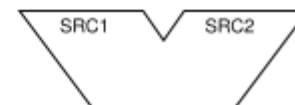
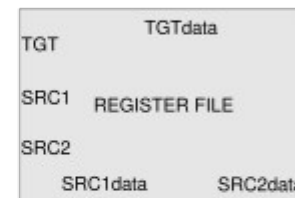
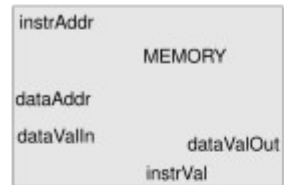


# E20 single-cycle implementation

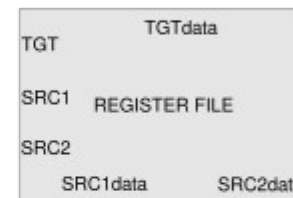
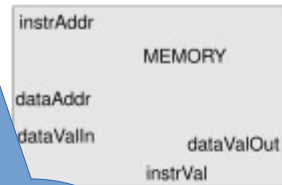
Program Counter



We start with the basic components: memory, registers, ALU, and program counter.

Each component has labeled *ports*, allowing it to accept input values or emit output values.

Program Counter



The **program counter** stores the 16-bit address of the current instruction.

Its output is the current address. Writing to this register will determine the address of the subsequent instruction.

It stores the values of instructions and *data*.

Access on `dataAddr` or `instrAddr`, and the corresponding value will be output on `dataValOut` or `instrVal`, respectively.

To write a cell, give the address on `dataAddr`, and the new value on `dataValIn`, and set the write-enable to true.

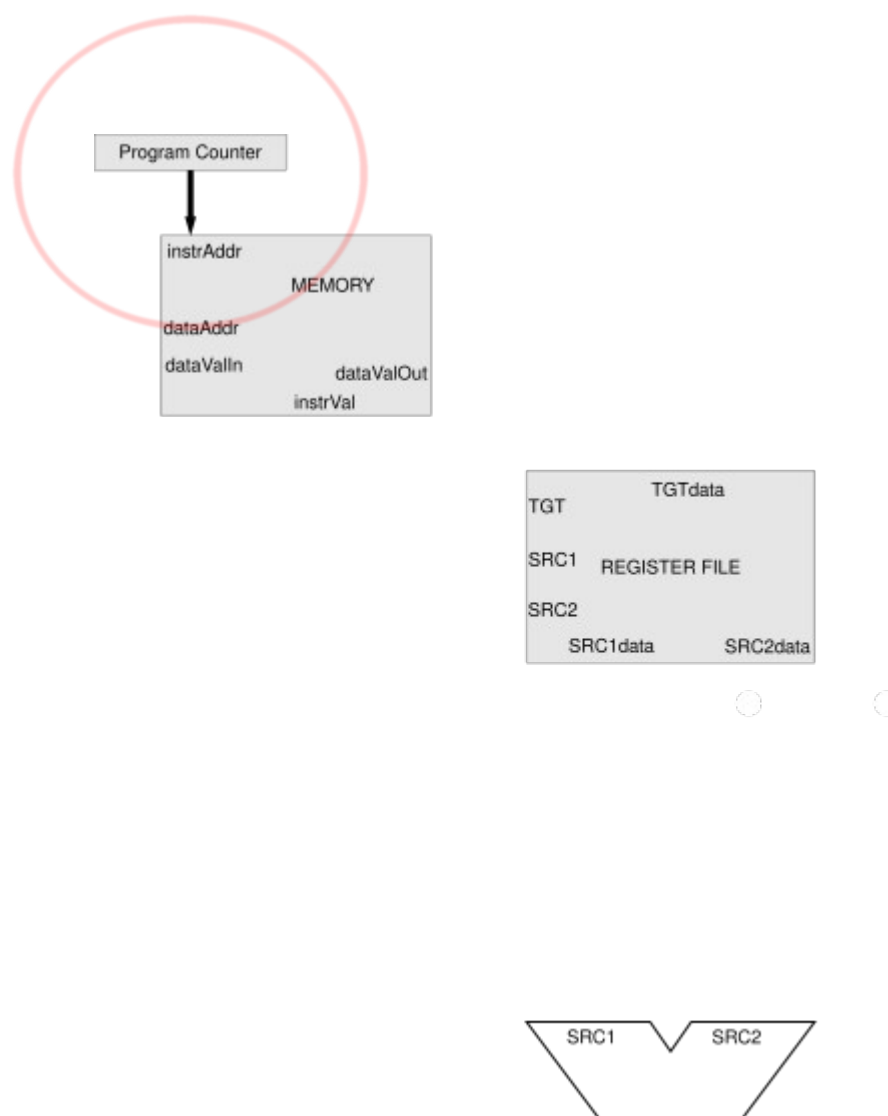
Each component has labeled ports and values.

It takes inputs from registers \$2, \$3, \$4, \$5, \$6, \$7, \$8, \$9, \$10, \$11, \$12, \$13, \$14, \$15, \$16, \$17, \$18, \$19, \$20, \$21, \$22, \$23, \$24, \$25, \$26, \$27, \$28, \$29, \$30, \$31. It outputs the value of the register on input `SRC1` or `SRC2` on `SRC1data` or `SRC2data`, respectively.

It takes inputs from registers \$2, \$3, \$4, \$5, \$6, \$7, \$8, \$9, \$10, \$11, \$12, \$13, \$14, \$15, \$16, \$17, \$18, \$19, \$20, \$21, \$22, \$23, \$24, \$25, \$26, \$27, \$28, \$29, \$30, \$31. It outputs the value of the register on input `SRC1` or `SRC2` on `SRC1data` or `SRC2data`, respectively.

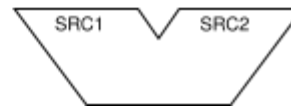
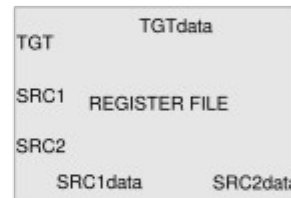
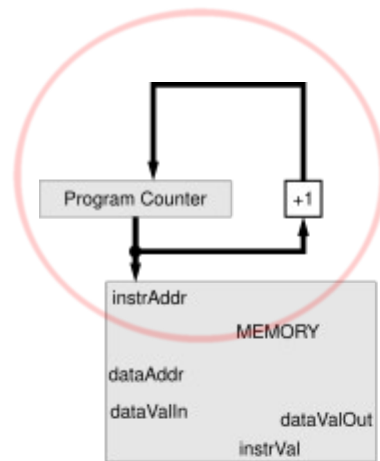
To write a cell, give the address on `dataAddr`, and the new value on `dataValIn`, and set the write-enable to true.

to true.



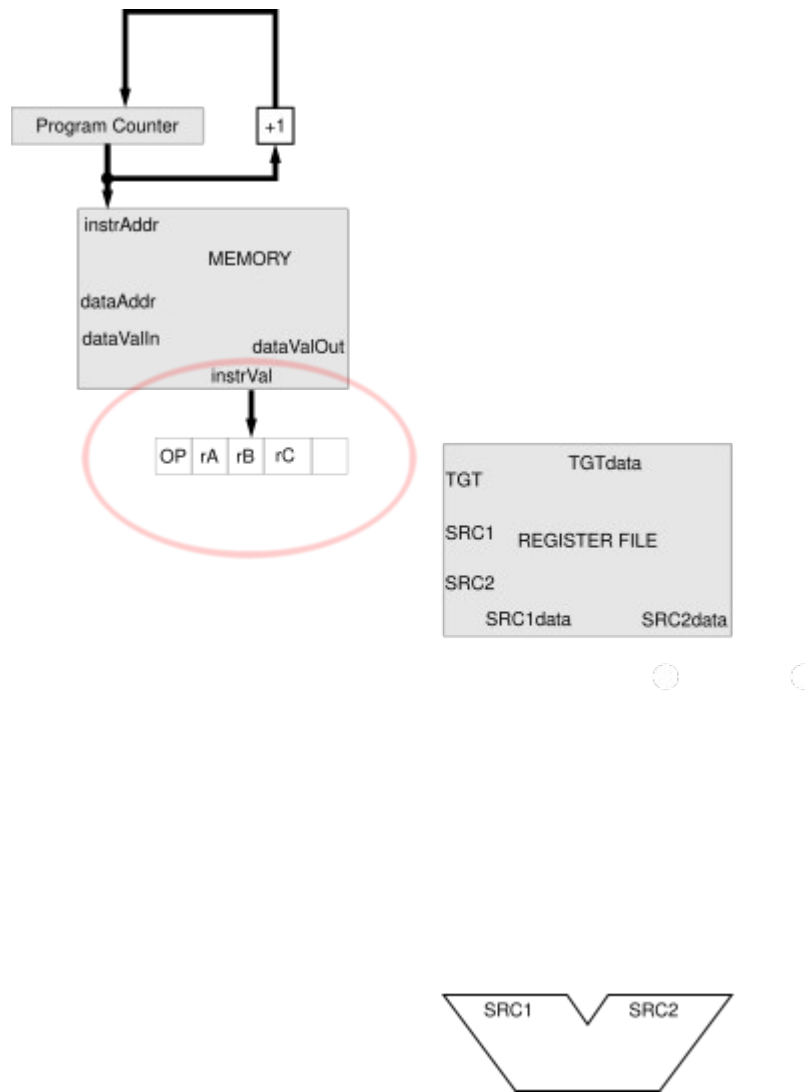
The program counter is used to index into instruction memory to retrieve the current instruction. We thus connect these components with a wire.

The `instrAddr` port on memory is where it receives the address of the current instruction.

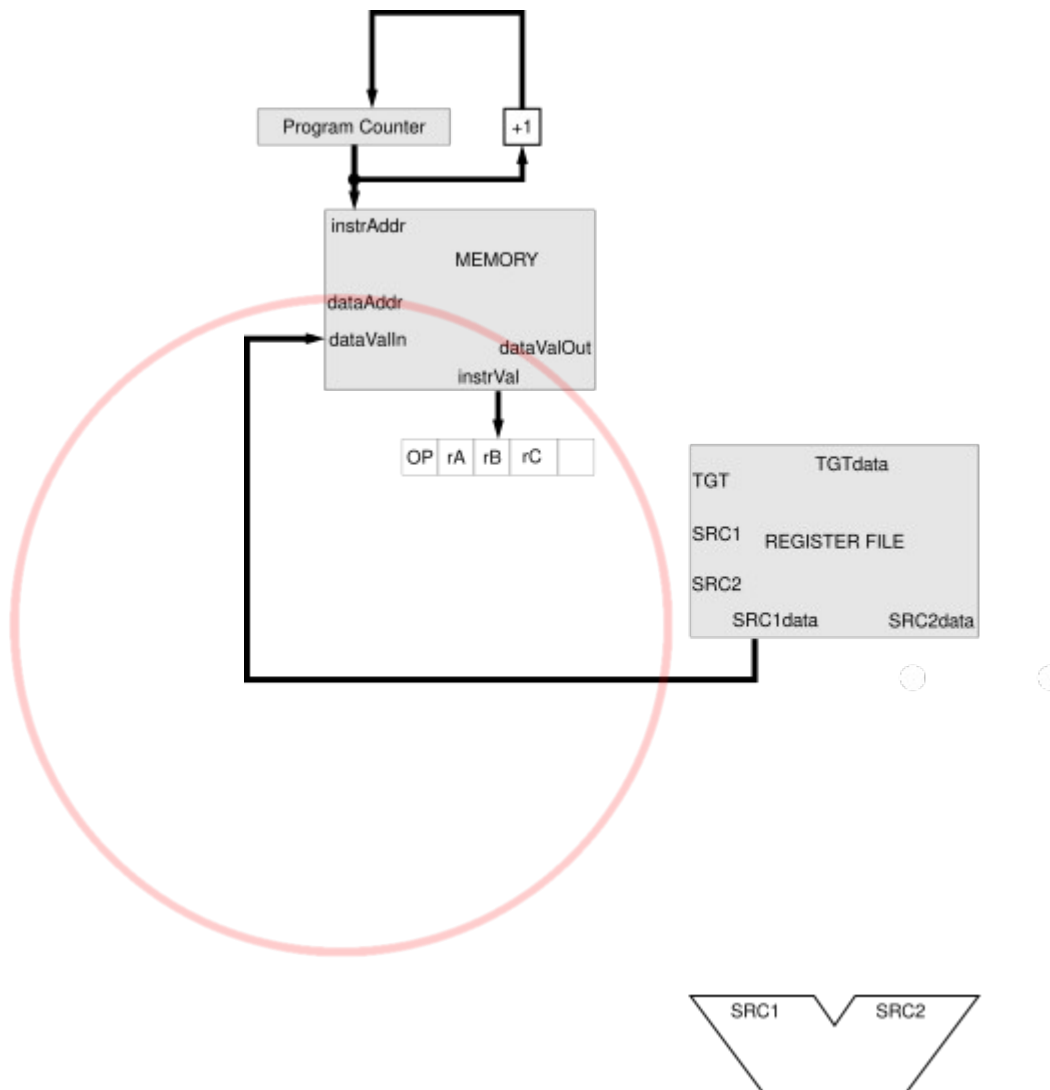


We'll need a way to increment the program counter. So the value read from the current program counter is passed to an adder (here marked +1), which is stored back into the program counter at the end of the clock cycle.

This configuration does not account for jumps; we'll deal with that later.

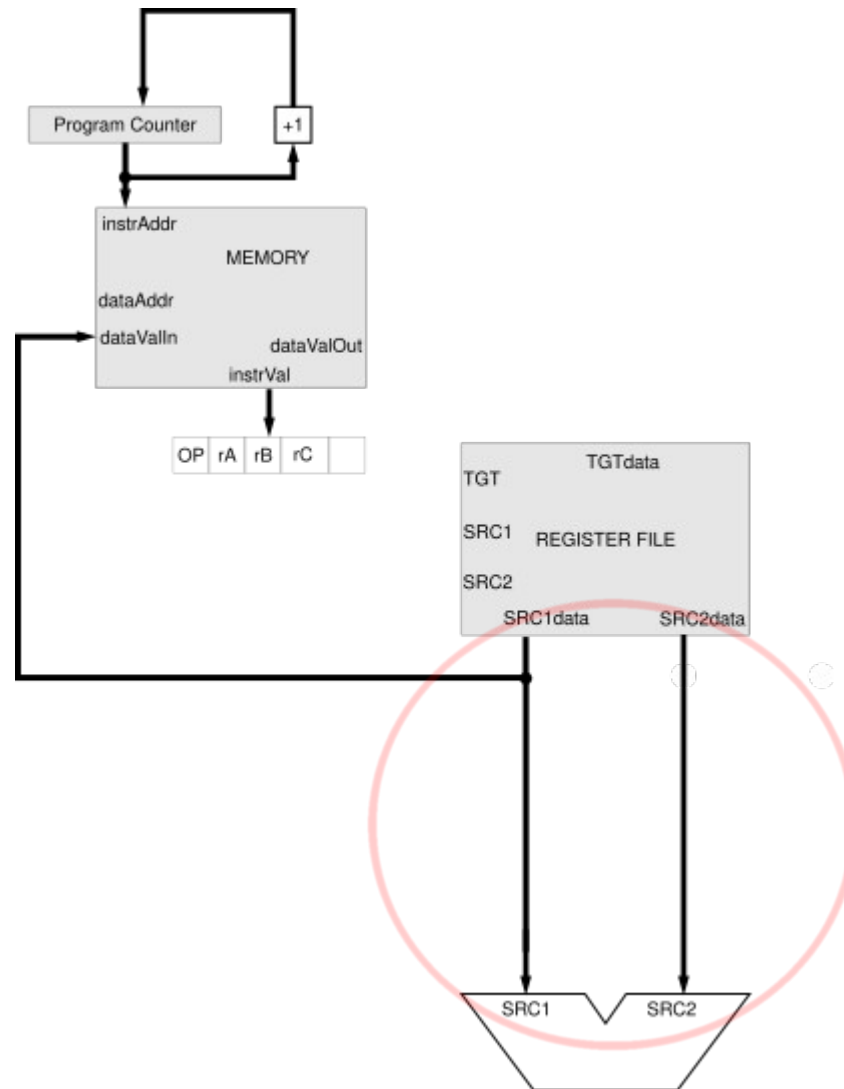


The instruction we retrieve from memory can be further analyzed as consisting of *fields*. Here, we break down the instruction into its opcode and several register fields. Different instructions will be divided in different ways.



The SRC1data output of the register file gives us data read from the value of a register. That value may be used as input to memory, specifically for the case of the **sw** instruction, where we store a register value into a memory cell.

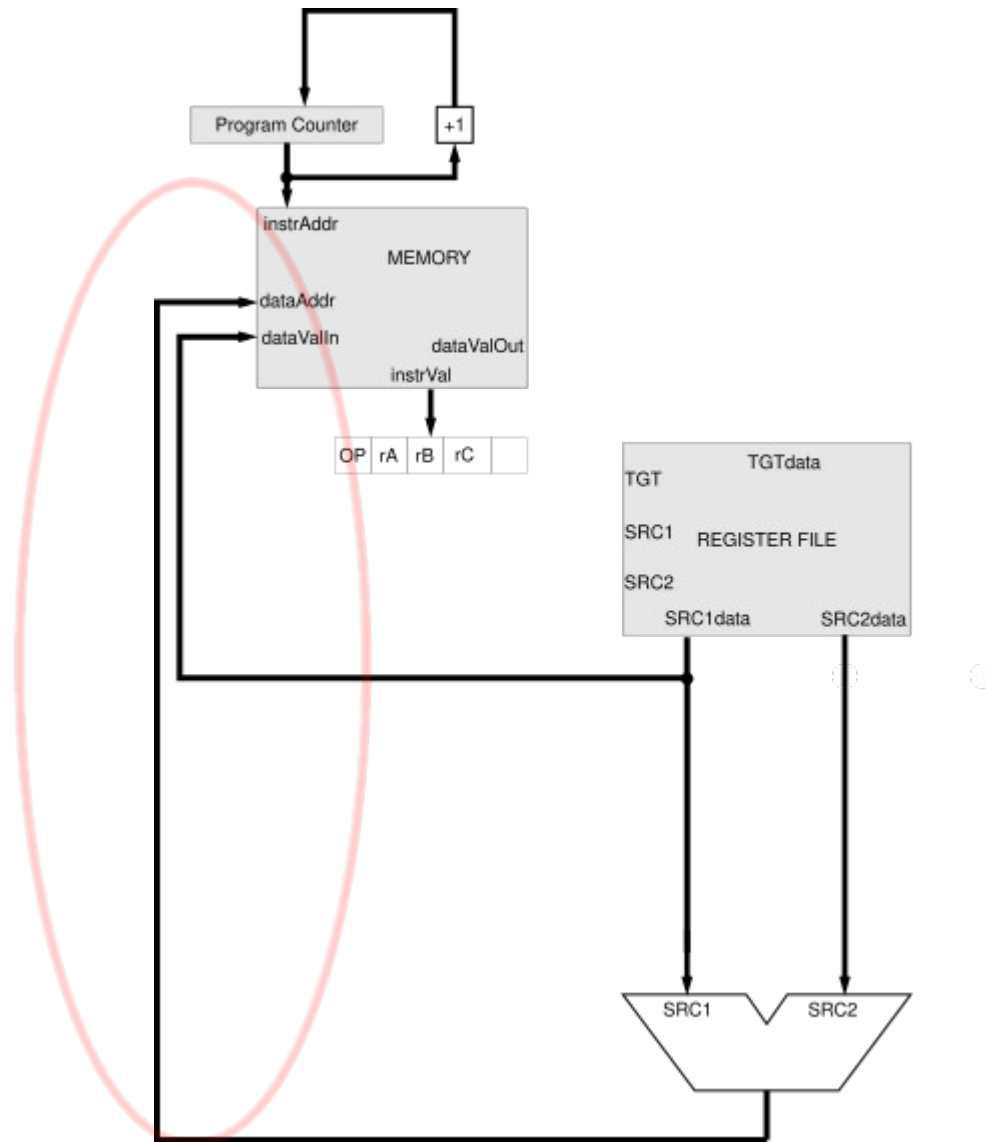
The dataValIn port receives the value to store into memory.



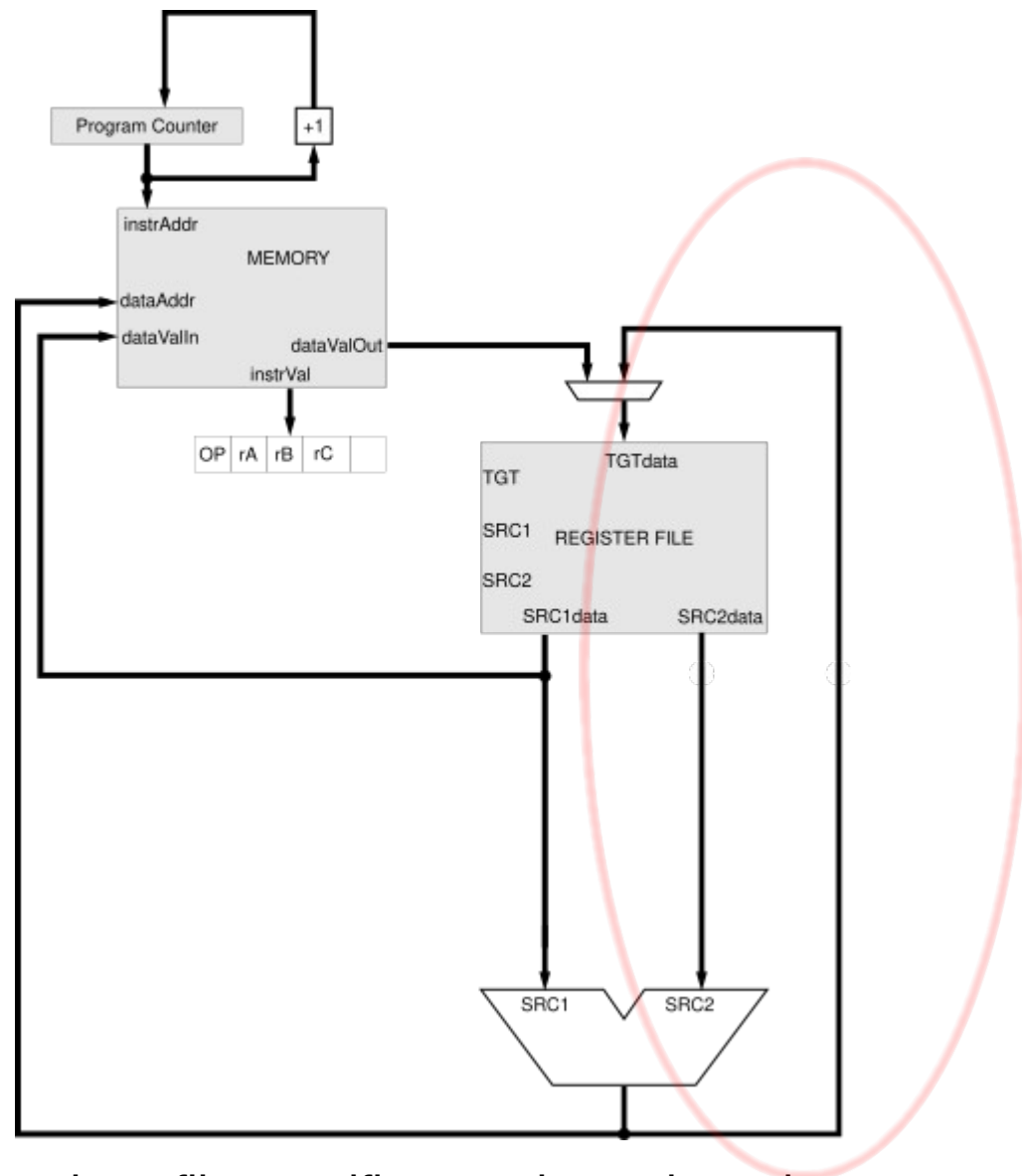
Often, the values from registers will be sent to the ALU, where they will be added or subtracted or ANDed or ORed. Thus both SRC1data and SRC2data are inputs to the ALU.

This is used in the case of an instruction like **add**, **sub**, **and**, **or**.



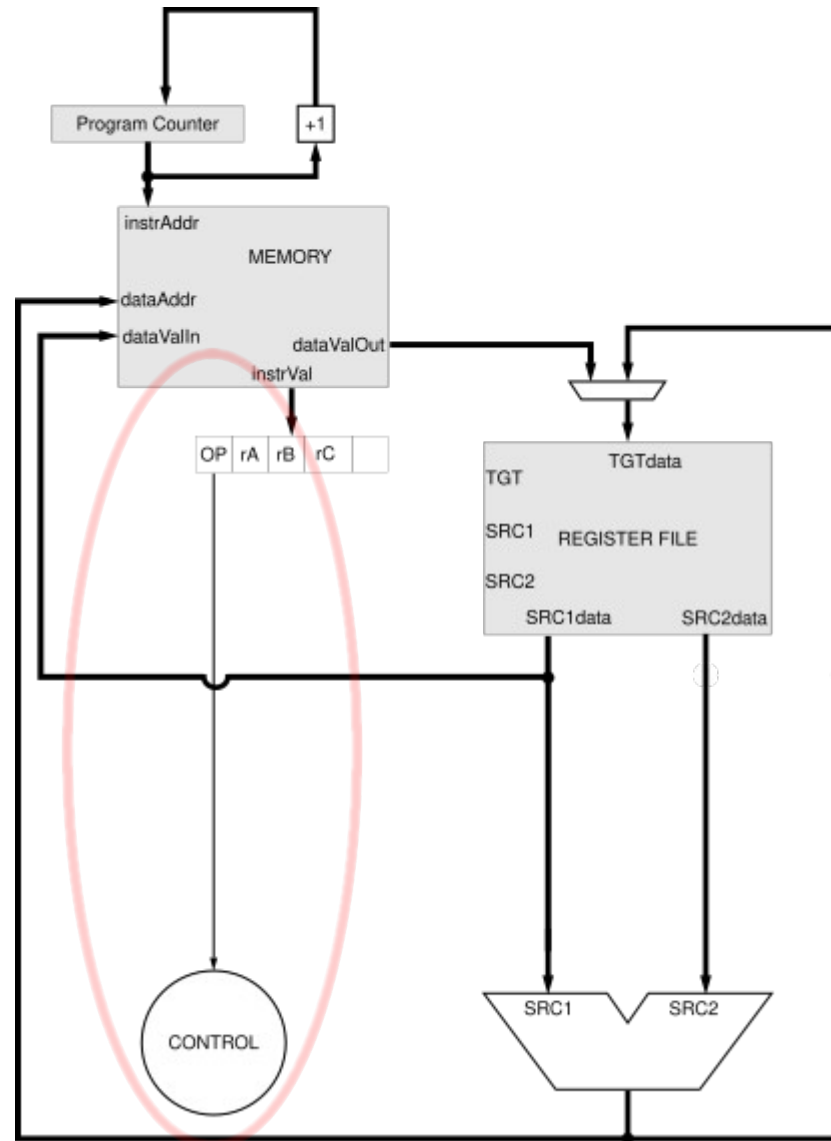


The output of the ALU can be connected to the **dataAddr** input to the memory. The **dataAddr** input specifies the address to be read or written. We'll use this wire for **sw** and **lw**, where the sum of a register and an immediate specifies a pointer.



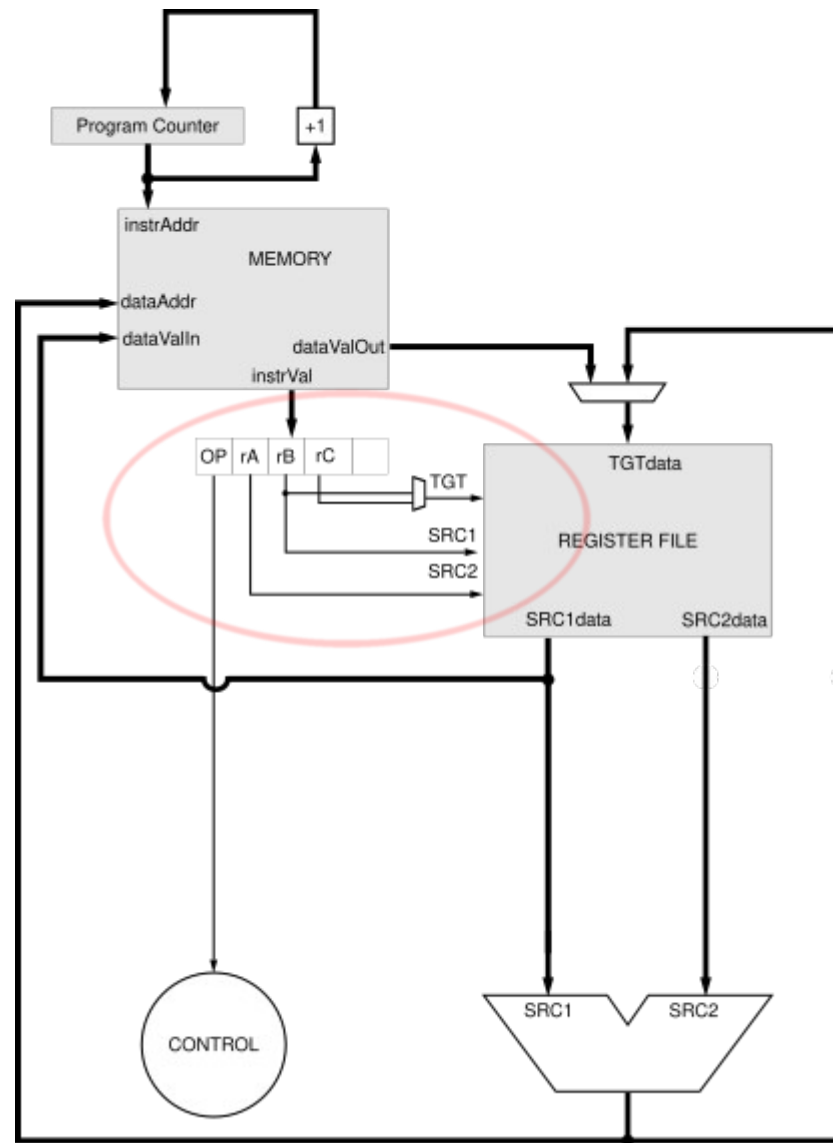
The TGTdata input to the register file specifies a value to be written to a register.

Here, we recognize two sources of data: the output of the ALU (for **add**, **sub**, etc) and the output of data memory (for **lw**). The two options are selected by a mux. For now, we're omitting the select input for this mux.



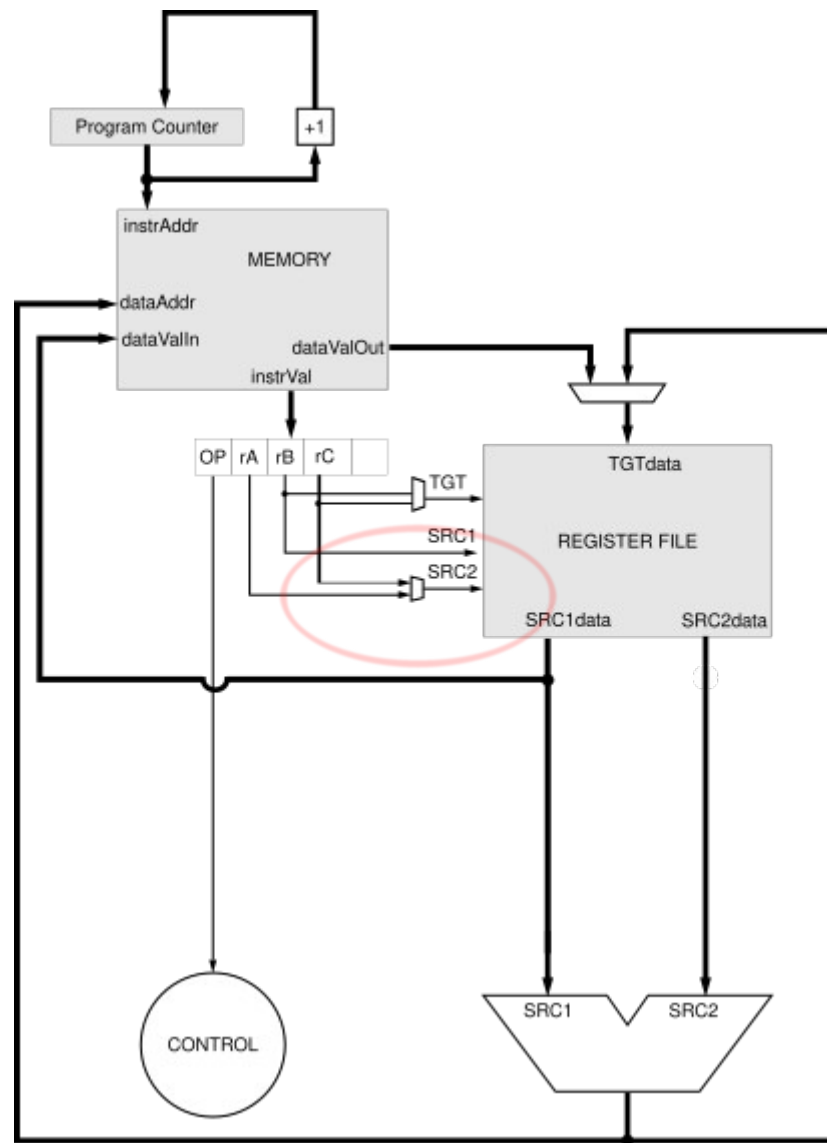
Since we have a mux, some component has to decide which input of the mux to use.

The *control module* sets inputs for the muxes in our processor. We'll eventually connect it to them. For now, know that the control module's input is the opcode of the current instruction.

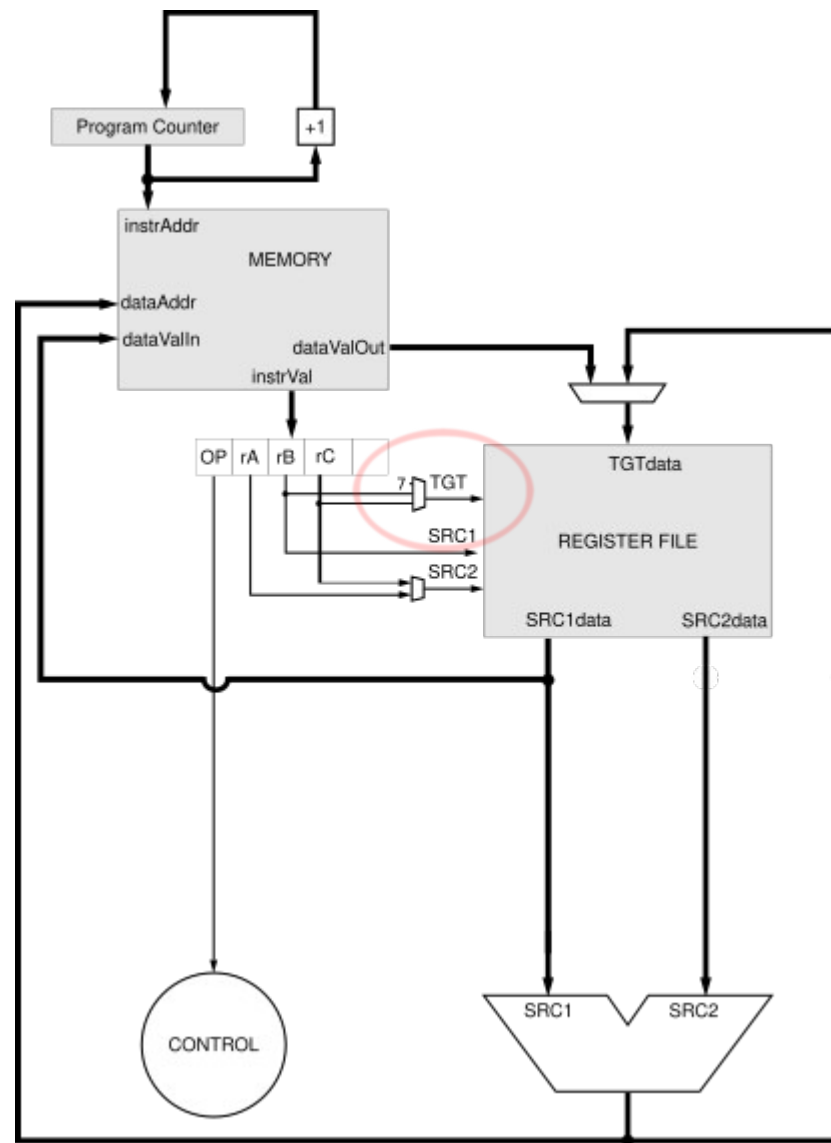


The other fields of the current instruction can be applied as inputs to the register file. Here, the register in the **rA** field is passed to the **SRC2** input, specifying the name of the register to read; its value will be output as **SRC2data**. Similarly, **rB** is used as **SRC1data**.

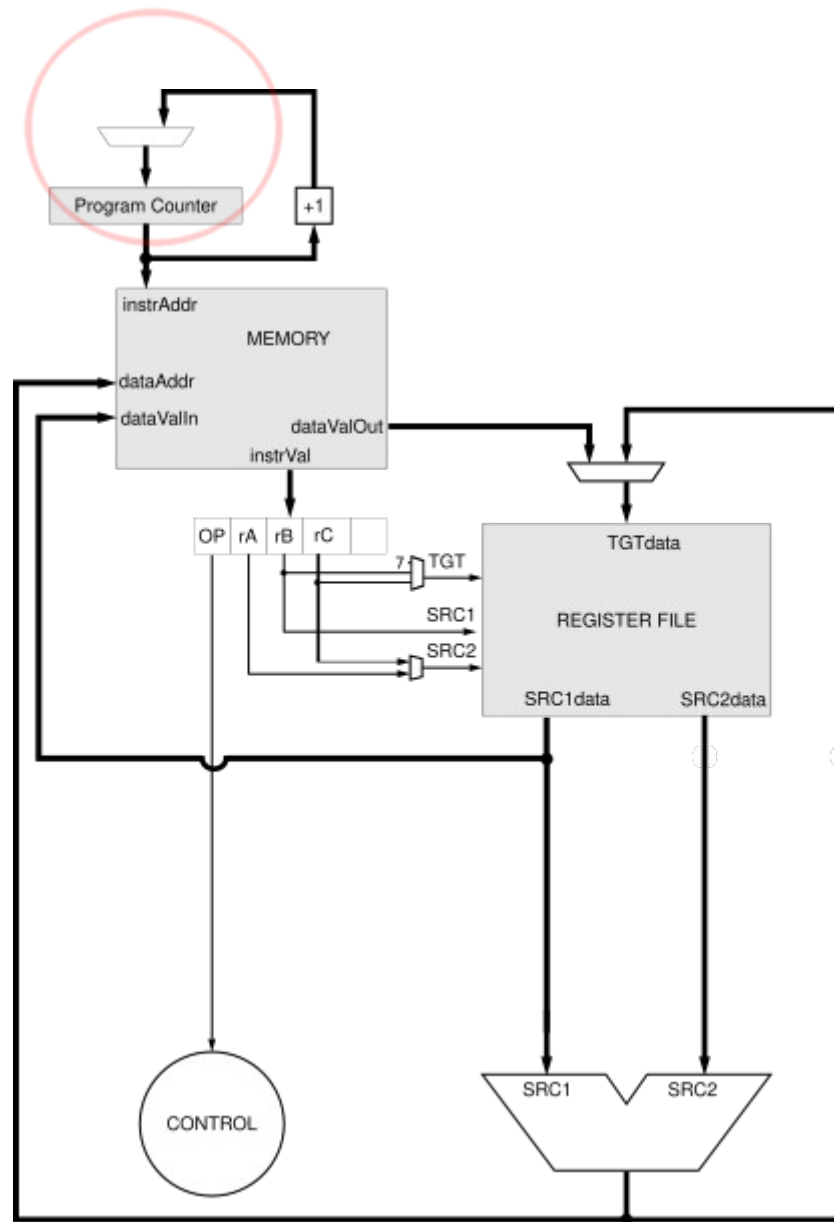
The **TGT** input gives the name of a register to write to. We use a mux to select between **rB** (for **lw**, **slti**, **addi**) and **rC** (for **add**, **sub**, **slt**).



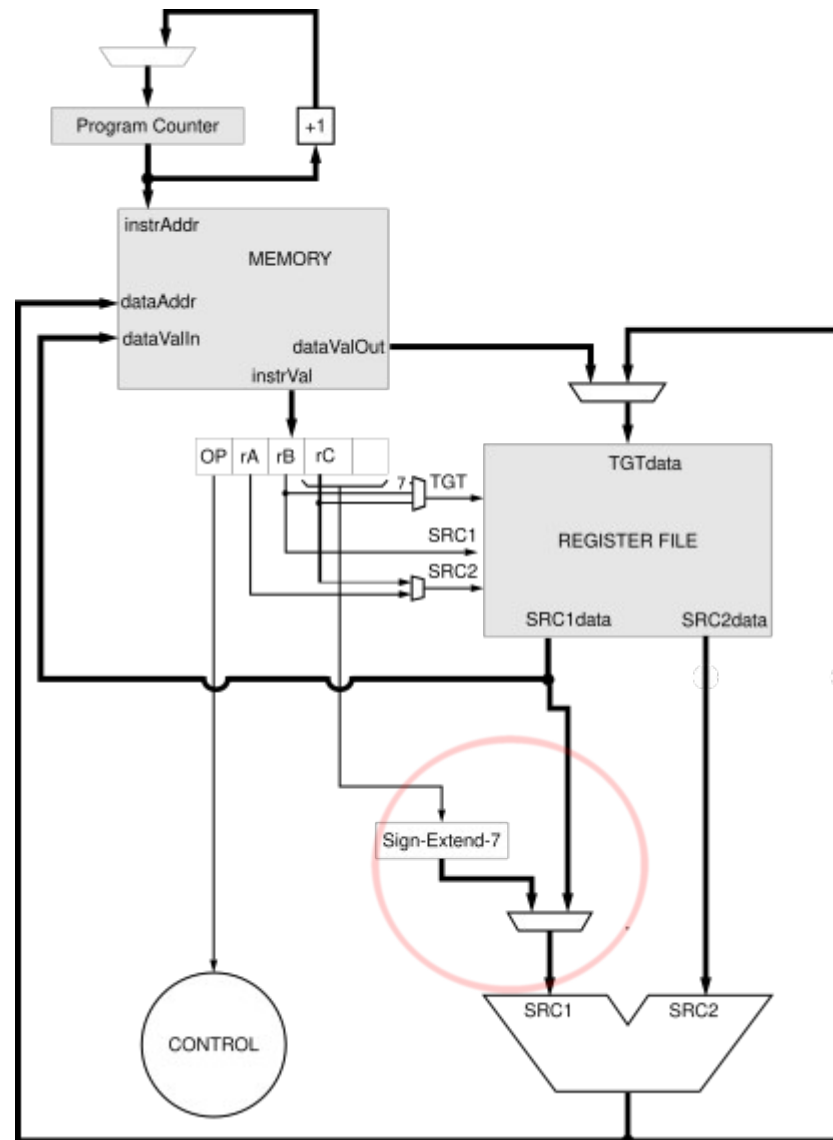
Actually, the rC field can also be used to as a SRC2 input.  
We don't use this yet, but we will in the future.



Also, for the **jal** instruction, the register to be written will always be 7. Therefore, we provide 7 as a fixed input as an additional input to the mux.

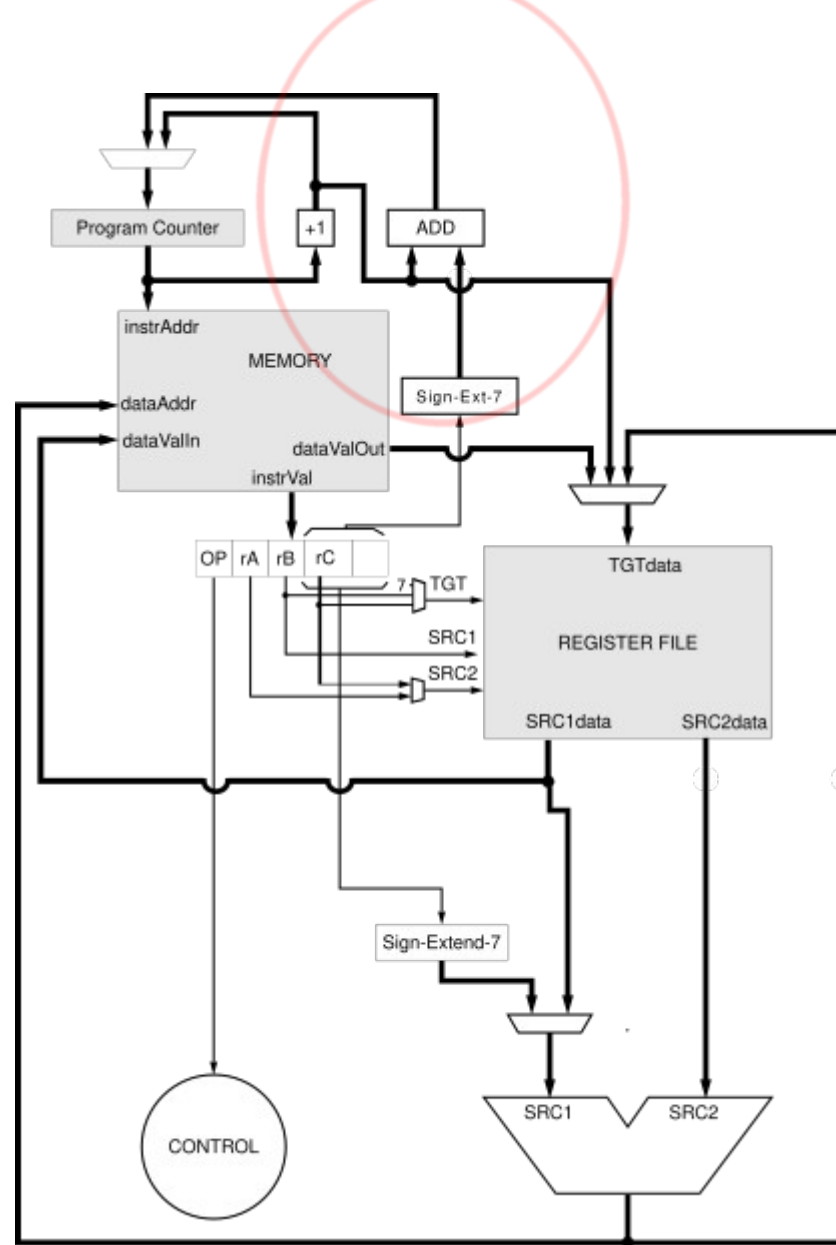


We still haven't dealt with jumps. At the very least, we'll need a mux to select from different options. Here we add the mux.

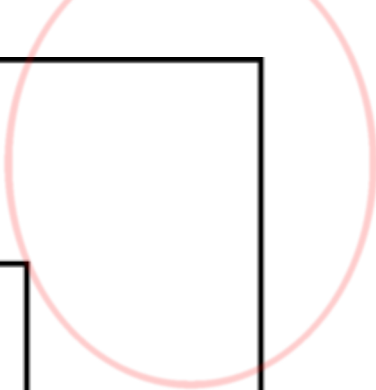


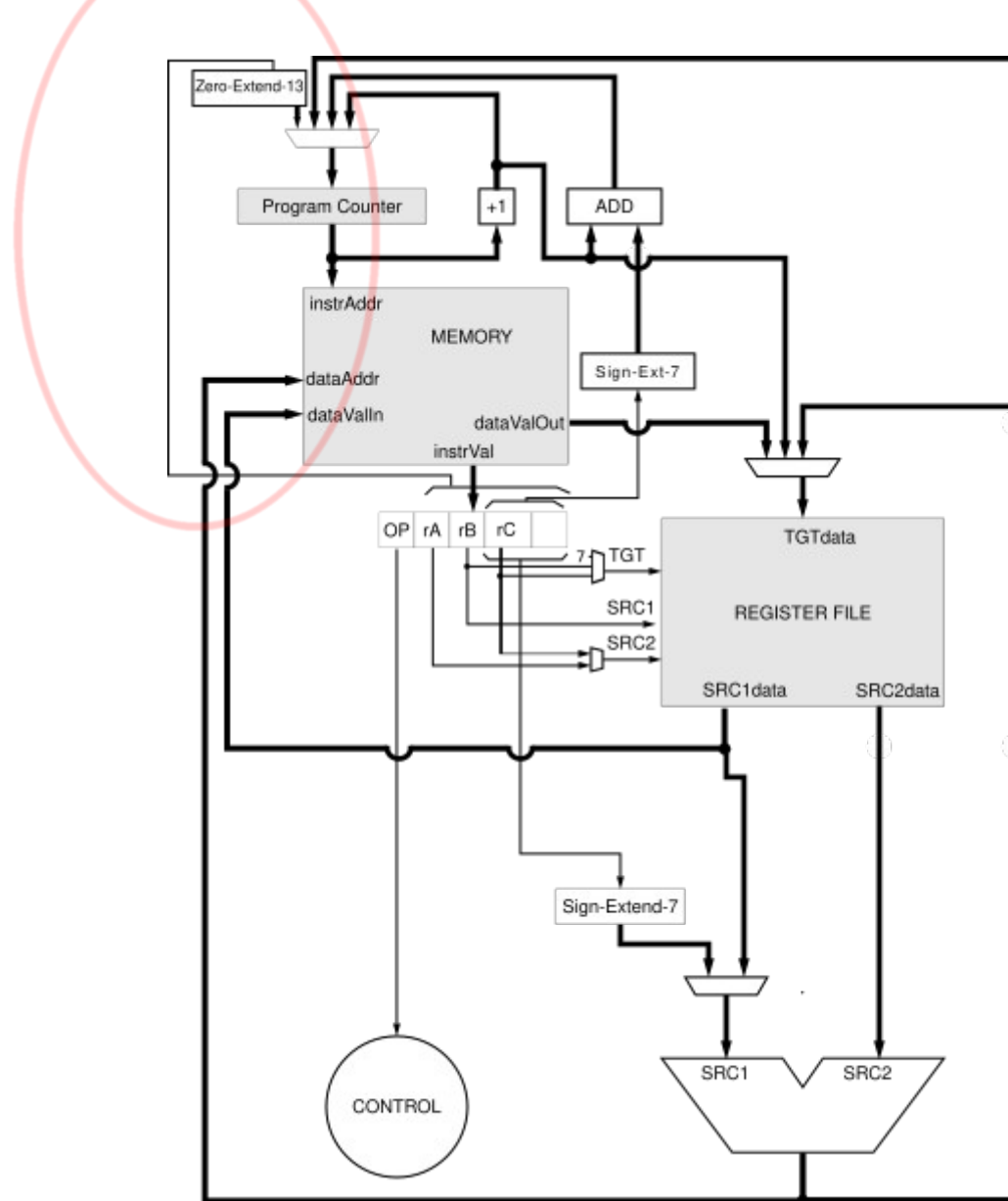
Here, we deal with 7-bit immediate values from the current instruction. This will be used for immediate arithmetic (like **addi**, **slli**). We sign-extend the 7-bit value and pass the result into the ALU. We add a mux to select input from either the immediate field or from the register file.



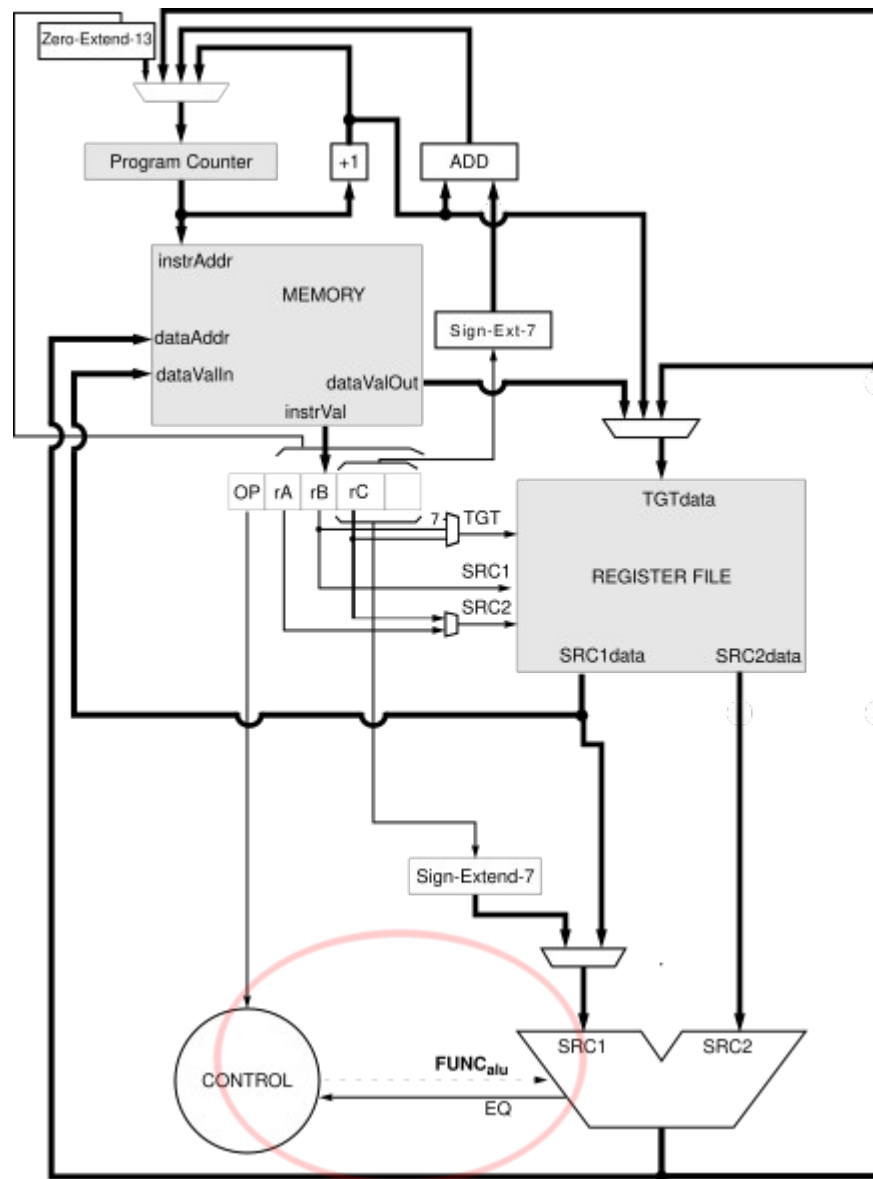


However, the 7-bit immediate will not always go to the ALU. For the **jeq** instruction, the 7-bit immediate will be sign-extended and added to the current program counter. The result will be stored into the program counter when the comparison is true. So we send the immediate to an adder, and the adder's result goes to the program counter's mux.

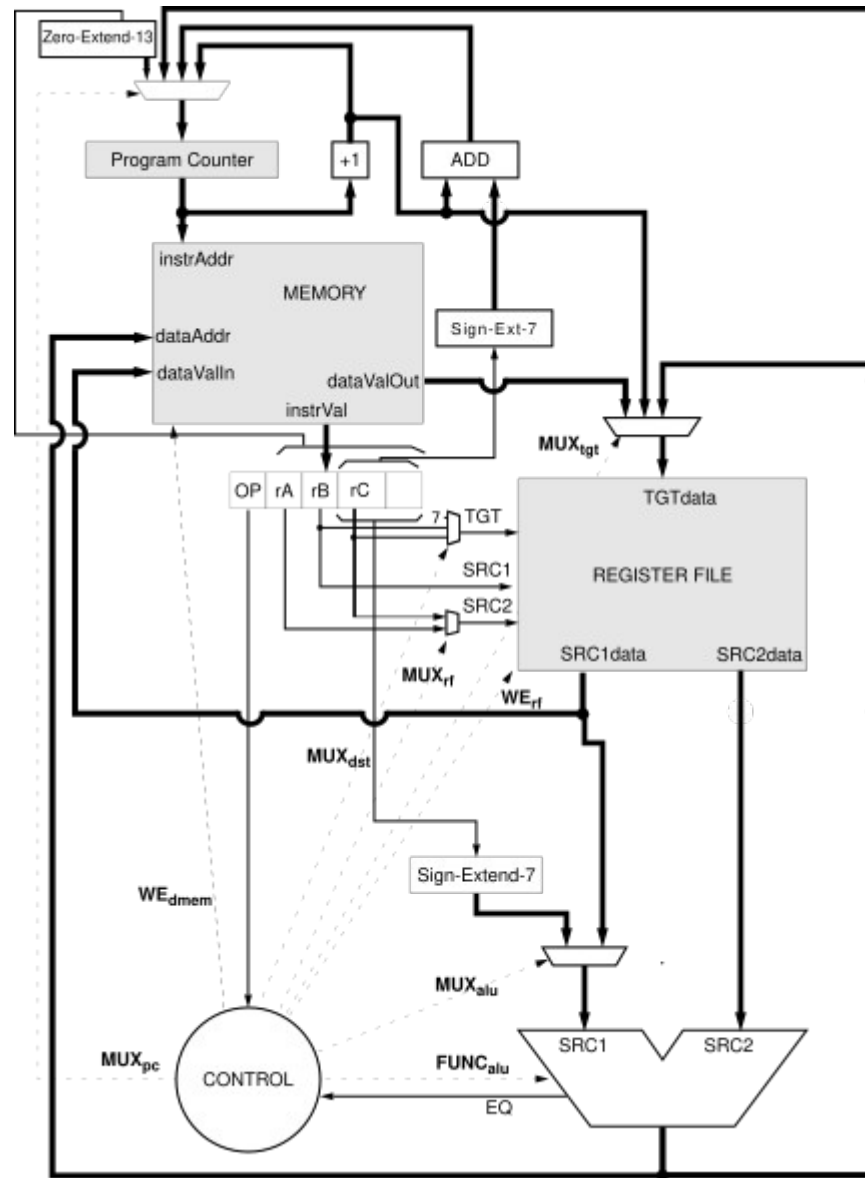




Yet another possibility is the **j** and **jal** instructions, which jump to a 13-bit immediate, as an absolute address. That immediate value is zero-extended and passed directly into the program counter's mux, bypassing the ALU.



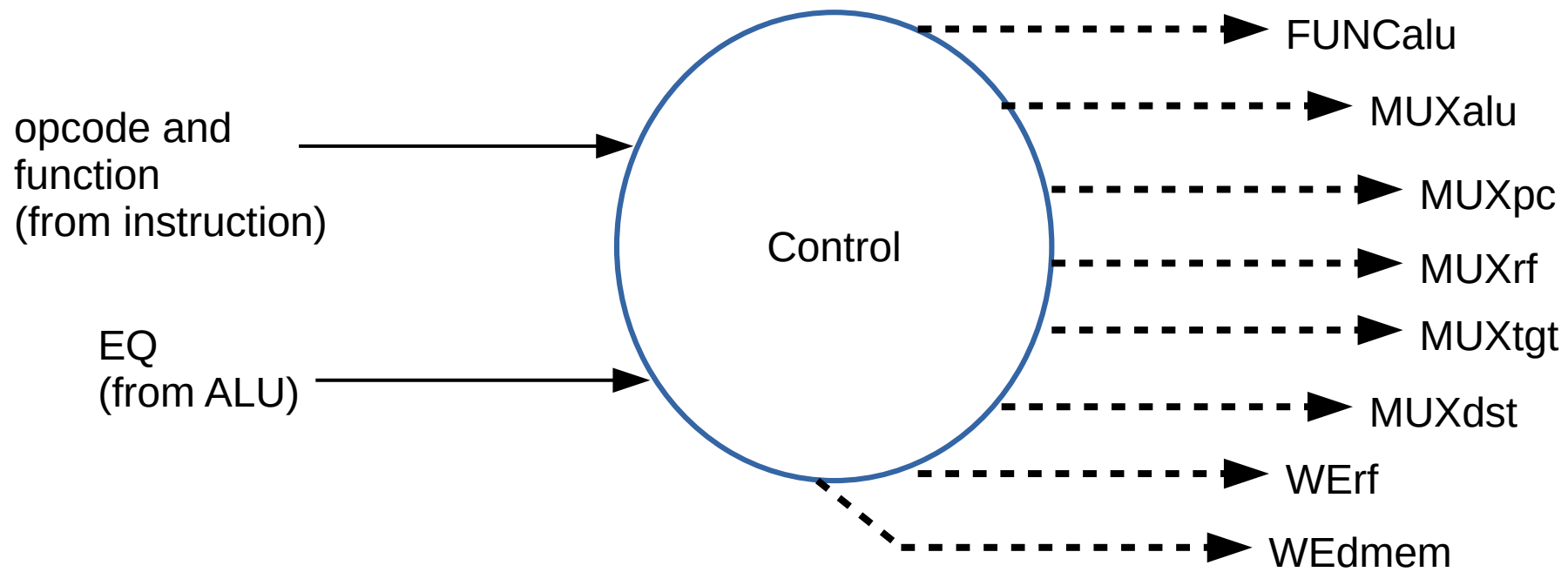
In the case of **jeq**, how does the program counter's mux know what to do? The mux must select between one of several options, depending not only on the opcode, but also on the result of the comparison. For this, we add two wires: the **FUNC<sub>alu</sub>** wire tells the ALU which operation, depending on the opcode; and the **EQ** wire tells the control module if the two ALU inputs are equal.



Finally, we can add *control lines*, connecting the control module to all the muxes.

This is our final E20 single-cycle processor.

# E20 control module implementation



Control module truth table

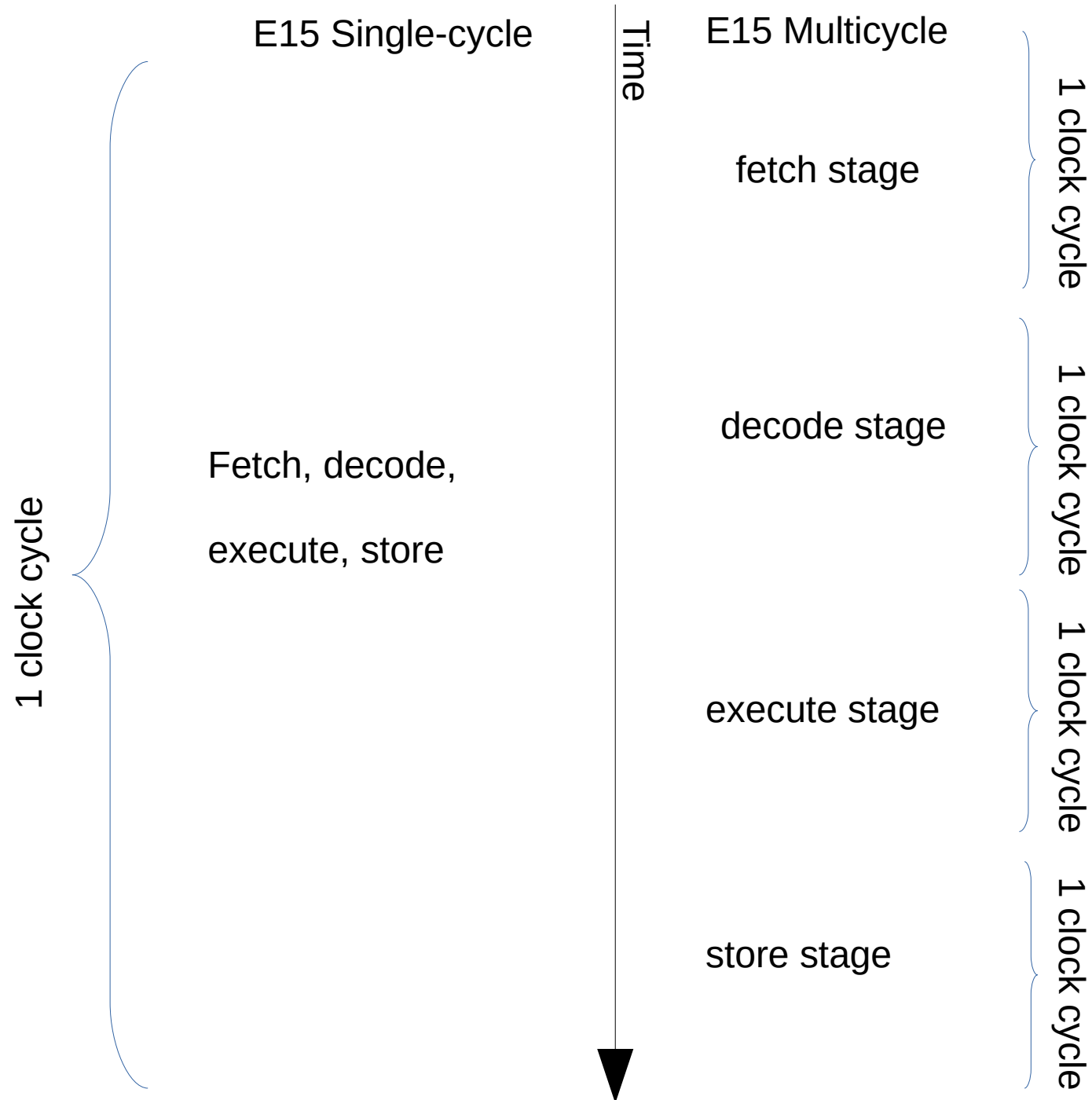
Inputs		Outputs							
opcode & func	EQ	FUNCalu	MUXalu	MUXpc	MUXrf	MUXtgt	MUXdst	WErf	WEdmem
add	DC	0	0	1	0	0	1	1	0
sub	DC	1	0	1	0	0	1	1	0
etc									
etc									

DC = don't care

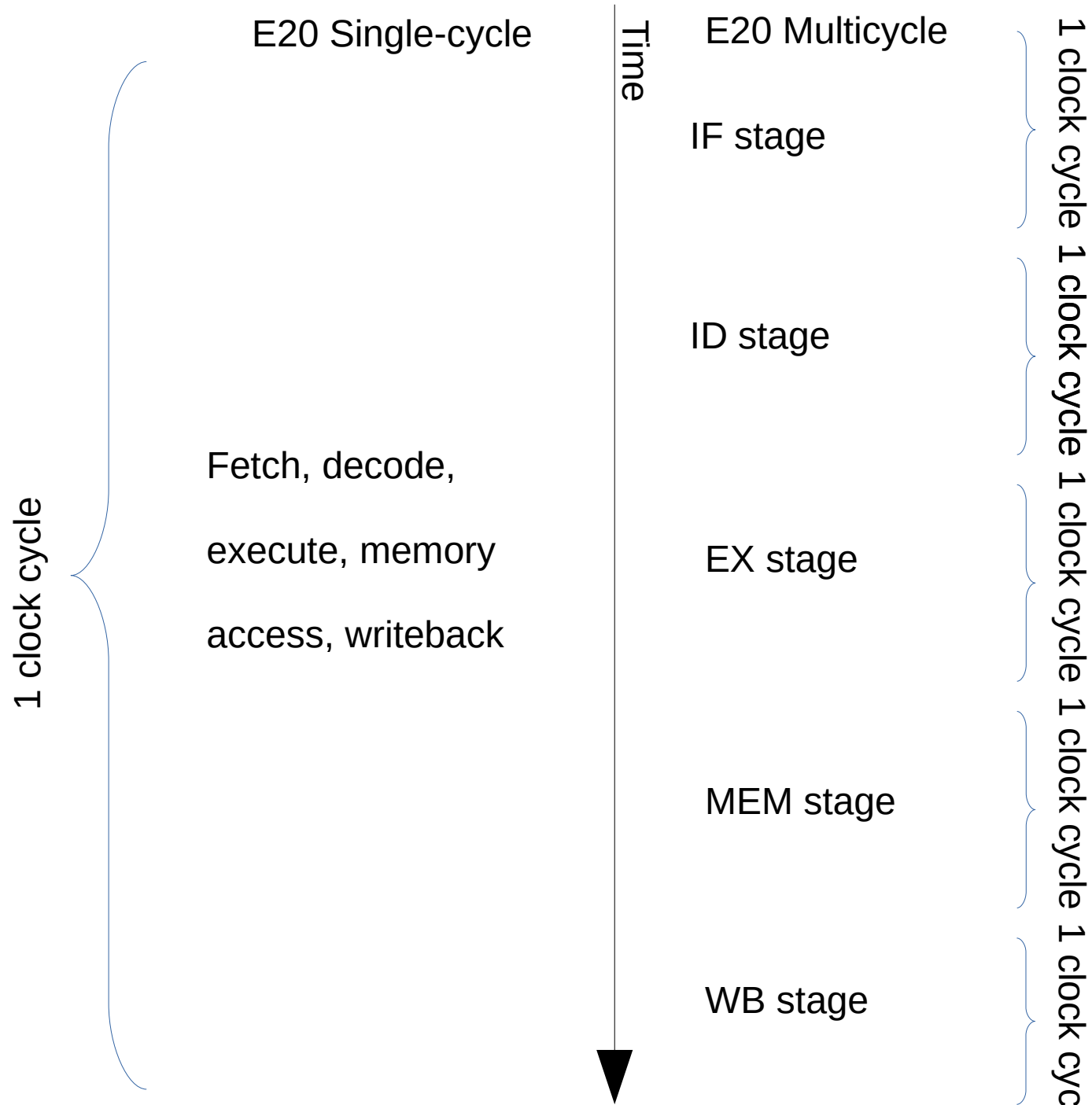


# E20 multicycle implementation

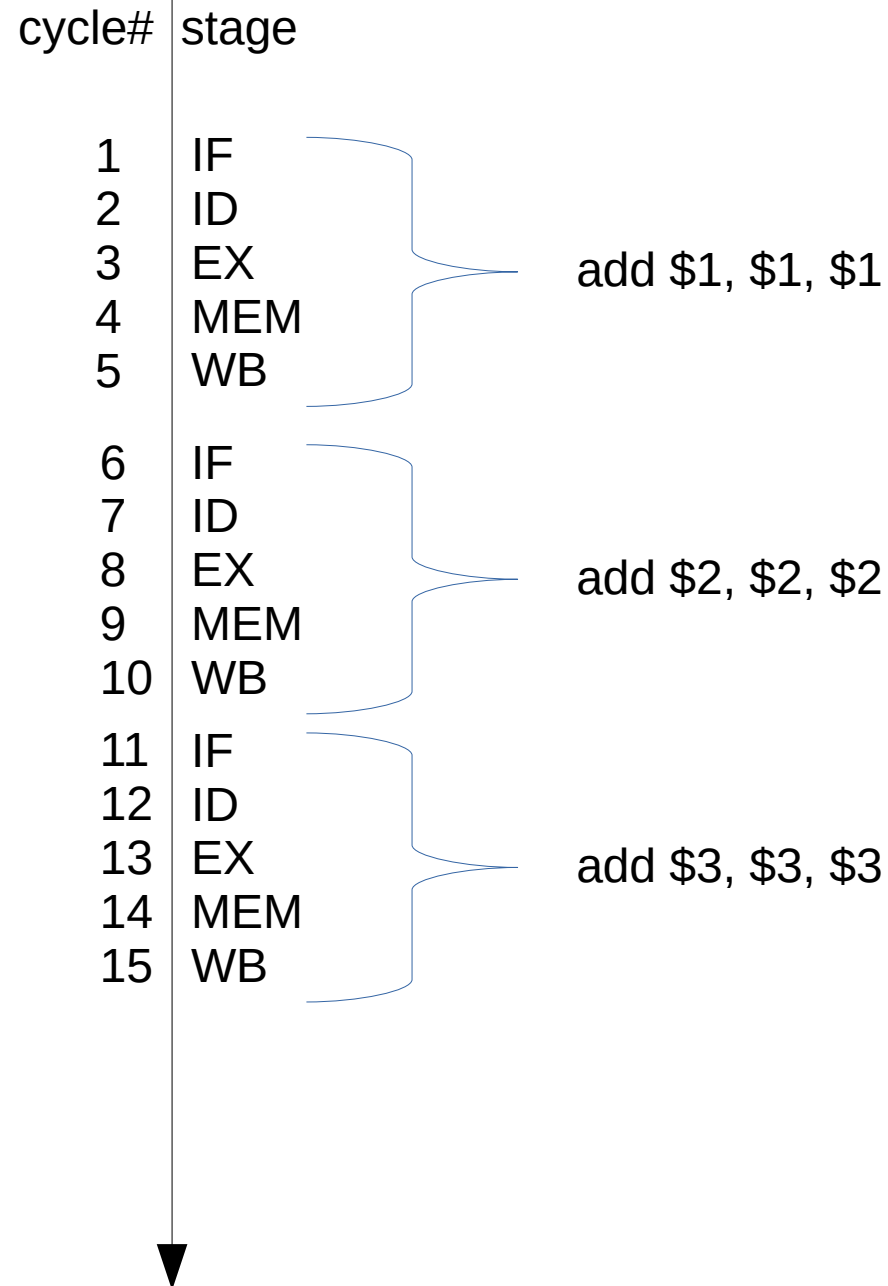
**{addi, RXX, Rg0, 4'b0001}**



# addi \$1, \$1, 1



add \$1, \$1, \$1  
add \$2, \$2, \$2  
add \$3, \$3, \$3



3.1.2 sub \$regDst, \$regSrcA, \$regSrcB

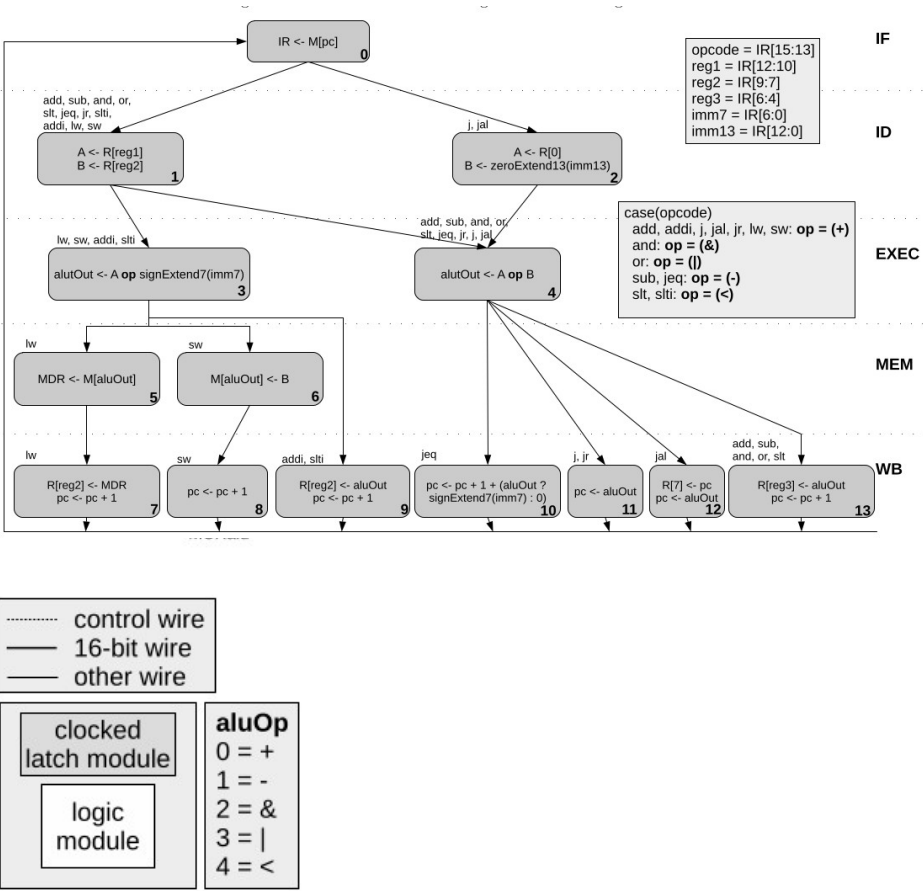
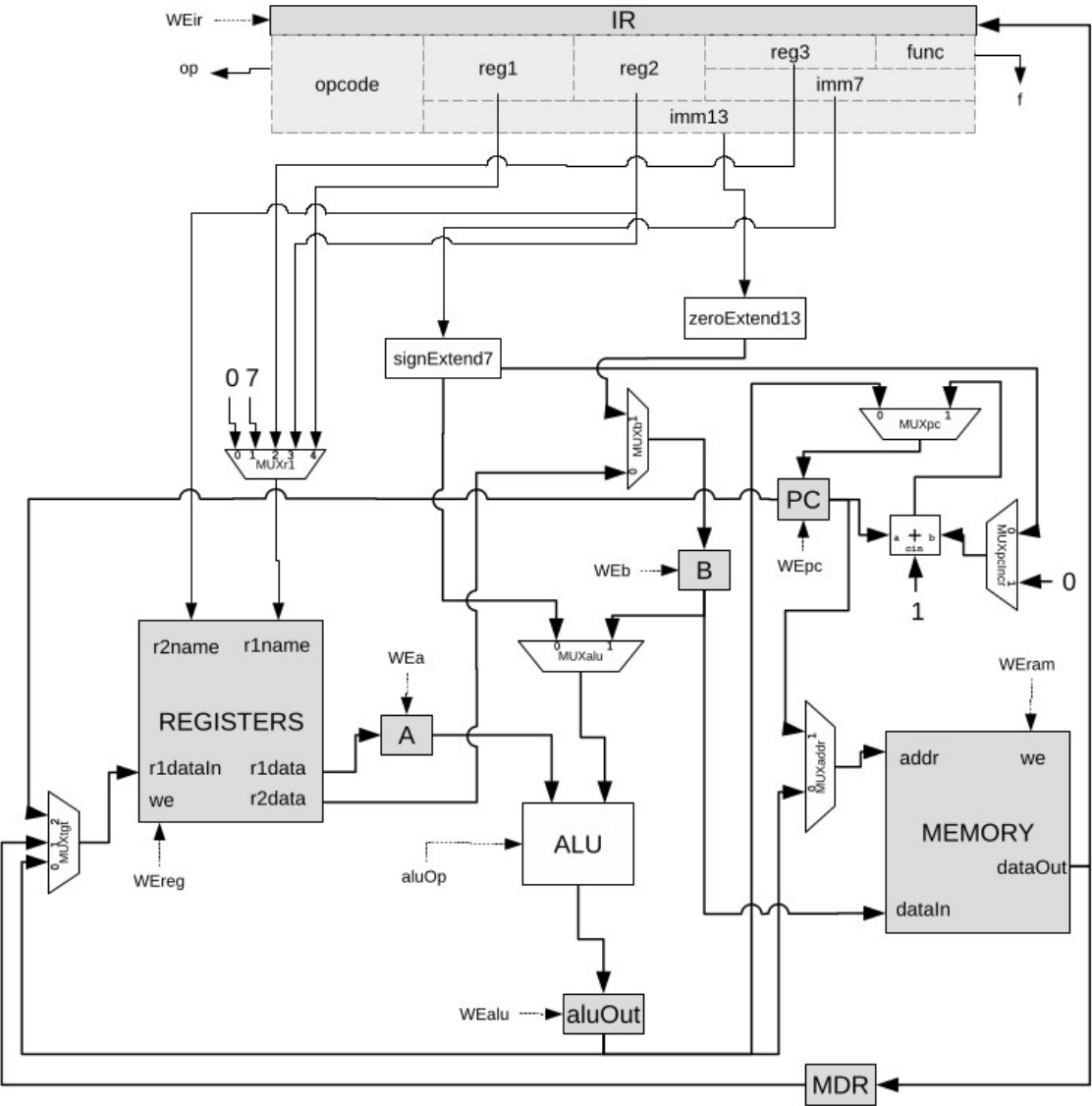
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
3 bits			3 bits			3 bits			3 bits			4 bits			
000			regSrcA			regSrcB			regDst			0001			

Mnemonic: *Subtract*

Example: `sub $1, $0, $5`

Subtracts the value of register `$regSrcB` from `$regSrcA`, storing the difference in `$regDst`.

Symbolically:  $R[\text{regDst}] \leftarrow R[\text{regSrcA}] - R[\text{regSrcB}]$



## Multicycle Example 1

3.1.2 sub \$regDst, \$regSrcA, \$regSrcB

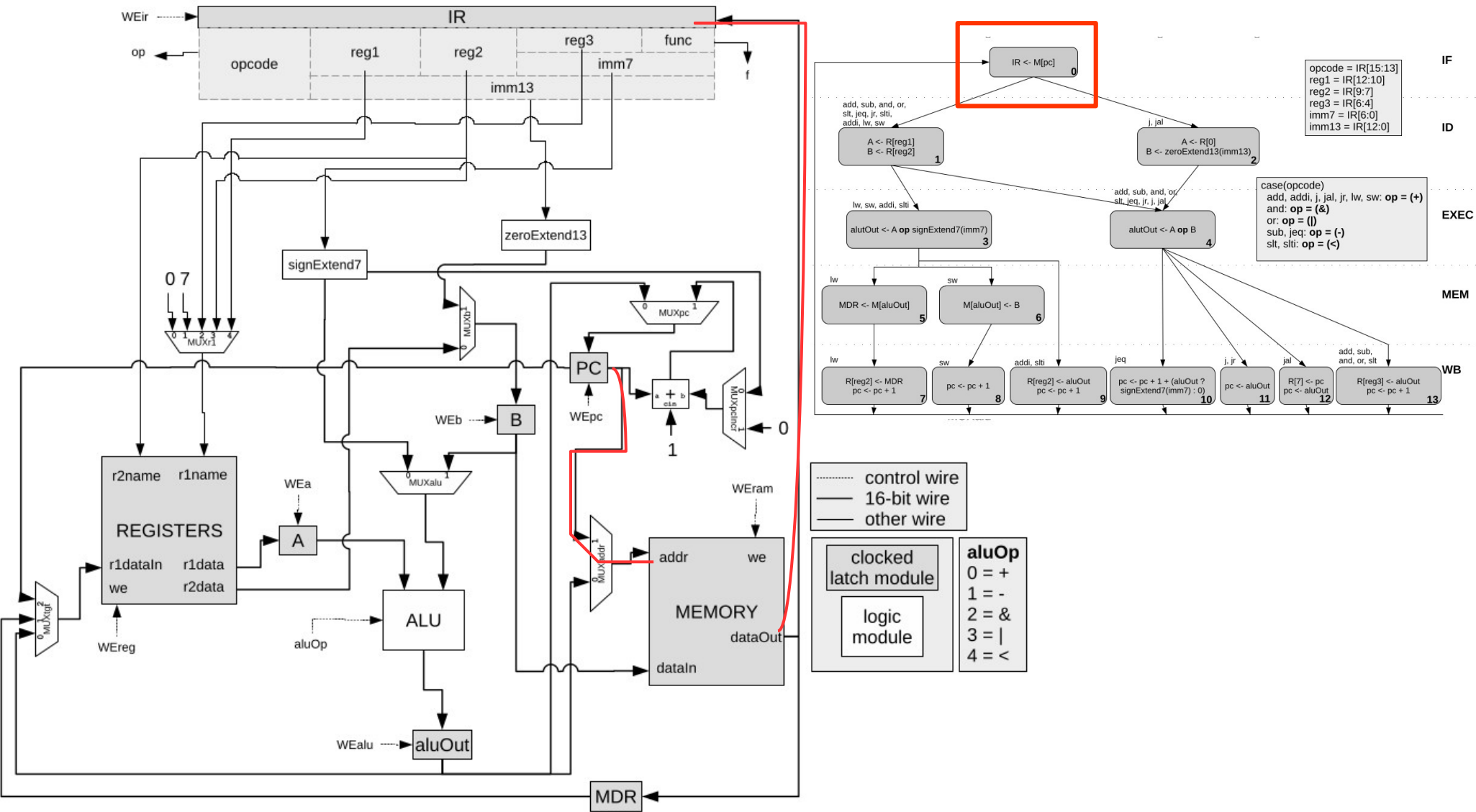
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
3 bits			3 bits			3 bits			3 bits			4 bits			
000			regSrcA			regSrcB			regDst			0001			

Mnemonic: *Subtract*

Example: `sub $1, $0, $5`

Subtracts the value of register `$regSrcB` from `$regSrcA`, storing the difference in `$regDst`.

Symbolically:  $R[\text{regDst}] \leftarrow R[\text{regSrcA}] - R[\text{regSrcB}]$



### 3.1.2 sub \$regDst, \$regSrcA, \$regSrcB

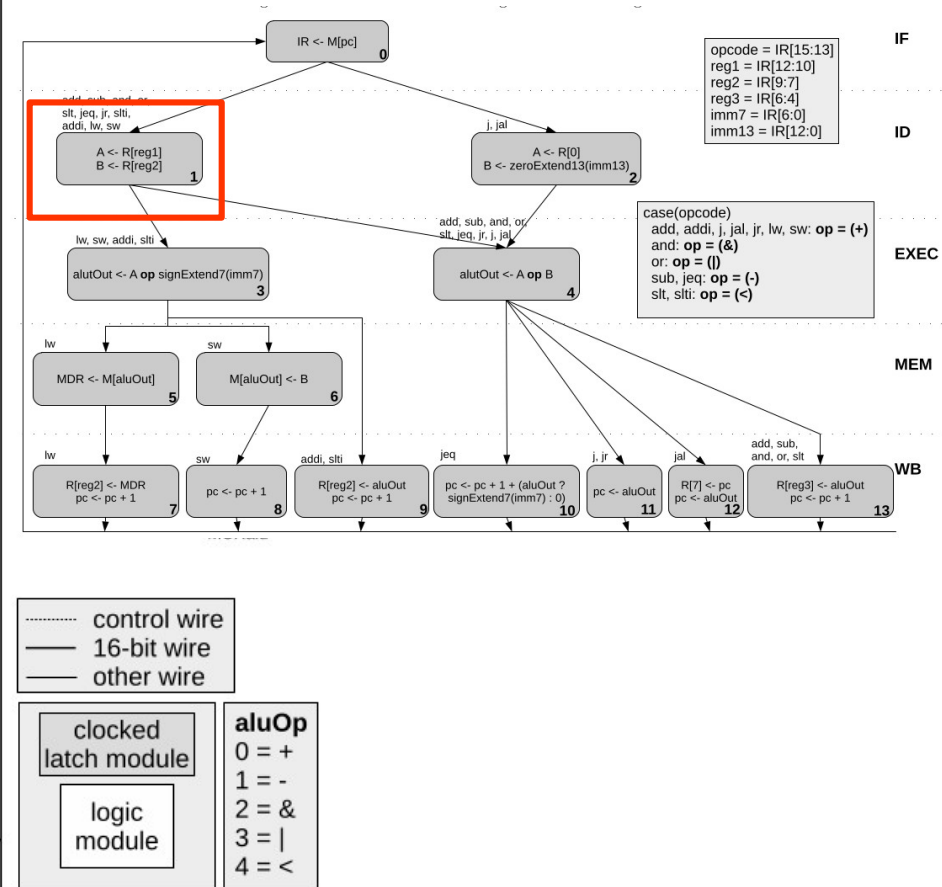
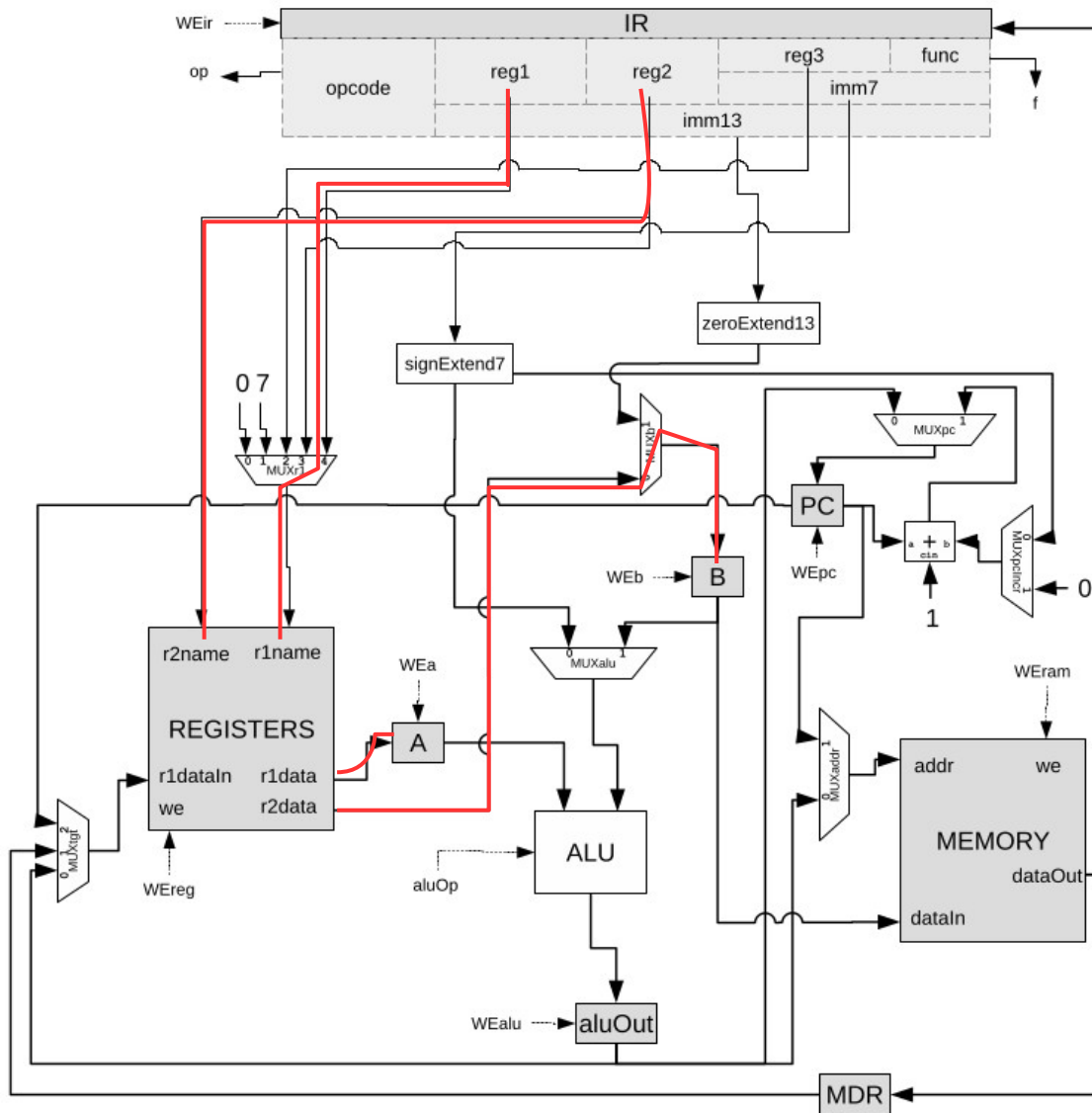
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
3 bits			3 bits			3 bits			3 bits			4 bits			
000			regSrcA			regSrcB			regDst			0001			

Mnemonic: *Subtract*

Example: `sub $1, $0, $5`

Subtracts the value of register `$regSrcB` from `$regSrcA`, storing the difference in `$regDst`.

Symbolically:  $R[\text{regDst}] \leftarrow R[\text{regSrcA}] - R[\text{regSrcB}]$





3.1.2 sub \$regDst, \$regSrcA, \$regSrcB

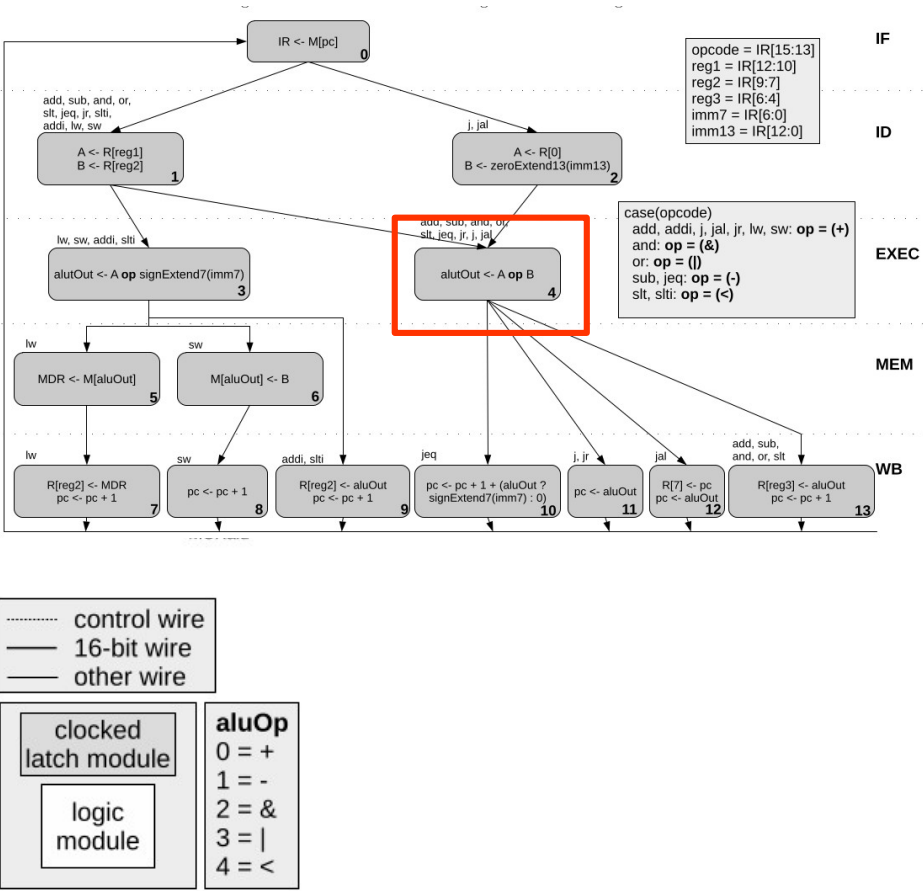
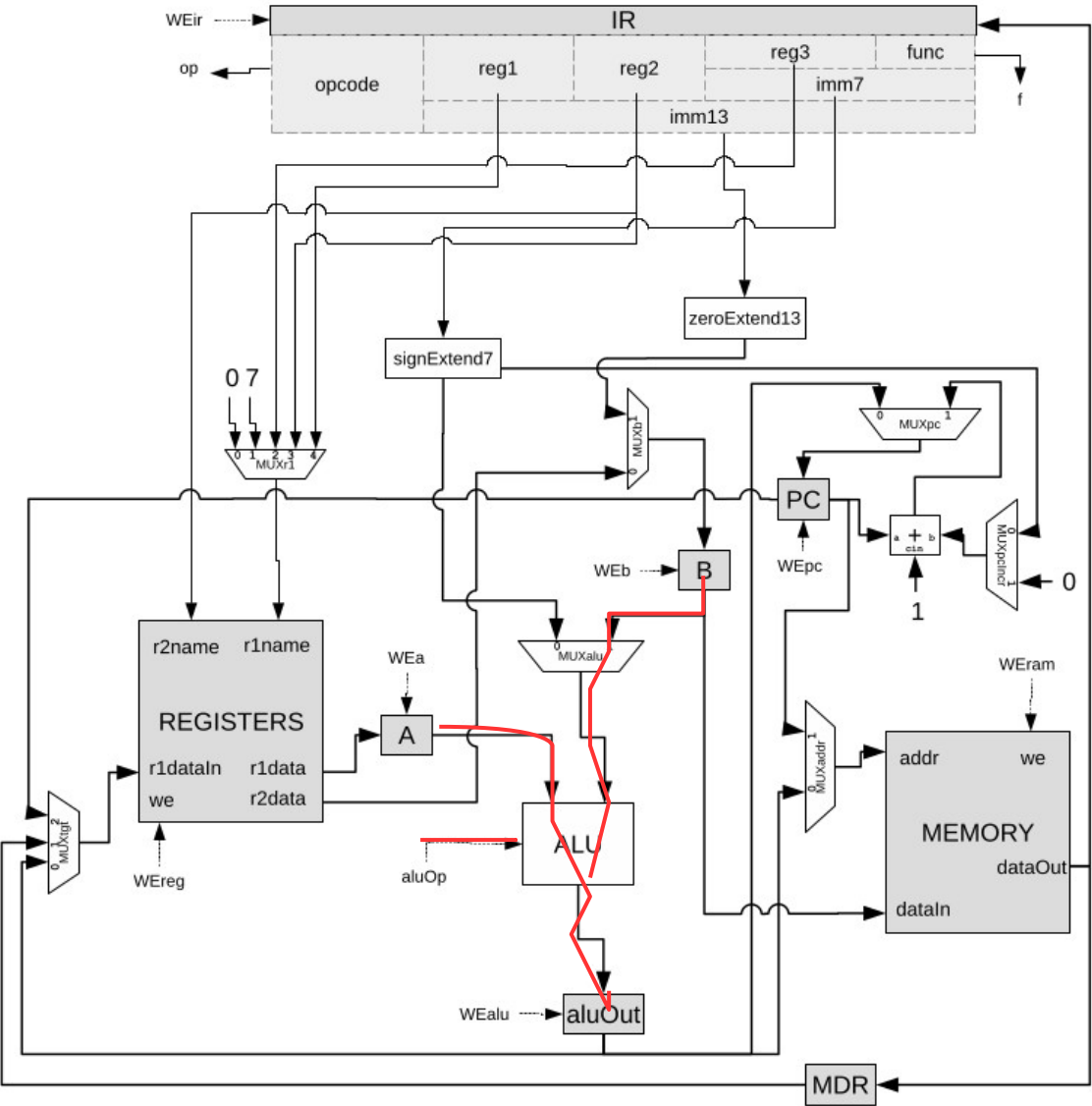
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
3 bits			3 bits			3 bits			3 bits			4 bits			
000			regSrcA			regSrcB			regDst			0001			

Mnemonic: *Subtract*

Example: `sub $1, $0, $5`

Subtracts the value of register `$regSrcB` from `$regSrcA`, storing the difference in `$regDst`.

Symbolically:  $R[\text{regDst}] \leftarrow R[\text{regSrcA}] - R[\text{regSrcB}]$



3.1.2 sub \$regDst, \$regSrcA, \$regSrcB

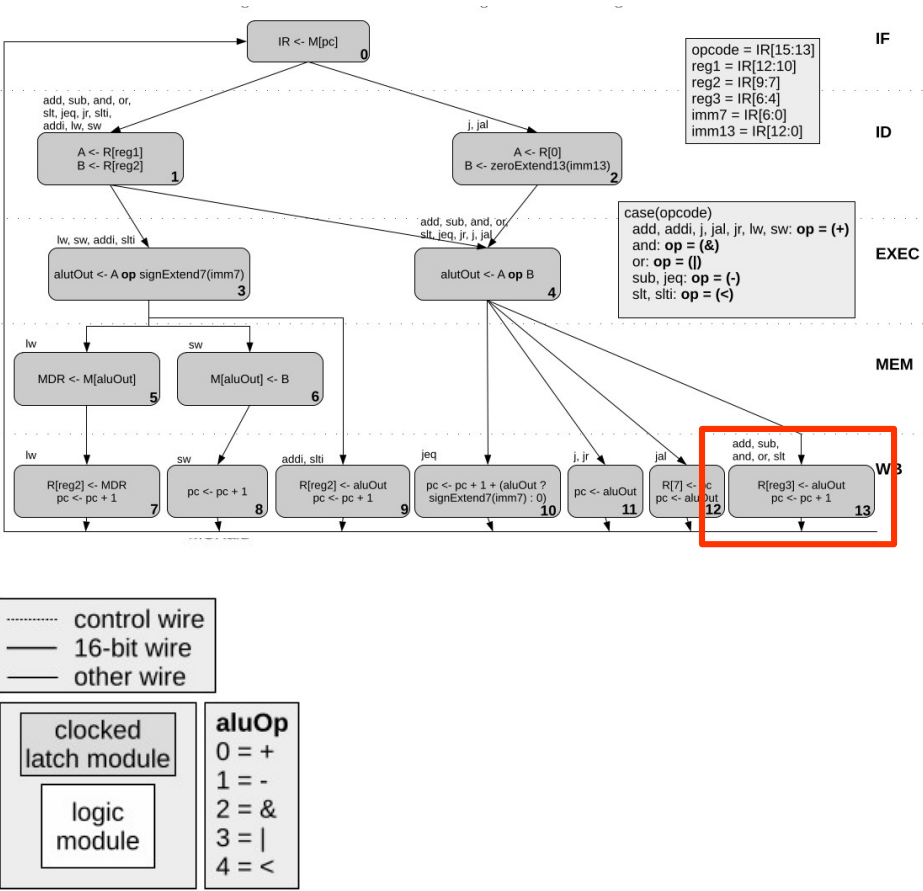
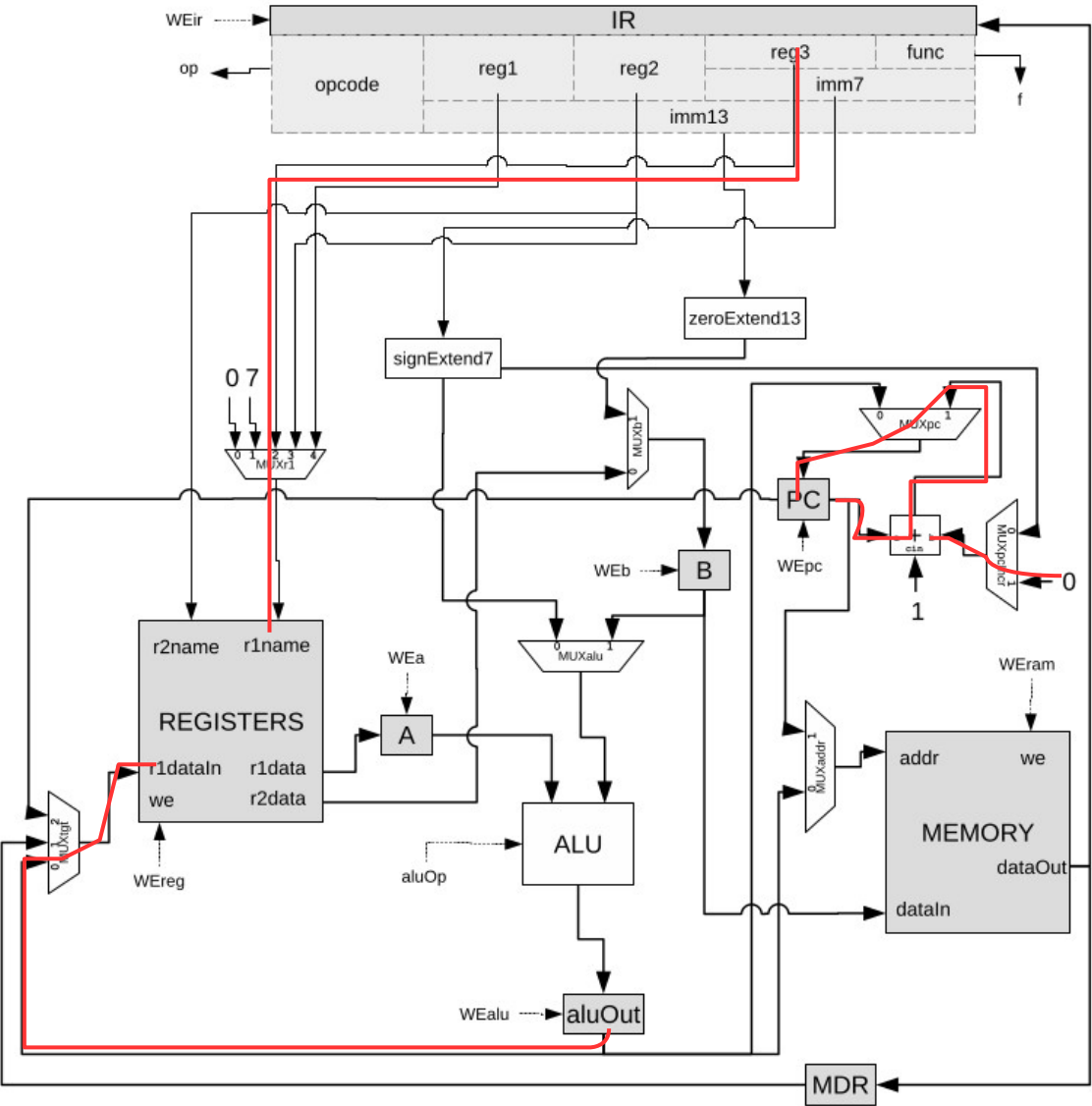
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
3 bits			3 bits			3 bits			3 bits			4 bits			
000			regSrcA			regSrcB			regDst			0001			

Mnemonic: *Subtract*

Example: `sub $1, $0, $5`

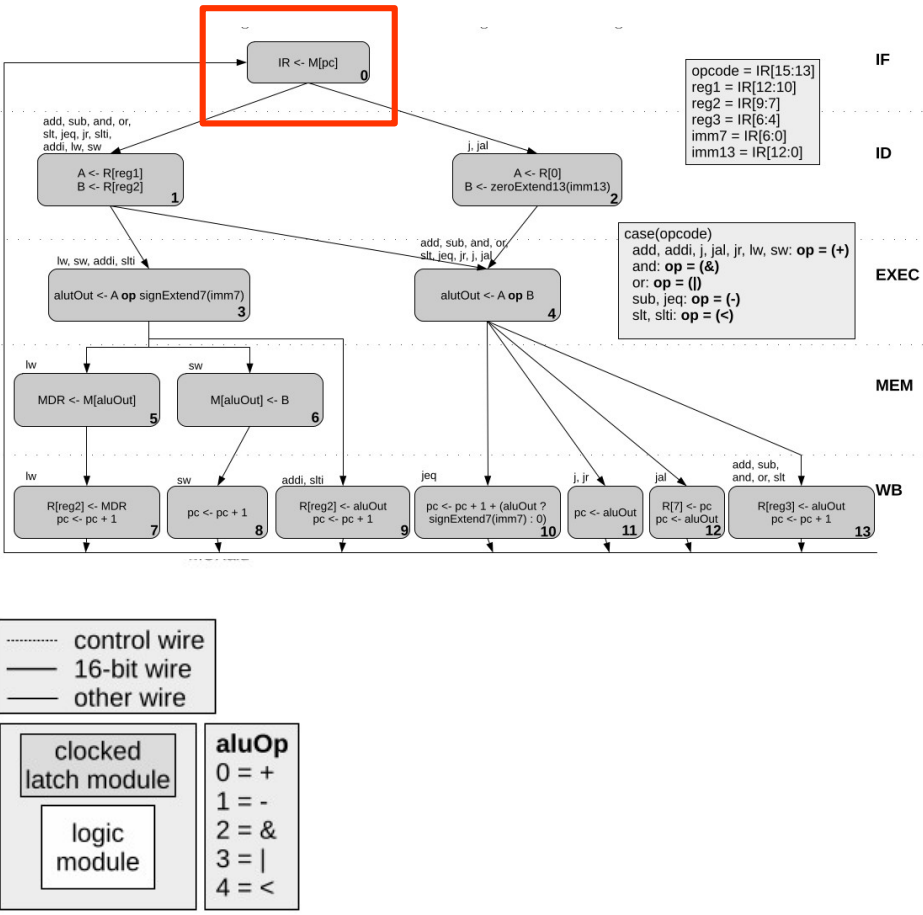
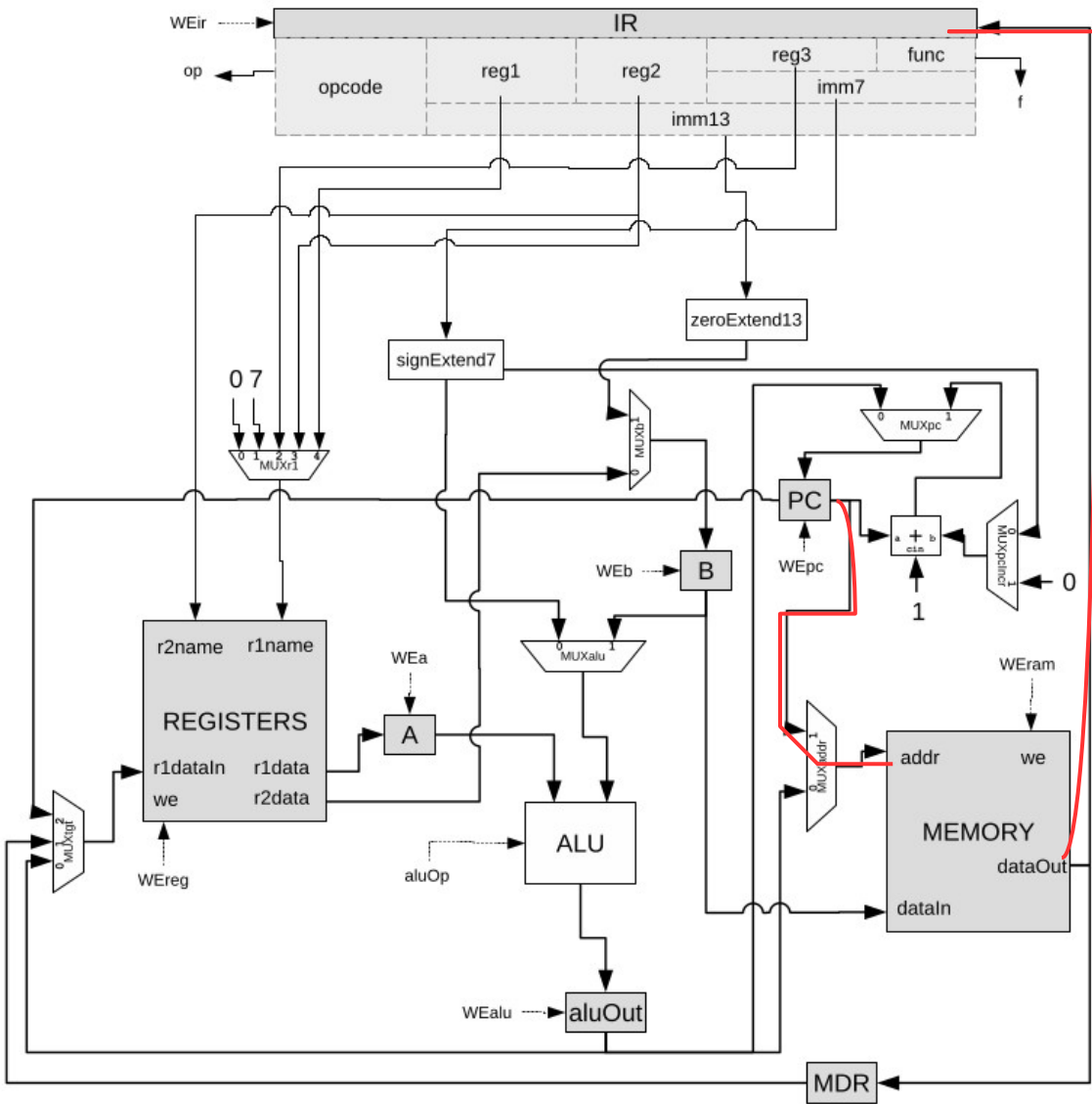
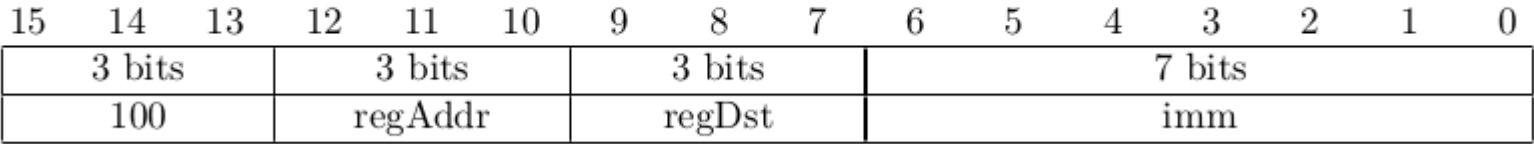
Subtracts the value of register \$regSrcB from \$regSrcA, storing the difference in \$regDst.

Symbolically:  $R[\text{regDst}] \leftarrow R[\text{regSrcA}] - R[\text{regSrcB}]$



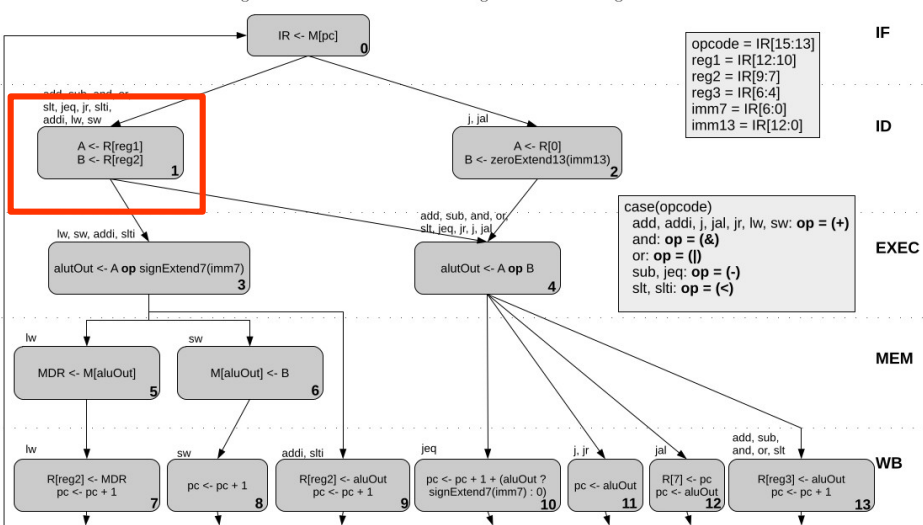
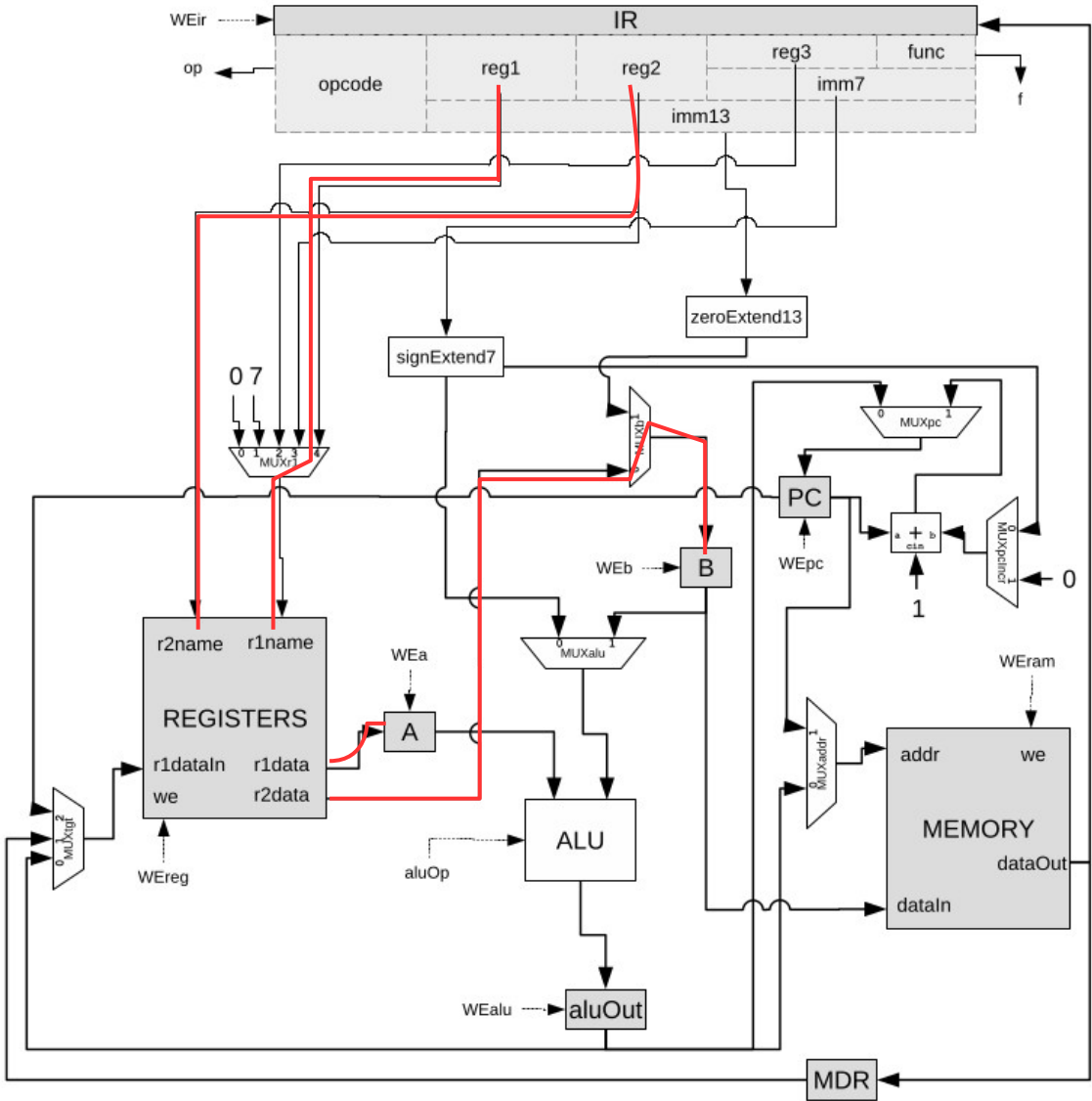
## Multicycle Example 2

3.2.2 lw \$regDst, imm(\$regAddr)



3.2.2 lw \$regDst, imm(\$regAddr)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
3 bits			3 bits			3 bits			7 bits						
100			regAddr			regDst			imm						



control wire

16-bit wire

other wire

clocked latch module

logic module

aluOp

0 = +

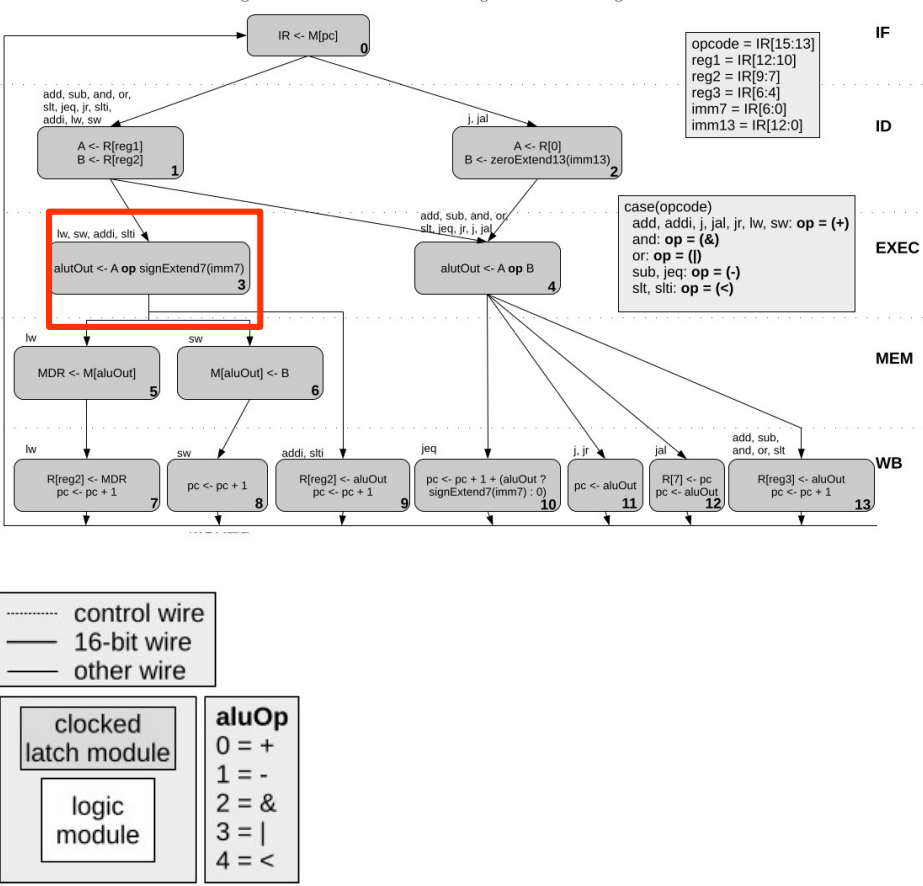
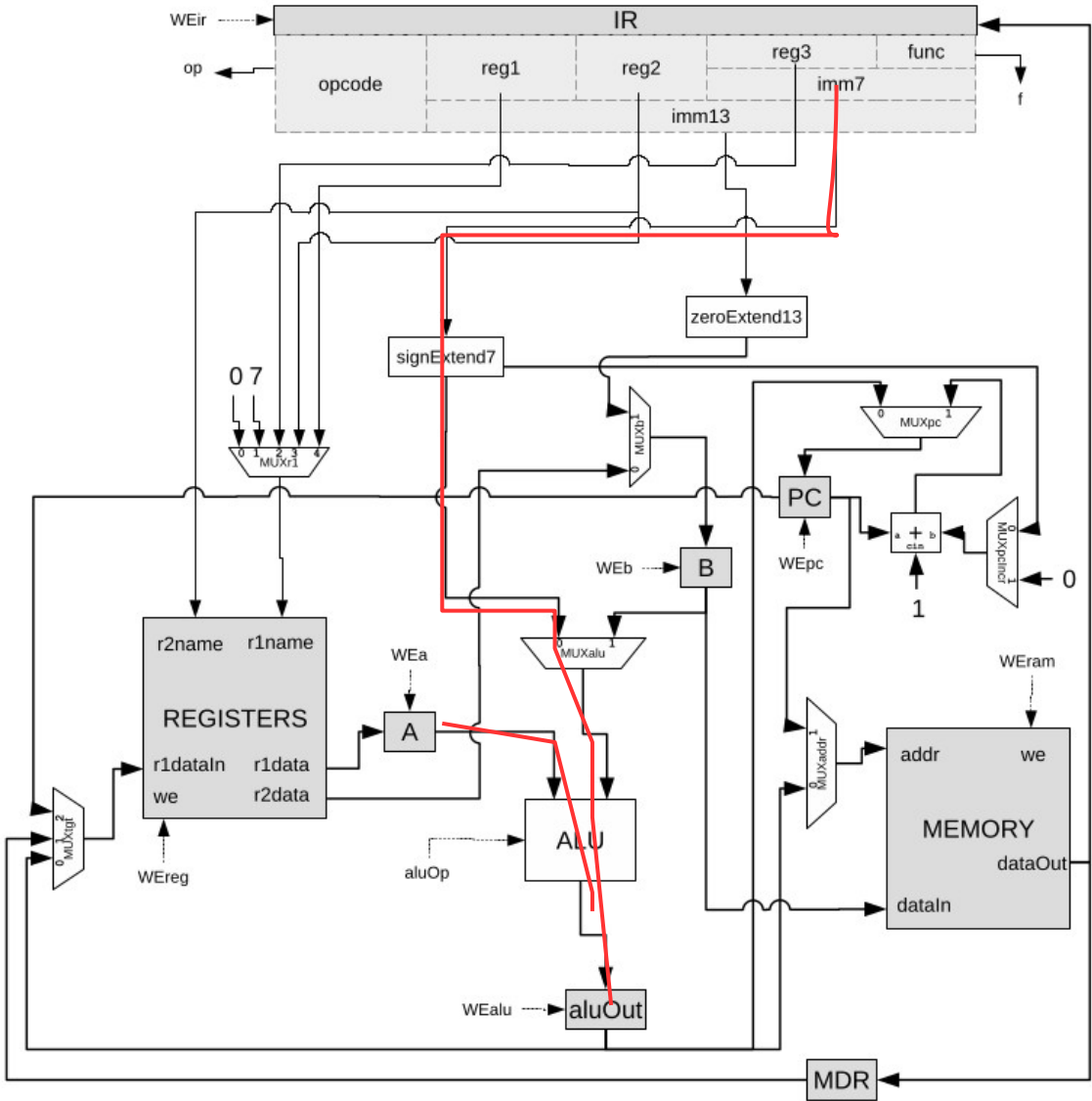
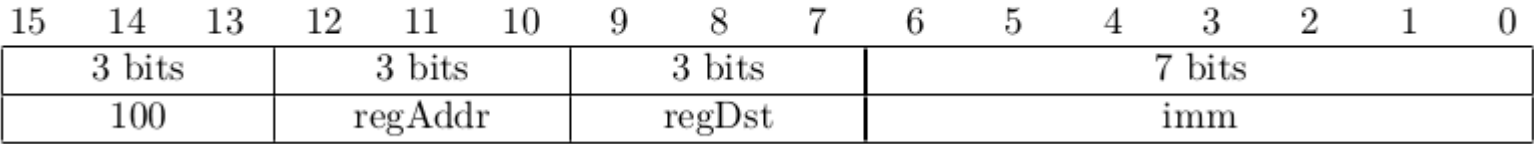
1 = -

2 = &

3 = |

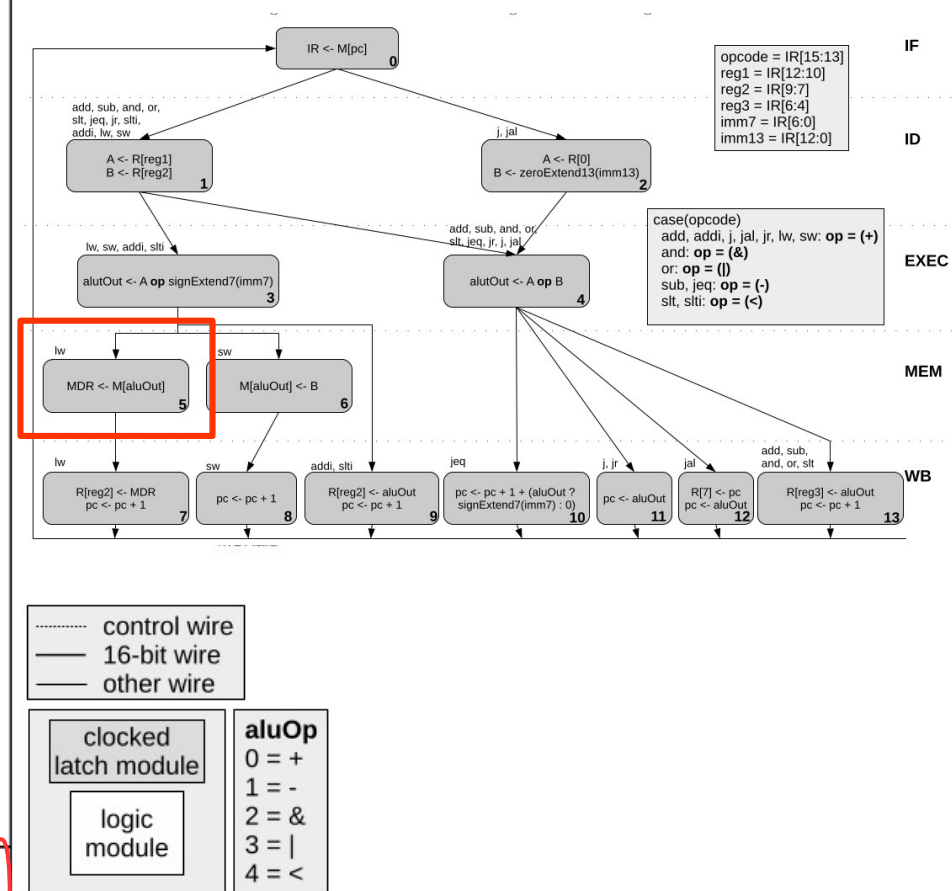
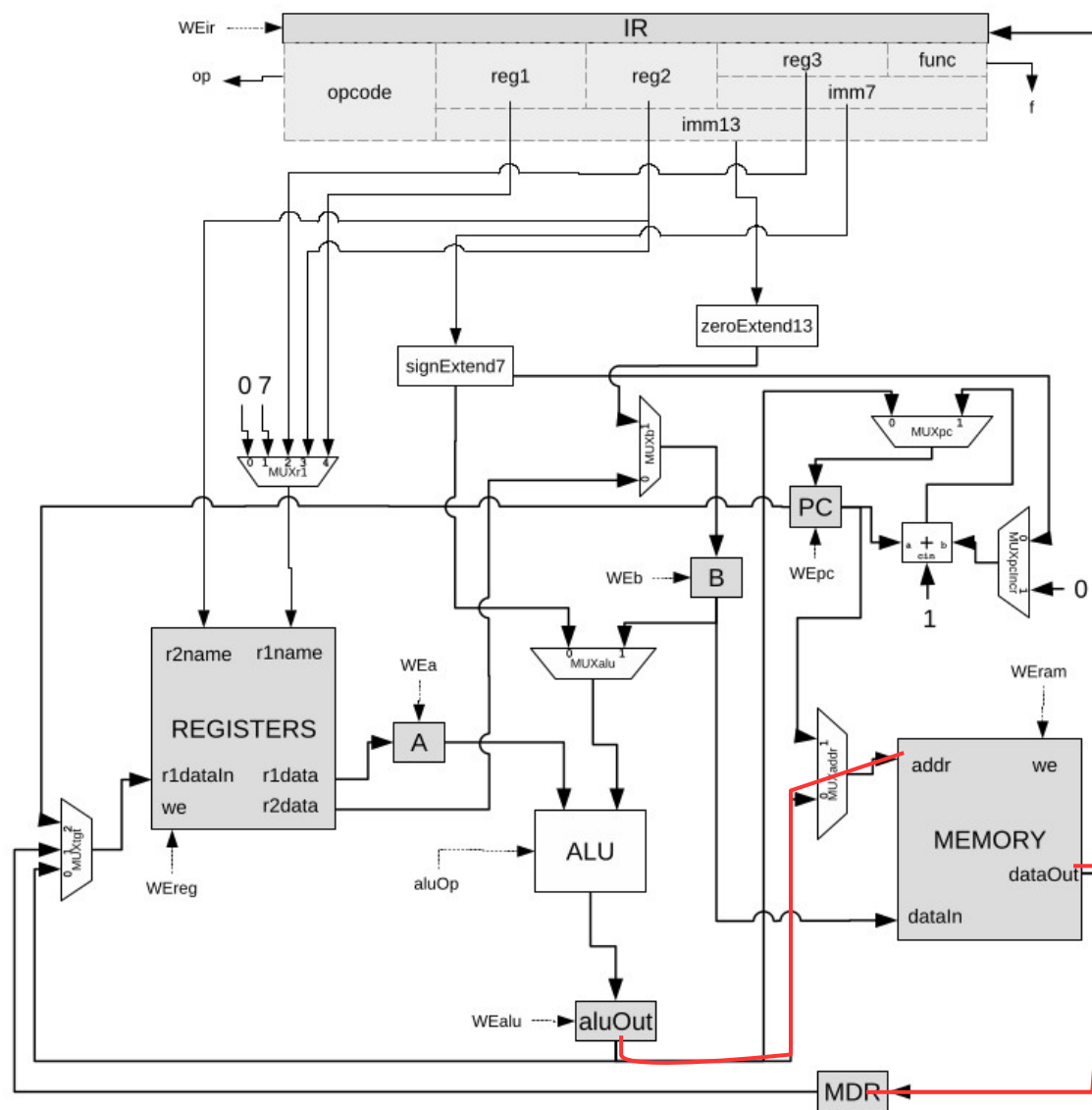
4 = <

3.2.2 lw \$regDst, imm(\$regAddr)



### 3.2.2 lw \$regDst, imm(\$regAddr)

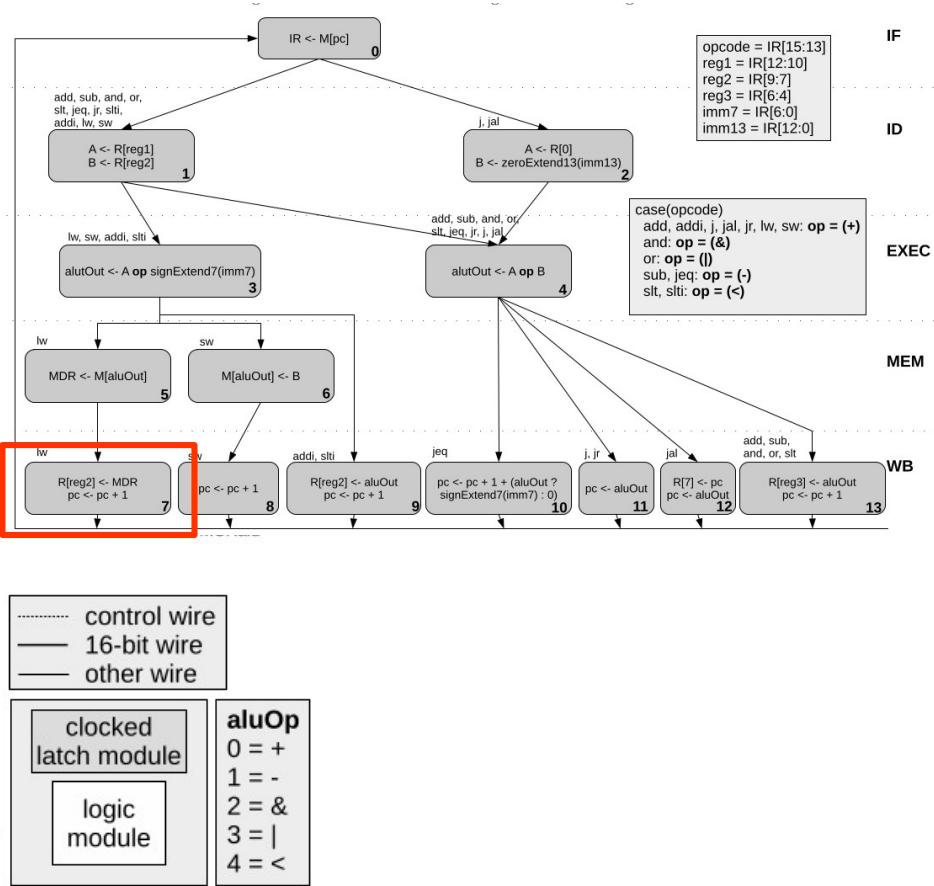
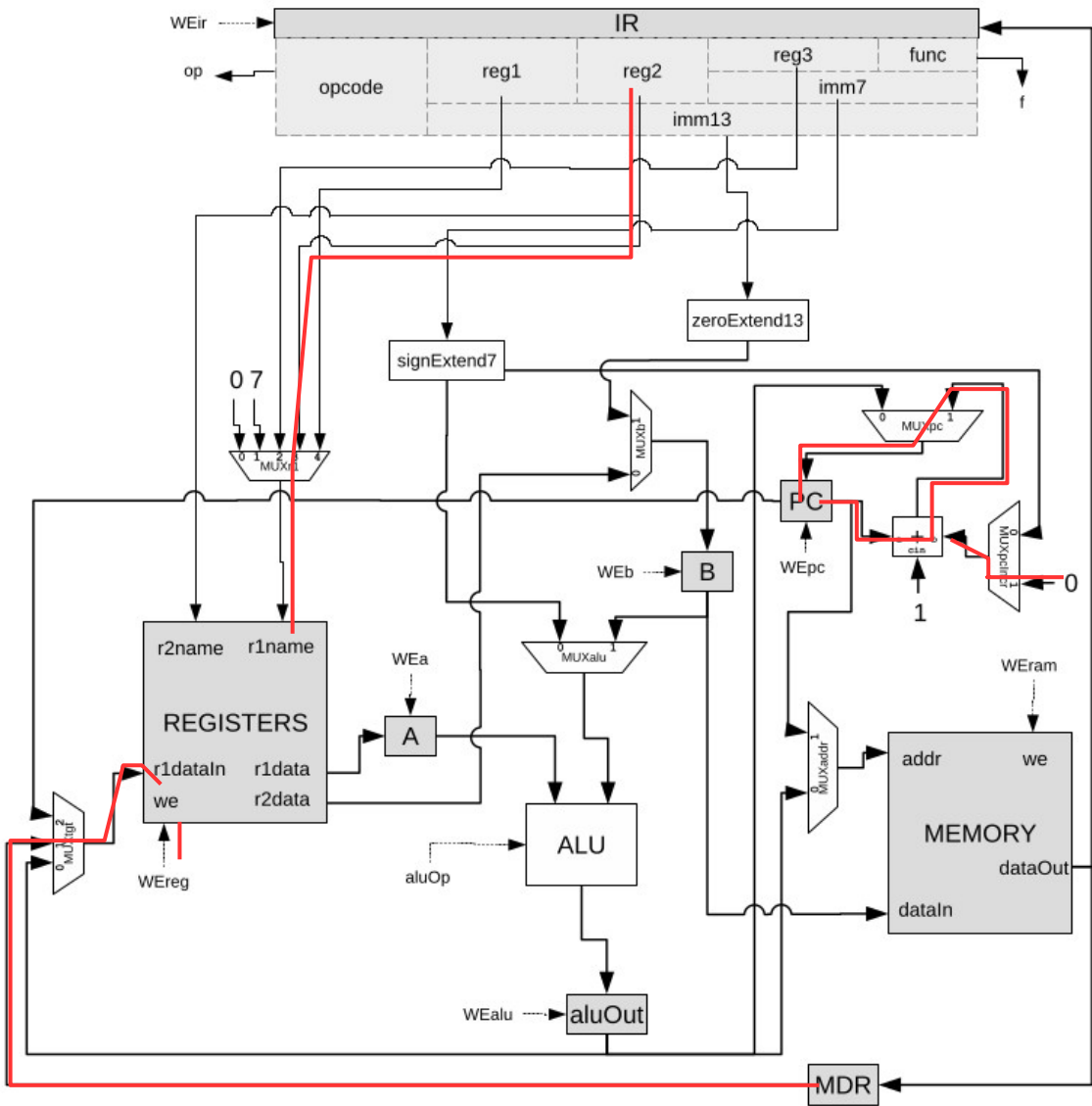
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
3 bits			3 bits			3 bits			7 bits						
100			regAddr			regDst			imm						





3.2.2 lw \$regDst, imm(\$regAddr)

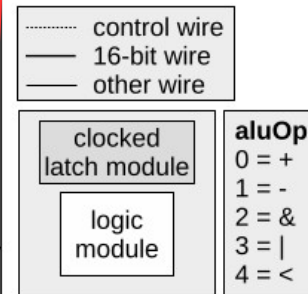
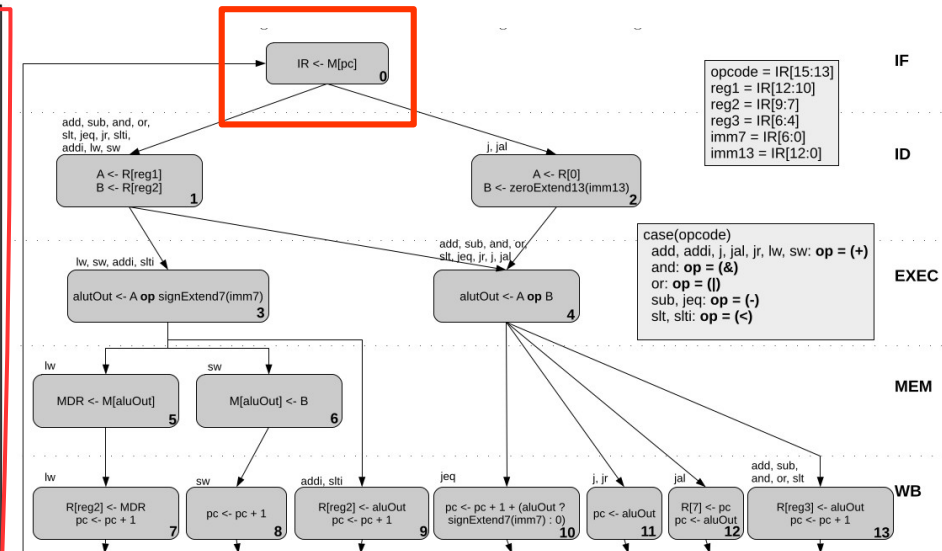
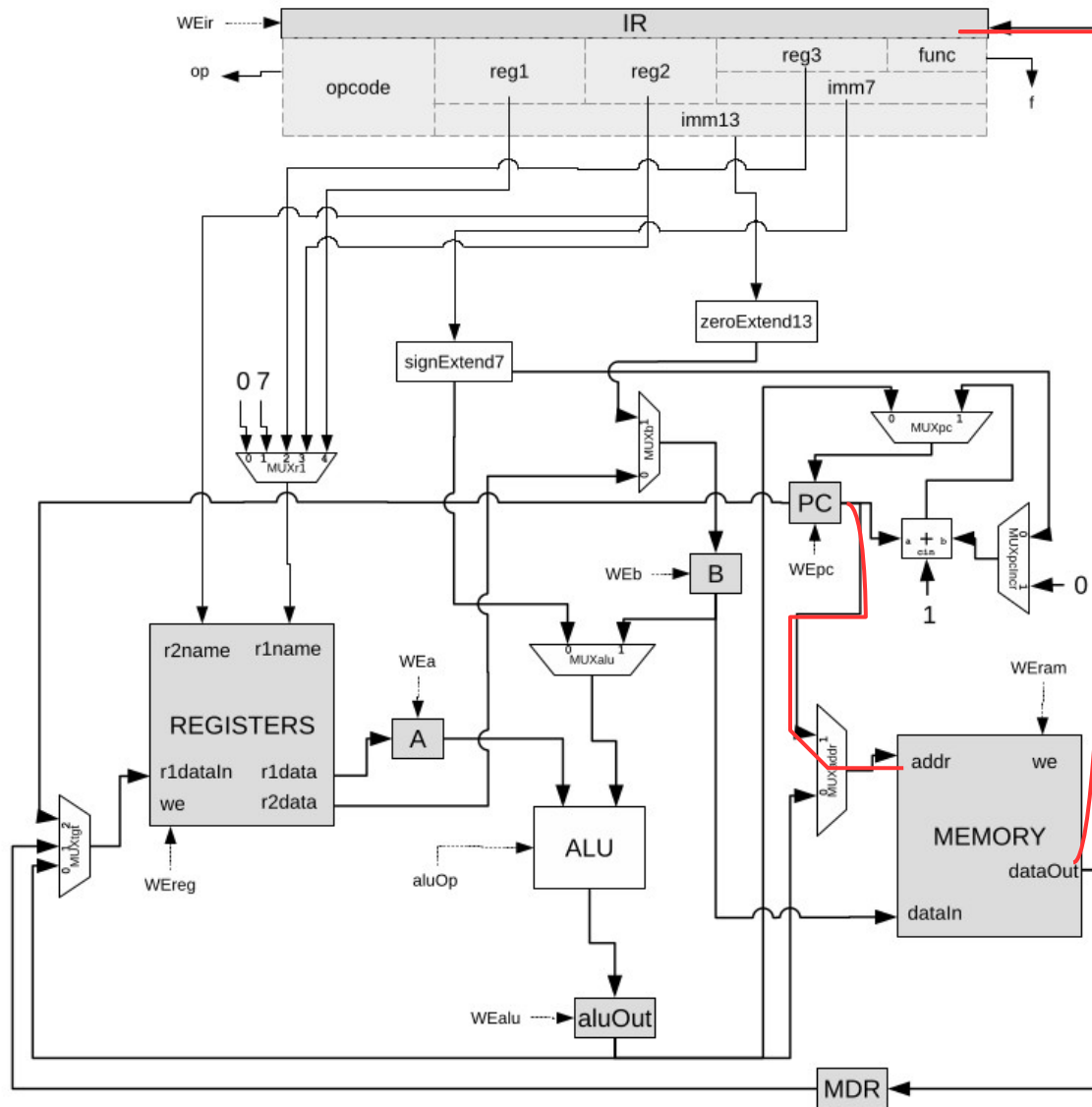
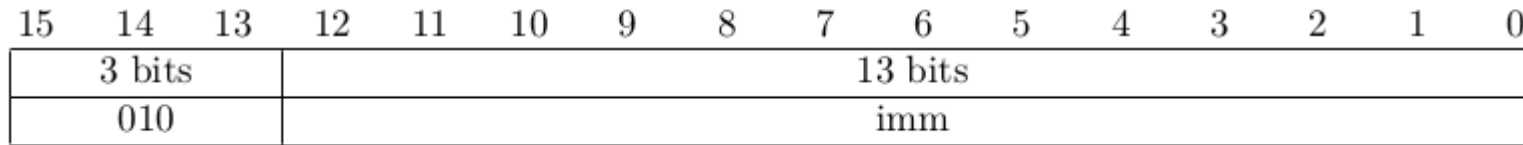
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
3 bits			3 bits			3 bits			7 bits						
100			regAddr			regDst			imm						



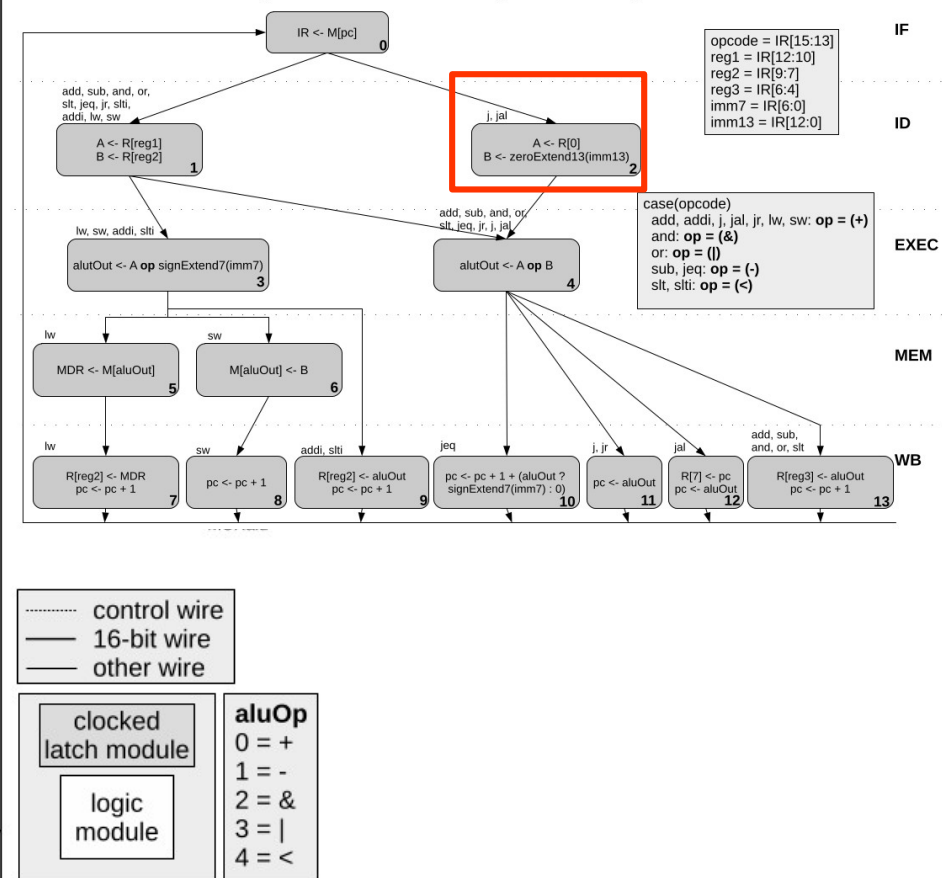
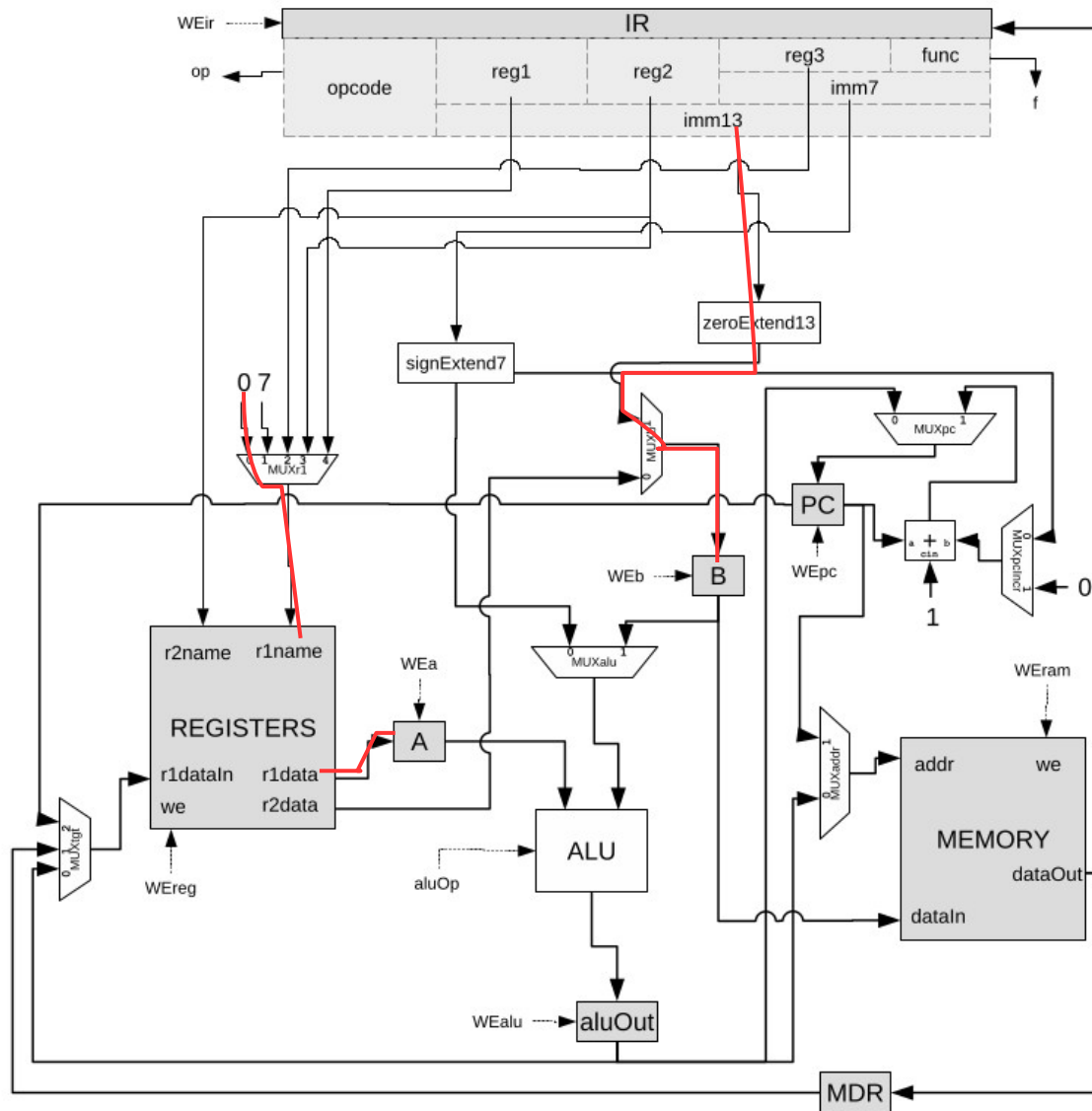
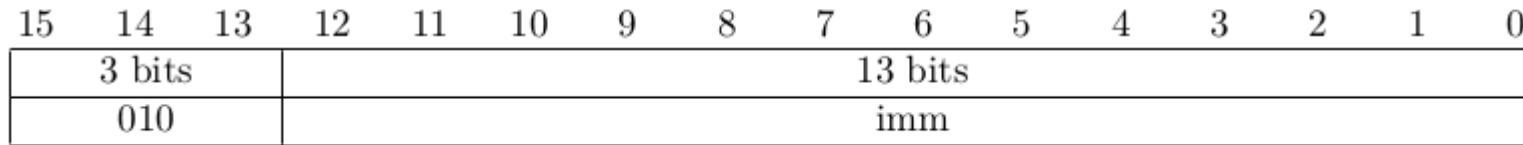


## Multicycle Example 3

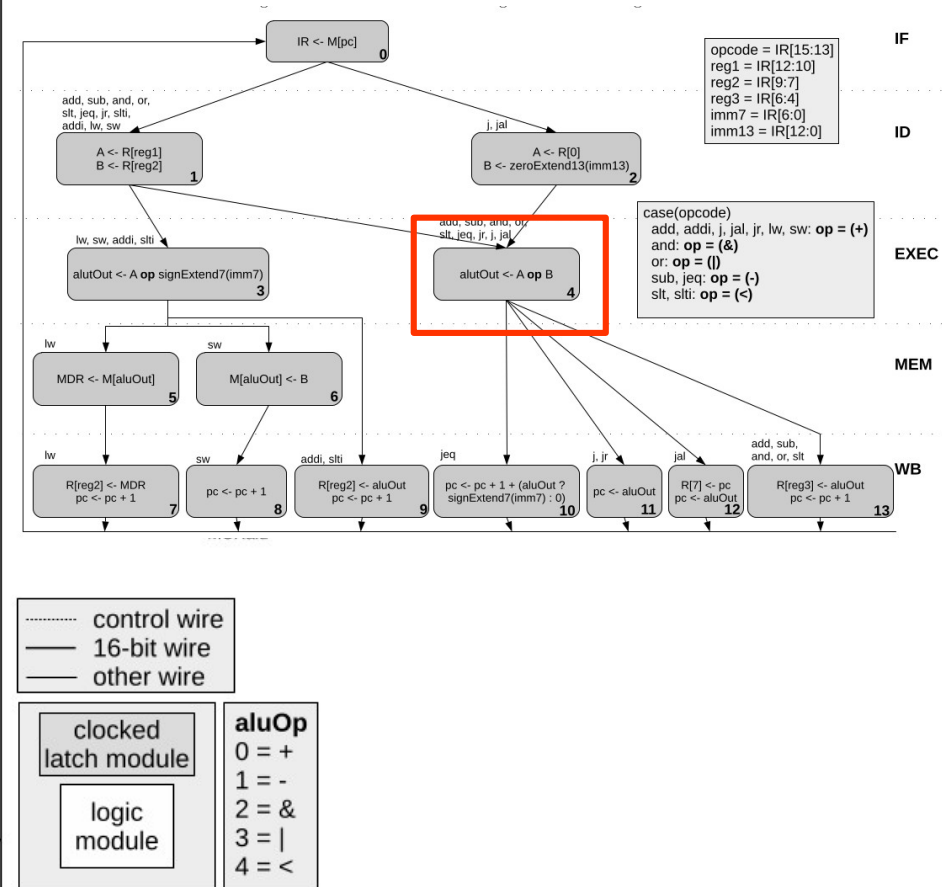
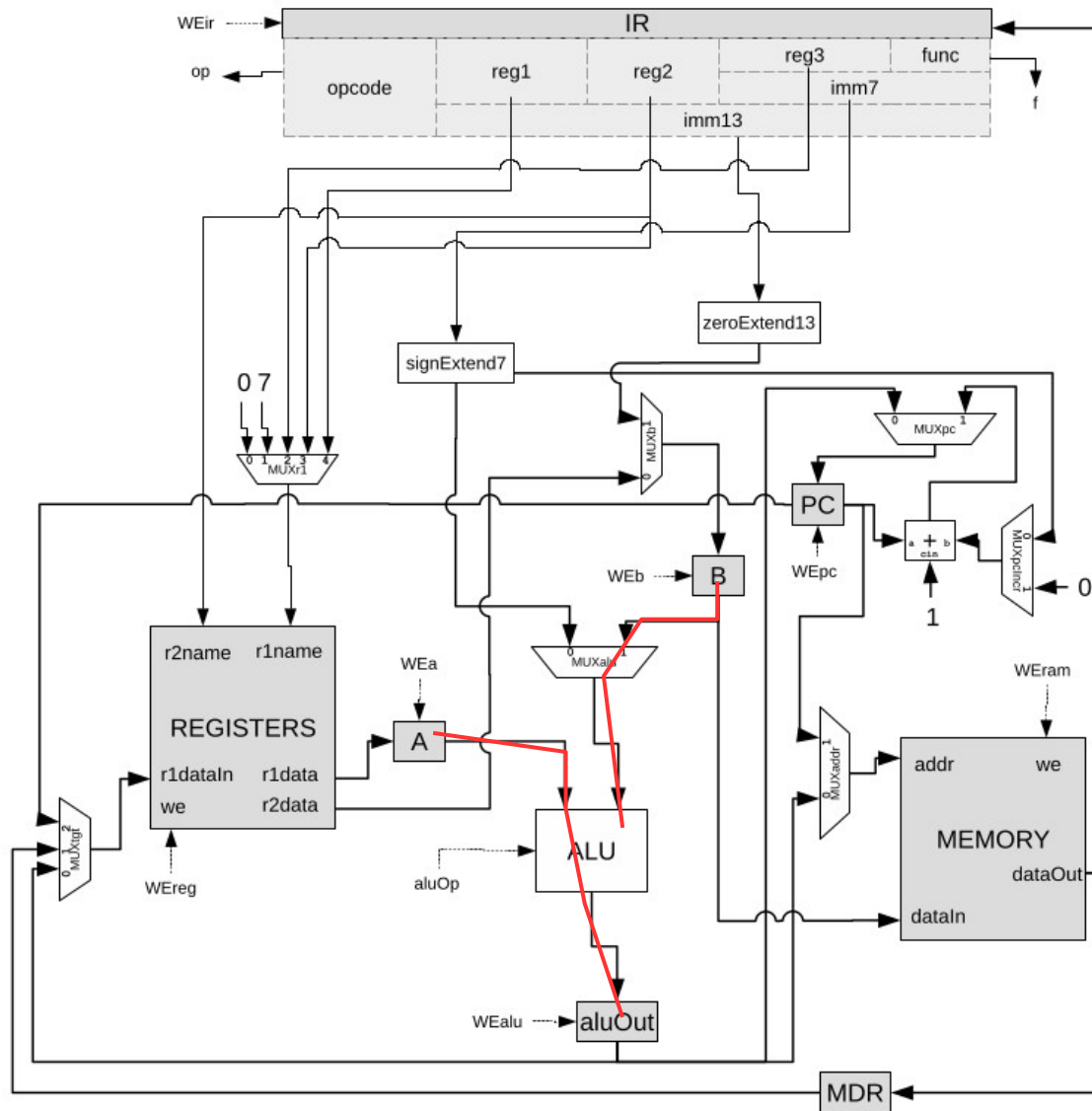
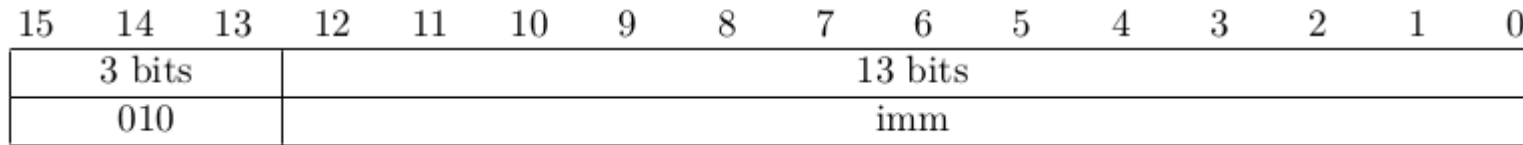
### 3.3.1 j imm



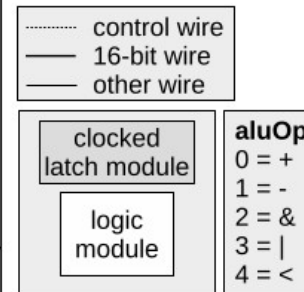
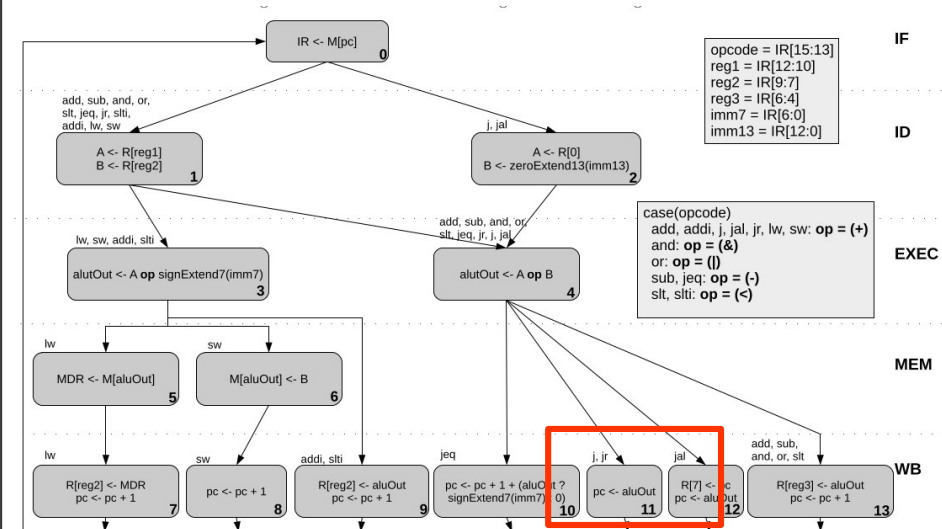
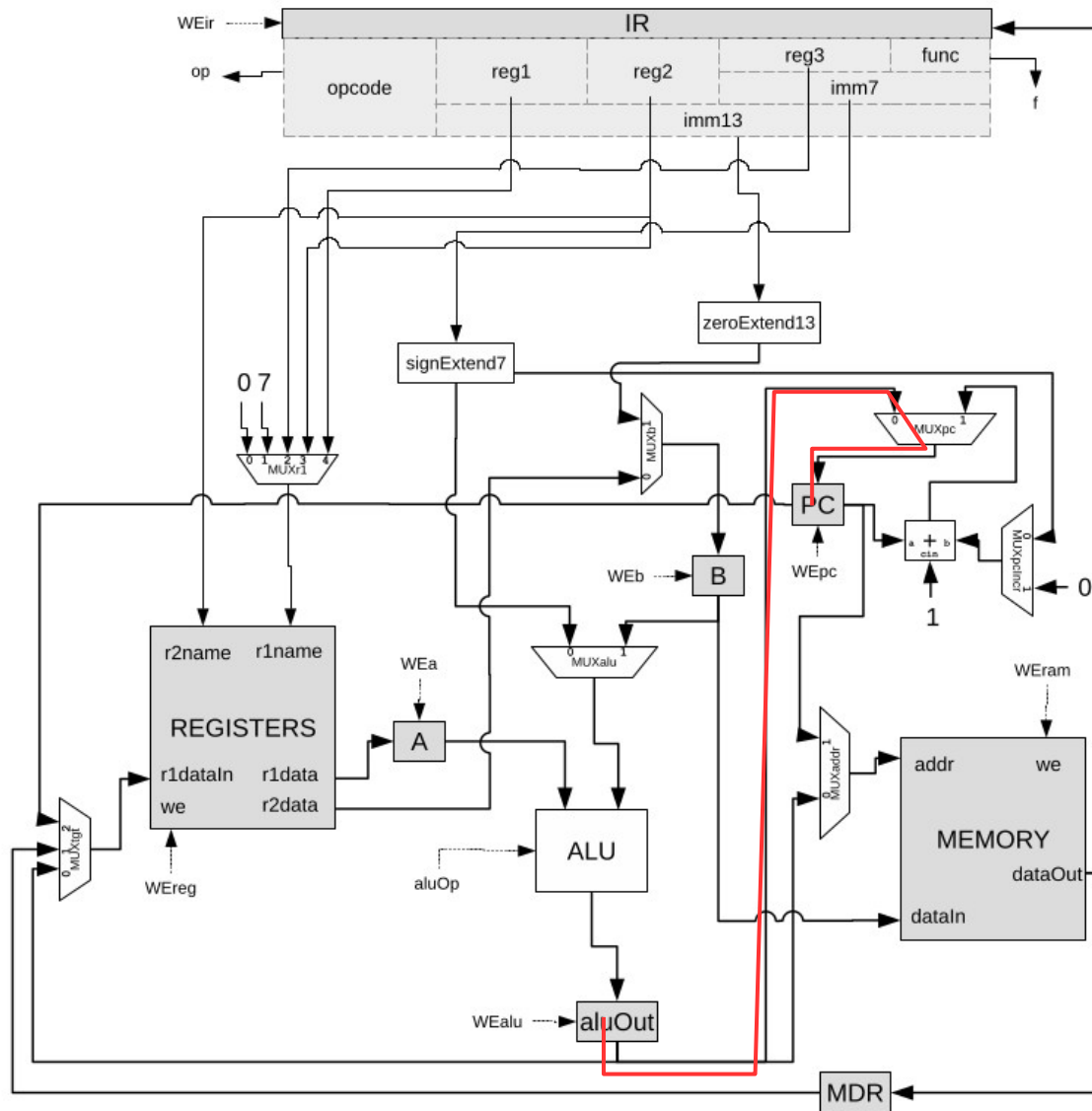
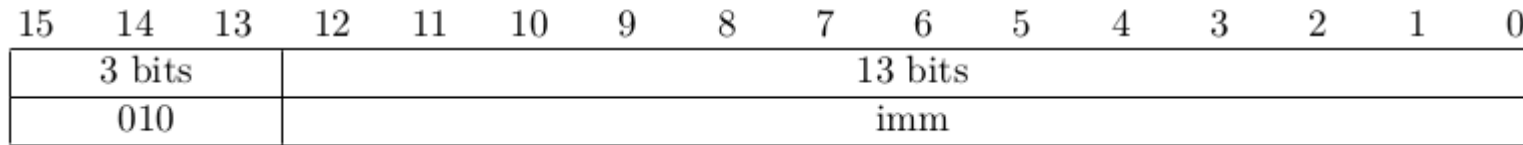
### 3.3.1 j imm

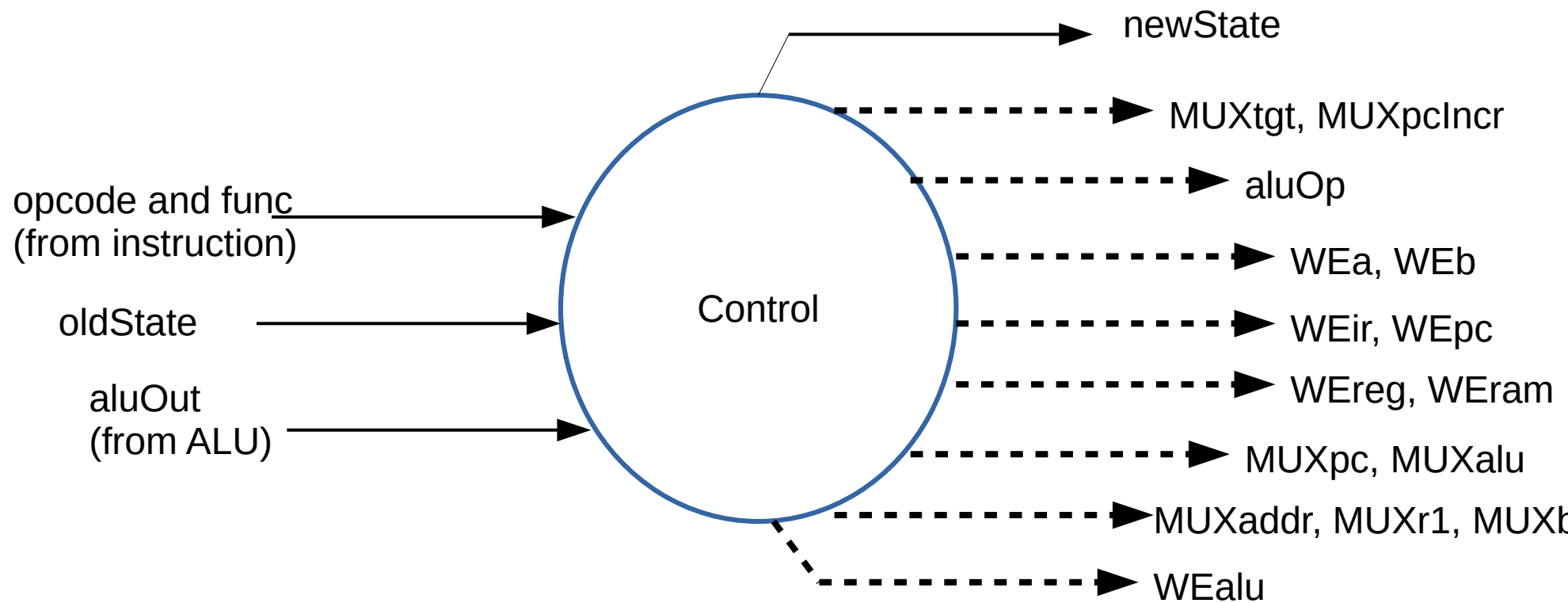


### 3.3.1 j imm



### 3.3.1 j imm





# Control module truth table: multicycle version

Inputs

Outputs

opcode& func	aluOut	old State	new State	MUXtgt	MUXpcIncr	aluOp	WEa	WEir	WEreg	WEpc	WEram	MUXpc	MUXalu	WEalu	WEb	MUXaddr	MUXr1	MUXb
add		4	13			0	0	0	0	0	0		1	1	0			
sub		4	13			1	0	0	0	0	0		1	1	0			
add		13	0	0	1		0	0	1	1	0	1		0	0		2	
etc																		

blank = don't care

# Laundry





Wash



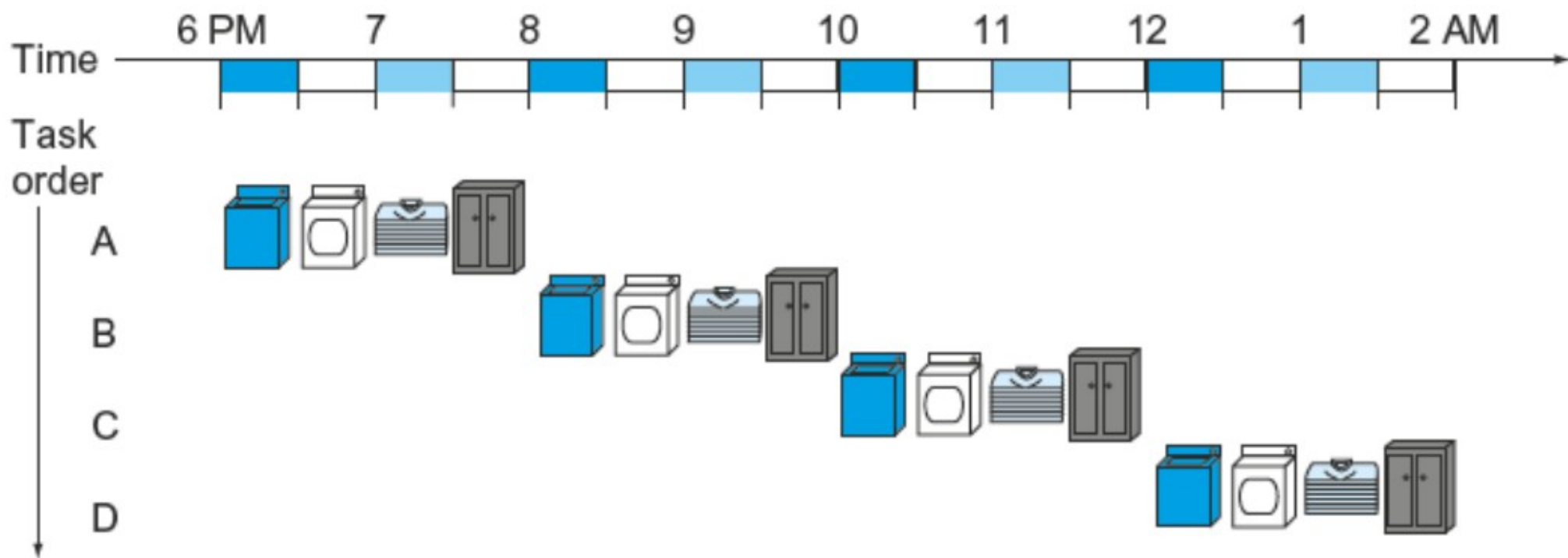
Dry

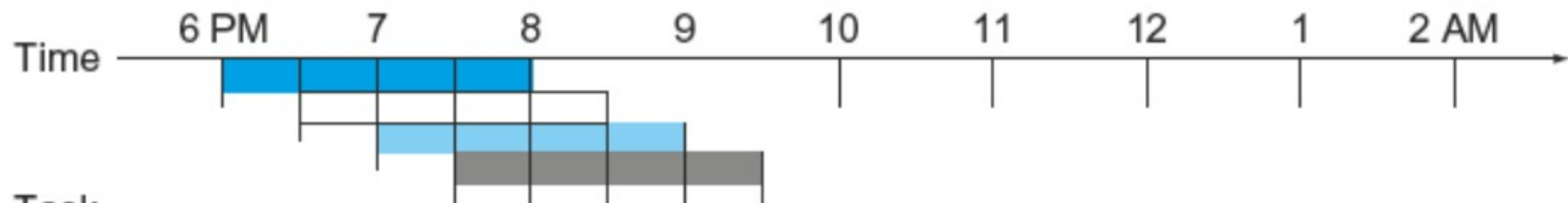


Fold



Store





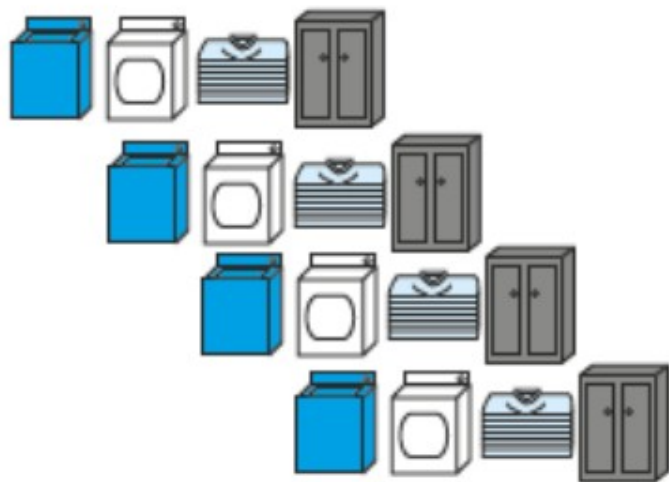
Task  
order

A

B

C

D



E20 pipelining

## E20 pipelining

Pipelining is an implementation technique in which multiple instructions are overlapped in execution.

## E20 pipelining

1. Fetch instruction from memory ---- IF
2. Read registers while decoding the instruction --- ID
3. Execute the operation or calculate an address --- EX
4. Access an operand in data memory --- MEM
5. Write the result into a register --- WB

addi \$1, \$1, 1  
addi \$2, \$2, 2  
addi \$3, \$3, 3  
addi \$4, \$4, 4  
addi \$5, \$5, 5

Instruction  
fetch

Instruction  
decode

Execute

Memory

Writeback

Time	IF	ID	EX	MEM	WB
1					
2					
3					
4					
5					
6					
7					

addi \$1, \$1, 1  
addi \$2, \$2, 2  
addi \$3, \$3, 3  
addi \$4, \$4, 4  
addi \$5, \$5, 5

"issue"

Time	IF	ID	EX	MEM	WB
1	addi \$1, \$1, 1				
2					
3					
4					
5					
6					
7					



addi \$1, \$1, 1  
addi \$2, \$2, 2  
addi \$3, \$3, 3  
addi \$4, \$4, 4  
addi \$5, \$5, 5

Time	IF	ID	EX	MEM	WB
1	addi \$1, \$1, 1				
2	addi \$2, \$2, 2	addi \$1, \$1, 1			
3					
4					
5					
6					
7					

addi \$1, \$1, 1  
addi \$2, \$2, 2  
addi \$3, \$3, 3  
addi \$4, \$4, 4  
addi \$5, \$5, 5

Time	IF	ID	EX	MEM	WB
1	addi \$1, \$1, 1				
2	addi \$2, \$2, 2	addi \$1, \$1, 1			
3	addi \$3, \$3, 3	addi \$2, \$2, 2	addi \$1, \$1, 1		
4					
5					
6					
7					

addi \$1, \$1, 1  
addi \$2, \$2, 2  
addi \$3, \$3, 3  
addi \$4, \$4, 4  
addi \$5, \$5, 5

Time	IF	ID	EX	MEM	WB
1	addi \$1, \$1, 1				
2	addi \$2, \$2, 2	addi \$1, \$1, 1			
3	addi \$3, \$3, 3	addi \$2, \$2, 2	addi \$1, \$1, 1		
4	addi \$4, \$4, 4	addi \$3, \$3, 3	addi \$2, \$2, 2	addi \$1, \$1, 1	
5	addi \$5, \$5, 5	addi \$4, \$4, 4	addi \$3, \$3, 3	addi \$2, \$2, 2	addi \$1, \$1, 1
6		addi \$5, \$5, 5	addi \$4, \$4, 4	addi \$3, \$3, 3	addi \$2, \$2, 2
7			addi \$5, \$5, 5	addi \$4, \$4, 4	addi \$3, \$3, 3

"retirement"

# Observations:

Each instruction takes 5 cycles to complete (i.e. until "retirement").

But when the pipeline is saturated, we can complete a new instruction every cycle.

In the previous slide, we can execute 5 instructions in 9 clock cycles, resulting in a CPI (cycles per instruction) of  $9/5=1.8$ .

Without pipelining, CPI would be 5.

# What does each stage do?

## What functional units does it require?

	IF	ID	EX	MEM	WB
What does it do?	Read instruction from memory	Read operand values from register file	Perform arithmetic operation	Read/write data from memory	Write result value into register file
Functional units?	Memory	Register file	ALU	Memory	Register file

addi \$1, \$0, 1  
addi \$1, \$1, 1

Do you seen the problem?

Time	IF	ID	EX	MEM	WB
1					
2					
3					
4					
5					
6					
7					

addi \$1, \$0, 1  
addi \$1, \$1, 1

Time	IF	ID	EX	MEM	WB
1	addi \$1, \$0, 1				
2					
3					
4					
5					
6					
7					

addi \$1, \$0, 1  
addi \$1, \$1, 1

Time	IF	ID	EX	MEM	WB
1	addi \$1, \$0, 1				
2	addi \$1, \$1, 1	addi \$1, \$0, 1			
3					
4					
5					
6					
7					



addi \$1, \$0, 1  
addi \$1, \$1, 1

Time	IF	ID	EX	MEM	WB
1	addi \$1, \$0, 1				
2	addi \$1, \$1, 1	addi \$1, \$0, 1			
3		addi \$1, \$1, 1	addi \$1, \$0, 1		
4					
5					
6					
7					

addi \$1, \$0, 1  
addi \$1, \$1, 1

Reading a value  
from \$1 before it's  
been written.  
Race condition!

Time	IF	ID	EX	MEM	WB
1	addi \$1, \$0, 1				
2	addi \$1, \$1, 1	addi \$1, \$0, 1			
3		addi \$1, \$1, 1	addi \$1, \$0, 1		
4					
5					
6					
7					

addi \$1, \$0, 1  
addi \$1, \$1, 1

Solution one: "stall", insert a bubble

We can't advance out  
of IF until our  
dependencies are met

Time	IF	ID	EX	WB
1	addi \$1, \$0, 1			
2	addi \$1, \$1, 1	addi \$1, \$0, 1		
3	addi \$1, \$1, 1		addi \$1, \$0, 1	
4				
5				
6				
7				

addi \$1, \$0, 1  
addi \$1, \$1, 1

Solution one: "stall", insert a bubble

Time	IF	ID	EX	MEM	WB
1	addi \$1, \$0, 1				
2	addi \$1, \$1, 1	addi \$1, \$0, 1			
3	addi \$1, \$1, 1		addi \$1, \$0, 1		
4	addi \$1, \$1, 1			addi \$1, \$0, 1	
5	addi \$1, \$1, 1				addi \$1, \$0, 1
6		addi \$1, \$1, 1			
7			addi \$1, \$1, 1		

addi \$1, \$0, 1  
addi \$1, \$1, 1

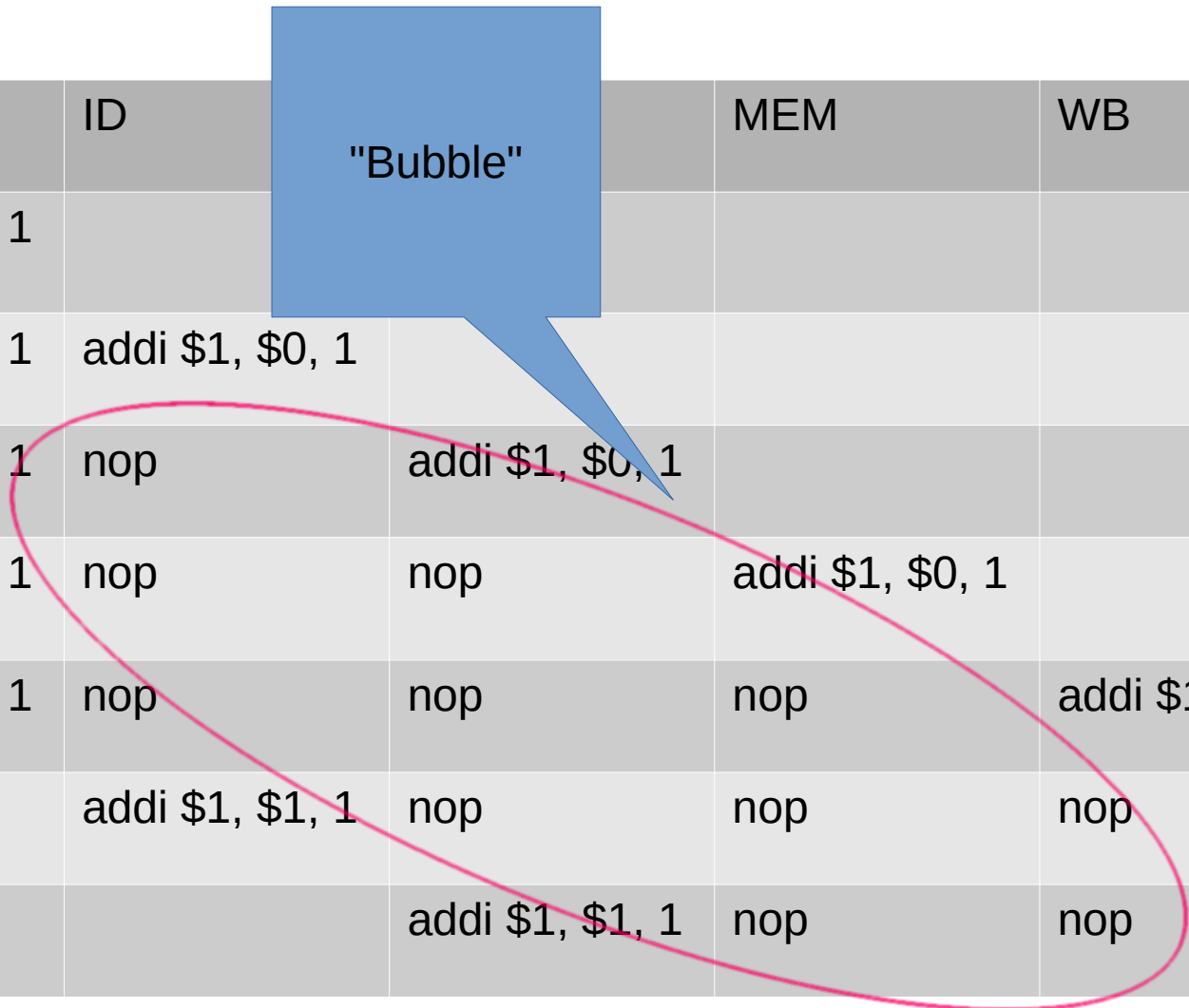
Solution one: "stall", insert a bubble

Time	IF	ID	EX	MEM	WB
1	addi \$1, \$0, 1				
2	addi \$1, \$1, 1	addi \$1, \$0, 1			
3	addi \$1, \$1, 1	nop	addi \$1, \$0, 1		
4	addi \$1, \$1, 1	nop	nop	addi \$1, \$0, 1	
5	addi \$1, \$1, 1	nop	nop	nop	addi \$1, \$0, 1
6		addi \$1, \$1, 1	nop	nop	nop
7			addi \$1, \$1, 1	nop	nop

addi \$1, \$0, 1  
addi \$1, \$1, 1

Solution one: "stall", insert a bubble

Time	IF	ID	"Bubble"	MEM	WB
1	addi \$1, \$0, 1				
2	addi \$1, \$1, 1	addi \$1, \$0, 1			
3	addi \$1, \$1, 1	nop	addi \$1, \$0, 1		
4	addi \$1, \$1, 1	nop	nop	addi \$1, \$0, 1	
5	addi \$1, \$1, 1	nop	nop	nop	addi \$1, \$0, 1
6		addi \$1, \$1, 1	nop	nop	nop
7			addi \$1, \$1, 1	nop	nop

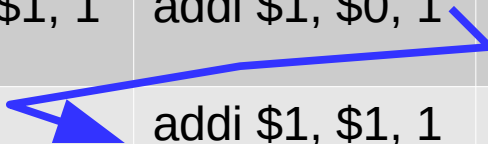


addi \$1, \$0, 1  
addi \$1, \$1, 1

Solution two: "forwarding"

Here: EX-to-EX forwarding

Time	IF	ID	EX	MEM	WB
1	addi \$1, \$0, 1				
2	addi \$1, \$1, 1	addi \$1, \$0, 1			
3		addi \$1, \$1, 1	addi \$1, \$0, 1		
4			addi \$1, \$1, 1	addi \$1, \$0, 1	
5				addi \$1, \$1, 1	addi \$1, \$0, 1
6					addi \$1, \$1, 1
7					



addi \$1, \$0, 1

movi \$2, 1

addi \$1, \$1, 1

???????

Time	IF	ID	EX	MEM	WB
1					
2					
3					
4					
5					
6					
7					

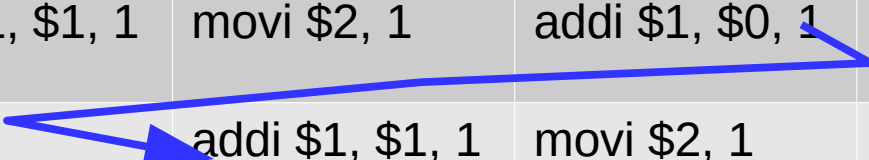


addi \$1, \$0, 1  
movi \$2, 1  
addi \$1, \$1, 1

Solution two: "forwarding"

Here: EX-to-ID forwarding

Time	IF	ID	EX	MEM	WB
1	addi \$1, \$0, 1				
2	movi \$2, 1	addi \$1, \$0, 1			
3	addi \$1, \$1, 1	movi \$2, 1	addi \$1, \$0, 1		
4		addi \$1, \$1, 1	movi \$2, 1	addi \$1, \$0, 1	
5			addi \$1, \$1, 1	movi \$2, 1	addi \$1, \$0, 1
6				addi \$1, \$1, 1	movi \$2, 1
7					addi \$1, \$1, 1



lw \$1, 1(\$0)  
addi \$1, \$1, 1

??????

Time	IF	ID	EX	MEM	WB
1					
2					
3					
4					
5					
6					
7					

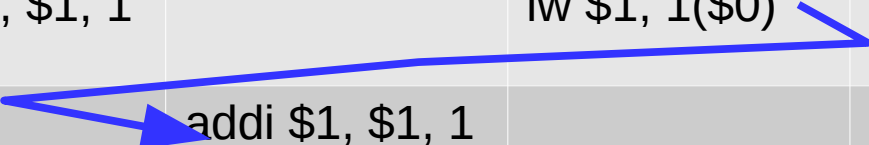
lw \$1, 1(\$0)  
addi \$1, \$1, 1

Solution two: "forwarding"

Here: MEM-to-EX forwarding

Note that even with forwarding, we still need to stall

Time	IF	ID	EX	MEM	WB
1	lw \$1, 1(\$0)				
2	addi \$1, \$1, 1	lw \$1, 1(\$0)			
3		addi \$1, \$1, 1	lw \$1, 1(\$0)		
4		addi \$1, \$1, 1		lw \$1, 1(\$0)	
5			addi \$1, \$1, 1		lw \$1, 1(\$0)
6				addi \$1, \$1, 1	
7					addi \$1, \$1, 1



foo:  
addi \$4, \$4, 1  
jeq \$1, \$2, foo  
addi \$3, \$3, 2

Do you seen the problem?

Time	IF	ID	EX	MEM	WB
1					
2					
3					
4					
5					
6					
7					

```
foo:
addi $4, $4, 1
jeq $1, $2, foo
addi $3, $3, 2
```

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2					
3					
4					
5					
6					
7					

```
foo:
addi $4, $4, 1
jeq $1, $2, foo
addi $3, $3, 2
```

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3					
4					
5					
6					
7					

```
foo:
addi $4, $4, 1
jeq $1, $2, foo
addi $3, $3, 2
```

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3	?????	jeq \$1, \$2, foo	addi \$4, \$4, 1		
4					
5					
6					
7					

foo:  
 addi \$4, \$4, 1  
 jeq \$1, \$2, foo  
 addi \$3, \$3, 2

One possibility: just stall until jeq  
 completes EX stage.

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3	nop	jeq \$1, \$2, foo	addi \$4, \$4, 1		
4	nop	nop	jeq \$1, \$2, foo	addi \$4, \$4, 1	
5	addi \$3, \$3, 2 OR addi \$4, \$4, 1	nop	nop	jeq \$1, \$2, foo	addi \$4, \$4, 1
6					
7					



```
foo:
addi $4, $4, 1
jeq $1, $2, foo
addi $3, $3, 2
```

Solution one: predict branch not taken

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3	addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1		
4					
5					
6					
7					

```
foo:
addi $4, $4, 1
jeq $1, $2, foo
addi $3, $3, 2
```

Solution one: predict branch not taken

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3	addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1		
4		addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1	
5					
6					
7					

```
foo:
addi $4, $4, 1
jeq $1, $2, foo
addi $3, $3, 2
```

Solution one: predict branch not taken

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3	addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1		
4		addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1	
5			addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1
6					
7					

```
foo:
addi $4, $4, 1
jeq $1, $2, foo
addi $3, $3, 2
```

Solution one: predict branch not taken

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3	addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1		
4		addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1	
5			addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1
6					
7					

At time 5, jeq has completed EXEC, and we know if the branch was really taken. If it *was taken*, great. If not, we have to kill the pipeline.

foo:  
addi \$4, \$4, 1  
jeq \$1, \$2, foo  
addi \$3, \$3, 2

Solution one: predict branch not taken  
Case one: prediction correct

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3	addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1		
4		addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1	
5			addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1
6					
7					

foo:  
 addi \$4, \$4, 1  
 jeq \$1, \$2, foo  
 addi \$3, \$3, 2

Solution one: predict branch not taken  
 Case one: prediction correct

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3	addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1		
4		addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1	
5			addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1
6				addi \$3, \$3, 2	jeq \$1, \$2, foo
7					

foo:  
 addi \$4, \$4, 1  
 jeq \$1, \$2, foo  
 addi \$3, \$3, 2

Solution one: predict branch not taken  
 Case one: prediction correct

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3	addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1		
4		addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1	
5			addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1
6				addi \$3, \$3, 2	jeq \$1, \$2, foo
7					addi \$3, \$3, 2

foo:  
 addi \$4, \$4, 1  
 jeq \$1, \$2, foo  
 addi \$3, \$3, 2

Solution one: predict branch not taken  
 Case two: prediction incorrect

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3	addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1		
4		addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1	
5	`		addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1
6					
7					



foo:  
 addi \$4, \$4, 1  
 jeq \$1, \$2, foo  
 addi \$3, \$3, 2

Solution one: predict branch not taken  
 Case two: prediction incorrect

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3	addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1		
4		addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1	
5			addi \$ <del>4</del> , \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1
6					
7					

foo:  
 addi \$4, \$4, 1  
 jeq \$1, \$2, foo  
 addi \$3, \$3, 2

Solution one: predict branch not taken  
 Case two: prediction incorrect

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3	addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1		
4		addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1	
5	addi \$4, \$4, 1		addi \$3, \$3, 2	jeq \$1, \$2, foo	addi \$4, \$4, 1
6		addi \$4, \$4, 1			jeq \$1, \$2, foo
7			addi \$4, \$4, 1		

```
foo:
addi $4, $4, 1
jeq $1, $2, foo
addi $3, $3, 2
```

Solution two: predict branch taken

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3	addi \$4, \$4, 1	jeq \$1, \$2, foo	addi \$4, \$4, 1		
4		addi \$4, \$4, 1	jeq \$1, \$2, foo	addi \$4, \$4, 1	
5			addi \$4, \$4, 1	jeq \$1, \$2, foo	addi \$4, \$4, 1
6					
7					

foo:  
 addi \$4, \$4, 1  
 jeq \$1, \$2, foo  
 addi \$3, \$3, 2

Solution two: predict branch taken  
 Case one: prediction correct

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3	addi \$4, \$4, 1	jeq \$1, \$2, foo	addi \$4, \$4, 1		
4		addi \$4, \$4, 1	jeq \$1, \$2, foo	addi \$4, \$4, 1	
5			addi \$4, \$4, 1	jeq \$1, \$2, foo	addi \$4, \$4, 1
6				addi \$4, \$4, 1	jeq \$1, \$2, foo
7					addi \$4, \$4, 1

foo:  
 addi \$4, \$4, 1  
 jeq \$1, \$2, foo  
 addi \$3, \$3, 2

Solution two: predict branch taken  
 Prediction incorrect

Time	IF	ID	EX	MEM	WB
1	addi \$4, \$4, 1				
2	jeq \$1, \$2, foo	addi \$4, \$4, 1			
3	addi \$4, \$4, 1	jeq \$1, \$2, foo	addi \$4, \$4, 1		
4		addi \$4, \$4, 1	jeq \$1, \$2, foo	addi \$4, \$4, 1	
5	addi \$3, \$3, 2		addi \$ <del>4</del> , \$4, 1	jeq \$1, \$2, foo	addi \$4, \$4, 1
6		addi \$3, \$3, 2			jeq \$1, \$2, foo
7			addi \$3, \$3, 2		

# Pipeline hazards:

## Data hazards

Data hazards may occur when a pipelined instruction shares the use of a register or memory location with another instruction concurrently in the pipeline. In E20, we are principally concerned with so-called *read-after-write* dependencies, wherein one instruction attempts to read data before that data has been written.

```
movi $1, 0  
add $2, $2, $1
```

```
lw $1, foo($0)  
add $2, $2, $1
```

```
sw $1, foo($0)  
lw $2, foo($0)
```

# Pipeline hazards:

## Data hazards

Data hazards may occur when a pipelined instruction shares the use of a register or memory location with another instruction concurrently in the pipeline. In E20, we are principally concerned with so-called *read-after-write* dependencies, wherein one instruction attempts to read data before that data has been written.

```
movi $1, 0  
add $2, $2, $1
```

```
lw $1, foo($0)  
add $2, $2, $1
```

```
sw $1, foo($0)  
lw $2, foo($0)
```

In each of the above three program fragments, the first instruction writes a value, and the second instruction reads the same value. The bolded term identifies the conflicted resource. Note that in the first two fragments, the conflicted resource is a register, while in the third fragment, the conflicted resource is a memory cell.

Read-after-write data hazards can be handled by the processor by stalling, i.e. automatically inserting a bubble (one or more nop instructions) into the pipeline. More sophisticated processors may be able to handle hazards with the use of *forwarding*, wherein data is moved between stages before it is fully written.

Less sophisticated processors may not have hardware to detect data hazards, resulting in possible race conditions. In that case, the assembler must detect hazards and insert nops into the instruction stream before the program is loaded.

Does this program have a  
conflict/dependency?  
Does it have a hazard?

```
movi $1, 0  
add $2, $2, $1
```

Does this program have a  
conflict/dependency?  
Does it have a hazard?

```
movi $1, 0  
addi $3, $3, 3  
addi $4, $4, 3  
addi $5, $5, 3  
addi $6, $6, 3  
addi $7, $7, 3  
add $2, $2, $1
```



Does this program have a  
conflict/dependency?  
Does it have a hazard?

```
movi $1, 0  
add $2, $2, $1
```

Yes, we have a read-after-write conflict on \$1, which results in a hazard. We can solve the hazard with a stall or forwarding, depending on the processor.

Does this program have a  
conflict/dependency?  
Does it have a hazard?

```
movi $1, 0  
addi $3, $3, 3  
addi $4, $4, 3  
addi $5, $5, 3  
addi $6, $6, 3  
addi $7, $7, 3  
add $2, $2, $1
```

We have the same conflict, but no hazard, because the pipeline will be clear by the time we get around to executing the add.

# Pipeline hazards:

## Data hazards

Consider the following program excerpt:

```
1. movi $1, 5
2. add $2, $2, $1
3. add $3, $3, $3
```

Now consider this version:

```
1. movi $1, 5
3. add $3, $3, $3
2. add $2, $2, $1
```

Answer two questions:

- Are these programs equivalent? That is, will they produce the same results in all cases? In other words, is it *safe* to order the first program into the second one?
- Is there any performance difference between these programs? Assume data hazards are resolved with stalls.

# Pipeline hazards:

## Data hazards

Sometimes we can avoid stalls by reordering instructions. Reordering can be done:

- Manually, by the assembly-language programmer.
- Automatically, by the compiler.
- Automatically, by the processor (so-called *out-of-order* execution).

For example, consider the following program excerpt:

```
1. movi $1, 5
2. add $2, $2, $1
3. add $3, $3, $3
```

There is a conflict between the first two instructions that could cause a stall of 3 cycles, while instruction 1 finishes WB and instruction 2 enters ID.

Now consider this version:

```
1. movi $1, 5
3. add $3, $3, $3
2. add $2, $2, $1
```

The program produces identical results, but completes one cycle sooner. Draw the cycle execution table to convince yourself of this fact.

### ORIGINAL VERSION

1  
2 1  
2 1  
2 1  
2 1  
3 2  
3 2  
3 2  
3 2  
3

10 cycles

### REORDERED VERSION

1  
3 1  
2 3 1  
2 3 1  
2 3 1  
2 3  
2  
2  
2

9 cycles

# Pipeline hazards:

## Data hazards

Reordering is tricky.

We must be careful that our reordered program produces the correct result. For example, consider the following code:

```
movi $1, 5
add $2, $2, $1
```

There is a read-after-write hazard on \$1, which could cause a stall. So, let's reorder the code like this:

```
add $2, $2, $1
movi $1, 5
```

We no longer have a read-after-write hazard, and can therefore avoid a stall. BUT this reordering is an *invalid transformation* (i.e. it is unsafe): it produces a different result than the original. A principal policy of reordering instructions is that the new version must produce identical results as the original. To prevent an invalid transformation, we should abide by this rule:

**Never change the relative order of two instructions that have a conflict, including read-after-write, write-after-write, and write-after-read.**

In this case, the read-after-write hazard prevents us from reordering them.

# Data conflicts (aka data dependencies)

## Not hazards!

We've already discussed read-after-write conflicts. Other data conflicts:

- *Write-after-read* conflict

addi \$1, **\$2**, 1

addi **\$2**, \$3, 1

This is a conflict, but is a concern only when instructions are potentially re-ordered.

- *Write-after-write* conflict

addi **\$1**, \$2, 1

addi **\$1**, \$3, 1

This is a conflict, but is a concern only when instructions are potentially re-ordered.

- *Read-after-read* non-conflict

addi \$1, **\$2**, 1

addi \$3, **\$2**, 1

This is not a conflict, and cannot cause a hazard. Instructions may read the same value without conflicting.

# Pipeline hazards:

## Control hazards

Control hazards occur when a pipelined instruction makes a wrong decision in branch prediction, causing some partially-executed instructions to be discarded.

Assume that we have a processor that always predicts branch-not-taken, i.e. the processor will pipeline the instruction immediately subsequent to a conditional jump.

In the following program, the processor will pipeline `add $2, $2, $2`, even though that instruction will not be executed. Its partial execution must therefore be discarded.

```
        movi $1, 0
        jeq $1, $0, somewhere
        add $2, $2, $2
somewhere:
        add $3, $3, $3
```

What specifically will cause a control hazard depends on the processor's branch prediction logic. Sophisticated processors will use clever techniques to reduce wrong prediction and therefore reduce control hazards.

For example, consider a processor that predicts branches in a loop based on the decision taken in the previous iteration.

# Pipeline hazards:

## Control hazards

Another way of thinking about control hazards: a control hazard is a read-after-write data hazard, with the special property of occurring on the program counter, rather than on a regular register or memory address.



# Pipeline hazards:

## Control hazards

Opcode	Can this opcode cause a control hazard?
<b>addi</b>	
<b>jeq</b>	
<b>jr</b>	
<b>j</b>	
<b>jal</b>	

# Pipeline hazards:

## Control hazards

Opcode	Can this opcode cause a control hazard?
<code>addi</code>	No. The address of the next instruction will always be $pc+1$ .
<code>jeq</code>	Yes. The address of the next instruction isn't known until <code>jeq</code> 's EX stage: it may be $pc+imm+1$ or it may be $pc+1$ .
<code>jr</code>	Yes. The address of the next instruction could literally be anything!
<code>j</code>	Maybe. The address of the next instruction is known: it's encoded in the immediate field. However, to prevent a stall, we would need to decode this address immediately in the IF stage, which requires more hardware. Otherwise, every <code>j</code> instruction would cause a stall.
<code>jal</code>	Same as above.

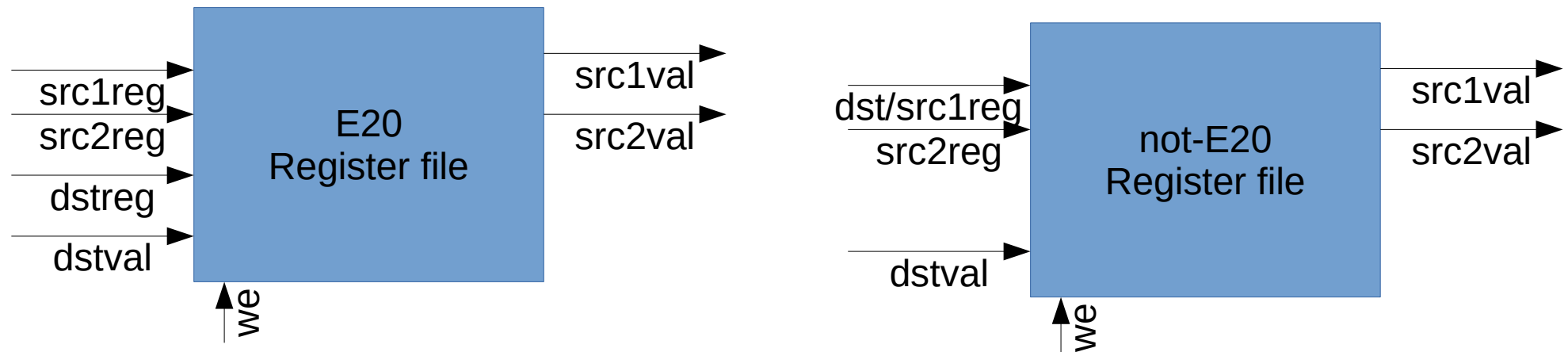
# Pipeline hazards:

## Structural hazards

Structural hazards occur when two pipelined instructions attempt to use the same CPU resource or component at the same time.

Structural hazards are not an issue on the E20 processor, but we can imagine a variant (the "not-E20") that exhibits structural hazards.

Below to the left, we see the actual E20 register file. It has two read ports and one write port, allowing it to read two register values (in the ID stage) and to write one register value (in the WB stage), concurrently.

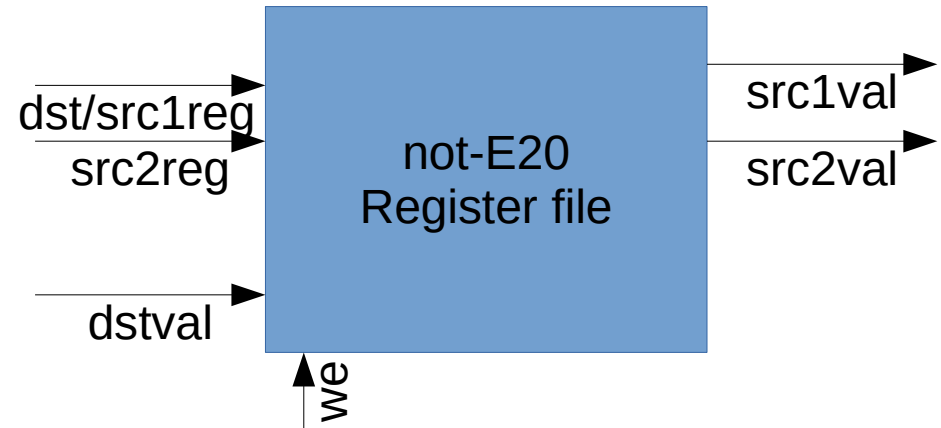


Above to the right, we have the variant not-E20 register file. Unlike the original, it has only two ports, one of which is shared for reading and writing. This means that we cannot simultaneously read and write, and so we cannot concurrently execute the ID and WB stages, resulting in a structural hazard.

# Pipeline hazards:

## Structural hazards

add \$1, \$1, \$1  
 add \$2, \$2, \$2  
 add \$3, \$3, \$3  
 add \$4, \$4, \$4



The following execution takes place on a variant not-E20 processor, to demonstrate the potential for structural hazards.

Time	IF	ID	EX	MEM	WB
1	add \$1, \$1, \$1				
2	add \$2, \$2, \$2	add \$1, \$1, \$1			
3	add \$3, \$3, \$3	add \$2, \$2, \$2	add \$1, \$1, \$1		
4	add \$4, \$4, \$4	add \$3, \$3, \$3	add \$2, \$2, \$2	add \$1, \$1, \$1	
5		add \$4 <del>X</del> , \$4, \$4	add \$3, \$3, \$3	add \$2, \$2, \$2	add \$1, \$1, \$1
6				add \$3, \$3, \$3	add \$2, \$2, \$2
7					add \$3, \$3, \$3

Hazard exists between the two marked instructions, because both are trying to use the **dst/src1reg** port of the register file concurrently.

# Branch prediction, speculative execution

```
movi $1, 50
loop:
jeq $1, $0, done
addi $1, $1, -1
j loop
done:
halt
```

```
movi $1, 50
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
halt
```

In general, what's the best branch prediction strategy, if we want to maximize correct predictions?

# Branch prediction, speculative execution

```
movi $1, 50
loop:
jeq $1, $0, done
addi $1, $1, -1
j loop
done:
halt
```

# Branch is usually NOT taken

```
movi $1, 50
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
halt
```

# Branch usually IS taken

In general, what's the best branch prediction strategy, if we want to maximize correct predictions?

```
no branch
no branch
no branch
no branch
.....
BRANCH
halt
```

```
branch
branch
branch
branch
...
NO BRANCH
halt
```

# Branch prediction, speculative execution

```
movi $1, 50
loop:
jeq $1, $0, done
addi $1, $1, -1
j loop
done:
halt
```

# Branch is usually NOT taken

```
movi $1, 50
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
halt
```

# Branch usually IS taken

Let's predict branch if the instruction branched last time. Let's predict no branch if the instruction didn't branch last time.  
For loops, this will usually be right.

We maintain a special 1-bit register for each jeq instruction. Each time we execute the jeq, we store 1 if it branched, 0 if it didn't. We use this value to predict branching or non-branching in the next iteration.

# Multilevel branch prediction



# One-Level Branch Predictor

Let's consider the first program:

```
    movi $1, 50
loop:
jeq $1, $0, done
    addi $1, $1, -1
    j loop
done:
    halt
```

If we iterate through it, and change our special register, this is what we expect:

```
0
0
0
. . .
1
halt
```

Special Register	State Name
0	Not Taken
1	Taken

# One-Level Branch Predictor

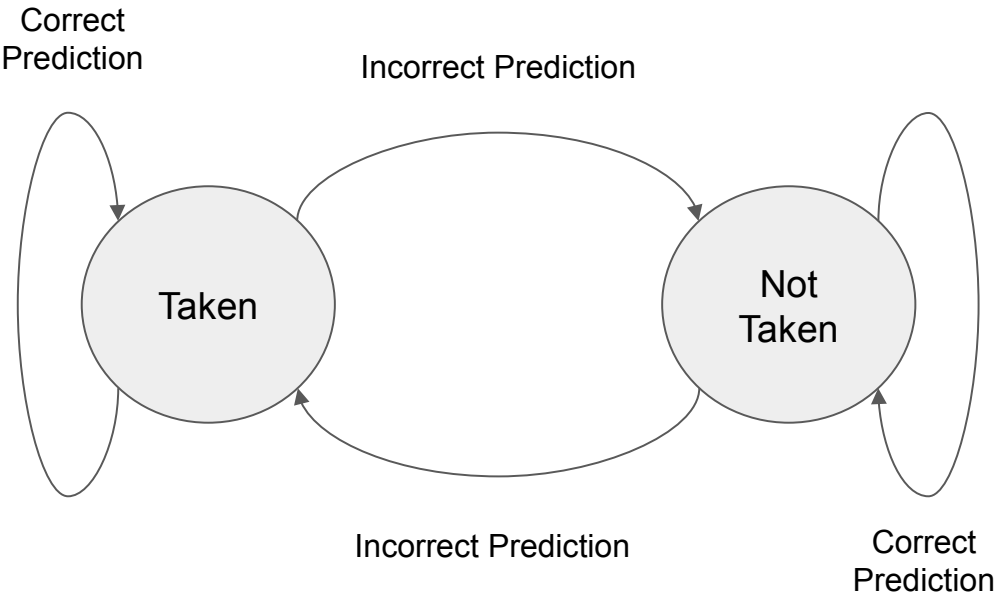
Let's consider the first program:

```
        movi $1, 50
loop:   jeq $1, $0, done
        addi $1, $1, -1
        j loop
done:   halt
```

If we iterate through it, and change our special register, this is what we expect:

We can also represent this using a state diagram and state table where the state changes if the prediction was incorrect and does not if the prediction was correct.

```
0
0
0
. . .
1
halt
```



Special Register	State Name
0	Not Taken
1	Taken

# One-Level Branch Predictor

Consider the program on the right

```
start:  
movi $1, 3  
loop:  
addi $1, $1, -1  
slti $2, $1, 1  
jeq $2, $0, loop  
j start  
  
halt
```

# One-Level Branch Predictor

Consider the program on the right

It does **not** jump every third loop.

What do you expect to be the pattern of the special register?

```
start:
movi $1, 3
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
j start

halt
```

# One-Level Branch Predictor

Consider the program on the right

It does **not** jump every third loop.

What do you expect to be the pattern of the special register?

taken, taken, not taken, taken, taken, not taken, . . .

1 , 1 , 0 , 1 , 1 , 0 , . . .

Do you spot an issue?

```
start:
movi $1, 3
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
j start

halt
```

# One-Level Branch Predictor

Consider the program on the right

It does **not** jump every third loop.

What do you expect to be the pattern of the special register?

taken, taken, not taken, taken, taken, not taken, . . .

1 , 1 , 0 , 1 , 1 , 0 , . . .

Do you spot an issue?

The program will mispredict every third iteration, which is inefficient in the long run.

```
start:
movi $1, 3
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
j start

halt
```

# Can we do better?

Instead of predicting only based on the most recent behaviour of the branch, the outcome can be correlated with all previous outcomes of the branch

# Can we do better? Two-Level Branch Predictors

Instead of predicting only based on the most recent behaviour of the branch, the outcome can be correlated with all previous outcomes of the branch

A two-level branch predictor tracks the history of the last  $n$  branch prediction outcomes in a shift register.

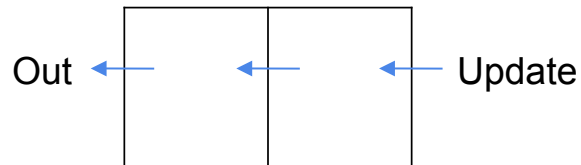


# Can we do better? Two-Level Branch Predictors

Instead of predicting only based on the most recent behaviour of the branch, the outcome can be correlated with all previous outcomes of the branch

A two-level branch predictor tracks the history of the last  $n$  branch prediction outcomes in a shift register. The register shifts left on every update and the update becomes the least significant bit.

' $n$ ' depends on the architecture. This example has  $n = 2$ .

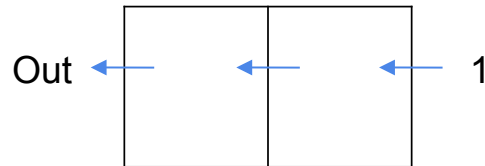


# Can we do better? Two-Level Branch Predictors

Instead of predicting only based on the most recent behaviour of the branch, the outcome can be correlated with all previous outcomes of the branch

A two-level branch predictor tracks the history of the last  $n$  branch prediction outcomes in a shift register. The register shifts left on every update and the update becomes the least significant bit.

' $n$ ' depends on the architecture. This example has  $n = 2$ .

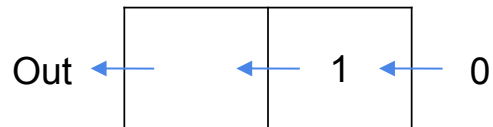


# Can we do better? Two-Level Branch Predictors

Instead of predicting only based on the most recent behaviour of the branch, the outcome can be correlated with all previous outcomes of the branch

A two-level branch predictor tracks the history of the last  $n$  branch prediction outcomes in a shift register. The register shifts left on every update and the update becomes the least significant bit.

' $n$ ' depends on the architecture. This example has  $n = 2$ .



# Can we do better? Two-Level Branch Predictors

Instead of predicting only based on the most recent behaviour of the branch, the outcome can be correlated with all previous outcomes of the branch

A two-level branch predictor tracks the history of the last  $n$  branch prediction outcomes in a shift register. The register shifts left on every update and the update becomes the least significant bit.

' $n$ ' depends on the architecture. This example has  $n = 2$ .

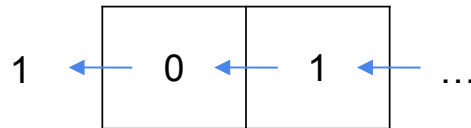


# Can we do better? Two-Level Branch Predictors

Instead of predicting only based on the most recent behaviour of the branch, the outcome can be correlated with all previous outcomes of the branch

A two-level branch predictor tracks the history of the last  $n$  branch prediction outcomes in a shift register. The register shifts left on every update and the update becomes the least significant bit.

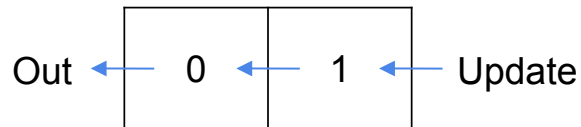
' $n$ ' depends on the architecture. This example has  $n = 2$ .



# Can we do better? Two-Level Branch Predictors

Instead of predicting only based on the most recent behaviour of the branch, the outcome can be correlated with all previous outcomes of the branch.

A two-level branch predictor tracks the history of the last  $n$  branch prediction outcomes in a shift register. The register shifts left on every update and the update becomes the least significant bit.



' $n$ ' depends on the architecture. This example has  $n = 2$ .

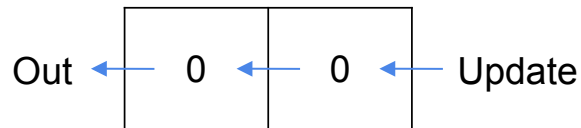
This special register is called the history register.

We get  $2^n$  possible history patterns, based on which we can predict the next branch. So,  $2^2 = 4$  in this case.

# Can we do better? Two-Level Branch Predictors

Let's go back to the example of the loop that does **not** jump every third iteration. Like before, each time we execute the jeq, we store 1 if it branched, 0 if it didn't.

Let's also use a 2-bit shift register and walk through the code



\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7
0	0	0	0	0	0	0	0

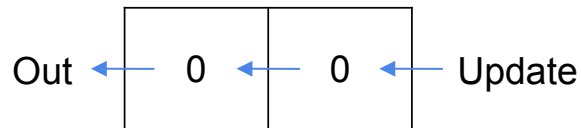
```
start:
movi $1, 3
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
j start

halt
```

# Can we do better? Two-Level Branch Predictors

Let's go back to the example of the loop that does **not** jump every third iteration. Like before, each time we execute the `jeq`, we store 1 if it branched, 0 if it didn't.

Let's also use a 2-bit shift register and walk through the code



\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7
0	3	0	0	0	0	0	0

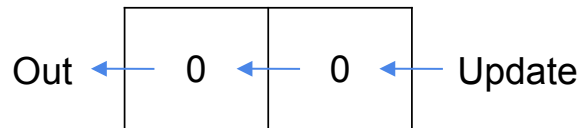
```
start:  
movi $1, 3  
loop:  
addi $1, $1, -1  
slti $2, $1, 1  
jeq $2, $0, loop  
j start  
  
halt
```



# Can we do better? Two-Level Branch Predictors

Let's go back to the example of the loop that does **not** jump every third iteration. Like before, each time we execute the `jeq`, we store 1 if it branched, 0 if it didn't.

Let's also use a 2-bit shift register and walk through the code



\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7
0	2	0	0	0	0	0	0

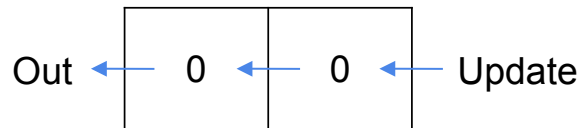
```
start:
movi $1, 3
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
j start

halt
```

# Can we do better? Two-Level Branch Predictors

Let's go back to the example of the loop that does **not** jump every third iteration. Like before, each time we execute the `jeq`, we store 1 if it branched, 0 if it didn't.

Let's also use a 2-bit shift register and walk through the code



\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7
0	2	0	0	0	0	0	0

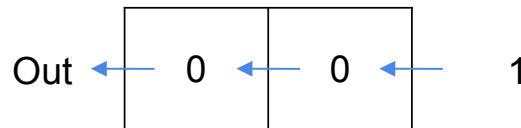
```
start:
movi $1, 3
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
j start

halt
```

# Can we do better? Two-Level Branch Predictors

Let's go back to the example of the loop that does **not** jump every third iteration. Like before, each time we execute the `jeq`, we store 1 if it branched, 0 if it didn't.

Let's also use a 2-bit shift register and walk through the code



\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7
0	2	0	0	0	0	0	0

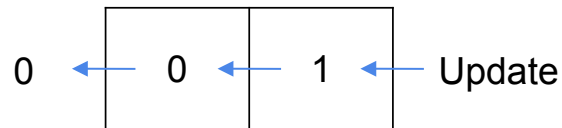
```
start:
movi $1, 3
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
j start

halt
```

# Can we do better? Two-Level Branch Predictors

Let's go back to the example of the loop that does **not** jump every third iteration. Like before, each time we execute the `jeq`, we store 1 if it branched, 0 if it didn't.

Let's also use a 2-bit shift register and walk through the code



\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7
0	1	0	0	0	0	0	0

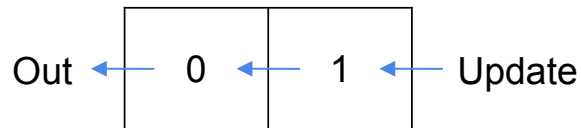
```
start:
movi $1, 3
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
j start

halt
```

# Can we do better? Two-Level Branch Predictors

Let's go back to the example of the loop that does **not** jump every third iteration. Like before, each time we execute the `jeq`, we store 1 if it branched, 0 if it didn't.

Let's also use a 2-bit shift register and walk through the code



\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7
0	1	0	0	0	0	0	0

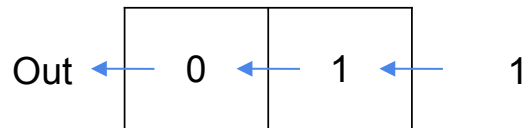
```
start:
movi $1, 3
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
j start

halt
```

# Can we do better? Two-Level Branch Predictors

Let's go back to the example of the loop that does **not** jump every third iteration. Like before, each time we execute the `jeq`, we store 1 if it branched, 0 if it didn't.

Let's also use a 2-bit shift register and walk through the code



\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7
0	1	0	0	0	0	0	0

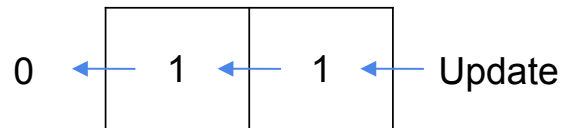
```
start:
movi $1, 3
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
j start

halt
```

# Can we do better? Two-Level Branch Predictors

Let's go back to the example of the loop that does **not** jump every third iteration. Like before, each time we execute the `jeq`, we store 1 if it branched, 0 if it didn't.

Let's also use a 2-bit shift register and walk through the code



\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7
0	0	0	0	0	0	0	0

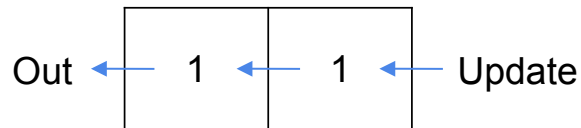
```
start:
movi $1, 3
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
j start

halt
```

# Can we do better? Two-Level Branch Predictors

Let's go back to the example of the loop that does **not** jump every third iteration. Like before, each time we execute the `jeq`, we store 1 if it branched, 0 if it didn't.

Let's also use a 2-bit shift register and walk through the code



\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7
0	0	1	0	0	0	0	0

```
start:
movi $1, 3
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
j start

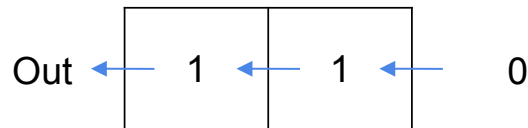
halt
```



# Can we do better? Two-Level Branch Predictors

Let's go back to the example of the loop that does **not** jump every third iteration. Like before, each time we execute the `jeq`, we store 1 if it branched, 0 if it didn't.

Let's also use a 2-bit shift register and walk through the code



\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7
0	0	1	0	0	0	0	0

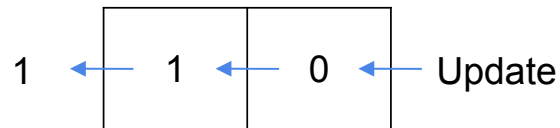
```
start:
movi $1, 3
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
j start

halt
```

# Can we do better? Two-Level Branch Predictors

Let's go back to the example of the loop that does **not** jump every third iteration. Like before, each time we execute the `jeq`, we store 1 if it branched, 0 if it didn't.

Let's also use a 2-bit shift register and walk through the code



\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7
0	0	1	0	0	0	0	0

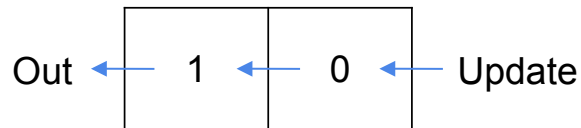
```
start:
movi $1, 3
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
j start

halt
```

# Can we do better? Two-Level Branch Predictors

Let's go back to the example of the loop that does **not** jump every third iteration. Like before, each time we execute the jeq, we store 1 if it branched, 0 if it didn't.

Let's also use a 2-bit shift register and walk through the code



\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7
0	3	1	0	0	0	0	0

```
start:
movi $1, 3
loop:
addi $1, $1, -1
slti $2, $1, 1
jeq $2, $0, loop
j start

halt
```

# Can we do better? Two-Level Branch Predictors

What did the branch history look like?

0 0 1 1 0 1 1 0 1 1 0 1 1 0 . . .

At the start, the processor does not know what to do, but soon, a pattern sets in.

# Can we do better? Two-Level Branch Predictors

What did the branch history look like?

0 0 1 1 0 1 1 0 1 1 0 1 1 0 . . .

At the start, the processor does not know what to do, but soon, a pattern sets in.

Since we chose  $n = 2$ , we get  $2^2 = 4$  possible history patterns:

1. 00
2. 01
3. 10
4. 11

History	Outcome	Observation
00	1	Taken
01	1	Taken
10	1	Taken
11	0	Not taken

# Why “two-level” branch predictor?

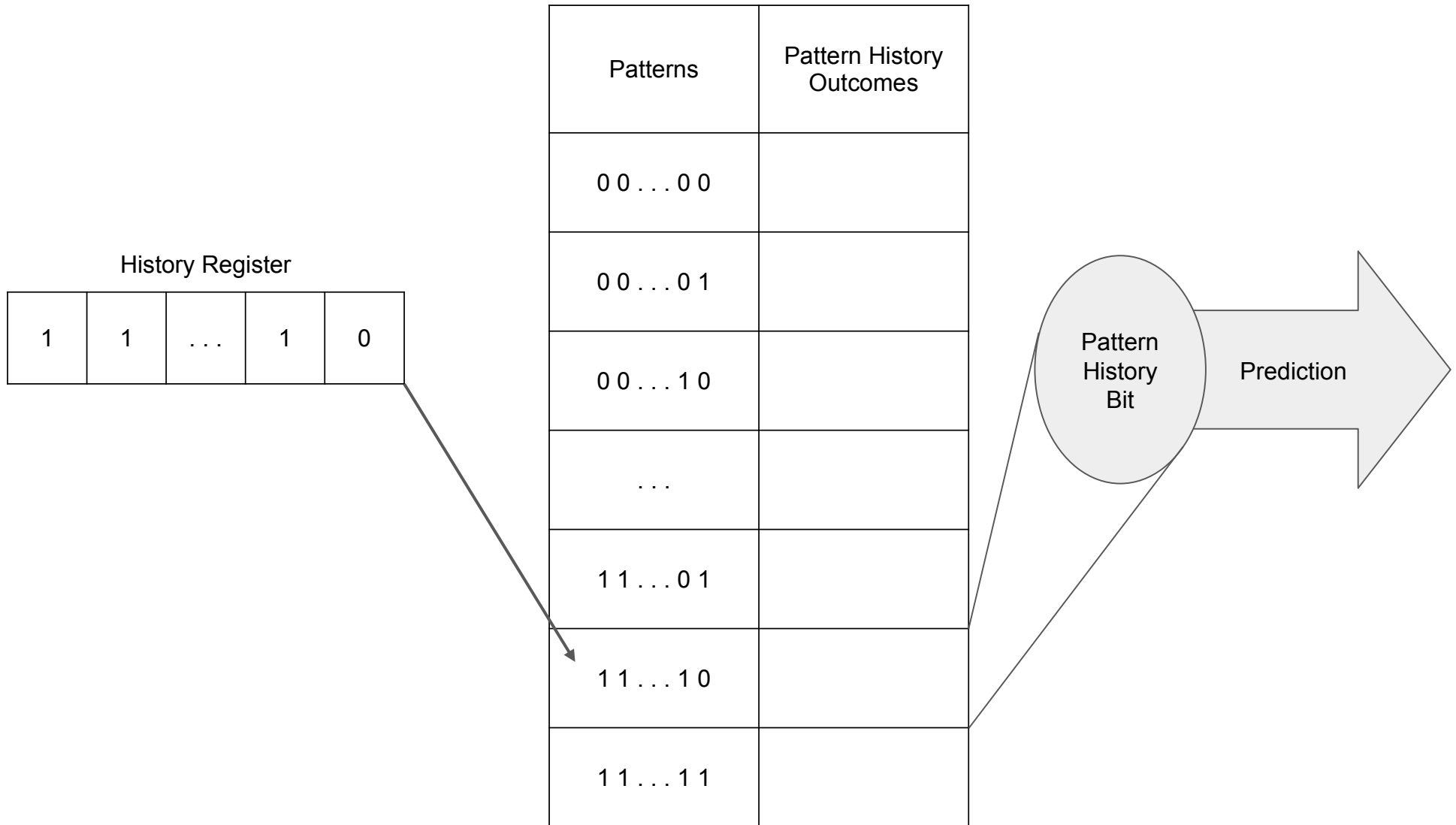
A much larger data structure stores each history register pattern and its outcome, similar to the table we saw above.

This added level of abstraction is the so-called “second-level”.

It is called the **branch history pattern table** or **history register table**. It indexes the histories of all branches in a program. Together, these make the “**pattern table**”.

The two-level branch predictor was developed by T.-Y. Yeh and Yale Patt at the University of Michigan in 1991. Variants of two-level branch predictors are most common in modern microprocessors.

# Why “two-level” branch predictor?



# Pipeline implementation

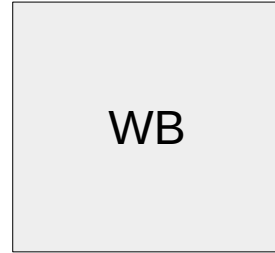
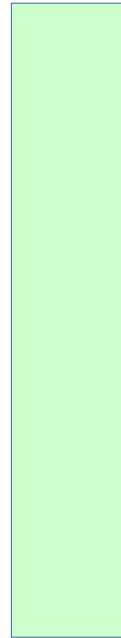
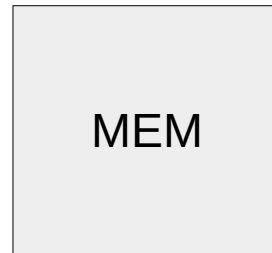
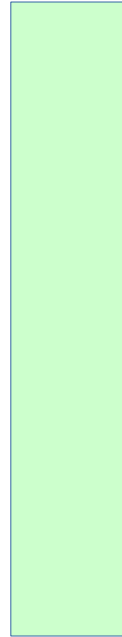
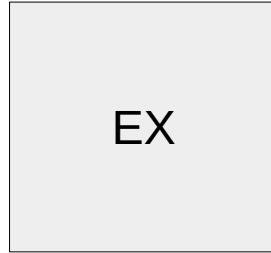
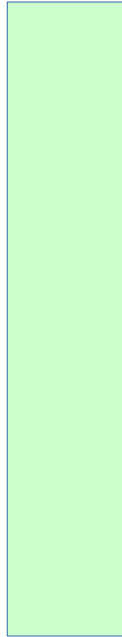
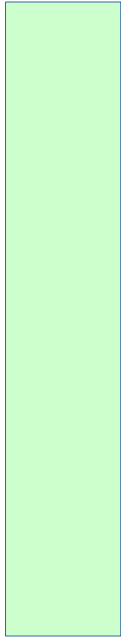


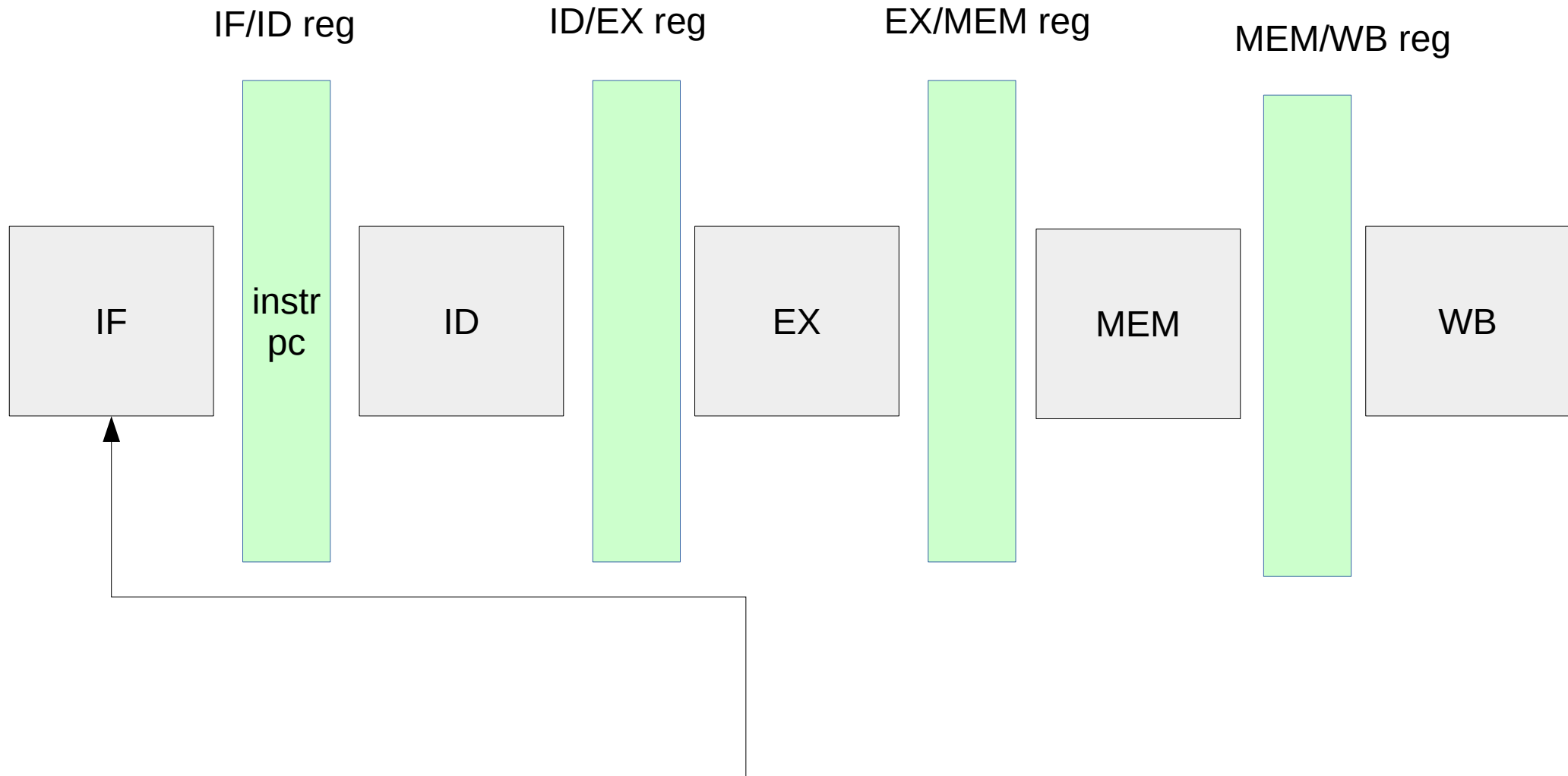
IF/ID reg

ID/EX reg

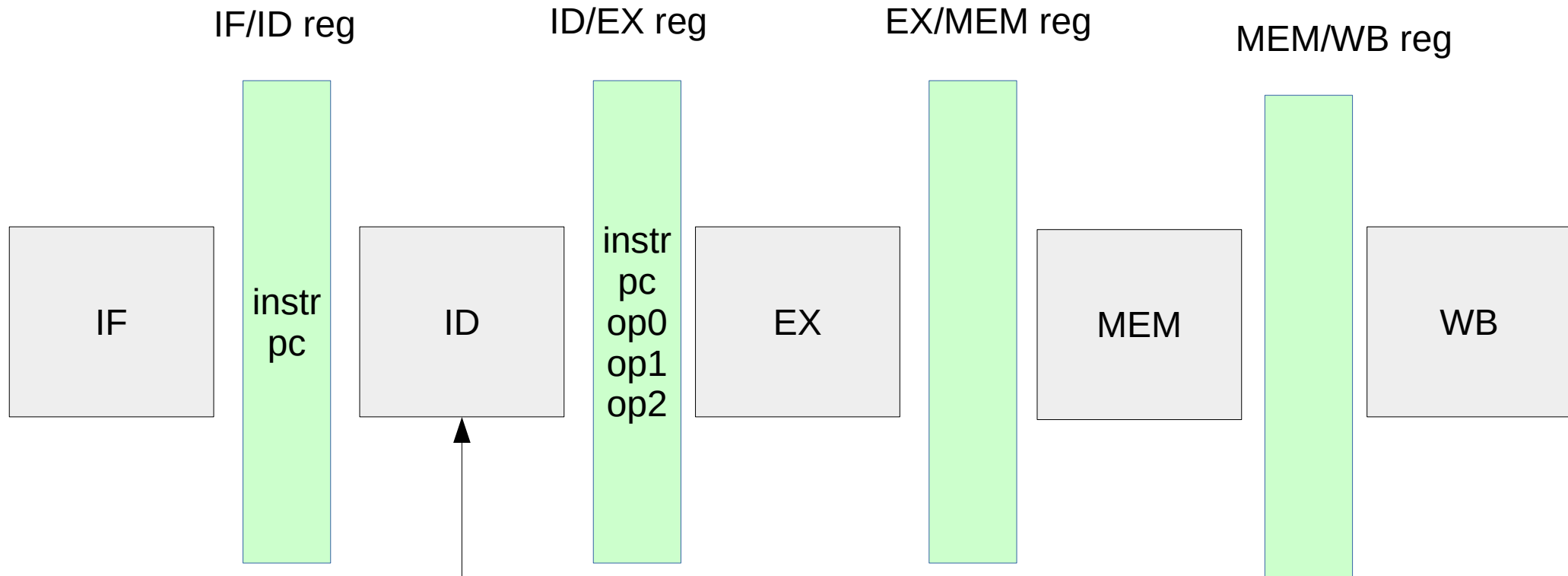
EX/MEM reg

MEM/WB reg

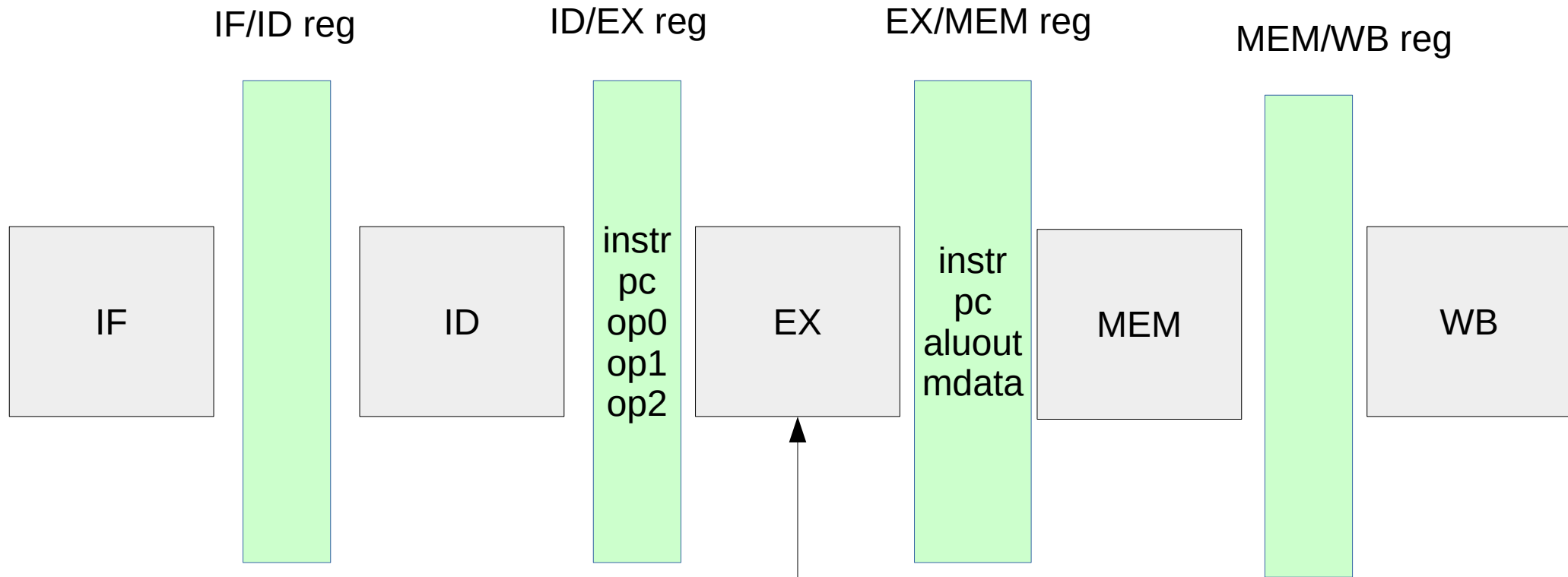




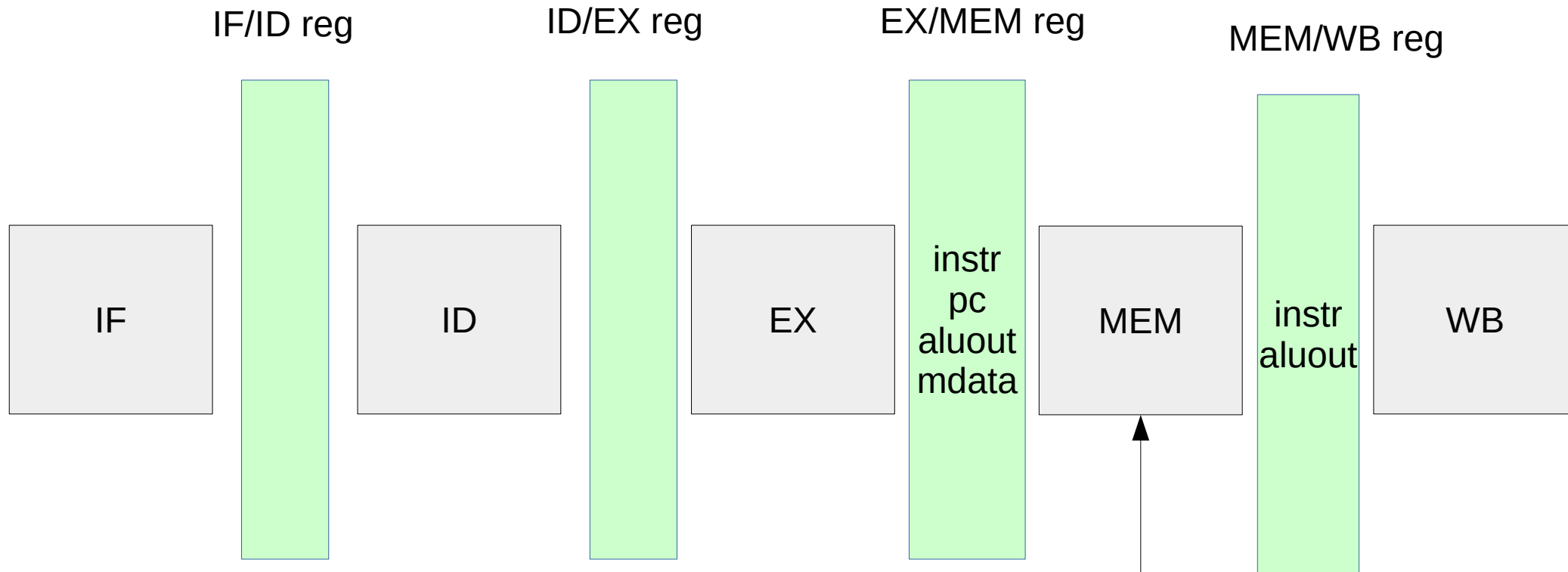
IF stage: read pc, read instruction from memory, store in pipeline reg.



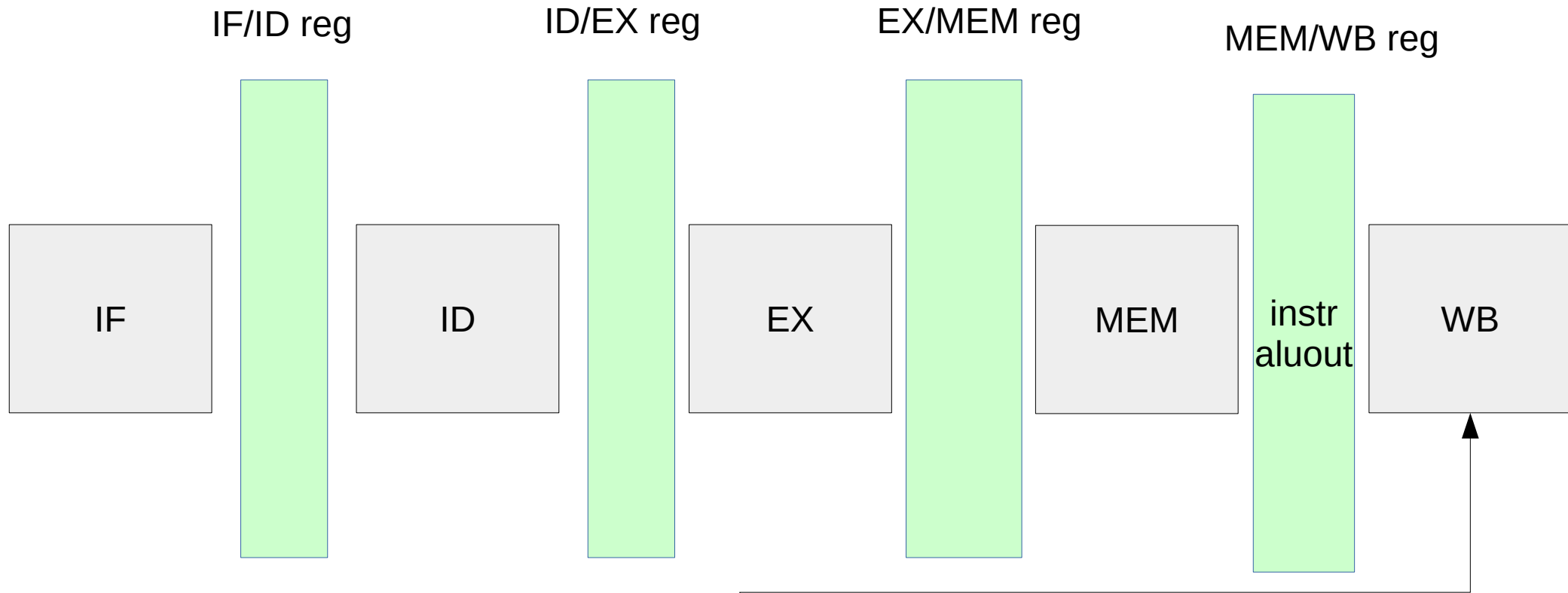
ID stage: based on instr from pipeline reg, sign-extend immediate value and read register values from register file. Store operands in pipeline register.



EX stage: based on instr and operands from pipeline reg, perform ALU operation and update master PC for next instruction.



MEM stage: based on instr (if sw), store mdata into address given by aluout. Aluout is propagated to next pipeline reg.



WB stage: based on instr and aluout, write value to register

**ram[5] = addi \$1, \$2, 3**

**IF**

read instruction from ram,  
store into IF/ID

pc=5, instr=1110100010000011

**ID**

**EX**

**MEM**

**WB**

**ram[5] = addi \$1, \$2, 3**

**IF**

read instruction from ram,  
store into IF/ID

pc=5, instr=1110100010000011

**ID**

read current value of \$2, sign extend 3,  
store into ID/EX

pc=5, instr=1110100010000011, op0=3,  
op1=\$2, op2=\$0

**EX**

**MEM**

**WB**



**ram[5] = addi \$1, \$2, 3**

**IF**

read instruction from ram,  
store into IF/ID

pc=5, instr=1110100010000011

**ID**

read current value of \$2, sign extend 3,  
store into ID/EX

pc=5, instr=1110100010000011, op0=3,  
op1=\$2, op2=\$0

**EX**

add op0 and op1, store into EX/MEM,  
increment master pc

pc=5, instr=1110100010000011,  
aluout=\$2+3, mdata=\$0

**MEM**

**WB**

**ram[5] = addi \$1, \$2, 3**

**IF**

read instruction from ram,  
store into IF/ID

pc=5, instr=1110100010000011

**ID**

read current value of \$2, sign extend 3,  
store into ID/EX

pc=5, instr=1110100010000011, op0=3,  
op1=\$2, op2=\$0

**EX**

add op0 and op1, store into EX/MEM,  
increment master pc

pc=5, instr=1110100010000011,  
aluout=\$2+3, mdata=\$0

**MEM**

do nothing here

instr=1110100010000011, aluout=\$2+3

**WB**

**ram[5] = addi \$1, \$2, 3**

**IF**

read instruction from ram,  
store into IF/ID

pc=5, instr=1110100010000011

**ID**

read current value of \$2, sign extend 3,  
store into ID/EX

pc=5, instr=1110100010000011, op0=3,  
op1=\$2, op2=\$0

**EX**

add op0 and op1, store into EX/MEM,  
increment master pc

pc=5, instr=1110100010000011,  
aluout=\$2+3, mdata=\$0

**MEM**

do nothing here

instr=1110100010000011, aluout=\$2+3

**WB**

write \$2+3 into \$1

Another example.....

**ram[6] = sw \$1, 4(\$2)**

**IF**

read instruction from ram,  
store into IF/ID

pc=6, instr=1010100010000100

**ID**

**EX**

**MEM**

**WB**

**ram[6] = sw \$1, 4(\$2)**

**IF**

read instruction from ram,  
store into IF/ID

pc=6, instr=1010100010000100

**ID**

read current value of \$1 and \$2,  
sign extend 4, store into ID/EX

pc=6, instr=1010100010000100, op0=4,  
op1=\$2, op2=\$1

**EX**

**MEM**

**WB**

**ram[6] = sw \$1, 4(\$2)**

**IF**

read instruction from ram,  
store into IF/ID

pc=6, instr=1010100010000100

**ID**

read current value of \$1 and \$2,  
sign extend 4, store into ID/EX

pc=6, instr=1010100010000100, op0=4,  
op1=\$2, op2=\$1

**EX**

add op0 and op1, store into EX/MEM,  
increment master pc

pc=6, instr=1010100010000100,  
aluout=\$2+4, mdata=\$1

**MEM**

**WB**

**ram[6] = sw \$1, 4(\$2)**

**IF**

read instruction from ram,  
store into IF/ID

pc=6, instr=1010100010000100

**ID**

read current value of \$1 and \$2,  
sign extend 4, store into ID/EX

pc=6, instr=1010100010000100, op0=4,  
op1=\$2, op2=\$1

**EX**

add op0 and op1, store into EX/MEM,  
increment master pc

pc=6, instr=1010100010000100,  
aluout=\$2+4, mdata=\$1

**MEM**

write \$1 to memory address \$2+4

instr=1010100010000100, aluout=\$2+4

**WB**



**ram[6] = sw \$1, 4(\$2)**

**IF**

read instruction from ram,  
store into IF/ID

pc=6, instr=1010100010000100

**ID**

read current value of \$1 and \$2,  
sign extend 4, store into ID/EX

pc=6, instr=1010100010000100, op0=4,  
op1=\$2, op2=\$1

**EX**

add op0 and op1, store into EX/MEM,  
increment master pc

pc=6, instr=1010100010000100,  
aluout=\$2+4, mdata=\$1

**MEM**

write \$1 to memory address \$2+4

instr=1010100010000100, aluout=\$2+4

**WB**

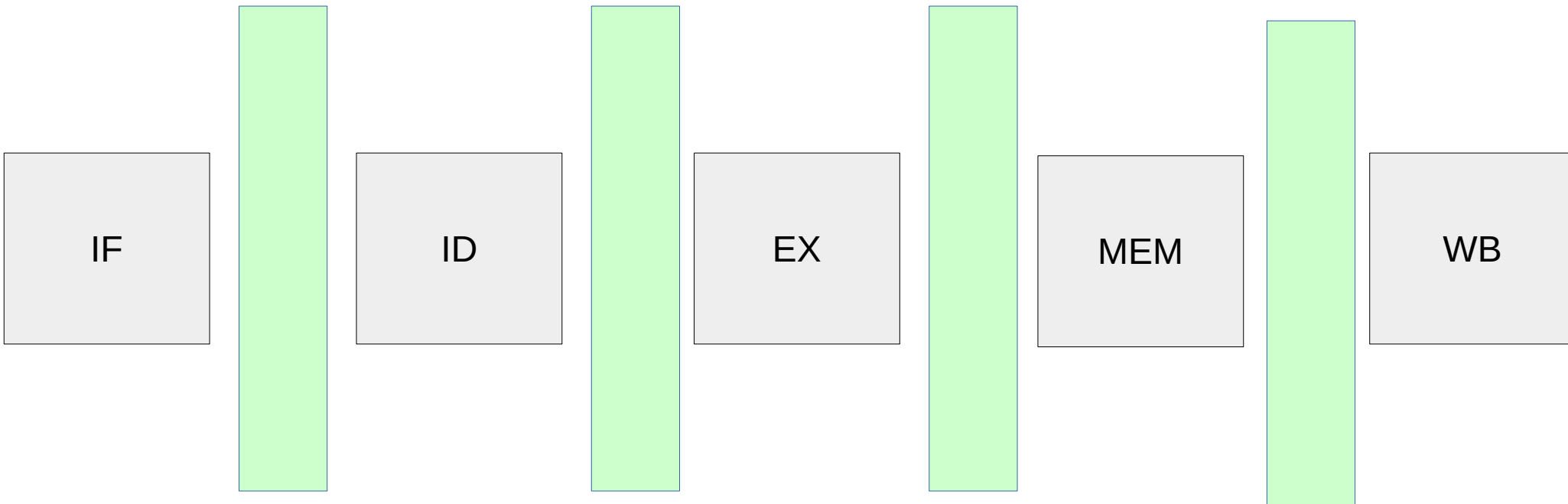
do nothing here

IF/ID reg

ID/EX reg

EX/MEM reg

MEM/WB reg



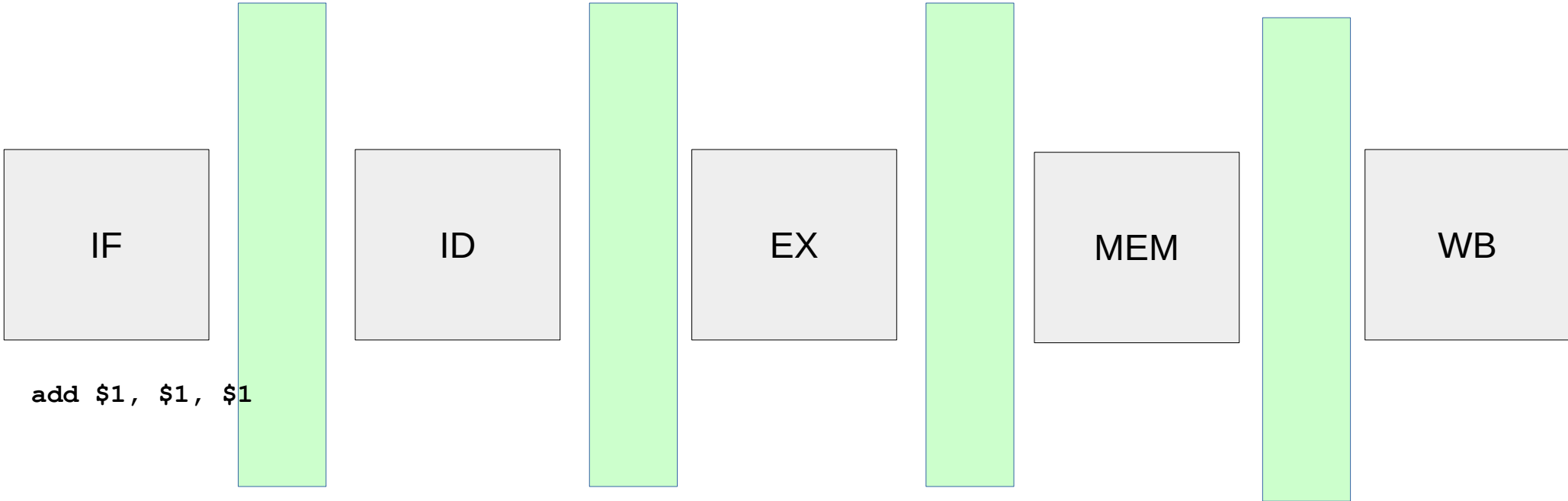
```
add $1, $1, $1
add $2, $2, $2
add $3, $3, $3
```

IF/ID reg

ID/EX reg

EX/MEM reg

MEM/WB reg



`add $1, $1, $1`

`add $1, $1, $1`

`add $2, $2, $2`

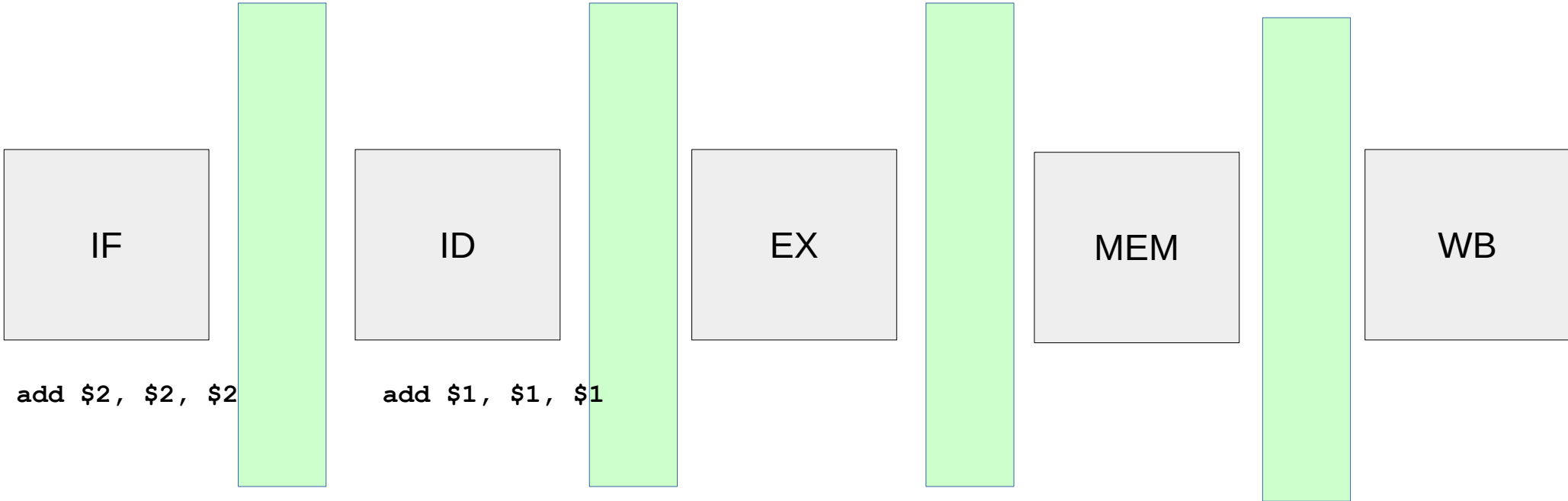
`add $3, $3, $3`

IF/ID reg

ID/EX reg

EX/MEM reg

MEM/WB reg



`add $2, $2, $2`

`add $1, $1, $1`

`add $1, $1, $1`

`add $2, $2, $2`

`add $3, $3, $3`

IF/ID reg

ID/EX reg

EX/MEM reg

MEM/WB reg

IF

ID

EX

MEM

WB

add \$3, \$3, \$3

add \$2, \$2, \$2

add \$1, \$1, \$1

add \$1, \$1, \$1

add \$2, \$2, \$2

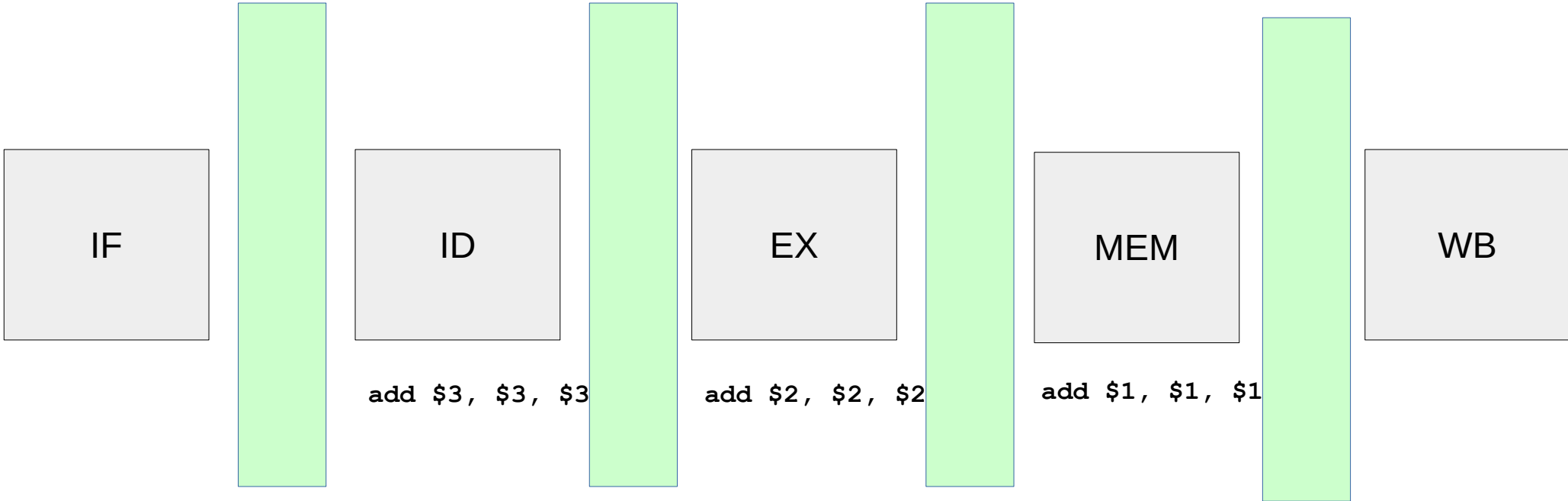
add \$3, \$3, \$3

IF/ID reg

ID/EX reg

EX/MEM reg

MEM/WB reg



`add $1, $1, $1`  
`add $2, $2, $2`  
`add $3, $3, $3`

IF/ID reg

ID/EX reg

EX/MEM reg

MEM/WB reg

IF

ID

EX

MEM

WB

add \$3, \$3, \$3

add \$2, \$2, \$2

add \$1, \$1, \$1

add \$1, \$1, \$1

add \$2, \$2, \$2

add \$3, \$3, \$3

IF/ID reg

ID/EX reg

EX/MEM reg

MEM/WB reg

IF

ID

EX

MEM

WB

add \$3, \$3, \$3

add \$2, \$2, \$

add \$1, \$1, \$1

add \$2, \$2, \$2

add \$3, \$3, \$3



IF/ID reg

ID/EX reg

EX/MEM reg

MEM/WB reg

IF

ID

EX

MEM

WB

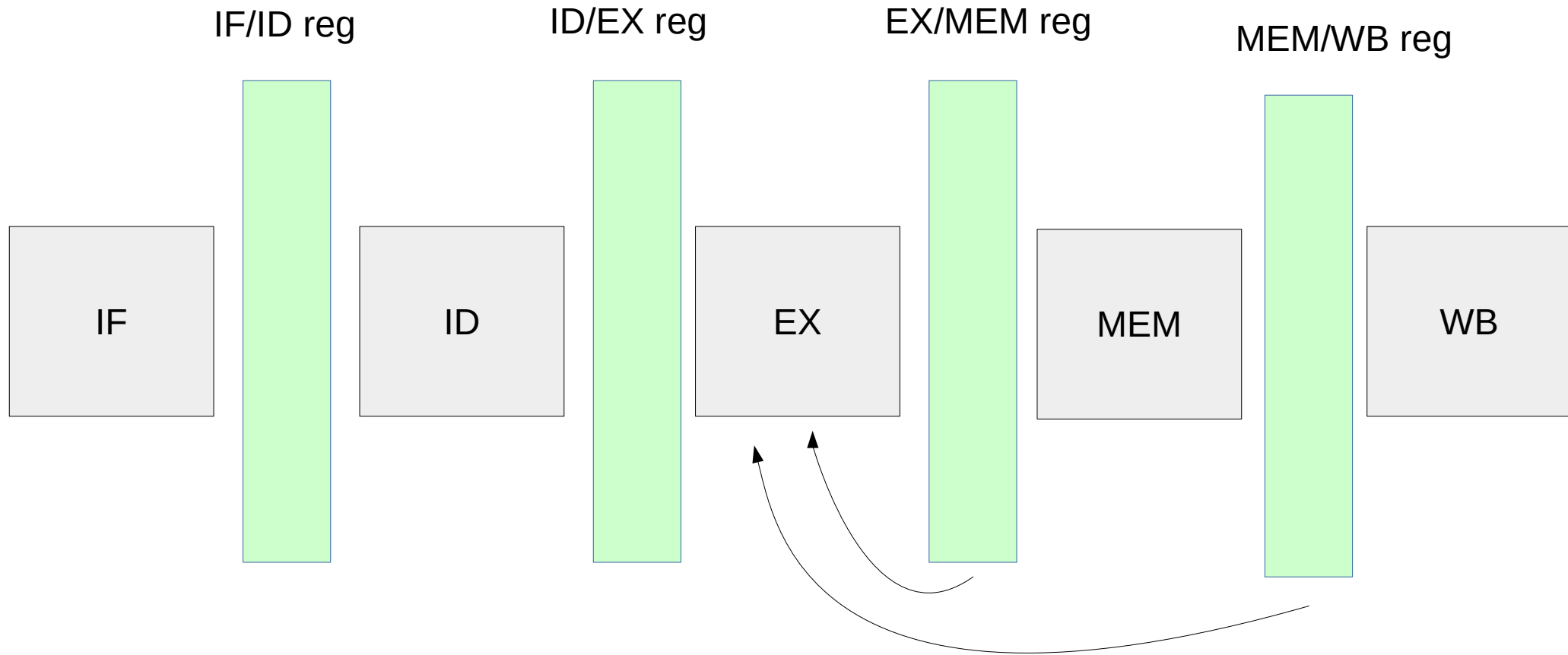
add \$3, \$3, \$3

add \$1, \$1, \$1

add \$2, \$2, \$2

add \$3, \$3, \$3

## E20 pipeline forwarding implementation



E15 pipelining

Instruction	fetch	decode	exec	store
add, addi, sub, subi	instruction ROM	register file, mBus, dBus	ALU	register file, mBus
mov, movi	instruction ROM	register file, mBus, dBus		register file, mBus
cmp, cmpi	instruction ROM	register file, mBus, dBus	ALU	
jmp	instruction ROM			
jz, jnz	instruction ROM			
lw	instruction ROM	register file, mBus, dBus	MMU	register file, mBus
sw	instruction ROM	register file, mBus, dBus	MMU	

```
1 movi, RXX, Rg0, 4'b0101
2 lw,    Rg0, Rg1, 4'b0000
3 add,    Rg0, Rg1, 4'b0000
4 movi, RXX, Rg2, 4'b0000
5 lw,    Rg2, Rg3, 4'b0000
6 add,    Rg3, Rg1, 4'b0000
7 addi, RXX, Rg1, 4'b0000
8 movi, RXX, Rg3, 4'b1000
9 sw,    Rg3, Rg1, 4'b0000
10 sub,   Rg0, Rg1, 4'b0000
```

```
1 movi, RXX, Rg0, 4'b0101
2 lw,    Rg0, Rg1, 4'b0000
3 add,   Rg0, Rg1, 4'b0000
4 movi,  RXX, Rg2, 4'b0000
5 lw,    Rg2, Rg3, 4'b0000
6 add,   Rg3, Rg1, 4'b0000
7 addi,  RXX, Rg1, 4'b0000
8 movi,  RXX, Rg3, 4'b1000
9 sw,    Rg3, Rg1, 4'b0000
10 sub,   Rg0, Rg1, 4'b0000
```

1->2 raw

1->3 raw

2->3 raw, waw

3->6 raw, war, waw

4->5 raw

5->6 raw

5->8 waw

6->7 raw, war, waw

6->8 war

7->9 raw

8->9 raw

7->10 raw, war, waw

1->10 raw

We can also say there's a dependency from 2 to 9, since they both use the same register, but not the same value. We don't need make this dependency explicit, since 9 depends on 7 and 7 depends on 6, which depends on 3.

```

1 movi, RXX, Rg0, 4'b0101
2 lw,    Rg0, Rg1, 4'b0000
3 add,   Rg0, Rg1, 4'b0000
4 movi,  RXX, Rg2, 4'b0000
5 lw,    Rg2, Rg3, 4'b0000
6 add,   Rg3, Rg1, 4'b0000
7 addi,  RXX, Rg1, 4'b0000
8 movi,  RXX, Rg3, 4'b1000
9 sw,    Rg3, Rg1, 4'b0000
10 sub,   Rg0, Rg1, 4'b0000

```

1->2 raw  
1->3 raw  
2->3 raw, waw  
3->6 raw, war, waw  
4->5 raw  
5->6 raw  
5->8 waw  
6->7 raw, war, waw  
6->8 war  
7->9 raw  
8->9 raw  
7->10 raw, war, waw  
1->10 raw

