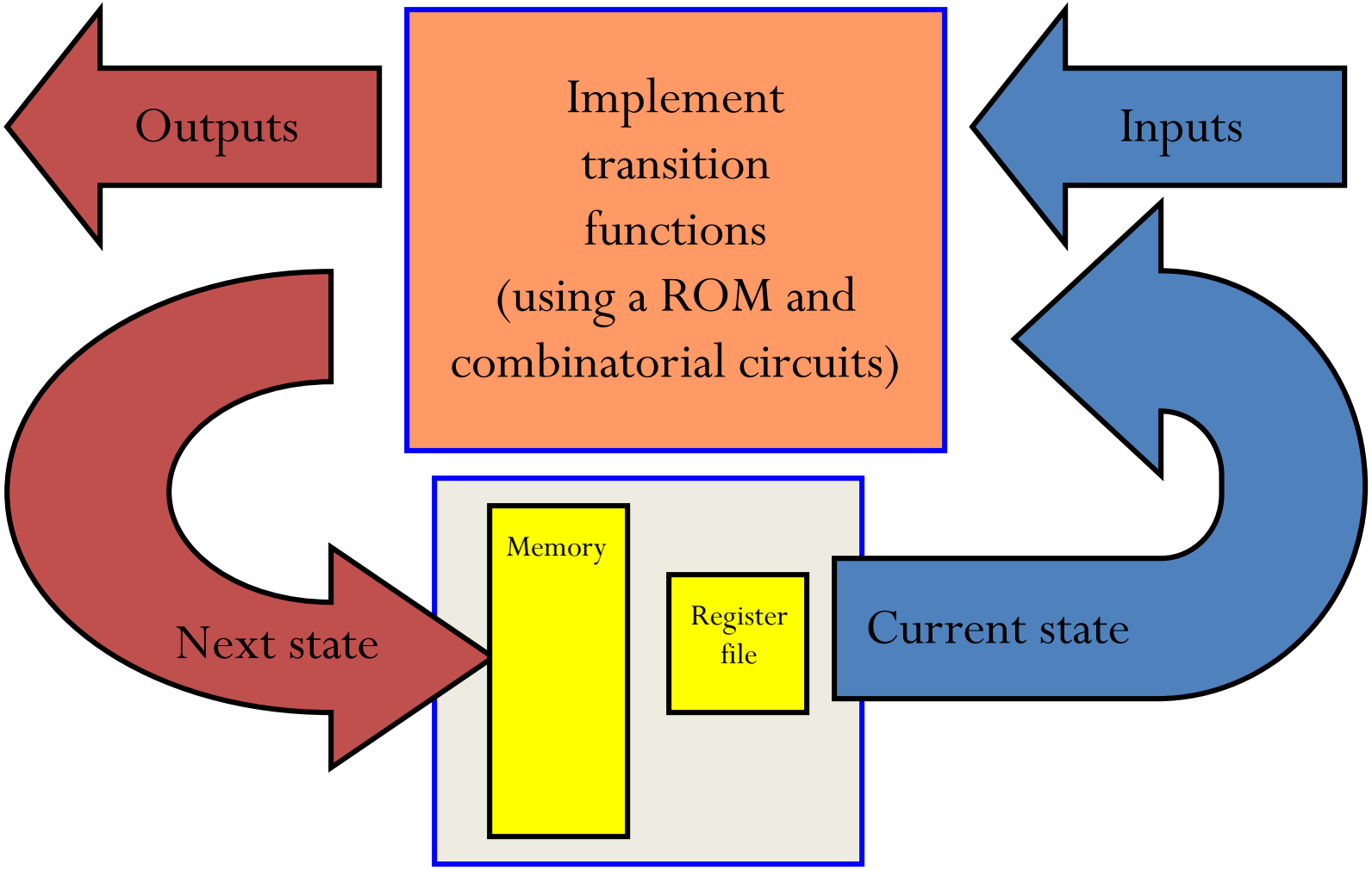
Microarchitecture

Outline

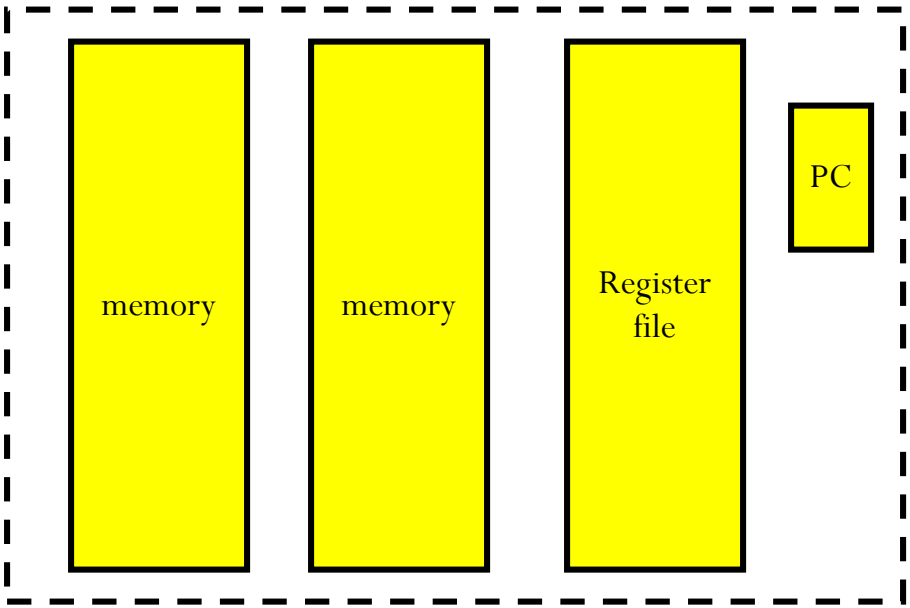
- **Introduction to Microarchitecture**
- ANNA Datapath
- ANNA Control
- Performance
- Pipelining
- Additional Performance Optimizations

Processor as a FSM



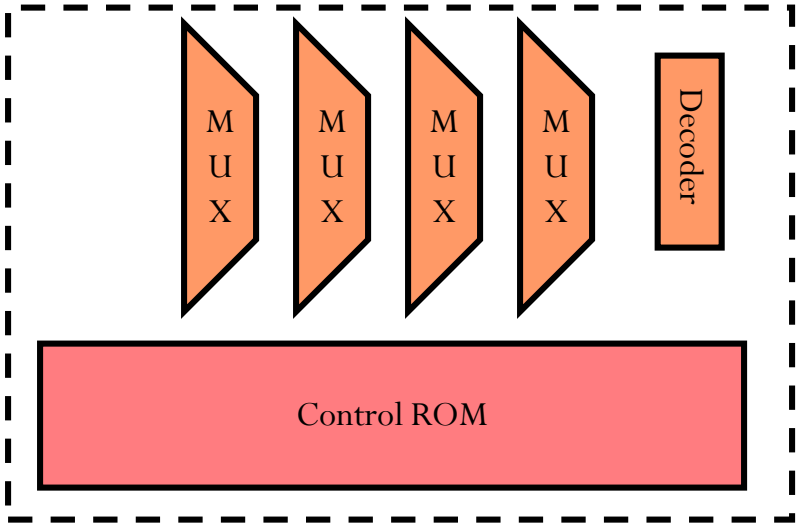
ANNA: $65{,}536 \times 16 + 8 \times 16 \approx 1{,}000{,}000$ bits

Microarchitecture Building Blocks

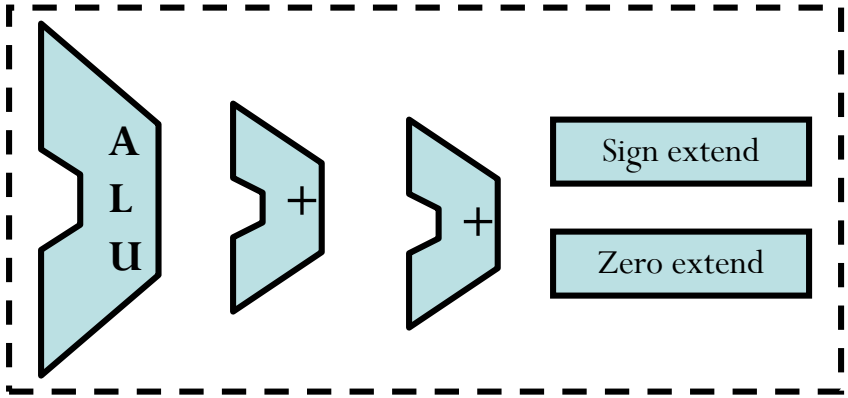


State

Control



Compute



Here are the pieces, go build yourself a processor!

Datapath and Control

Common to decomposes the system in two parts:

- **Datapath:** A collection of interconnected modules that perform all the relevant computation on the data.
  - For a CPU: Set of components that perform arithmetic operations and hold data.
- **Control Unit:** Coordinates the behavior of the datapath by issuing appropriate control signals.
  - For a CPU: Commands the datapath, memory, I/O devices, according to the current instruction.

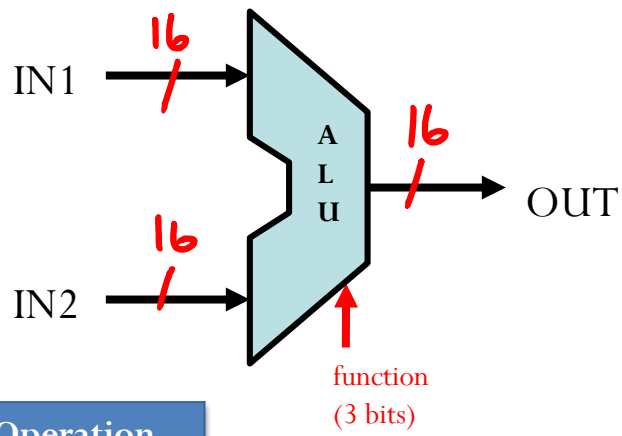
Outline

- Introduction to Microarchitecture
- **ANNA Datapath**
- ANNA Control
- Performance
- Pipelining
- Additional Performance Optimizations

ANNA Datapath Assumptions

- Datapath will omit these instructions:
  - `jalr`: left as an exercise
  - `in`, `out`: requires special I/O hardware
- Ignore startup: assume registers, memory, and PC have proper initial state.
- Initial datapath will be single cycle. Each instruction takes one cycle to execute.

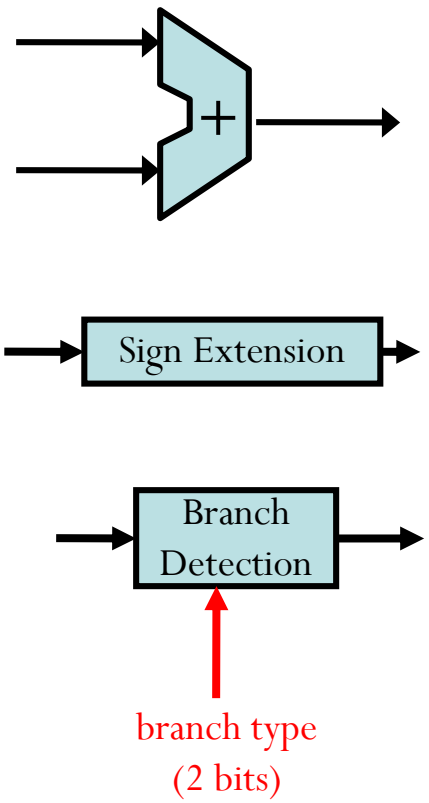
ALU



Function	Operation
000	add
001	sub
010	and
011	or
100	not
101	shf
110	lli
111	lui

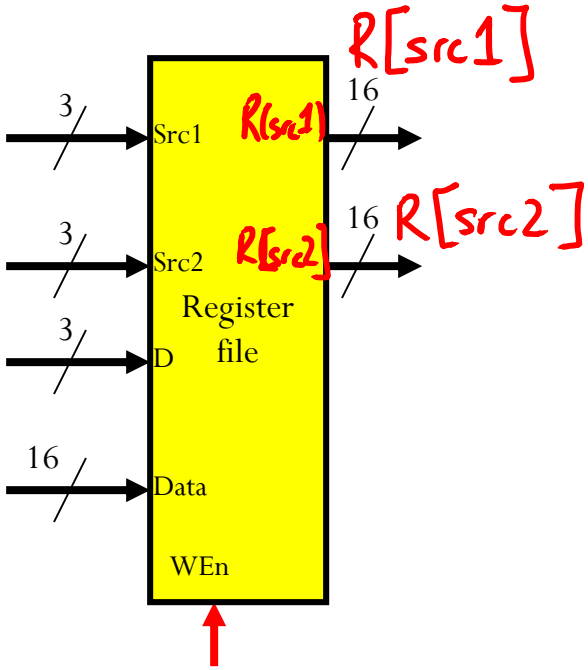
- For instructions listed, function bits are the lower three bits of the opcode.
  - `addi` also uses add
- For `lui` instructions:
  - IN1 contains current value of Rd (for lower 8 bits)
  - IN2 contains immediate
- For `lli` instructions:
  - IN1 is ignored
  - IN2 contains immediate.

Compute Building Blocks



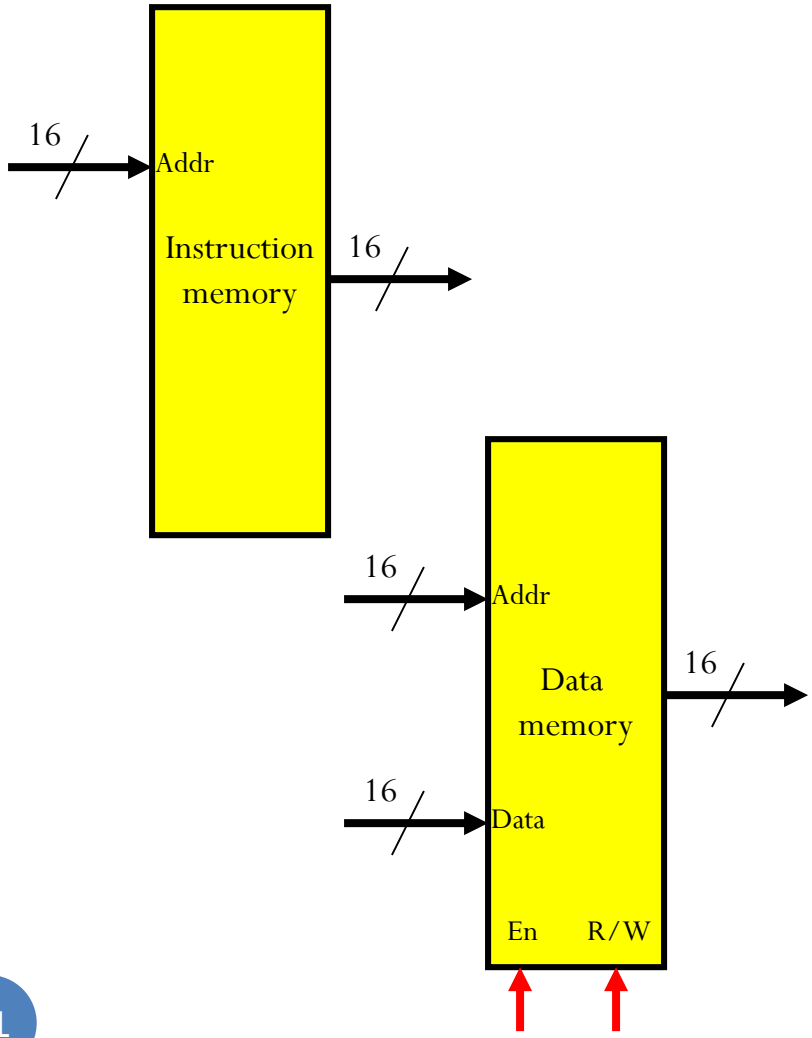
- Adder only adds numbers.
- Sign extension converts smaller number into 16-bit number by extending sign.
- Branch detection determines if a branch should be taken or not.
  - branch type input:
    - 00: `bez`
    - 01: `bgz`
    - 11: no branch
  - output:
    - 1: branch taken
    - 0: branch not taken

Register File

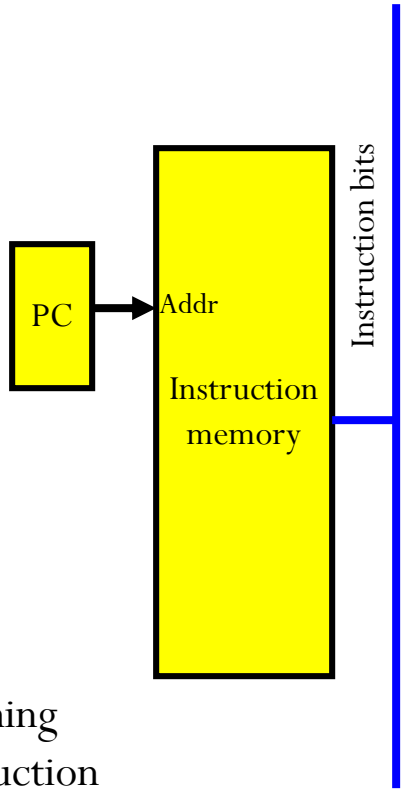


- Contains 8 registers.
  - Each is 16 bits.
- Two read ports (Src1, Src2) and one write (D) port.
  - Contains register number.
- Always enabled – can always read data.
- Write enable signal will be set for instructions that write to the register file.
- Has special logic to ignore writes to register 0.

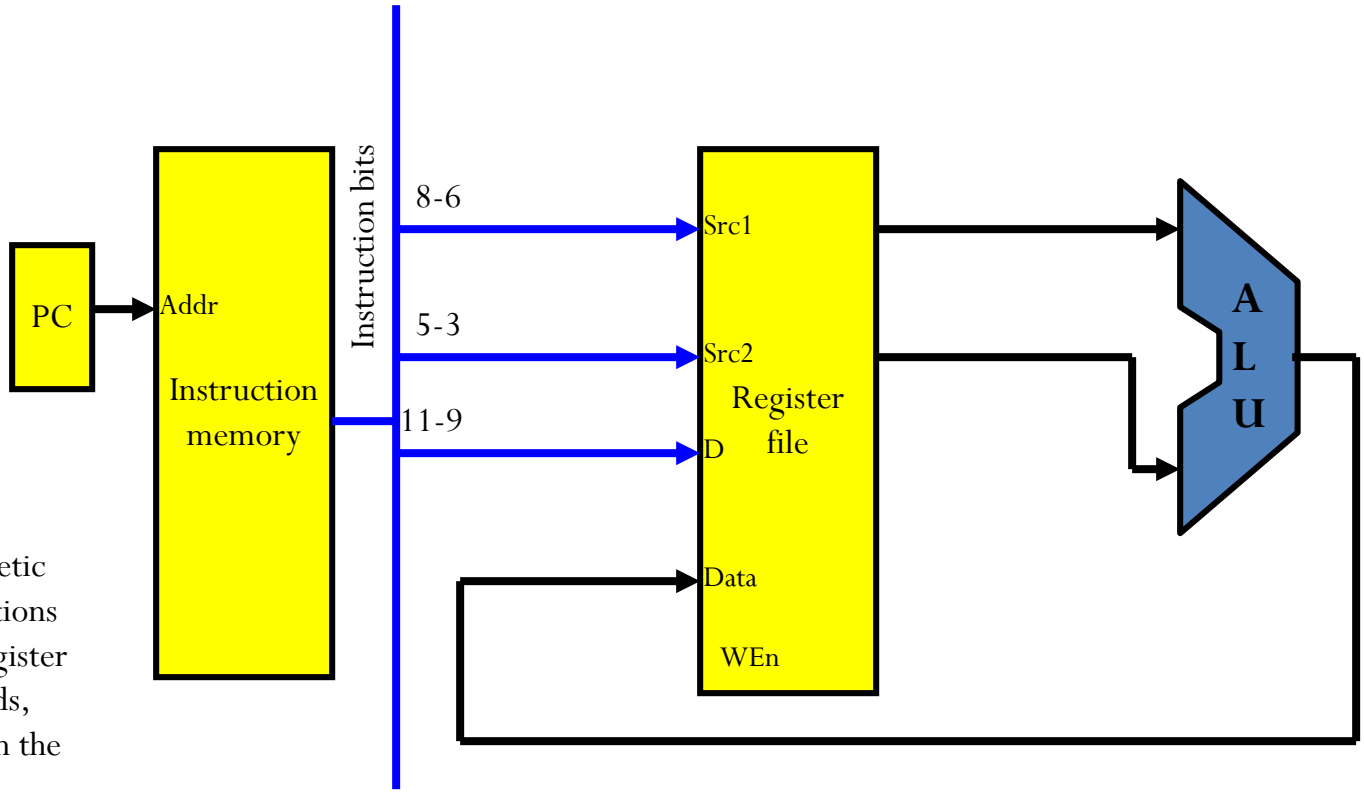
Memory



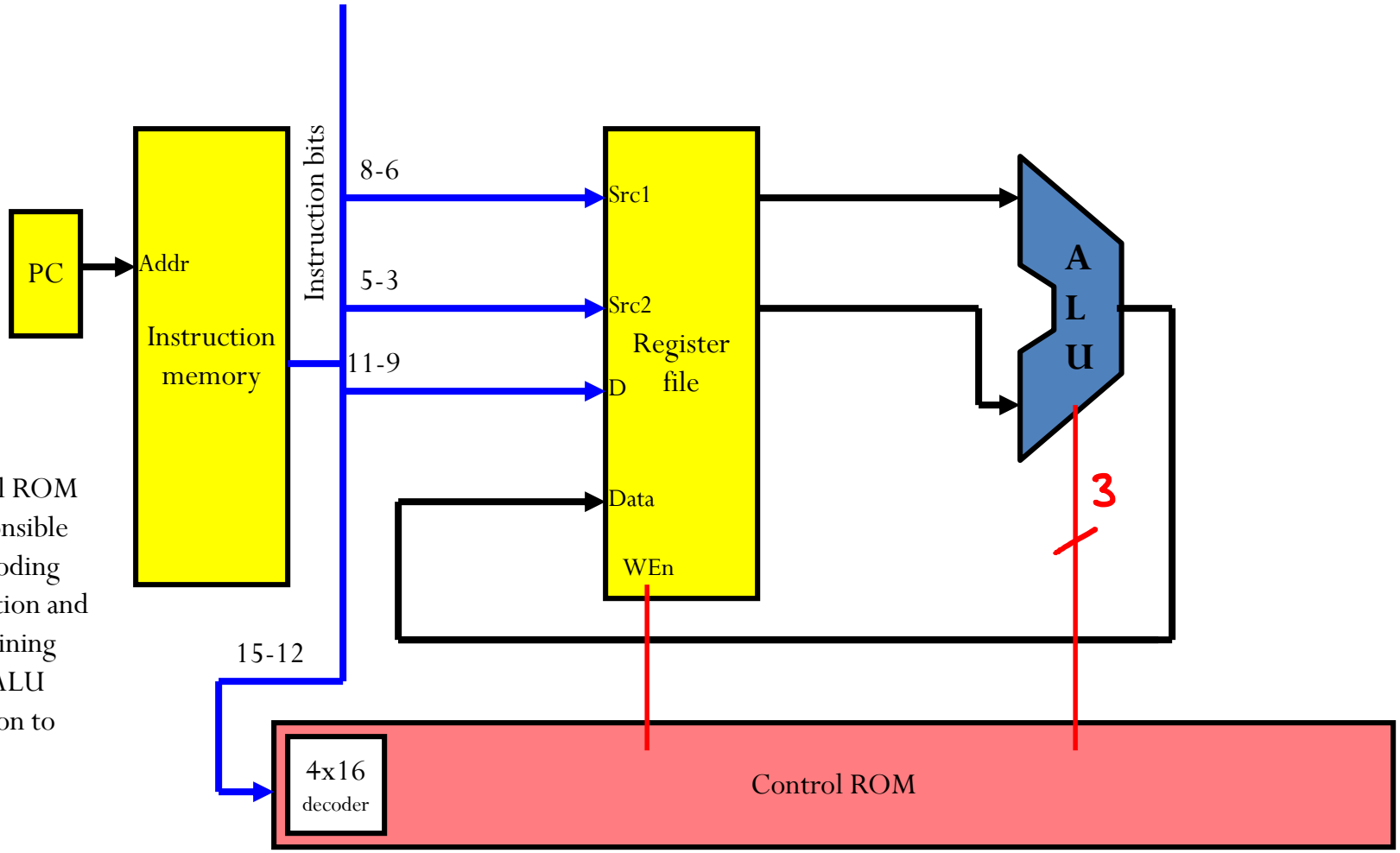
- Two separate memories
  - one for instructions
  - one for data
- Will be replaced with caches later
  - instruction and data caches are separate
- Ignore case where code uses value written to data memory
  - no self-modifying code
- Instruction memory is always enabled and can only read.
- Data memory has an enable (En) signal (set for `lw`/`sw`) and R/W signal (read: 0, write: 1).



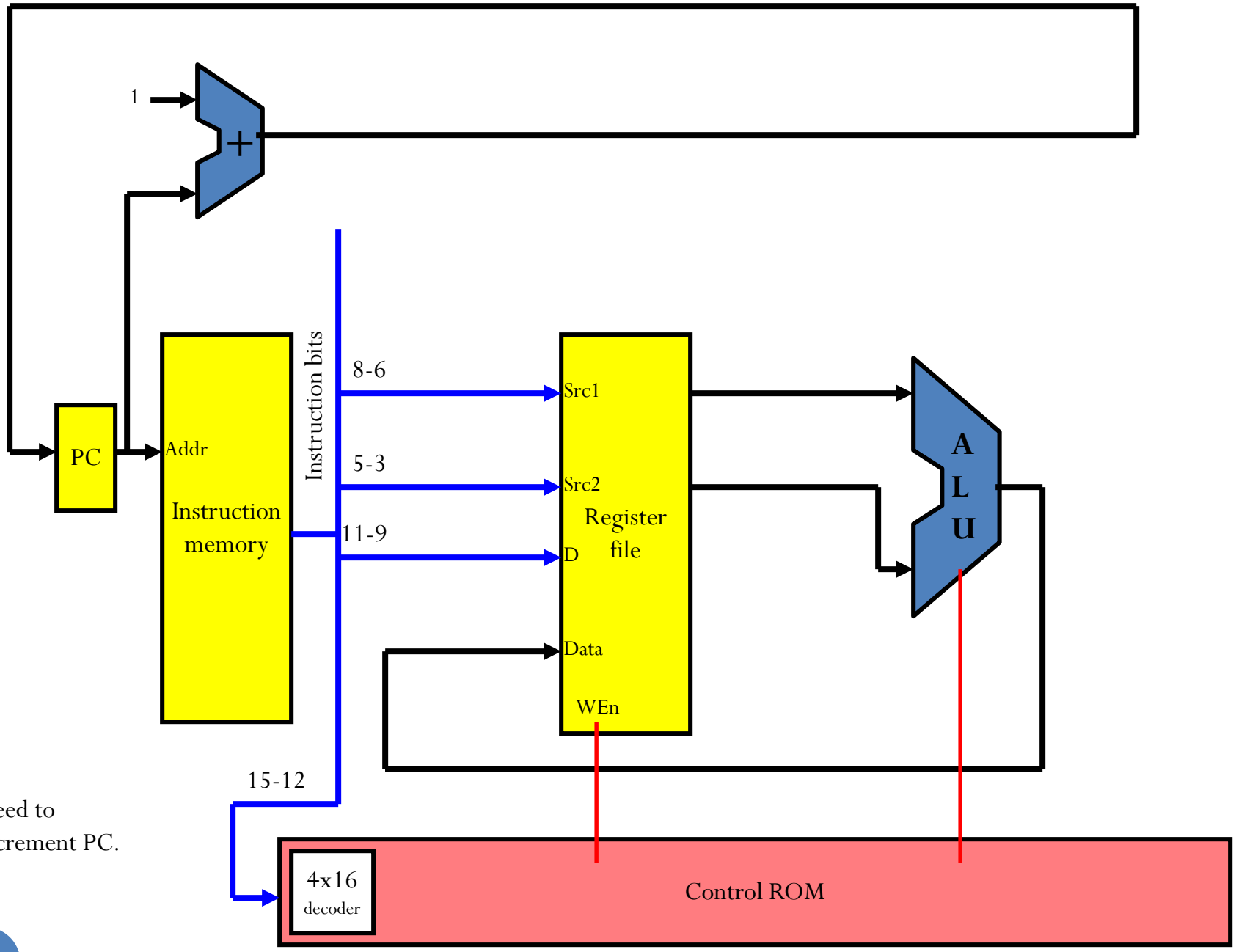
Fetching instruction from memory.



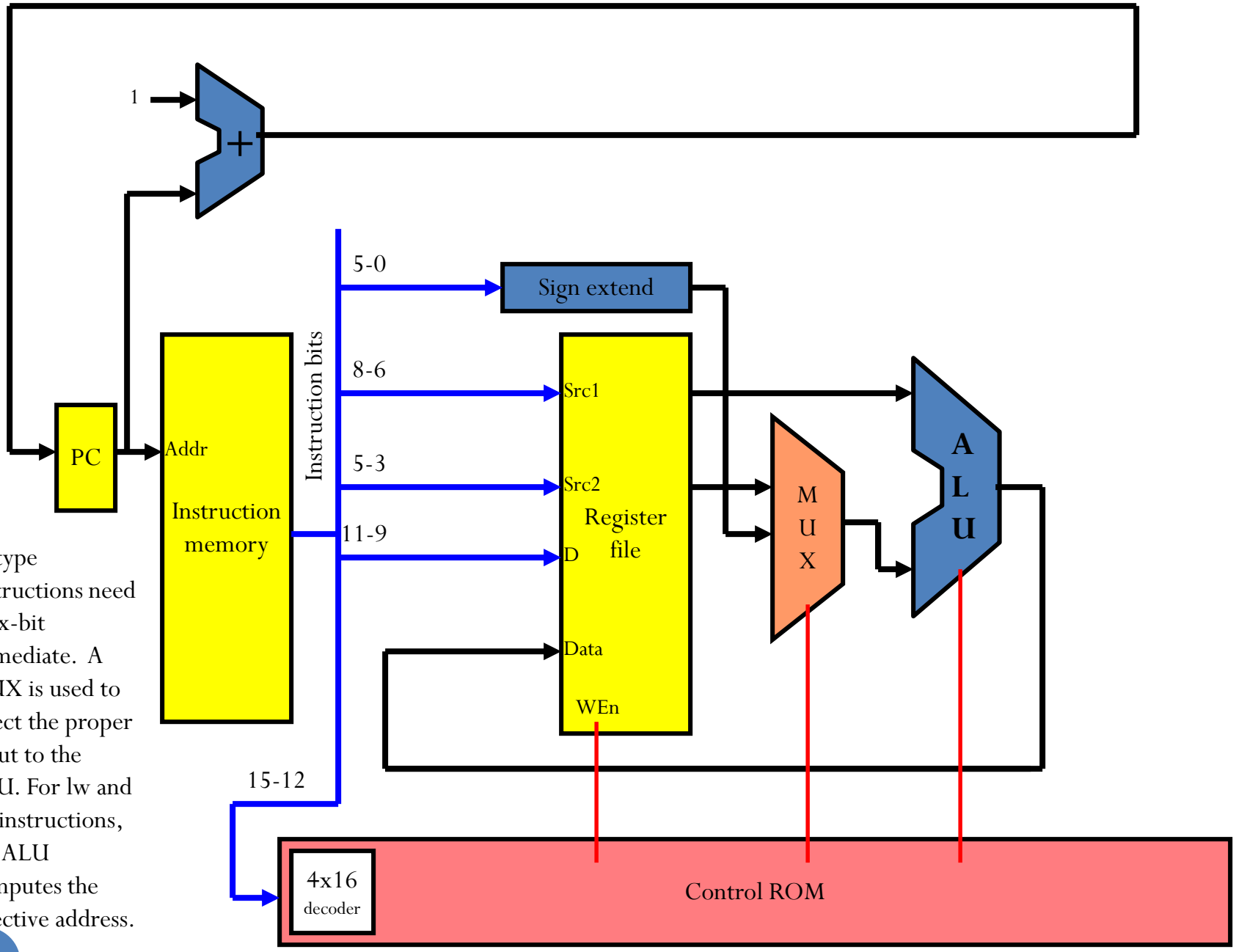
Arithmetic instructions read register operands, perform the desired operation in the ALU, and write back to register.



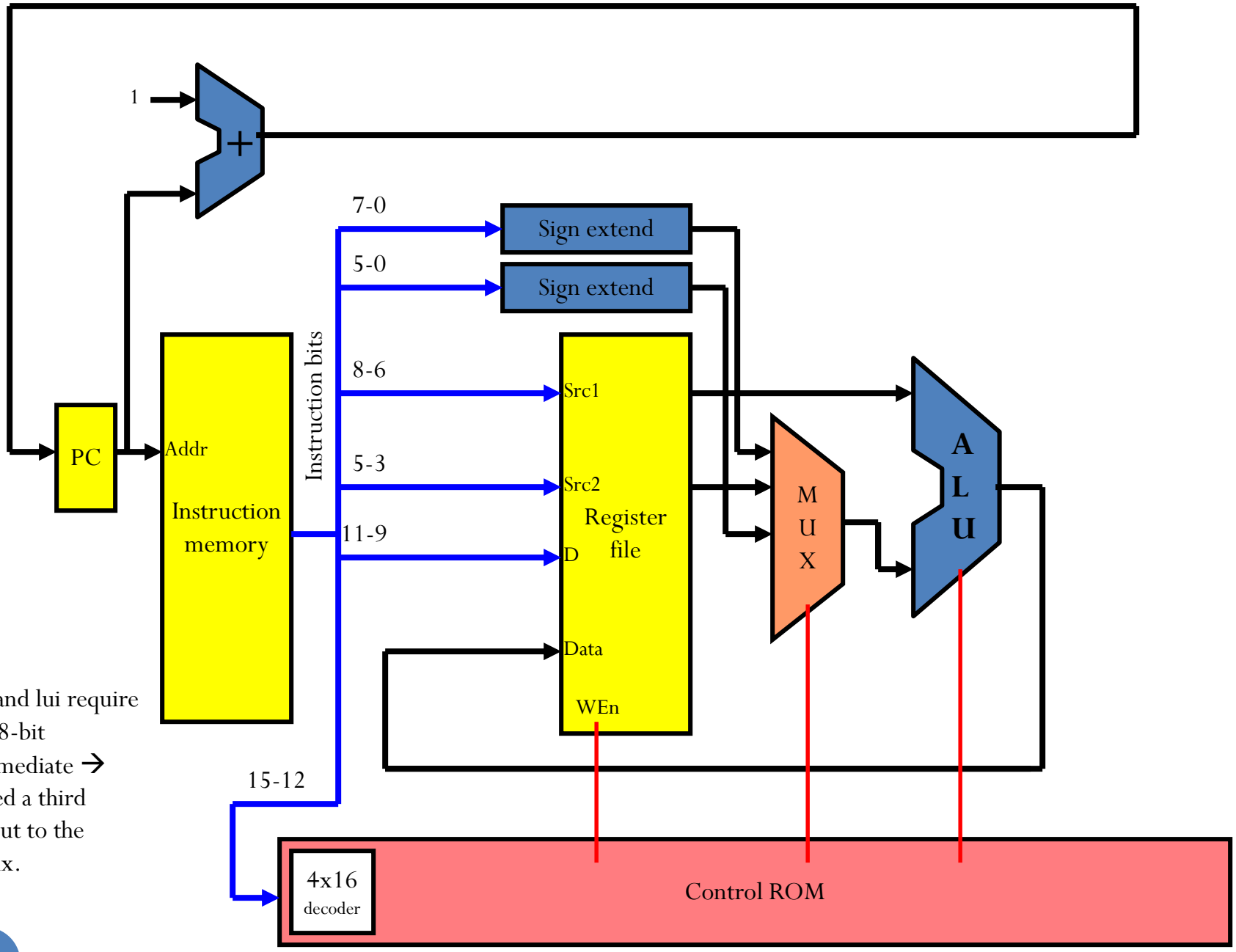
Control ROM is responsible for decoding instruction and determining which ALU operation to use.



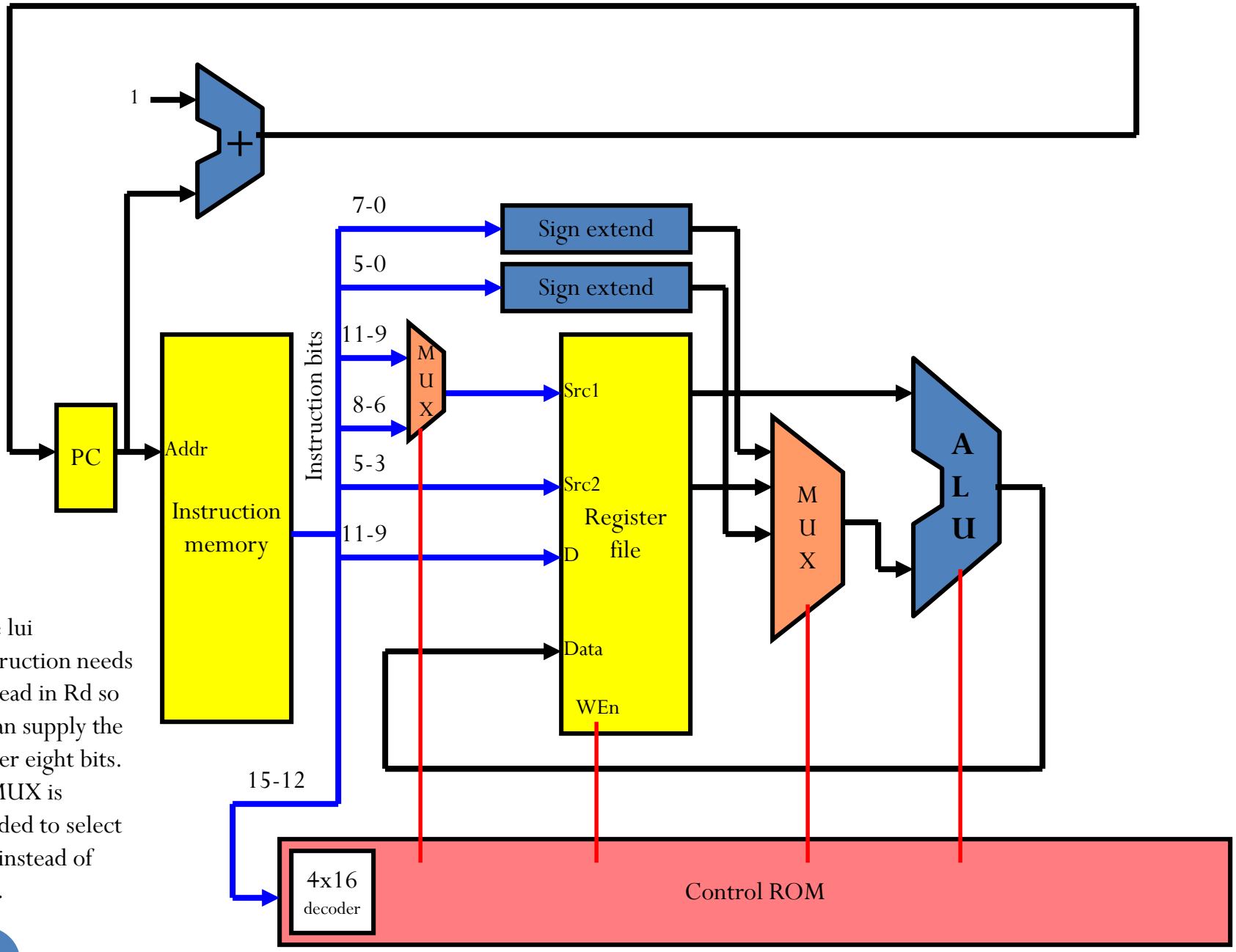
Need to increment PC.



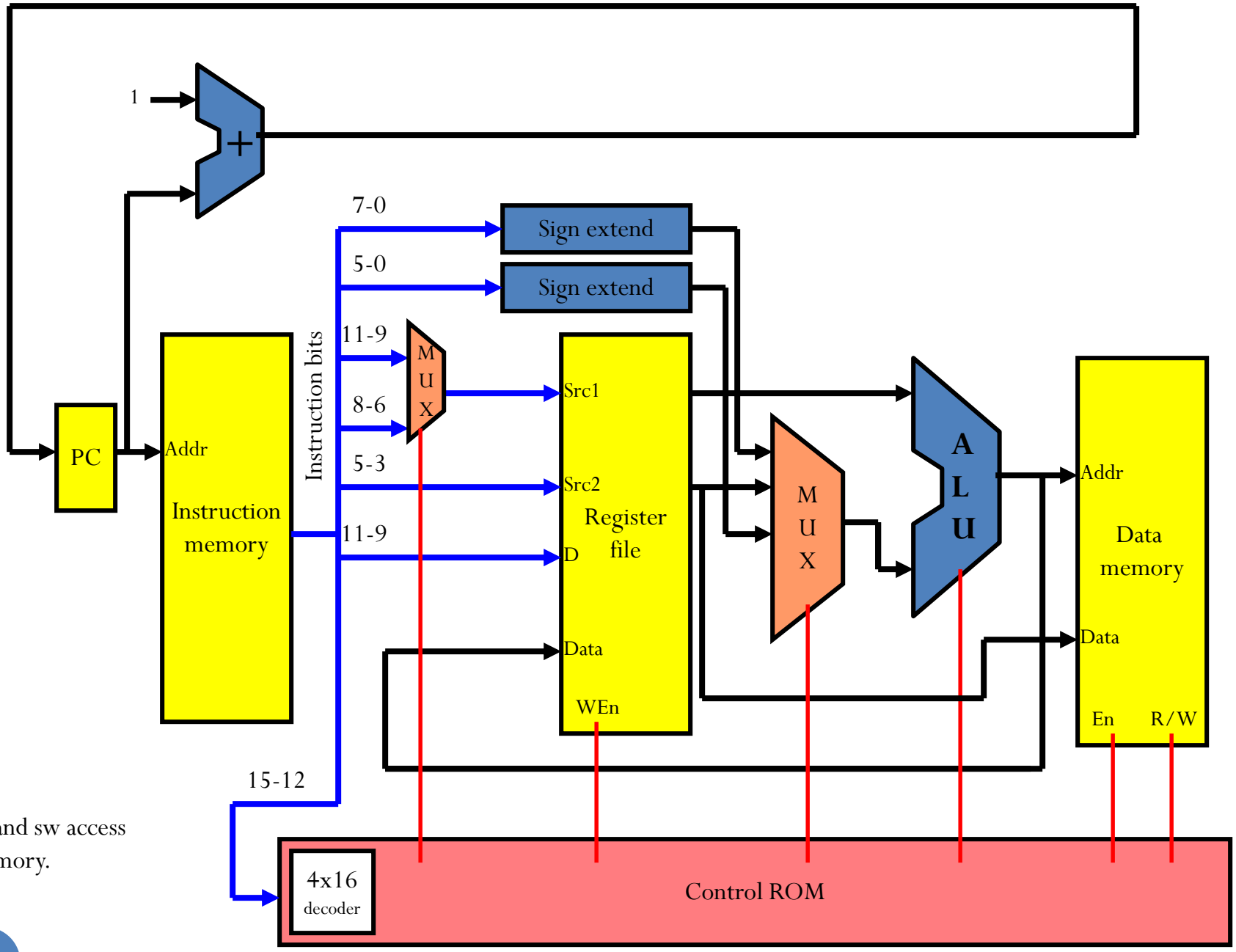
I6-type instructions need a six-bit immediate. A MUX is used to select the proper input to the ALU. For lw and sw instructions, the ALU computes the effective address.



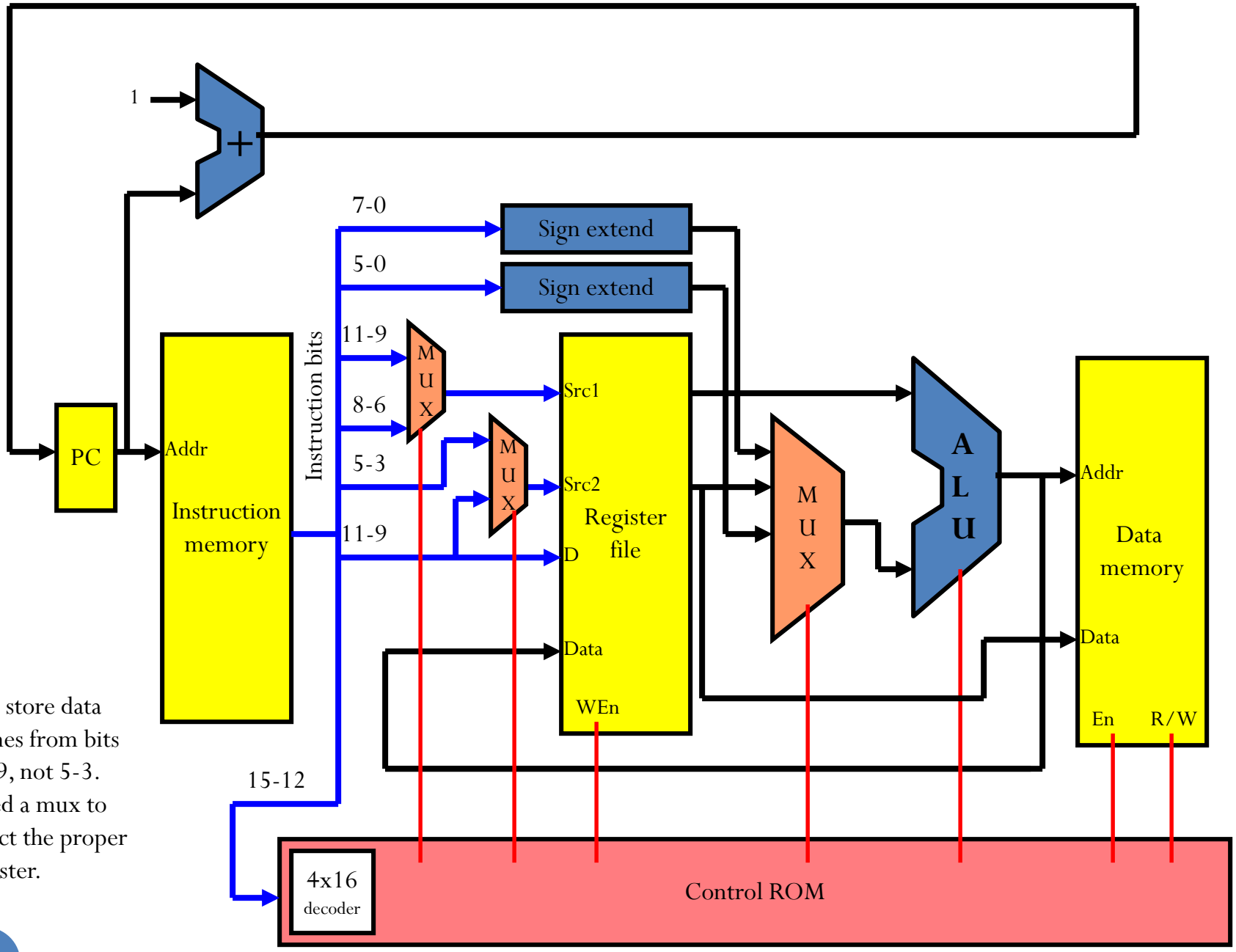
lli and lui require an 8-bit immediate → need a third input to the mux.



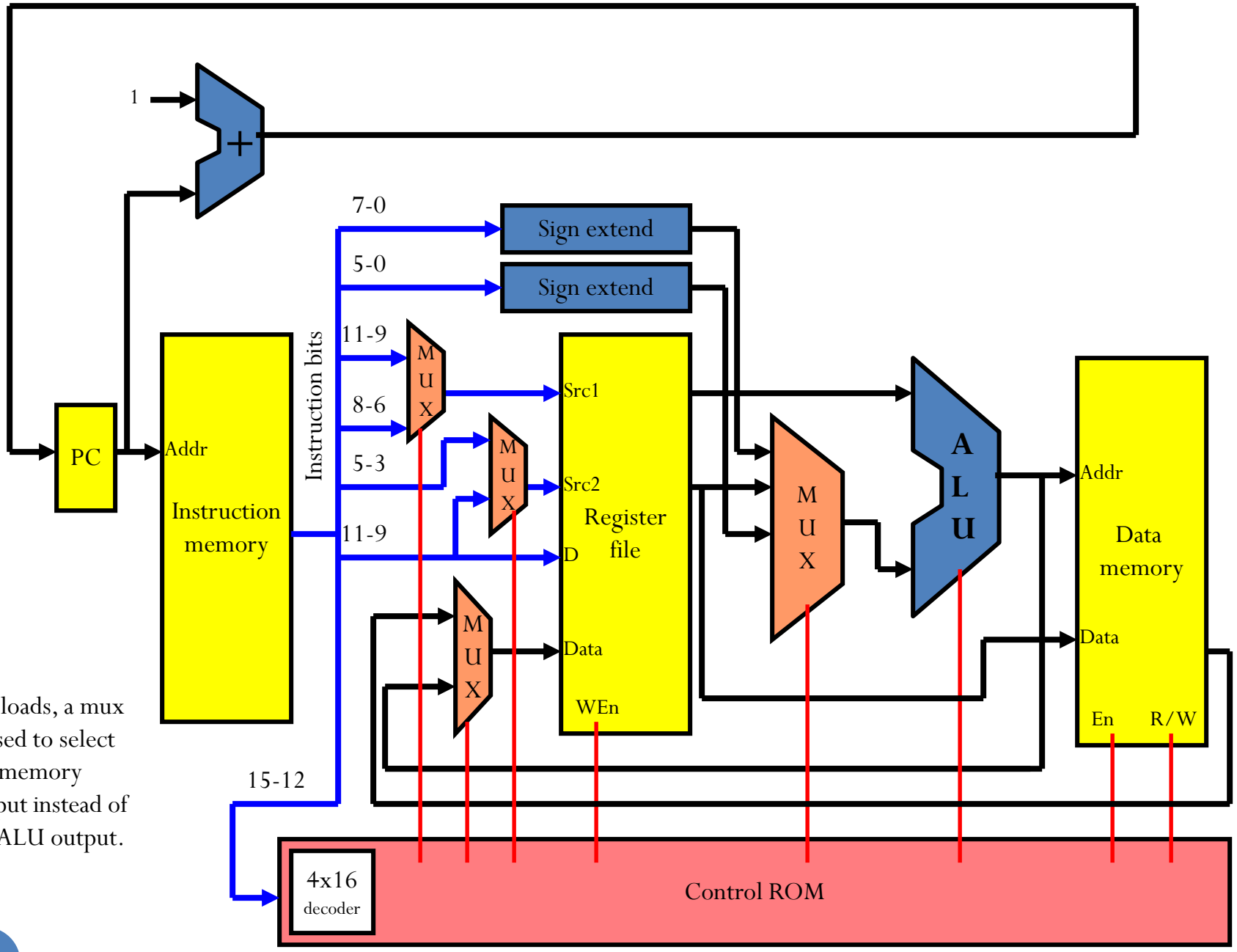
The lui instruction needs to read in Rd so it can supply the lower eight bits. A MUX is needed to select Rd instead of Rs_1.



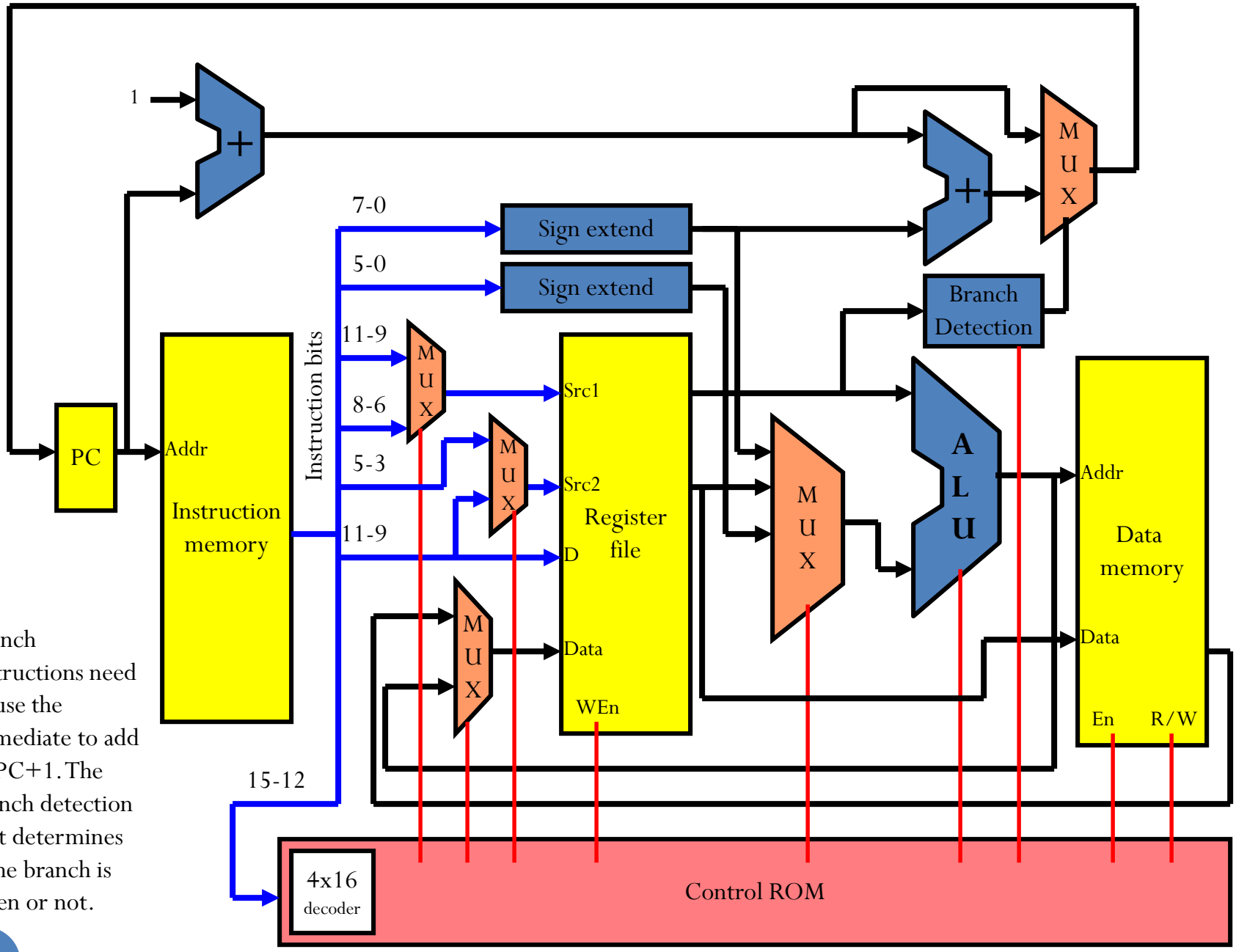
lw and sw access memory.



The store data comes from bits 11-9, not 5-3. Need a mux to select the proper register.



For loads, a mux is used to select the memory output instead of the ALU output.



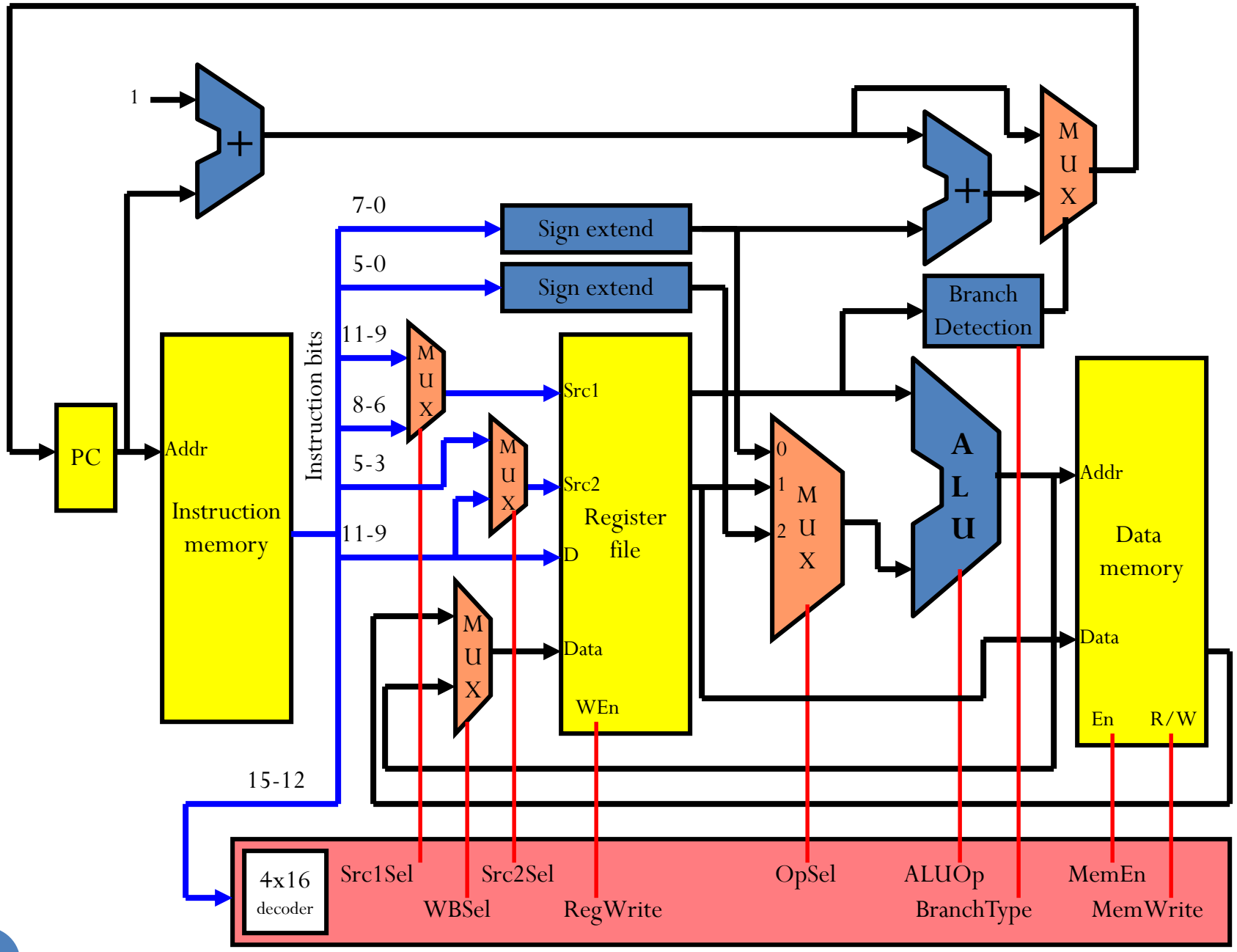
Branch instructions need to use the immediate to add to PC+1. The branch detection unit determines if the branch is taken or not.

Outline

- Introduction to Microarchitecture
- ANNA Datapath
- **ANNA Control**
- Performance
- Pipelining
- Additional Performance Optimizations

Control ROM

- The **control ROM** is a memory that contains an entry for each instruction.
  - Each entry contains the values for the control lines.
- Uses a 4 to 16 decoder to use the opcode to select the proper ROM entry.
- Control lines where the value does not matter are called don't cares (marked with an X).
  - The ROM can use any value.


1
+
+
M
U
X
7-0
5-0
Sign extend
Sign extend
Branch
Detection
11-9
8-6
5-3
11-9
M
U
X
M
U
X
M
U
X
Instruction bits
PC
Addr
Instruction
memory
Src1
Src2
D
Register
file
Data
WEn
0
1
2
M
U
X
A
L
U
Addr
Data
memory
Data
En
R/W
15-12
4x16
decoder
Src1Sel
WBSel
Src2Sel
RegWrite
OpSel
ALUOp
BranchType
MemEn
MemWrite

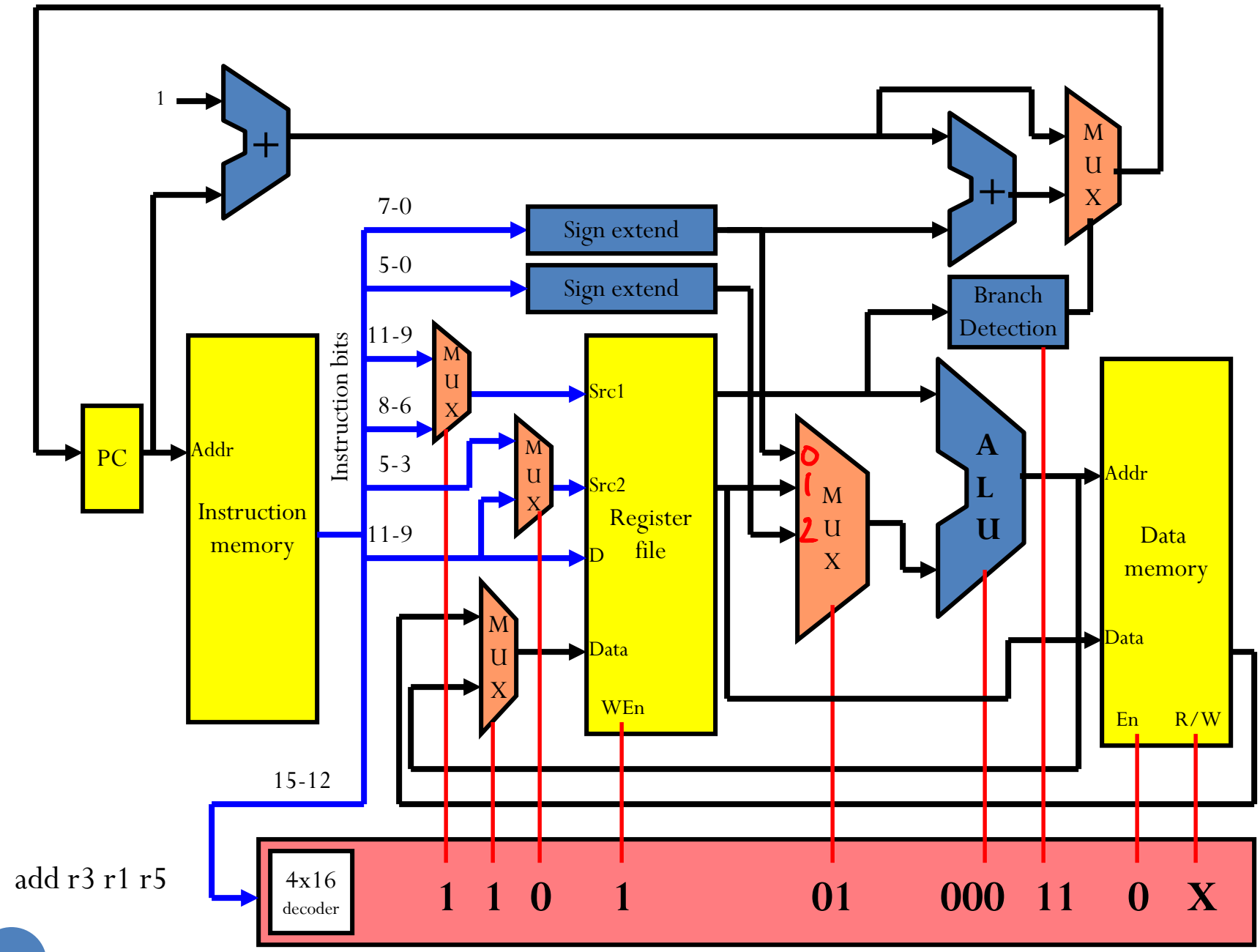
Control Lines in ANNA

- *Src1Sel*: Selects register to read for src1.
- *Src2Sel*: Selects register to read for src2.
- *WBSel*: Selects where to get data to write back to the register file.
- *RegWrite*: Set if instruction writes to register file.
- *OpSel*: Selects the operand that is used as the second input to the ALU.
- *ALUOp*: Tells the ALU what operation to perform.
- *MemEn*: Set if data memory is enabled.
- *MemWrite*: Set if data memory needs to be written.

Control Lines in ANNA

- For 2 input muxes:
  - 0: top input
  - 1: bottom input
- For the *OpSel* 3 input mux:
  - 0 (00): top input
  - 1 (01): middle input
  - 2 (10): bottom input
  - 3 (11): not used
- *ALUOp*:
  - Arithmetic/logic: three lower opcode bits
  - `addi/lw/sw`: use add (000)
- *BranchType*:
  - 00: `bez`
  - 01: `bgz`
  - 11: no branch
- For the enable lines:
  - 0: disabled
  - 1: enabled
- *MemWrite*:
  - 0: read
  - 1: write

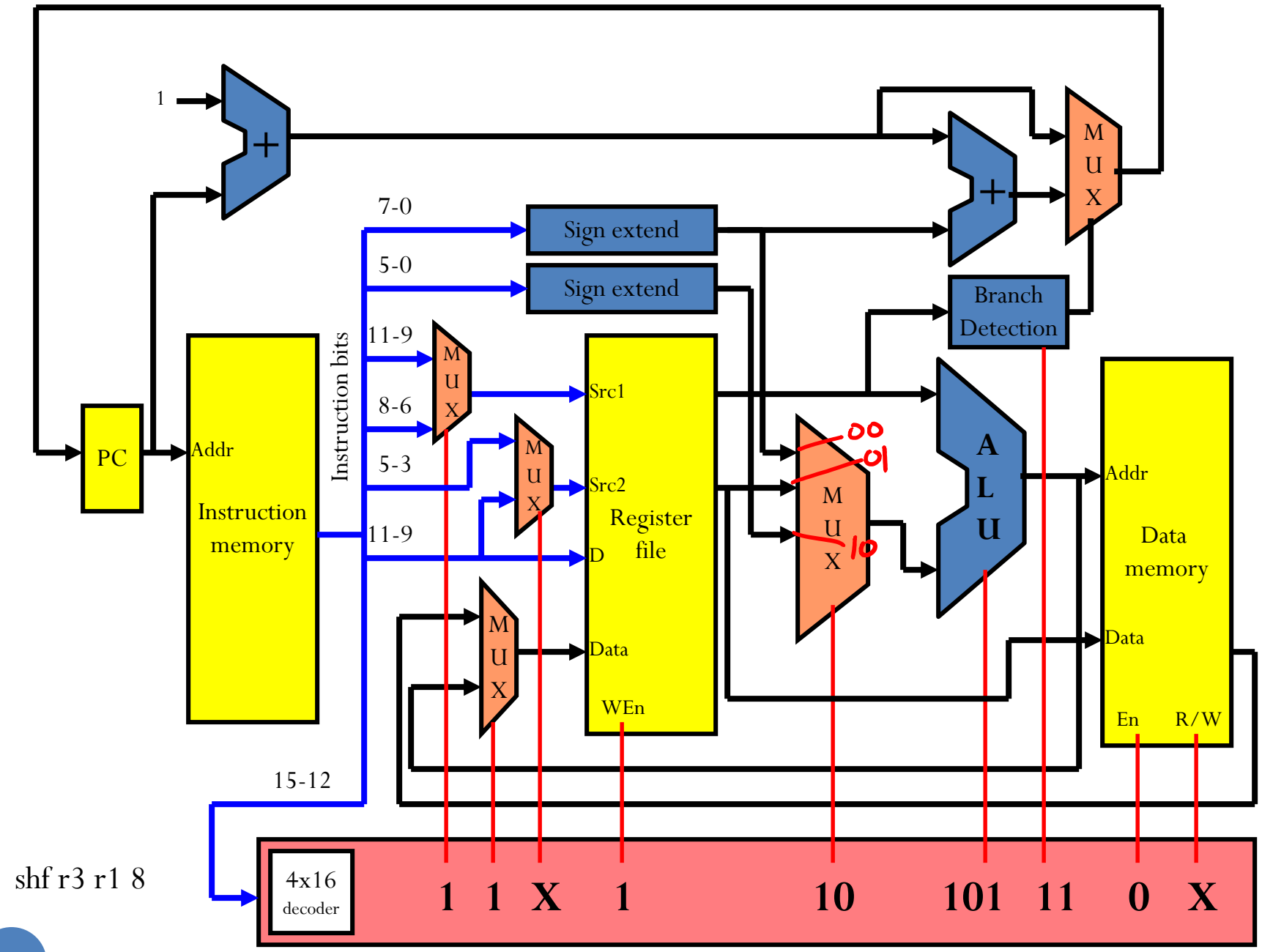
Opcode	*Src1 Sel*	*WB Sel*	*Src2 Sel*	*Reg Write*	*OpSel*	*ALUOp*	*Branch Type*	*MemEn*	*Mem Write*
add									
sub									
and									
or									
not									
shf									
addi									
lli									
lui									
lw									
sw									
bez									
bgz									


1
+
+
M
U
X
7-0
5-0
Sign extend
Sign extend
Branch
Detection
11-9
8-6
5-3
11-9
M
U
X
M
U
X
M
U
X
M
U
X
Instruction bits
PC
Addr
Instruction
memory
Src1
Src2
D
Register
file
Data
WEn
0
1
2
M
U
X
A
L
U
Addr
Data
memory
Data
En
R/W
15-12
add r3 r1 r5
4x16
decoder
1 1 0 1 01 000 11 0 X

Opcode	*Src1 Sel*	*WB Sel*	*Src2 Sel*	*Reg Write*	*OpSel*	*ALUOp*	*Branch Type*	*MemEn*	*Mem Write*
add	**1**	**1**	**0**	**1**	**01**	**000**	**11**	**0**	**X**
sub									
and									
or									
not									
shf									
addi									
lli									
lui									
lw									
sw									
bez									
bgz									

Opcode	*Src1 Sel*	*WB Sel*	*Src2 Sel*	*Reg Write*	*OpSel*	*ALUOp*	*Branch Type*	*MemEn*	*Mem Write*
add	1	1	0	1	01	000	11	0	X
sub	1	1	0	1	01	**001**	11	0	X
and	1	1	0	1	01	**010**	11	0	X
or	1	1	0	1	01	**011**	11	0	X
not									
shf									
addi									
lli									
lui									
lw									
sw									
bez									
bgz									

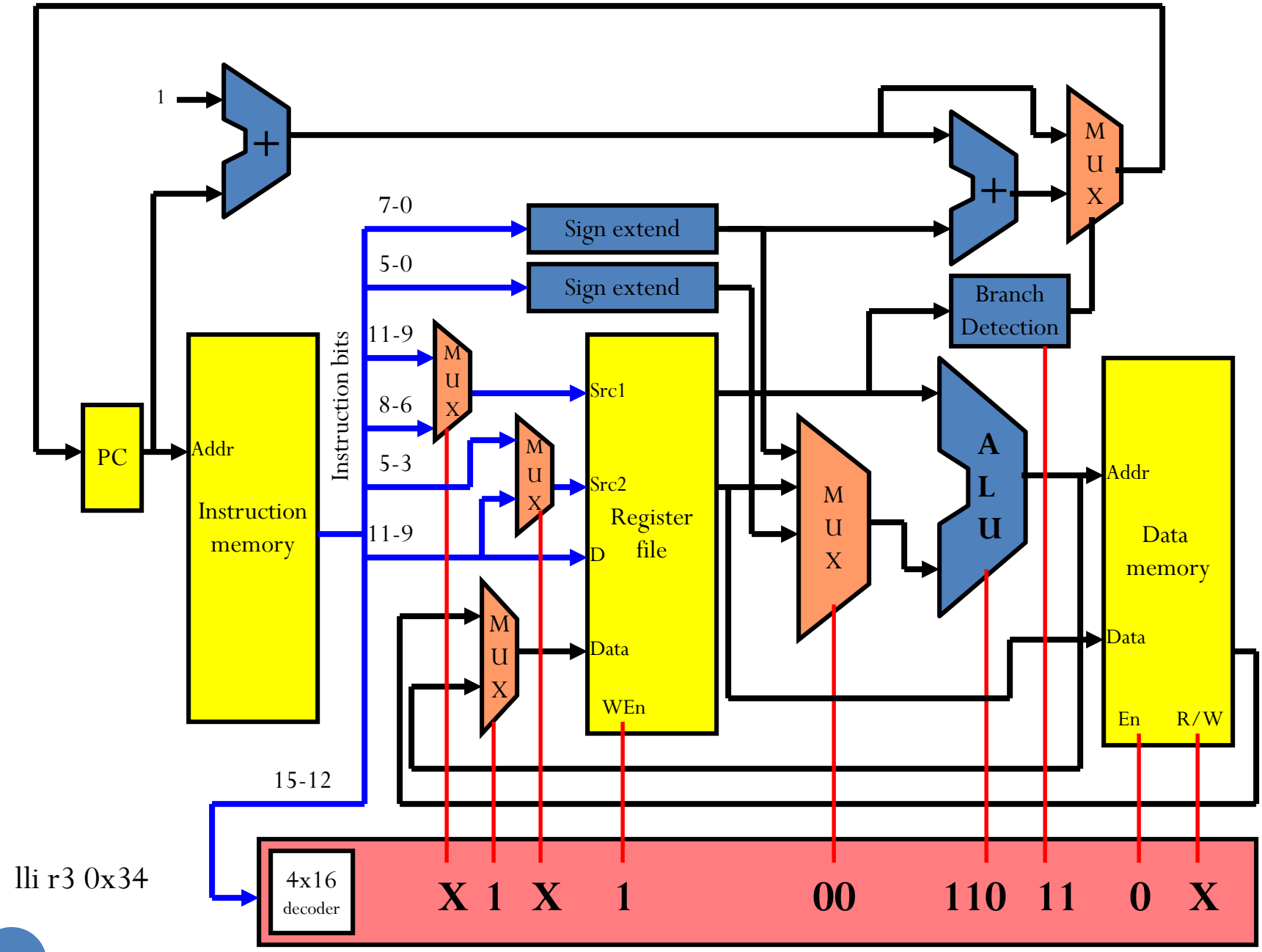
Opcode	*Src1 Sel*	*WB Sel*	*Src2 Sel*	*Reg Write*	*OpSel*	*ALUOp*	*Branch Type*	*MemEn*	*Mem Write*
add 0000	1	1	0	1	01	000	11	0	X
sub 0001	1	1	0	1	01	001	11	0	X
and	1	1	0	1	01	010	11	0	X
or	1	1	0	1	01	011	11	0	X
not	1	1	**X**	1	**X**	**100**	11	0	X
shf									
addi									
lli									
lui									
lw									
sw									
bez									
bgz									


1
+
+
M
U
X
7-0
5-0
Sign extend
Sign extend
Branch
Detection
11-9
8-6
5-3
11-9
M
U
X
M
U
X
M
U
X
M
U
X
00
01
10
PC
Addr
Instruction
memory
Instruction bits
Src1
Src2
Register
file
D
Data
WEn
A
L
U
Addr
Data
memory
Data
En
R/W
15-12
4x16
decoder
1 1 X 1 10 101 11 0 X

shf r3 r1 8

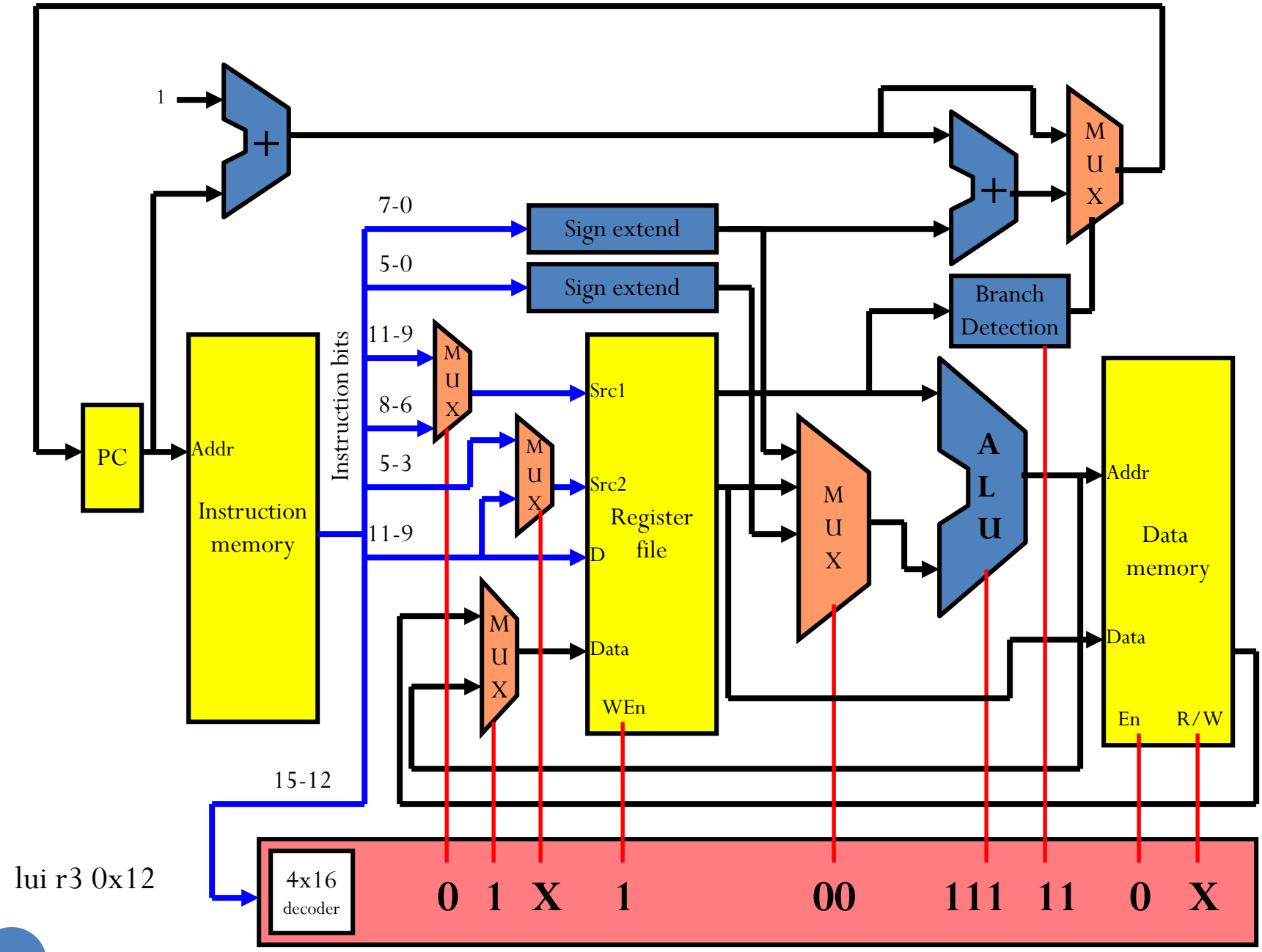
Opcode	*Src1 Sel*	*WB Sel*	*Src2 Sel*	*Reg Write*	*OpSel*	*ALUOp*	*Branch Type*	*MemEn*	*Mem Write*
add	1	1	0	1	01	000	11	0	X
sub	1	1	0	1	01	001	11	0	X
and	1	1	0	1	01	010	11	0	X
or	1	1	0	1	01	011	11	0	X
not	1	1	X	1	X	100	11	0	X
shf	**1**	**1**	**X**	**1**	**10**	**101**	**11**	**0**	**X**
addi									
lli									
lui									
lw									
sw									
bez									
bgz									

Opcode	*Src1 Sel*	*WB Sel*	*Src2 Sel*	*Reg Write*	*OpSel*	*ALUOp*	*Branch Type*	*MemEn*	*Mem Write*
add	1	1	0	1	01	000	11	0	X
sub	1	1	0	1	01	001	11	0	X
and	1	1	0	1	01	010	11	0	X
or	1	1	0	1	01	011	11	0	X
not	1	1	X	1	X	100	11	0	X
shf	1	1	X	1	10	101	11	0	X
addi	1	1	X	1	10	**000**	11	0	X
lli									
lui									
lw									
sw									
bez									
bgz									


1
+
+
M
U
X
7-0
Sign extend
5-0
Sign extend
Branch
Detection
11-9
M
U
X
8-6
PC
Addr
Instruction
memory
Instruction bits
5-3
M
U
X
11-9
Src1
Src2
Register
file
D
Data
WEn
M
U
X
M
U
X
A
L
U
Addr
Data
memory
Data
En
R/W
15-12
4x16
decoder
X 1 X 1 00 110 11 0 X

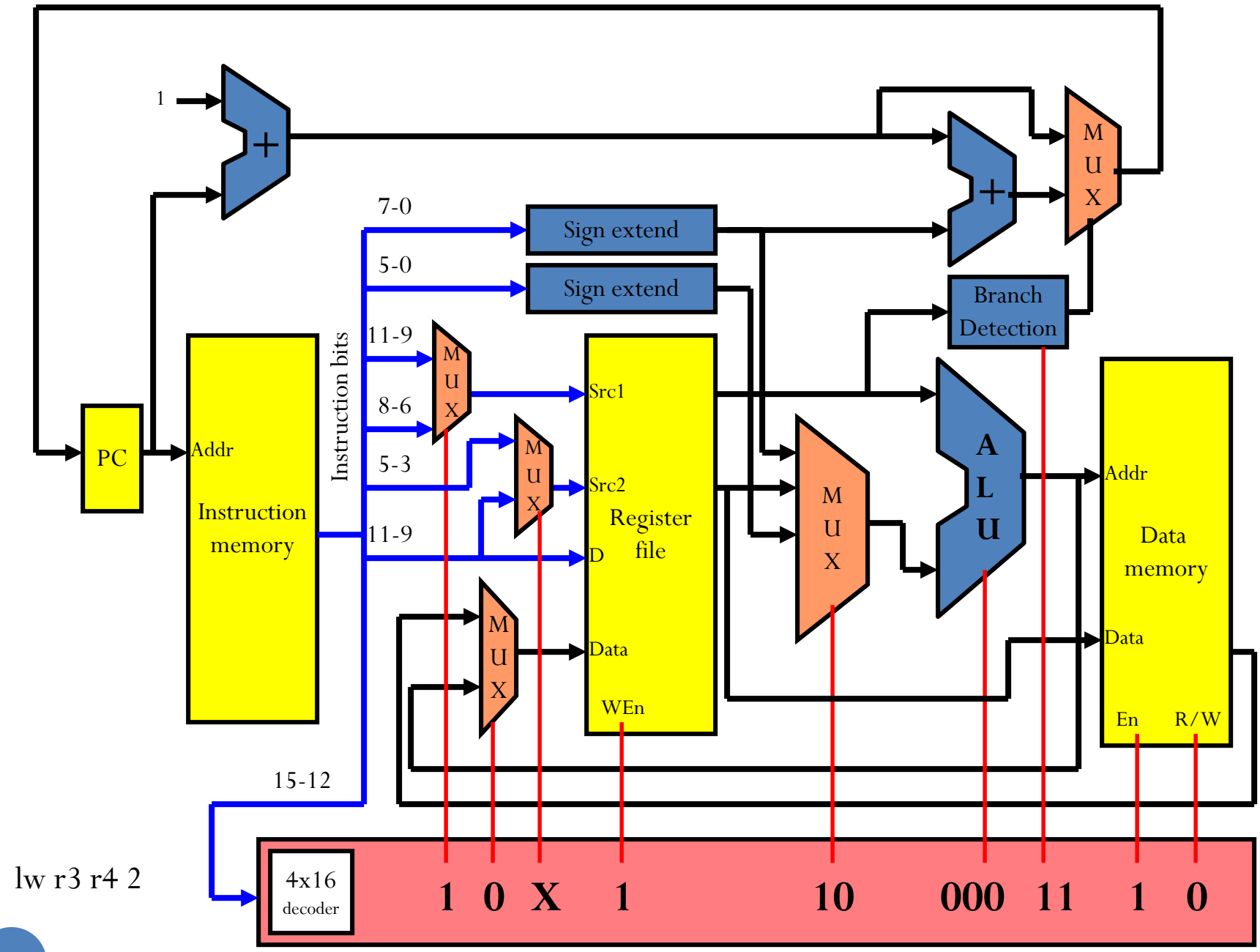
lli r3 0x34

Opcode	*Src1 Sel*	*WB Sel*	*Src2 Sel*	*Reg Write*	*OpSel*	*ALUOp*	*Branch Type*	*MemEn*	*Mem Write*
add	1	1	0	1	01	000	11	0	X
sub	1	1	0	1	01	001	11	0	X
and	1	1	0	1	01	010	11	0	X
or	1	1	0	1	01	011	11	0	X
not	1	1	X	1	X	100	11	0	X
shf	1	1	X	1	10	101	11	0	X
addi	1	1	X	1	10	000	11	0	X
lli	**X**	**1**	**X**	**1**	**00**	**110**	**11**	**0**	**X**
lui									
lw									
sw									
bez									
bgz									


1
+
+
M
U
X
7-0
Sign extend
5-0
Sign extend
Branch
Detection
11-9
M
U
X
8-6
PC
Addr
Instruction
memory
Instruction bits
5-3
M
U
X
11-9
Src1
Src2
Register
file
D
Data
WEn
M
U
X
M
U
X
A
L
U
Addr
Data
memory
Data
En
R/W
15-12
4x16
decoder
0
1
X
1
00
111
11
0
X

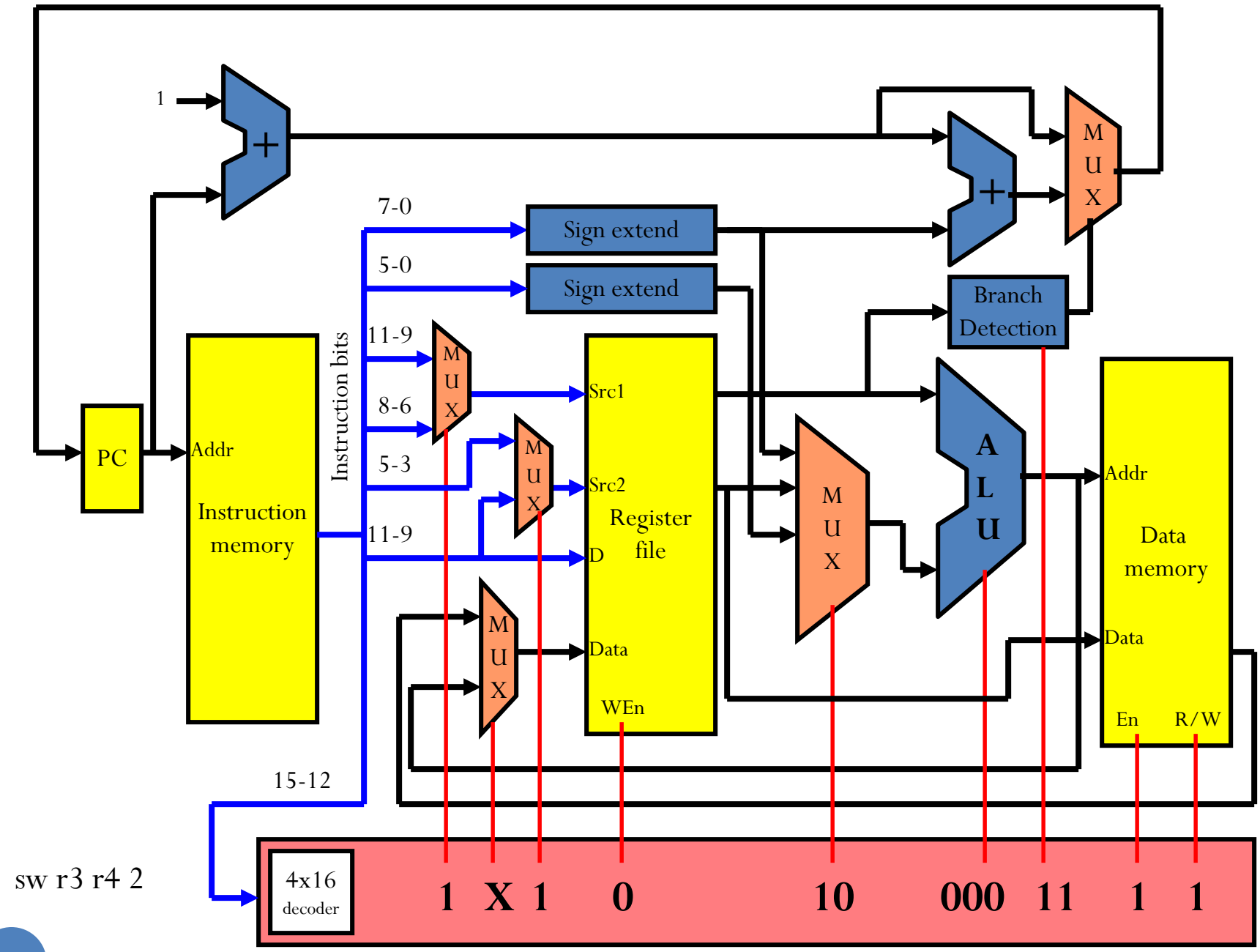
lui r3 0x12

Opcode	*Src1 Sel*	*WB Sel*	*Src2 Sel*	*Reg Write*	*OpSel*	*ALUOp*	*Branch Type*	*MemEn*	*Mem Write*
add	1	1	0	1	01	000	11	0	X
sub	1	1	0	1	01	001	11	0	X
and	1	1	0	1	01	010	11	0	X
or	1	1	0	1	01	011	11	0	X
not	1	1	X	1	X	100	11	0	X
shf	1	1	X	1	10	101	11	0	X
addi	1	1	X	1	10	000	11	0	X
lli	X	1	X	1	00	110	11	0	X
lui	**0**	**1**	**X**	**1**	**00**	**111**	**11**	**0**	**X**
lw									
sw									
bez									
bgz									


1
+
+
M
U
X
7-0
Sign extend
5-0
Sign extend
Branch
Detection
11-9
M
U
X
8-6
PC
Addr
Instruction
memory
Instruction bits
5-3
M
U
X
11-9
Src1
Src2
Register
file
D
Data
WEn
M
U
X
M
U
X
A
L
U
Addr
Data
memory
Data
En
R/W
15-12
4x16
decoder
1 0 X 1 10 000 11 1 0

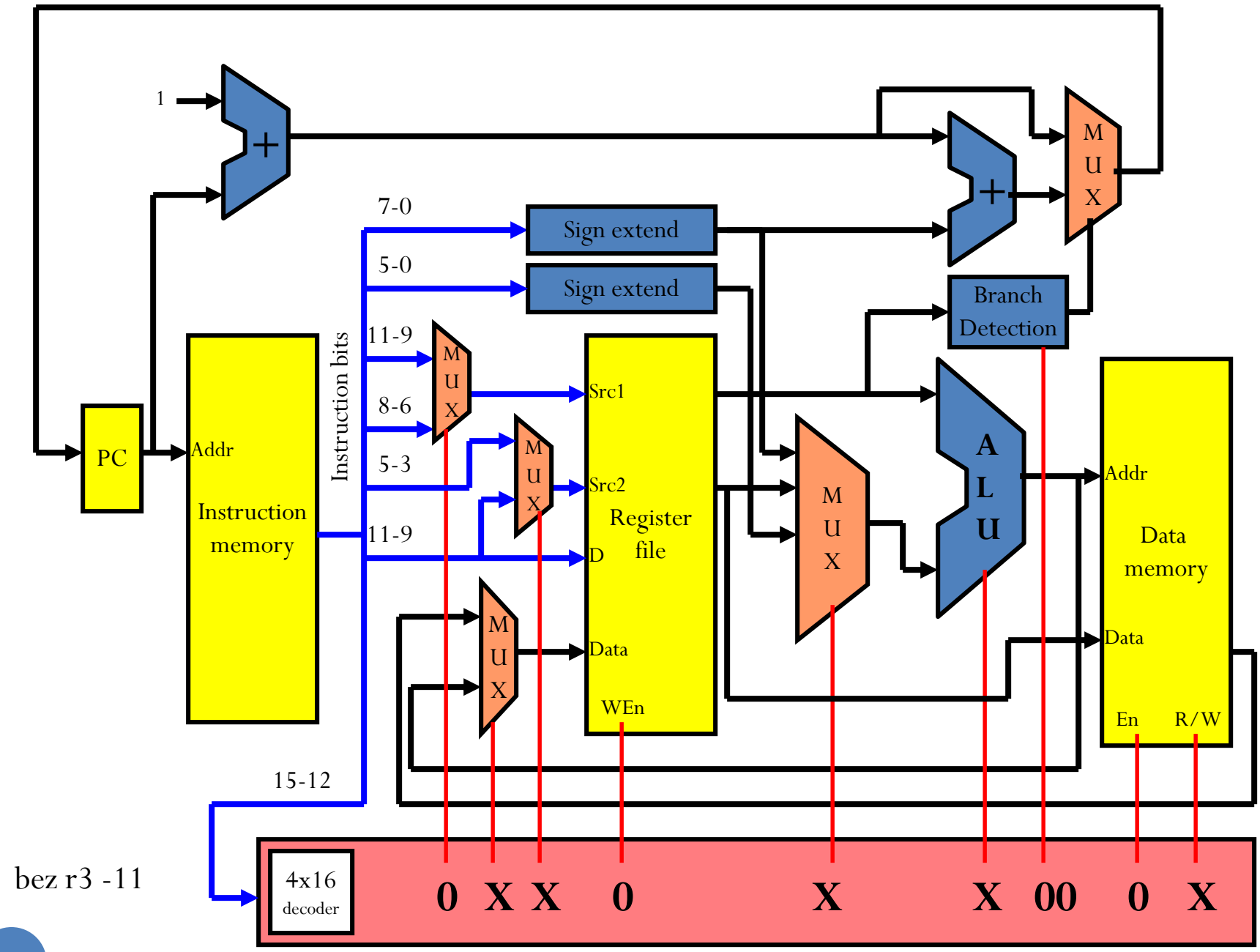
lw r3 r4 2

Opcode	*Src1 Sel*	*WB Sel*	*Src2 Sel*	*Reg Write*	*OpSel*	*ALUOp*	*Branch Type*	*MemEn*	*Mem Write*
add	1	1	0	1	01	000	11	0	X
sub	1	1	0	1	01	001	11	0	X
and	1	1	0	1	01	010	11	0	X
or	1	1	0	1	01	011	11	0	X
not	1	1	X	1	X	100	11	0	X
shf	1	1	X	1	10	101	11	0	X
addi	1	1	X	1	10	000	11	0	X
lli	X	1	X	1	00	110	11	0	X
lui	0	1	X	1	00	111	11	0	X
lw	**1**	**0**	**X**	**1**	**10**	**000**	**11**	**1**	**0**
sw									
bez									
bgz									


1
+
+
M
U
X
7-0
Sign extend
5-0
Sign extend
Branch
Detection
11-9
M
U
X
8-6
PC
Addr
Instruction
memory
Instruction bits
5-3
M
U
X
11-9
Src1
Src2
Register
file
D
Data
WEn
M
U
X
M
U
X
A
L
U
Addr
Data
memory
Data
En
R/W
15-12
4x16
decoder
1
X
1
0
10
000
11
1
1

sw r3 r4 2

Opcode	*Src1 Sel*	*WB Sel*	*Src2 Sel*	*Reg Write*	*OpSel*	*ALUOp*	*Branch Type*	*MemEn*	*Mem Write*
add	1	1	0	1	01	000	11	0	X
sub	1	1	0	1	01	001	11	0	X
and	1	1	0	1	01	010	11	0	X
or	1	1	0	1	01	011	11	0	X
not	1	1	X	1	X	100	11	0	X
shf	1	1	X	1	10	101	11	0	X
addi	1	1	X	1	10	000	11	0	X
lli	X	1	X	1	00	110	11	0	X
lui	0	1	X	1	00	111	11	0	X
lw	1	0	X	1	10	000	11	1	0
sw	**1**	**X**	**1**	**0**	**10**	**000**	**11**	**1**	**1**
bez									
bgz									


1
+
7-0
5-0
Sign extend
Sign extend
+
M
U
X
Branch
Detection
11-9
8-6
5-3
11-9
M
U
X
M
U
X
M
U
X
M
U
X
PC
Addr
Instruction
memory
Instruction bits
Src1
Src2
D
Register
file
Data
WEn
A
L
U
Addr
Data
memory
Data
En
R/W
15-12
4x16
decoder
0 X X 0 X X 00 0 X
bez r3 -11

Opcode	*Src1 Sel*	*WB Sel*	*Src2 Sel*	*Reg Write*	*OpSel*	*ALUOp*	*Branch Type*	*MemEn*	*Mem Write*
add	1	1	0	1	01	000	11	0	X
sub	1	1	0	1	01	001	11	0	X
and	1	1	0	1	01	010	11	0	X
or	1	1	0	1	01	011	11	0	X
not	1	1	X	1	X	100	11	0	X
shf	1	1	X	1	10	101	11	0	X
addi	1	1	X	1	10	000	11	0	X
lli	X	1	X	1	00	110	11	0	X
lui	0	1	X	1	00	111	11	0	X
lw	1	0	X	1	10	000	11	1	0
sw	1	X	1	0	10	000	11	1	1
bez	**0**	**X**	**X**	**0**	**X**	**X**	**00**	**0**	**X**
bgz									

Opcode	*Src1 Sel*	*WB Sel*	*Src2 Sel*	*Reg Write*	*OpSel*	*ALUOp*	*Branch Type*	*MemEn*	*Mem Write*
add	1	1	0	1	01	000	11	0	X
sub	1	1	0	1	01	001	11	0	X
and	1	1	0	1	01	010	11	0	X
or	1	1	0	1	01	011	11	0	X
not	1	1	X	1	X	100	11	0	X
shf	1	1	X	1	10	101	11	0	X
addi	1	1	X	1	10	000	11	0	X
lli	X	1	X	1	00	110	11	0	X
lui	0	1	X	1	00	111	11	0	X
lw	1	0	X	1	10	000	11	1	0
sw	1	X	1	0	10	000	11	1	1
bez	0	X	X	0	X	X	00	0	X
bgz	0	X	X	0	X	X	**01**	0	X

Outline

- Introduction to Microarchitecture
- ANNA Datapath
- ANNA Control
- **Performance**
- Pipelining
- Additional Performance Optimizations

Computer Performance

- What factors are important when considering performance of a computer system?

Average time it takes to process an instruction.

Memory access time

Memory load

Number of instructions

Temperature

Power consumption

Clock speed/frequency

I/O performance

Response time

CPU Time Equation

$$\text{CPU time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{avg. cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}} = \frac{\text{seconds}}{\text{program}}$$

CPI

Compare systems A + B

$$\frac{\text{time on } B}{\text{time on } A} = n$$

Misleading / Deceptive Statistic Practices

- Only providing clock frequency (often given in GHz).
- Using statistics such as MIPS (Millions of Instructions Per Second) / FLOPS (Floating Point Operations Per Second).
- Selective statistics: Citing only favorable results while omitting others.
- Using biased or irrelevant benchmarks.
- Citing only peak performance numbers while ignoring the average case.
- Vagueness in the use of words like "almost," "nearly," "more," and "less," in comparing performance data.

Measuring Performance

Measuring performance is hard!

- Hard to come up with a representative and realistic set of benchmarks.
- The compiler and how the program is written factor into the overall time.
- Dependencies on I/O devices, operating systems, libraries, etc.
- Measuring the CPU usage at a fine-grained level can be challenging.
- Different input sizes can drastically change the results.

Single Cycle Datapath

- CPI: 1 (every instruction takes 1 cycle)
- Cycle time (inverse of frequency) is set based on time for longest instruction
- Longest instruction is typically `lw`:
  - Read base address register
  - Compute effective address using ALU
  - Get value from memory
  - Write value from memory to register file
- Problem: All other instructions waste time.

Multiple Cycle Datapath

- Instructions can take multiple cycles to execute.
- Longer instructions take more cycles; shorter instructions take fewer cycles.
- Cycle time is shorter → less work per cycle
- Is it faster than single cycle?

Lower cycle time → decrease time

Higher CPI → increases time

Multiple Cycle vs. Single Cycle

- Loads and stores take longer than other instructions because of the additional memory access for data.
  - Single cycle: cycle time based on load instruction execution time.
  - Multiple cycle: may take longer than single cycle because the division into cycles may not be clean.
- Instructions with only register operands will typically be faster in a multiple cycle implementation.
  - Single cycle: will sit idle for some portion of the cycle
  - Multiple cycle: will take fewer cycles than load and store
- Is multiple cycle faster than single cycle?
  - depends on number of loads and stores

Most Frequently Executed Instructions (SPECInt Benchmarks)

1.	load	22%
2.	sub / compare	21%
3.	conditional branch	20%
4.	store	12%
5.	add / move	12%
6.	and	6%

Strikes: X X X

All other instruction types 1% or less per type

Class Problem

- Single cycle time: 45 ns
- Multiple cycle time: 10 ns
- Multiple cycle implementation:
  - `lw`: 5 cycles
  - `sw`: 4 cycles
  - all other instructions: 3 cycles
- Determine how long it would take to run the same 100 instruction program on each implementation.
  - Use the instruction frequencies from the previous slide.

Single cycle: $100\,\text{inst}\left(\frac{1\,\text{cycle}}{\text{inst}}\right)\left(\frac{45\,\text{ns}}{1\,\text{cycle}}\right)$

$= 4500\,\text{ns}$

Mult cycle: $100\,\text{inst}\left(\frac{3.56\,\text{cycles}}{\text{inst}}\right)\left(\frac{10\,\text{ns}}{1\,\text{cycle}}\right)$

$= 3560\,\text{ns}$

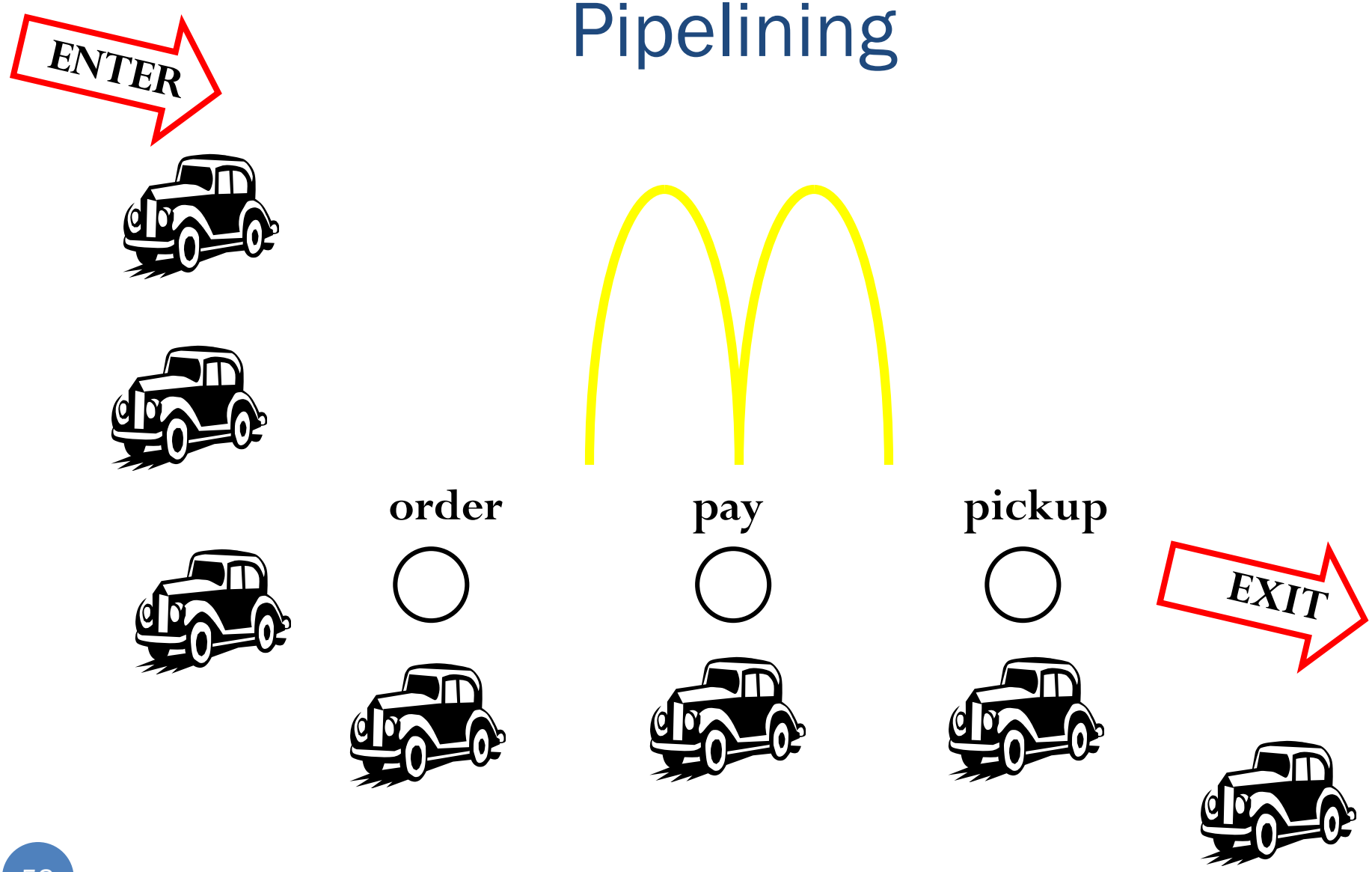
Multiple cycle CPI: $(0.22 \times 5) + (0.12 \times 4) + (0.66 \times 3)$

$= 3.56$

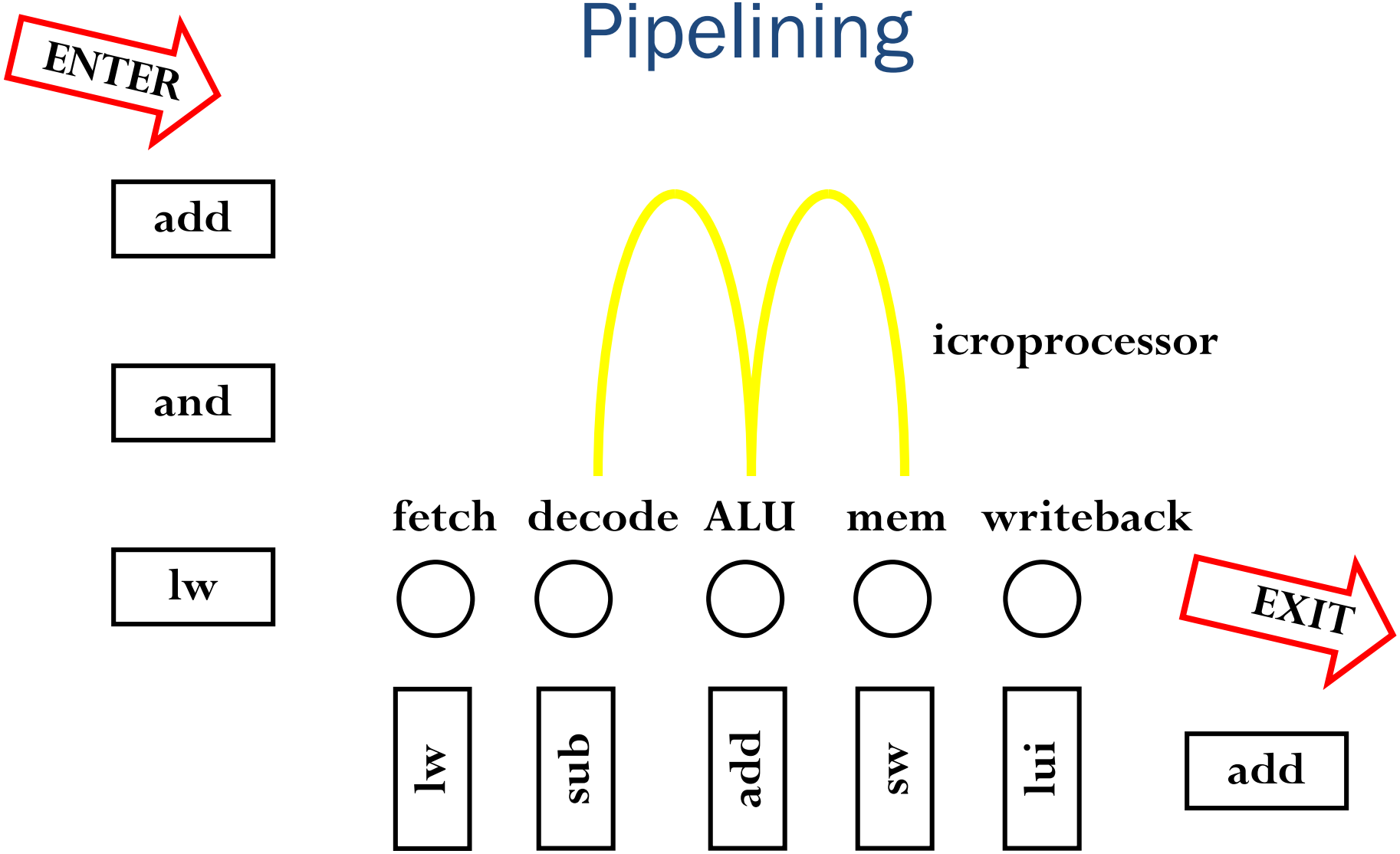
Outline

- Introduction to Microarchitecture
- ANNA Datapath
- ANNA Control
- Performance
- **Pipelining**
- Additional Performance Optimizations

Pipelining

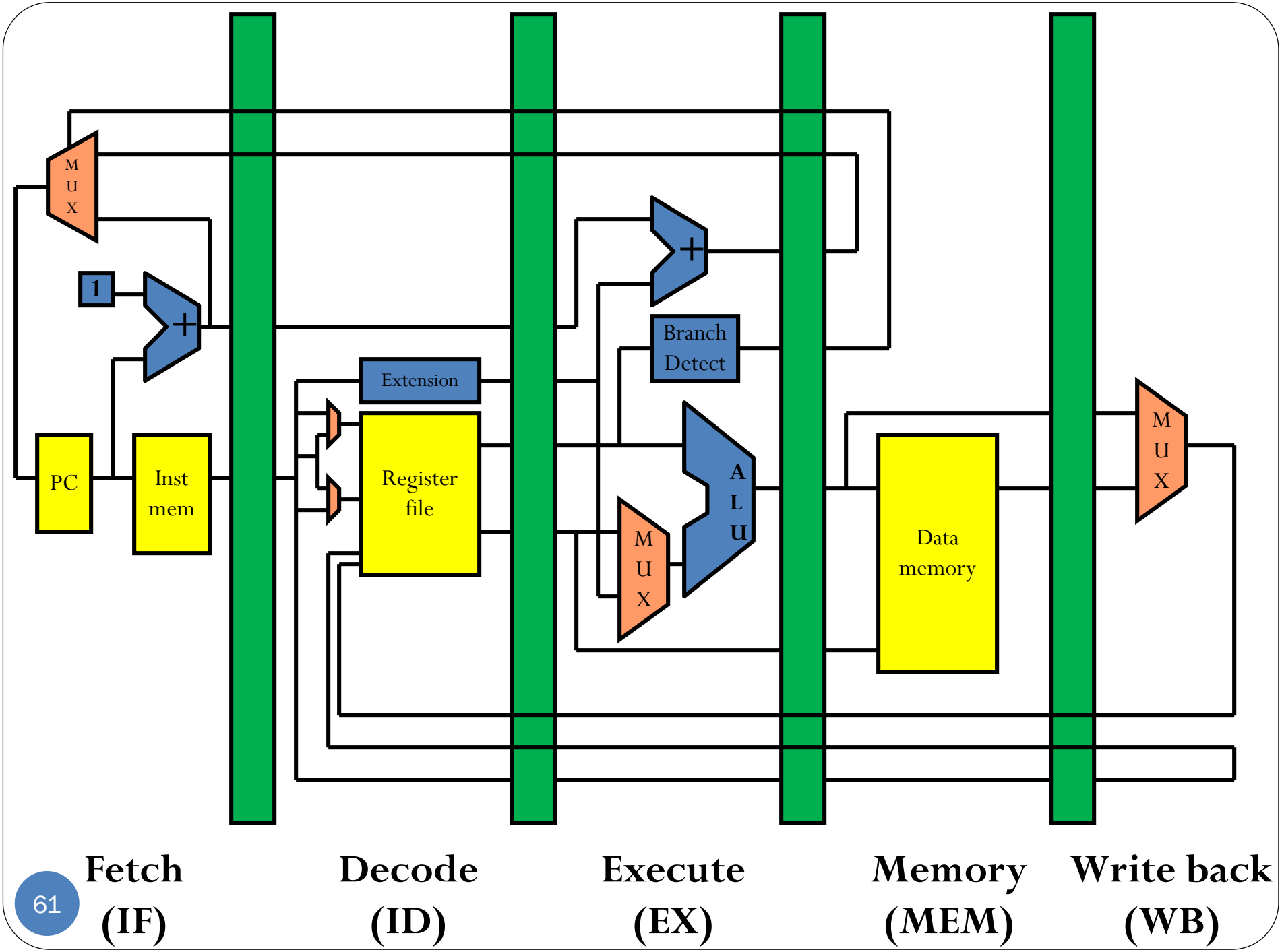


Pipelining


ENTER
add
and
lw
icroprocessor
fetch
decode
ALU
mem
writeback
EXIT
lw
sub
add
sw
lui
add

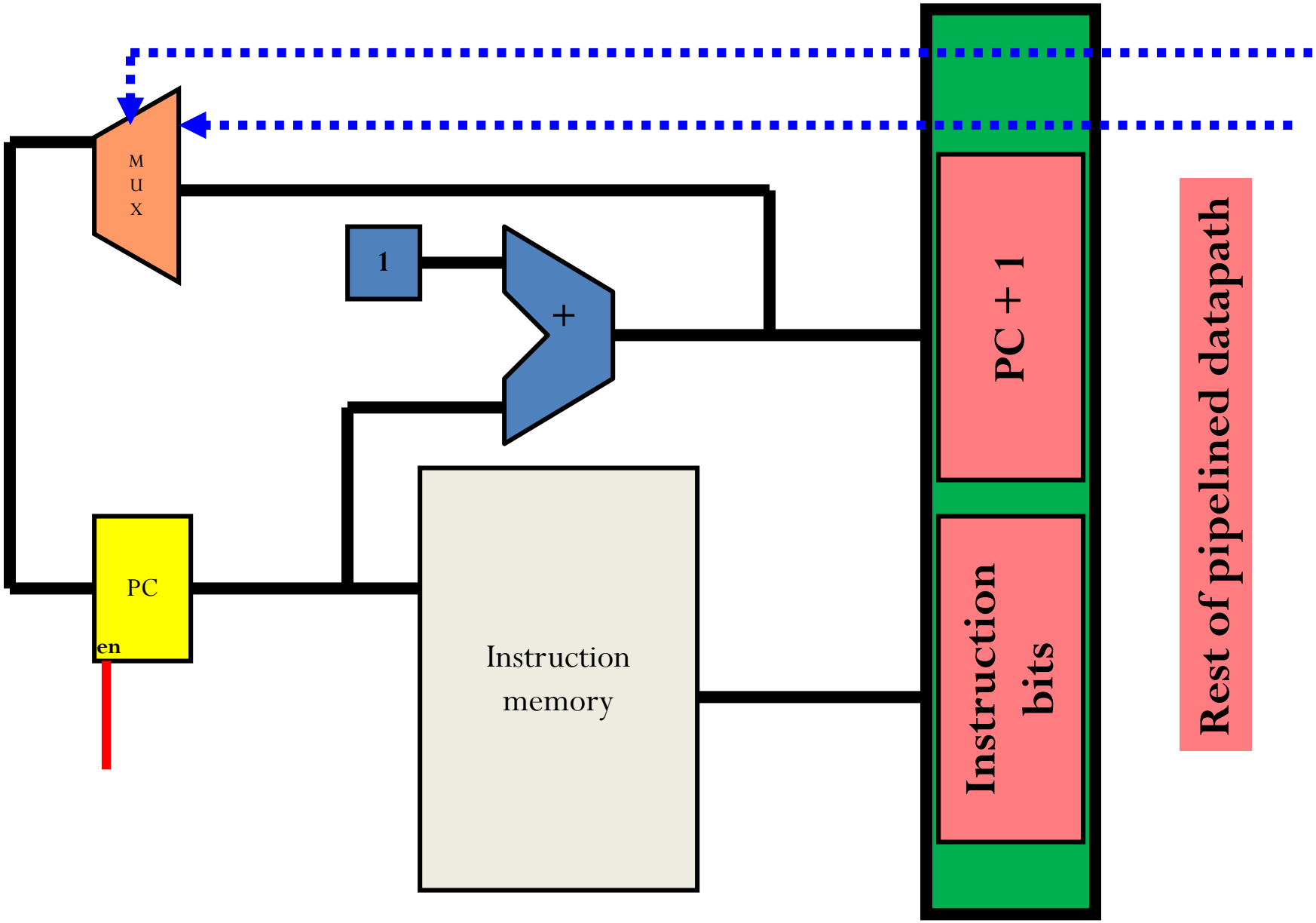
Pipelined Implementation of ANNA

- Break the execution of the instructions into cycles.
  - Similar to multiple cycle datapath.
- Design a separate datapath stage for the execution performed during each cycle.
- Instructions can overlap execution.
  - Different instructions can be in different cycles.
- Build pipeline registers to communicate between different stages.


M
U
X
1
+
PC
Inst
mem
Extension
Register
file
+
Branch
Detect
A
L
U
M
U
X
Data
memory
M
U
X
Fetch
(IF)
Decode
(ID)
Execute
(EX)
Memory
(MEM)
Write back
(WB)

Stage 1: Fetch (IF)

- Design a datapath that can fetch an instruction from memory every cycle.
  - Use PC to index instruction memory.
  - Increment the PC (assume no branches for now).
- Write everything needed to complete execution to the pipeline register (IF/ID).
  - The next stage will read this pipeline register.
  - Note that pipeline register must be edge triggered.

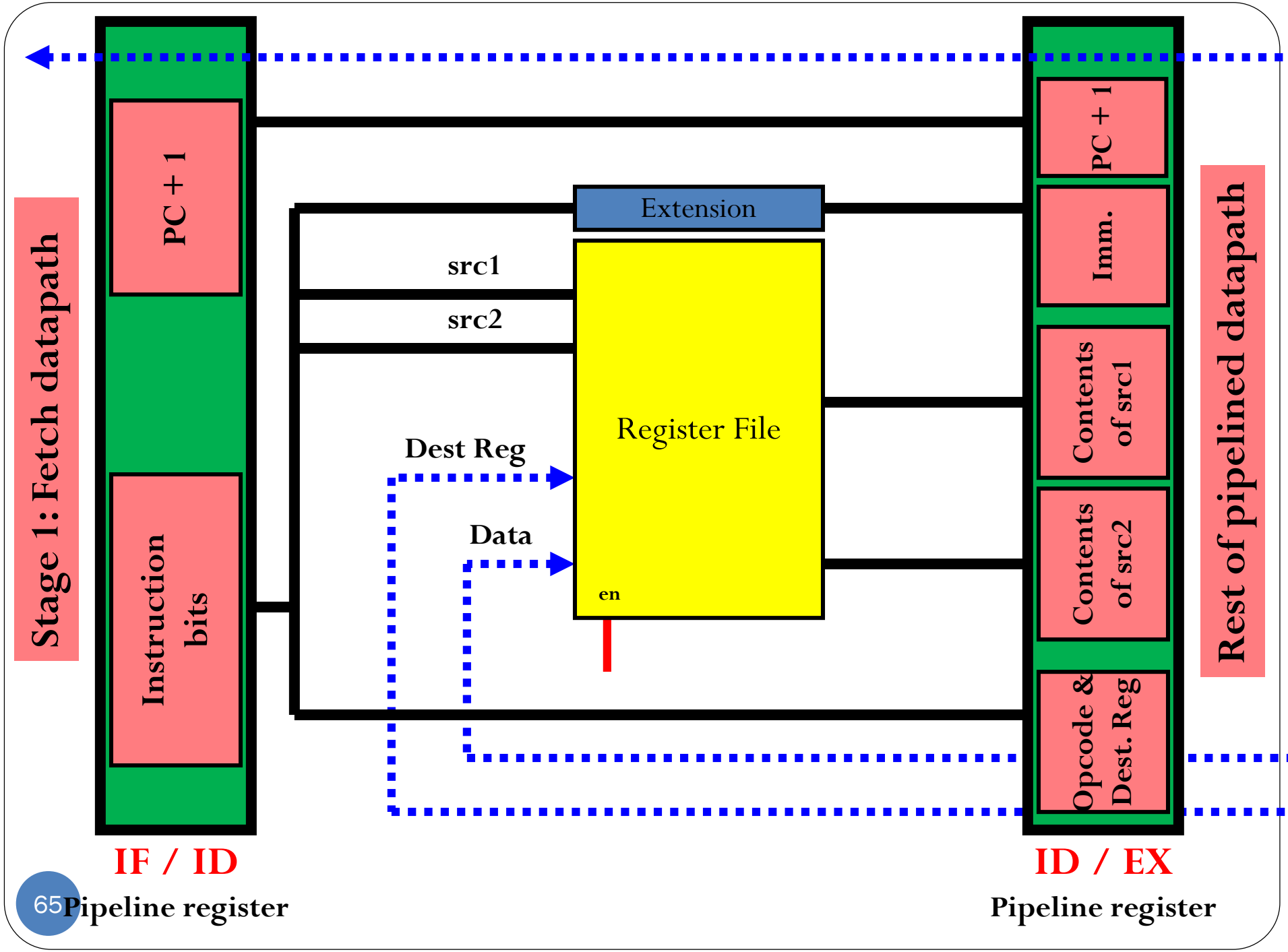


IF / ID

Pipeline register

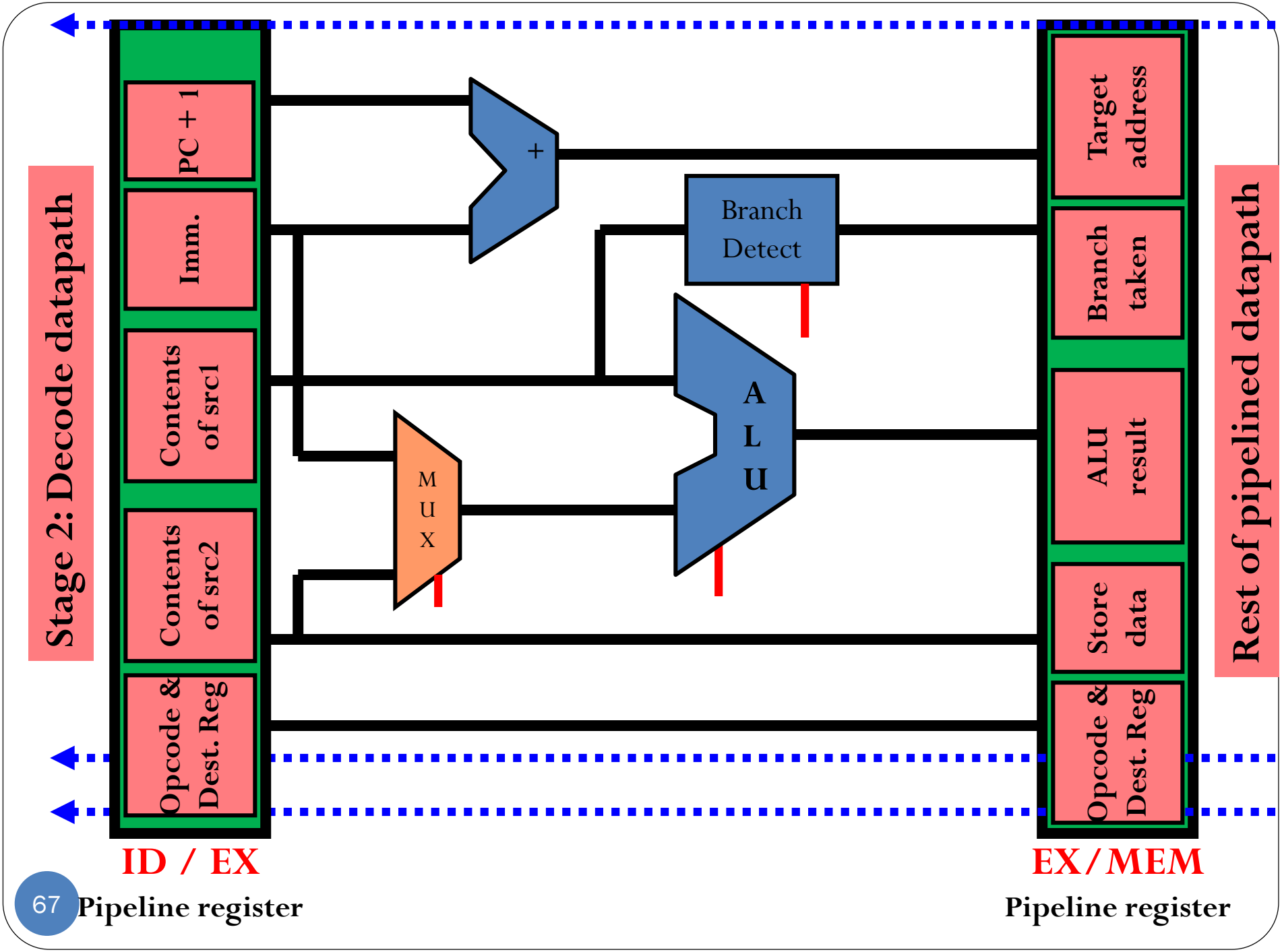
Stage 2: Decode (ID)

- Reads the IF/ID pipeline register and decodes instruction.
  - Decode is easy: pass opcode to later stages and let them figure out what to do.
- Reads register file.
- Determine immediate (based on instruction).
- Write everything needed to complete execution to the pipeline register (ID/EX):
  - register values
  - immediate
  - opcode and destination register number (could pass entire instruction)
  - PC+1 (even though decode doesn't use it)


Stage 1: Fetch datapath
PC + 1
Instruction bits
Extension
src1
src2
Register File
Dest Reg
Data
en
PC + 1
Imm.
Contents of src1
Contents of src2
Opcode & Dest. Reg
Rest of pipelined datapath
IF / ID
Pipeline register
ID / EX
Pipeline register

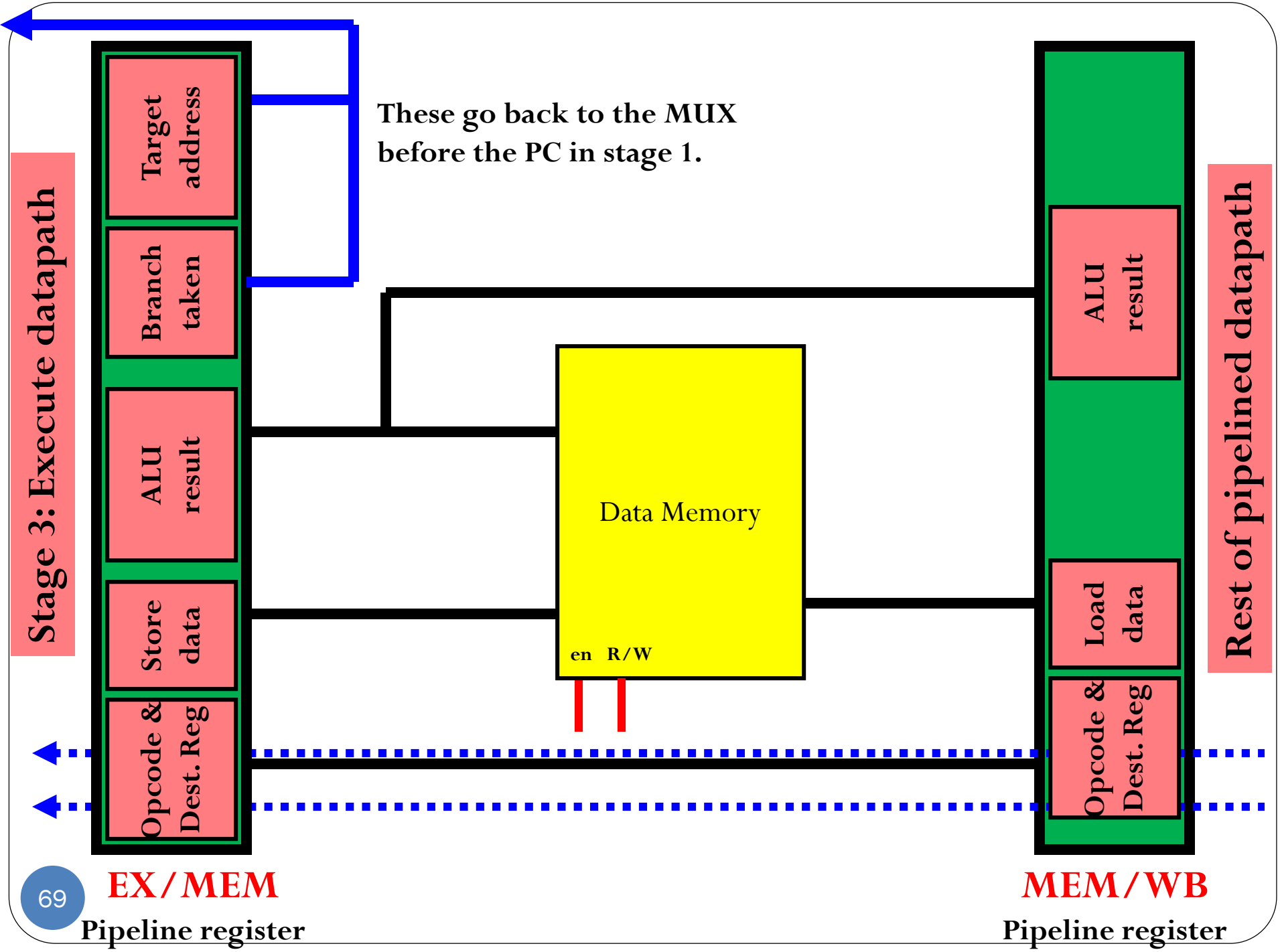
Stage 3: Execute (EX)

- Performs the proper ALU operation for the instruction specified and the values present in the ID/EX pipeline register.
- Calculates target address in case this is a branch.
  - Also determines if branch is taken or not.
- Write everything needed to complete execution to the pipeline register (EX/MEM):
  - ALU result
  - contents of register src2 (store data)
  - target address
  - result from branch detection unit
  - instruction bits for opcode and destination register


Stage 2: Decode datapath
PC + 1
Imm.
Contents of src1
Contents of src2
Opcode & Dest. Reg
+
Branch Detect
M U X
A L U
Target address
Branch taken
ALU result
Store data
Opcode & Dest. Reg
Rest of pipelined datapath
ID / EX
Pipeline register
EX/MEM
Pipeline register

Stage 4: Memory Operation (MEM)

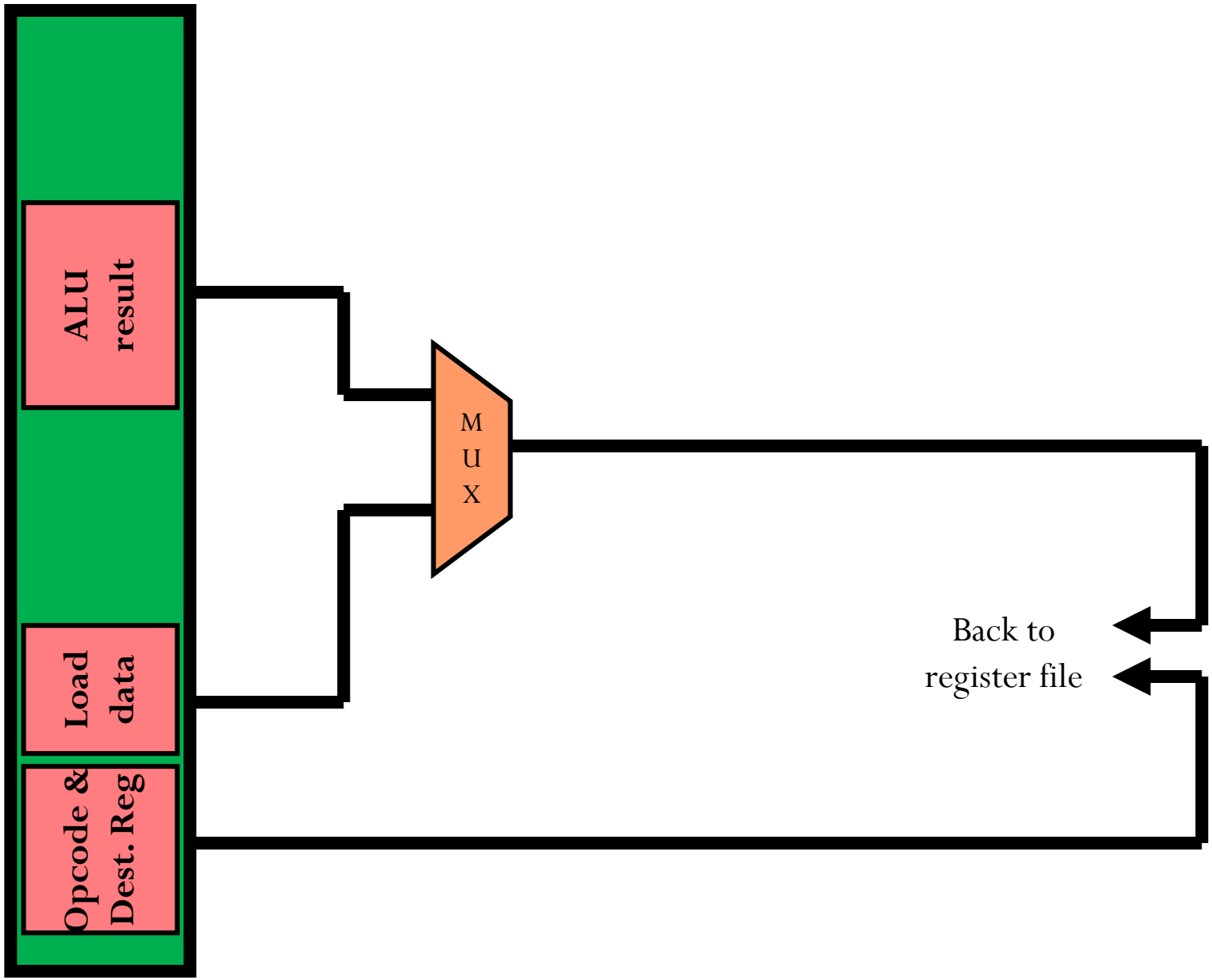
- Performs the proper memory operation based on the instruction in EX/MEM pipeline register.
- Also resolves taken branches (more later).
- Nothing is done for ALU operations.
- Write everything needed to complete execution to the pipeline register (MEM/WB)
  - ALU result
  - data from memory (result of load)
  - instruction bits for opcode and destination register


Stage 3: Execute datapath
Target address
Branch taken
ALU result
Store data
Opcode & Dest. Reg
These go back to the MUX before the PC in stage 1.
Data Memory
en
R/W
ALU result
Load data
Opcode & Dest. Reg
Rest of pipelined datapath
EX/MEM
Pipeline register
MEM/WB
Pipeline register

Stage 5: Write back (WB)

- Completes execution of instruction by writing a value back to the register file.
- Instructions that do not write back to the register file do not do anything.

Stage 4: Memory datapath



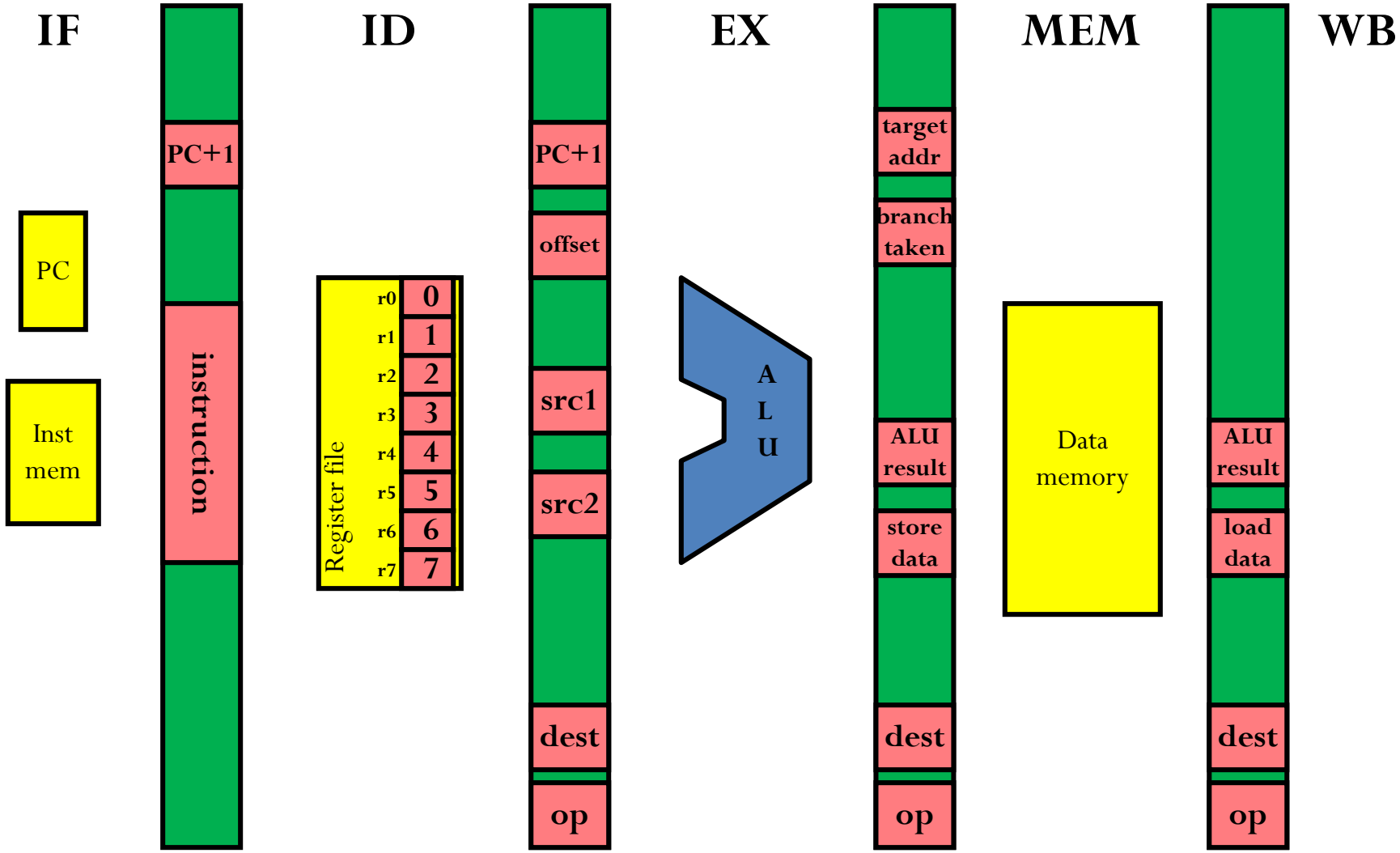
MEM/WB

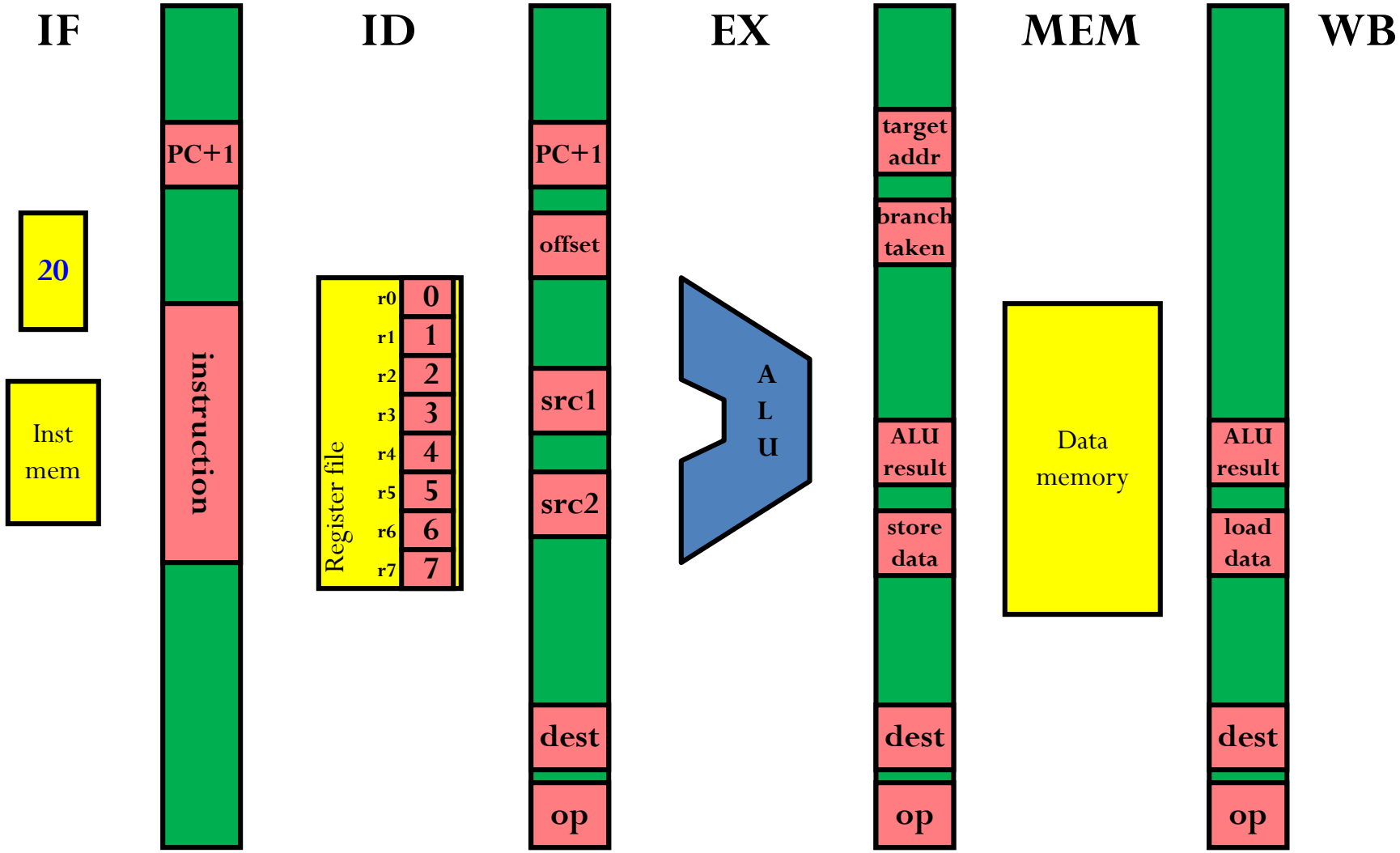
Pipeline register

Example: Sample program

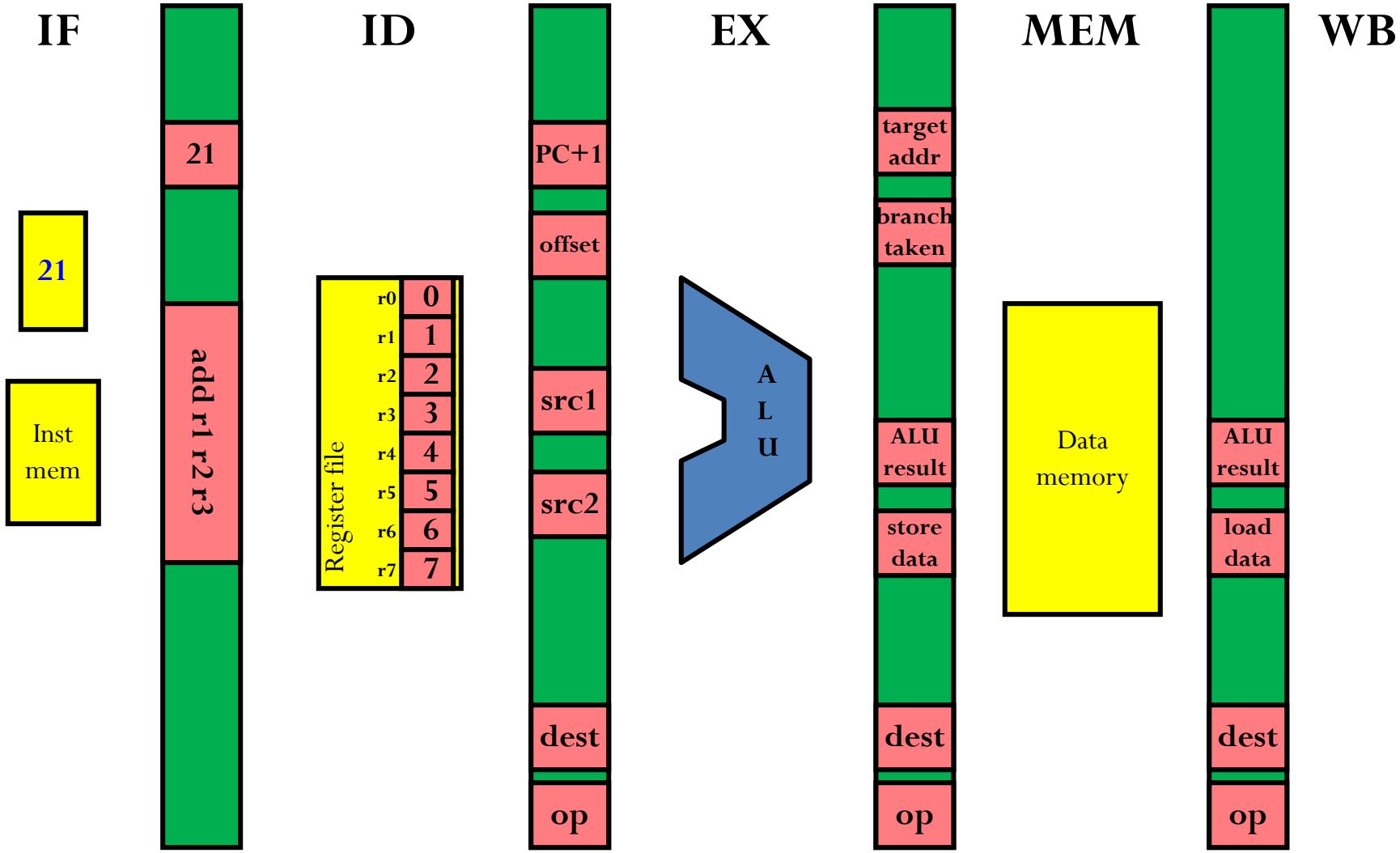
Run the following program on pipeline ANNA processor:

```
add r1 r2 r3          // PC = 20
lw r4 r5 1
and r6 r2 r3
sw r7 r5 9
or r4 r3 r7
```


IF
ID
EX
MEM
WB
PC
Inst mem
PC+1
instruction
Register file
r0
r1
r2
r3
r4
r5
r6
r7
0
1
2
3
4
5
6
7
PC+1
offset
src1
src2
dest
op
A
L
U
target addr
branch taken
ALU result
store data
dest
op
Data memory
ALU result
load data
dest
op

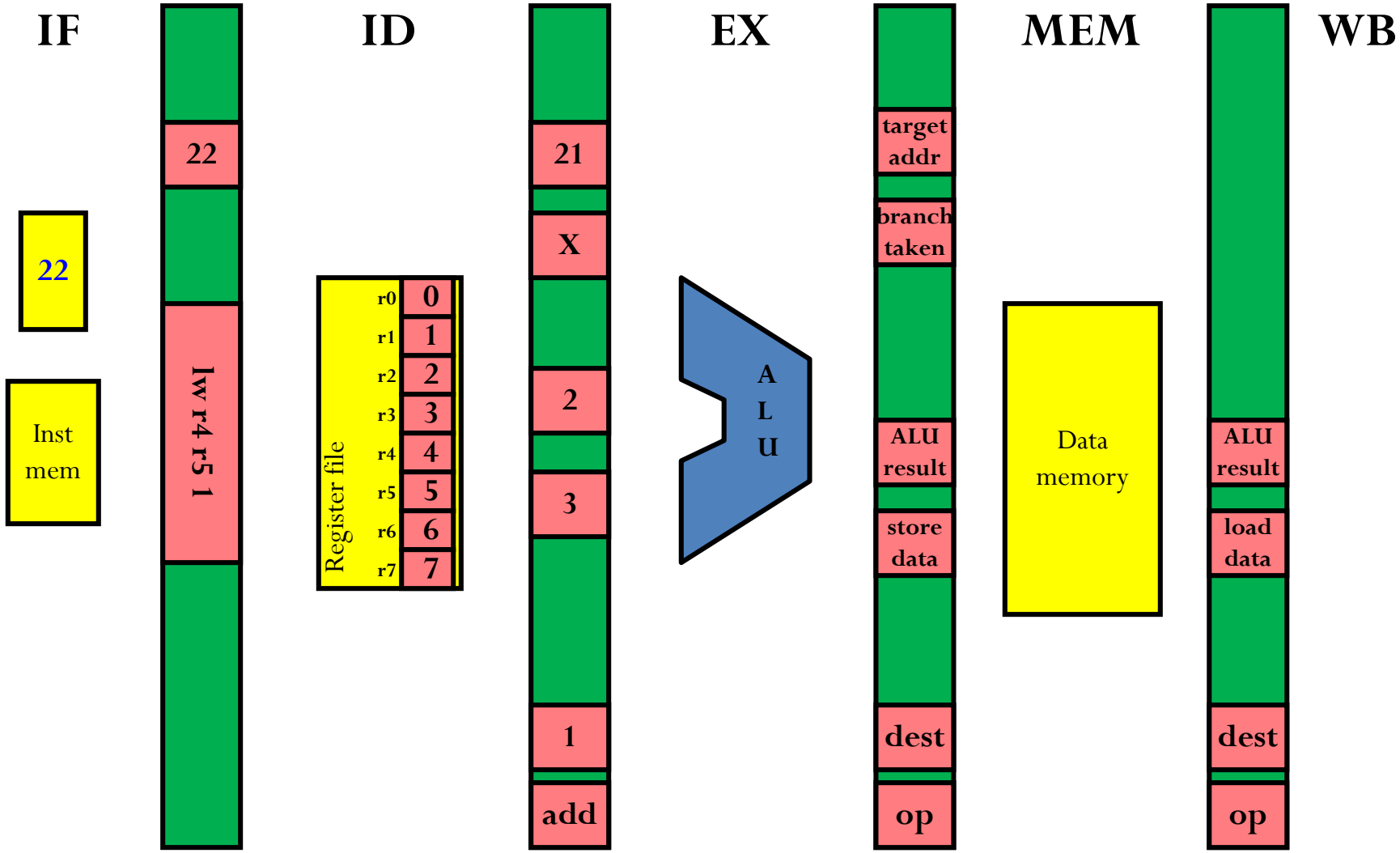

IF
ID
EX
MEM
WB
20
Inst mem
PC+1
instruction
Register file
r0
r1
r2
r3
r4
r5
r6
r7
0
1
2
3
4
5
6
7
PC+1
offset
src1
src2
dest
op
A
L
U
target addr
branch taken
ALU result
store data
dest
op
Data memory
ALU result
load data
dest
op

add r1 r2 r3

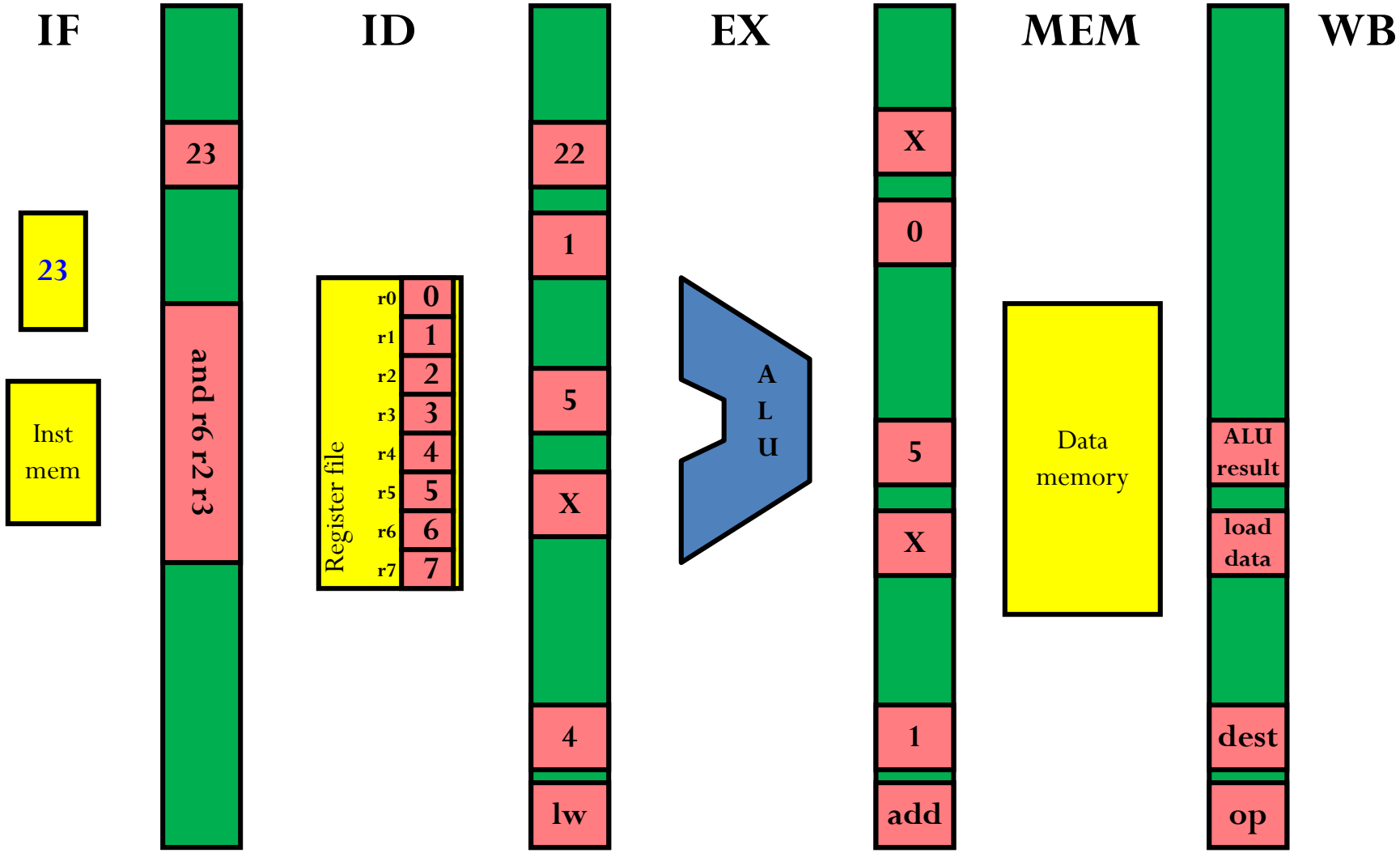


lw r4 r5 1

add r1 r2 r3



and r6 r2 r3 lw r4 r5 1 add r1 r2 r3

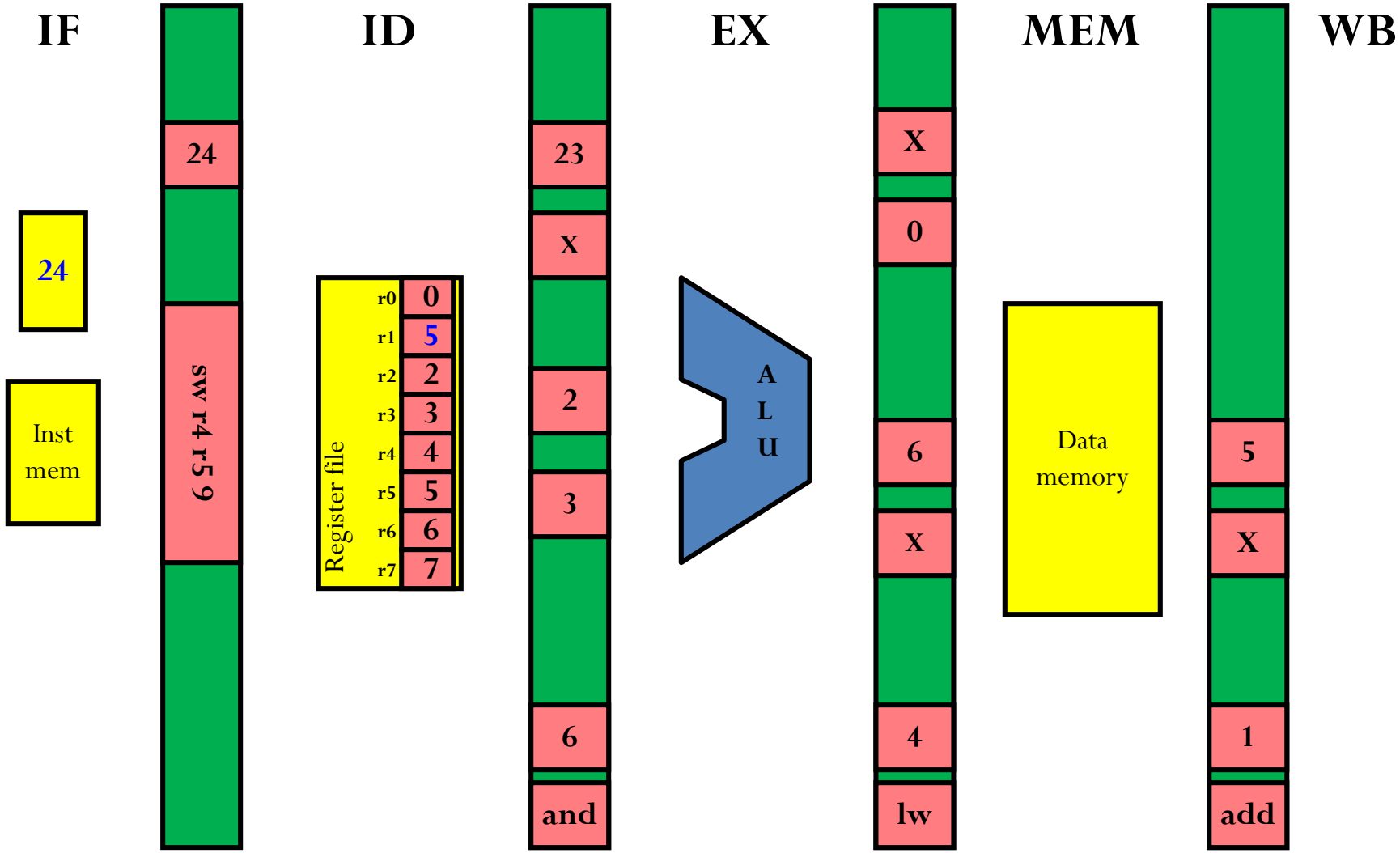


sw r7 r5 9

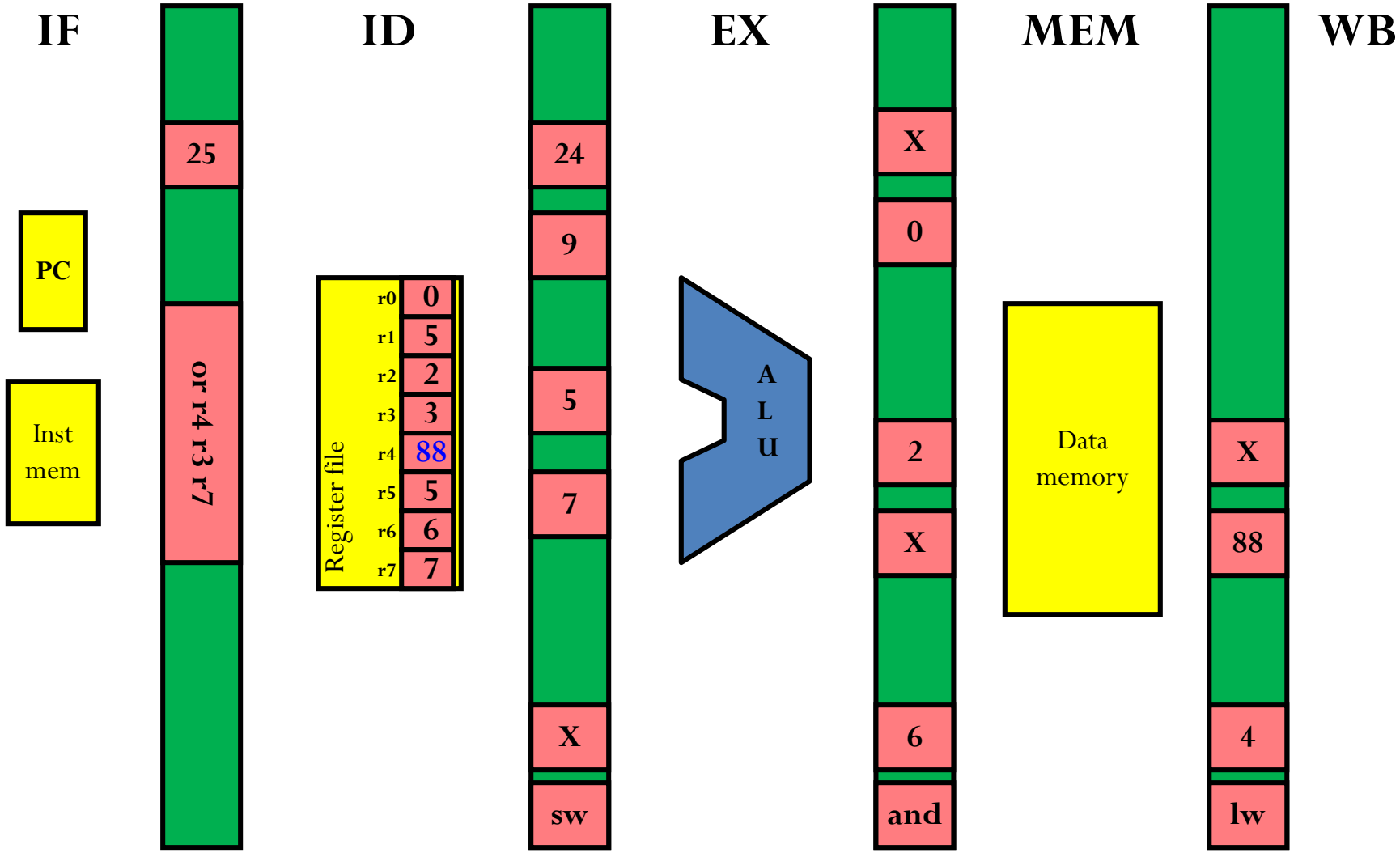
and r6 r2 r3

lw r4 r5 1

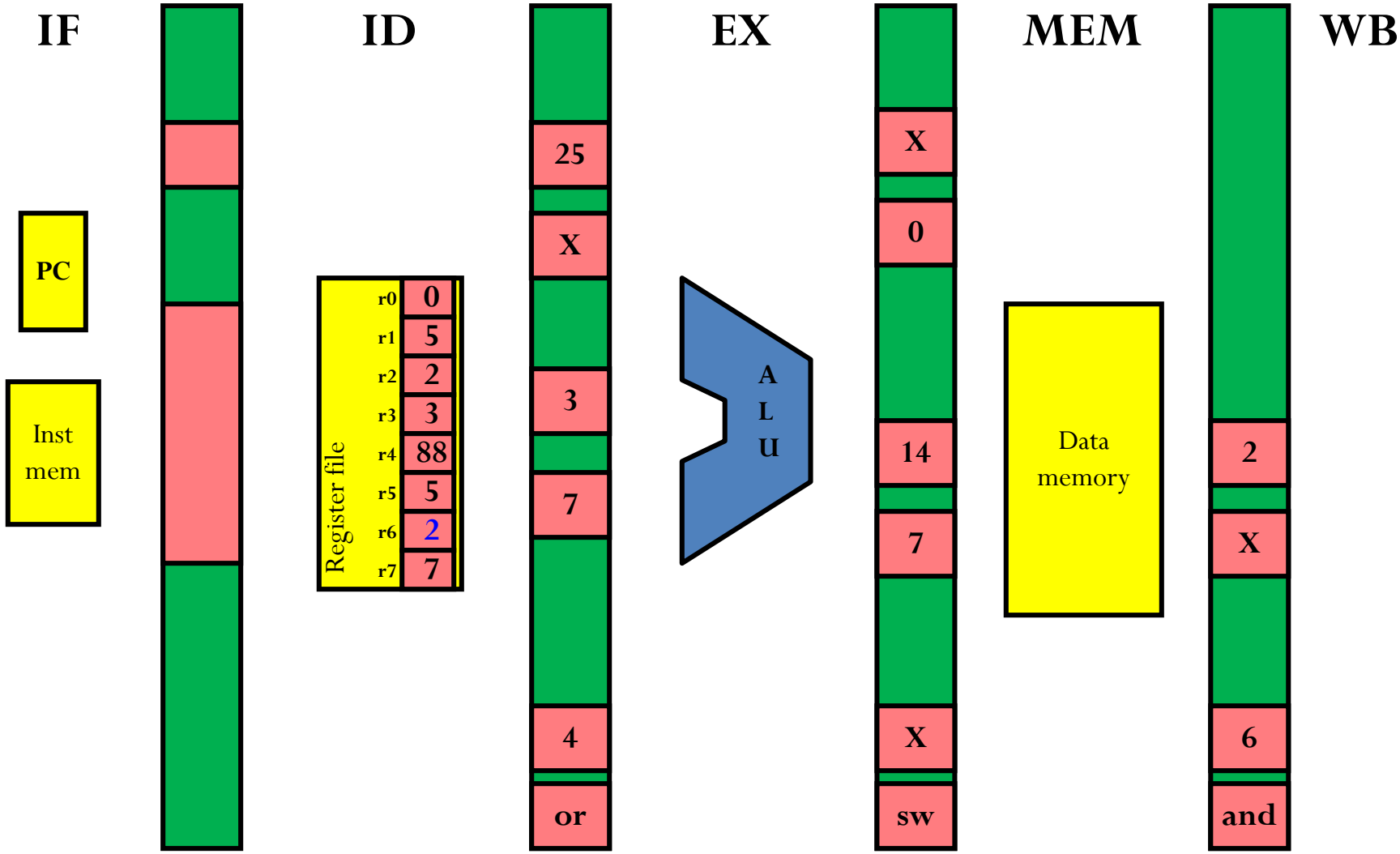
add r1 r2 r3



or r4 r3 r7 | sw r7 r5 9 | and r6 r2 r3 | lw r4 r5 1 | add r1 r2 r3



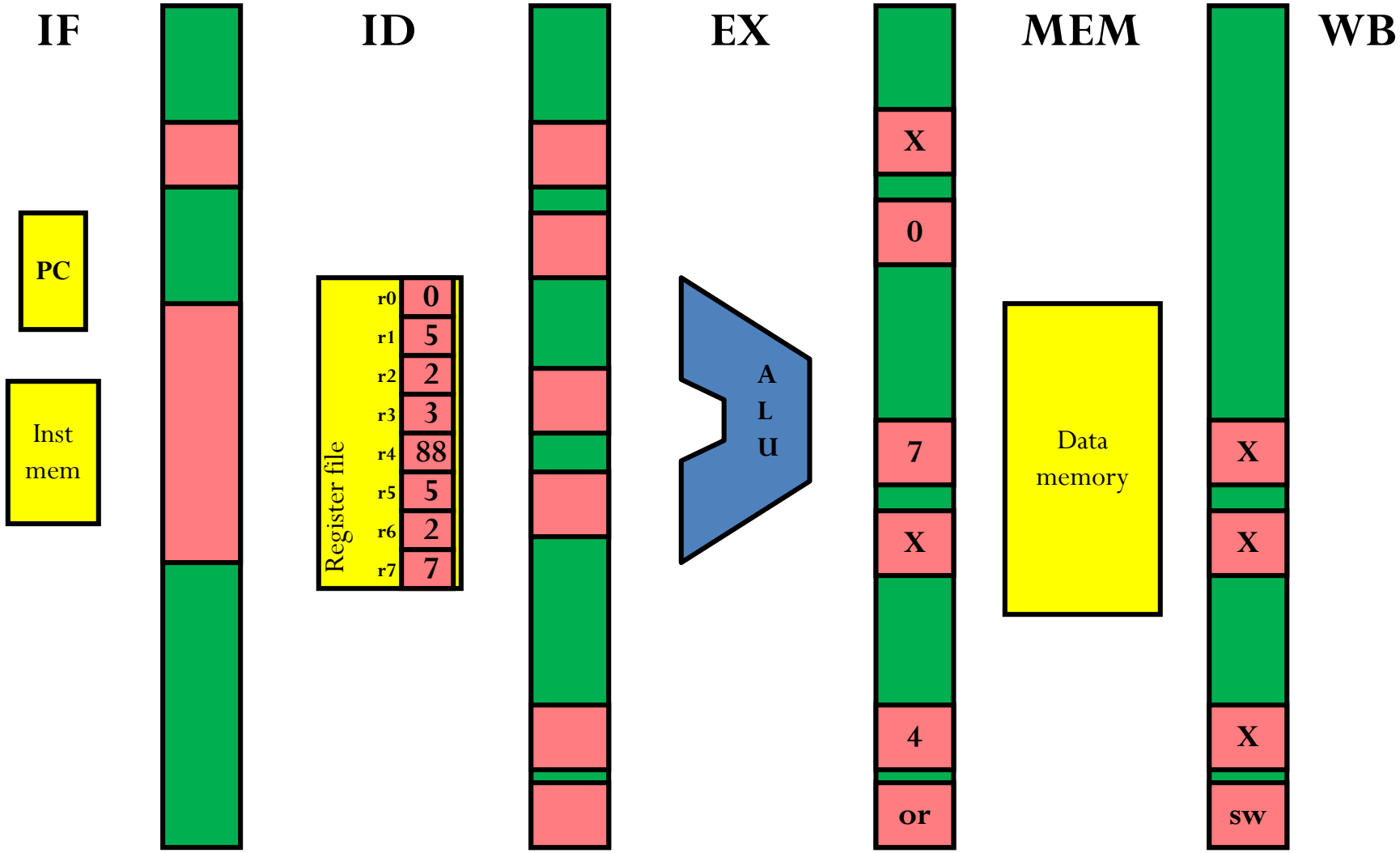
or r4 r3 r7 sw r7 r5 9 and r6 r2 3 lw r4 r5 1



or r4 r3 r7

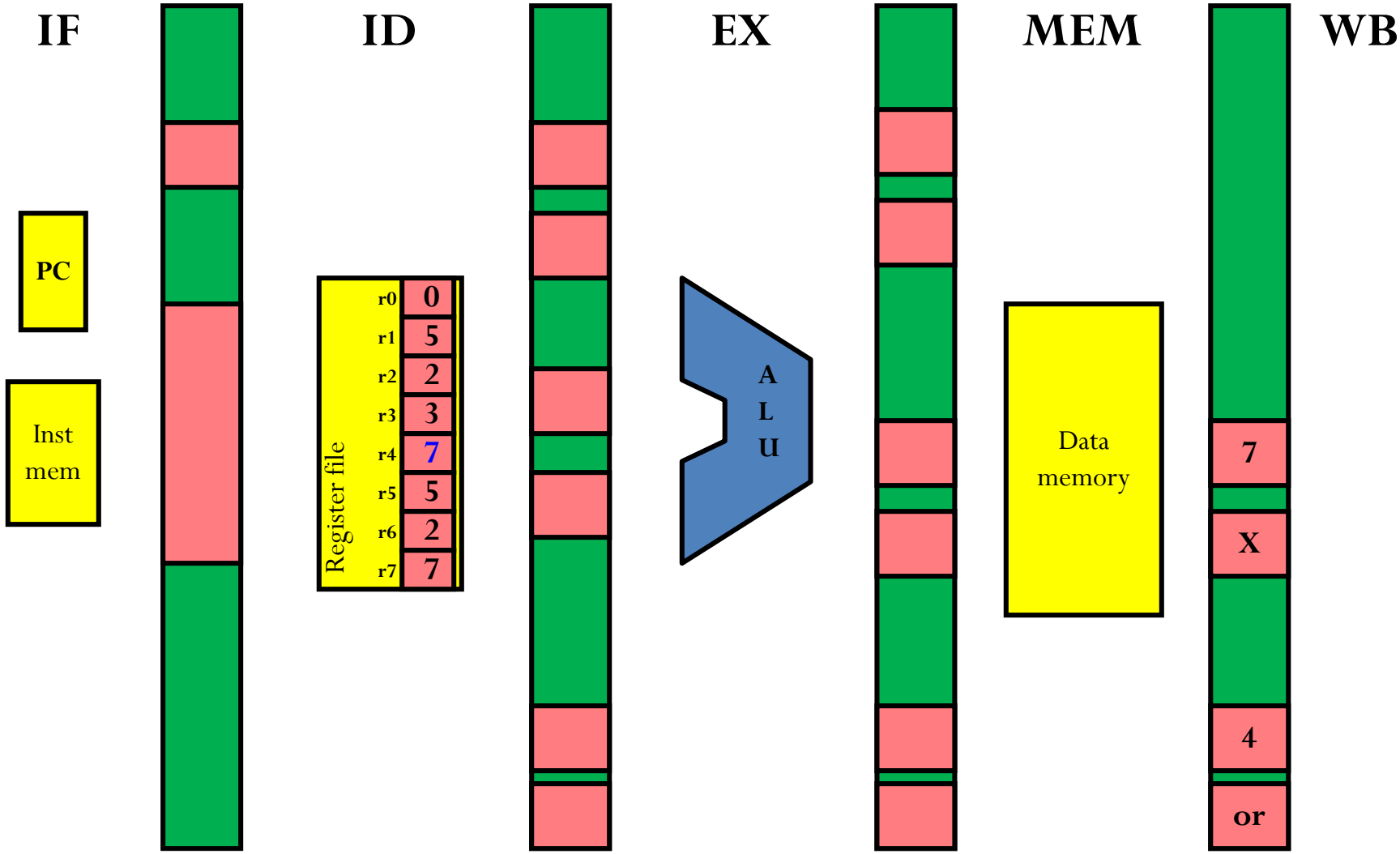
sw r7 r5 9

and r6 r2 r3



or r4 r3 r7

sw r7 r5 9


IF
ID
EX
MEM
WB
PC
Inst mem
Register file
r0 0
r1 5
r2 2
r3 3
r4 7
r5 5
r6 2
r7 7
A L U
Data memory
7
X
4
or

or r4 r3 r7

Time Graph

cycles

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13
add r1 r2 r3	IF	ID	EX	ME	WB								
lw r4 r5 1		IF	ID	EX	ME	WB							
and r6 r2 r3			IF	ID	EX	ME	WB						
sw r7 r5 9				IF	ID	EX	ME	WB					
or r4 r3 r7					IF	ID	EX	ME	WB				

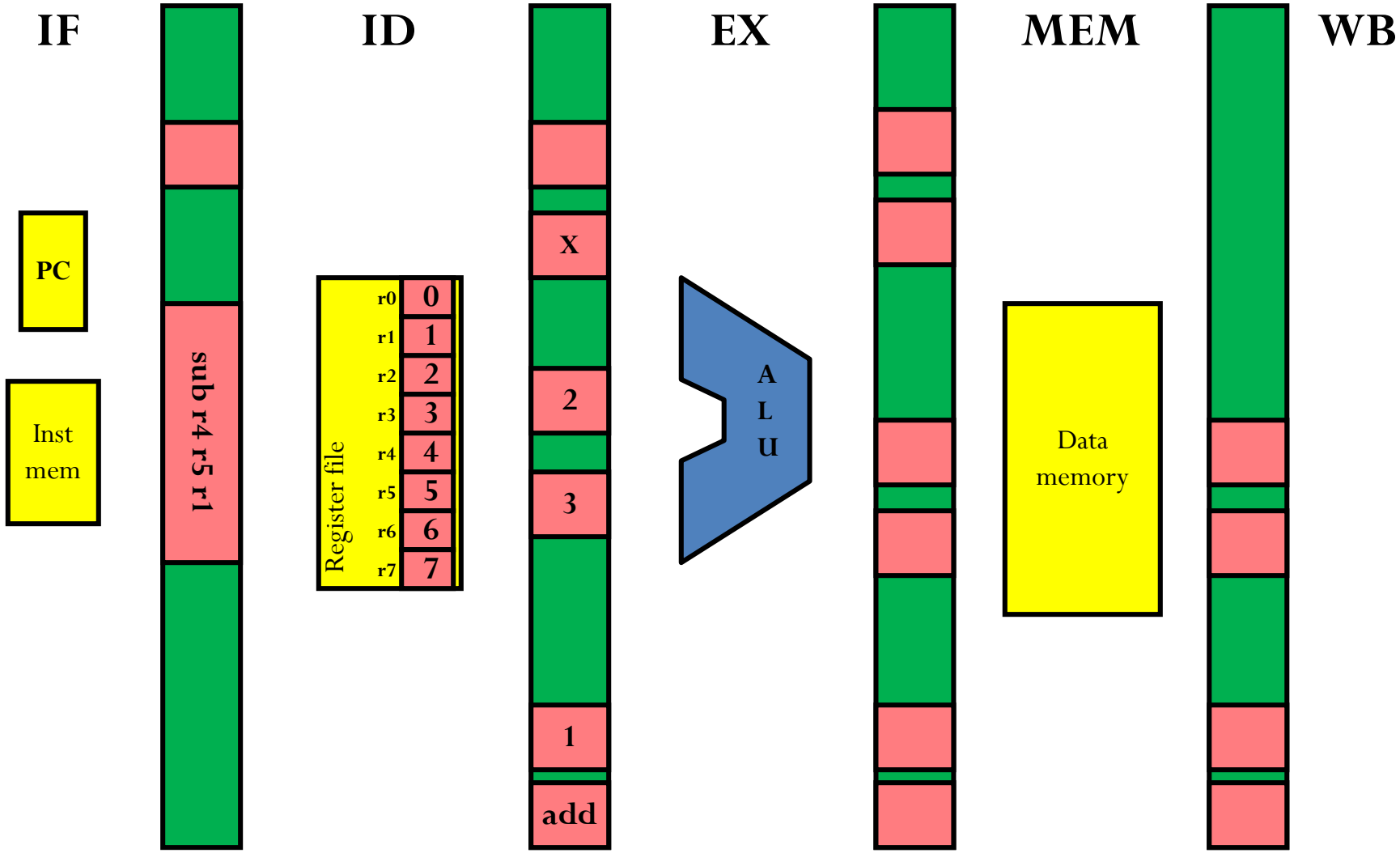
$$CPI = \frac{9 \text{ cycles}}{5 \text{ instructions}} = 1.8$$

Ignoring startup, $CPI \approx 1$.

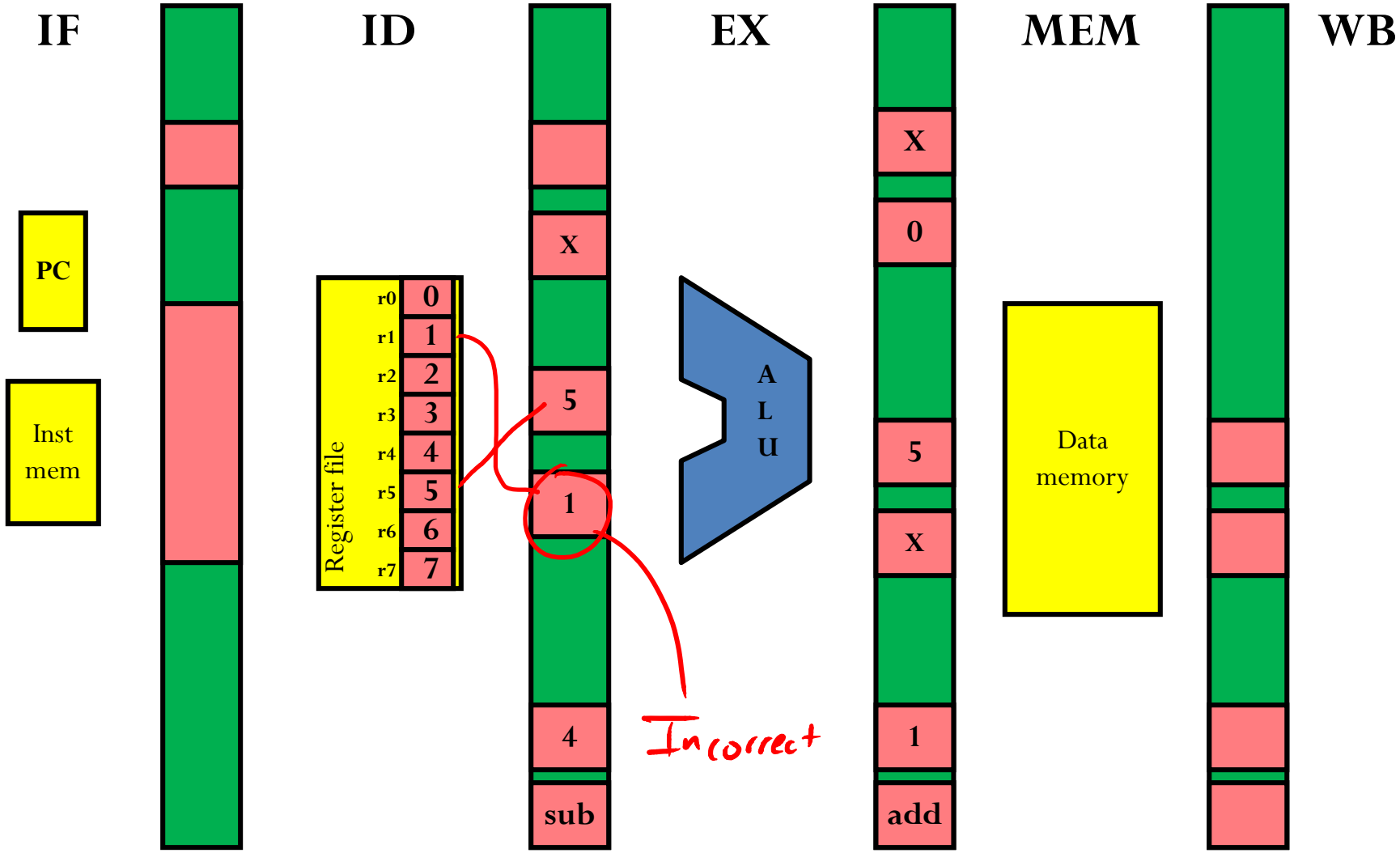
Problem: Dependent Instructions

What about this sequence of instructions?

```
add r1 r2 r3
sub r4 r5 r1
```



sub r4 r5 r1 add r1 r2 r3



sub r4 r5 r1

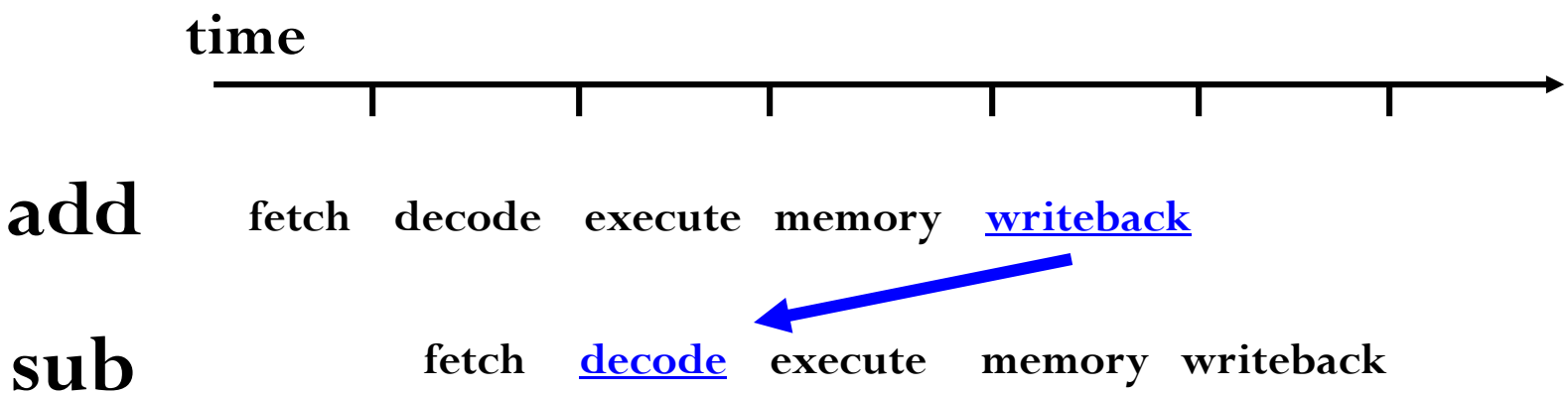
add r1 r2 r3

Data Hazards

- **hazard:** A problem in pipeline that leads to inconsistent results.
- **data hazard:** A hazard due to an instruction dependent on data produced by a later instruction in the pipeline.
- First question to ask:
  - How far apart, at a minimum, must dependent instructions be to avoid a data hazard?

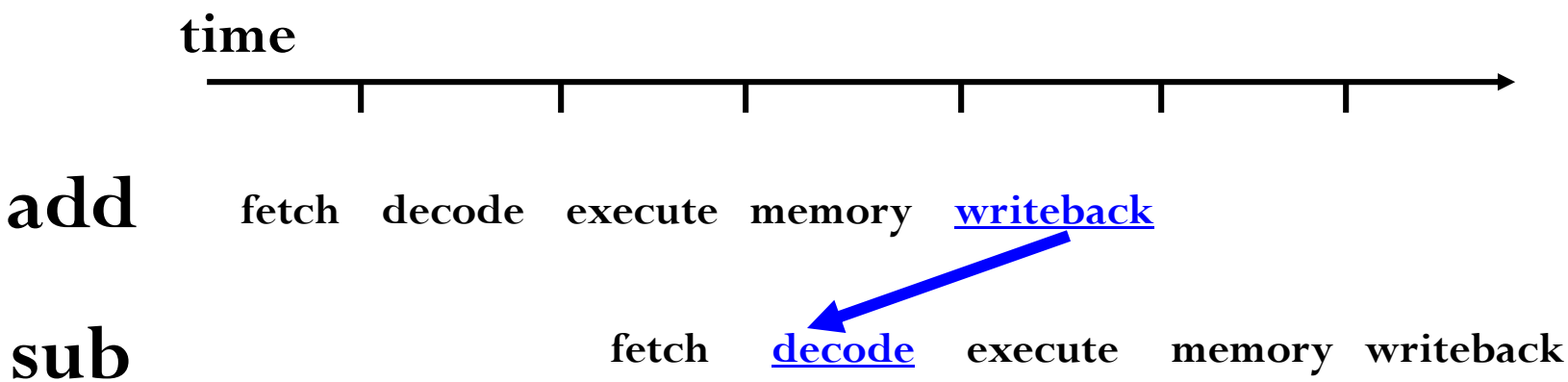
Data Hazards

```
add  r1 r2 r3
sub  r4 r5 r1
```



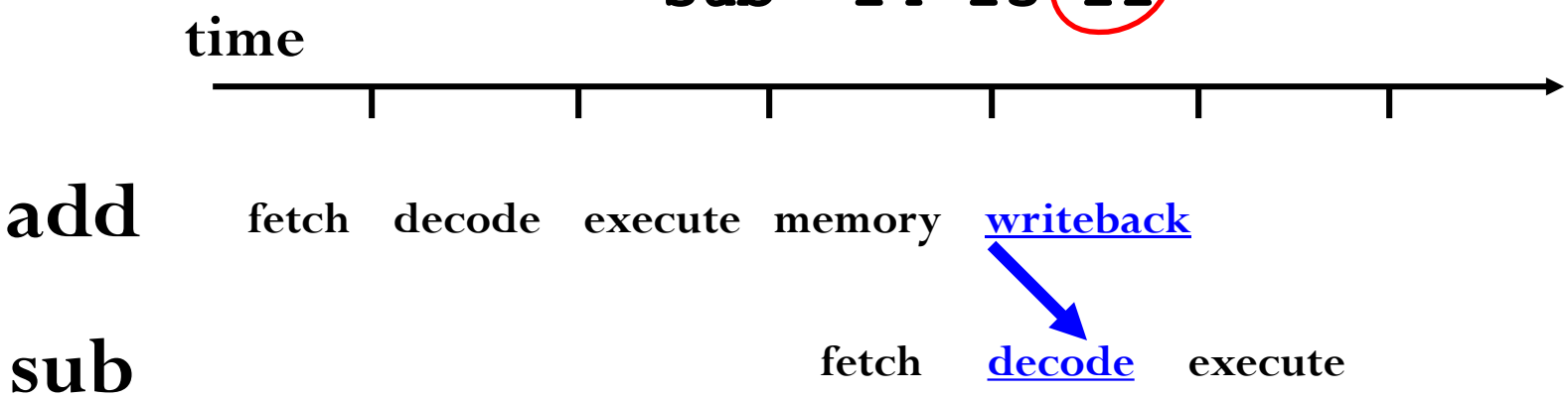
Data Hazards

```
add  r1 r2 r3
add  r0 r0 r0  ← nop
sub  r4 r5 r1
```



Data Hazards

```
add  r1 r2 r3
add  r0 r0 r0
add  r0 r0 r0
sub  r4 r5 r1
```



Assume that register is written in first half of cycle, read in second half clock cycle.

Data Hazards in ANNA

- A data hazard occurs in the ANNA pipeline when an instruction is dependent on the result of either of the two instructions that precede it.
- Dependent instructions must be separated with at least two additional instructions to avoid data hazards.
- Are data hazards possible with memory (i.e. a load is dependent on the value of a store)?

No - reads and writes occur in the same stage.
(loads) (stores)

Class Problem

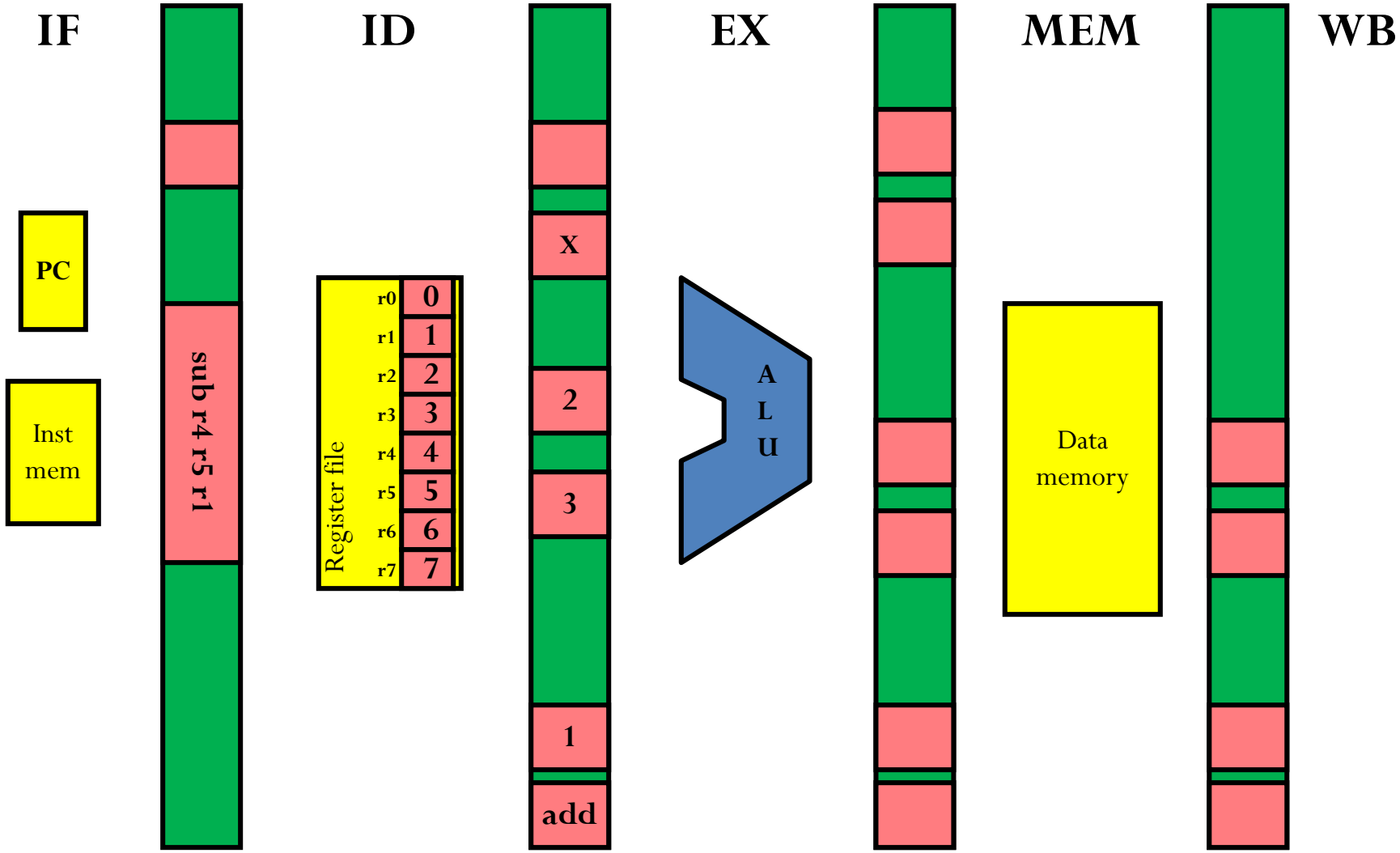
How many data hazards do you see?

SIX

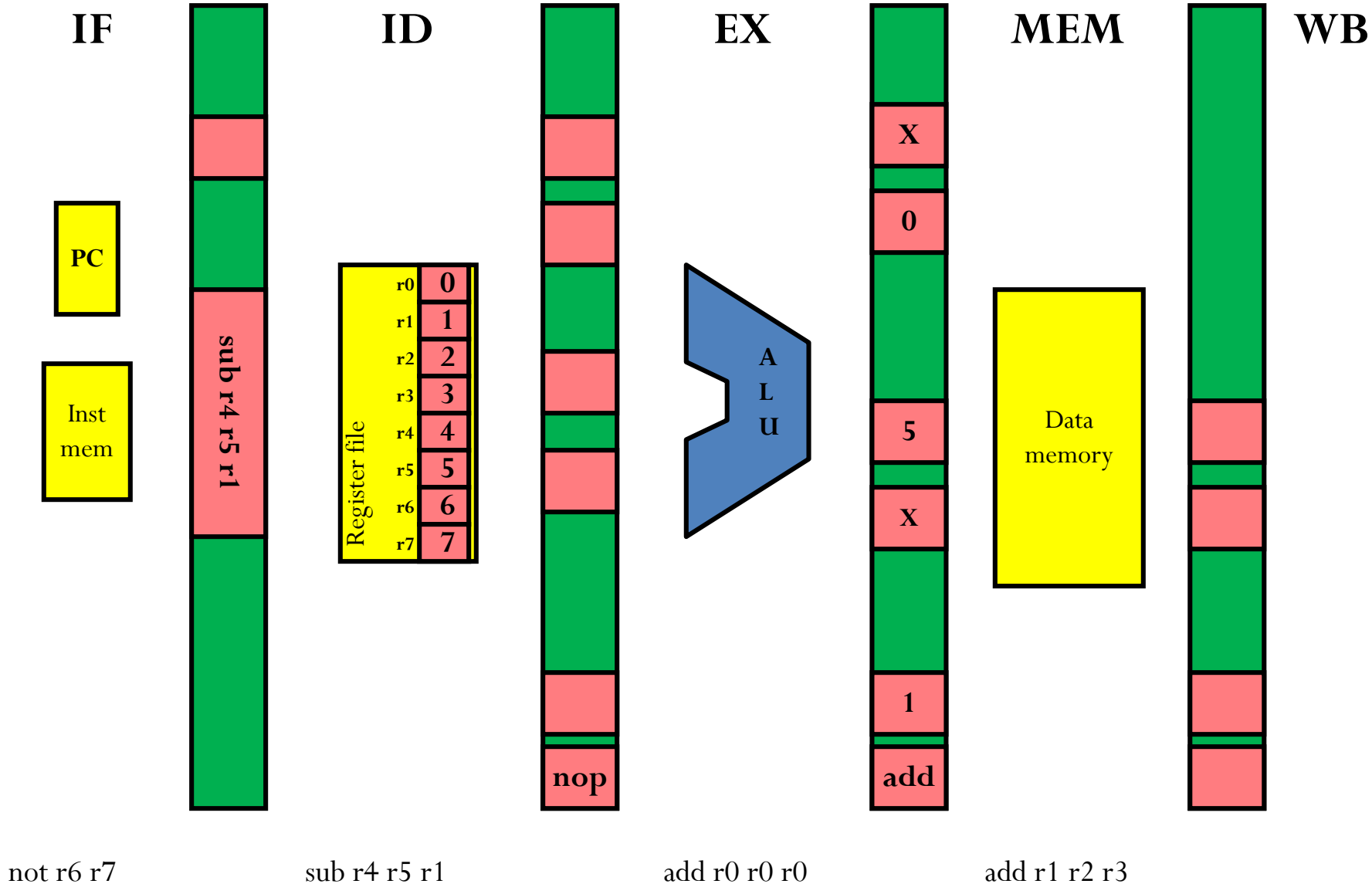
```
add r1 r2 r3
and r3 r1 r4
sub r7 r3 r1
lw  r7 r3 7
sw  r7 r3 6
lui r7 0x56
```

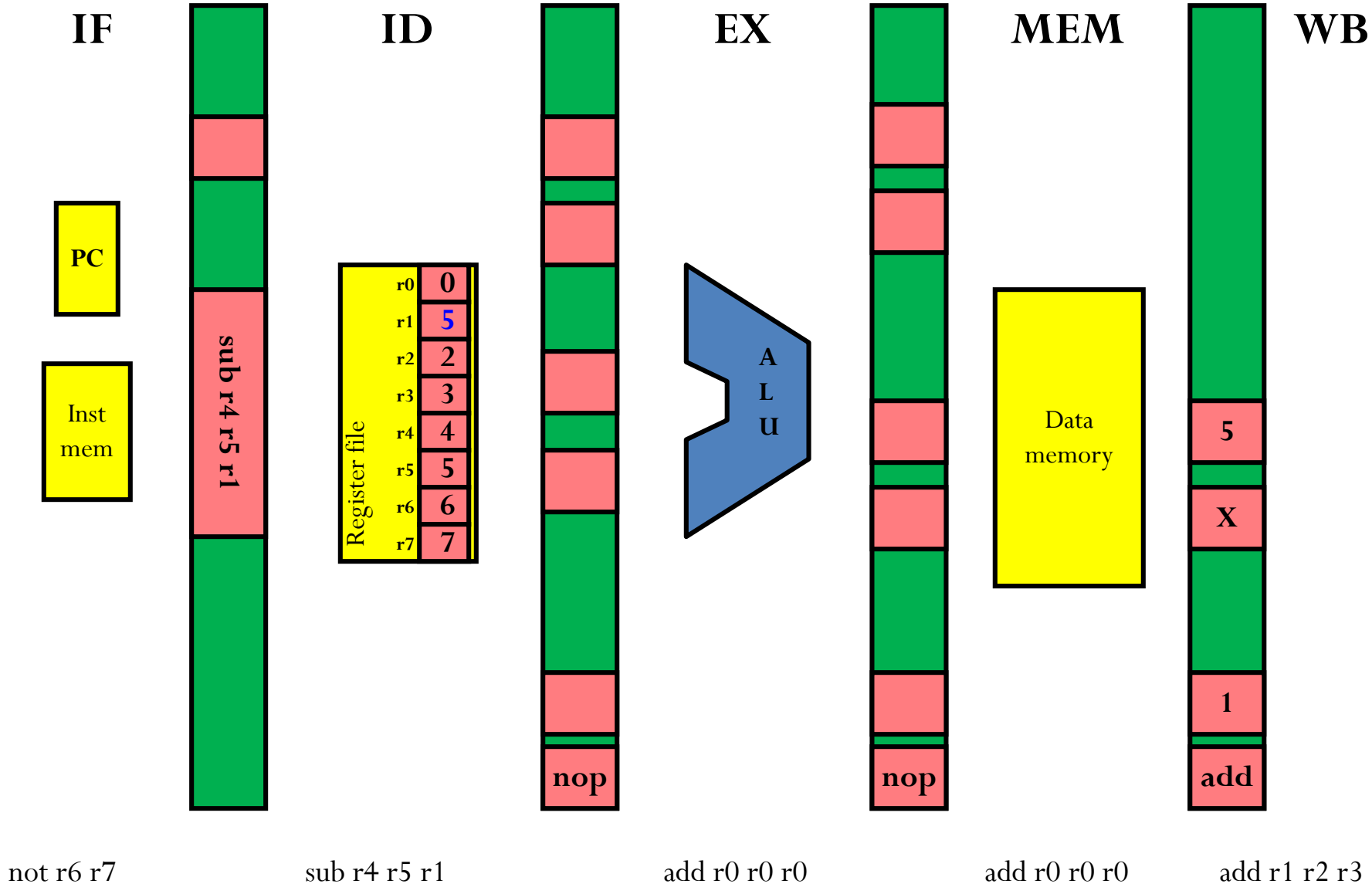
Handling Data Hazards

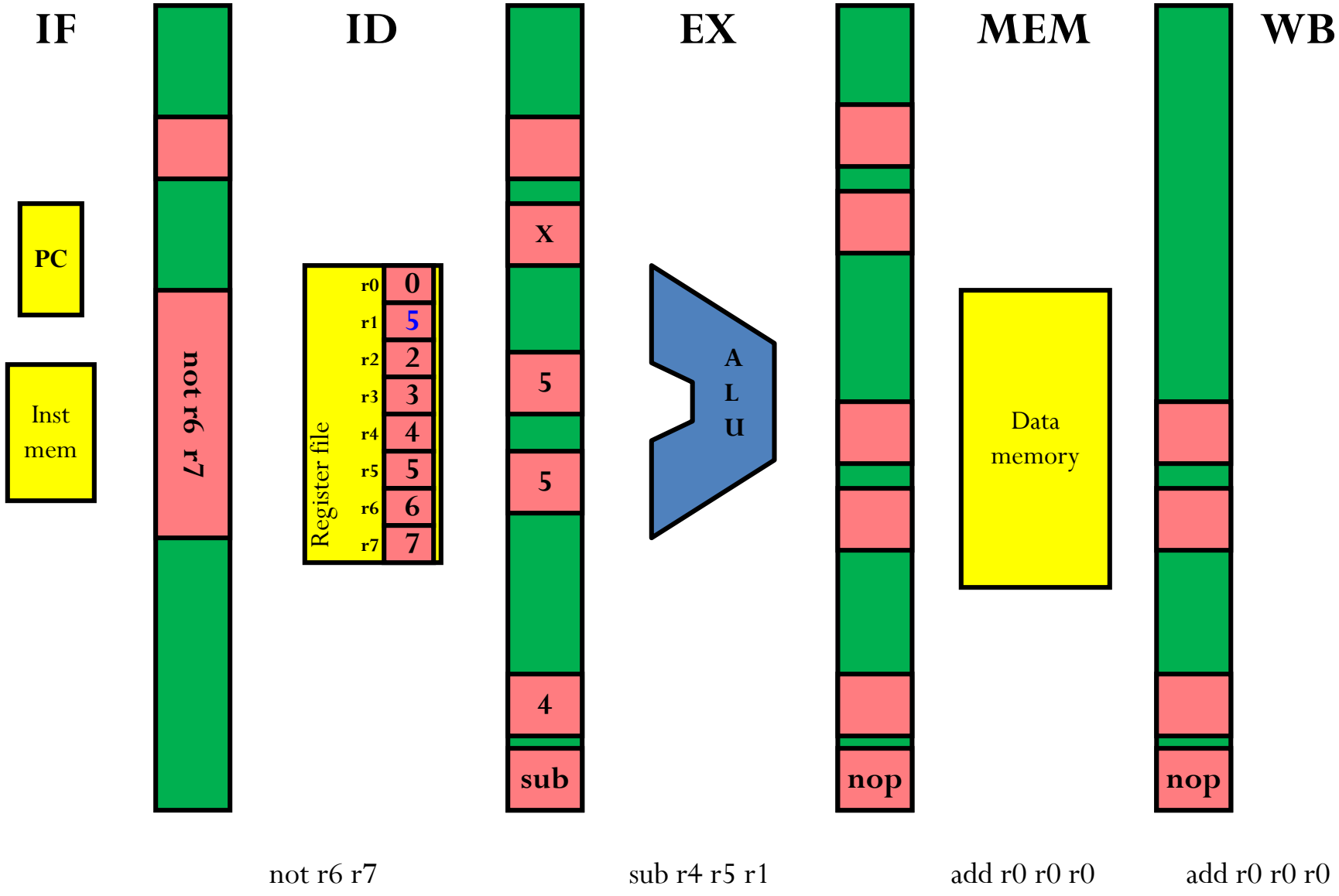
- The microprocessor detects if a data hazard occurs. If so, stall the pipeline.
- Detection:
  - Compare the two register sources (if used) with the destination registers (if used) of the previous two instructions sent to EX stage.
- Stall:
  - Keep current instructions in fetch and decode.
  - Pass a nop (`add r0 r0 r0`) to EX stage.



not r6 r7 sub r4 r5 r1 add r1 r2 r3


IF
ID
EX
MEM
WB
PC
Inst mem
sub r4 r5 r1
Register file
r0 0
r1 1
r2 2
r3 3
r4 4
r5 5
r6 6
r7 7
nop
A L U
X
0
5
X
1
add
Data memory
not r6 r7
sub r4 r5 r1
add r0 r0 r0
add r1 r2 r3


IF
ID
EX
MEM
WB
PC
Inst mem
sub r4 r5 r1
Register file
r0 0
r1 5
r2 2
r3 3
r4 4
r5 5
r6 6
r7 7
A L U
nop
nop
Data memory
5
X
1
add
not r6 r7
sub r4 r5 r1
add r0 r0 r0
add r0 r0 r0
add r1 r2 r3


IF
ID
EX
MEM
WB
PC
Inst mem
not r6 r7
Register file
r0 0
r1 5
r2 2
r3 3
r4 4
r5 5
r6 6
r7 7
x
5
5
4
sub
A L U
nop
Data memory
nop
not r6 r7
sub r4 r5 r1
add r0 r0 r0
add r0 r0 r0

Time Graph

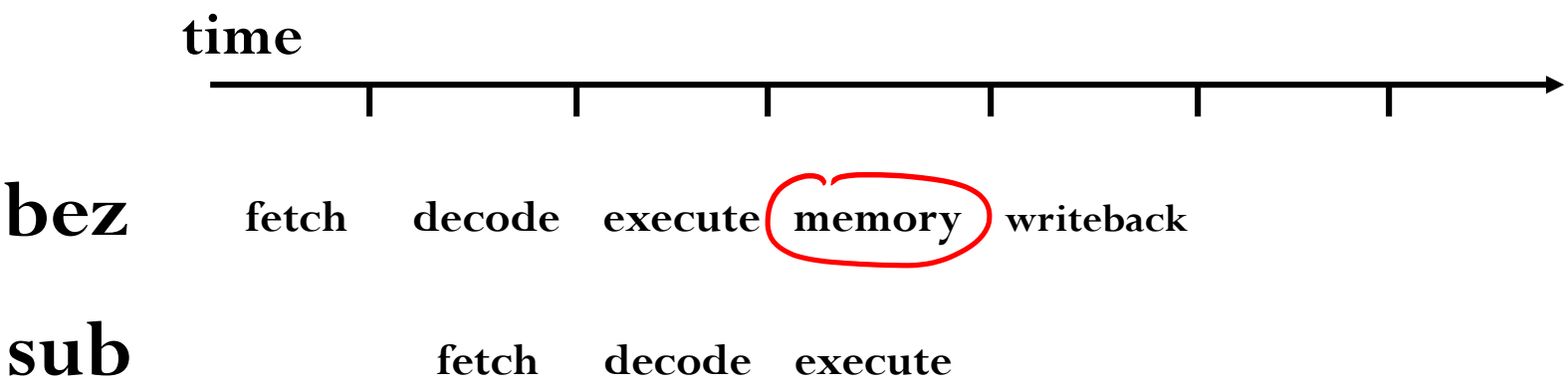
Time:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
add r1 r2 r3	IF	ID	EX	ME	WB															
and r3 r1 r4		IF	ID	ID	ID	EX	ME	WB												
sub r7 r3 r1			IF	IF	IF	ID	ID	ID	EX	ME	WB									
lw r7 r3 -7						IF	IF	IF	ID	EX	ME	WB								
sw r7 r3 6									IF	ID	ID	ID	EX	ME	WB					
lui r7 0x56										IF	IF	IF	ID	EX	ME	WB				

$$CPI = 16/6 = 2.67$$

Branches in Pipelines

Consider a `bez` instruction in pipeline:

```
bez r1 0x10
sub r3 r4 r5
```



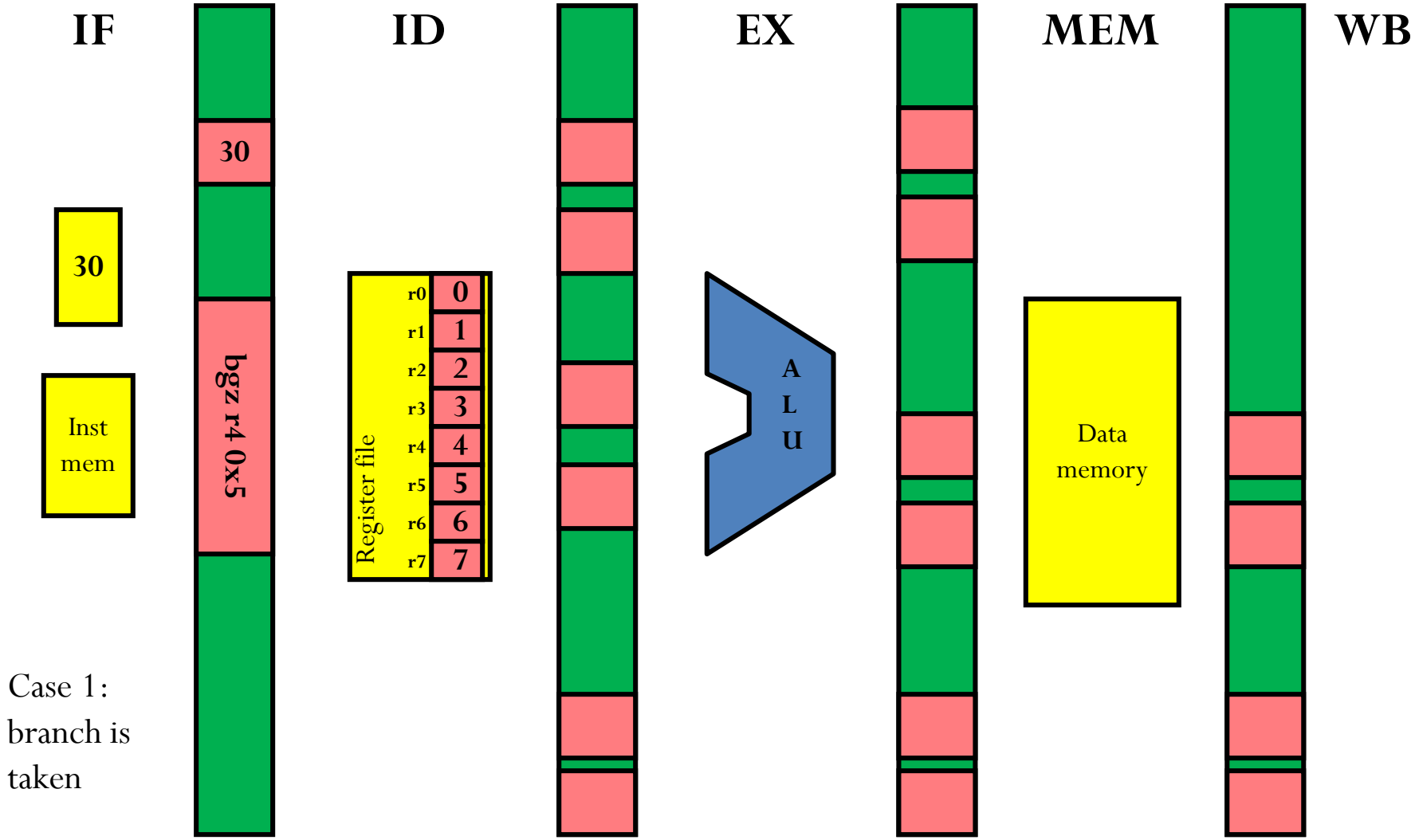
Control Hazards

- A **control hazard** occurs anytime a branch or jump instruction is executed.
- Fetch stage does not know what address to fetch next instruction from.
- In ANNA, branches are resolved in the MEM stage.

jalr

Handling Control Hazards: Stall

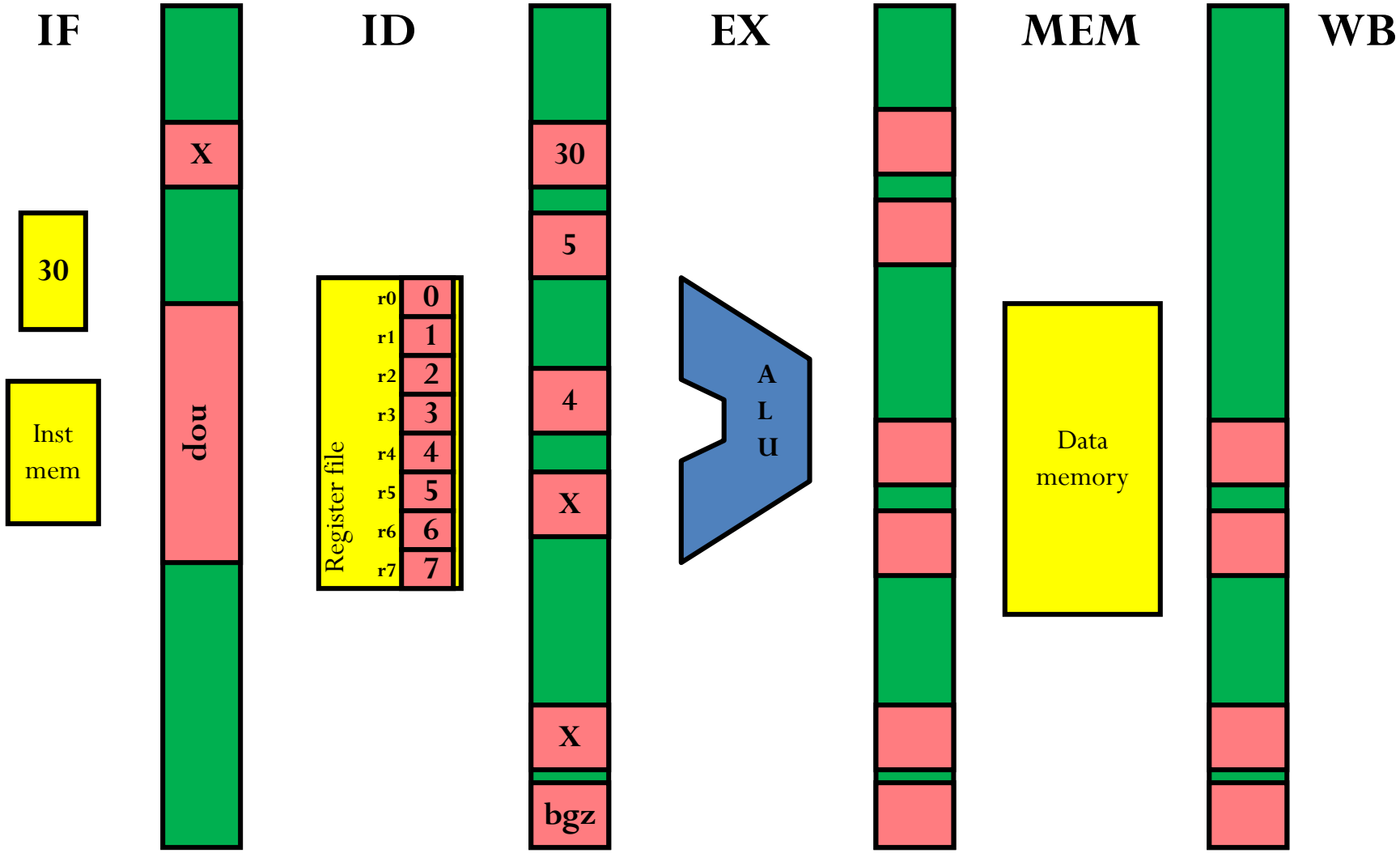
- Detect a branch and stall fetch.
- Earliest point that a branch can be detected is decode.
  - Detection is easy: look for branch or jump opcode.
- Stall:
  - Keep current instruction in fetch.
  - Pass nop to decode stage (not execute)!



Case 1: branch is taken

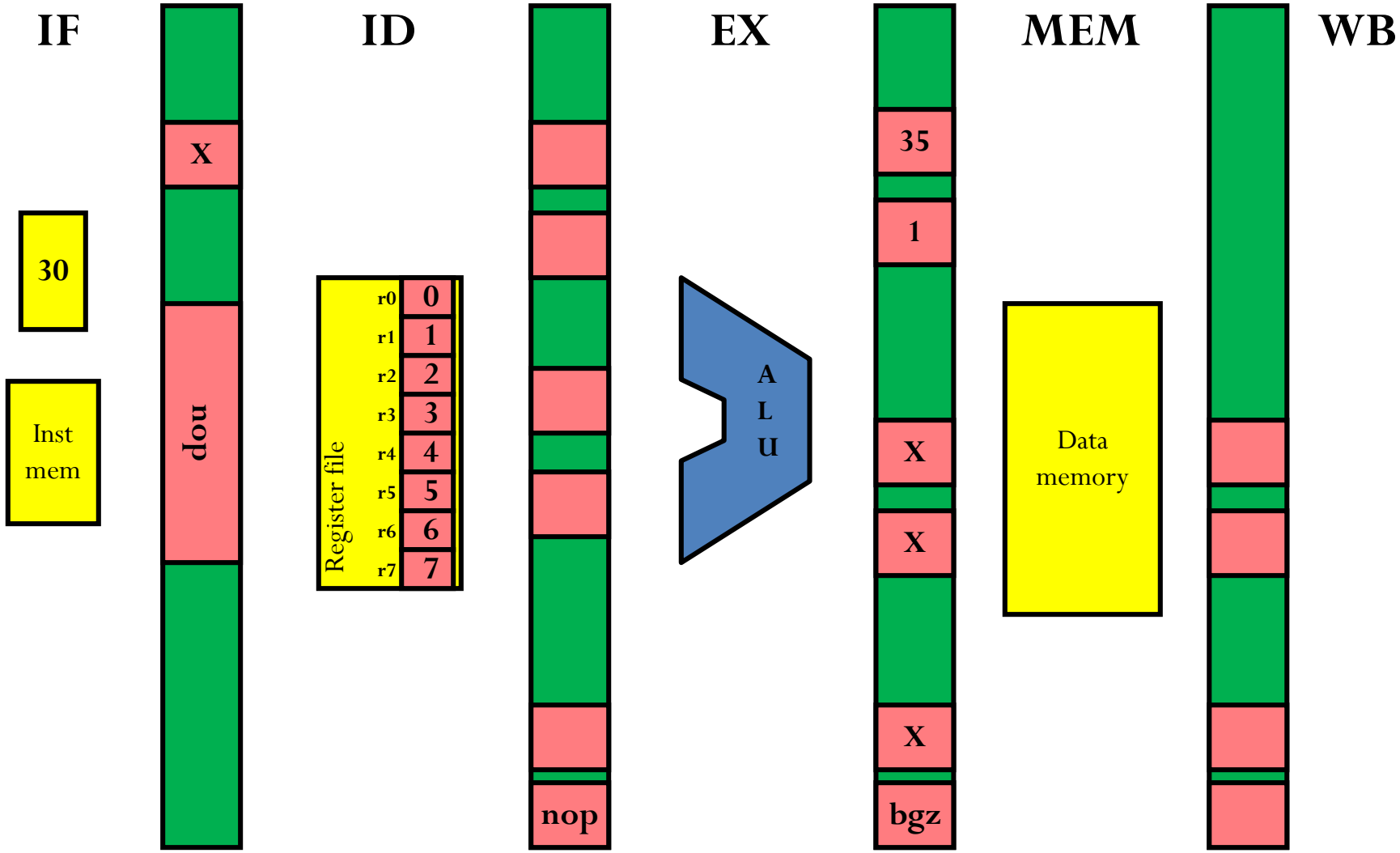
add r1 r2 r3

bgz r4 0x5



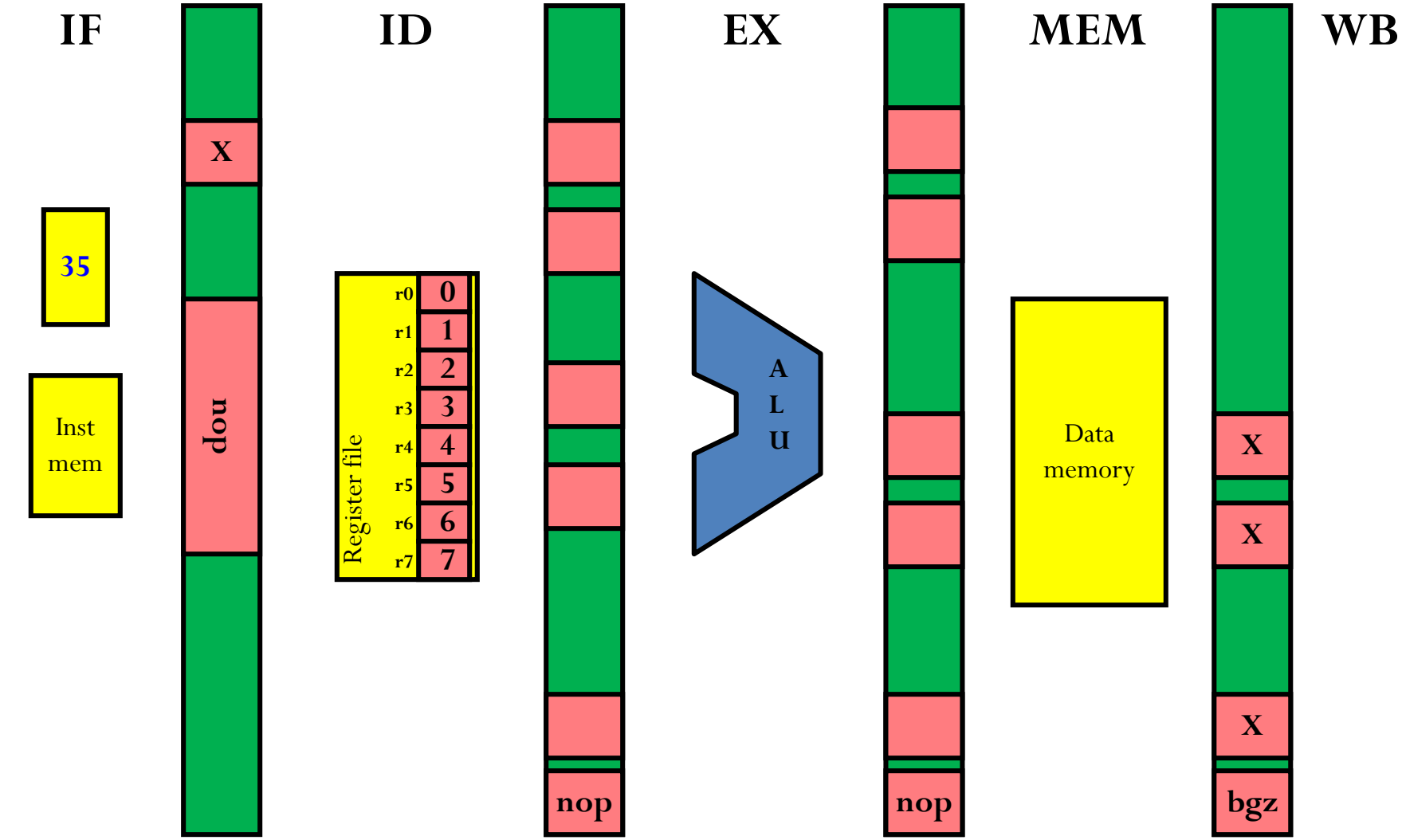
add r1 r2 r3

bgz r4 0x5



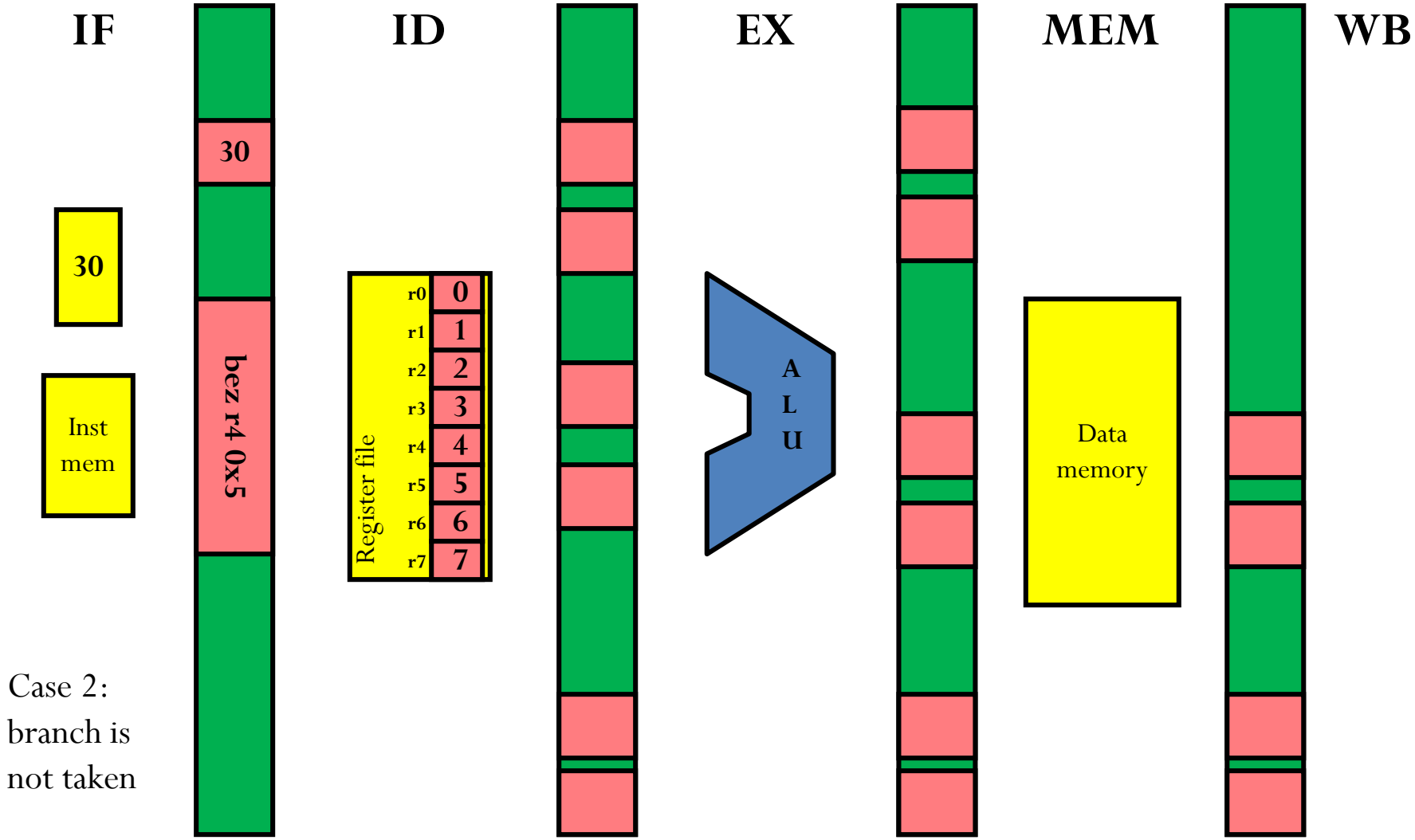
add r1 r2 r3

bgz r4 0x5



sw r5 r8 3

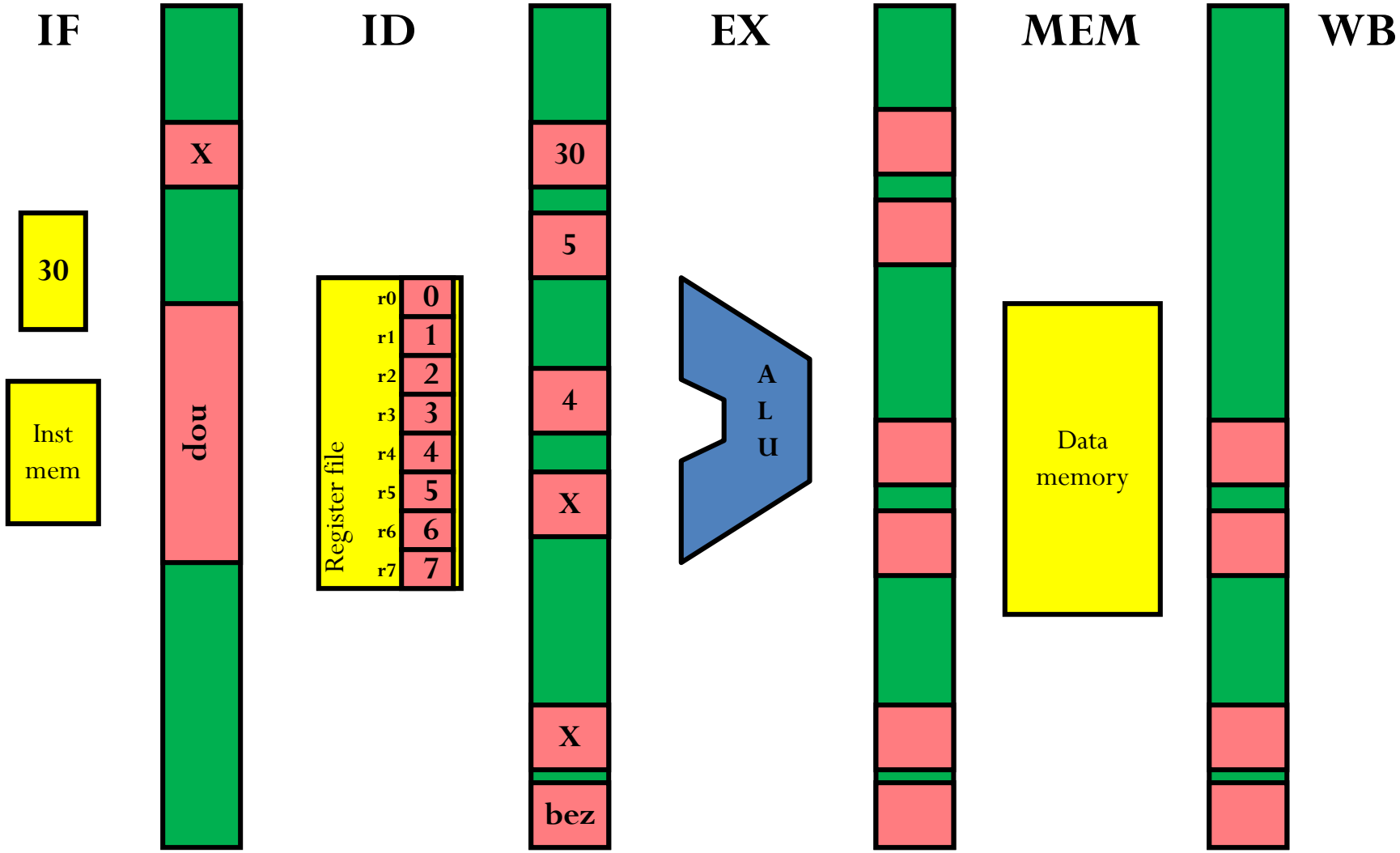
bgz r4 0x5



Case 2: branch is not taken

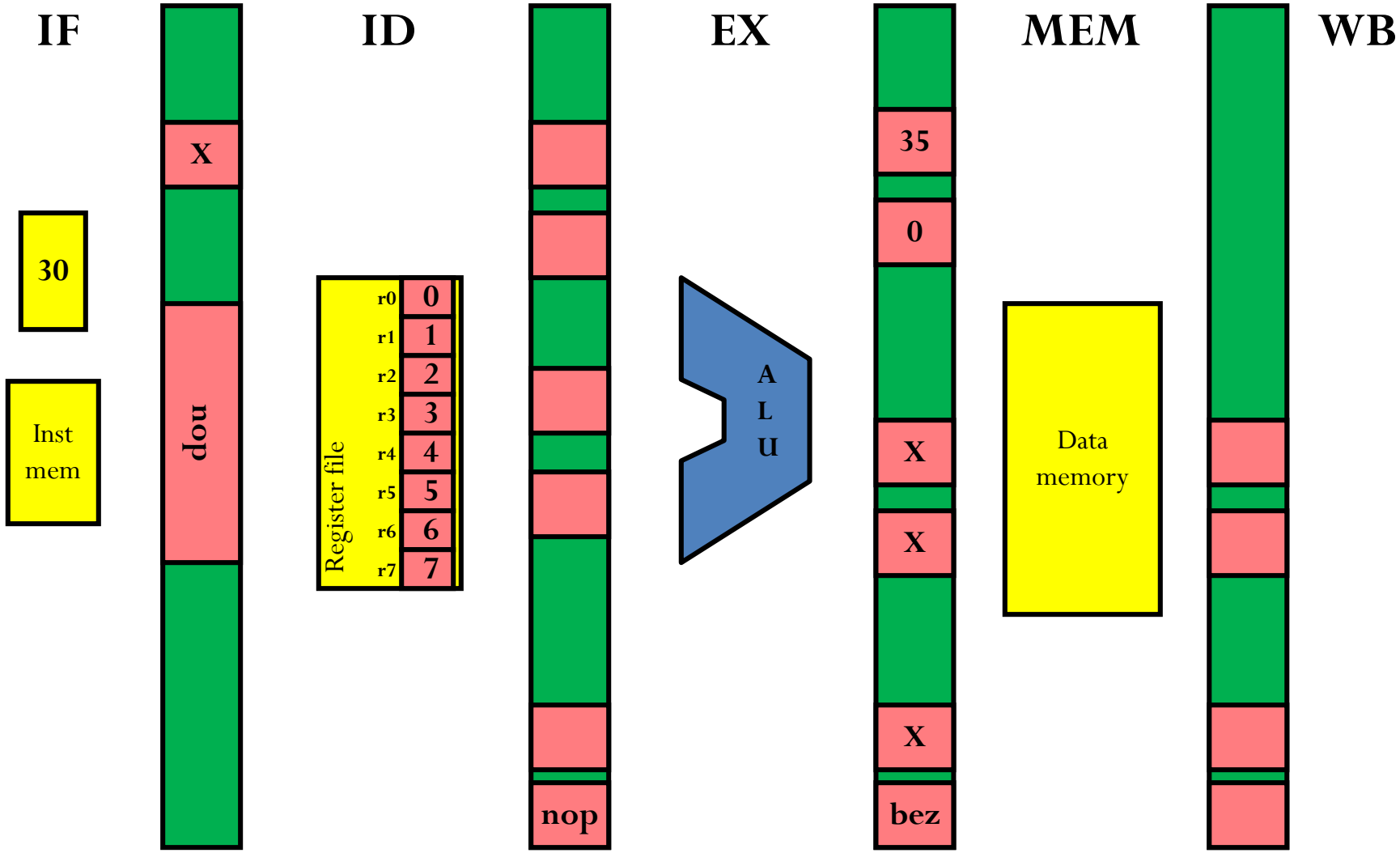
add r1 r2 r3

bez r4 0x5



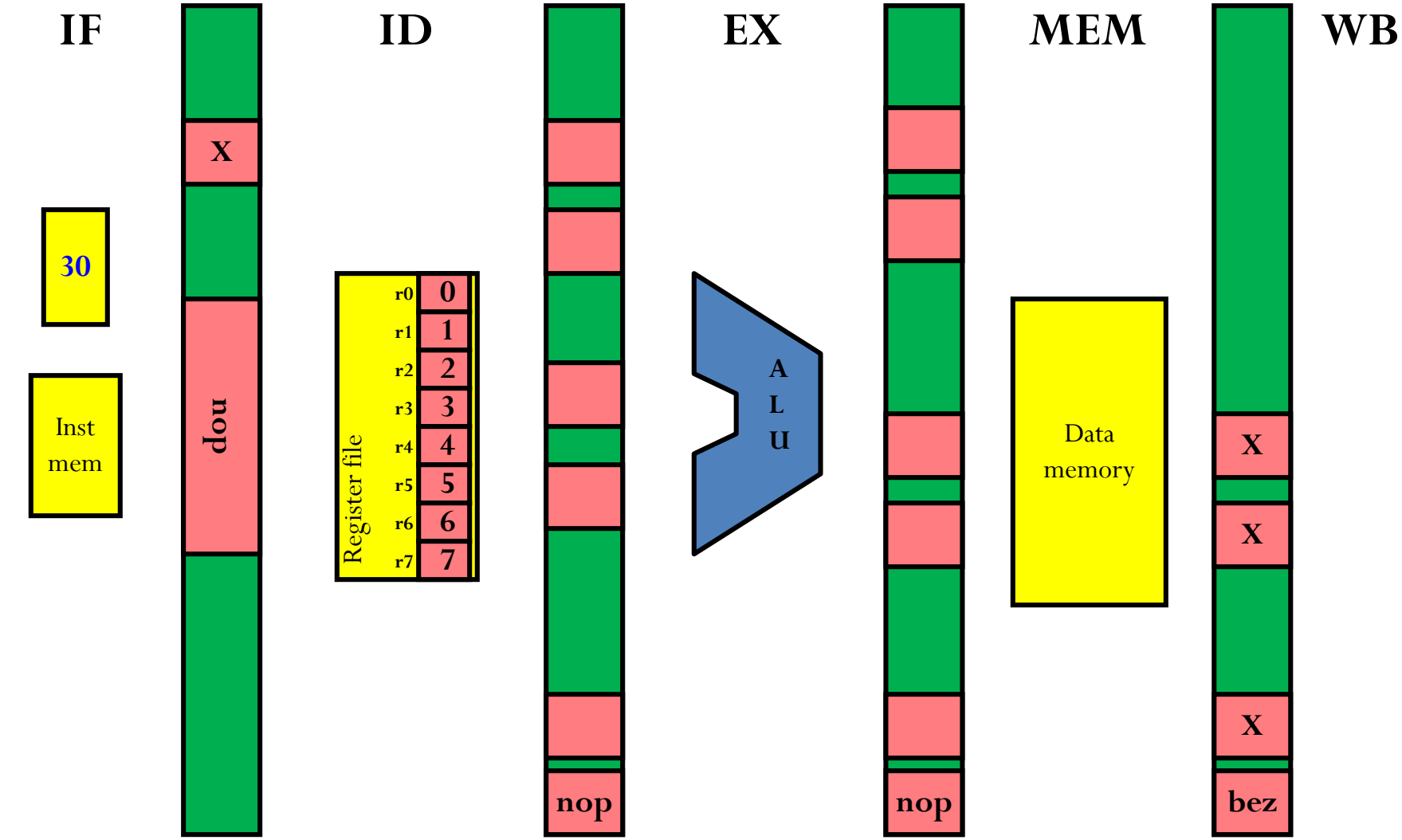
add r1 r2 r3

bez r4 0x5



add r1 r2 r3

bez r4 0x5



add r1 r2 r3

bez r4 0x5

Pipelining Performance

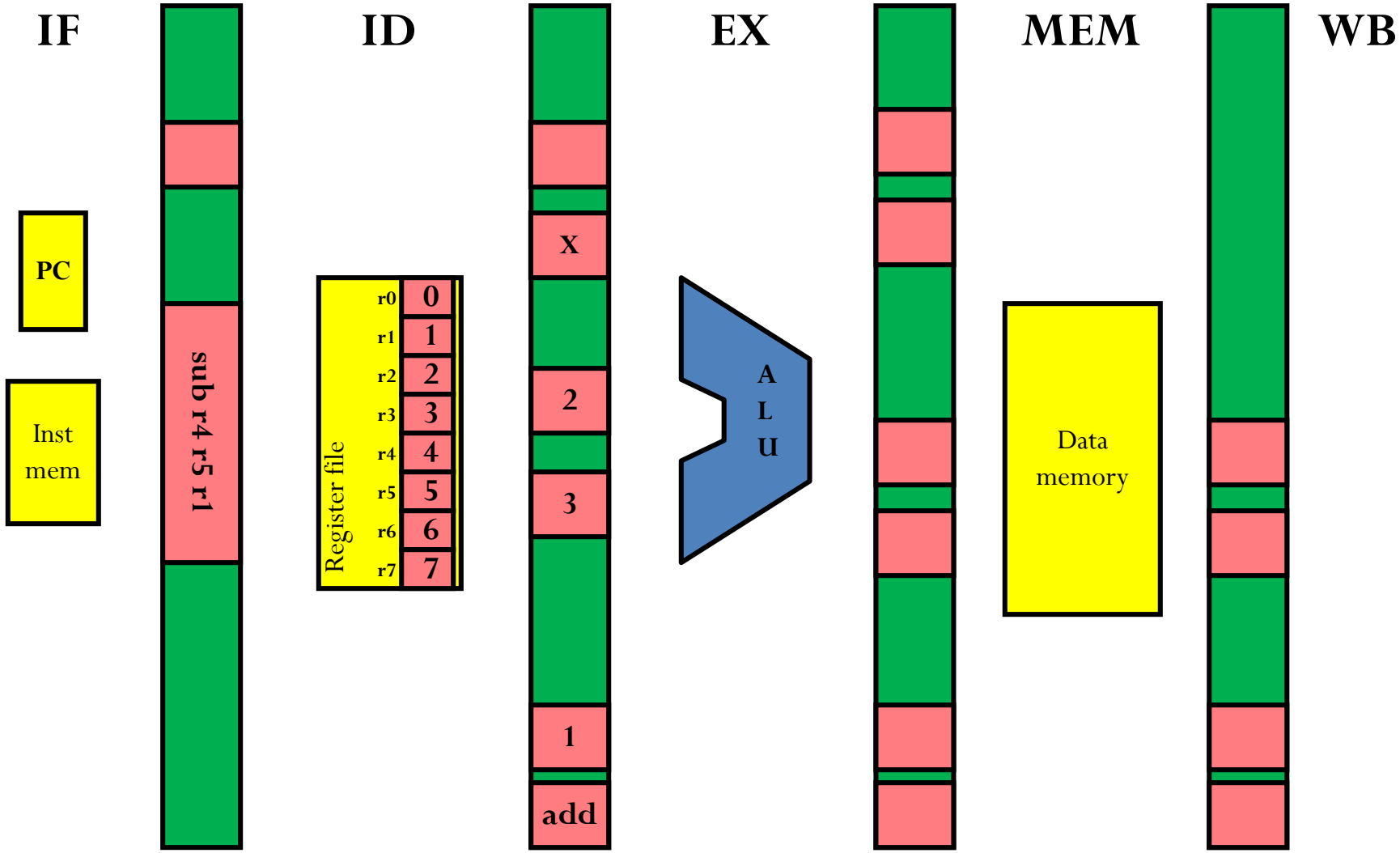
- CPI (ignore start up)
  - Without hazards: 1
  - With hazards: >1
- Cycle time:
  - Comparable to multiple cycle implementation
- Overall performance:
  - Better than single cycle or multiple cycle
  - Can be improved with enhancements

Improving Performance Further

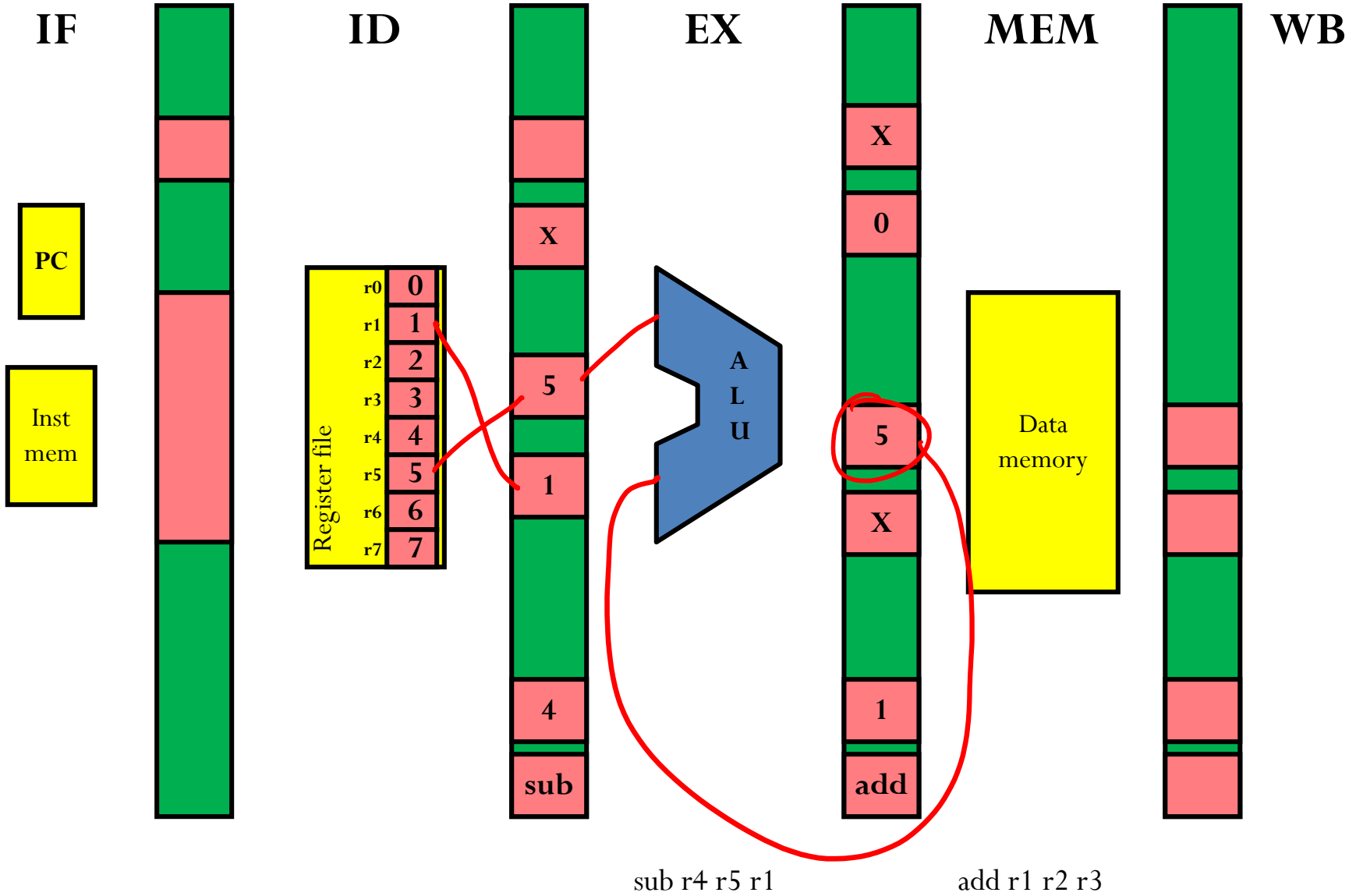
- Data forwarding
- Branch prediction
- Reordering instructions
- Parallel pipelines (superscalar)

Data Forwarding

- Observation: The value that will be written to the register is known at the EX stage for arithmetic and logic operations.
- **Data Forwarding:** Find the result of a previous instruction in a future pipeline register and forward it to the EX stage.
- Prevents stalling the pipeline for most (but not all) data hazards.
  - Result for `lw` not known until MEM stage.
- Requires more hardware but is worth it!

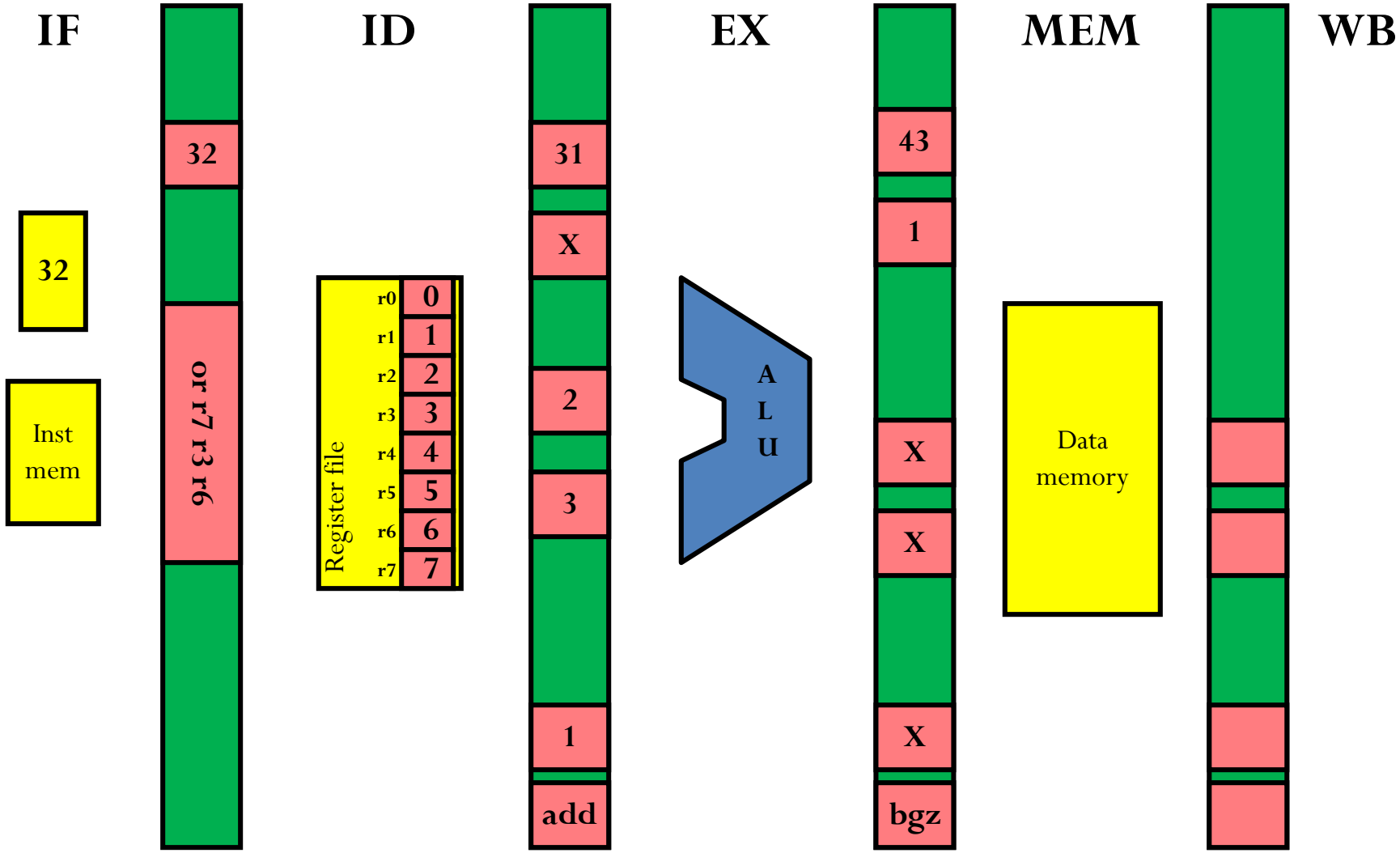


sub r4 r5 r1 add r1 r2 r3

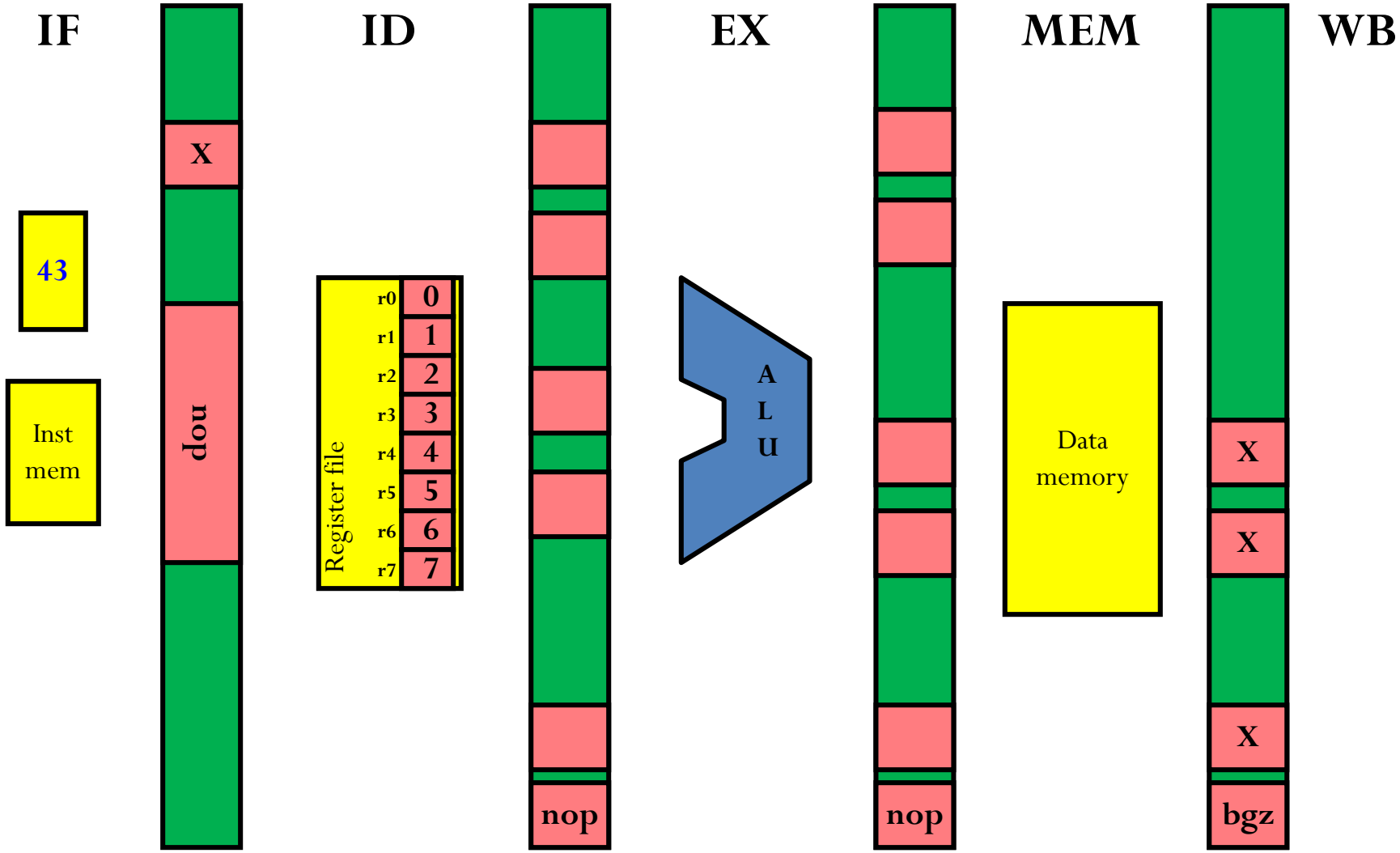

IF
ID
EX
MEM
WB
PC
Inst mem
Register file
r0
r1
r2
r3
r4
r5
r6
r7
0
1
2
3
4
5
6
7
X
5
1
4
sub
A
L
U
X
0
5
X
1
add
Data memory
sub r4 r5 r1
add r1 r2 r3

Branch Prediction

- 3 cycle delay is not necessary all the time.
- Simple prediction: assume branch is not taken.
  - Continue to populate pipeline with instructions at PC+1, PC+2, …
- If branch is not taken → no delay.
- If branch is taken:
  - Squash all instructions in pipeline.
  - Start fetching from target address next cycle.



or r7 r3 r6 add r1 r2 r3 bgz r4 13



lw r1 ~~1019 5~~

r1 r6 3

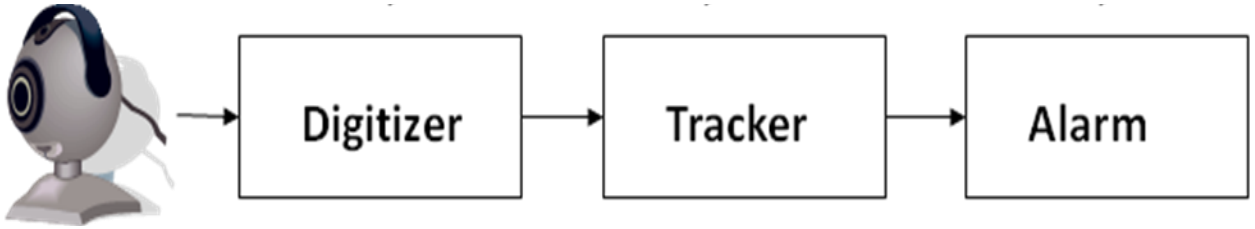
bgz r4 13

Summarizing Pipelining

- With data forwarding and branch prediction, only stall ANNA in the following cases:
  - A dependent instruction immediately follows a `lw` (stall for one cycle).
  - A branch is taken (squashes three instructions).
- In real processors:
  - Some instructions may take longer than one cycle (causing stalls). to execute
    - Examples: multiply, divide, floating point arithmetic
  - Additional stalls due to cache misses.
  - Need to deal with I/O.

Pipelining as a Computing Concept

- Pipelining is not limited to microprocessors.
- Pipelining can be applied to computing systems.
  - Useful in cases where a calculation is broken into phases.
  - Form of parallelism.
- Example: Real-time surveillance camera:



Outline

- Introduction to Microarchitecture
- ANNA Datapath
- ANNA Control
- Performance
- Pipelining
- **Additional Performance Optimizations**

Better Branch Prediction

- Can do more sophisticated predictions:
  - Predict taken (~60% branches are taken)
  - Predict the same as last time
  - Predict based on pattern (loops)

- Modern branch predictors are correct over 90% of the time.

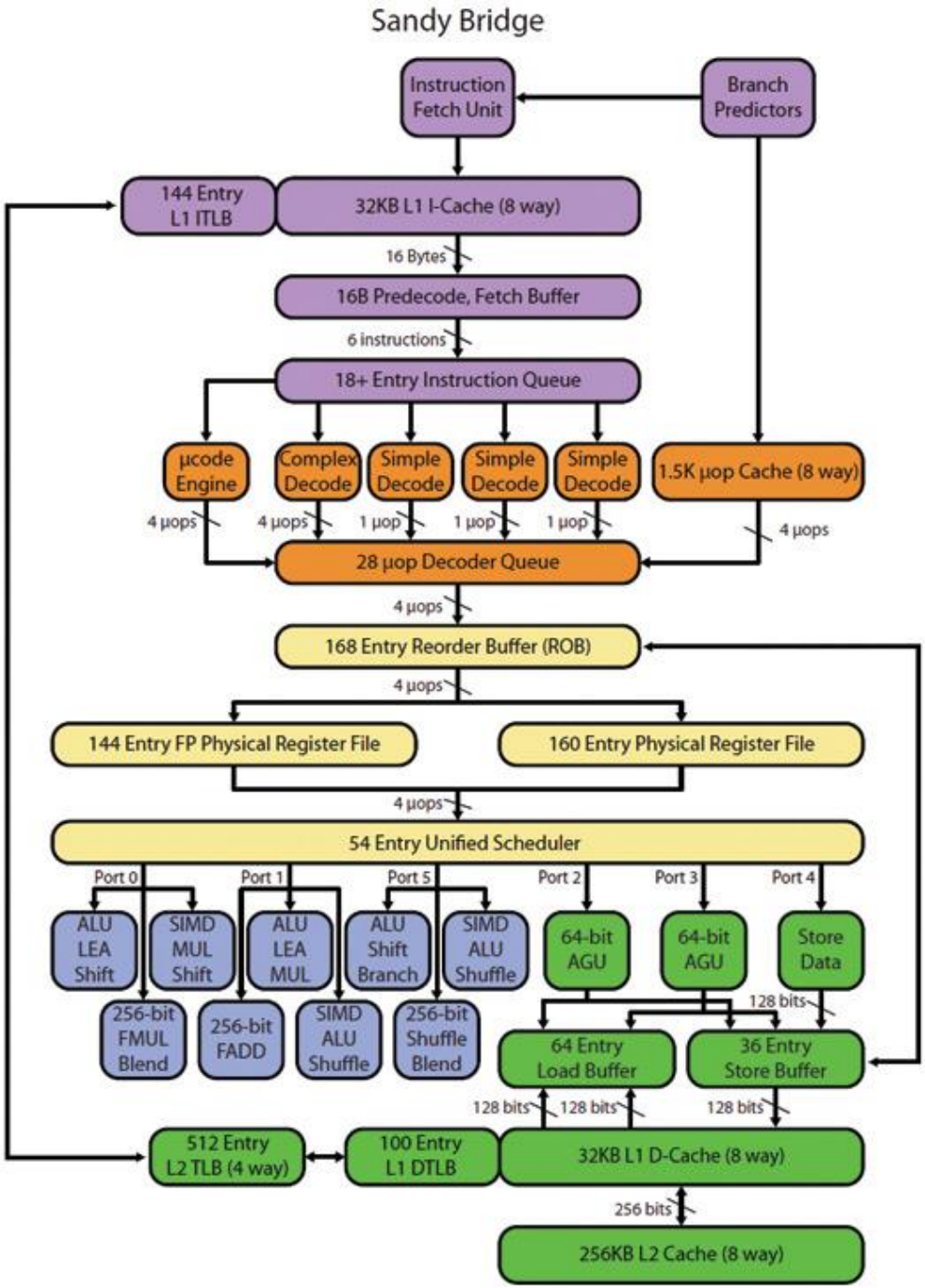
Reordering Instructions

- In order to avoid data hazards, it may be possible to reorder instructions.
- Goals:
  - Move dependent instructions further away.
  - Cannot violate program functionality.
- Especially helpful for slow instructions:
  - loads / store (if not in cache)
  - multiply / divide / FP operations
- Reordering can be done by the compiler or inside the CPU.

Parallel Pipelines (Superscalar)

- If you max out the performance of the pipeline and still want more, what can you do?
  - Build a second pipeline
- Two or more instructions can be executed at the same time.
  - Must be independent.
- It is possible to forward data between the two pipelines.
- Lower level parallelism than multiple cores.

Intel Core Microarchitecture



Thank You!