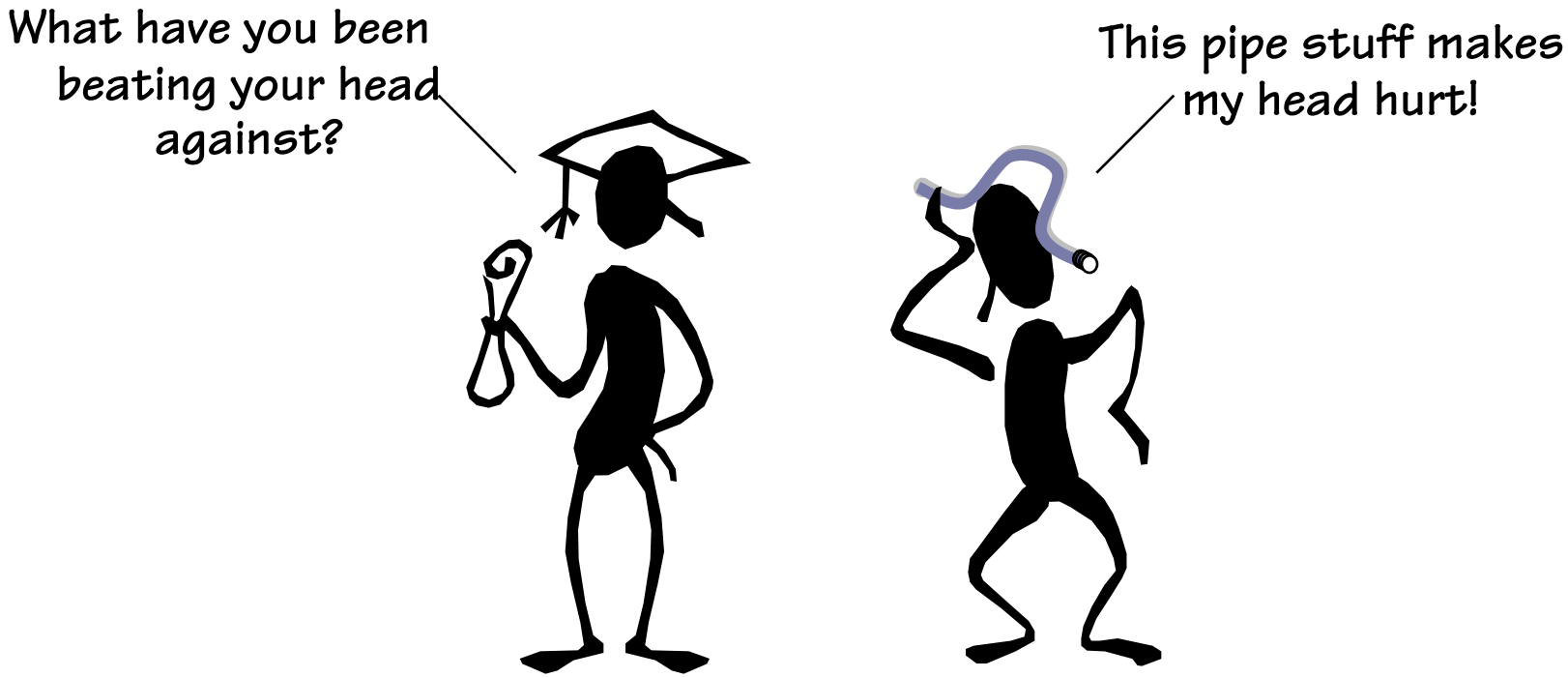
More CPU Pipelining Issues



Important Stuff:

- Study Session for Problem Set 5 tomorrow night (11/11) 5:30-9:00pm
- Study Session for 2nd Midterm on Friday (11/13) during Lab time (1:30-3:00)
- 2nd Midterm next Tuesday (11/17)

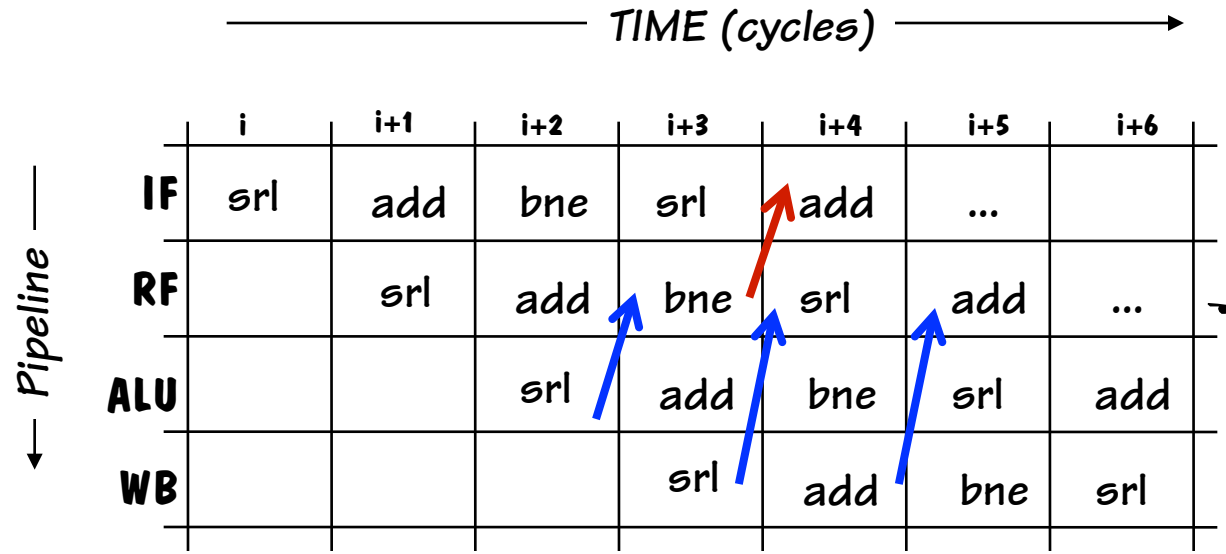
A Quick Review of Last Time

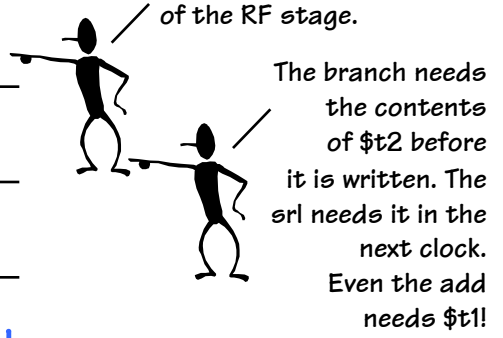
Thus far, while attempting to "speed-up" our miniMIPS using pipelining, we added 4-stages and encountered 2 issues.

Control Hazards: Deal with instructions fetched before a branch or jump determines the next PC.

```
        srl   $t2,$t2,1
loop:   add   $t1,$t1,$t0
        bne   $t2,$0,loop
        srl   $t2,$t2,1
        mov   $v0,$t1
```

New "loop"





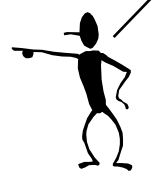
Structural Hazards: When instruction results are needed before they are written into the register file in the WB stage

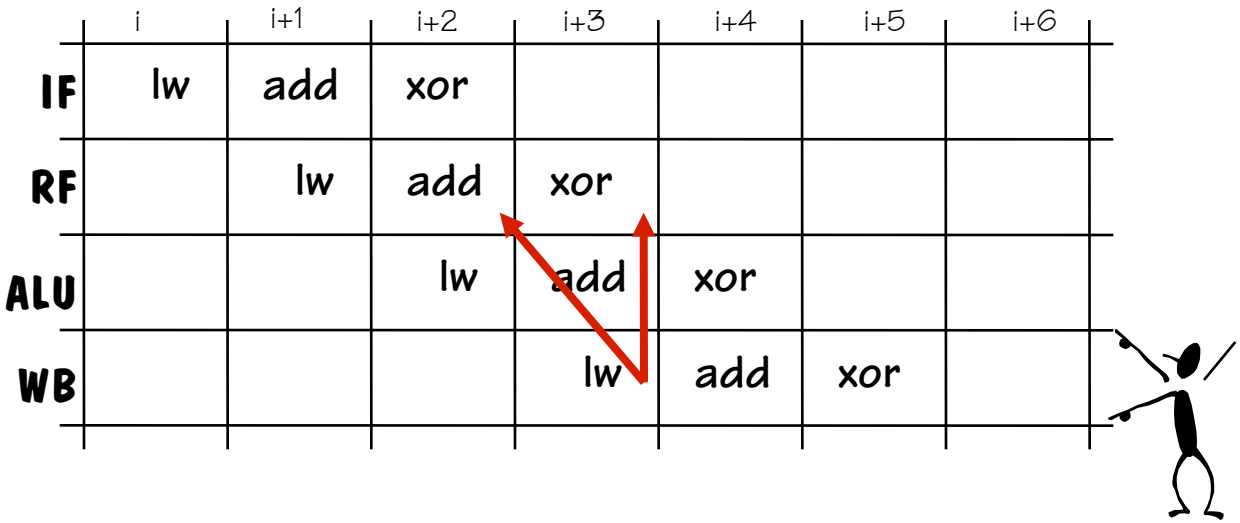
Structural Data Hazard

Consider LOADS:

Can we fix this problem using bypass paths like before?

```
lw   $t4,0($t1)
add  $t3,$t1,$t4
xor  $t6,$t3,$t4
```





Load data hazards are complicated by the fact that their result is resolved later than the ALU pipeline stage.

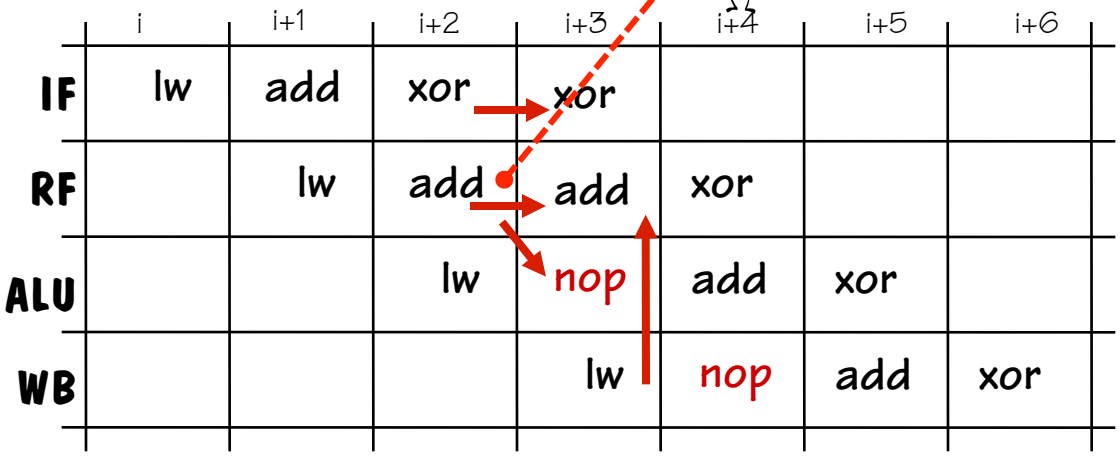
For a `lw` instruction fetched during cycle *i*, data isn't returned from memory until late into cycle *i+3 (in a 4-stage pipeline)*.
Bypassing will fix `xor` but not `add`!

Load Delays

Bypassing CAN'T fix the problem with add since the data simply isn't available! In order to fix it we have to add *pipeline interlock hardware* to stall the add's execution, or else program around it.

Here's where the add detects it needs a result from the lw instruction. It then stalls the pipe, and inserts a bubble.

```
lw  $t4,0($t1)
add $t5,$t1,$t4
xor $t6,$t3,$t4
```



Recall, adding stalls to the pipeline in order to assure proper operation is called inserting pipeline BUBBLES

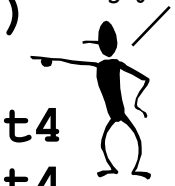
This requires "inserting a MUX" just before the instruction register of the ALU stage, IR^{ALU}, to insert a NOP, and "clock enables" on the PC and IR pipeline registers of earlier pipeline stages to stall the pipeline. Unlike branching, no instructions are annulled.

Punting on Load Interlock

Early versions of MIPS did not include a pipeline interlock, thus, requiring the compiler/programmer to work around it. Old code still worked once interlocks were added, it just used more memory.

If you put an instruction here it was not allowed to access $t4, thus complicating the ISA. S/W guys rebelled.

```
lw   $t4,0($t1)
nop
add  $t5,$t1,$t4
xor  $t6,$t3,$t4
```



	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	lw	nop	add	xor			
RF		lw	nop	add	xor		
ALU			lw	nop	add	xor	
WB				lw	nop	add	xor

OMG! What if there was a lw instruction in a branch-delay slot? This is getting complicated!

If compiler knows about load delay, it can often rearrange the code sequence to eliminate the hazard. Many compilers can provide implementation-specific *instruction scheduling*. This requires no additional H/W, but it leads to awkward instruction semantics. We'll include interlocks in miniMIPS.

Load Delays (cont'd)

But, what about FASTER processors?

FACT: Processors have been become very fast relative to memories!

Can we just stall the pipe longer? Add more NOPs?

ALTERNATIVE: Longer pipelines.

1. Add "MEMORY WAIT" stages between INITIATION of load operation and when it returns data.
2. Build pipelined memories, so that multiple (say, N) memory transactions can be in progress at once.
3. (Optional). Stall pipeline when the N limit is exceeded.

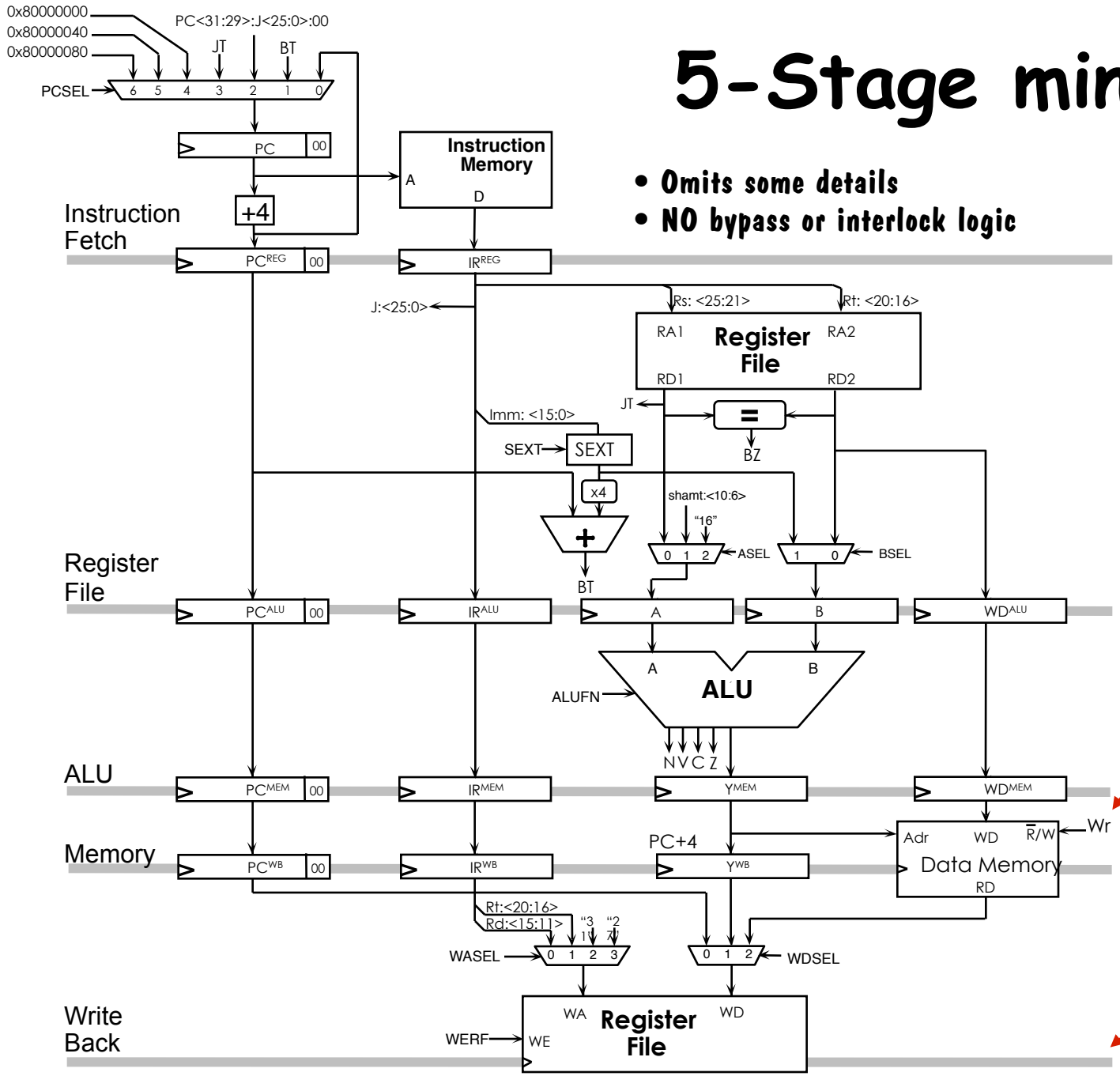
4-Stage pipeline requires READ access in LESS than one clock.

Sadly, this IS the bottleneck of most CPUs. If we want to go faster will have to surround it with pipeline stages

SOLUTION: A 5-Stage pipeline that allows nearly two clocks for data memory accesses...

5-Stage miniMIPS

- Omits some details
- NO bypass or interlock logic



Address is available right after instruction enters Memory stage

almost 2 clock cycles

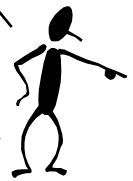
Data is needed just before rising clock edge at end of Write Back stage

One More Fly in the Ointment

There is one more structural hazard that we have not discussed. That is, the saving, and subsequent accesses, of the return address resulting from the jump-and-link, `jal`, and `jalr` instructions.

Moreover, given that we have bought into a single delay-slot, which is always executed, we now need to store the address of the instruction FOLLOWING the delay slot instruction.

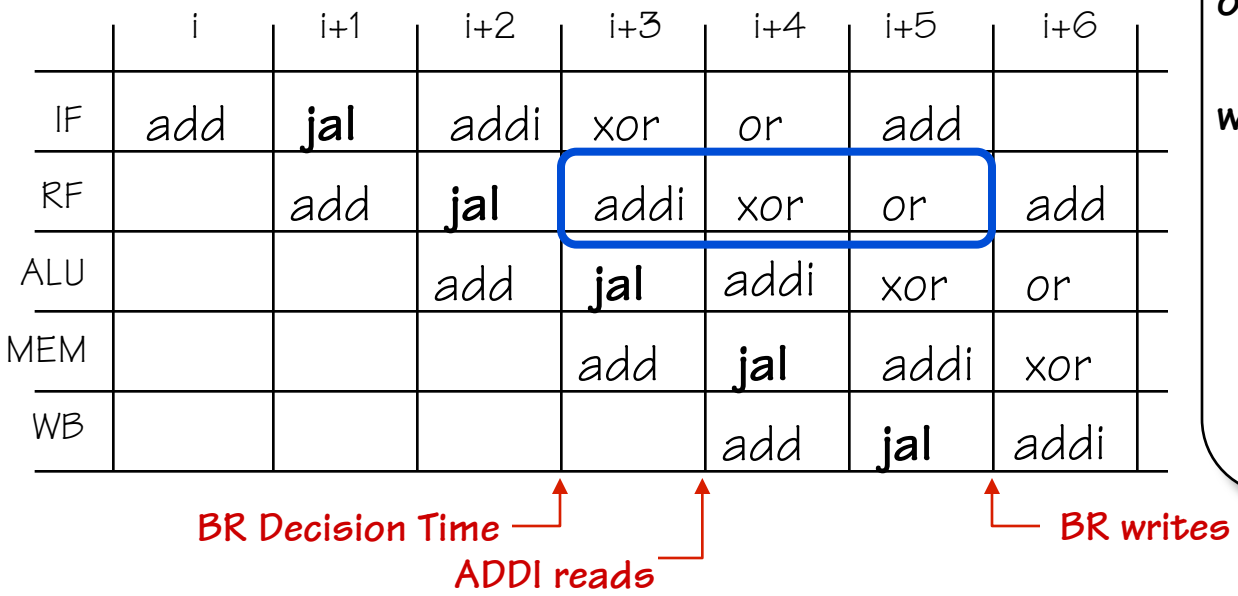
We need to return here, to PC+8, not PC+4. Once more we need to rewrite the ISA spec!



```
jal    sqr           # call procedure
addi   $a0,$0,10     # set arg in delay slot
addi   $t0,$v0,-1    # return address
```

Return Address Register Writes

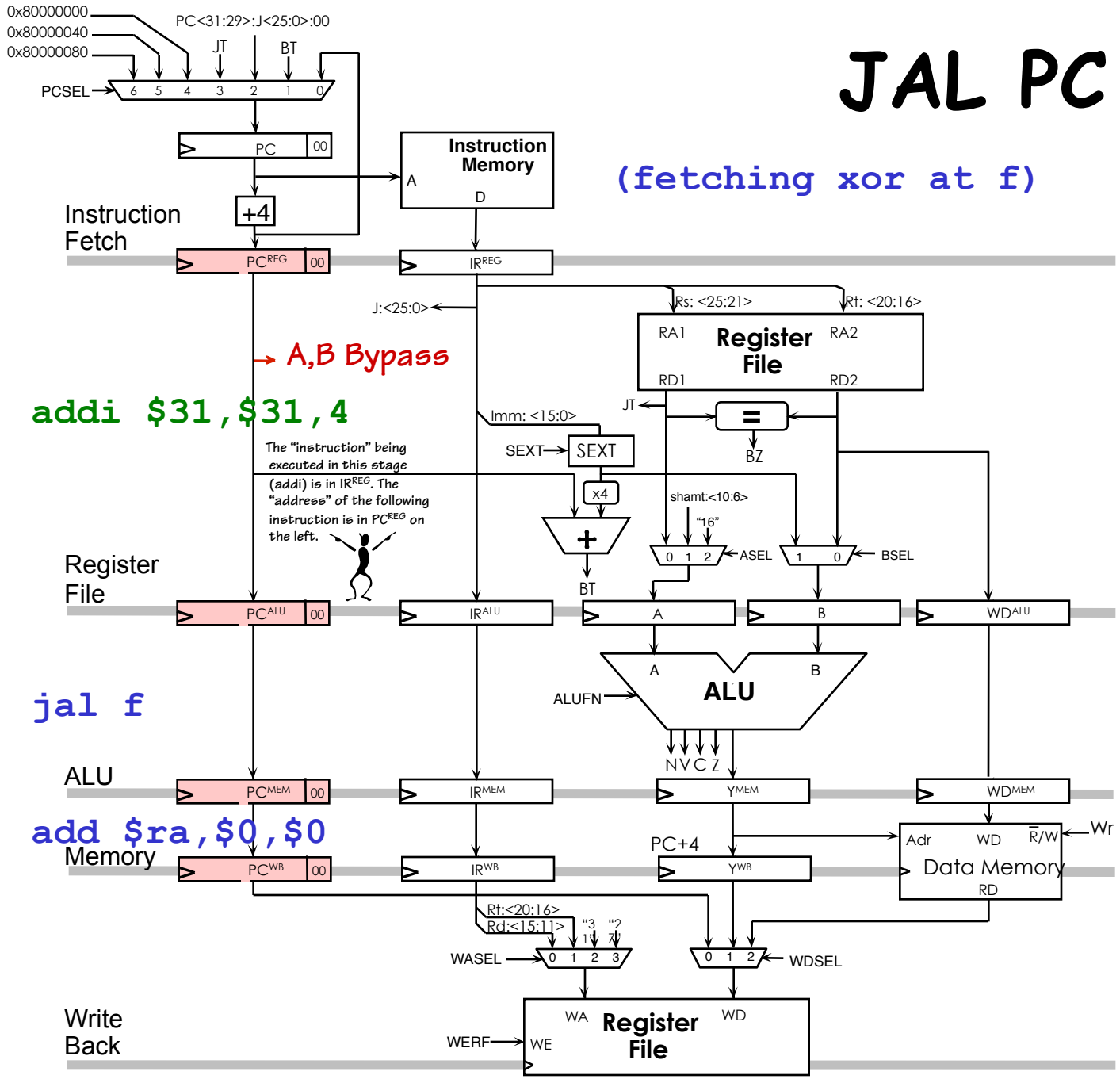
```
# The code: Does crazy stuff!
        add  $ra,$0,$0
        jal  f
        addi $ra,$ra,4  # In delay slot
        ...
f:      xor  $t0,$ra,$0
        or   $r1,$0,$ra
        add  $t2,$0,$ra
```



Can we make the register accesses of the 3 instructions following the jal work with bypassing?

Where do we get the right return address from?

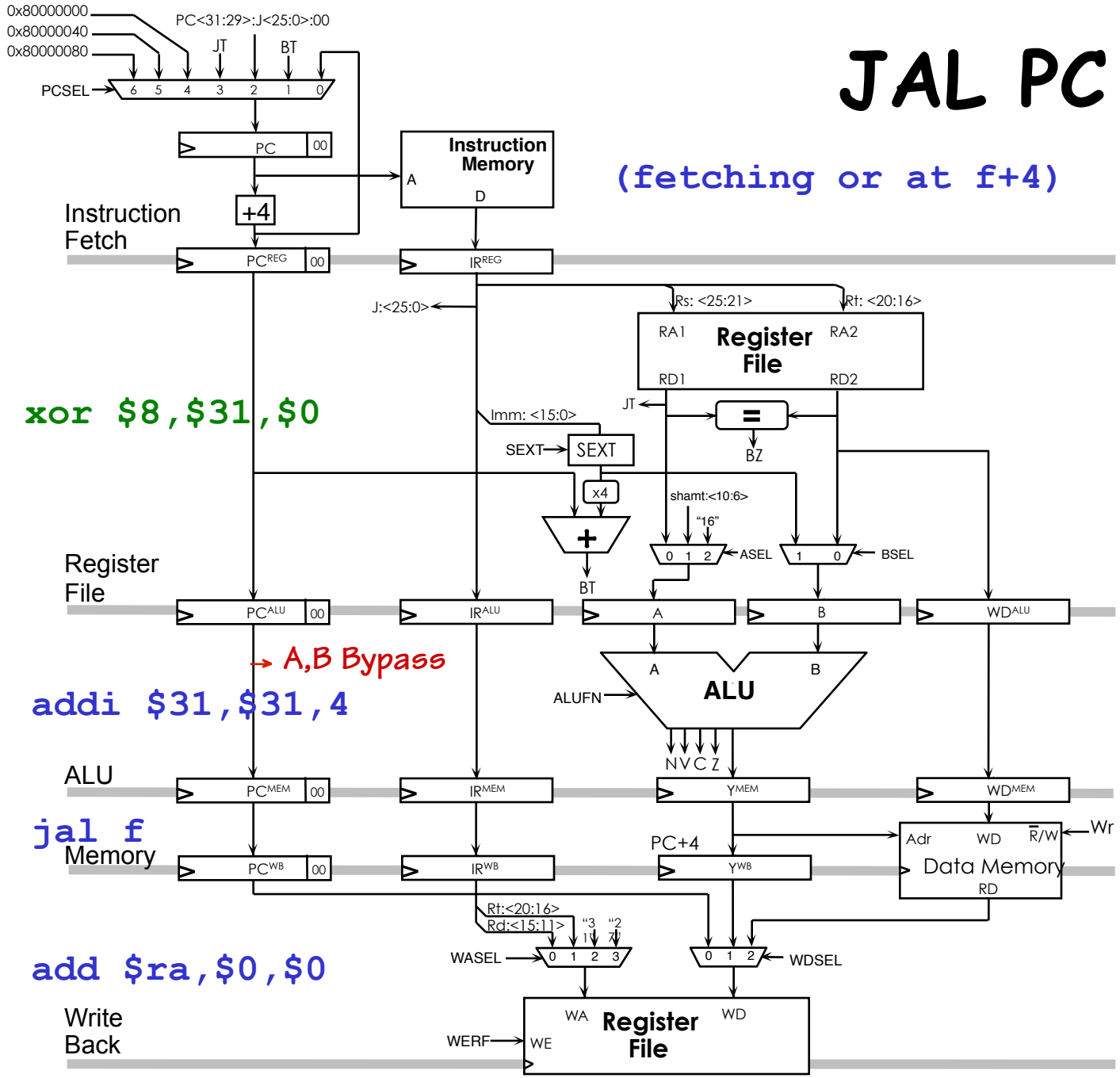
JAL PC Bypasses



On JALs, we need to store the next PC of the DELAY SLOT instruction (often PC+8).

Note this bypass is routed from the PC pipeline not from the ALU output. Thus, we need to add bypass paths for PC^{MEM}.

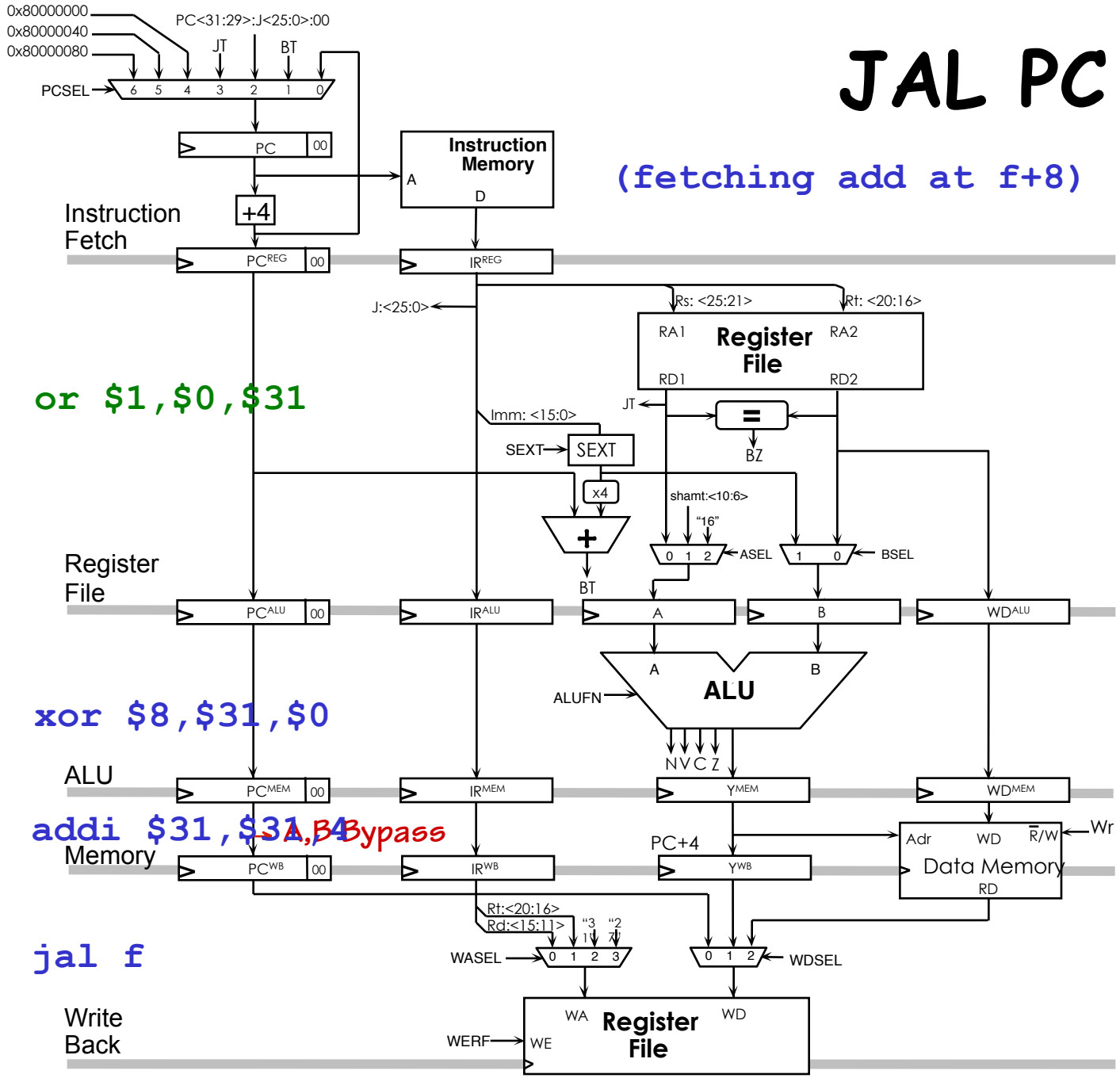
JAL PC Bypasses



We need another PC^{ALU} bypass.

In this case, the bypass path supplies the $31 operand for the XOR instruction.

JAL PC Bypasses

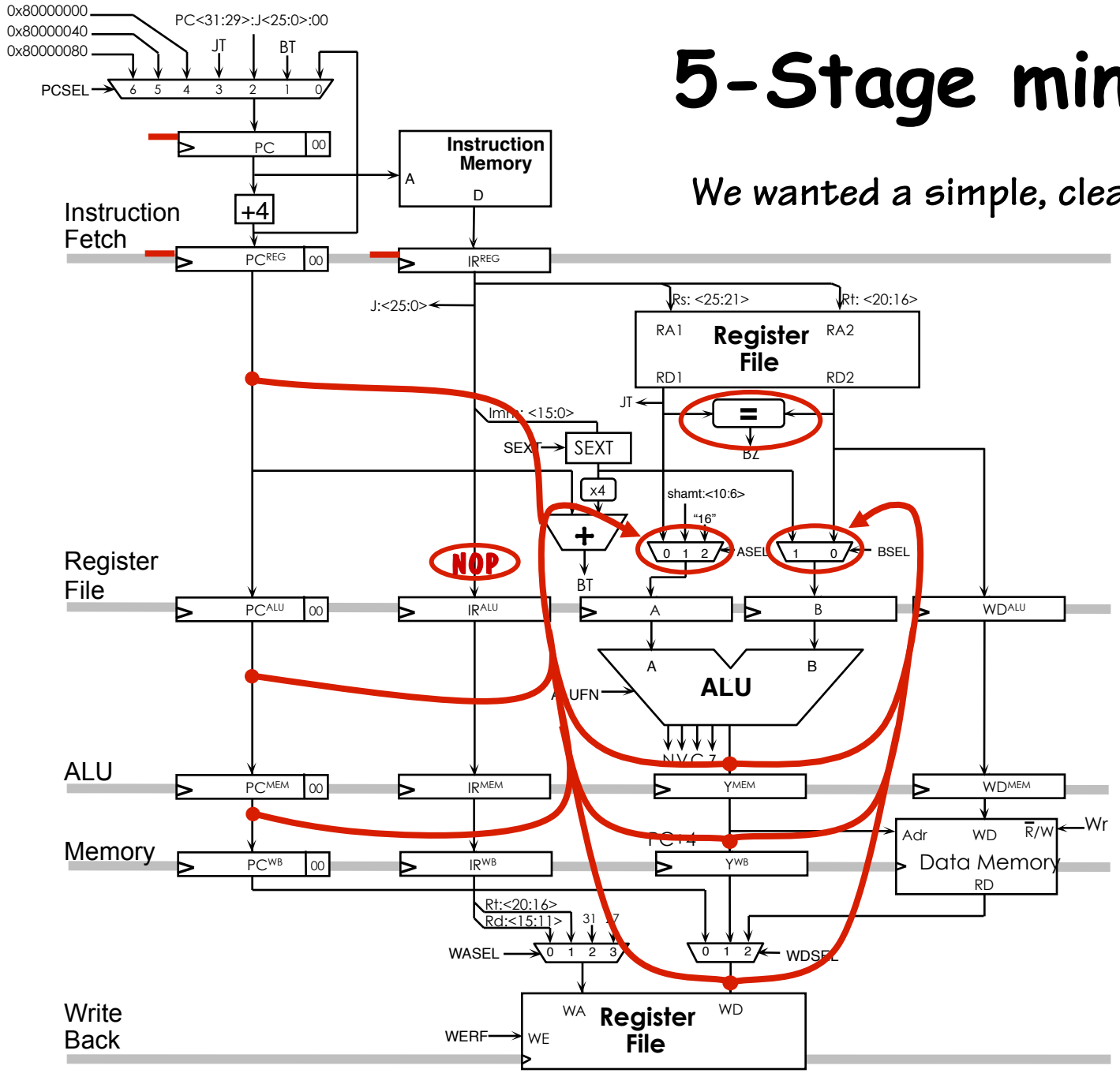


And, we need another PC^{MEM} bypass.

In this case, the bypass path supplies the $31 operand for the OR instruction.

PC^{WB} is already taken care of, for the following ADD, using the WB stage bypass at the output of the WDSEL mux.

5-Stage miniMIPS

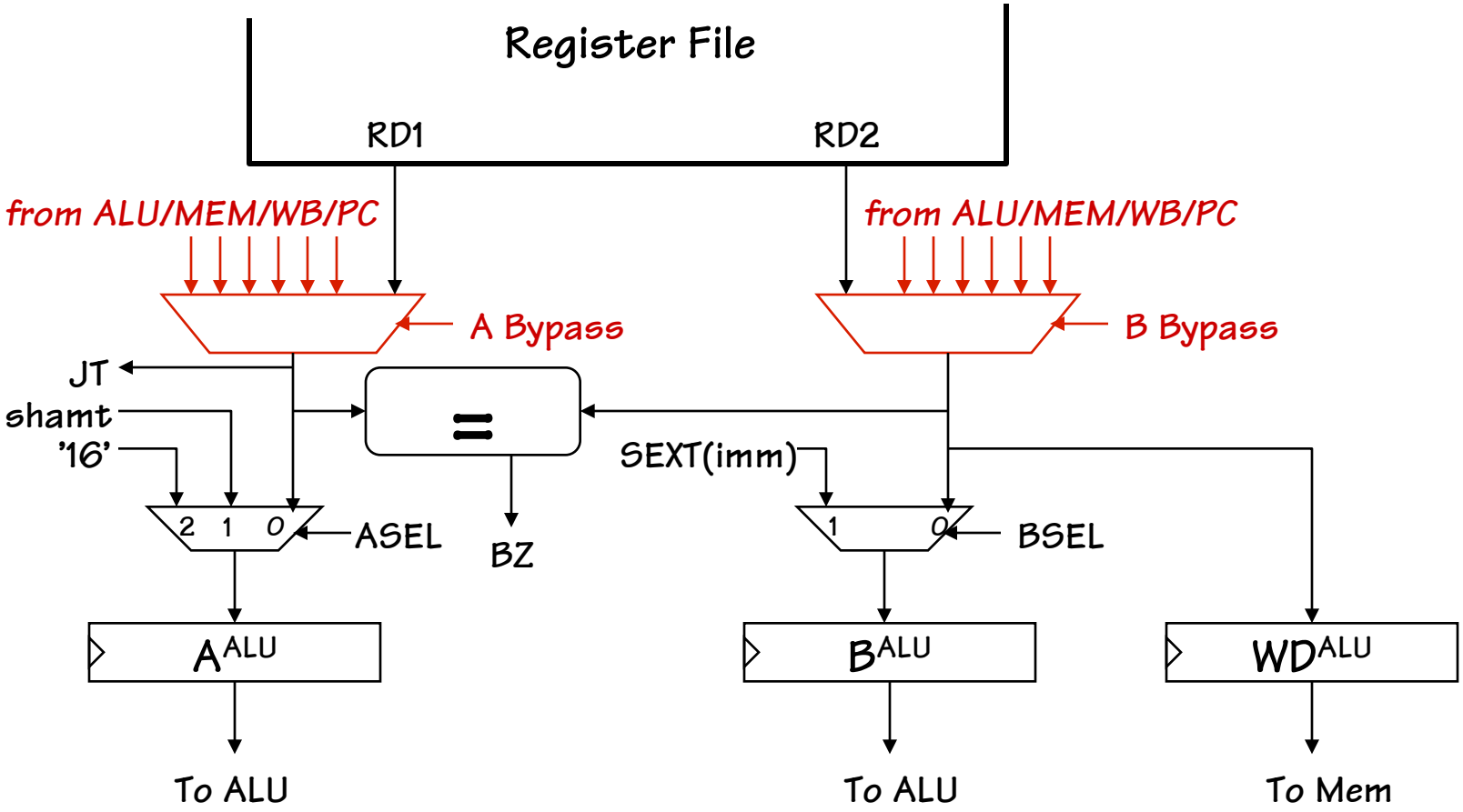


We wanted a simple, clean pipeline but…

- broke the sequential semantics of ISA by adding a branch delay-slot and early branch resolution logic
- added A/B bypass muxes to get data before it's written to regfile
- added CLK EN to freeze IF/RF stages so we can wait for `lw` to reach WB stage

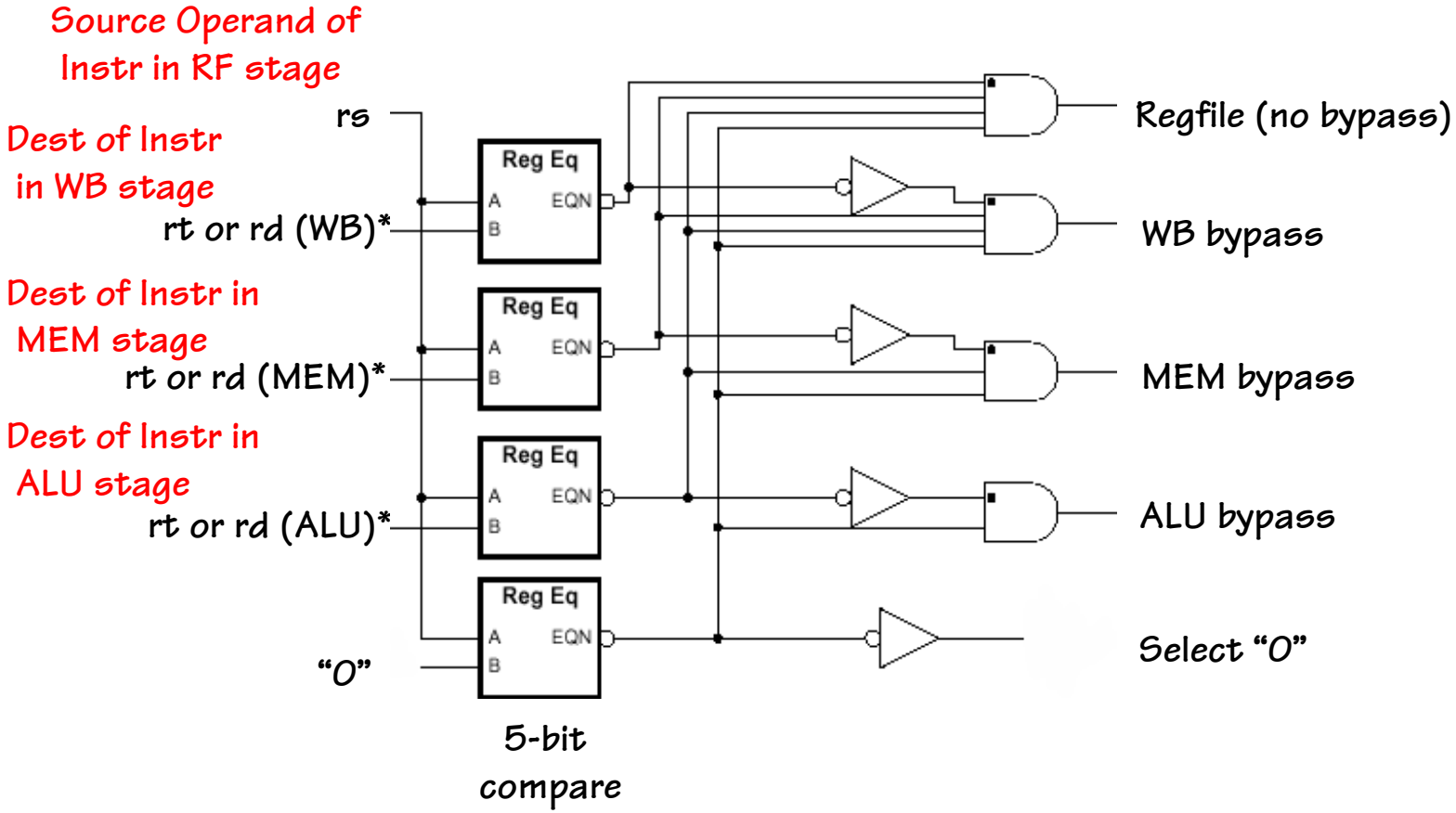
Bypass MUX Details

The previous diagram was oversimplified. Really need for the bypass muxes to precede the A and B muxes to provide the correct values for the jump target (JT), write data, and early branch decision logic.



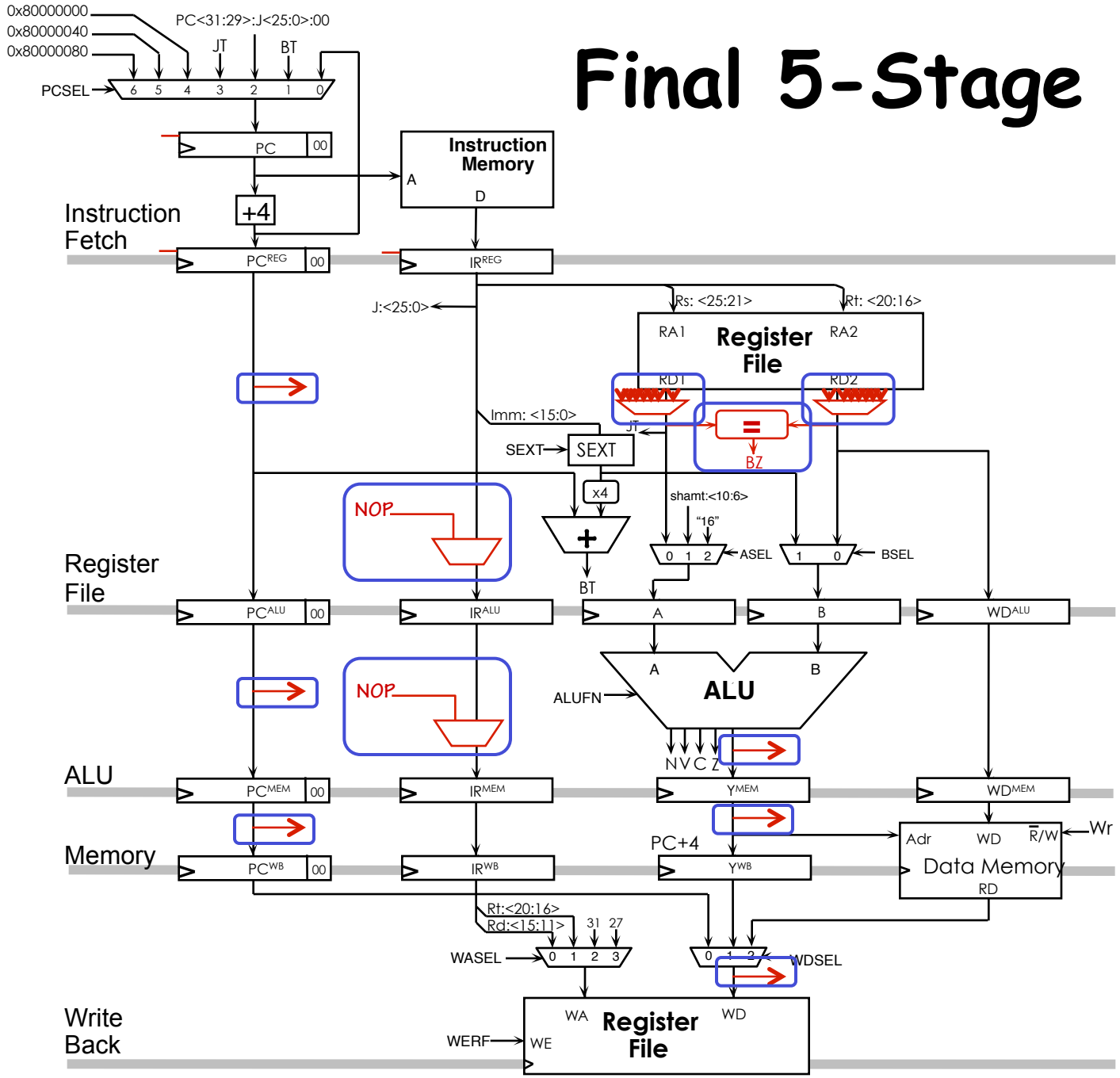
Bypass Logic

miniMIPS A bypass logic
(need another copy for B bypass that compares to rt rather than rs):



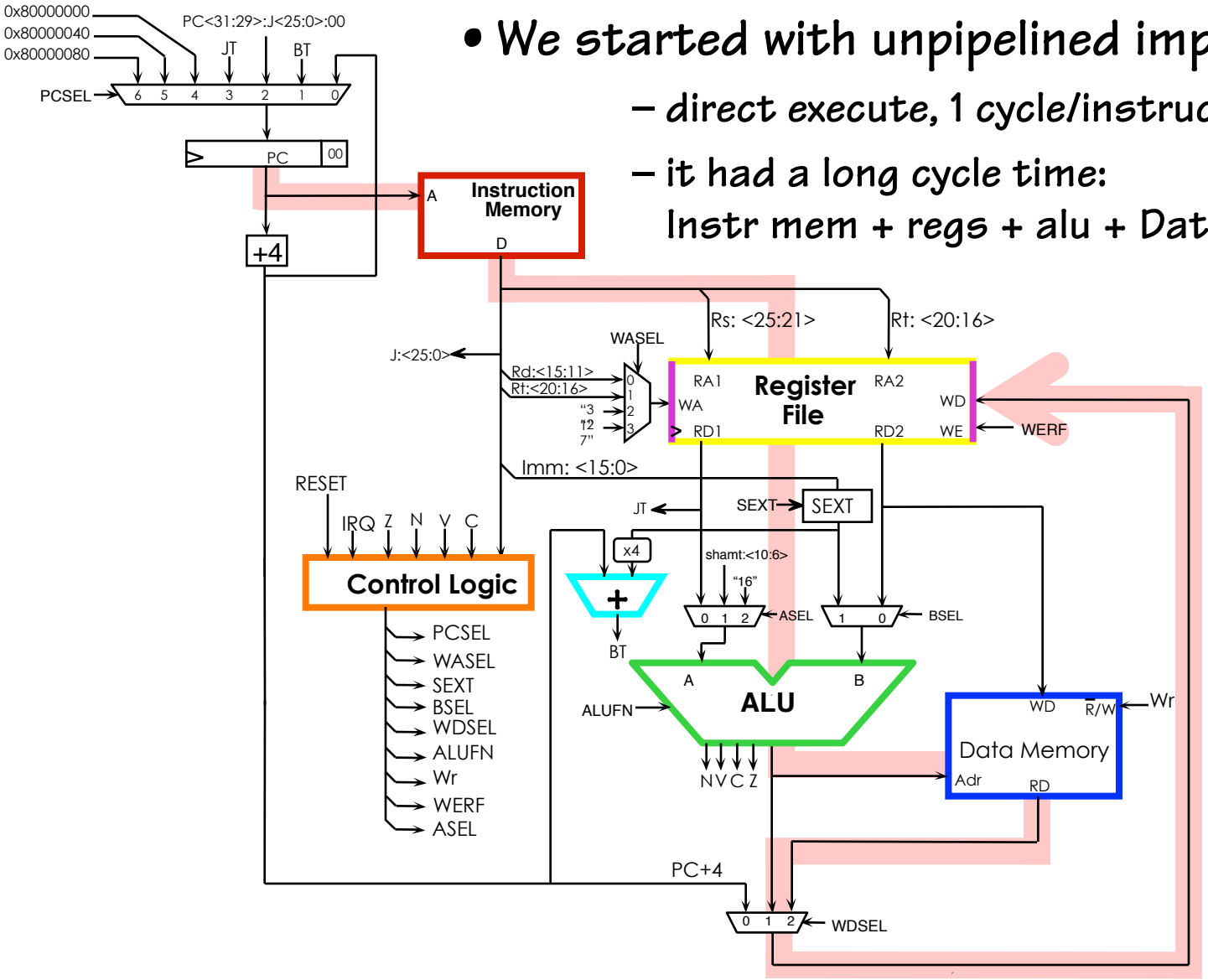
* If instruction is a sw (doesn't write into regfile), set rt for ALU/MEM/WB to $0

Final 5-Stage miniMIPS



- Added branch delay slot and early branch resolution logic to fix a CONTROL hazard
- Added lots of bypass paths and detection logic to fix various STRUCTURAL hazards
- Added pipeline interlocks to fix load delay STRUCTURAL hazard

CPU Pipeline Summary (I)



- We started with unpipelined implementation
  - direct execute, 1 cycle/instruction
  - it had a long cycle time:
    Instr mem + regs + alu + Data mem + wb

CPU Pipeline Summary (II)

- We ended up with a 5-stage pipelined implementation. It increased throughput (3x-4x), but it had impact
  1) We added delayed branch decisions (1 stage)

     Chose to *always* execute instruction after branch, so pipelined code does not work the same as before (Changed ISA).

  2) We added bypasses to forward results ahead of register write-back stage (3 stages)

     Did not impact instruction semantics, but it adds delays (due to 2 six-input Muxes) to forward correct values.

  3) We stall the CPU if any of the 2 instructions following a LW reference its destination register

     Introduced NOPs at IR^{RF} and IR^{ALU}, to stall until LW result was ready. No impact on ISA, but timing of LW varies. (1, 2, or 3 clocks)

CPU Pipeline Summary (III)

Fallacy #1: Pipelining is easy

Smart people get it wrong all of the time! Costs? Re-spins of the design. Force S/W folks to devise program/compiler workarounds.

Fallacy #2: Pipelining is independent of ISA

Many ISA decisions impact how easy/costly it is to implement pipelining (i.e. branch semantics, addressing modes). Bad decisions impact future implementations. (delay slot vs. annul?, load interlocks?) and break otherwise clean semantics. For performance, S/W must be aware!

Fallacy #3: Increasing Pipeline stages improves performance

Diminishing returns. Increasing complexity. Can introduce unusable delay slots, long interlock stalls.

Fallacy of my Generation
RISC == Simplicity???

"The P.T. Barnum World's Tallest Dwarf Competition"
World's Most Complex RISC?

- RISC was conceived to be SIMPLE
- SIMPLE --> FAST
- MORE SPEED --> Pipelining
- Pipelining --> Complexity
- Complexity increases delays and impacts S/W (Compilers must schedule loads and reorganize code)
- Other ways to improve performance (reduce CPI)