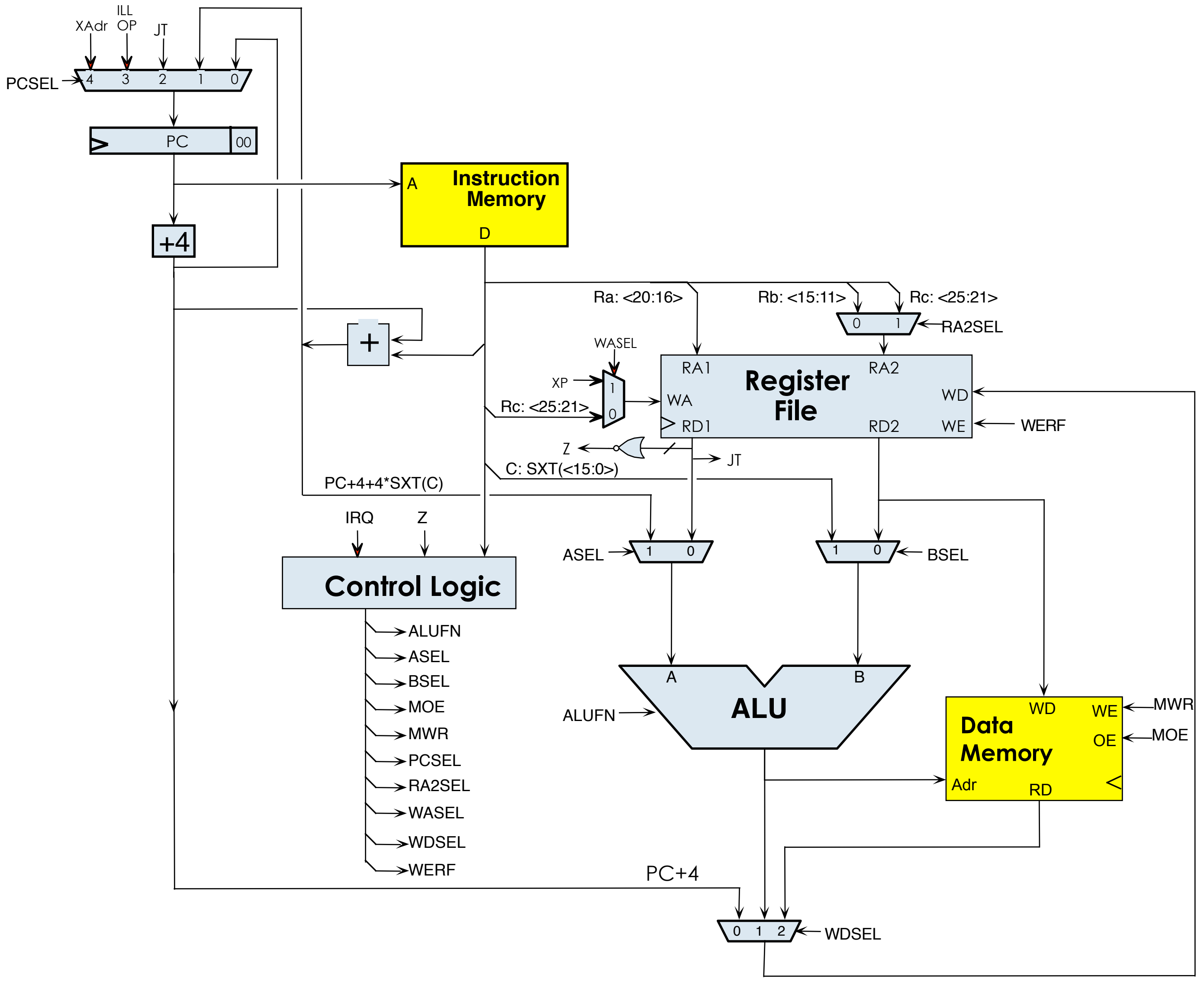
15. Pipelining the Beta

6.004x Computation Structures
Part 3 – Computer Organization

Reminder: Single-Cycle Beta



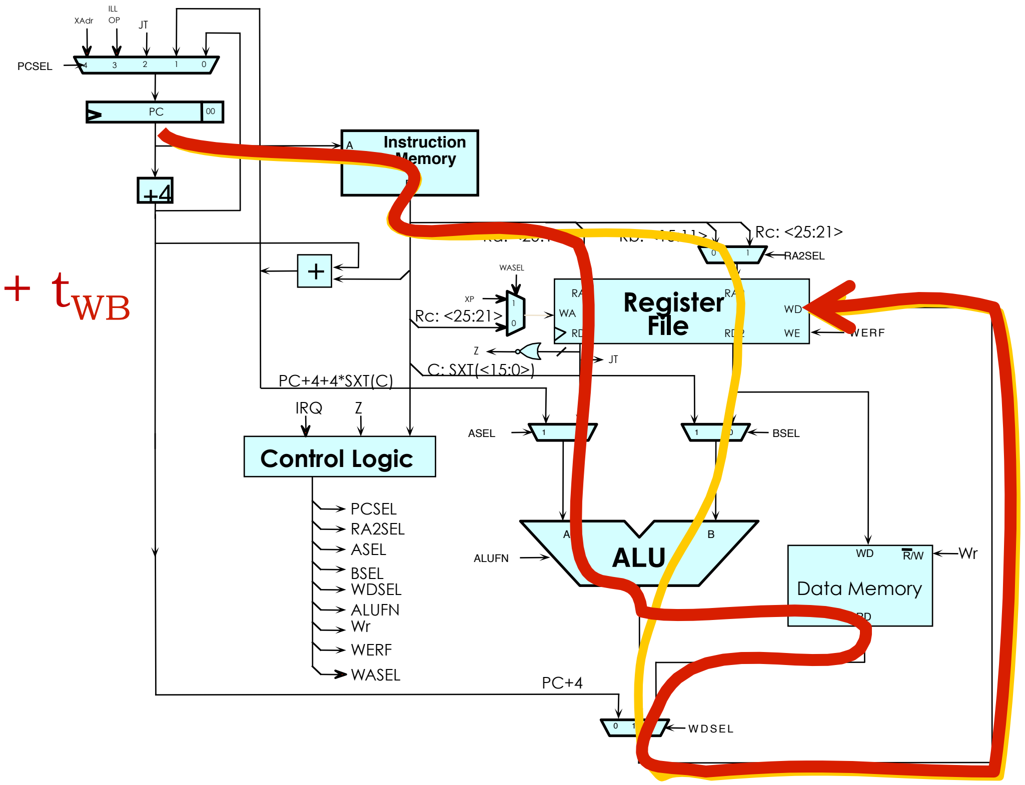
Single-Cycle Beta Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \underbrace{\frac{\text{Cycles}}{\text{Instruction}}}_{\text{CPI}} \cdot \underbrace{\frac{\text{Time}}{\text{Cycle}}}_{t_{CLK}}$$

- CPI = 1
- t_{CLK} = Longest path for any instruction

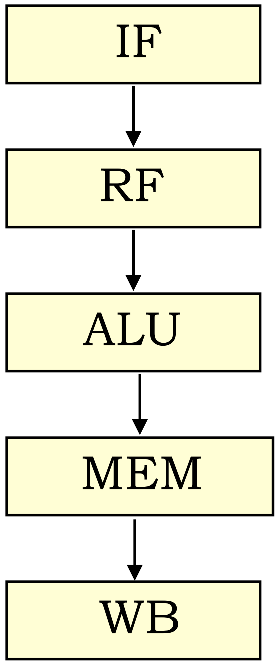
$$t_{CLK} \approx t_{IFETCH} + t_{RF} + t_{ALU} + t_{MEM} + t_{WB}$$

- – Slow
- – Inflexible: Instructions with smaller critical path cannot execute faster



Pipelined Implementation

- Divide datapath in multiple pipeline stages to reduce t_{CK}
  - Each instruction executes over multiple cycles
  - Consecutive instructions are overlapped to keep CPI ≈ 1.0
- We'll study the classic 5-stage pipeline:



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

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

ALU stage: Performs indicated operation in ALU, passes result to

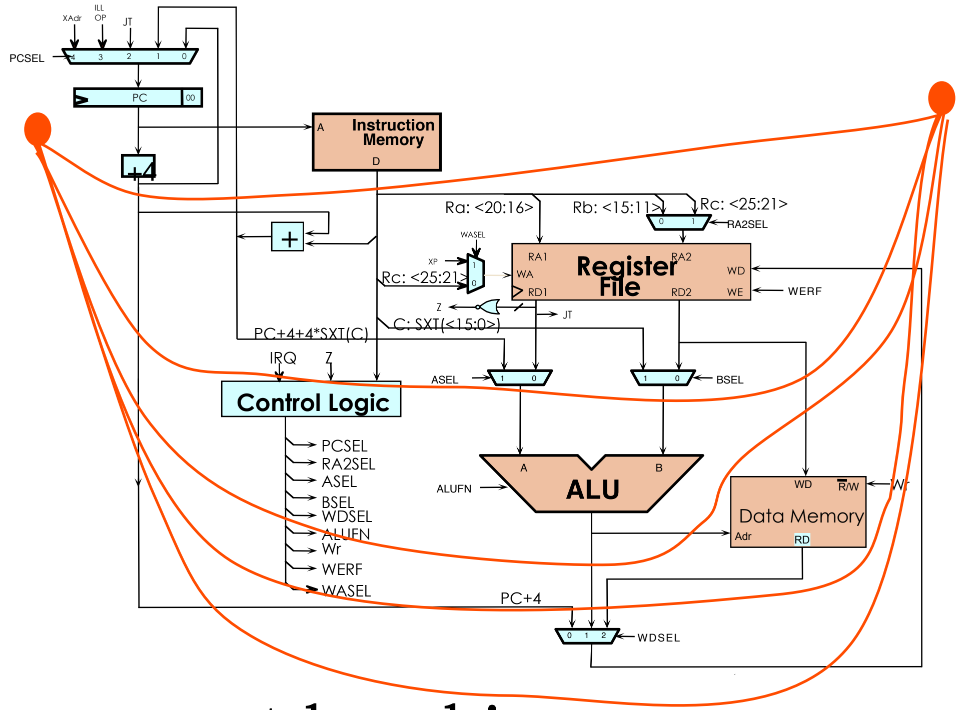
Memory stage: If it's a LD, use ALU result as an address, pass mem data (or ALU result if not LD) to

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

$t_{CLK} = \max\{t_{IFETCH}, t_{RF}, t_{ALU}, t_{MEM}, t_{WB}\}$

Why isn't this a 20-minute lecture?

We know how to pipeline combinational circuits, what's the big deal?



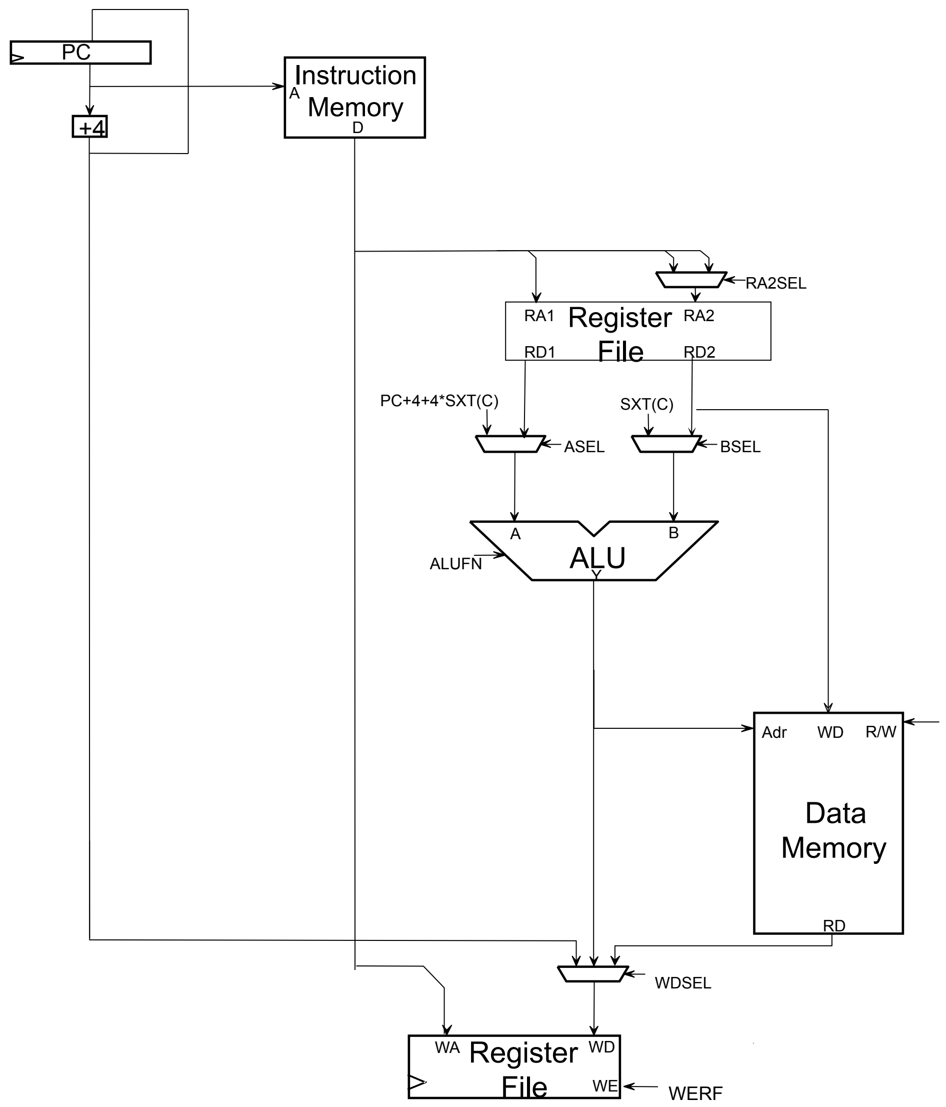
- Beta has state: PC, Register file, Memories
- There are dependencies we cannot break!
  - To compute the next PC
  - To write result into the register file
- We'll be addressing these issues as we examine the operation of our execution pipeline.

Pipeline Hazards

- Pipelining tries to overlap the execution of multiple instructions, but an instruction may depend on something produced by an earlier instruction
  - A data value → Data hazard
  - The program counter → Control hazard (branches, jumps, exceptions)

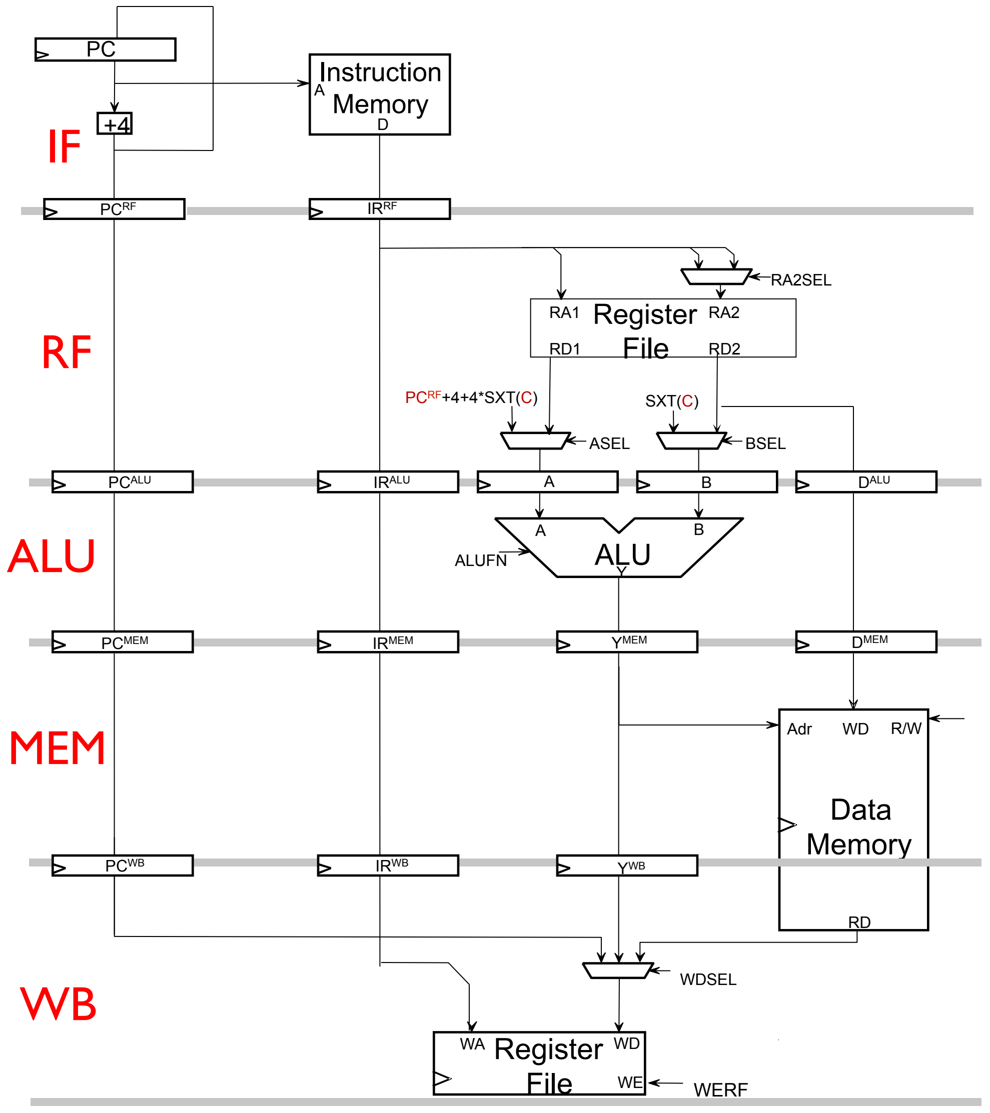
- Plan of attack:
  1. Design a 5-stage pipeline that works with sequences of independent instructions
  2. Handle data hazards
  3. Handle control hazards

Simplified Unpipelined Beta Datapath



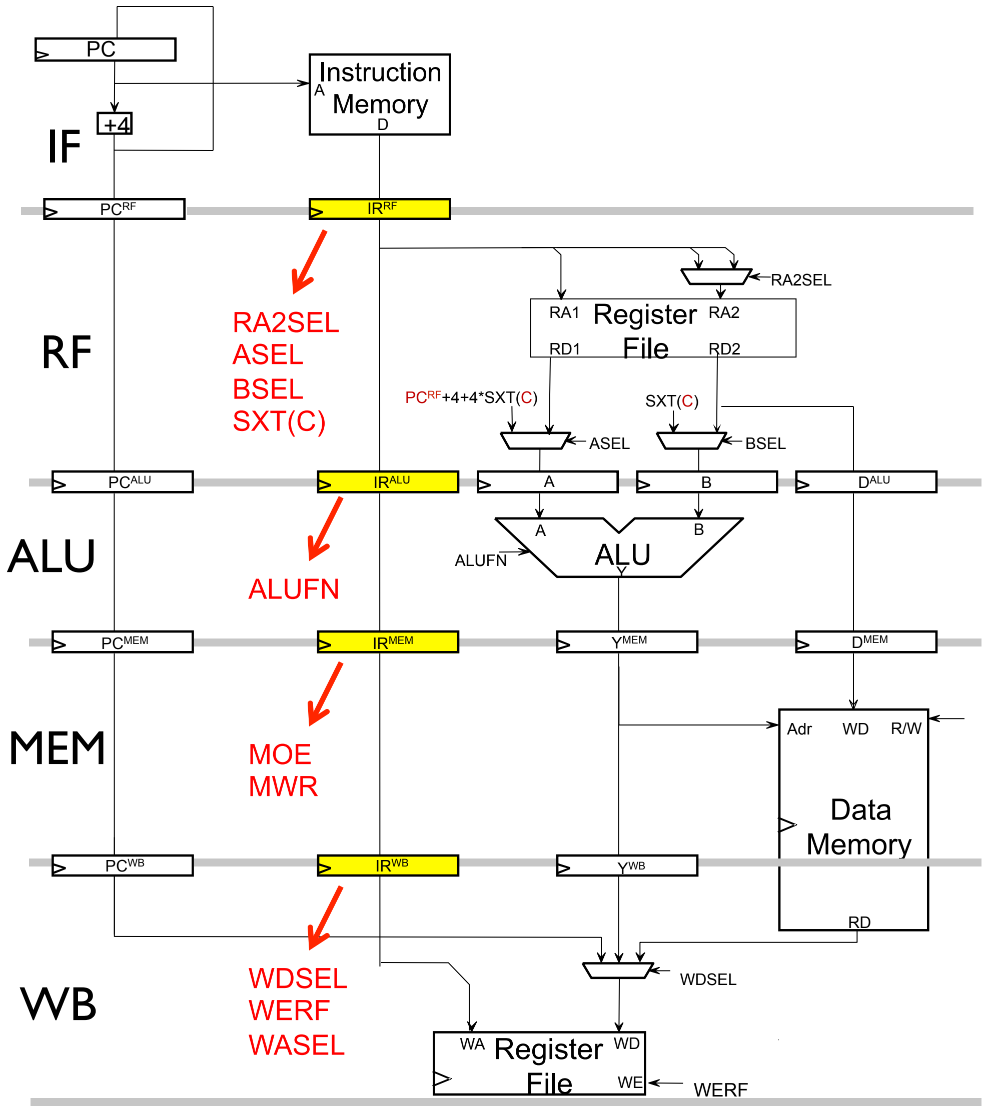
- NextPC = PC+4 (we'll worry about control hazards later)
- Same register file appears twice in the diagram
  - Top: reads
  - Bottom: writes

5-Stage Pipelined Datapath



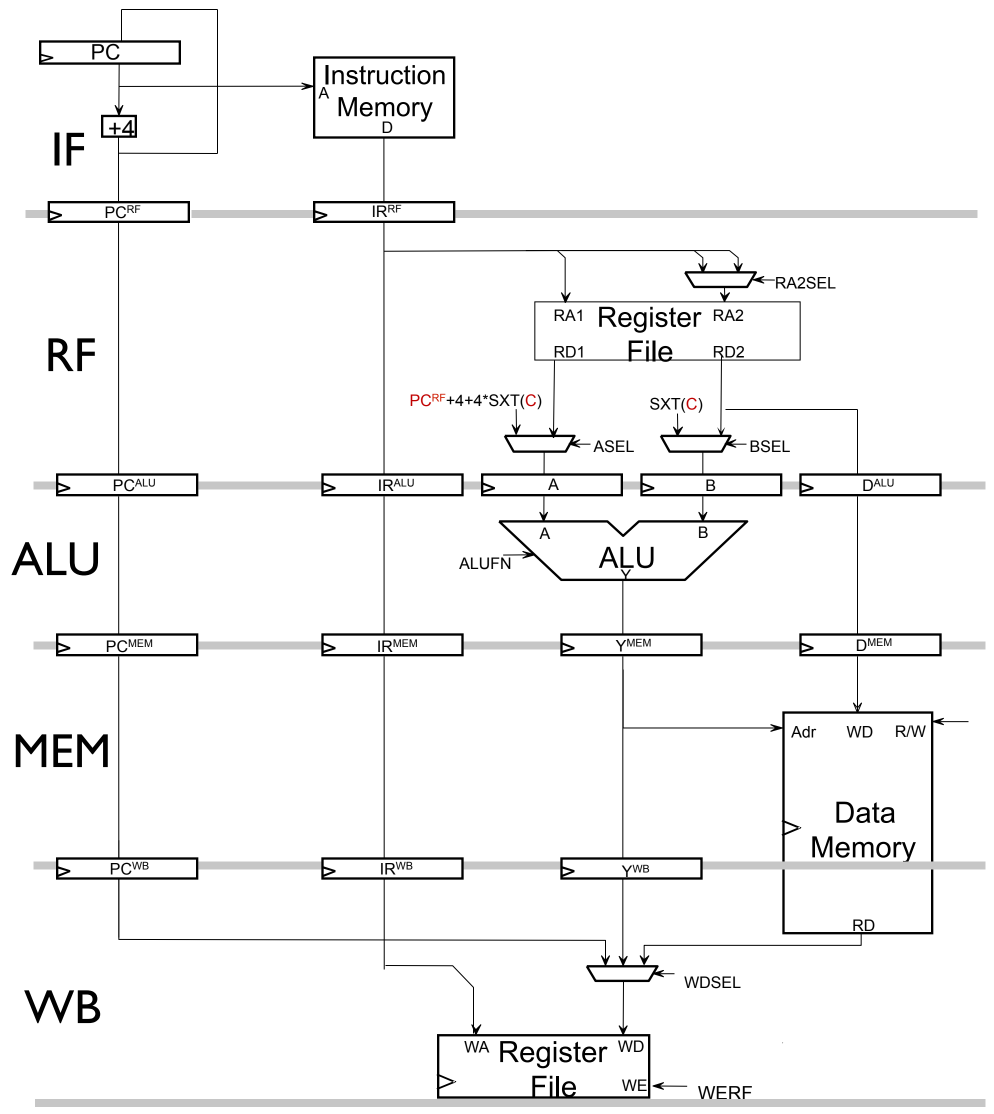
- Pipeline registers separate different stages:
  - IF – instruction fetch
  - RF – register file access
  - ALU – compute result
  - MEM – memory access
  - WB – write back to reg. file
- Each stage services one instruction per cycle
- Data memory reads are now pipelined, not combinational
  - Data read appears in RD the next cycle

Pipelined Control



- Instruction contents propagated through the pipeline in Instruction Registers (IR^{RF}, IR^{ALU}, …)
- Control signals for each stage generated from corresponding IR
  - e.g. ASEL uses IR^{RF} opcode, WERF uses IR^{WB}, etc
- Pipeline hazards will require new control signals

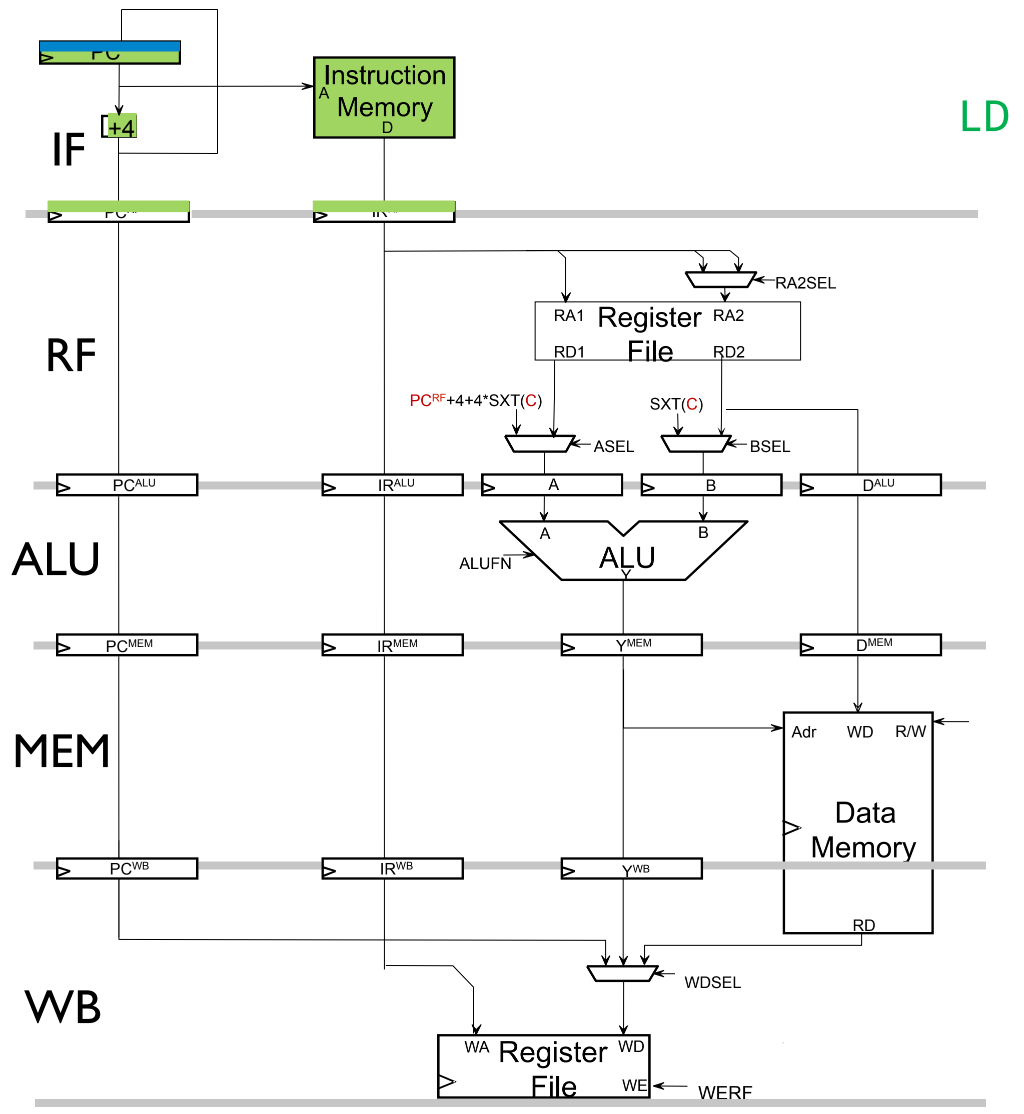
Pipelined Execution Example



```
LD(R1, 4, R2)
LD(R3, 8, R4)
SUB(R6, R7, R8)
XOR(R9, R10, R11)
MUL(R12, R13,R14)
ADD(R15, 1, R16)
```

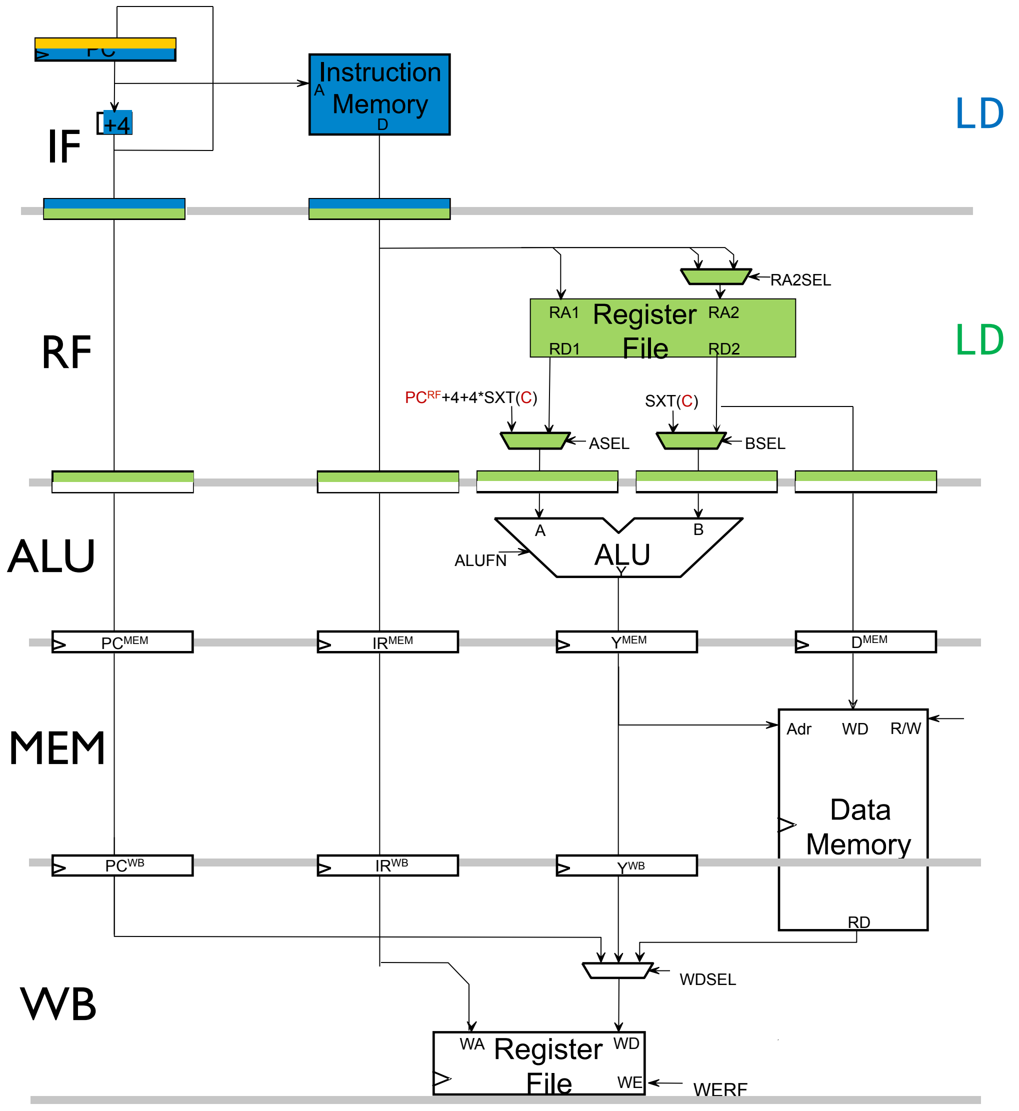
Sequence of instructions without data or control dependences

Example: Cycle 1



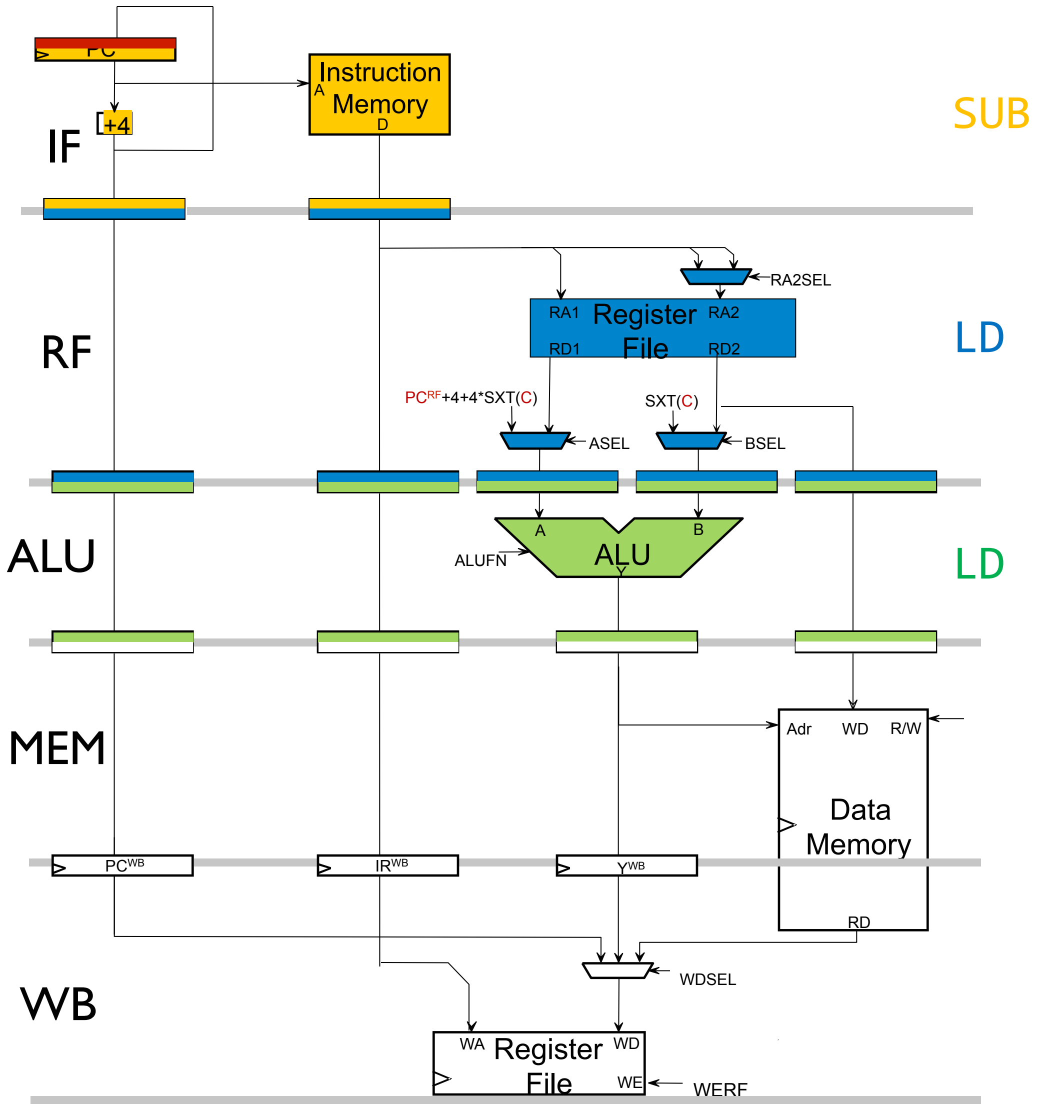
LD(R1, 4, R2)
LD(R3, 8, R4)
SUB(R6, R7, R8)
XOR(R9, R10, R11)
MUL(R12, R13,R14)
ADD(R15, 1, R16)

Example: Cycle 2



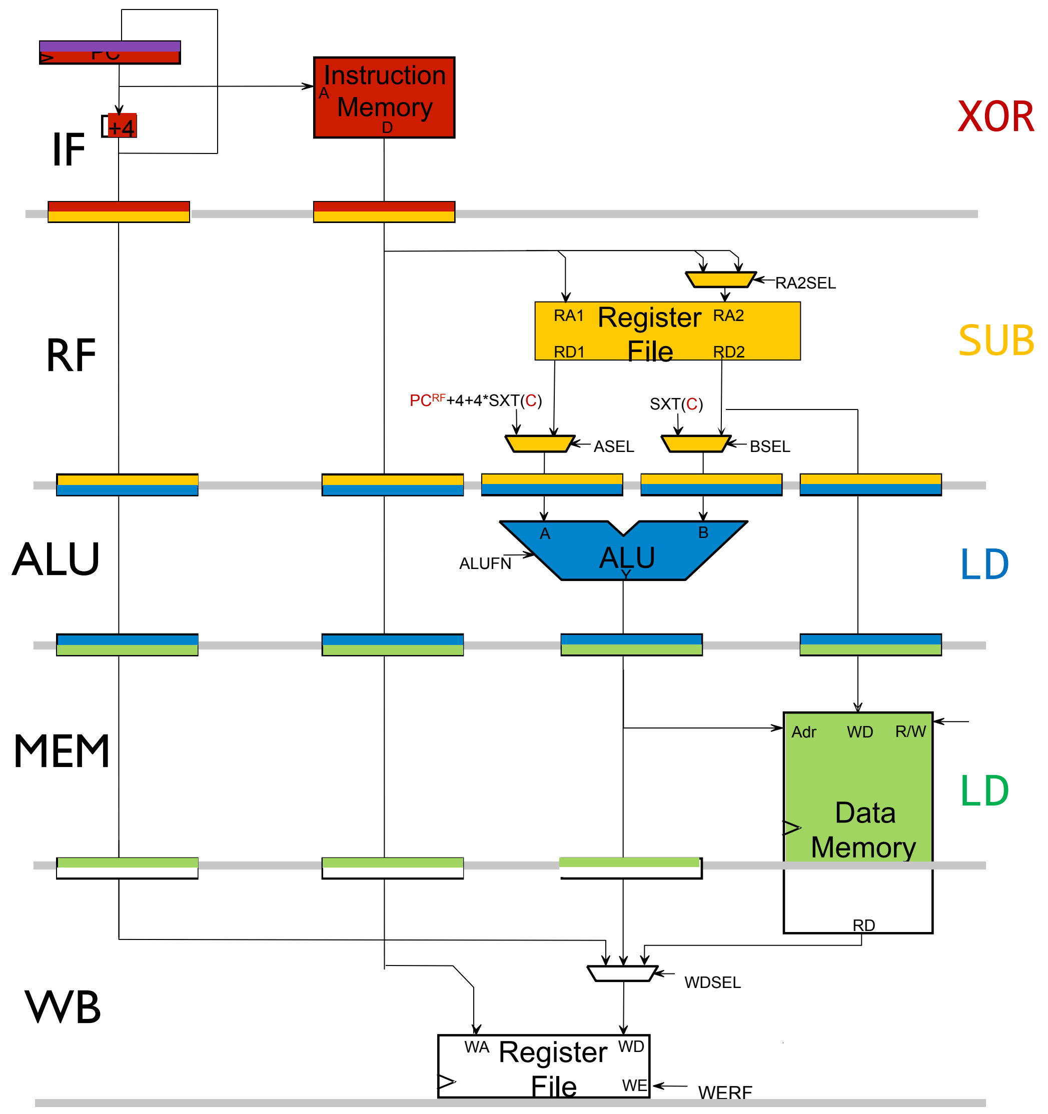
LD(R1, 4, R2)
LD(R3, 8, R4)
SUB(R6, R7, R8)
XOR(R9, R10, R11)
MUL(R12, R13,R14)
ADD(R15, 1, R16)

Example: Cycle 3



```
LD(R1, 4, R2)
LD(R3, 8, R4)
SUB(R6, R7, R8)
XOR(R9, R10, R11)
MUL(R12, R13,R14)
ADD(R15, 1, R16)
```

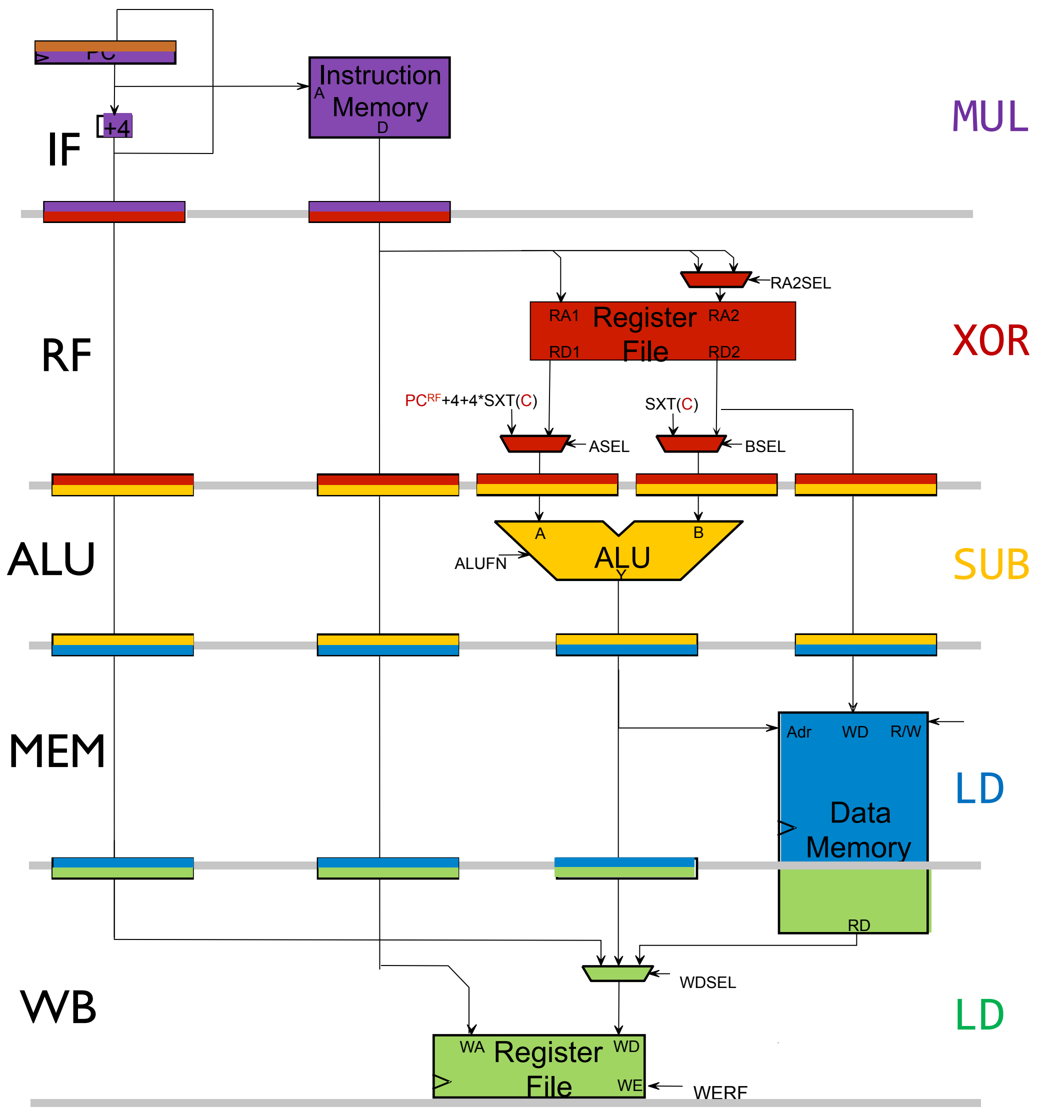
Example: Cycle 4



```
LD(R1, 4, R2)
LD(R3, 8, R4)
SUB(R6, R7, R8)
XOR(R9, R10, R11)
MUL(R12, R13,R14)
ADD(R15, 1, R16)
```

First LD starts data memory read
Data not yet available!

Example: Cycle 5



LD(R1, 4, R2)
LD(R3, 8, R4)
SUB(R6, R7, R8)
XOR(R9, R10, R11)
MUL(R12, R13,R14)
ADD(R15, 1, R16)

Second LD starts data memory read

Data for first LD available in RD

Pipeline Diagrams

- Represent pipeline utilization over time.
- When do reads and writes happen?

  Read REGFILE in RF stage
  Write REGFILE at end of WB stage

```
LD(R1, 4, R2)
LD(R3, 8, R4)
SUB(R6, R7, R8)
XOR(R9, R10, R11)
MUL(R12, R13,R14)
ADD(R15, 1, R16)
```

Cycles $\longrightarrow$

Stages

	1	2	3	4	5	6
IF	LD	LD	SUB	XOR	MUL	ADD
RF		LD	LD	SUB	XOR	MUL
ALU			LD	LD	SUB	XOR
MEM				LD	LD	SUB
WB					LD	LD

Read R1

Write R2

Data Hazards

- Consider this instruction sequence:

```
ADDC(R1, 1, R2)
SUBC(R2, 4, R3)
MUL(R6, R7, R8)
XOR(R9, R10, R11)
…
```

	1	2	3	4	5	6
IF	ADDC	SUBC	MUL	XOR		
RF		ADDC	SUBC	MUL	XOR	
ALU			ADDC	SUBC	MUL	XOR
MEM				ADDC	SUBC	MUL
WB					ADDC	SUBC

- SUBC reads R2 on cycle 3, but ADDC does not update it until end of cycle 5 → R2 is stale!
- Pipeline must maintain correct behavior…

Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
- Strategy 2: Bypass (aka Forward). Route data to the earlier pipeline stage as soon as it is calculated
- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value

Resolving Data Hazards (1)

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages

```
ADDC(R1, 1, R2)
SUBC(R2, 4, R3)
MUL(R6, R7, R8)
XOR(R9, R10, R11)
…
```

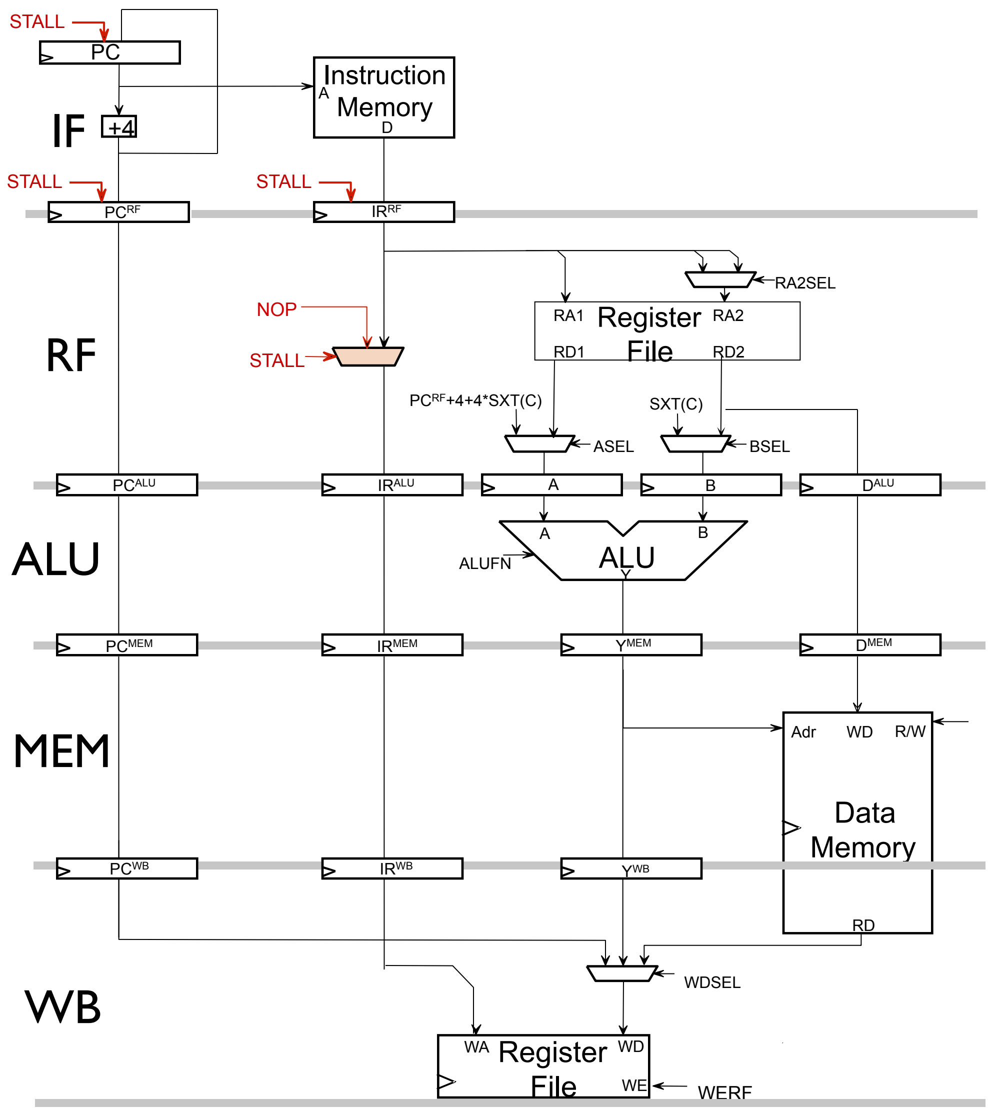
Stall

	1	2	3	4	5	6	7	8
IF	ADDC	SUBC	MUL	*MUL*	*MUL*	*MUL*	XOR	
RF		ADDC	SUBC	*SUBC*	*SUBC*	*SUBC*	MUL	XOR
ALU			ADDC	**NOP**	**NOP**	**NOP**	SUBC	MUL
MEM				ADDC	NOP	NOP	NOP	SUBC
WB					ADDC	NOP	NOP	NOP

R2 updated

Stalls increase CPI!

Stall Logic



- New STALL control signal
- STALL==1
  - Disables PC and RF pipeline registers
  - Injects NOP instruction into ALU stage
- NOP = No-operation, e.g., ADD(R31, R31, R31)

- Control logic sets STALL=1 if source registers of instruction in RF match destination register on ALU, MEM, or WB *(except when source is R31)*

Resolving Data Hazards (2)

- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated

```
ADDC(R1, 1, R2)
SUBC(R2, 4, R3)
MUL(R6, R7, R8)
XOR(R9, R10, R11)
…
```

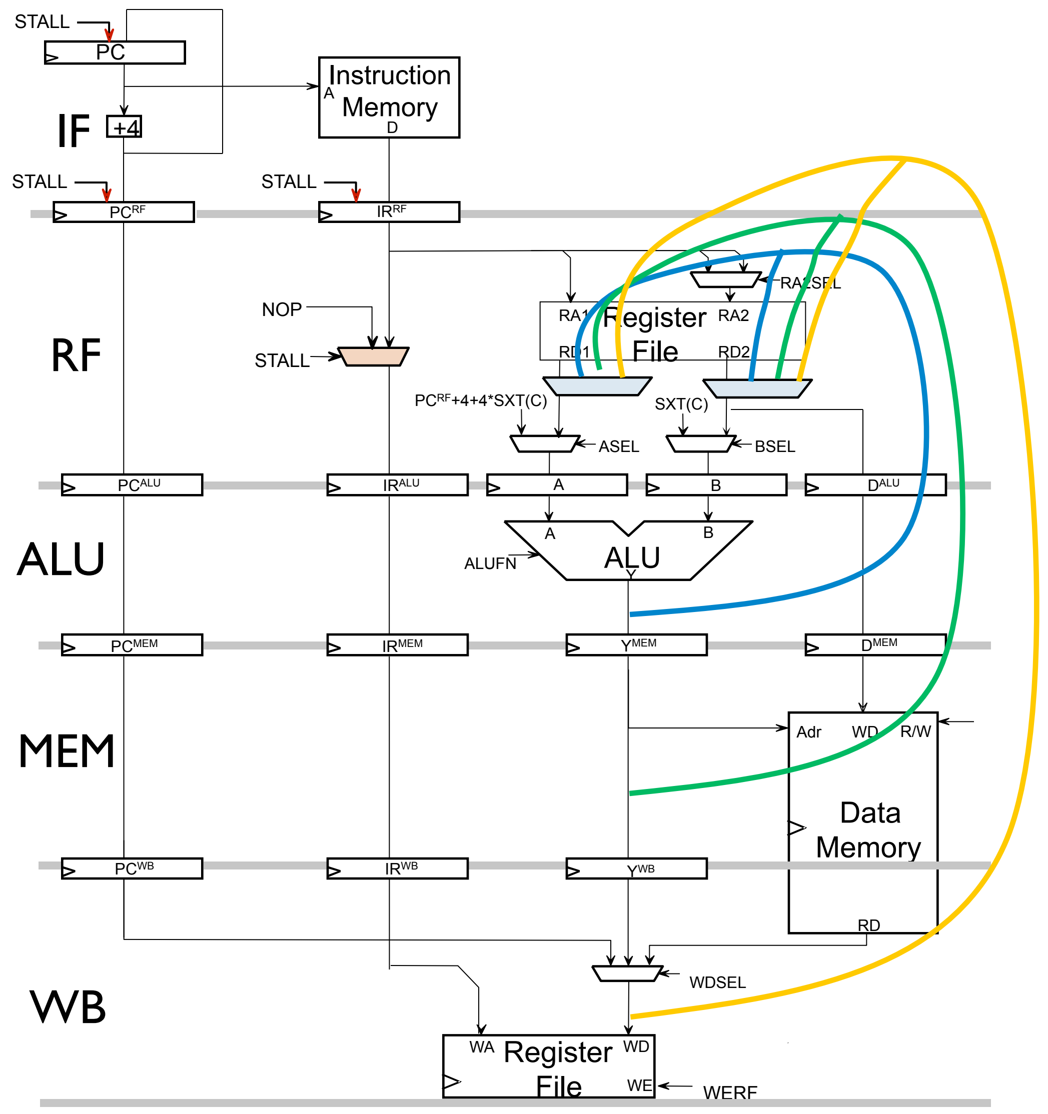
- ADDC writes to R2 at the end of cycle 5… but the result is available at the end of the ALU stage!

	1	2	3	4	5
IF	ADDC	SUBC	MUL	XOR	
RF		ADDC	SUBC	MUL	XOR
ALU			ADDC	SUBC	MUL
MEM				ADDC	SUBC
WB					ADDC

ADDC result computed

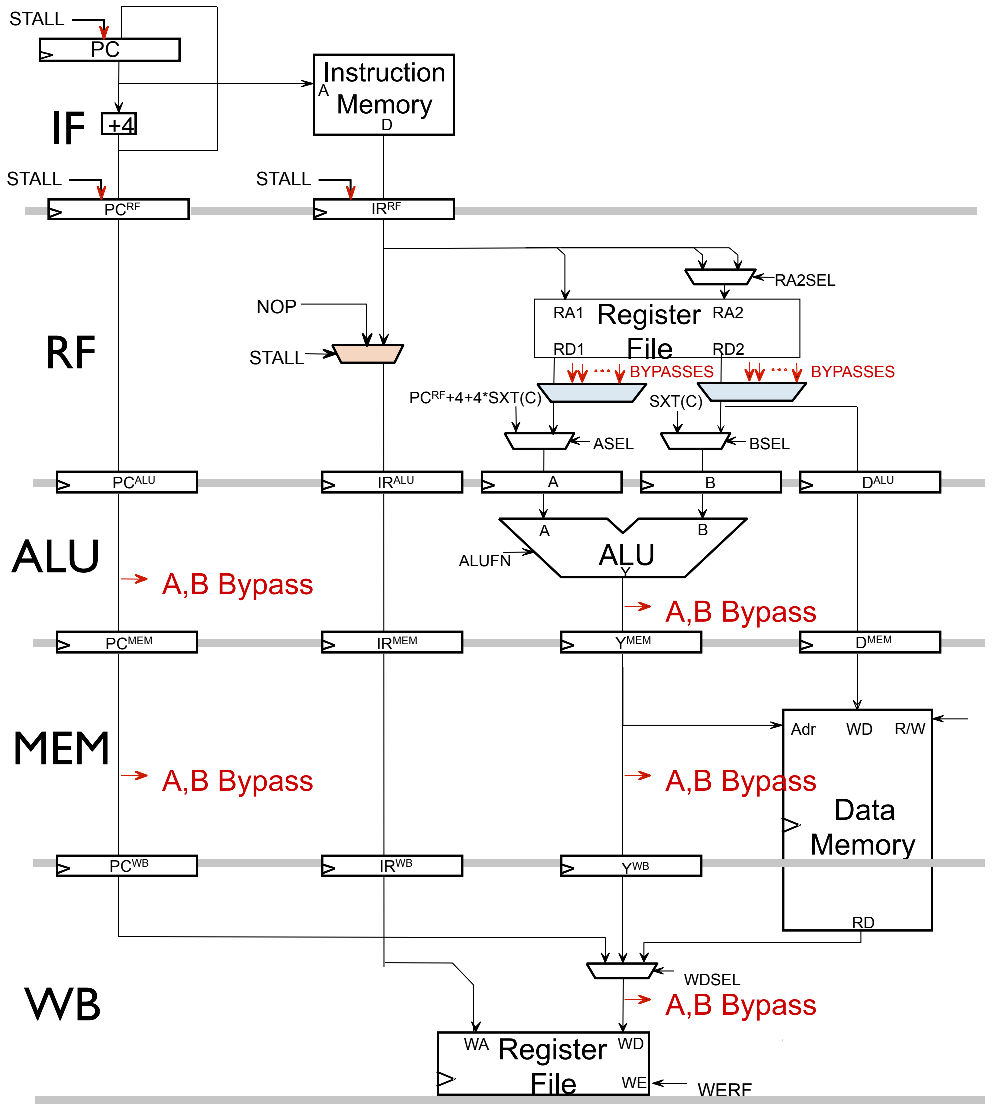
R2 updated

Bypass Logic



- Add bypass muxes to RF outputs
- Route ALU, MEM, WB outputs to mux inputs
- Bypass value if destination register of instruction in ALU, MEM, or WB matches source register of instruction in RF
  - But not R31!?
- What to do if multiple matches?
  - Select value from most recent instruction! (ALU > MEM > WB)

Fully Bypassed Pipeline



- Some instructions write PC +4…
- Route PC^{ALU} and PC^{MEM} as additional bypass mux inputs
- Bypasses are expensive
  - Lots of wiring & large muxes
  - May affect clock cycle time…
- But full bypassing is not needed! We can always stall
  - e.g., just bypass from ALU
- With a fully bypassed pipeline, do we still need the stall signal?

Load-To-Use Stalls

- Bypassing cannot eliminate load delays because data is not available until the WB stage!

- Bypassing from WB still saves a cycle:

```
LD(R1, 1, R2)
SUBC(R2, 4, R3)
MUL(R6, R7, R8)
XOR(R9, R10, R11)
…
```

	1	2	3	4	5	6	7
IF	LD	SUBC	MUL	MUL	MUL	XOR	
RF		LD	SUBC	SUBC	SUBC	MUL	XOR
ALU			LD	**NOP**	**NOP**	SUBC	MUL
MEM				LD	NOP	NOP	SUBC
WB					LD	NOP	NOP

LD data available ↗ ↖ R2 updated

Summary: Pipelining with Data Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
  - Simple, wastes cycles, higher CPI
- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated
  - More expensive, lower CPI
  - Still needs stalls when result is produced after ALU stage
  - Can use fewer bypasses & stall more often
- More pipeline stages → More frequent data hazards
  - Lower t_{CK}, but higher CPI

Compilers Can Help

- Compilers can rearrange code to put dependent instructions farther away
- Example:

```
LD(R1, 1, R2)
SUBC(R2, 4, R3)
MUL(R6, R7, R8)
XOR(R9, R10, R11)
…
```

2 stalls (w/ bypasses)

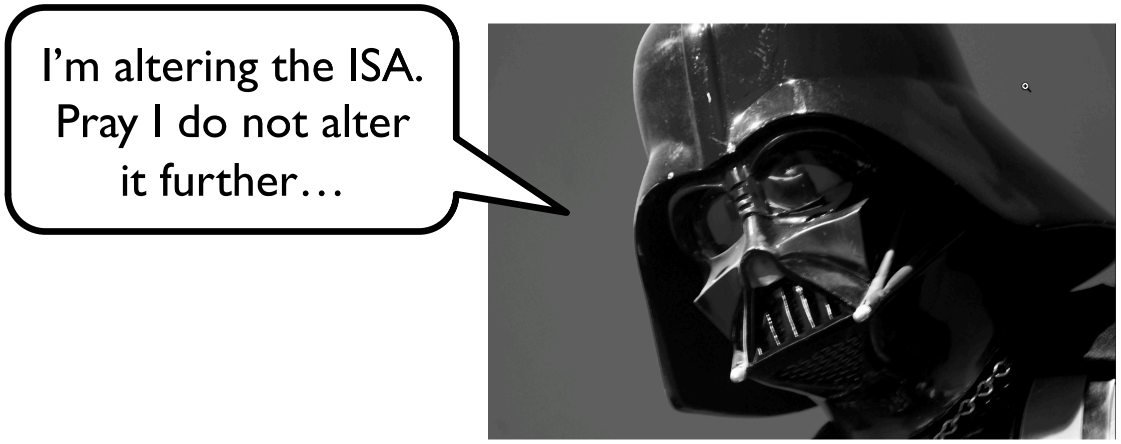
```
LD(R1, 1, R2)
MUL(R6, R7, R8)
XOR(R9, R10, R11)
SUBC(R2, 4, R3)
…
```

No stalls

- Only works well when compiler can find independent instructions to move around!

Or take the lazy route…

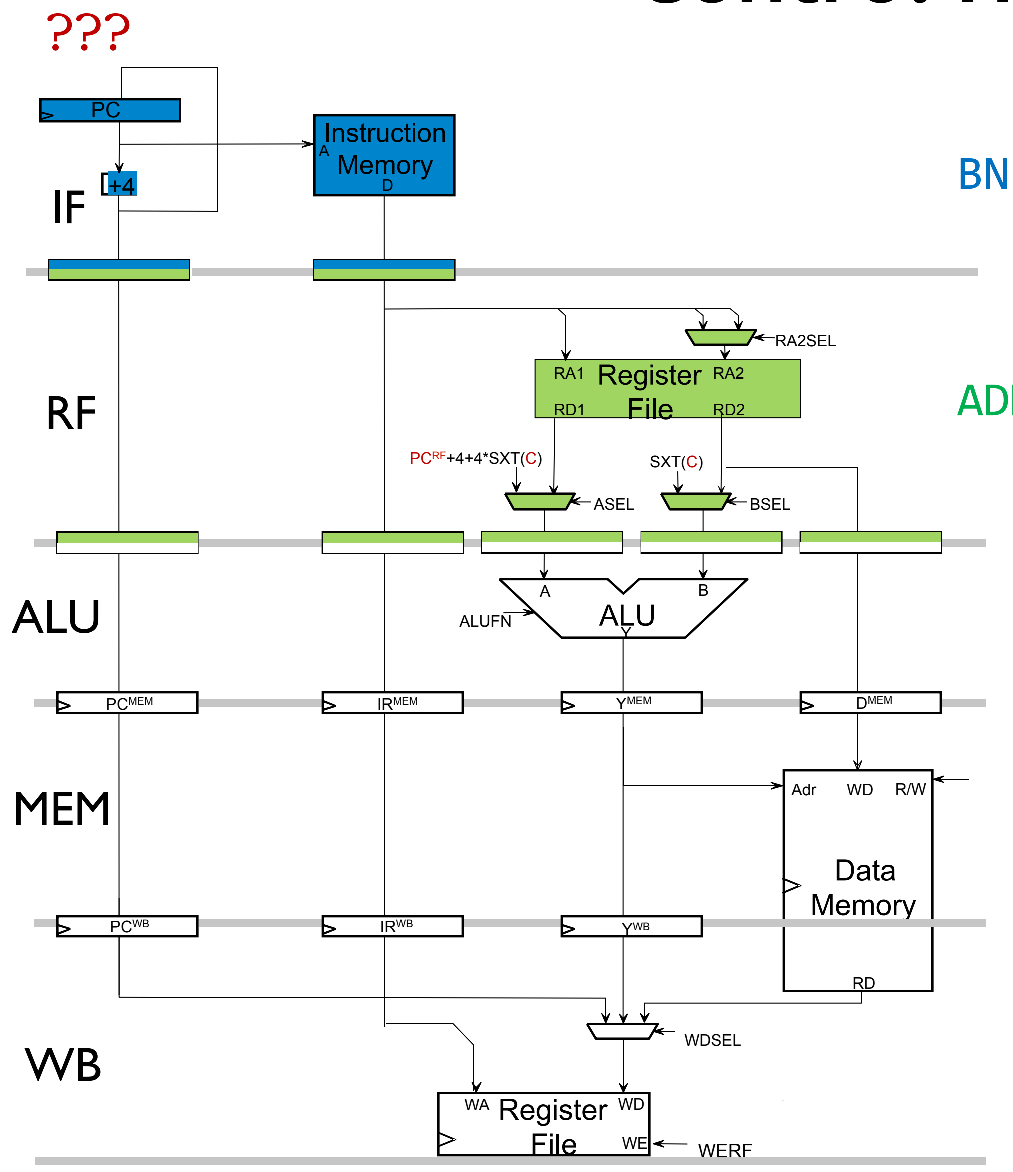
- Don’t stall or bypass, just change the ISA so that registers are updated with a 3-instruction delay!
  - Compiler writers will love this!
  - Programmers will love this!
  - You will love this when you decide to release an 8-stage pipelined processor!



Roger Schultz
(CC-BY 2.0)

- ISAs outlive implementations, this is a bad idea

Control Hazards



BNE

ADDC

```
loop:   ADDC(R1, 4, R2)
        BNE(R3, loop)
        SUB(R6, R7, R8)
        ...
```

How do we set NextPC?

Control Hazards

- What do we need to compute NextPC?
  - BEQ/BNE:

```
BEQ(Ra, label, Rc):
    Reg[Rc] ← PC + 4
    if (Reg[Ra] == 0)
        PC ← PC + 4 + 4*SXT(offset)
    else
        PC ← PC + 4
```

Opcode, offset, PC+4, Reg[Ra]

  - JMP:

```
JMP(Ra, Rc):
    Reg[Rc] ← PC + 4
    PC ← Reg[Ra]
```

Opcode, Reg[Ra]

Unknown until RF stage…

  - All other instructions: Opcode, PC+4
  - (Exceptions also change PC, we'll deal with them later)

Resolving Control Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages

- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated

- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value

Resolving Control Hazards With Stalls

- If branch or jump in IF, stall IF for one cycle
- Assume BNE is always taken in example code

```
loop:   ADDC(R1, -1, R3)
        MUL(R4, R5, R6)
        BNE(R3, loop)
        SUB(R6, R7, R8)
       …
```

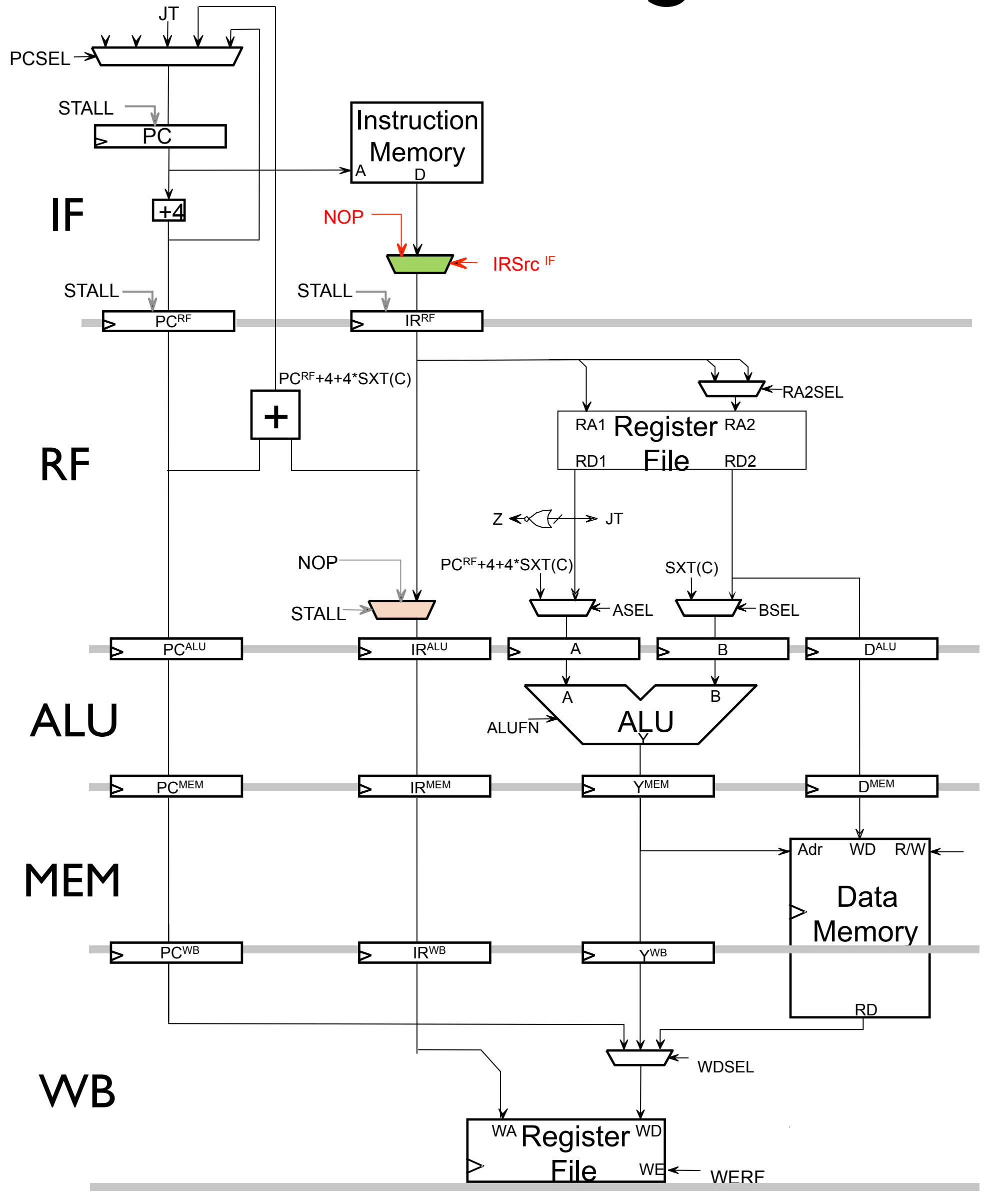
	1	2	3	4	5	6	7	8	9
IF	ADDC	MUL	BNE	**NOP**	ADDC	MUL	BNE	**NOP**	ADDC
RF		ADDC	MUL	BNE	NOP	ADDC	MUL	BNE	NOP
ALU			ADDC	MUL	BNE	NOP	ADDC	MUL	BNE
MEM				ADDC	MUL	BNE	NOP	ADDC	MUL
WB					ADDC	MUL	BNE	NOP	ADDC

$\nwarrow$ R3!=0 → Taken

$\nwarrow$ R3!=0

- Steady-state CPI?

Stall Logic For Control Hazards



- IRSrc^{IF} control signal
- If $\text{opcode}^{\text{RF}}$ == JMP, BEQ, BNE
  - IRSrc^{IF}=1, inject NOP
  - Set PCSEL to load branch or jump target

ISA Issue: Simple vs Complex Branches

- Beta has very simple branch condition
  - Reg[Ra]==0 easily computed in RF
- Other ISAs have more complex branches (e.g., branch if greater than) that are resolved in ALU
- What if branches were resolved in ALU stage?

	1	2	3	4	5	6	7	8
IF	ADDC	MUL	BNE	SUB	**NOP**	ADDC	MUL	BNE
RF		ADDC	MUL	BNE	**NOP**	NOP	ADDC	MUL
ALU			ADDC	MUL	BNE	NOP	NOP	ADDC
MEM				ADDC	MUL	BNE	NOP	NOP
WB					ADDC	MUL	BNE	NOP

More annulments (but sometimes fewer instructions)

Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages

- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated

- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → annul & restart with correct value

Resolving Hazards with Speculation

- What's a good guess for NextPC? PC+4

- Assume BNE is not taken in example

```
loop:   ADDC(R1, -1, R3)
        MUL(R4, R5, R6)
        BNE(R3, loop)
        SUB(R6, R7, R8)
        XOR(R9, R10, R11)
        …
```

	1	2	3	4	5	6	7	8	9
IF	ADDC	MUL	BNE	SUB	XOR				
RF		ADDC	MUL	BNE	SUB	XOR			
ALU			ADDC	MUL	BNE	SUB	XOR		
MEM				ADDC	MUL	BNE	SUB	XOR	
WB					ADDC	MUL	BNE	SUB	XOR

Start fetching at PC+4 (SUB) but BNE not resolved yet…

Guessed right, keep going

Resolving Hazards with Speculation

- What's a good guess for NextPC? PC+4
- Assume BNE is taken in example

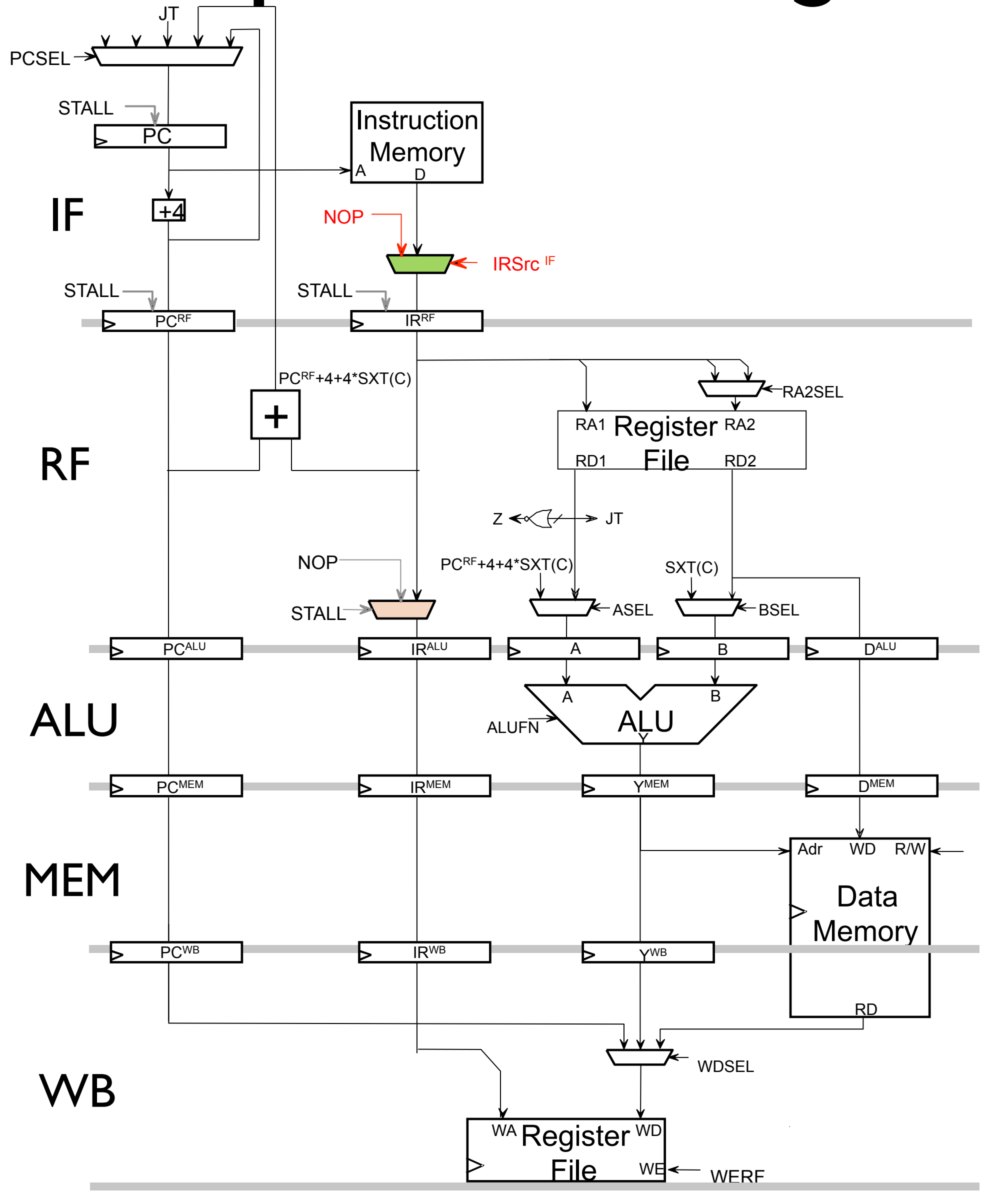
```
loop:   ADDC(R1, -1, R3)
        MUL(R4, R5, R6)
        BNE(R3, loop)
        SUB(R6, R7, R8)
        XOR(R9, R10, R11)
        …
```

	1	2	3	4	5	6	7	8	9
IF	ADDC	MUL	BNE	SUB	ADDC	MUL	BNE	SUB	ADDC
RF		ADDC	MUL	BNE	**NOP**	ADDC	MUL	BNE	**NOP**
ALU			ADDC	MUL	BNE	NOP	ADDC	MUL	BNE
MEM				ADDC	MUL	BNE	NOP	ADDC	MUL
WB					ADDC	MUL	BNE	NOP	ADDC

Start fetching at PC+4 (SUB) but BNE not resolved yet…

Guessed wrong, annul SUB

Speculation Logic For Control Hazards



- This looks familiar…
- IRSrc^{IF} control signal
- If opcode^{RF} == JMP or taken BEQ/BNE
  - IRSrc^{IF}=1, inject NOP to annul fetched inst. (aka “branch annulment”)
  - Set PCSEL to load branch or jump target

Branch Prediction

- Always guessing PC+4 wastes a cycle on taken branches and jumps
  - ~10% higher CPI
- With deeper pipelines, taken branches waste many more cycles
  - E.g., Intel Nehalem takes about 17 cycles to resolve whether a branch is taken
- Modern CPUs dynamically predict the outcome of control-flow instructions
  - Predict <u>both</u> the branch condition and the target
  - Works well because branches have repeated behavior
    - E.g. branches for loops are usually taken
    - E.g. termination/limit/error tests are usually not taken

Branch Delay Slots

- Change the ISA so that the instruction following a jump or branch is always executed

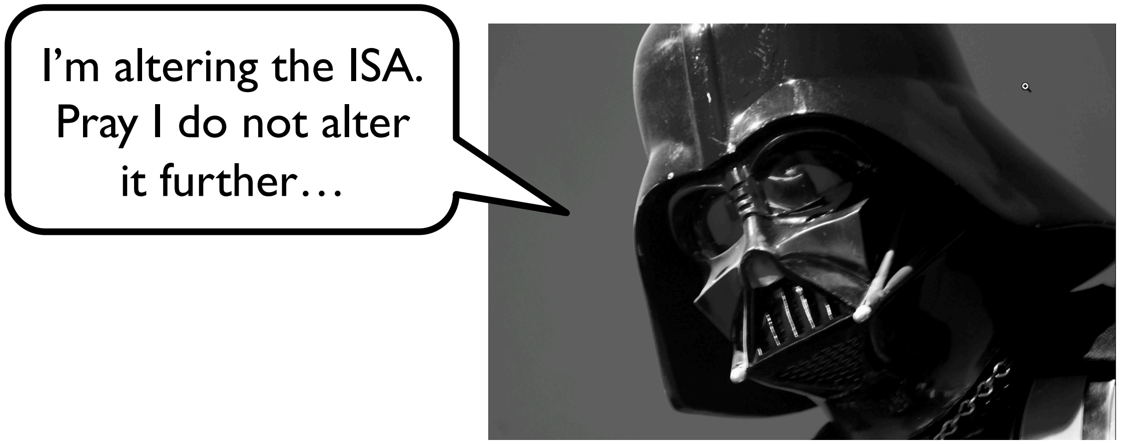
Delay slot instruction executes regardless of branch outcome

```
loop:   ADDC(R1, -1, R3)
        BNE(R3, loop)
        MUL(R4, R5, R6)
        SUB(R6, R7, R8)
       XOR(R9, R10, R11)
        …
```

	1	2	3	4	5	6	7	8
IF	ADDC	BNE	MUL	ADDC	BNE	MUL	ADDC	BNE
RF		ADDC	BNE	MUL	ADDC	BNE	MUL	ADDC
ALU			ADDC	BNE	MUL	ADDC	BNE	MUL
MEM				ADDC	BNE	MUL	ADDC	BNE
WB					ADDC	BNE	MUL	ADDC

Branch Delay Slots

- Pro: If compiler can fill slot with useful instruction, no branch/jump penalty
- Cons:
  - Can't fill slot with useful work ~50% of the time → Must insert NOP, longer code
  - Longer pipeline → More delay slots?
  - Branch prediction works better in practice



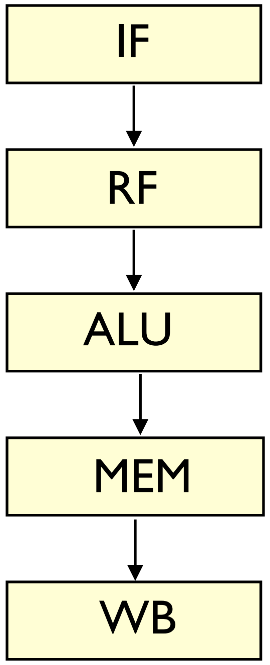
Roger Schultz (CC-BY 2.0)

- ISAs outlive implementations, this is a bad idea

Exceptions

- On an exception, need to:
  - Save current PC+4 in XP (R30)
  - Load PC with exception vector (IllOp or XAdr)
- Exceptions cause control flow hazards!
  - They are implicit branches
- Want precise exceptions:
  - All preceding instructions must have completed
  - Instruction causing exception and future instructions must not have executed
    - No updates to register or memory
  - Simple in single-cycle machines, more complex with pipelining

When Can Exceptions Happen?



Memory fault (e.g., illegal memory address)

Illegal instruction

Arithmetic exception (e.g., divide by zero)

Memory fault (e.g., illegal memory address)

- Instructions following the one that causes the exception may already be in the pipeline…
- … but none has written registers or memory yet ☺

Resolving Exceptions

- If an instruction has an exception at stage i
  - Turn that instruction into BNE(R31, 0, XP) to save PC+4
  - Annul instructions in stages i-1,…,1 (flush the pipeline)
  - Set PC ← IllOp or XAdr

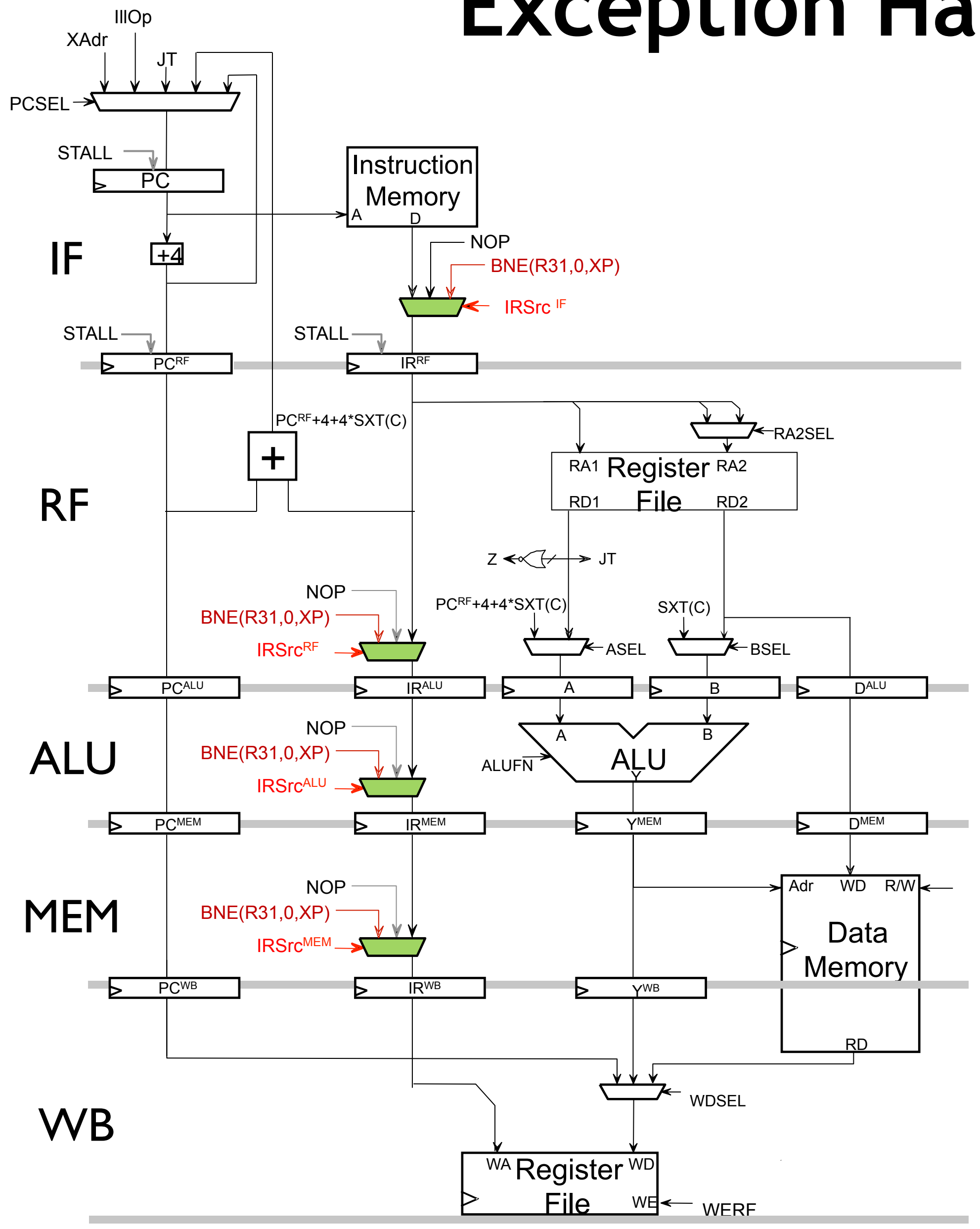
- Example: LD has memory fault

```
LD(R1, 4, R2)
ST(R3, 0, R4)
MUL(R4, R5, R6)
SUB(R7, R8, R9)
```

```
XAdr:   ADDC
        ST
```

	1	2	3	4	5	6
IF	LD	ST	MUL	SUB	ADDC	ST
RF		LD	ST	MUL	**NOP**	ADDC
ALU			LD	ST	**NOP**	NOP
MEM				LD	**NOP**	NOP
WB					**BNE**	NOP

Exception Handling Logic



- $\text{IRSrc}^{\{IF,RF,ALU,MEM\}}$ muxes to inject NOP or BNE
  - NOP if preceding instruction has an exception
  - BNE if instruction in current stage has an exception

Multiple Exceptions?

Causes memory fault

Invalid opcode

```
LD(R1, 4, R2)          Xadr:  ADDC       IllOp:  XORC
???                           ST                 SUBC
MUL(R4, R5, R6)               …                  …
SUB(R7, R8, R9)
```

	1	2	3	4	5	6
IF	LD	???	MUL	XORC	ADDC	ST
RF		LD	???	**NOP**	**NOP**	ADDC
ALU			LD	BNE	**NOP**	NOP
MEM				LD	**NOP**	NOP
WB					**BNE**	NOP

Invalid opcode detected

Memory fault detected

Works fine even if exception from latter instruction is detected first!

Asynchronous Interrupts

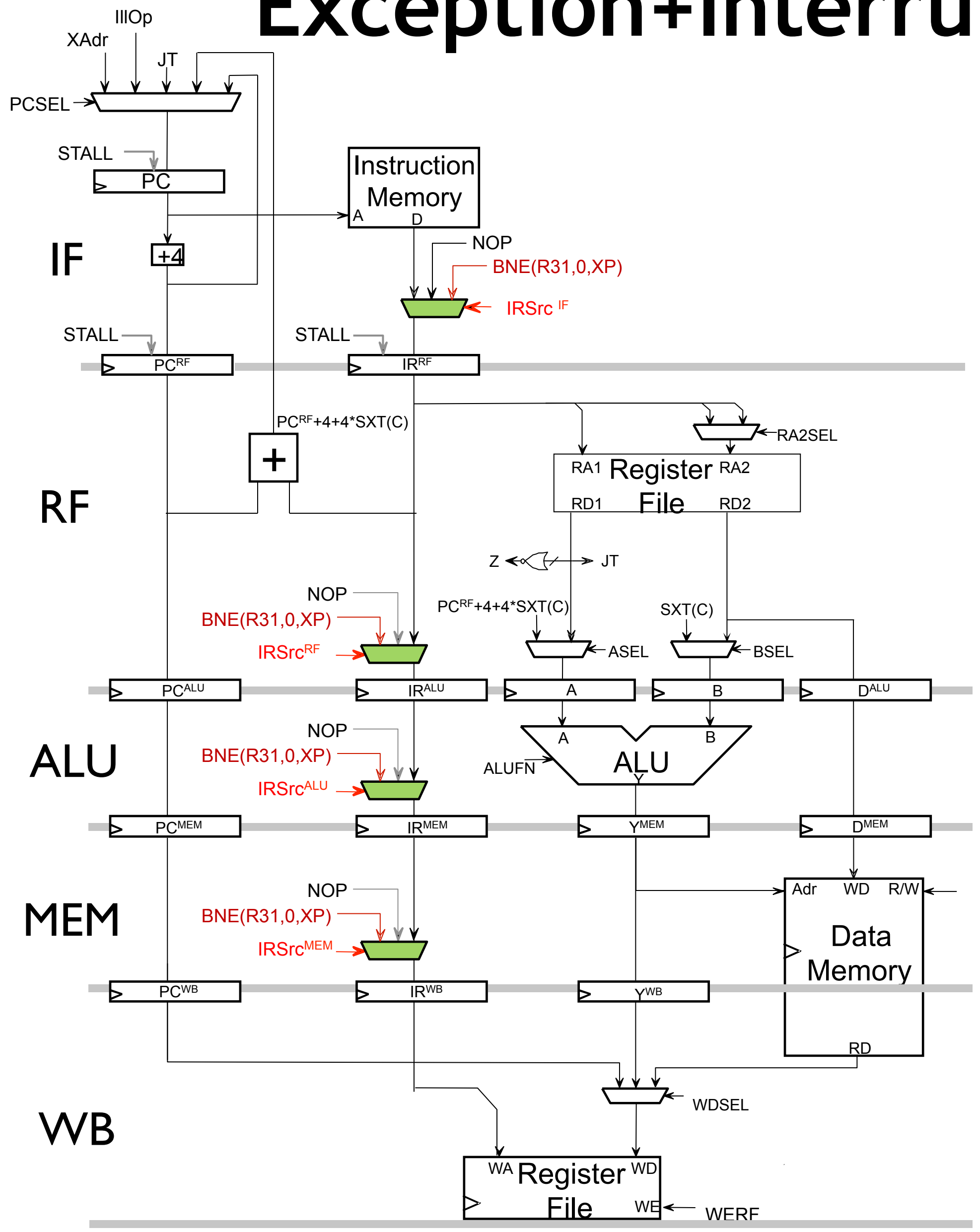
Interrupts are easier:

```
// Interrupted code:
        ...
        LD(...)
        ADD(...)
        SUB(...) <—— Interrupt Taken HERE
        …
// Interrupt handler:
XAdr:   OR(...)
        ...
        SUBC(xp,4,xp)
        JMP(xp)
```

	1	2	3	4
IF	ADD	**BNE**	OR	...
RF	LD	ADD	BNE	OR
ALU		LD	ADD	BNE
MEM			LD	ADD
WB				LD

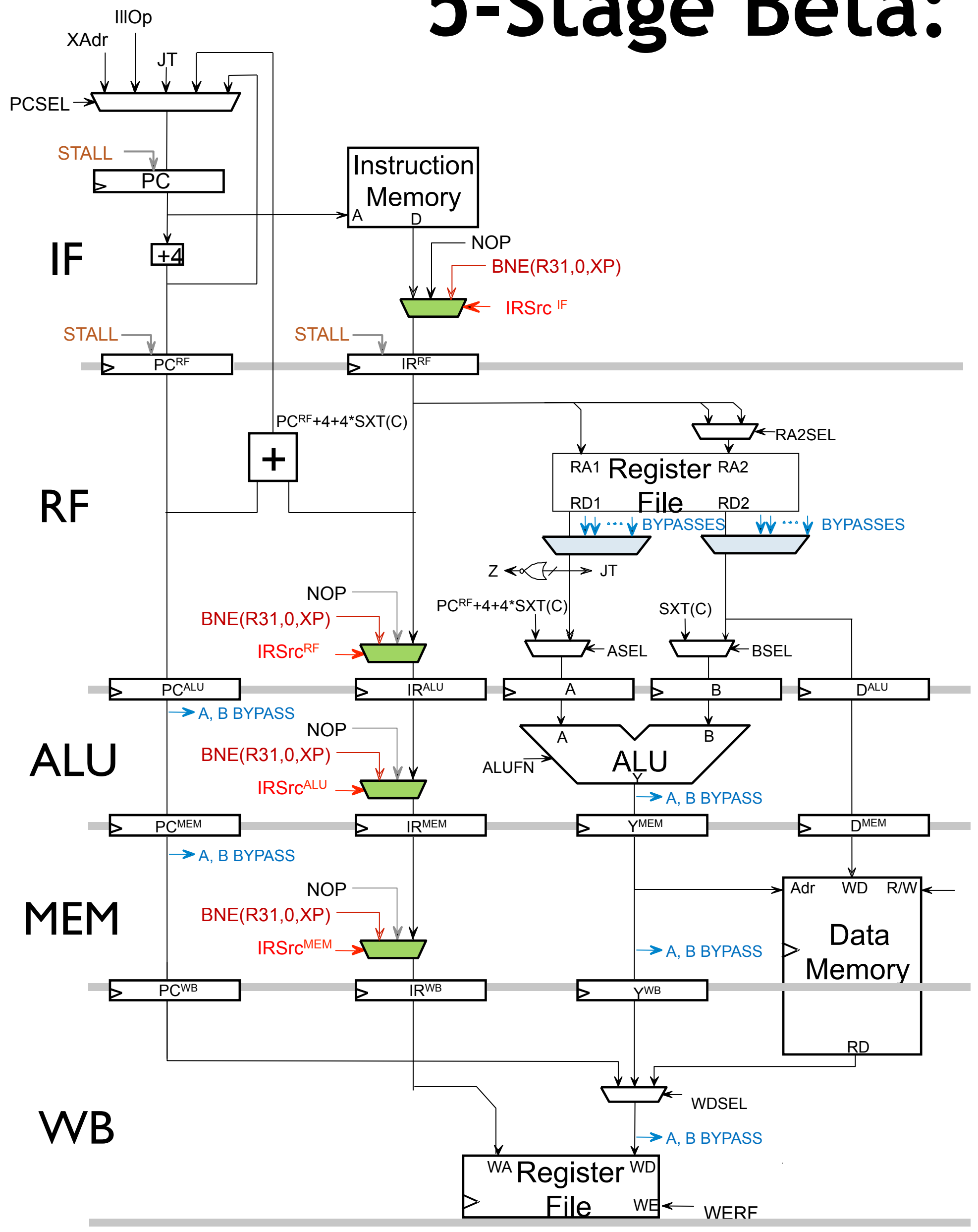
- Suppose interrupt is requested while SUB is in the IF stage (cycle 2)
- To handle:
  - Replace SUB instruction with BNE(…,XP)
  - Select Xadr as next PC
  - Code handler to return to SUB instruction
  - ADD and earlier insts. are unaffected

Exception+Interrupt Handling Logic



- Same as before
- $\text{IRSrc}^{\{\text{IF,RF,ALU,MEM}\}}$ muxes to inject NOP or BNE
  - NOP if preceding instruction has an exception
  - BNE if instruction in current stage has an exception
- Use IRSrc^{IF} mux to inject BNE on an interrupt (same as an exception in IF)

5-Stage Beta: Final Version



- Data hazards:
  - Stall IF and RF (STALL=1 + IRSrc^{RF}=NOP)
  - Bypass
- Control hazards: Speculate PC+4 and
  - JMP or taken branch in RF,
    - IRSrc^{IF}=NOP
    - PCSEL → JT/branch target
  - If exception at stage X
    - IRSrc^{X}=BNE
    - Previous IRSrc^{X}=NOP
    - PCSEL → XAdr or IllOp
  - If interrupt
    - IRSrc^{IF}=BNE
    - PCSEL → XAdr

Reminder: Resolving Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages

- Strategy 2: Bypass. Route data to the earlier pipeline stage as soon as it is calculated

- Strategy 3: Speculate
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value