DEPARTMENT OF INFORMATION SYSTEMS AND COMPUTER SCIENCE

Fixing the Pipelined Beta

While Still Making it Backwards-Compatible

Pipelining the Beta Needs Some More Work

- Program must be edited or smart-compiled for each target platform!
  - Need to distinguish between unpipelined and pipelined code:
  - Correct unpipelined code may not be correct if run as-is on a pipelined machine.
  - Correct pipelined code may not be correct if run as-is on an unpipelined machine.

original / correct unpipelined code

```
DIVC(r2,2,r4)
ADD(r1, r2, r3)
CMPLEC(r3, 5, r0)
MUL(r4, r2, r3)
SUB(r1, r2, r4)
```

$\Rightarrow$

correct pipelined code

```
DIVC(r2,2,r4)
ADD(r1, r2, r3)
NOP()
NOP()
CMPLEC(r3, 5, r0)
MUL(r4, r2, r3)
SUB(r1, r2, r4)
```

$\Rightarrow$

```
DIVC(r2,2,r4)
ADD(r1, r2, r3)
SUB(r1, r2, r4)
MUL(r4, r2, r3)
CMPLEC(r3, 5, r0)
```

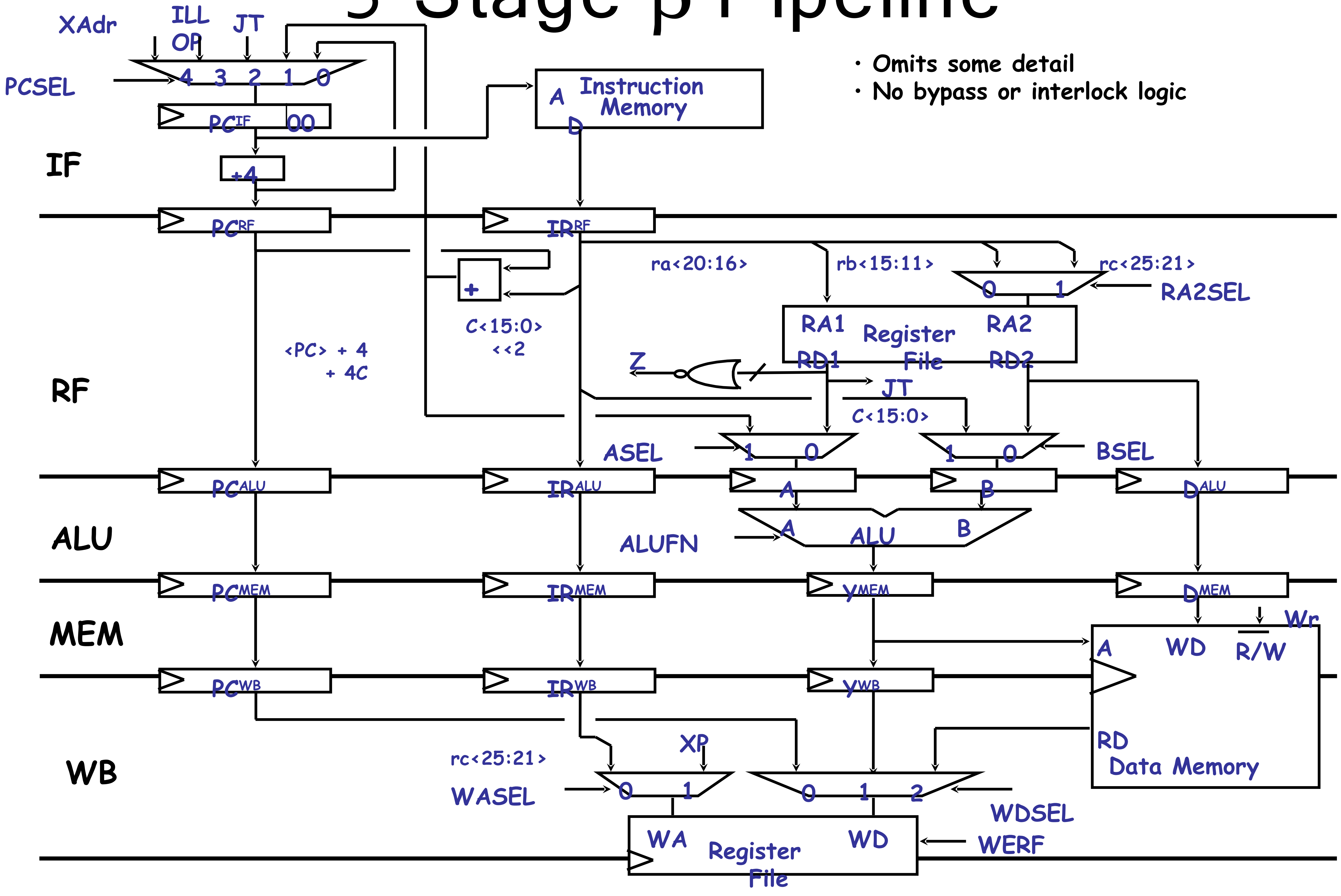
DISCS

Reading Memory: Load Timing

- **Fact**: Processors are (and will continue to be) faster relative to memories!
  - Do we just lengthen the cycle time?
  - Alternative: Longer pipelines
- Longer pipelines by:
  - Add “Memory Wait” stages between start of read operation and return of data.
  - Build pipelined memories, so that multiple memory transactions can be in progress at once.
    - Similar to the split-stage involving 2 different dryers in the laundromat. (Insert explanation/review here!)
- Memory read access time for:
  - 4-Stage pipeline: 1 clock
  - 5-Stage pipeline: 2 clocks

5–Stage β Pipeline

- Omits some detail
- No bypass or interlock logic



Delay Slots: “Rules”

- For Write-Back Delays (Data Dependencies)
  - <u>For 4-stage pipeline: 2 write-back delay slot(s)</u>
  - In general: Put reading instruction in RF stage in the same cycle that correct values of regs become available, then fill resulting delay slots with NOPs or non-dependent instructions.
    - # delay slots = # of lines between the RF stage (or where the registers are read) and the “ghost” stage after WB (or where updated register value/s is/are finally available).
- For Branch Delays
  - <u>For 4-stage pipeline: 1 branch delay slot(s)</u> (for standard BEQ, BNE, and JMP only)
  - In general: Add delay slots (filled with NOPs or non-dependent instructions) up to and including cycle where correct *target PC* is decided.
    - # of delay slots = # of lines before RF stage (or where the destination can be decided).
  - Remember to take note of write-back delay for return address in branches!

DISCS

Problems w/ Software Solutions

- We now need special compilers unless you want to continue editing the code manually.
- Software becomes hardware-specific.
  - A version will run for a particular pipeline.
  - What if our target system adds more stages?
  - What if our client is cheap and can’t pipeline?
    - Need to compile different versions!
    - No backward/upward software compatibility!
- Software solutions have performance limitations.
  - It is not always possible to reorder code to remove NOPs.
    - Is there a better way?
    - What else can be changed?
    - What do you think got us into this mess in the first place?

Hardware Solutions

- Stalling
  - Make instructions *wait* in RF stage until correct values become available.
- Annulment
  - On a *successful* branch, “cancel” the pre-fetched instruction.
  - Otherwise, just let it run since it should run anyway.
- Bypass paths
  - Get the desired data from some other stage of the pipeline without waiting for WB.

Hardware Solution 1: Stalling

Detect problem and "stall" the pipeline!

- Freeze IF, RF stages for 2 cycles and insert NOPs into IR^{ALU} for 2 cycles.

ADD(r1, r2, r3)
CMPLEC(r3, 5, r0)
MUL(r1,r2,r4)

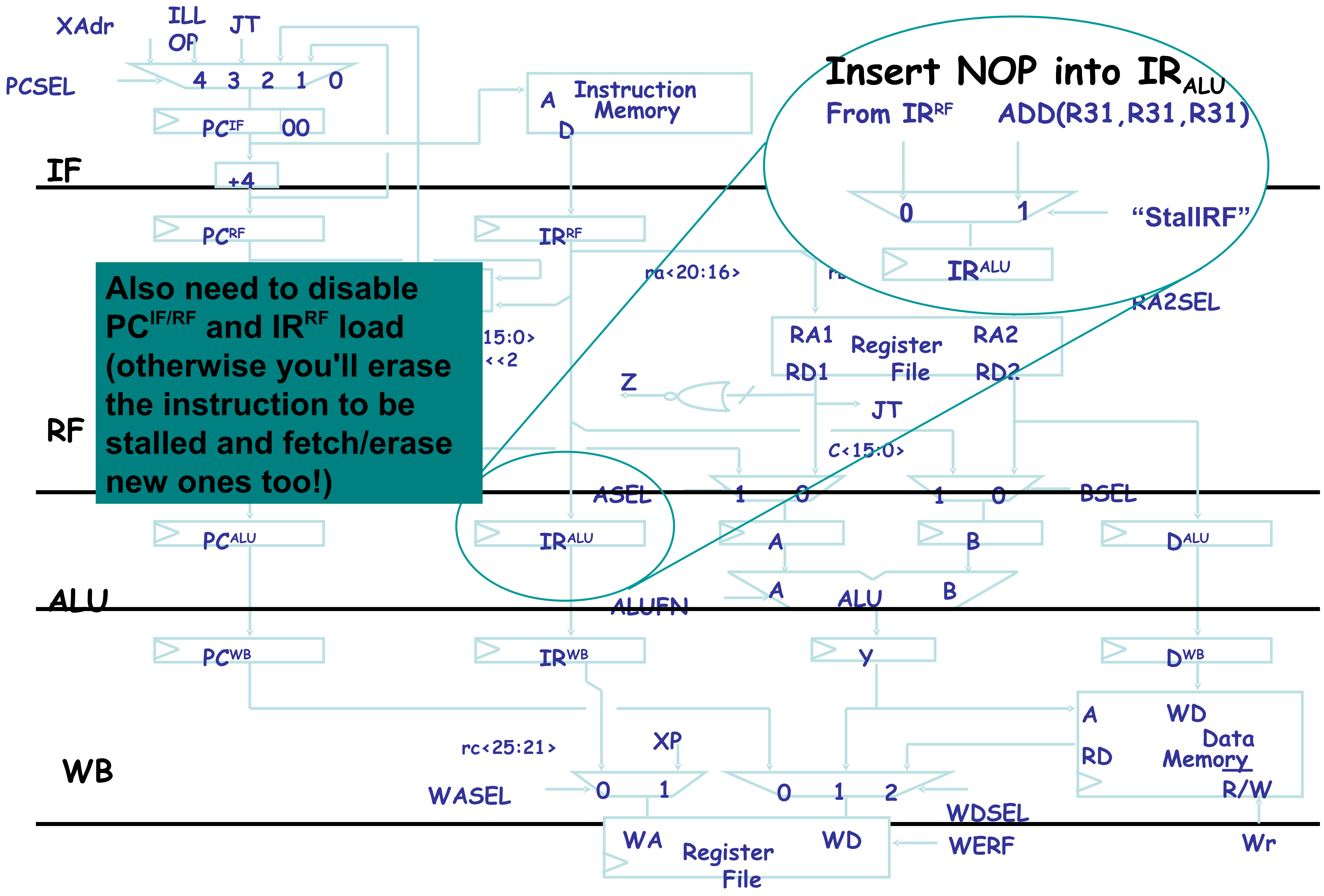
CMP stalled on r3 r3 read

	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_1	NOP_2	CMP	MUL
WB				ADD	NOP_1	NOP_2	CMP
R3			"stale"	"stale"	correct		

ADD writes r3

Note: Code stays the same!

4-Stage β Pipeline w/Stalling



Also need to disable PC^IF/RF and IR^RF load (otherwise you'll erase the instruction to be stalled and fetch/erase new ones too!)

Stalling: Pros and Cons

- Performance is still limited.
  - Running time is the same as if we added NOPs.
  - Note: we can still save cycles by reordering code like before.
- But we get better software compatibility.
  - Unpipelined code will run without modifications.
  - Don’t need write-back delay slots anymore.
  - Note: still can’t handle branches.
- Stall is used when other techniques don’t work.
  - e.g., bypassing loads / LDs (shown later)

DISCS

Stalling: Control Logic

- StallRF is determined by looking at contents of IR_{RF}, IR_{ALU}, and IR_{WB} instruction words.
  - Check for dependencies between ra, rb, and rc.
- Example: some conditions that make StallRF 1
  - ((IR_{RF}.ra == IR_{ALU}.rc) || (IR_{RF}.ra == IR_{WB}.rc))
    - Instruction in RF is reading ra from data being written by instruction in ALU or WB.
  - ((IR_{RF}.opcode != OPC) &&
    (IR_{RF}.rb == IR_{ALU}.rc) || (IR_{RF}.rb == IR_{WB}.rc)
    - Similar check for rb, but only if instruction in RF needs it since rb is meaningless if 2nd arg is a constant.
  - Note: no stall if register involved is R31 since its value is always zero.

DISCS

H/W Solution # 2: Branch Delay Slot Annulment

- **Problem**: One (or more) instructions after branch are fetched and run.
- **Software solutions:** Insert NOPs, Reorder code
- **Hardware solution:** "Annul" the XOR

	i	i + 1	i + 2	i + 3	i + 4	i + 5	i + 6
IF	CMP	ADD	SUB	BNE	XOR	CMP	
RF		CMP	ADD	SUB	BNE	XOR	CMP
ALU			CMP	ADD	SUB	BNE	XOR
WB				CMP	ADD	SUB	BNE
PC_{IF}	0x100	0x104	0x108	0x10C	0x110	0x100	
Z					0		
PCSEL					1		

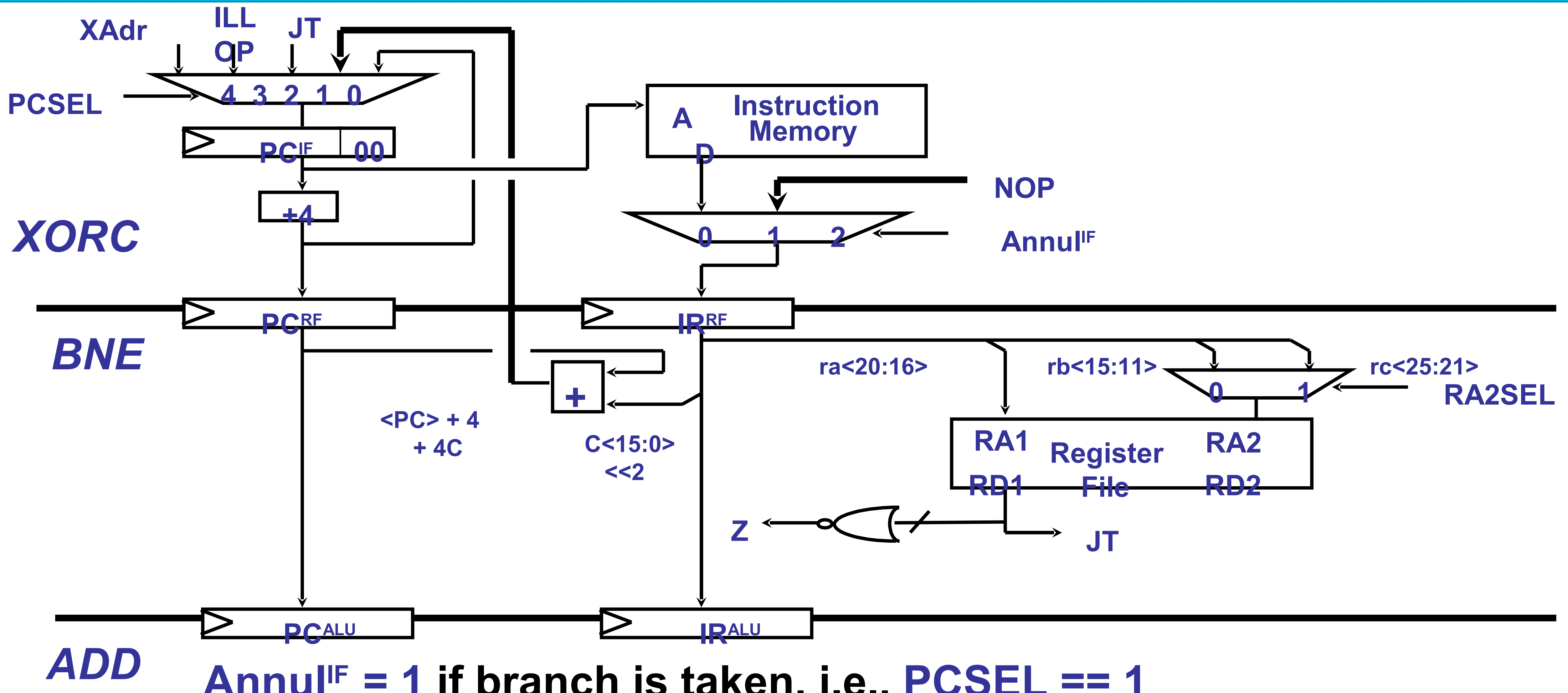
Annulment in Action

```
LOOP:  CMPLEC(r3, 100, r0)
       ADD(r1, r2, r3)
       SUB(r1, r2, r4)
       BNE(r0, LOOP)
        XOR(r31, r31, r3)
```

- Cancel XOR, then insert NOPs automatically.
- Code is the same as unpipelined code!

	i	i + 1	i + 2	i + 3	i + 4	i + 5	i + 6
IF	CMP	ADD	SUB	BNE	XOR	CMP	
RF		CMP	ADD	SUB	BNE	NOP	CMP
ALU			CMP	ADD	SUB	BNE	NOP
WB				CMP	ADD	SUB	BNE
PC_{IF}	0x100	0x104	0x108	0x10C	0x110	0x100	
Z					0		
PCSEL					1		

Annulment



Annul^{IF} = 1 if branch is taken, i.e., PCSEL == 1
or if a *JMP* instruction is used, i.e., PCSEL == 2.
Note: a 3-way mux is used here - Annul^{IF} = 2 to annul fault/trap operations (the ones that use the XP register) if so desired.
Alternatively, Annul^{IF} can just match PCSEL.

Annulment

- Slight performance improvement?
  - Performance is better than putting a NOP if the branch is *not taken* since XOR can proceed.
  - On the other hand, *we lose the chance to put useful instructions* in the delay slots through reordering.
- Software compatibility
  - No need to add NOPs after branches
- When used with stalling, the system can now run unpipelined code without any modifications.
- What else can we do to improve performance?
  - "branch prediction": Predicting whether the branch will be taken or not depending on past behavior of a branch instruction.

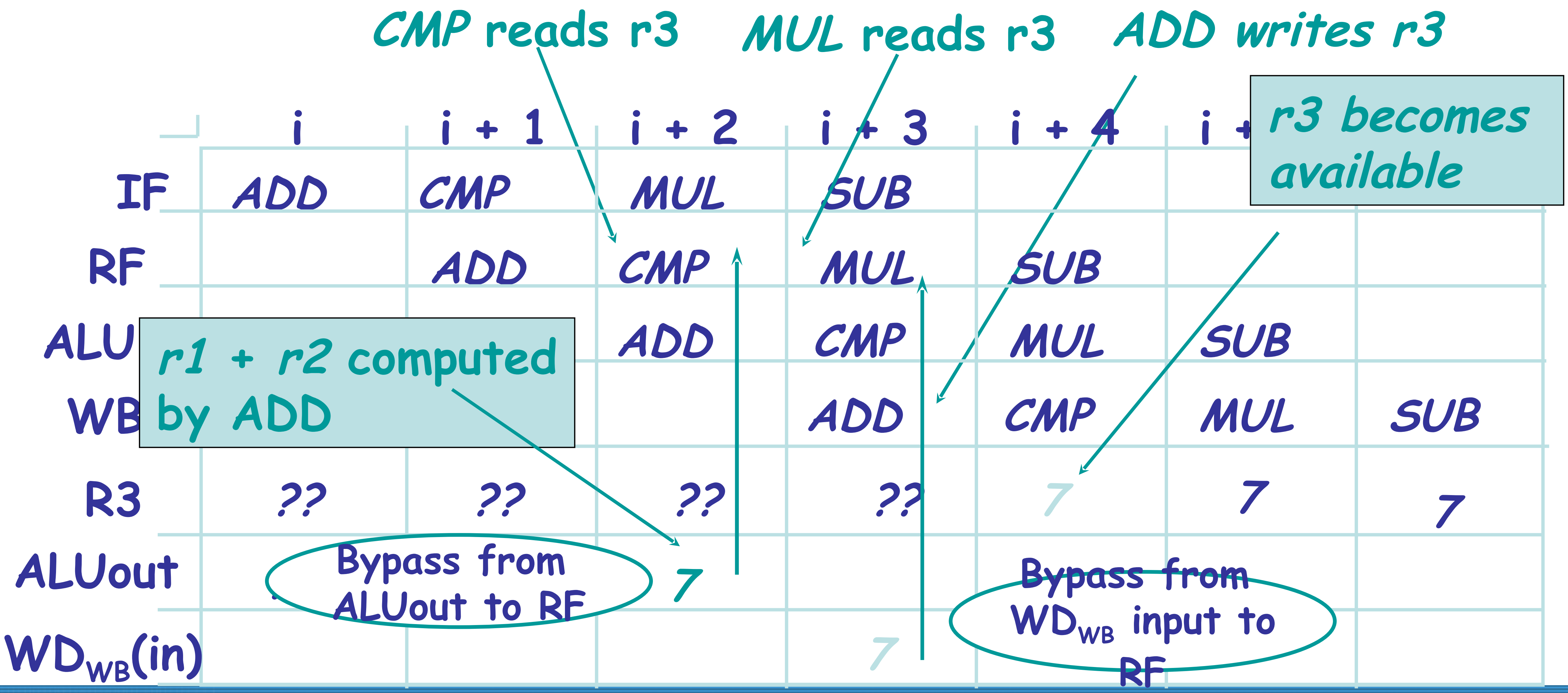
H/W Solution #3: Bypass Paths

- Write-back Delay Problem
  - Desired value is not written to Reg File until after WB stage.
  - However, even though value is not yet in Reg File, it is most likely already available somewhere else before it is actually written.
- Bypass Paths
  - Idea: Use the value directly, without passing through Reg file.
  - “Bypass” the Reg file.
    - aka “forwarding”

Bypass Paths Example

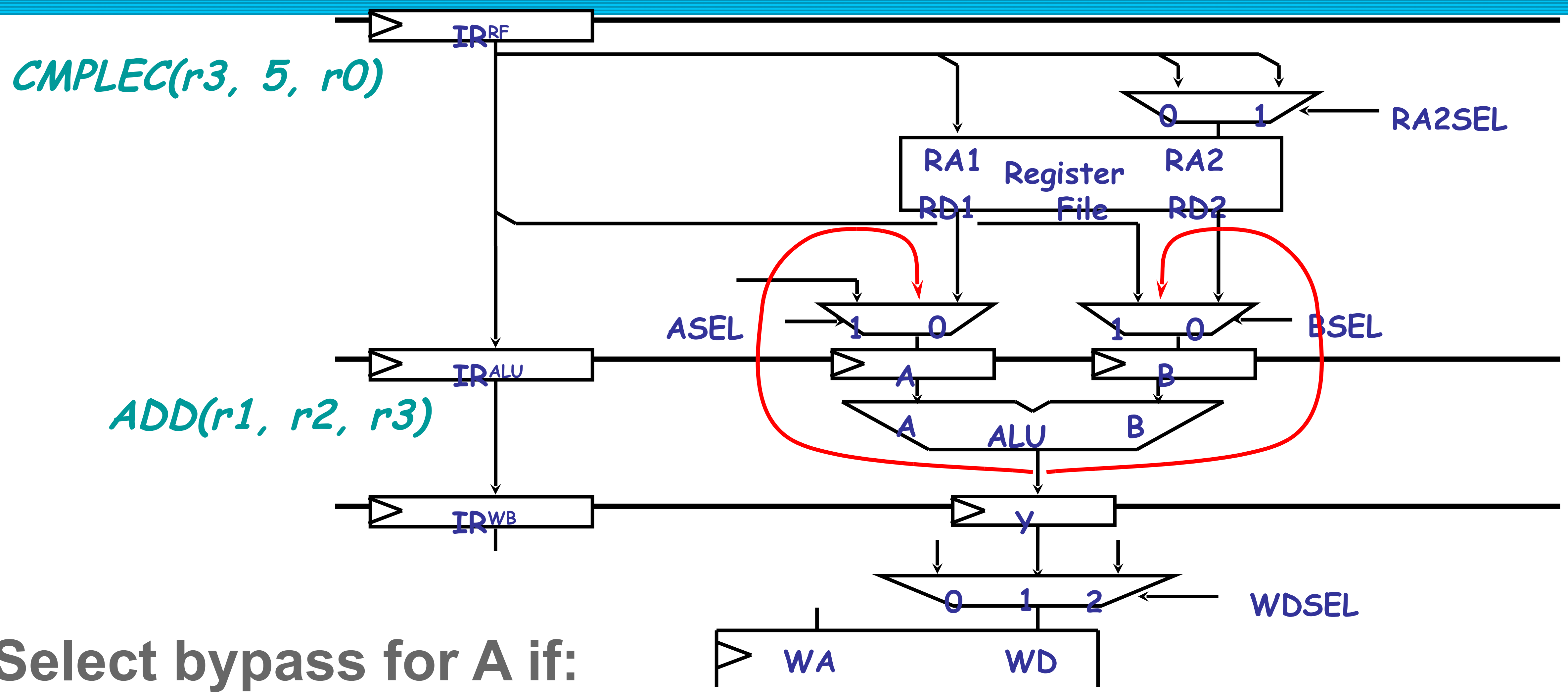
- Without bypassing, CMPLEC and MUL get stale value of R3.
- Bypassing allows us to get the new value directly from other stages!

| suppose R1=3, R2=4
ADD(r1, r2, r3)
CMPLEC(r3, 5, r0)
MUL(r3,r1,r0)



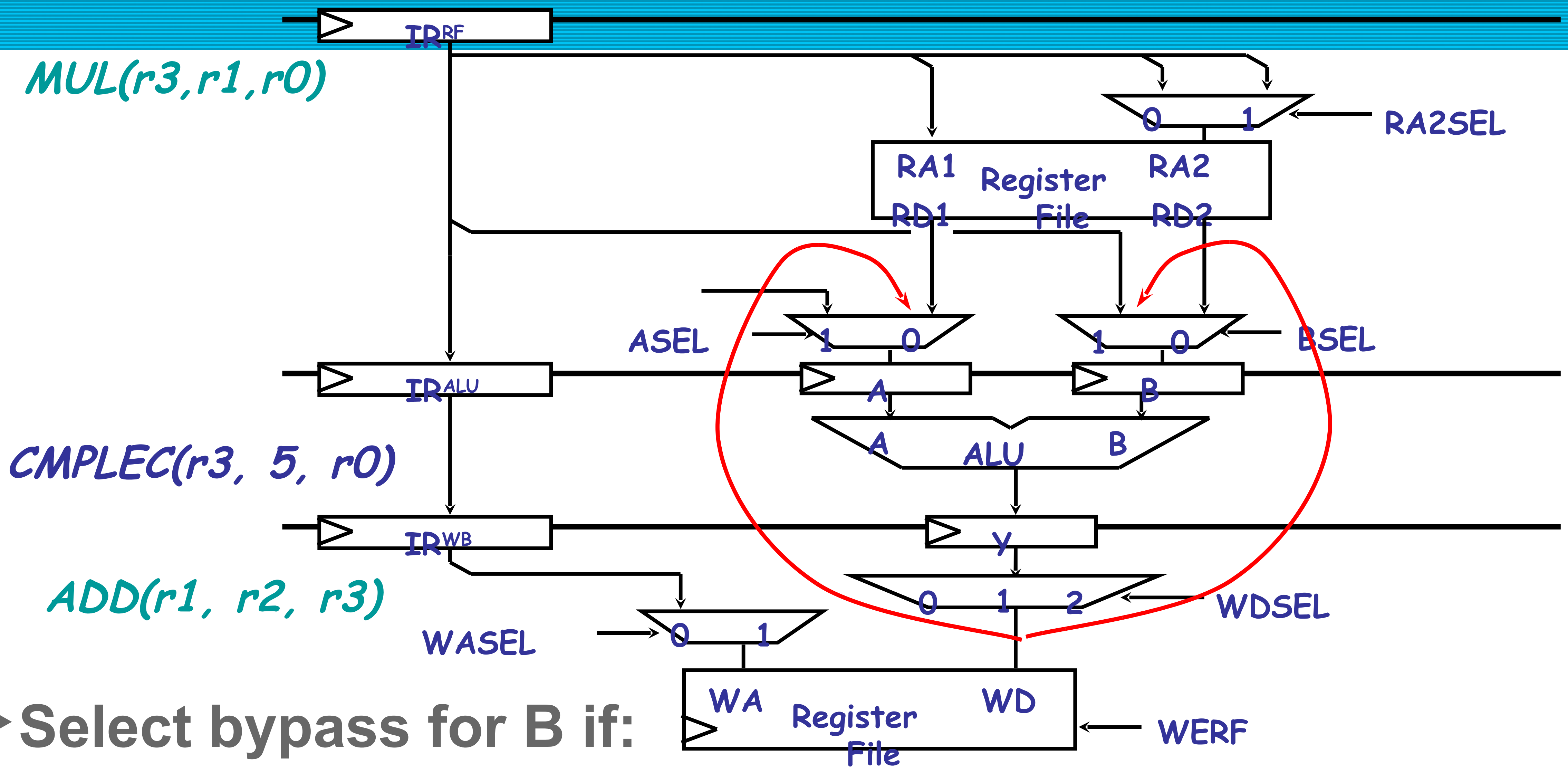
	i	i + 1	i + 2	i + 3	i + 4	i +	
IF	ADD	CMP	MUL	SUB			
RF		ADD	CMP	MUL	SUB		
ALU			ADD	CMP	MUL	SUB	
WB				ADD	CMP	MUL	SUB
R3	??	??	??	??	7	7	7
ALUout			7				
WD_{WB}(in)				7			

Bypass Paths (ALU-RF)



- Select bypass for A if:
  - OPCODE^{RF} == OP, OPC, ... &&
  - OPCODE^{ALU} == OP, OPC, ... &&
  - ra^{RF} == rc^{ALU}
- (Similar path for BSEL mux input if applicable)

Bypass Paths (WB-RF)



- Select bypass for B if:
  - OPCODE^{RF} == OP &&
  - WERF == 1 &&
  - rb^{RF} == WA
- (Similar path for ASEL mux input if applicable)

Loads

- Consider the sequence:

  LD(r1, 0, r4)
  ADD(r1, r4, r5)
  XOR(r3, r4, r6)

Will our previous bypass paths fix <u>both</u> *r4* read problems?

r4 read 1

r4 read 2

	i	i + 1	i + 2	i + 3	i + 4	i + 5	i + 6
IF	LD	ADD	XOR				
RF		LD	ADD	XOR			
ALU			LD	ADD	XOR		
WB				LD	ADD	XOR	

r4 written

Loads

- Consider the sequence:

  LD(r1, 0, r4)
  ADD(r1, r4, r5)
  XOR(r3, r4, r6)

Bypass will not help ADD, since new R4 not available *anywhere* yet! We have no choice but to stall.

r4 read 1

r4 read 2

r4 written

	i	i + 1	i + 2	i + 3	i + 4	i + 5	i + 6
IF	LD	ADD	XOR				
RF		LD	ADD	XOR			
ALU			LD	ADD	XOR		
WB				LD	ADD	XOR	
R4	??	??	??	??	23	23	23
ALUout			??				
WD_{WB}(in)				23			

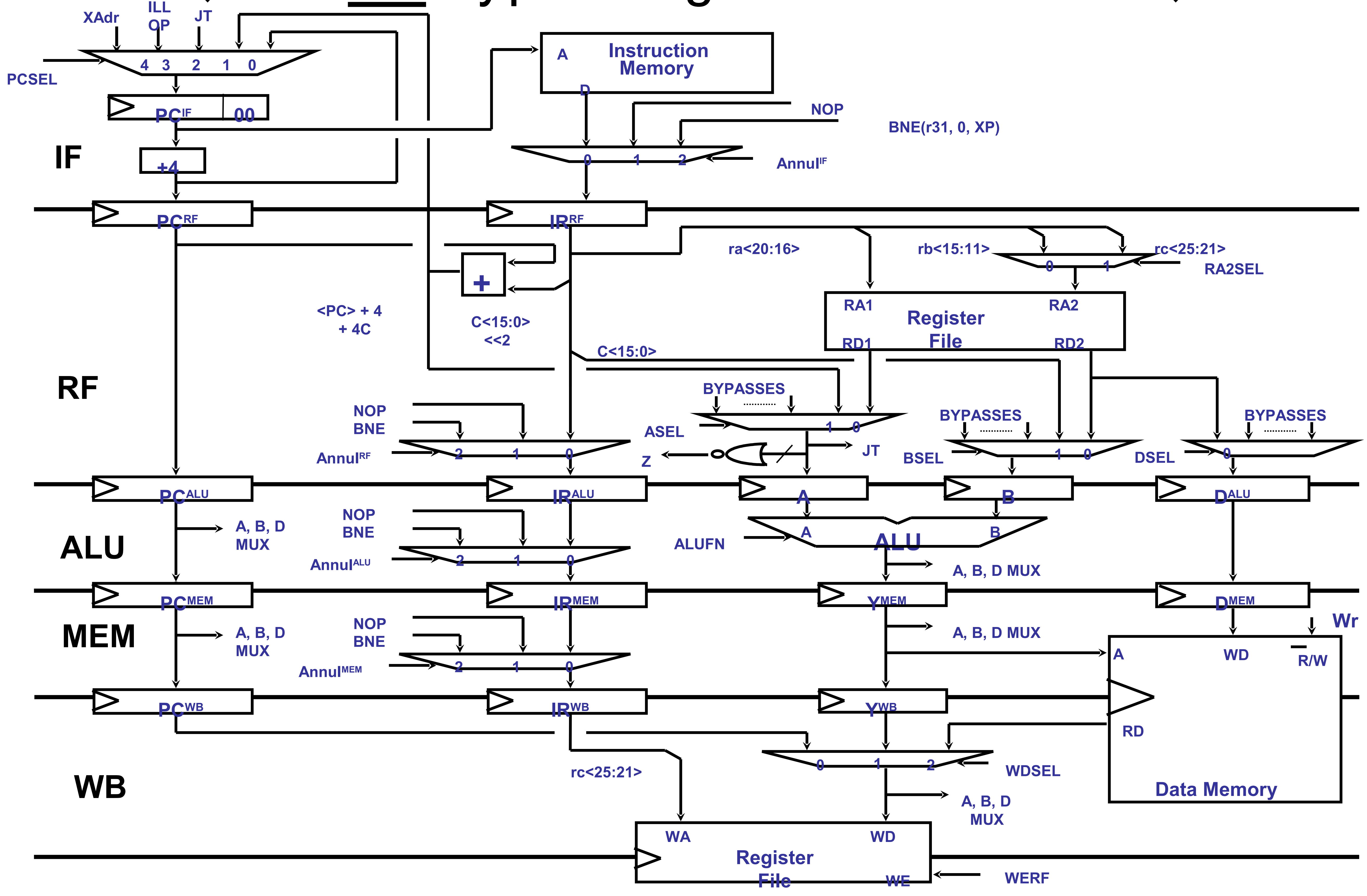
Bypass from ALUout to RF

Bypass from WD_{WB} input to RF

Bypass Paths: Summary

- Idea: *We can't get the data from the RF yet, but we can get it somewhere else.*
  - Also known as "data forwarding"
- But this will only work if data is available somewhere.
- Otherwise, we still need to stall or insert NOPs or other instructions.
  - The earlier LD problem should stall or have delay slots.
  - Full bypassing will include a bypass path from data memory.
- Performance is improved.
  - In the 4- and 5- stage pipeline with standard Beta instructions, we can now run consecutive ALU instructions even if they have dependencies.
  - Closer to 1.0 CPI.
    - Stalling and annulment actually increase average CPI if you look at them closely.

5-Stage β Pipeline (with "full bypassing" and annulment)



Exercise

- Consider the sequence:

```
loop: LD(r1, 0, r4)
      ADD(r1, r4, r5)
      XOR(r3, r4, r6)
      BEQ(r5,loop,r31)
      MULC(r7,2,r8)
```

Insert NOPs, or rearrange code as necessary to make it run correctly on a 4-stage Beta pipeline assuming:

1) No stalling, No bypassing, No annulment
2) No stalling, Full bypassing, Annulment
3) Full stalling, Full bypassing, No Annulment

Optimize running time, but be sure it works correctly.