

# CPU Pipelining Issues

What have you been beating your head against?

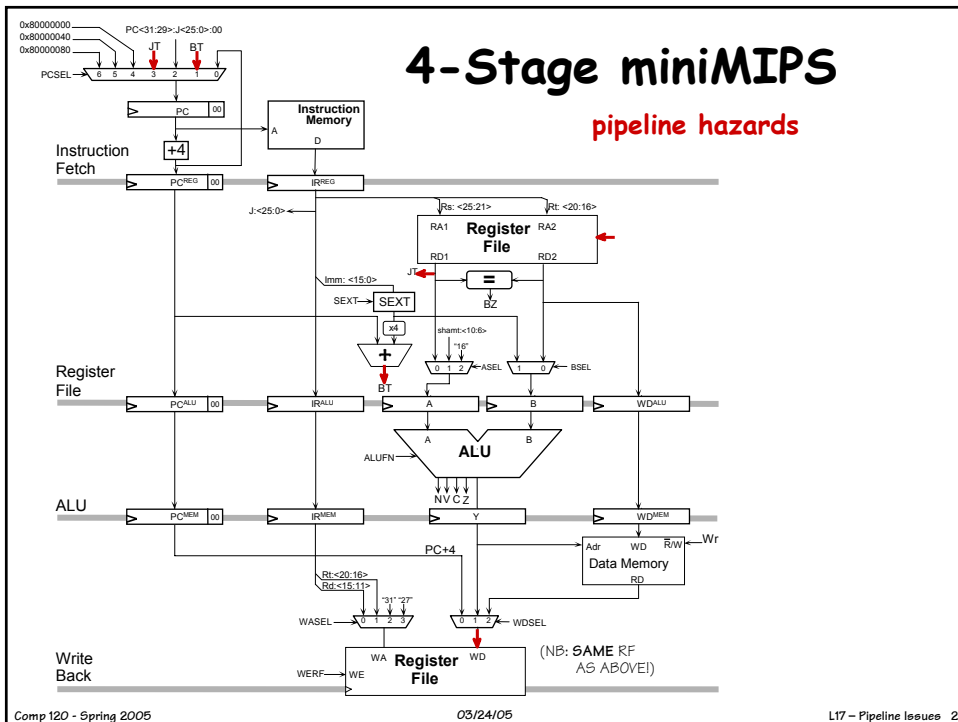


This pipe stuff makes my head hurt!

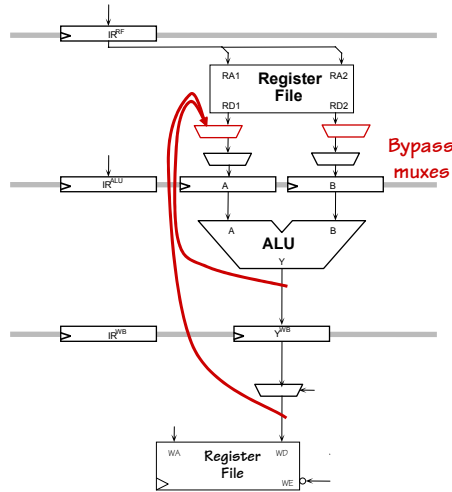


## 4-Stage miniMIPS

pipeline hazards



# Bypass Paths

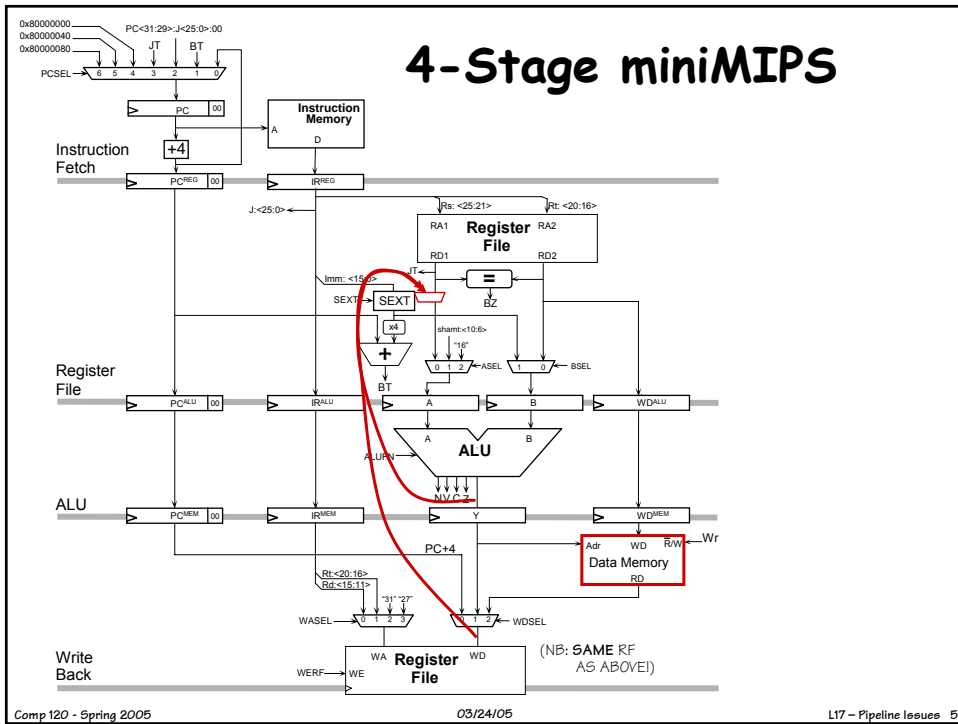


# Structural Data Hazard

Consider LOADS:  
 Can we fix all these  
 problems using our  
 previous bypass paths?

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

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



## Structural Data Hazard

Consider LOADS:  
Can we fix all these problems using our previous bypass paths?

Can we fix all these problems using our previous bypass paths?

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

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

For a lw instruction fetched during clock i, data isn't returned from memory until late into cycle i+3. Bypassing will fix xor but not add!

Comp 120 - Spring 2005 03/24/05 L17 - Pipeline Issues 6

## 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* add's execution, or else program around it.

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

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

Adding stalls to the pipeline in order to assure proper operation is sometimes called inserting pipeline BUBBLES



This requires inserting a MUX just before the instruction register of the ALU stage, IR<sup>ALU</sup>, to annul the add (by inserting a NOP) as well as, clock enables on the PC and IR pipeline registers of earlier pipeline stages to stall the execution without annulling any instructions. This is how the simulator, SPIM works.

## Punting on Load Interlock

Early versions of MIPS did not include a pipeline interlock, thus, requiring the compiler/programmer to work around it.

```
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

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 further confuses the instruction semantics. We'll include interlocks for SPIM compatibility.

# Load Delays (cont'd)

But, 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.

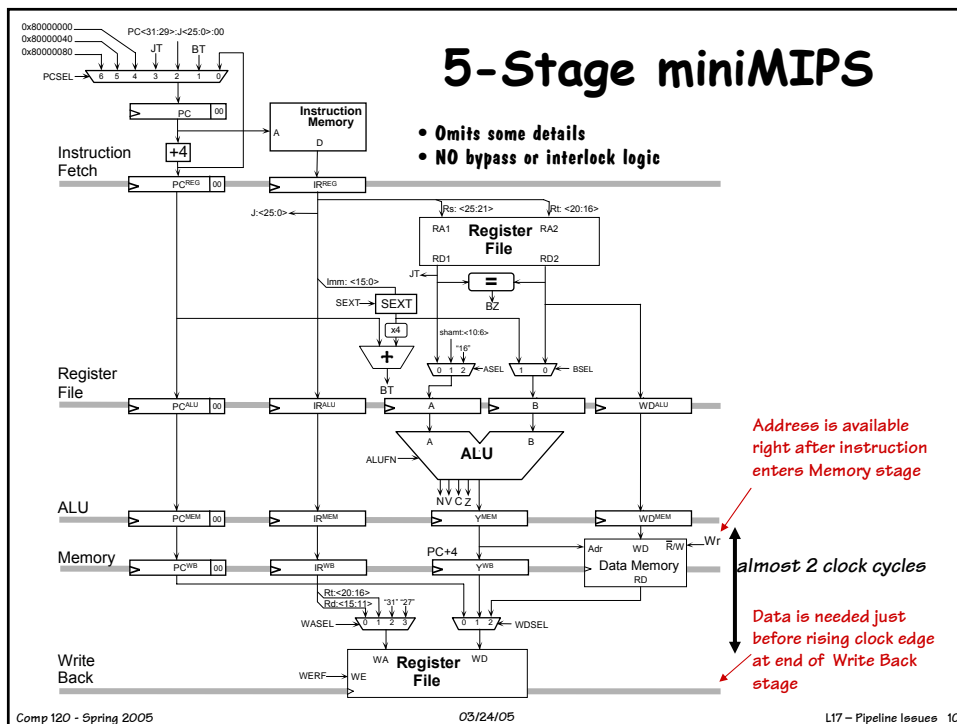
1. Add "MEMORY WAIT" stages between START of read operation & return of data.
2. Build pipelined memories, so that multiple (say, N) memory transactions can be in progress at once.
3. (Optional). Stall pipeline when the N limit is exceeded.

Sadly, this is our bottleneck. If we want to go faster will have to surround it with pipeline stages

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



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



## 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, instruction.

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    # delay slot
addi   $t0,$v0,-1   # return address
    
```

## Return Address Register Writes

```

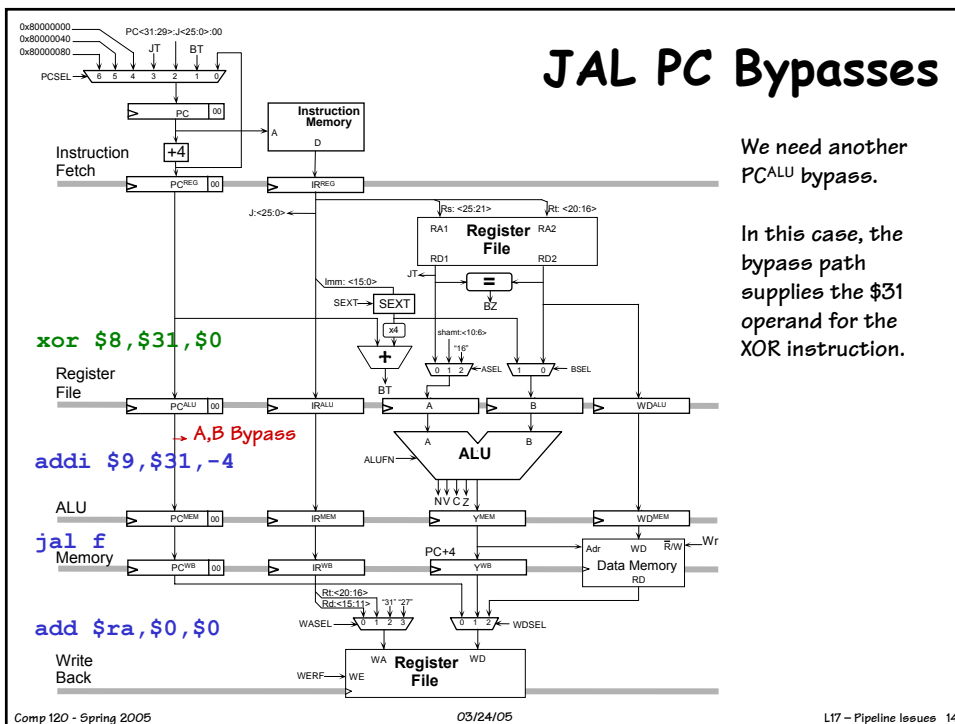
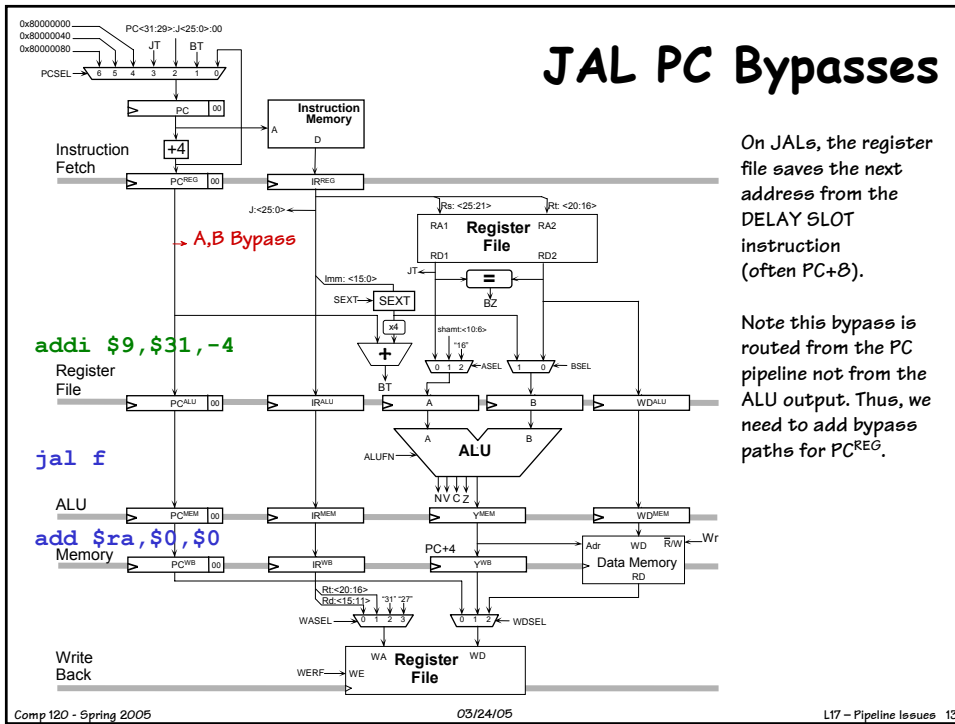
# The code: Assume Reg[LP] = 100...
add    $ra,$0,$0
jal    f
addi   $t1,$ra,-4 # In delay slot
...
f:     xor    $t0,$ra,$0
       or     $r1,$0,$ra
       add    $t2,$0,$ra
    
```

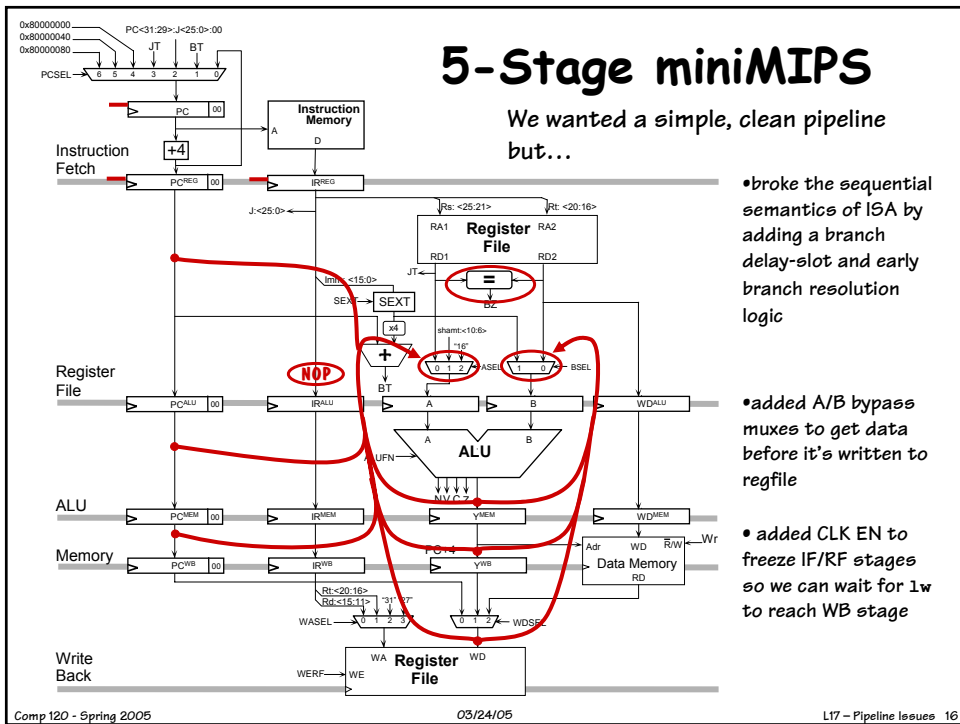
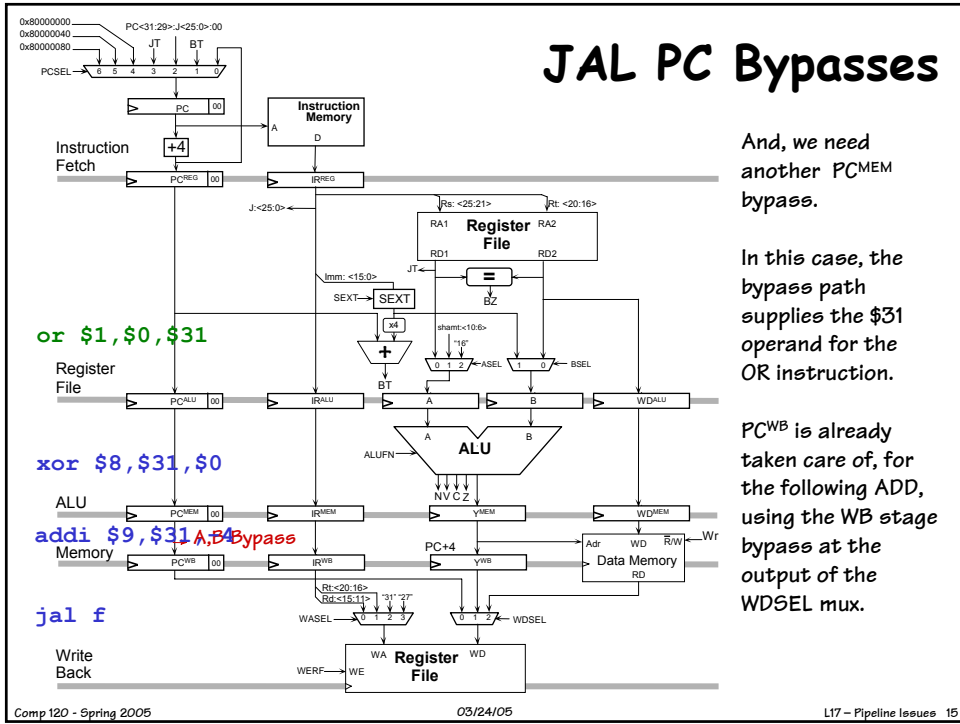
	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	add	jal	addi	xor	or	add	
RF		add	jal	addi	xor	or	add
ALU			add	jal	addi	xor	or
MEM				add	jal	addi	xor
WB					add	jal	addi

BR Decision Time ↑  
 ADDI reads ↑  
 BR writes ↑

Can we make the regfile accesses of the 3 instructions following the jal work by bypassing?

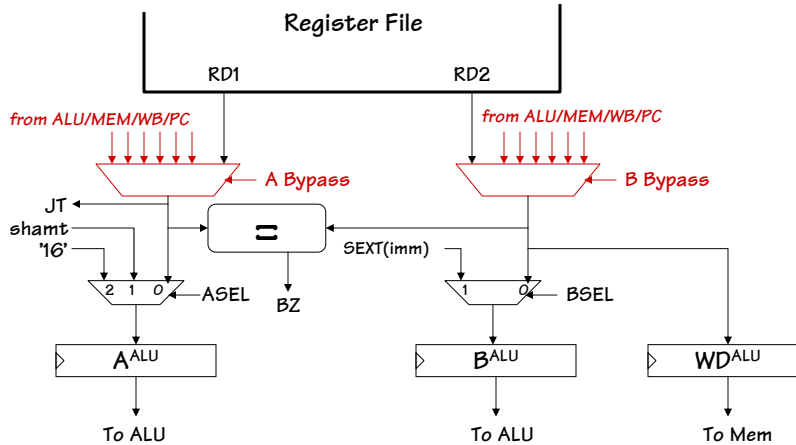
Where do we get the right return address from?





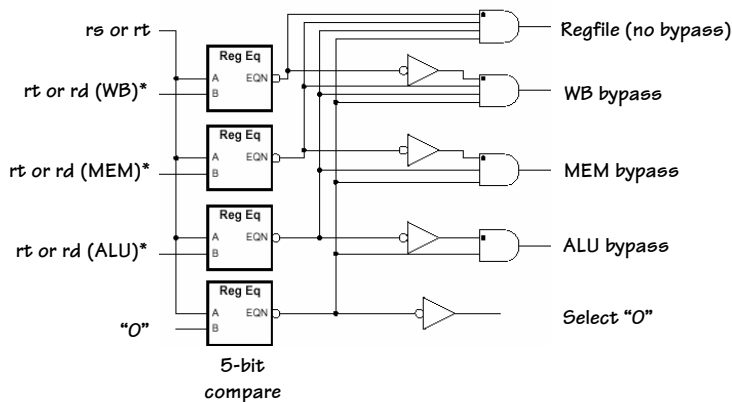
## Bypass MUX Details

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.

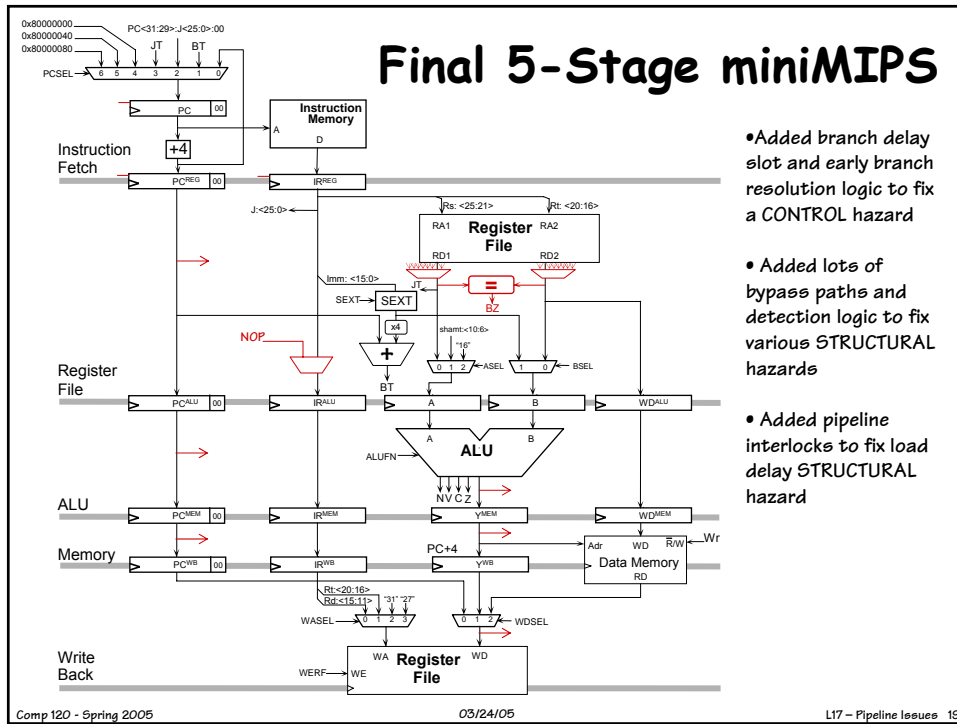


## Bypass Logic

miniMIPS bypass logic (need two copies for A/B data):



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



## Pipeline Summary (I)

- Started with unpipelined implementation
  - direct execute, 1 cycle/instruction
  - it had a long cycle time: mem + regs + alu + mem + wb
- We ended up with a 5-stage pipelined implementation
  - increase throughput (3x???)
  - delayed branch decision (1 cycle)
    - Choose to execute instruction after branch*
  - delayed register writeback (3 cycles)
    - Add bypass paths (6 x 2 = 12) to forward correct value*
  - memory data available only in WB stage
    - Introduce NOPs at IR<sup>ALU</sup>, to stall IF and RF stages until LD result was ready*

## Pipeline Summary (II)

**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.

PowerPC4: 7; PowerPC5: 23

Pentium: 5; Pentium/MMX:6; PentiumIII: 10; Pentium4: 20, 31

## RISC = Simplicity???

“The P.T. Barnum World’s Tallest Dwarf Competition”

World’s Most Complex RISC?

