Pipeline design

Mehran Rezaei

How Can We Improve the Performance?

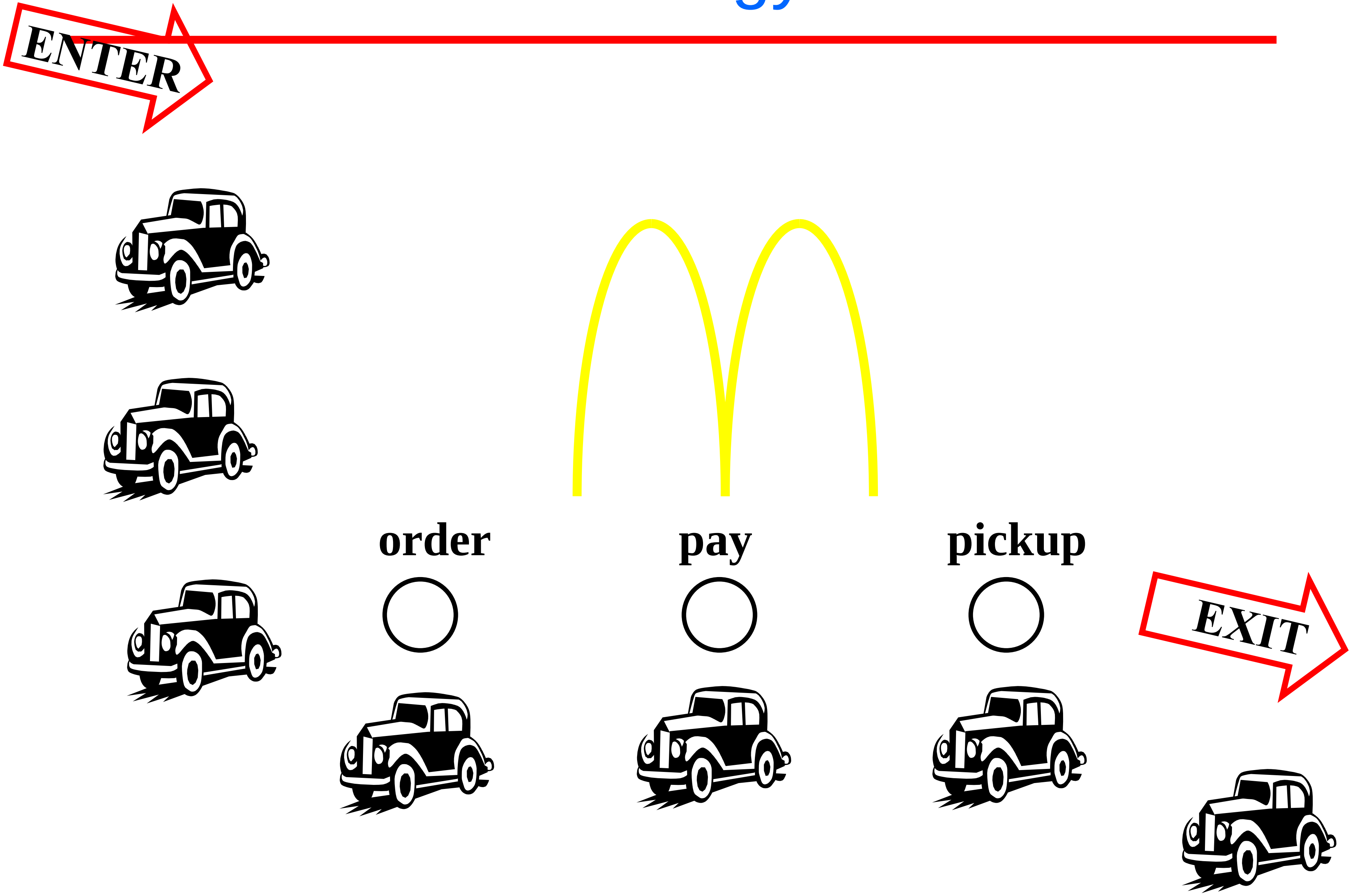
- Exec Time = IC * CPI * CCT

Optimization	IC	CPI	CCT
Source Level	*		
Compiler	*	*	
ISA	*	*	
Organization		*	*
Technology			*

With Pipelining
We want to get
5 times faster
Clock rate

- Single Cycle machine: CPI is one

Analogy



ENTER

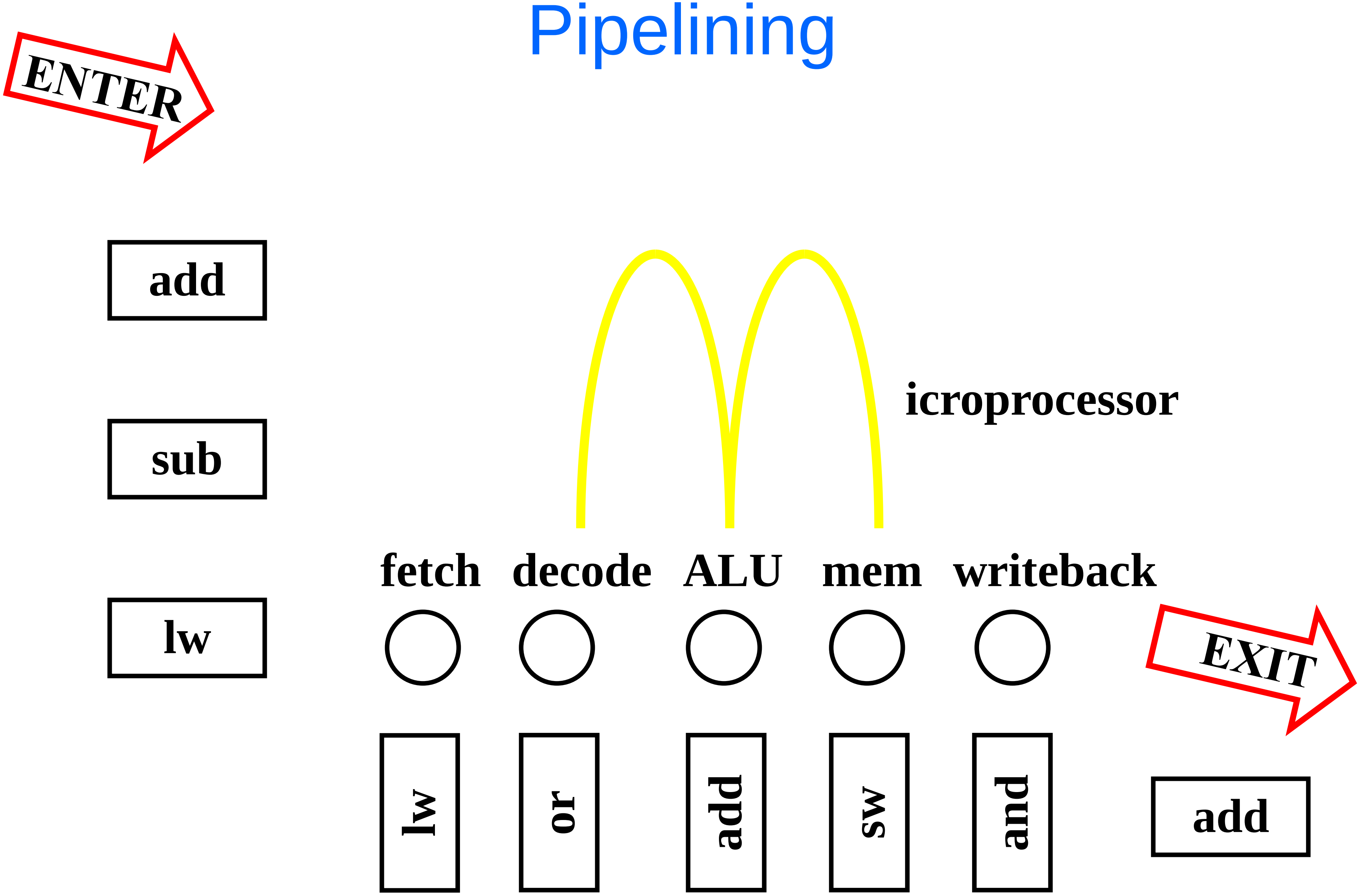
order

pay

pickup

EXIT

Pipelining



ENTER

add

sub

lw

icroprocessor

fetch decode ALU mem writeback

lw

or

add

sw

and

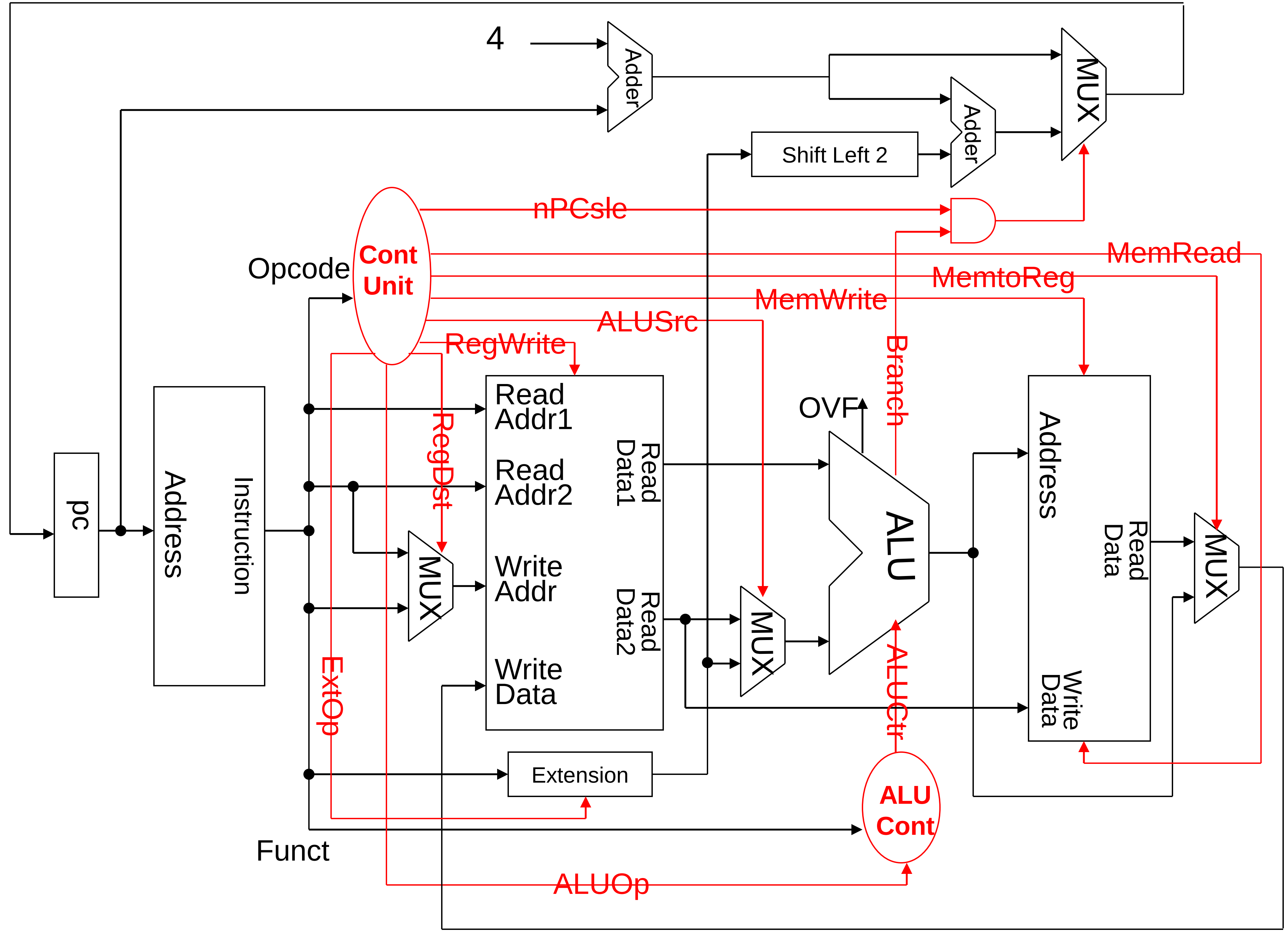
EXIT

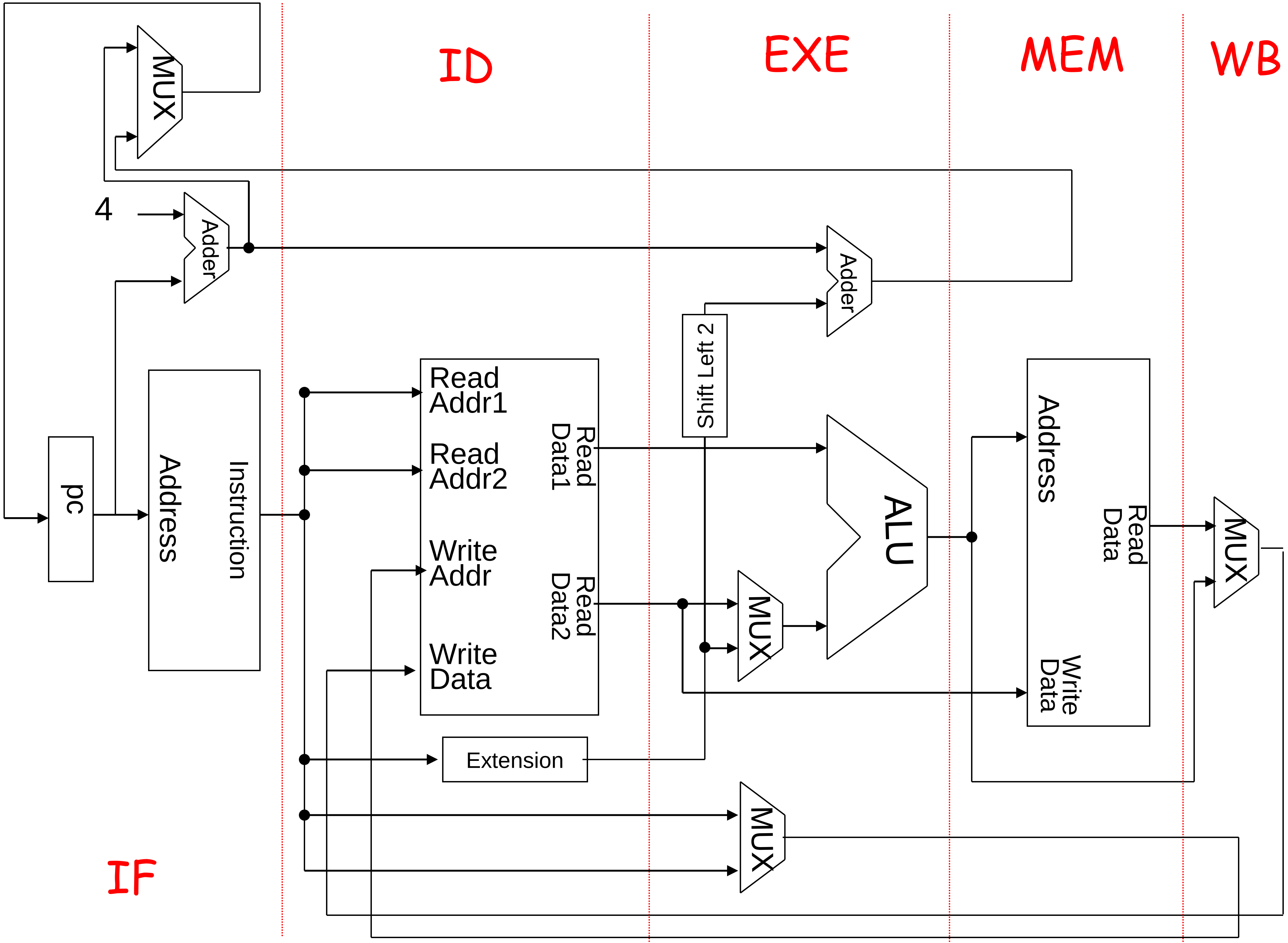
add

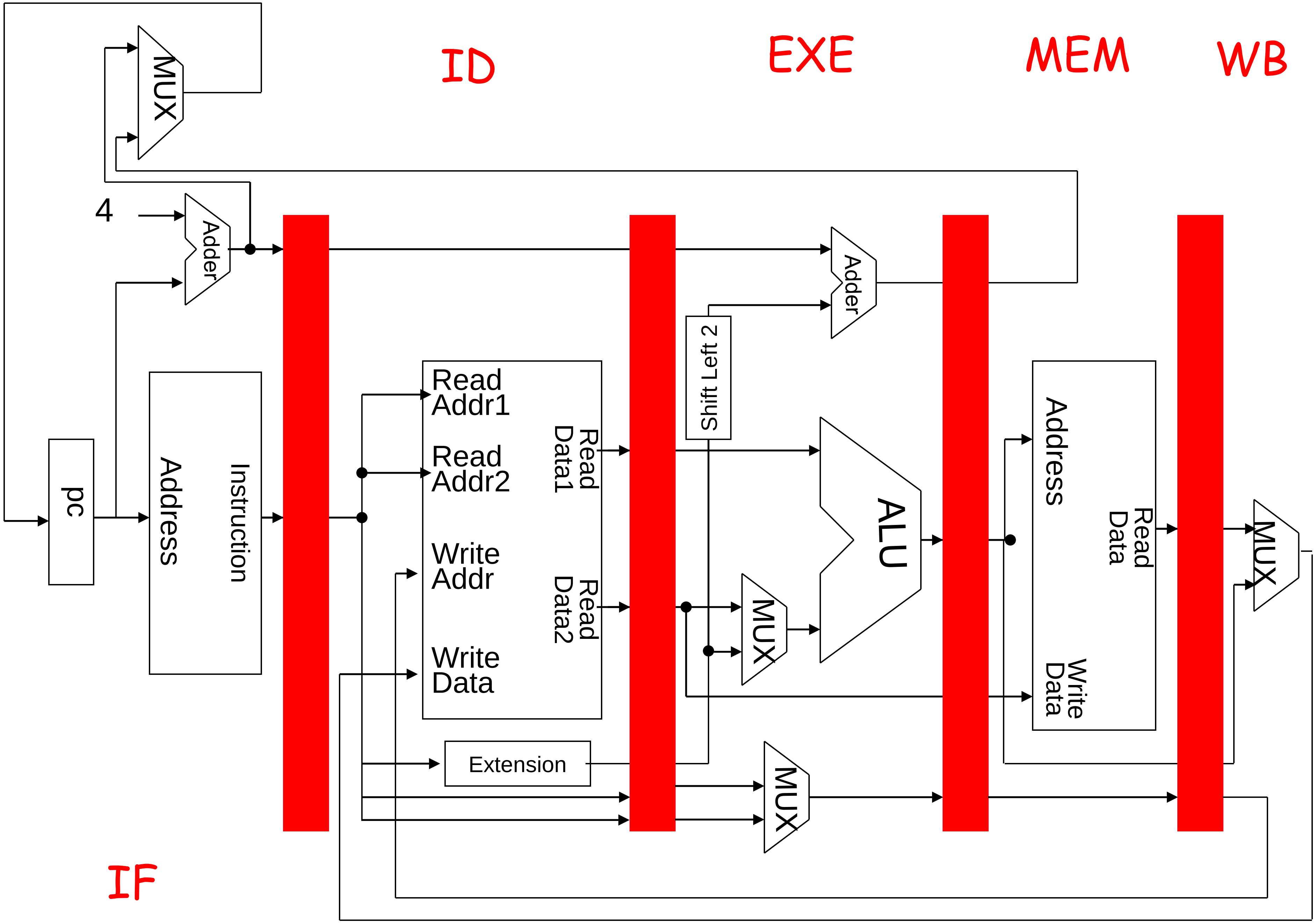
Pipeline design

- Break the execution of the instruction into cycles.

- Design a separate datapath stage for the execution performed during each cycle.
  - Build pipeline registers to communicate between the stages.


4
Adder
Adder
MUX
Shift Left 2
nPCsle
Cont Unit
Opcode
MemRead
MemtoReg
MemWrite
ALUSrc
RegWrite
Branch
OVF
pc
Address
Instruction
Read Addr1
Read Addr2
Write Addr
Write Data
Read Data1
Read Data2
RegDst
MUX
MUX
ALU
Address
Read Data
Write Data
MUX
ExtOp
ALUCtr
Extension
ALU Cont
Funct
ALUOp


ID
EXE
MEM
WB
IF
MUX
4
Adder
Adder
pc
Address
Instruction
Read Addr1
Read Addr2
Write Addr
Write Data
Read Data1
Read Data2
Shift Left 2
ALU
MUX
Extension
MUX
Address
Read Data
Write Data
MUX

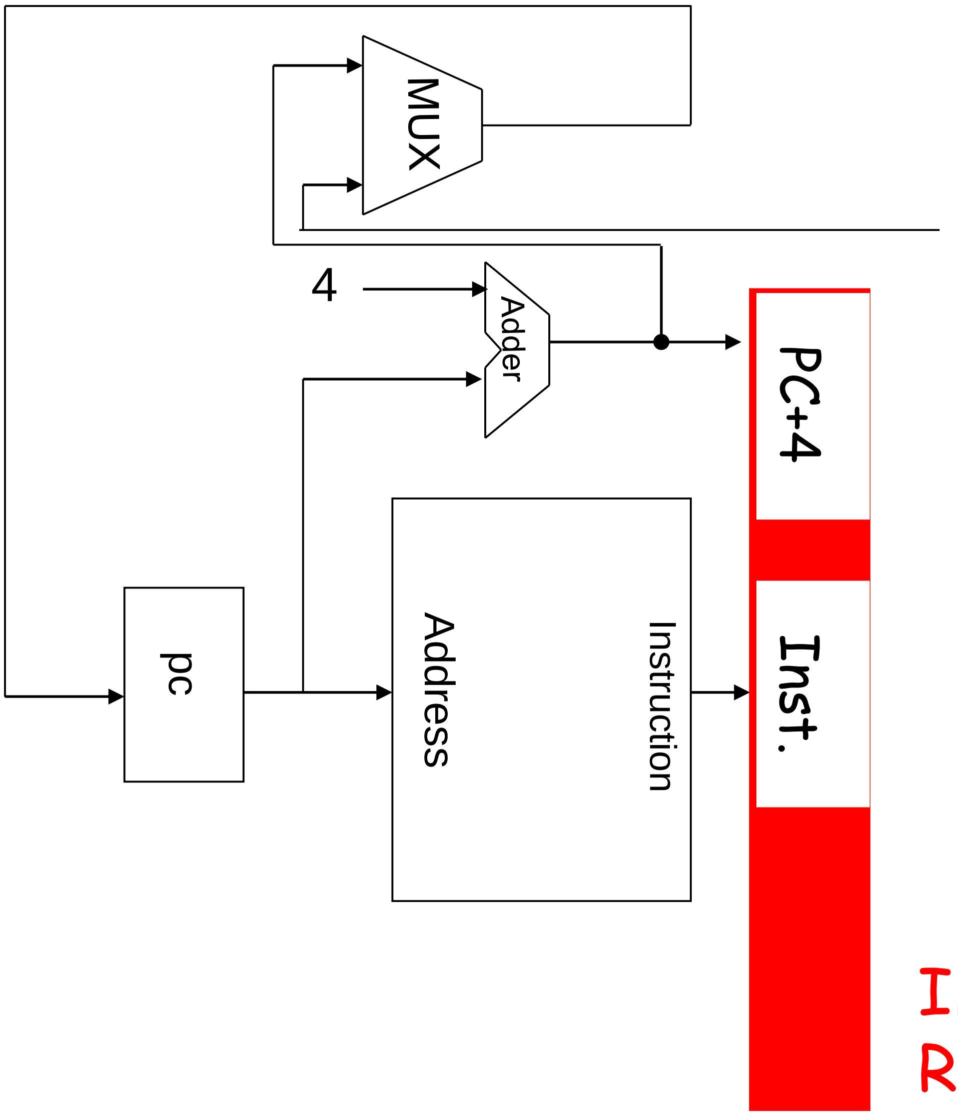

ID
EXE
MEM
WB
IF
MUX
4
Adder
pc
Address
Instruction
Read Addr1
Read Addr2
Write Addr
Write Data
Read Data1
Read Data2
Extension
Shift Left 2
Adder
ALU
MUX
MUX
Address
Read Data
Write Data
MUX

Instruction Fetch

- Design a datapath that can fetch an instruction from memory every cycle.
  - Use PC to index memory to read instruction
  - Increment the PC (assume no branches for now)

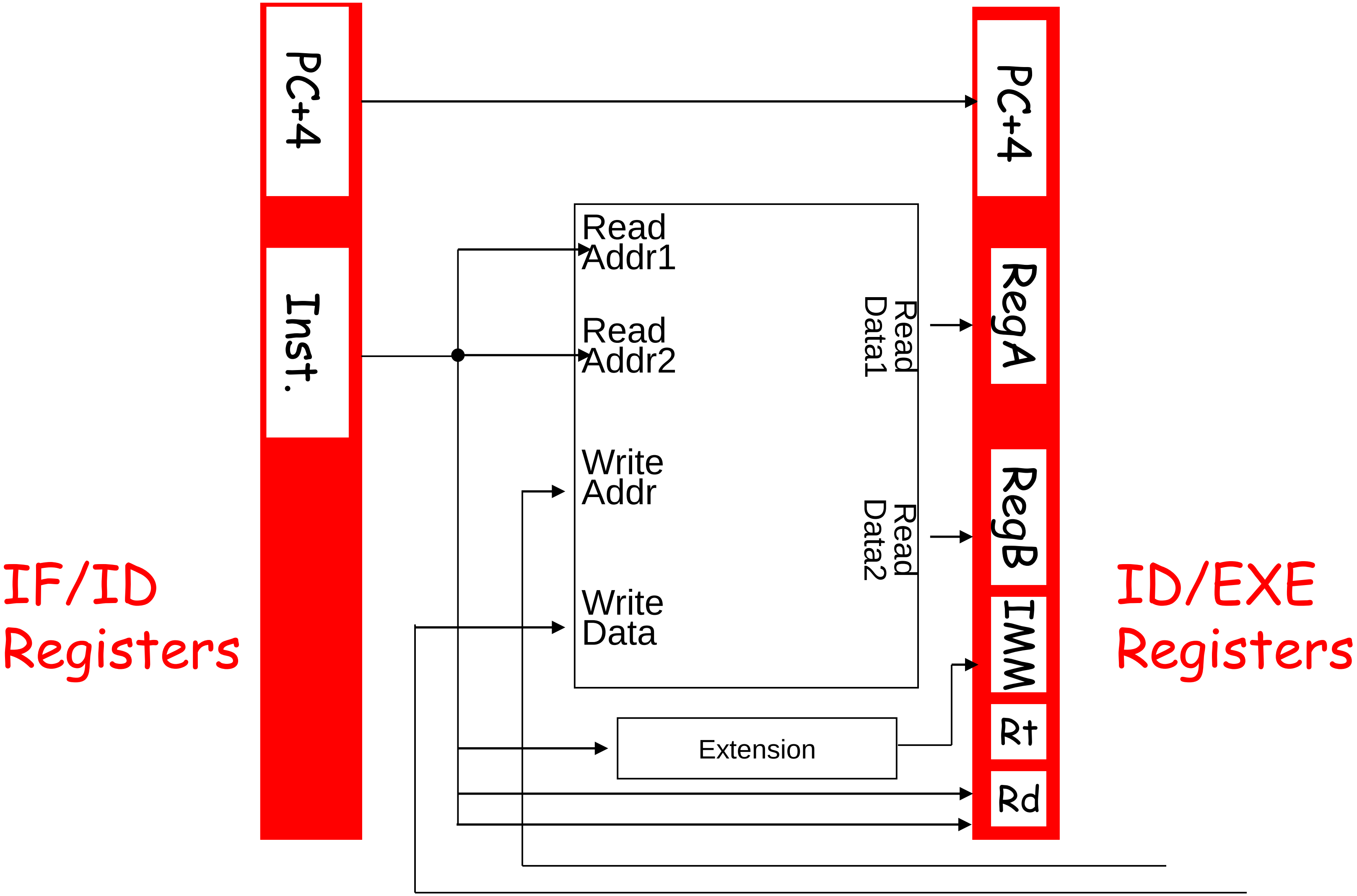
- Write everything needed to complete execution to the pipeline register (IF/ID)
  - The next stage will read this pipeline register.
  - Note that pipeline register must be edge triggered

IF

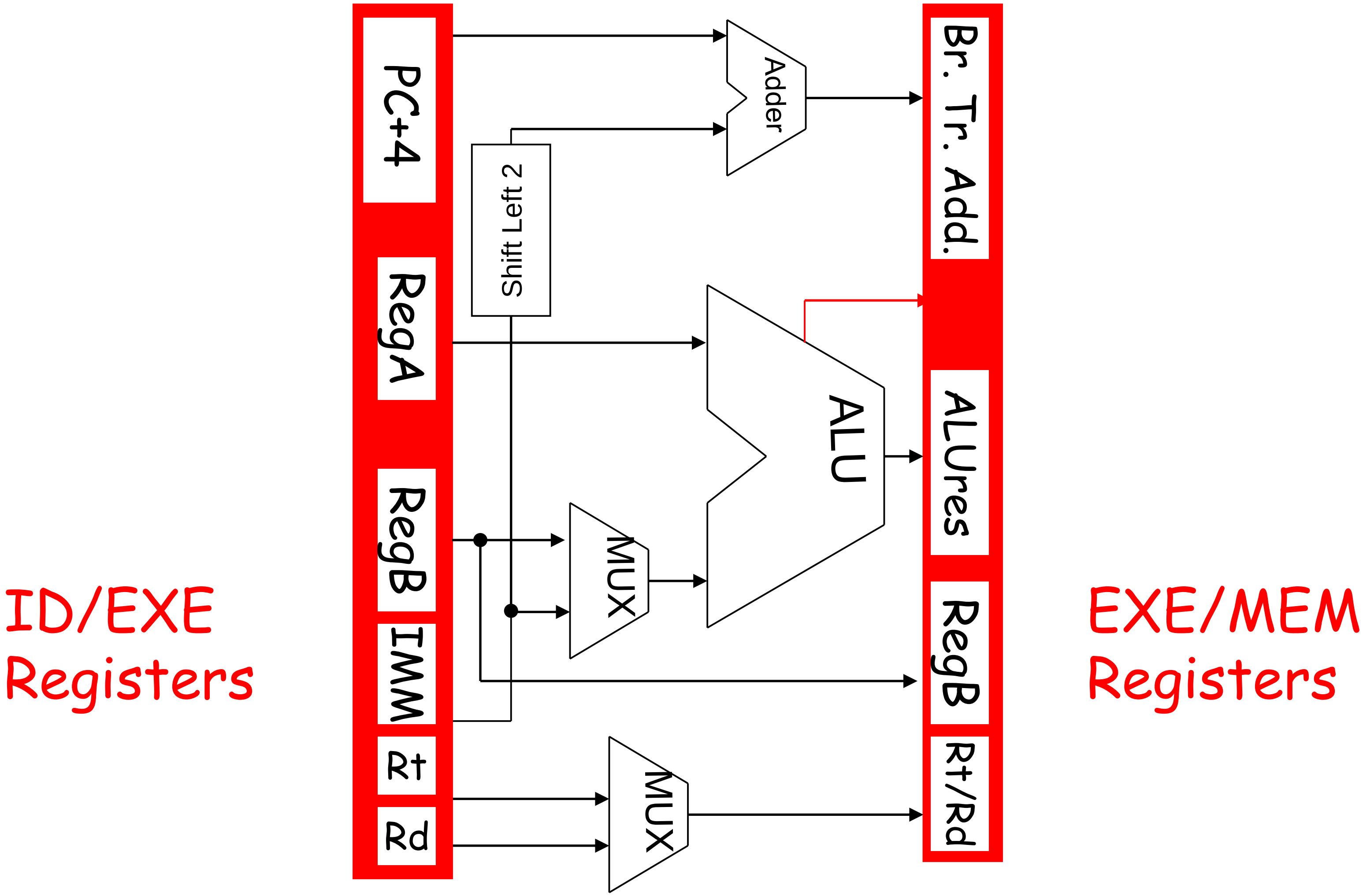


IF/ID
Registers

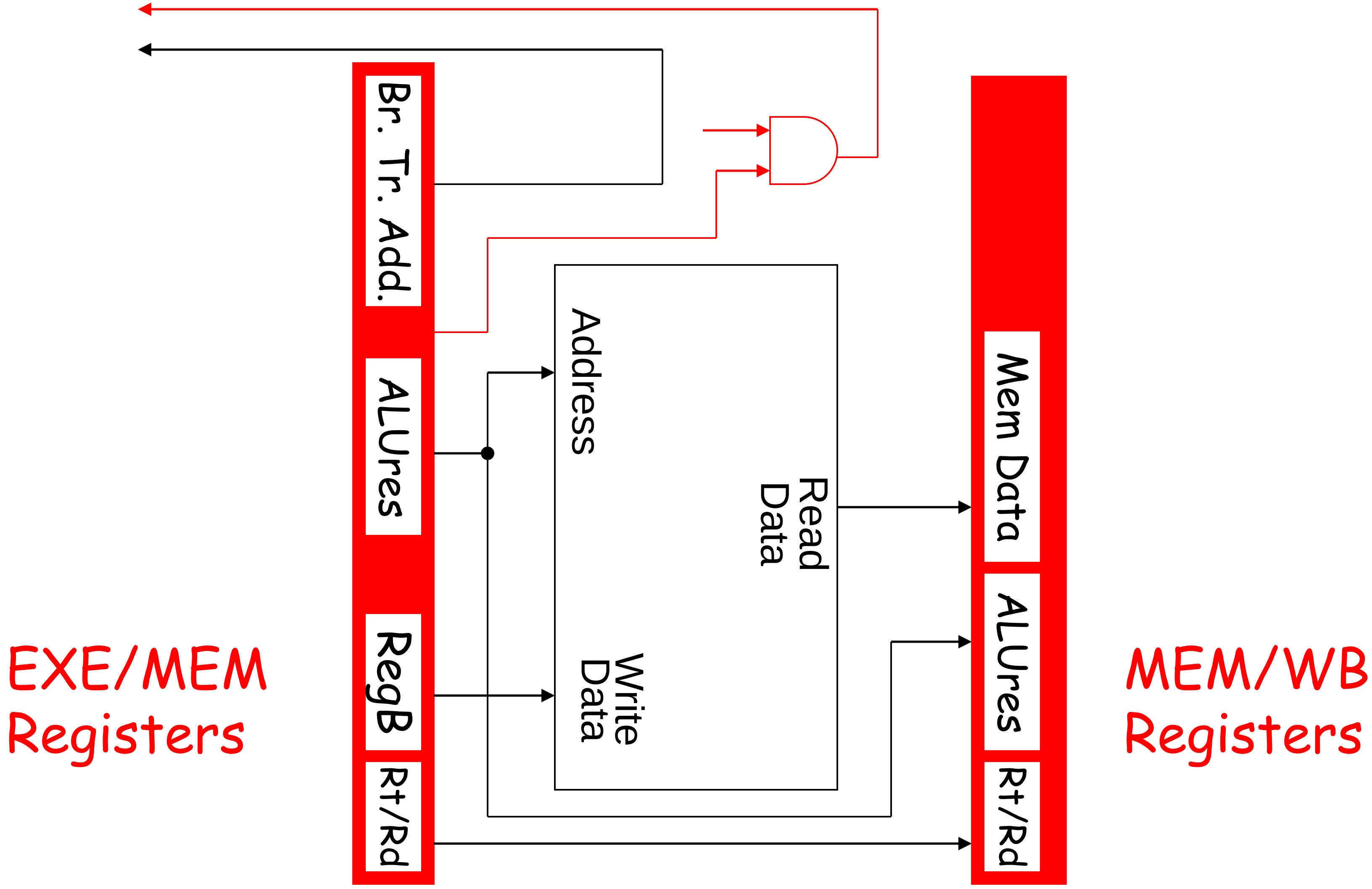
ID


PC+4
Inst.
Read Addr1
Read Addr2
Write Addr
Write Data
Read Data1
Read Data2
Extension
PC+4
RegA
RegB
IMM
Rt
Rd
IF/ID Registers
ID/EXE Registers

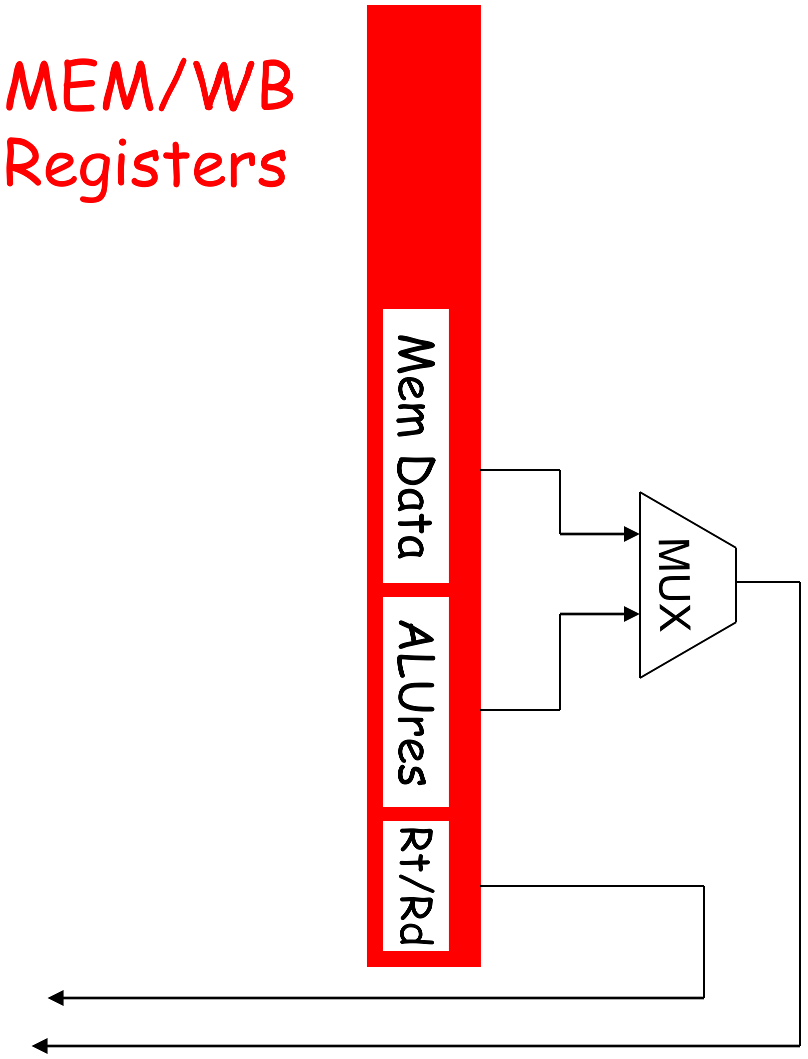
EXE

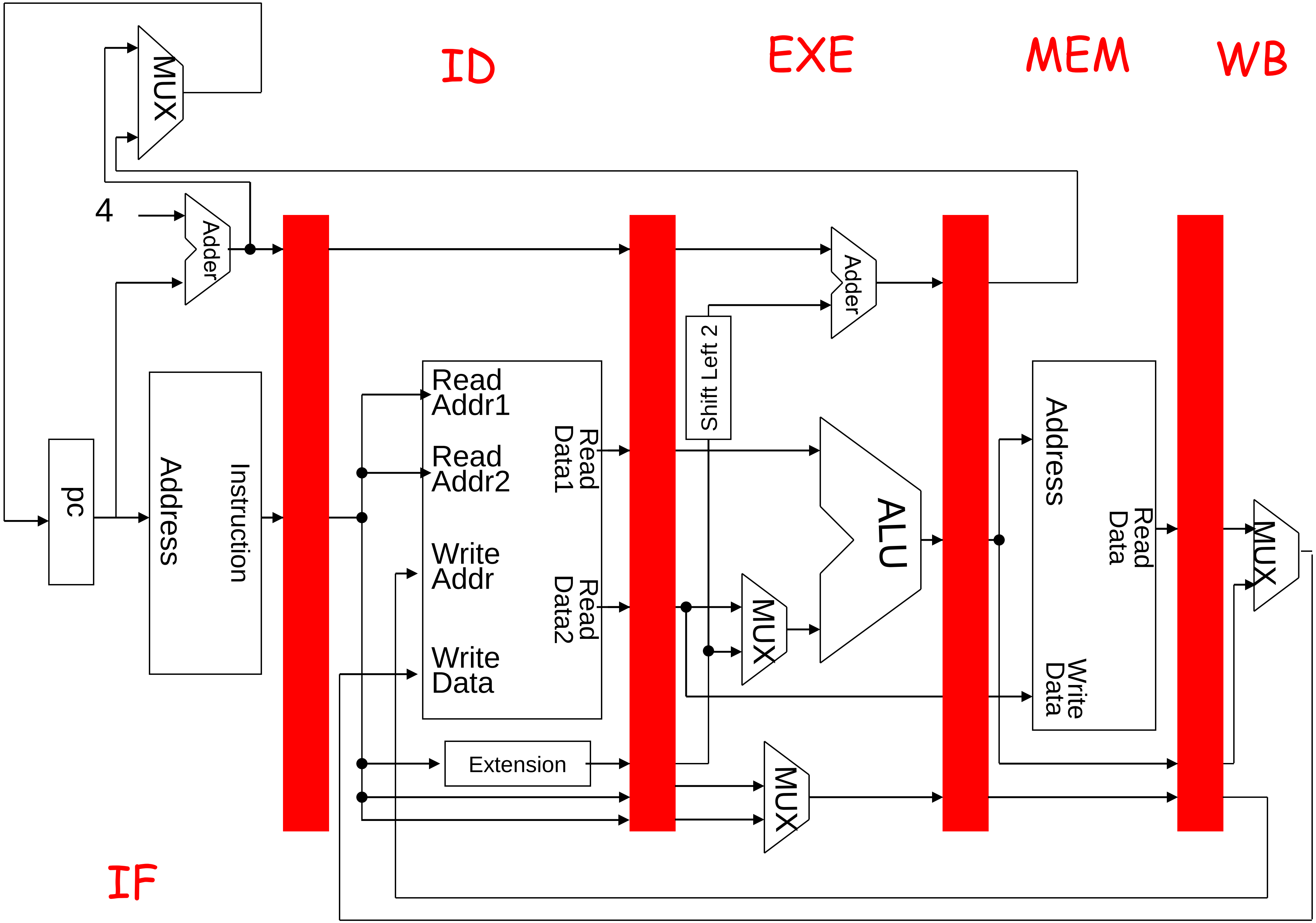

ID/EXE Registers
PC+4
RegA
RegB
IMM
Rt
Rd
Shift Left 2
Adder
ALU
MUX
MUX
Br. Tr. Add.
ALUres
RegB
Rt/Rd
EXE/MEM Registers

MEM


Br. Tr. Add.
ALUres
RegB
Rt/Rd
Address
Read Data
Write Data
Mem Data
ALUres
Rt/Rd
EXE/MEM Registers
MEM/WB Registers

WB

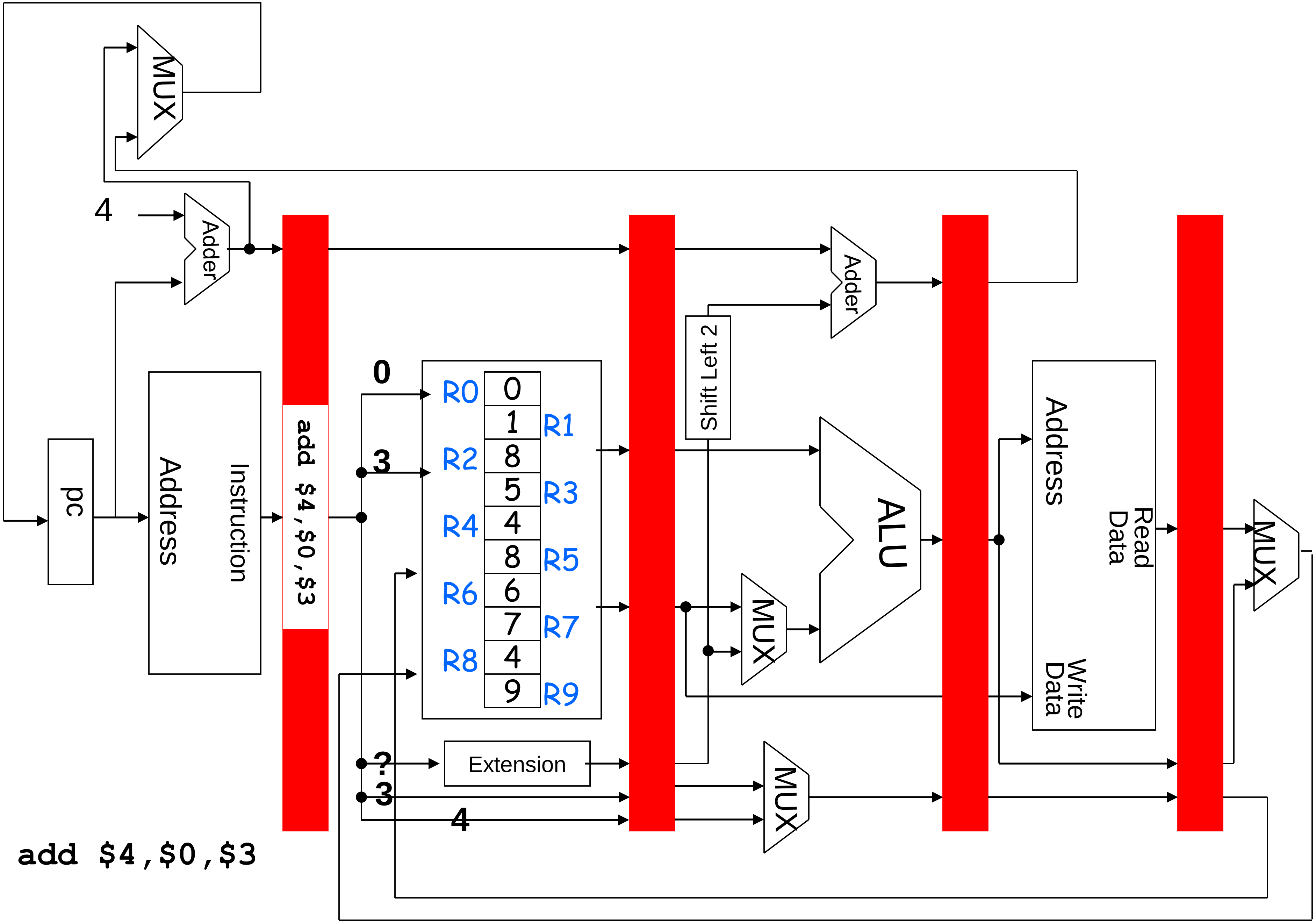



ID
EXE
MEM
WB
MUX
4
Adder
pc
Address
Instruction
Read Addr1
Read Addr2
Write Addr
Write Data
Read Data1
Read Data2
Extension
Shift Left 2
Adder
ALU
MUX
MUX
Address
Read Data
Write Data
MUX
IF

Example

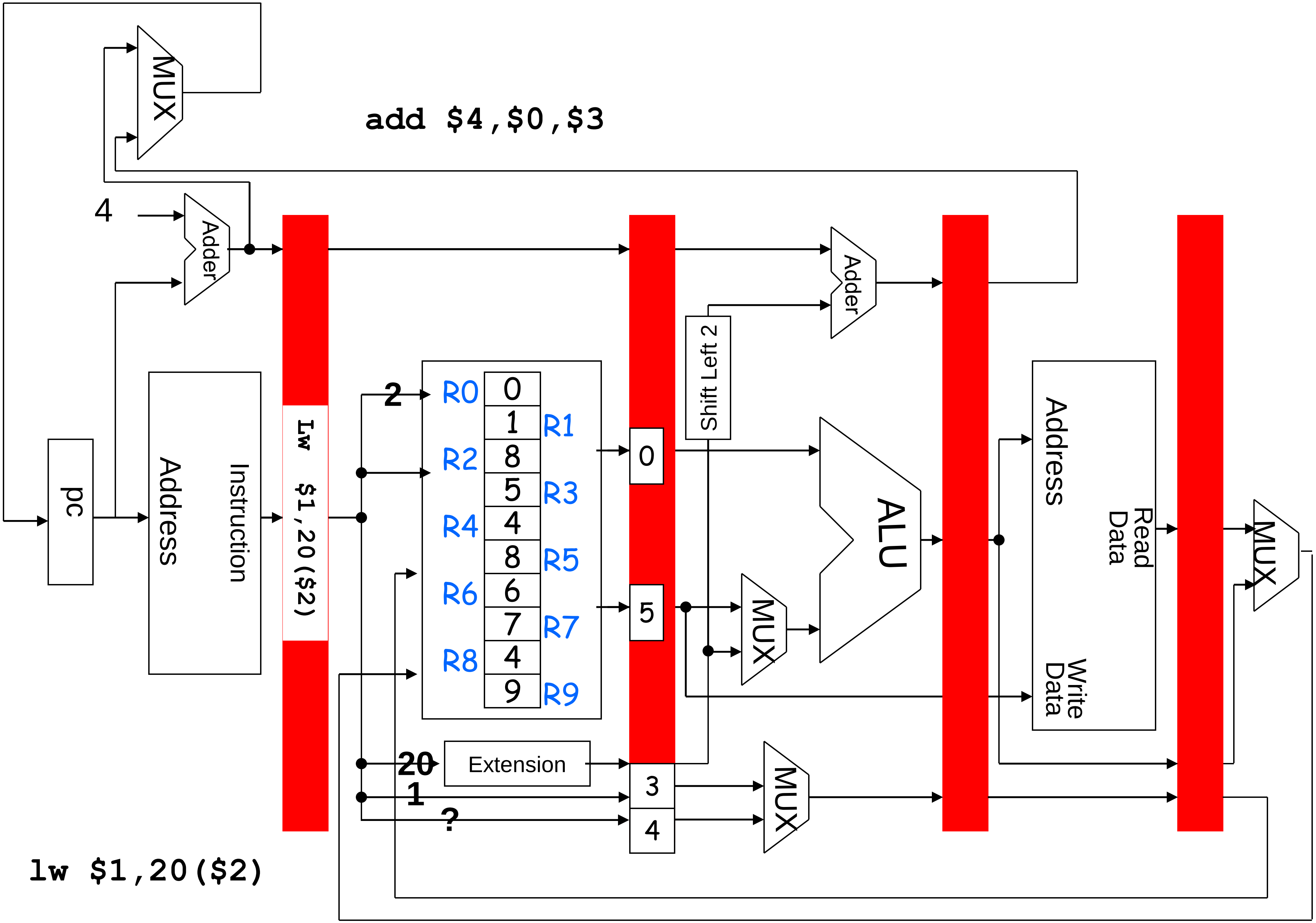
- Run the following code on our pipeline machine

```
add     $4,$0,$3
lw      $1,20($2)
sub     $5,$6,$6
sw      $7,8($8)
add     $9,$4,$3
```


MUX
4
Adder
pc
Address
Instruction
add $4,$0,$3
0
3
R0 0
R1 1
R2 8
R3 5
R4 4
R5 8
R6 6
R7 7
R8 4
R9 9
?
3
4
Extension
Shift Left 2
Adder
ALU
MUX
MUX
Address
Read Data
Write Data
MUX

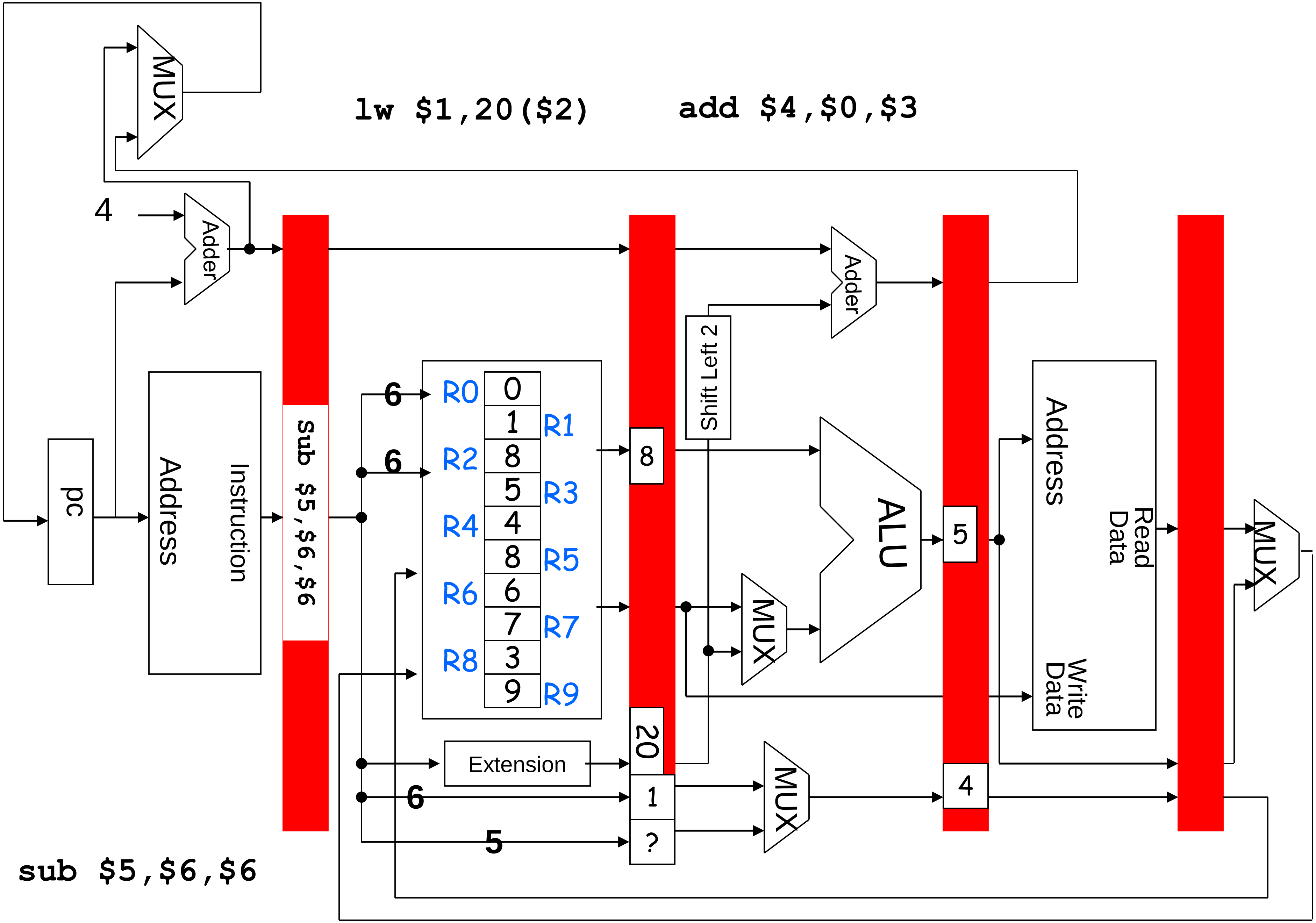
`add $4,$0,$3`

add $4,$0,$3



R0	0	
	1	R1
R2	8	
	5	R3
R4	4	
	8	R5
R6	6	
	7	R7
R8	4	
	9	R9

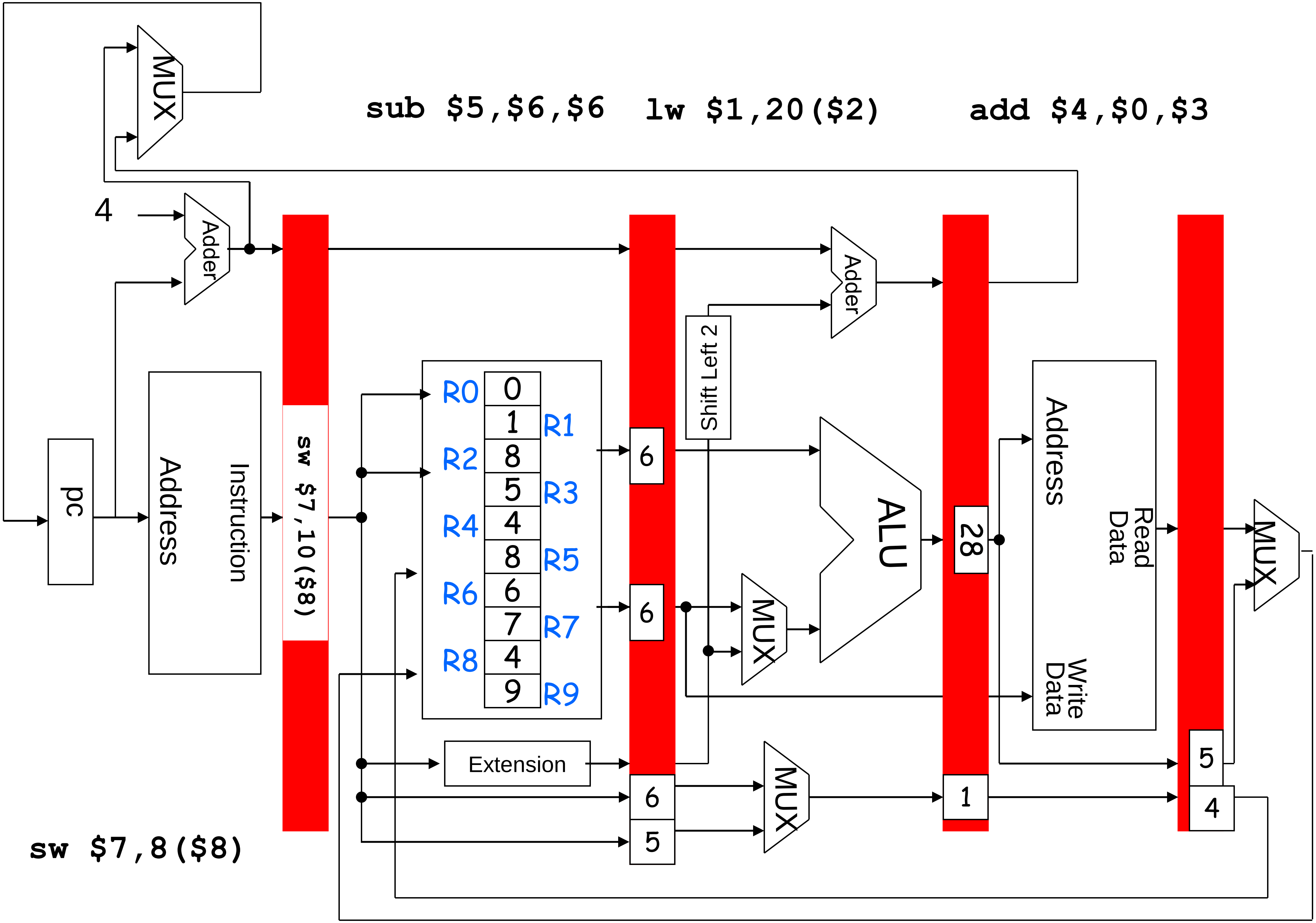
lw $1,20($2)



lw $1,20($2)   add $4,$0,$3

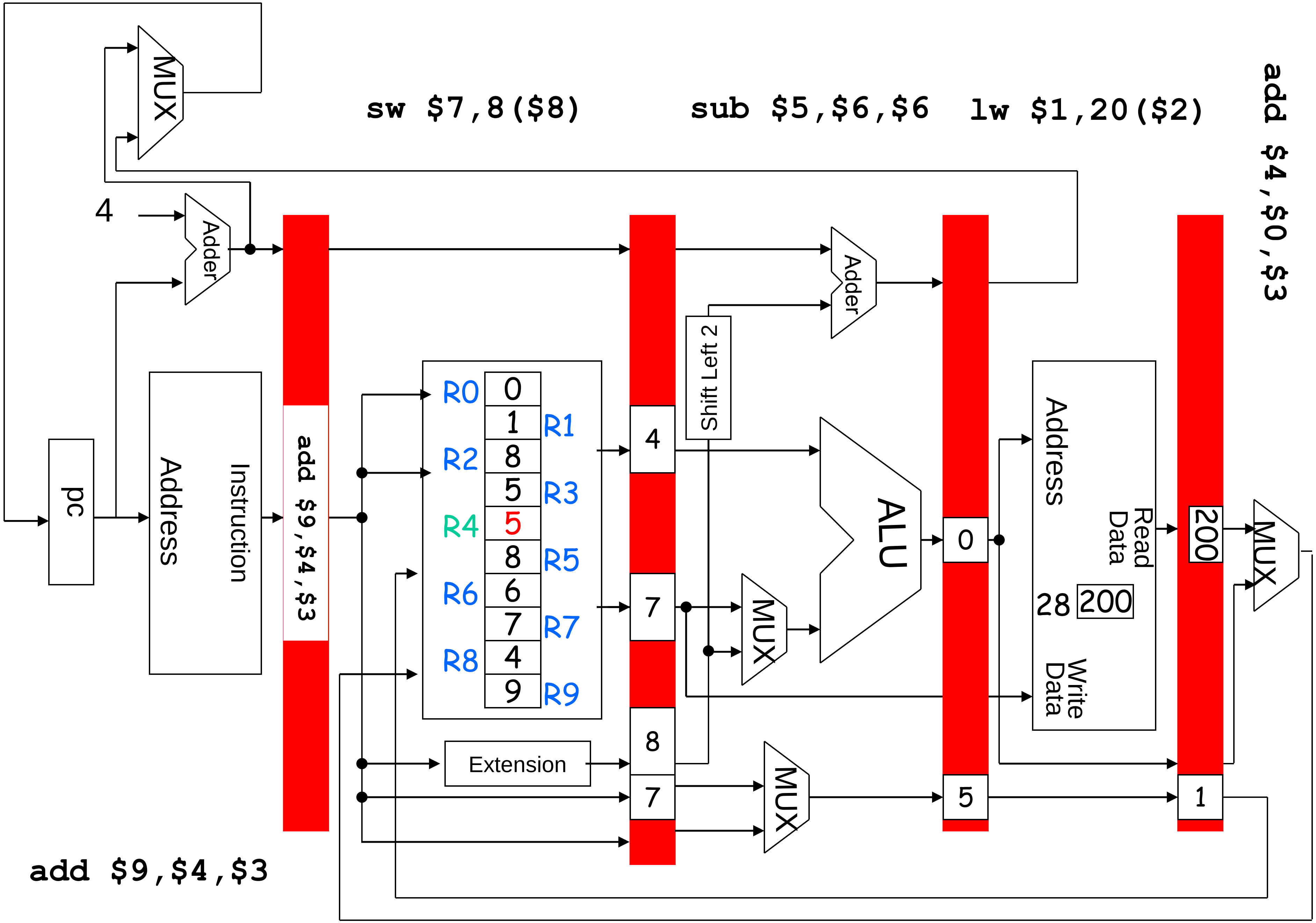
sub $5,$6,$6

sub $5,$6,$6 lw $1,20($2) add $4,$0,$3



sw $7,8($8)

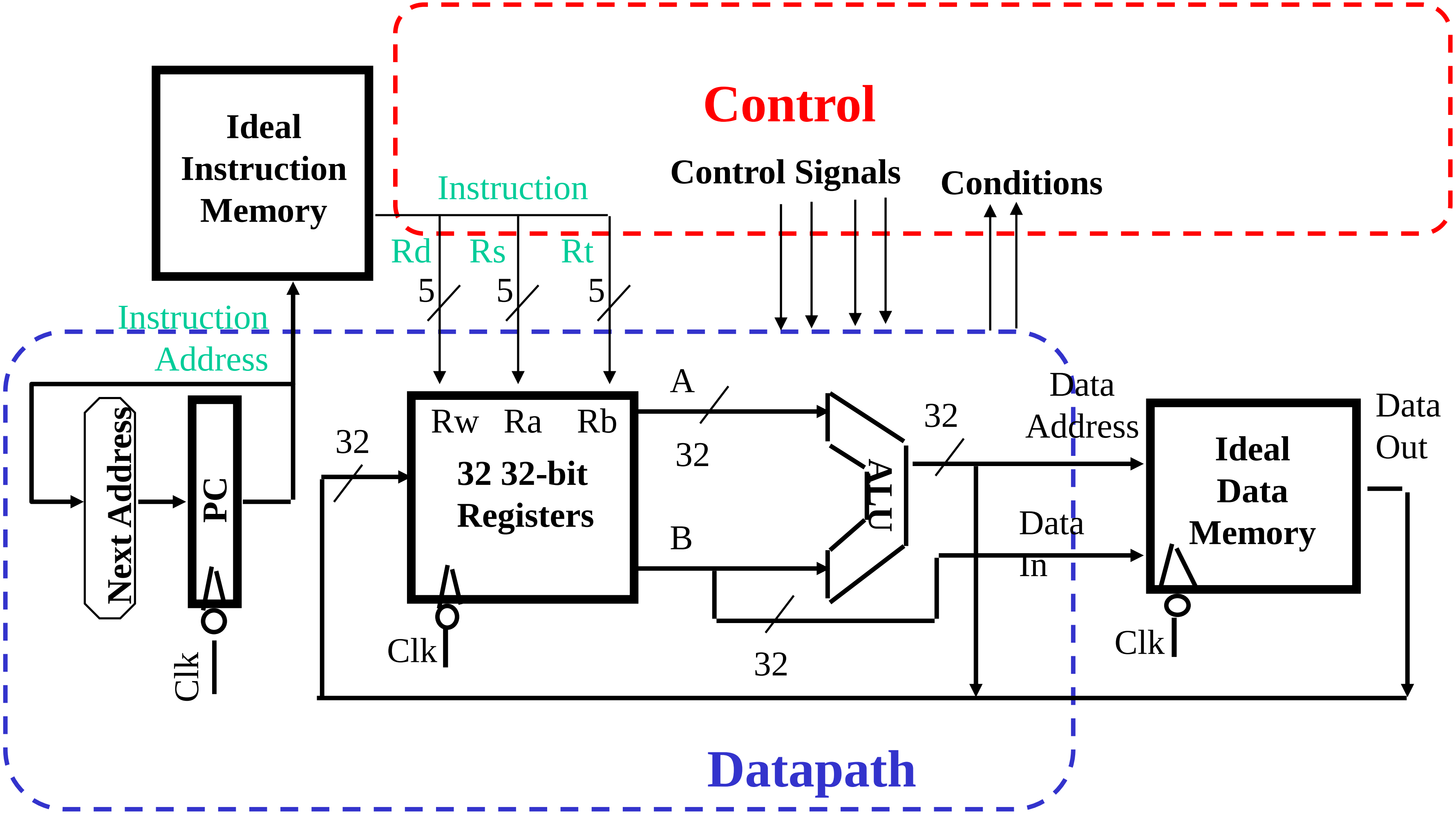
sw $7,8($8)   sub $5,$6,$6   lw $1,20($2)   add $4,$0,$3



R0	0	
	1	R1
R2	8	
	5	R3
R4	5	
	8	R5
R6	6	
	7	R7
R8	4	
	9	R9

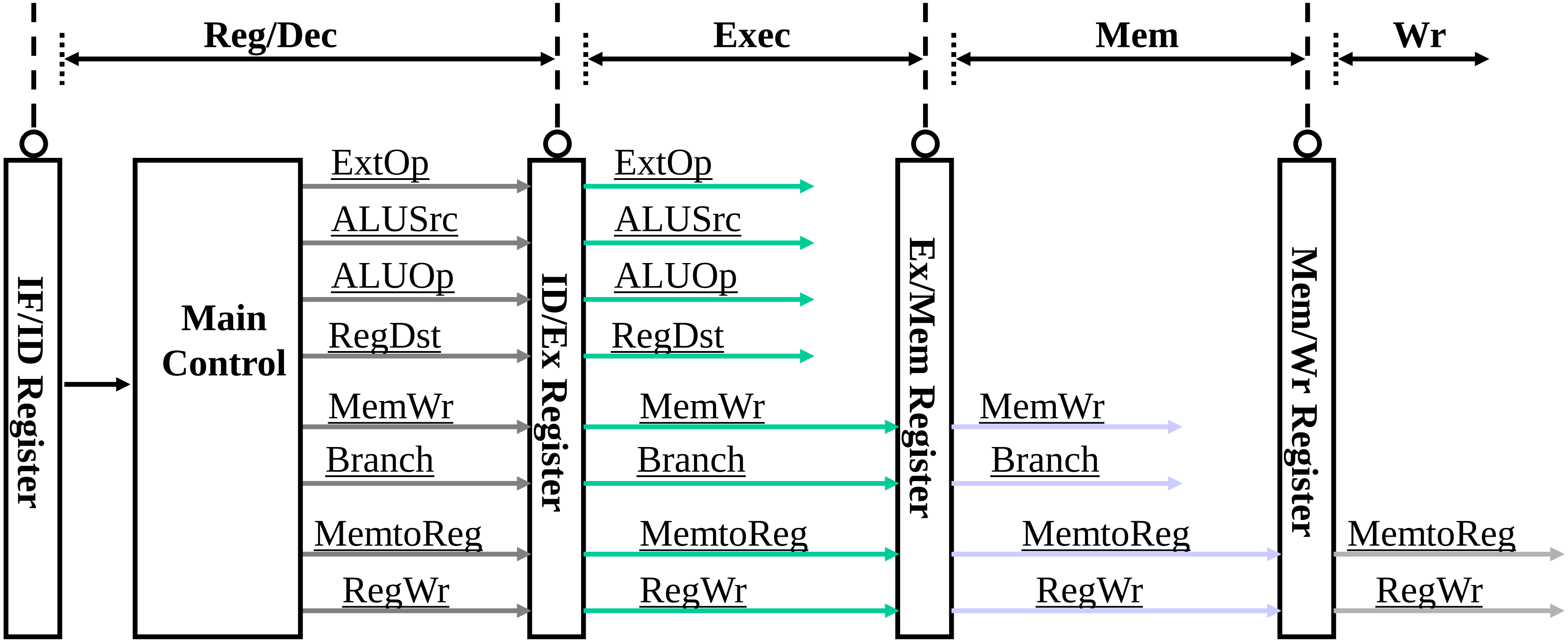
add $9,$4,$3

Recall: Single cycle control!

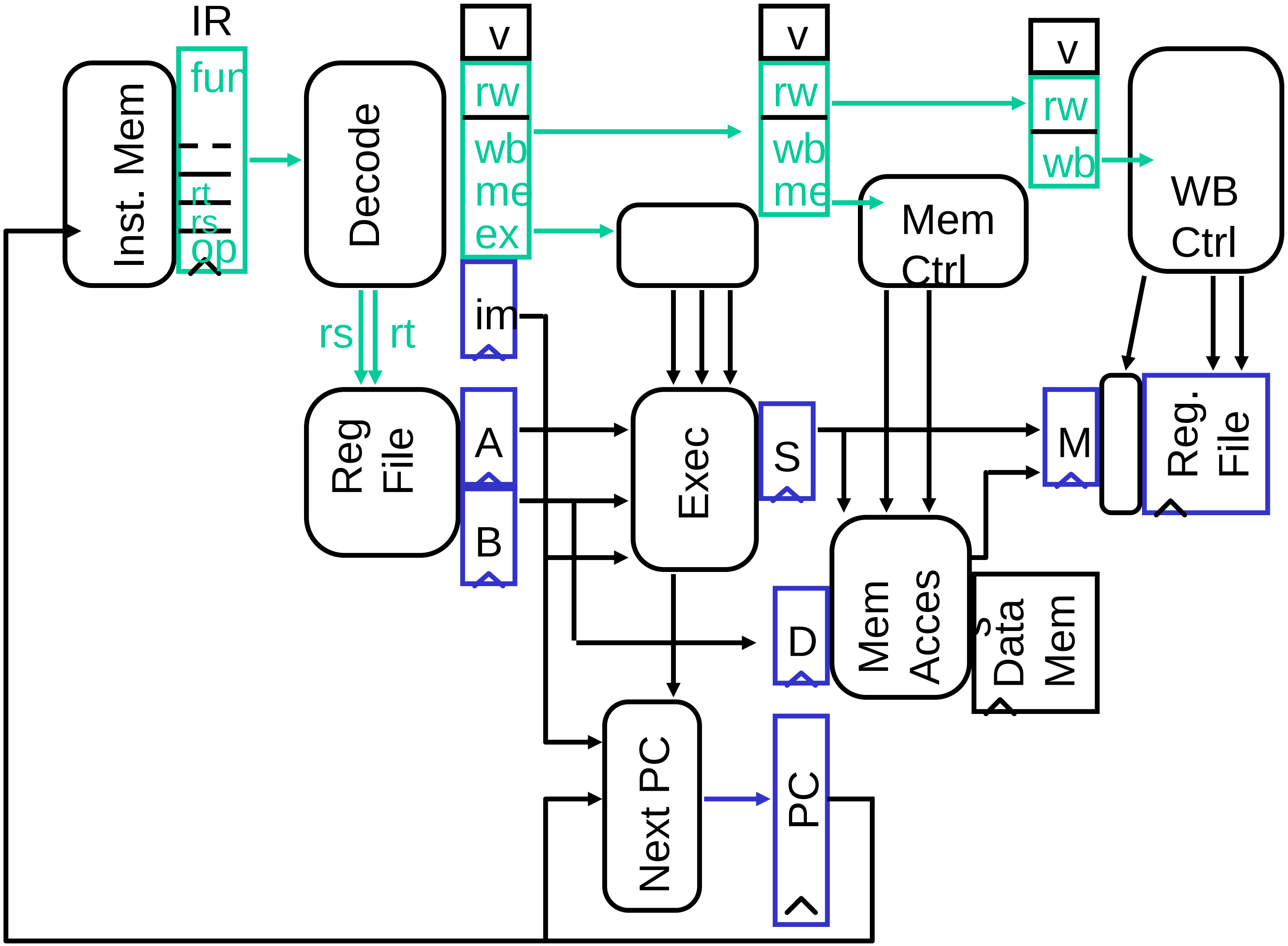


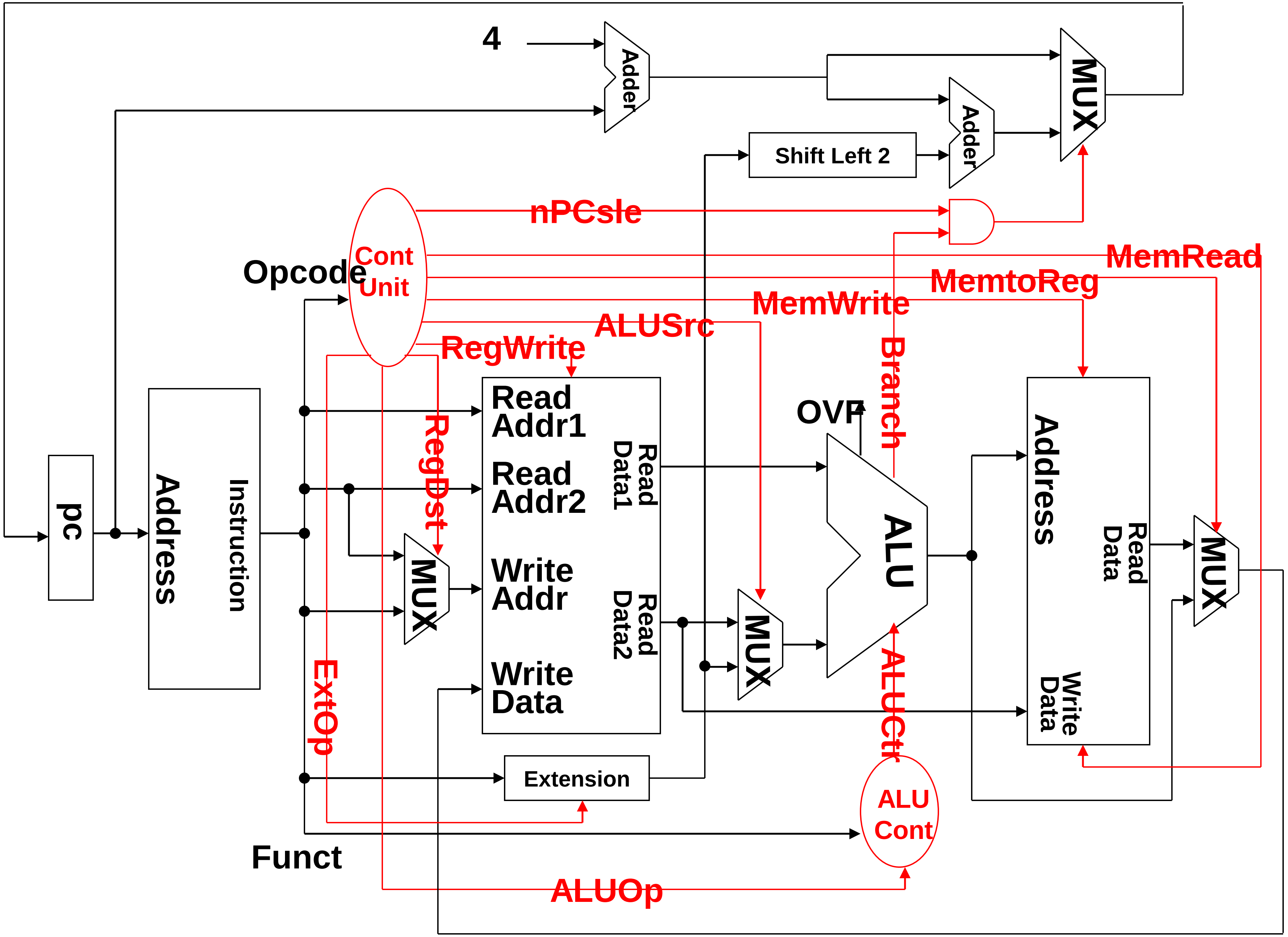
Data Stationary Control

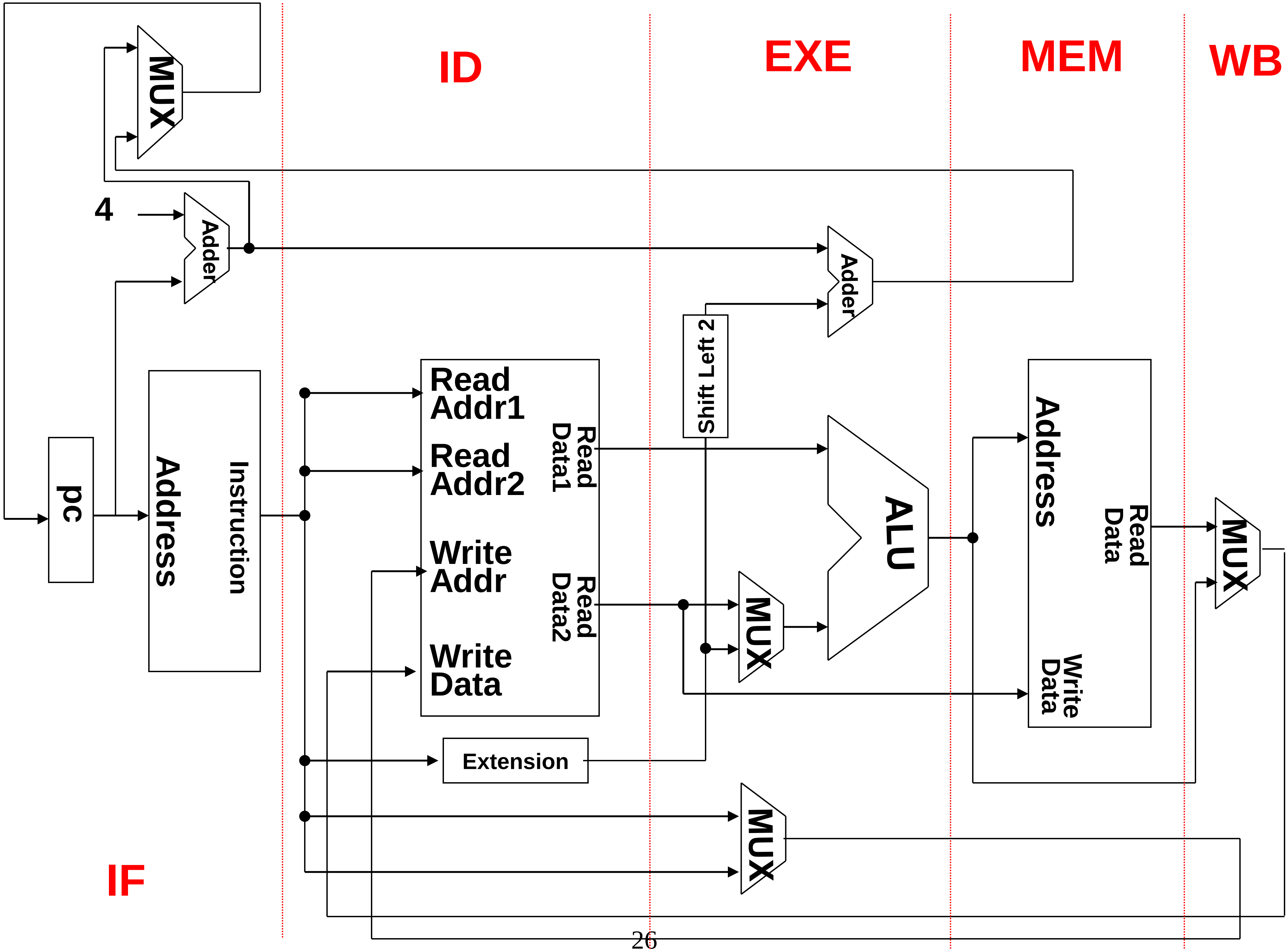
- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later

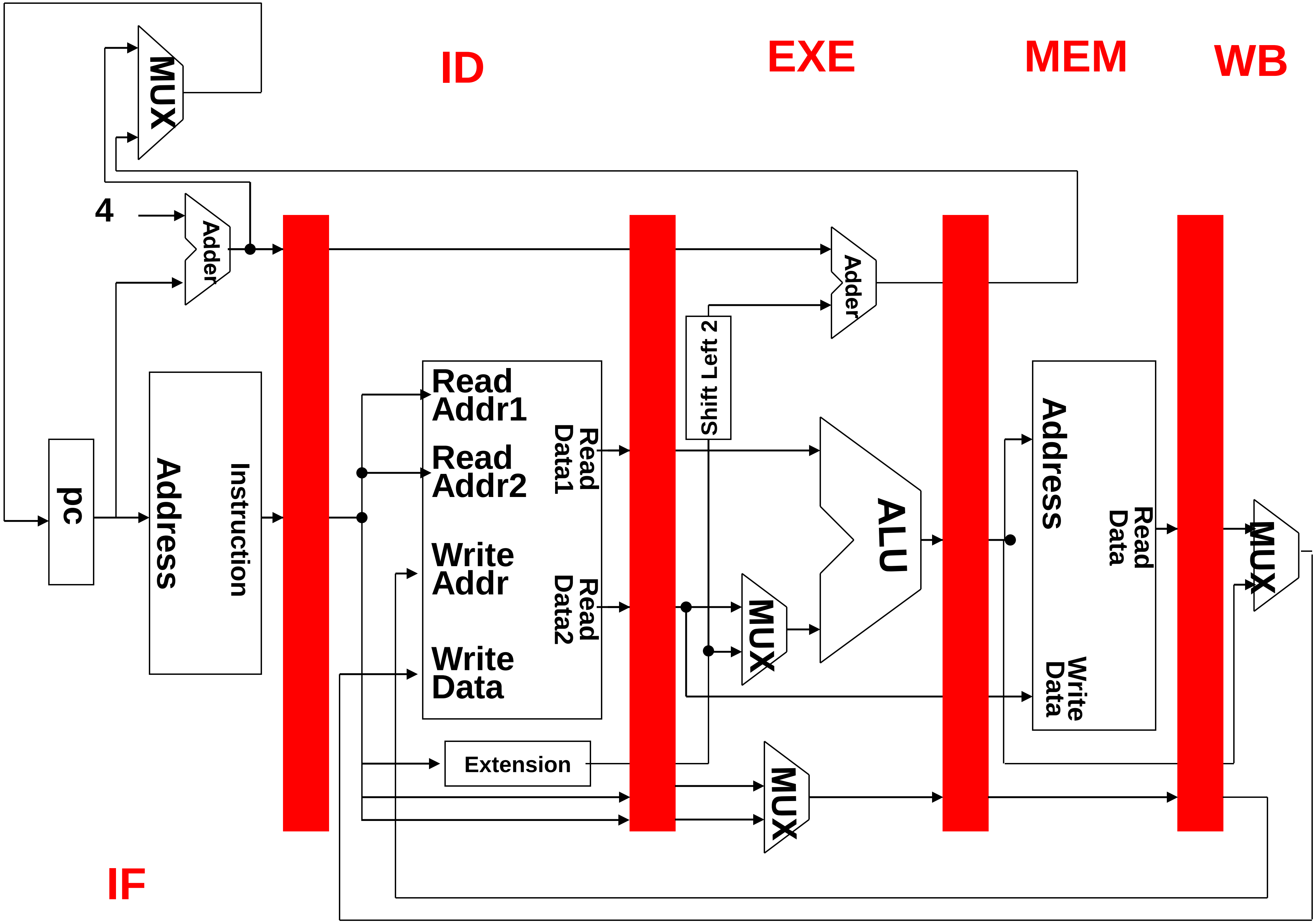


Datapath + Data Stationary Control




4
Adder
MUX
Adder
Shift Left 2
nPCsle
Cont
Unit
Opcode
MemRead
MemtoReg
MemWrite
ALUSrc
RegWrite
Branch
OVF
pc
Address
Instruction
RegDst
Read
Addr1
Read
Addr2
Write
Addr
Write
Data
Read
Data1
Read
Data2
MUX
MUX
ALU
Address
Read
Data
Write
Data
MUX
ExtOp
Extension
ALUCtr
ALU
Cont
Funct
ALUOp


ID
EXE
MEM
WB
MUX
4
Adder
Adder
Shift Left 2
pc
Address
Instruction
Read Addr1
Read Addr2
Write Addr
Write Data
Read Data1
Read Data2
Extension
MUX
ALU
Address
Read Data
Write Data
MUX
MUX
IF


ID
EXE
MEM
WB
MUX
4
Adder
Adder
Shift Left 2
Read Addr1
Read Addr2
Write Addr
Write Data
Read Data1
Read Data2
ALU
MUX
Address
Read Data
Write Data
MUX
pc
Address
Instruction
Extension
MUX
IF

Pipeline timing diagram

```
add $4,$0,$3
lw  $1,20($2)
sub $5,$6,$6
sw  $7,8($8)
add $9,$4,$3
```

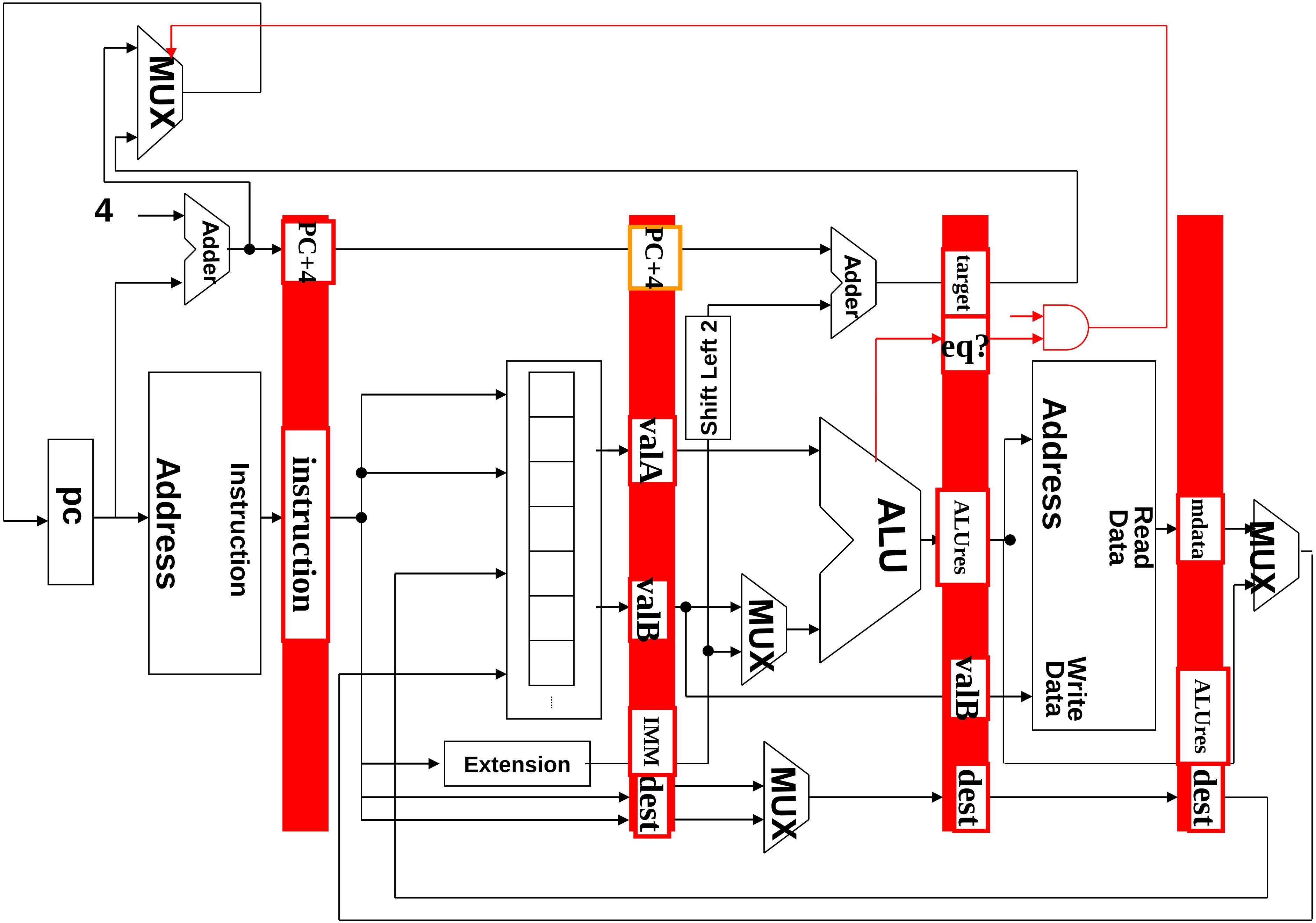
IF	ID	EXE	MEM	WB				
	IF	ID	EXE	MEM	WB			
		IF	ID	EXE	MEM	WB		
			IF	ID	EXE	MEM	WB	
				IF	ID	EXE	MEM	WB

Data Hazards

- What are they?

- How do you detect them?

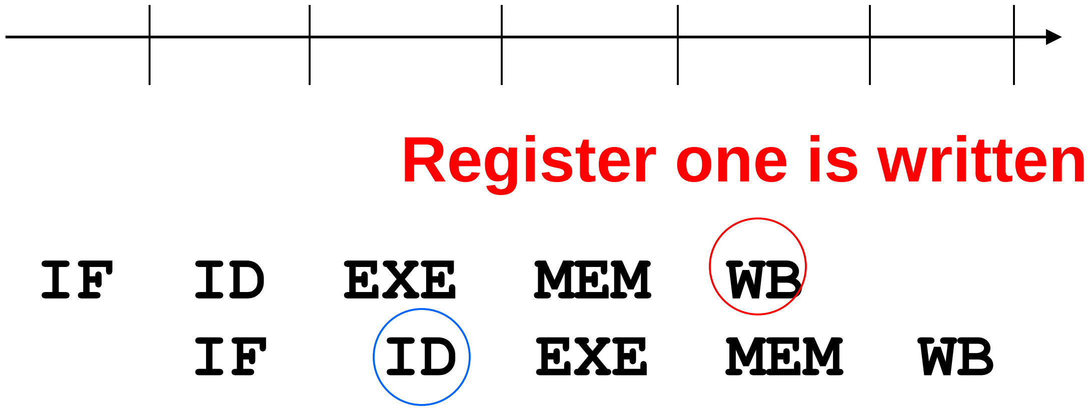
- How do you deal with them?


MUX
4
Adder
PC+4
pc
Address
Instruction
instruction
PC+4
valA
valB
IMM
dest
Extension
Shift Left 2
Adder
MUX
ALU
MUX
target
eq?
ALUres
valB
dest
Address
Read Data
Write Data
mdata
ALUres
dest
MUX

Pipeline cycles for add

- IF - Fetch: read instruction from memory
- ID - Decode: **read source operands from reg**
- EXE - Execute: calculate sum
- MEM - Memory: pass results to next stage
- WB - Writeback: **write sum (ALUres) into register file**

Data Hazard



Register one is written

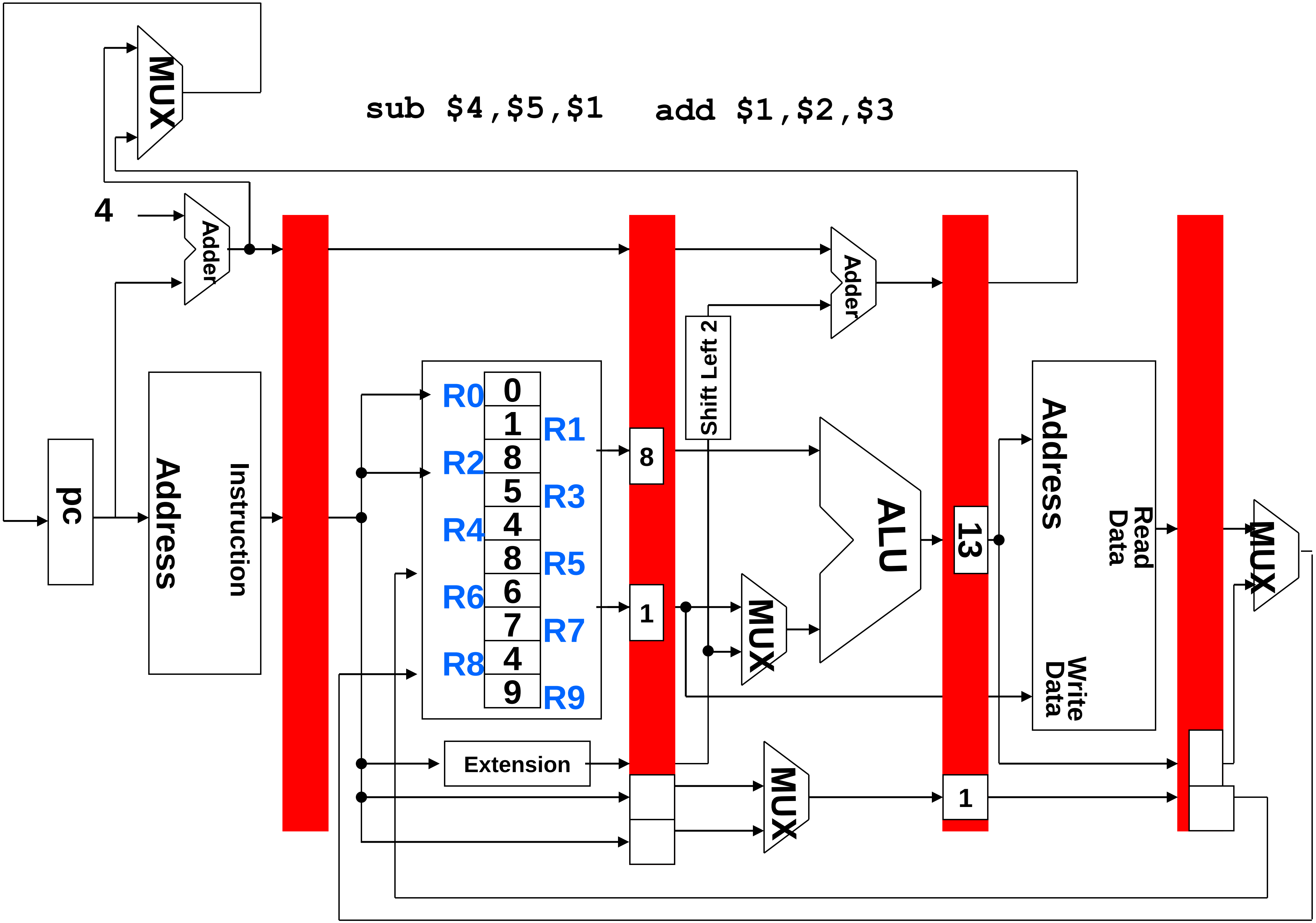
```
add   $1,$2,$3   IF   ID   EXE   MEM   WB
sub   $4,$5,$1        IF   ID    EXE   MEM   WB
```

Register one is read

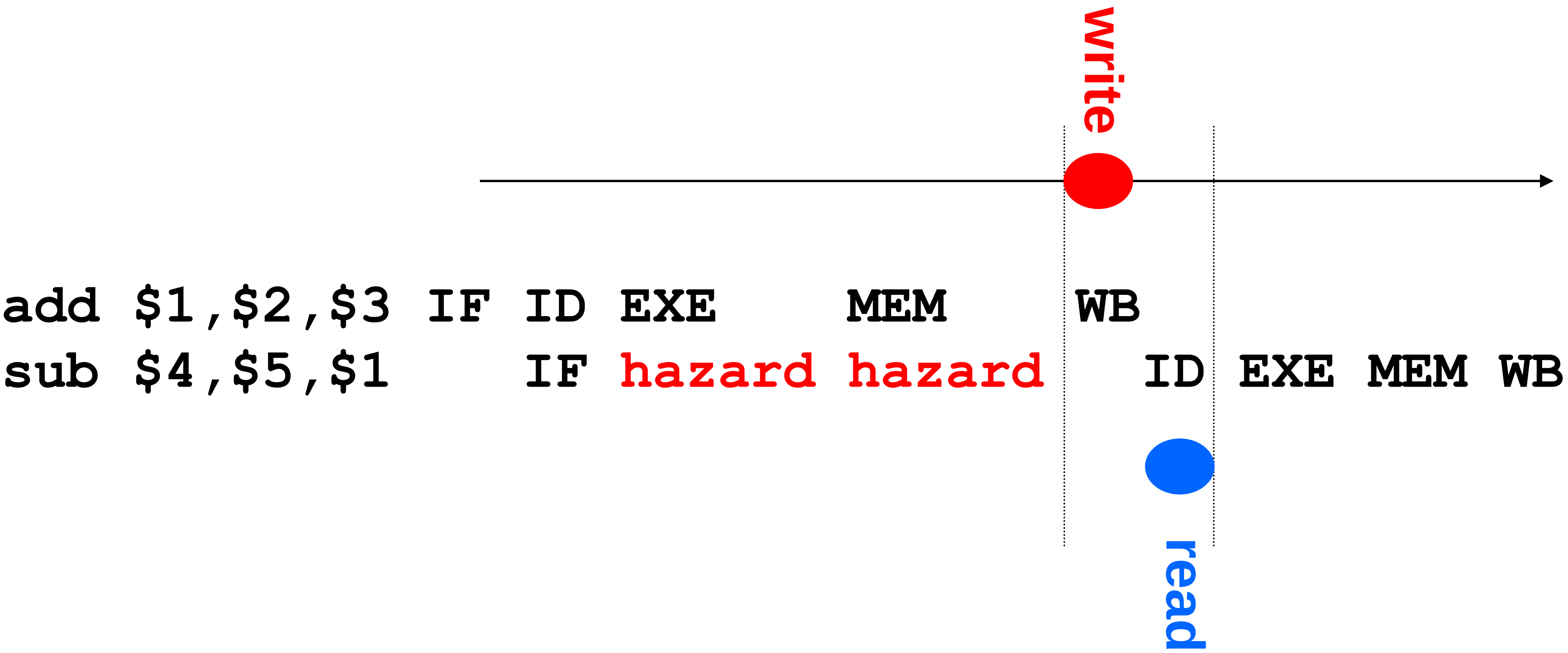
If we are not careful, we will read the wrong value!

If sub is supposed to read updated value (not stale), how many instructio
should be in between add and sub?

`sub $4,$5,$1   add $1,$2,$3`



Data Hazard



```
add $1,$2,$3 IF ID EXE     MEM      WB
sub $4,$5,$1    IF hazard hazard   ID EXE MEM WB
```

Class work

What are the data hazards in this piece of code?

```
add  $1,$2,$3
sub  $2,$1,$3
xor  $4,$3,$5
nor  $5,$2,$4
add  $5,$3,$5
```

What to do with them?

- Avoid
  - Make sure there are no hazards in the code
- Detect and Stall
  - If hazards exist, stall the processor until they go away.
- Detect and Forward
  - If hazards exist, fix up the pipeline to get the correct value (if possible)

First Approach: avoid all hazards

- Assume the programmer (or the compiler) knows about the processor implementation.
  - Make sure no hazards exist.
    - Consider if I have an instruction called *noop*. Put noops between any dependent instructions.

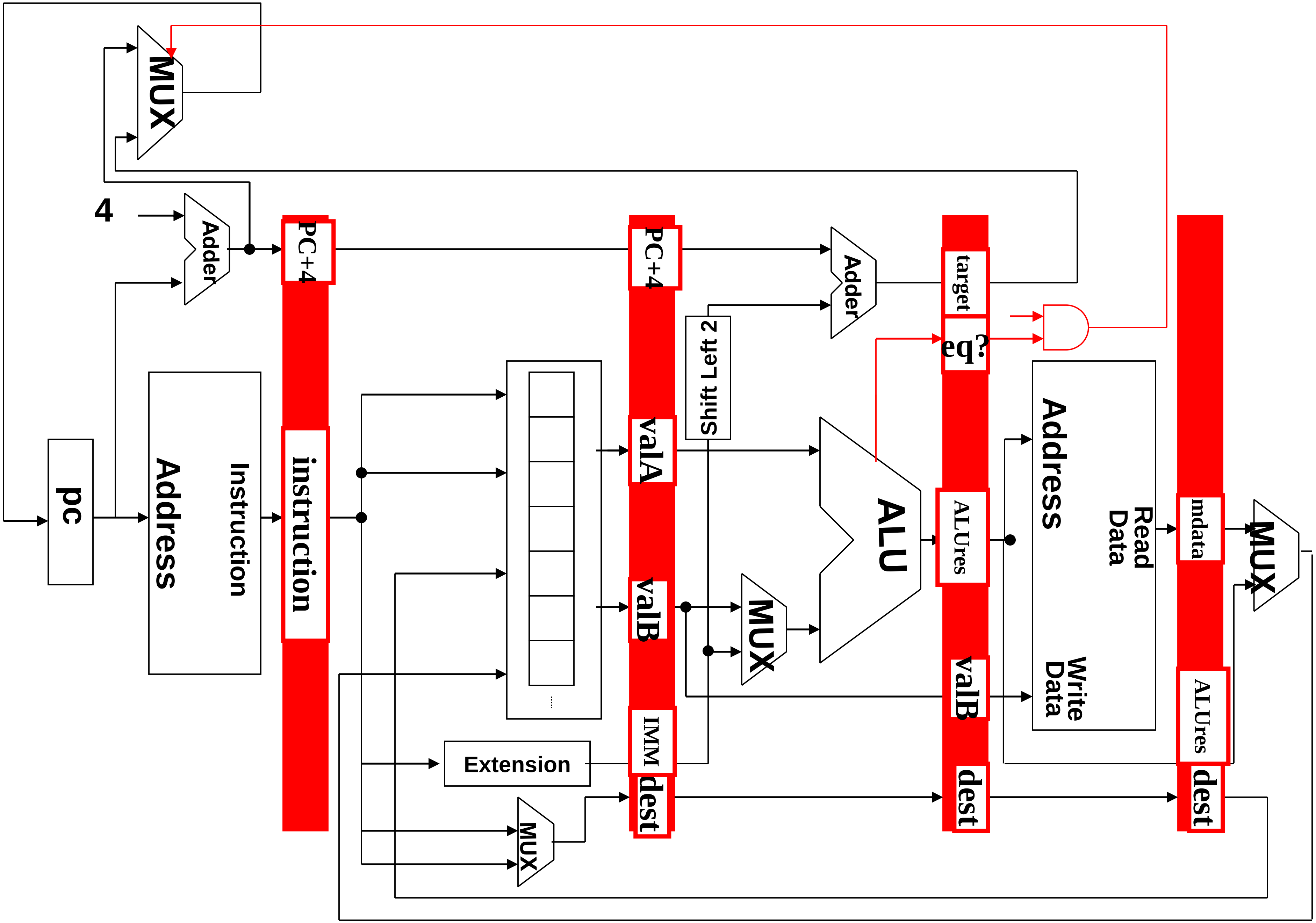
```
add  $1,$2,$3   IF  ID  EXE  MEM  WB
noop
noop
sub  $4,$5,$1                 IF  ID  EXE  MEM  WB
```

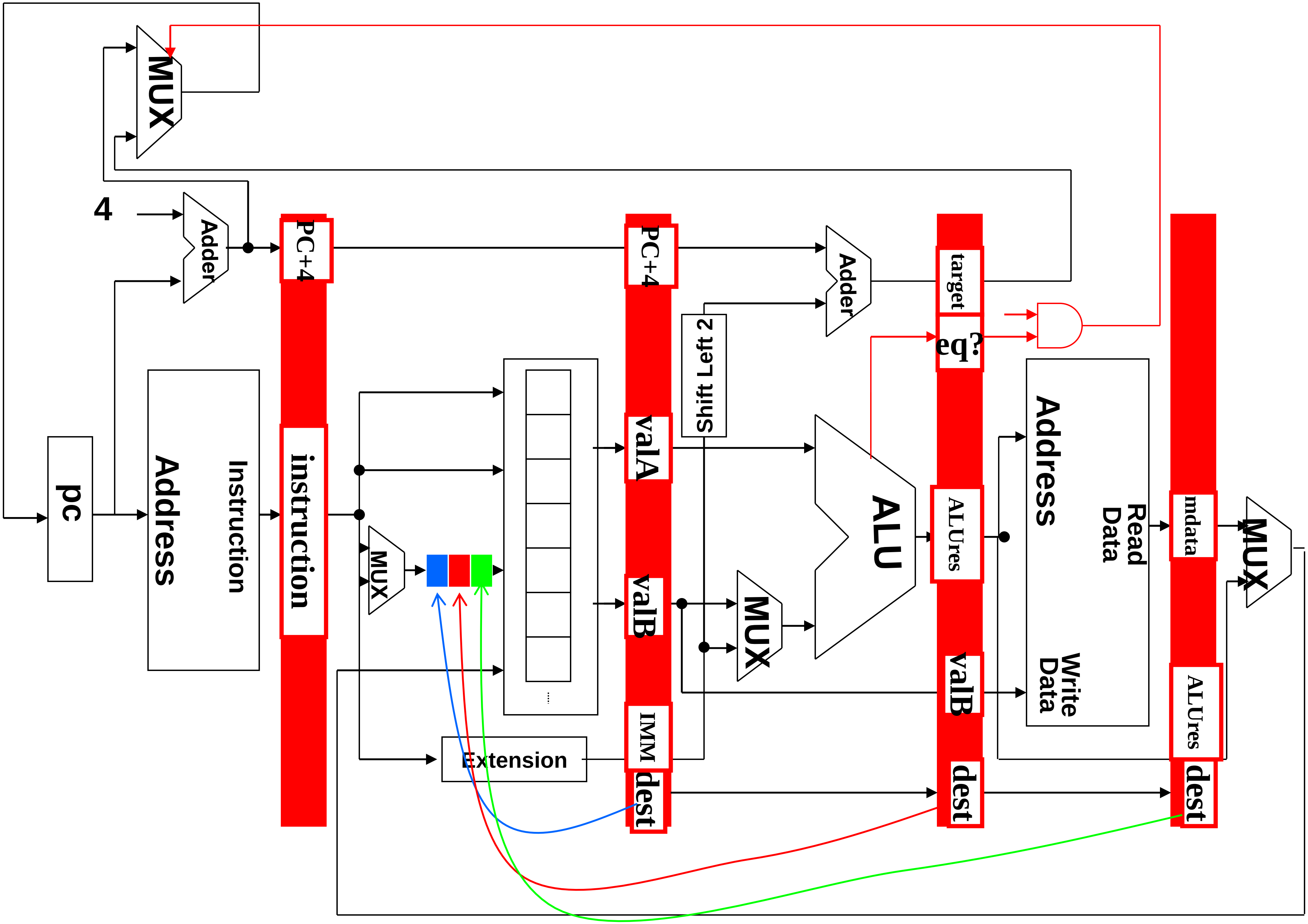
What is the problem with this solution?

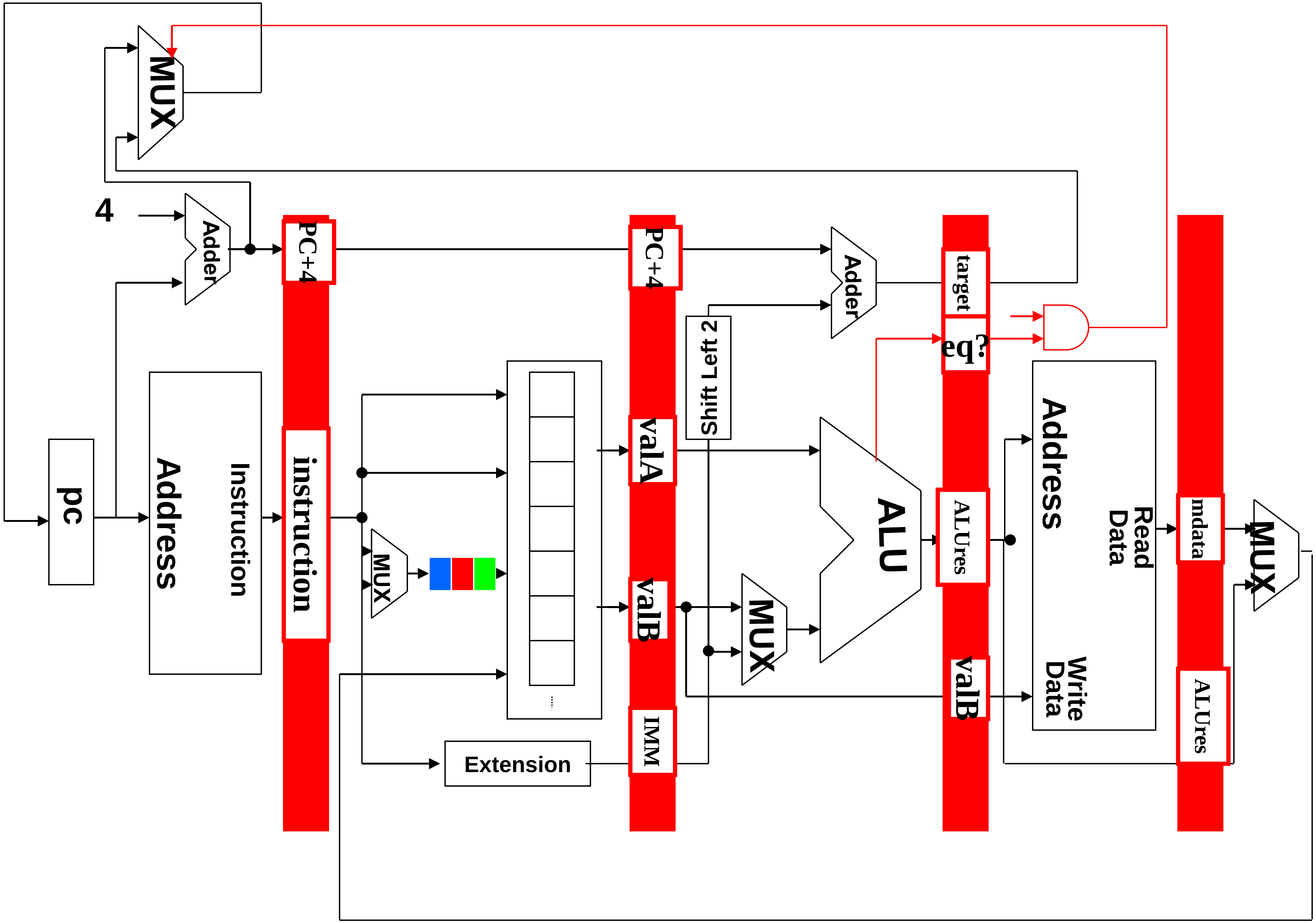
- Old programs (legacy code) may not run correctly on new implementations
  - Longer pipelines need more noops
- Programs get larger as noops are included
  - Especially a problem for machines that try to execute more than one instruction every cycle
  - Intel EPIC: Often 25% - 40% of instructions are noops
- Program execution is slower
  - CPI is 1, but some instructions are noops

The second solution

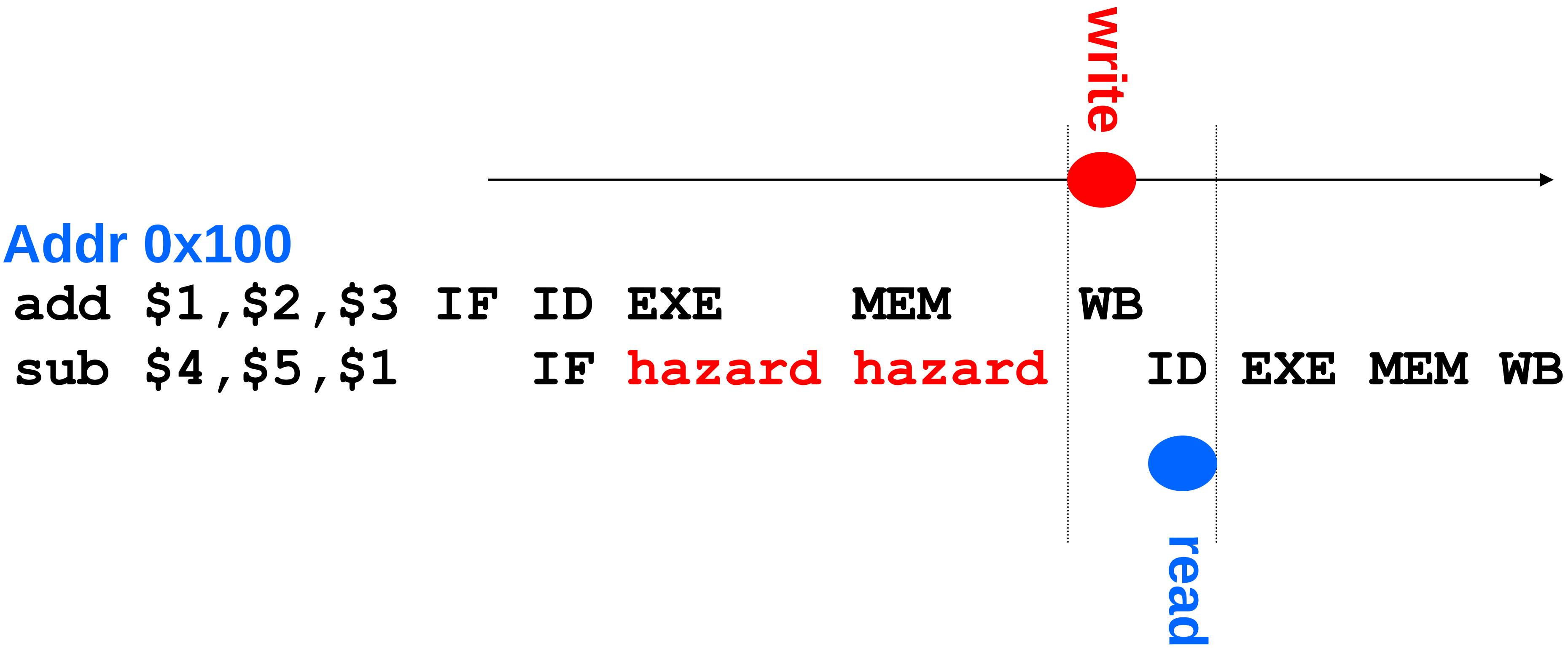
- Detect:
  - Compare regA with previous DestRegs
    - 5 bit operand fields
  - Compare regB with previous DestRegs
    - 5 bit operand fields
- Stall:
  - Keep current instructions in fetch and decode
  - Pass a noop to execute


MUX
4
Adder
PC+4
pc
Address
Instruction
instruction
PC+4
valA
valB
IMM
dest
Extension
MUX
Shift Left 2
MUX
ALU
Adder
target
eq?
ALUres
valB
dest
Address
Read Data
Write Data
mdata
ALUres
dest
MUX


MUX
4
Adder
pc
Address
Instruction
PC+4
instruction
MUX
Extension
PC+4
valA
valB
IMM
dest
Shift Left 2
Adder
MUX
ALU
target
eq?
ALUres
valB
dest
Address
Read Data
Write Data
mdata
ALUres
dest
MUX


MUX
4
Adder
PC+4
PC+4
Adder
target
eq?
pc
Address
Instruction
instruction
MUX
Shift Left 2
valA
valB
IMM
MUX
ALU
ALUres
valB
Address
Read Data
Write Data
mdata
ALUres
MUX
Extension

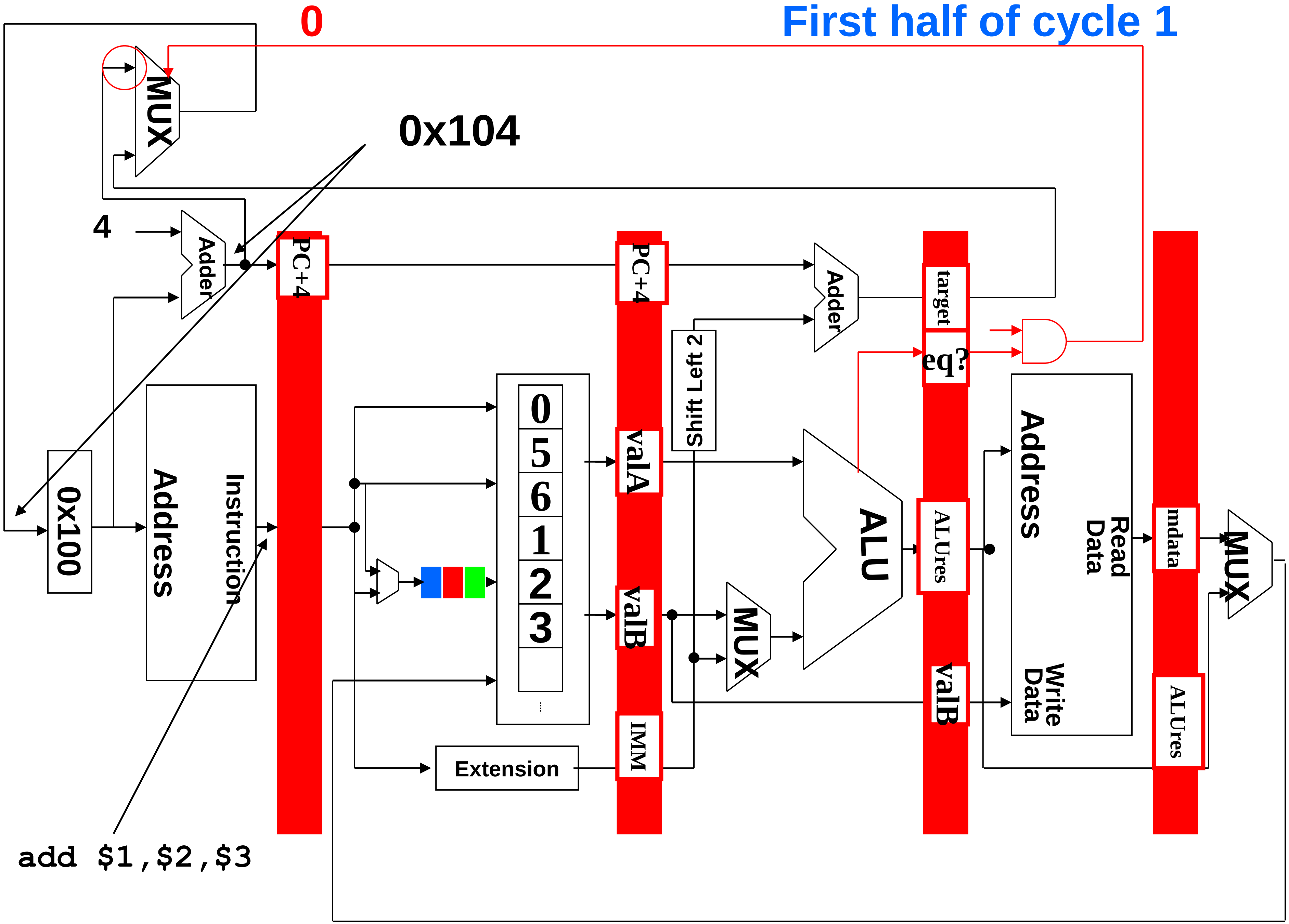
Data Hazard



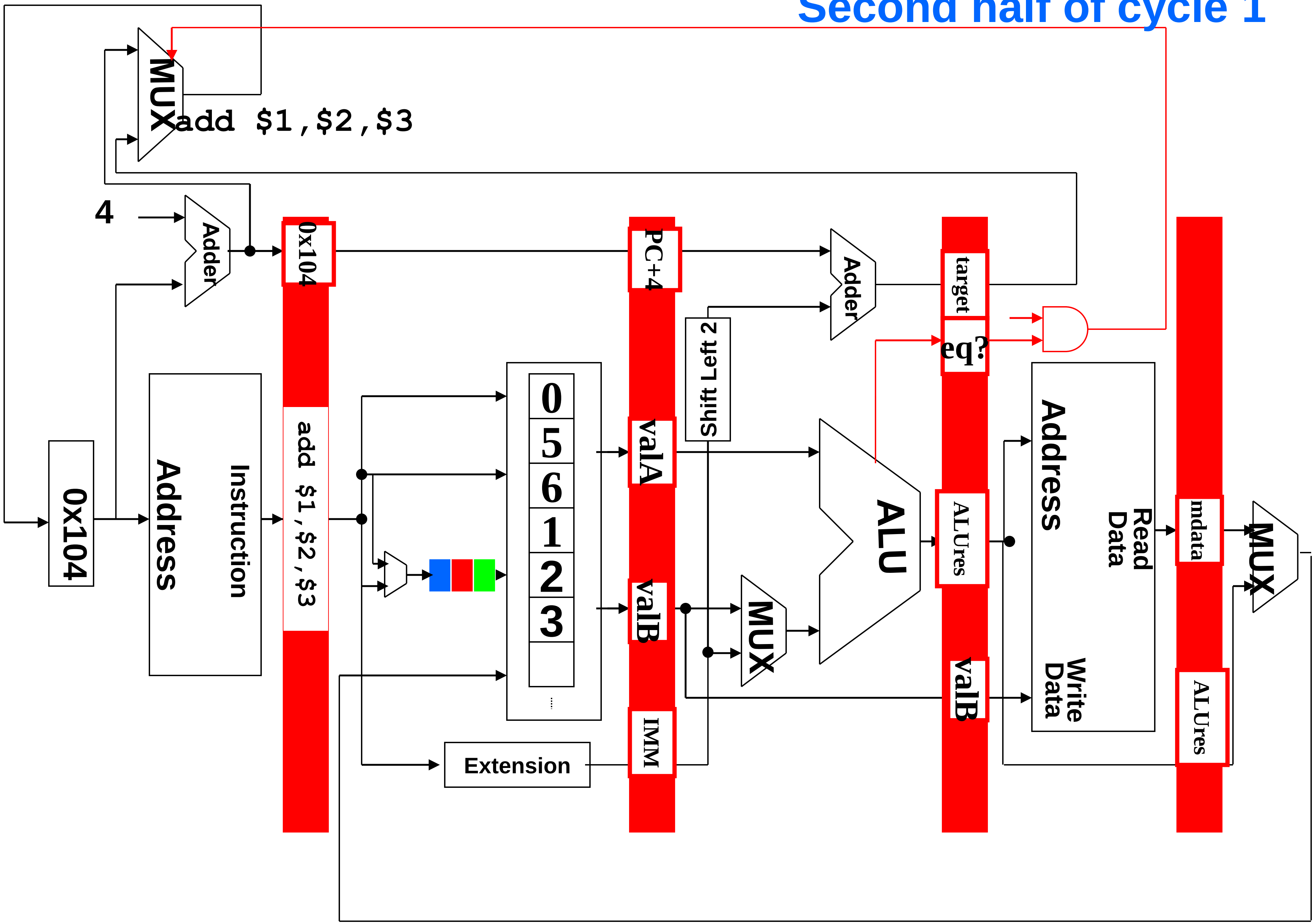
Addr 0x100

```
add $1,$2,$3 IF ID EXE     MEM     WB
sub $4,$5,$1    IF hazard hazard    ID EXE MEM WB
```

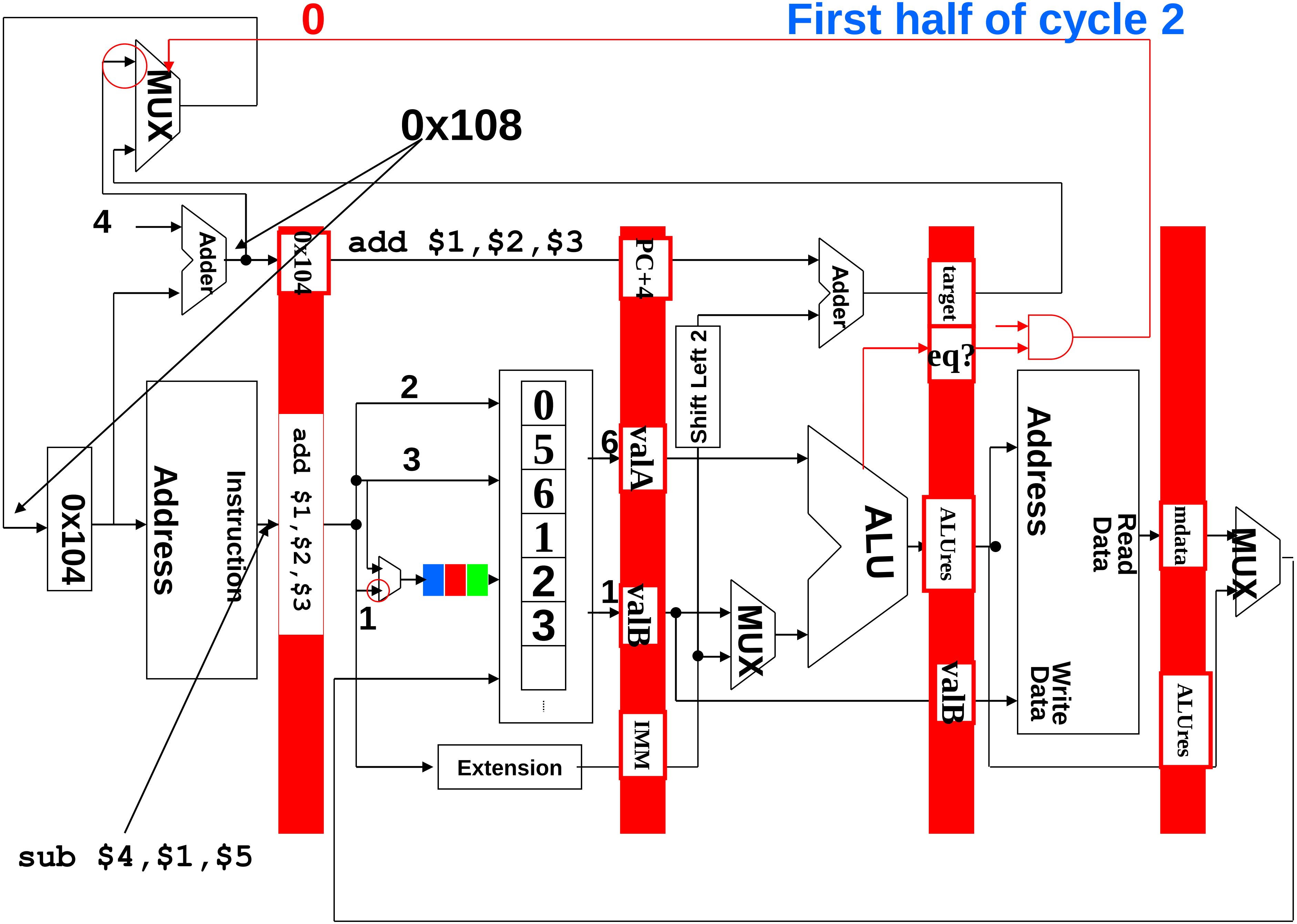
First half of cycle 1



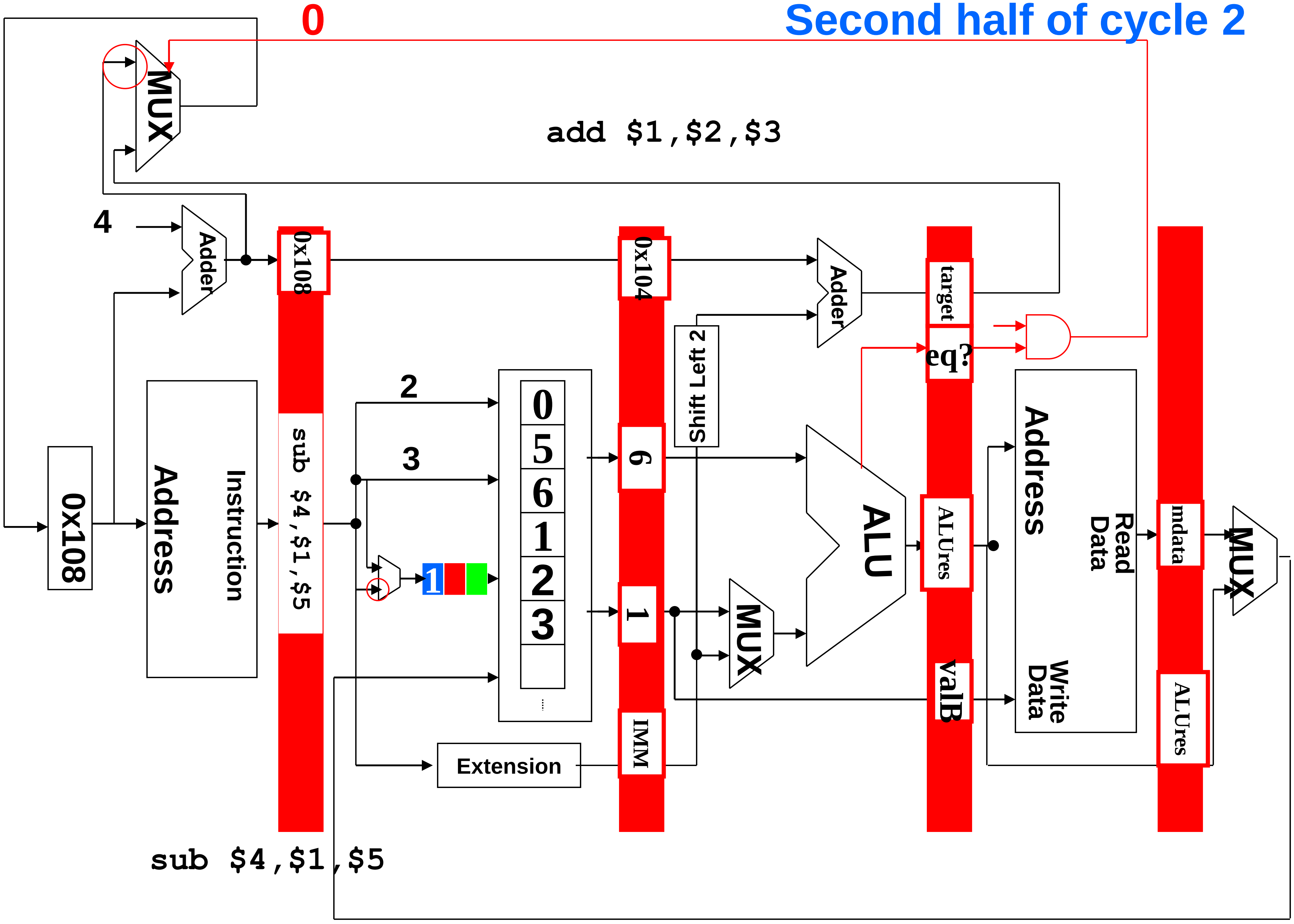
Second half of cycle 1



First half of cycle 2



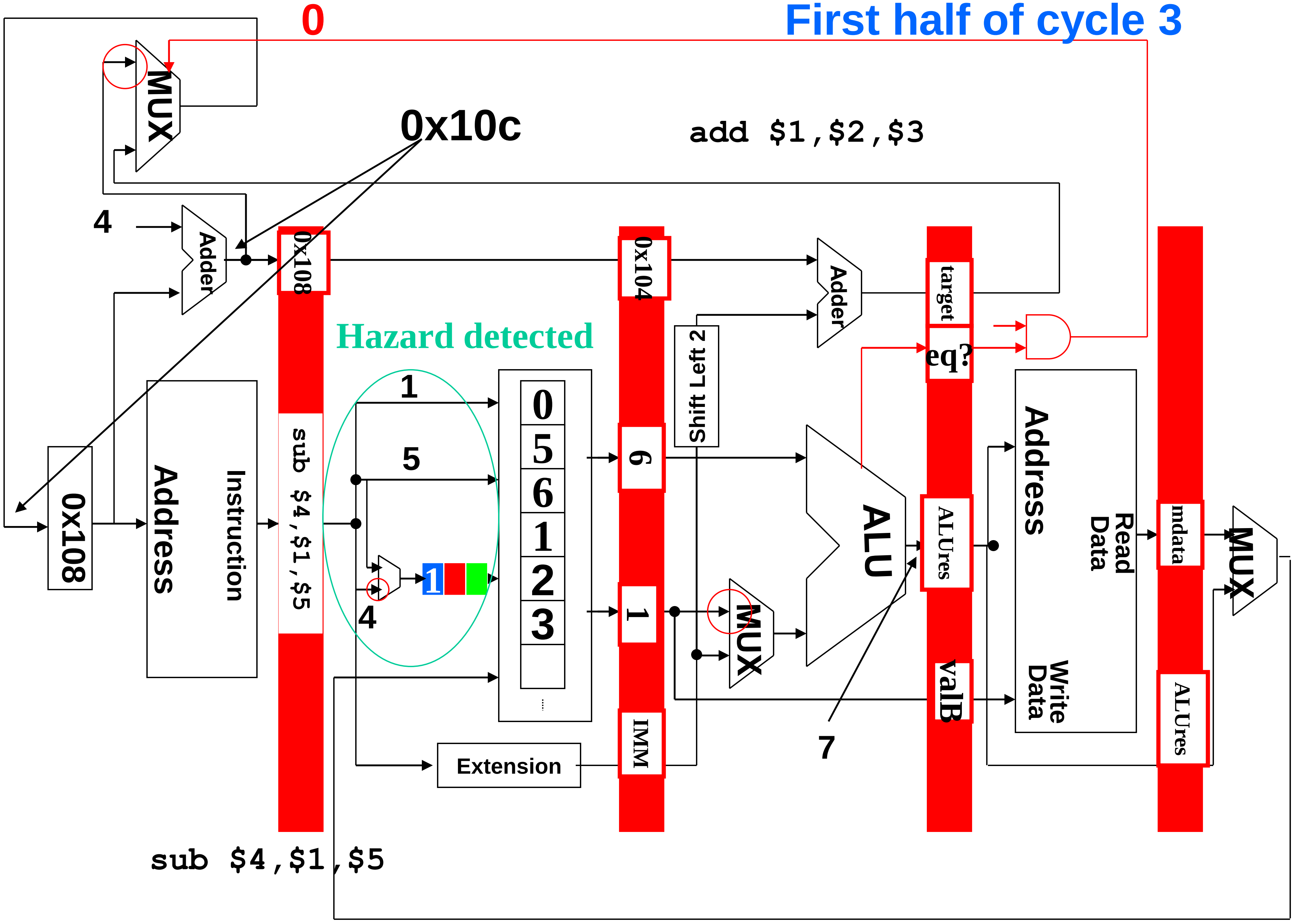
Second half of cycle 2

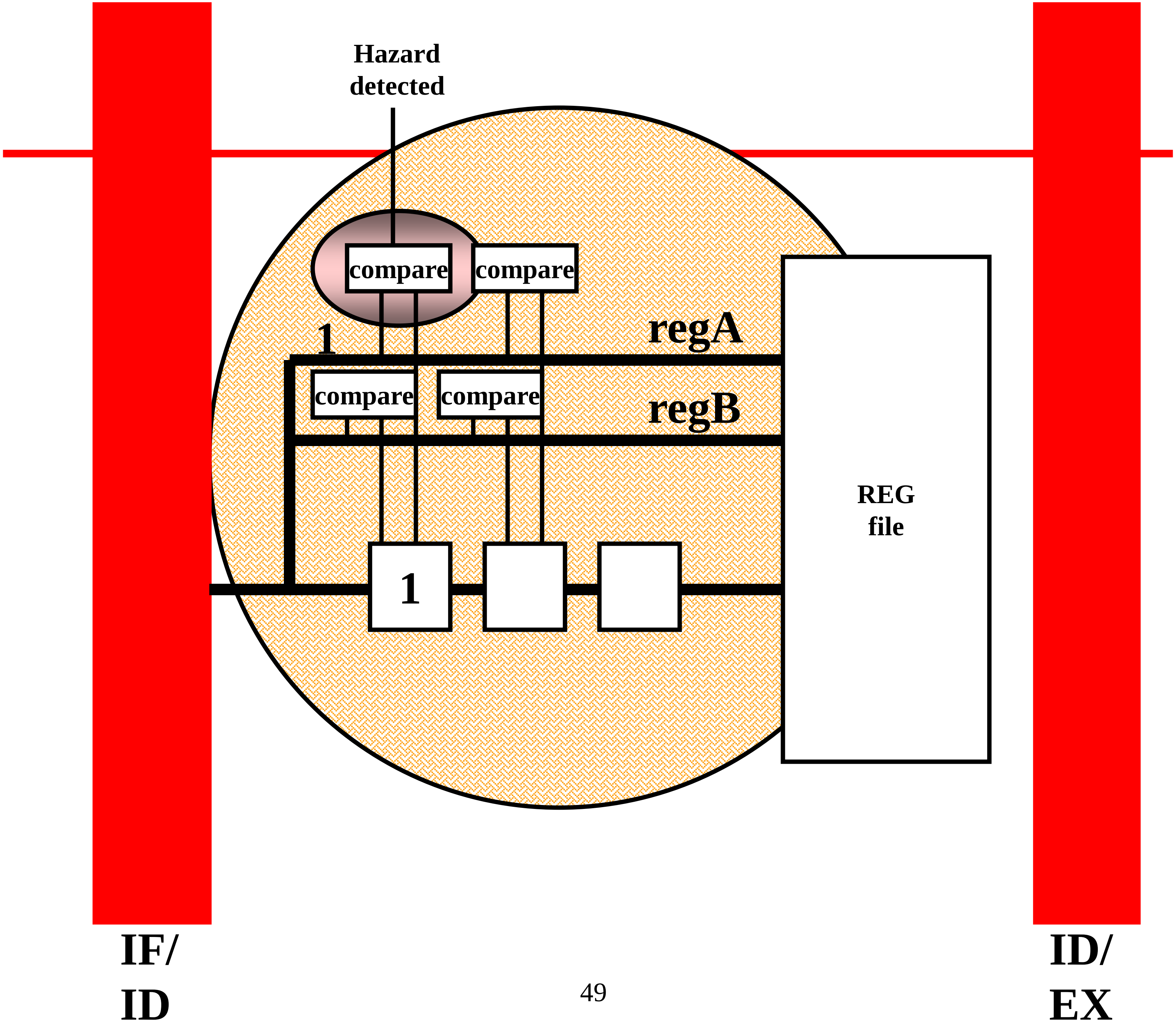


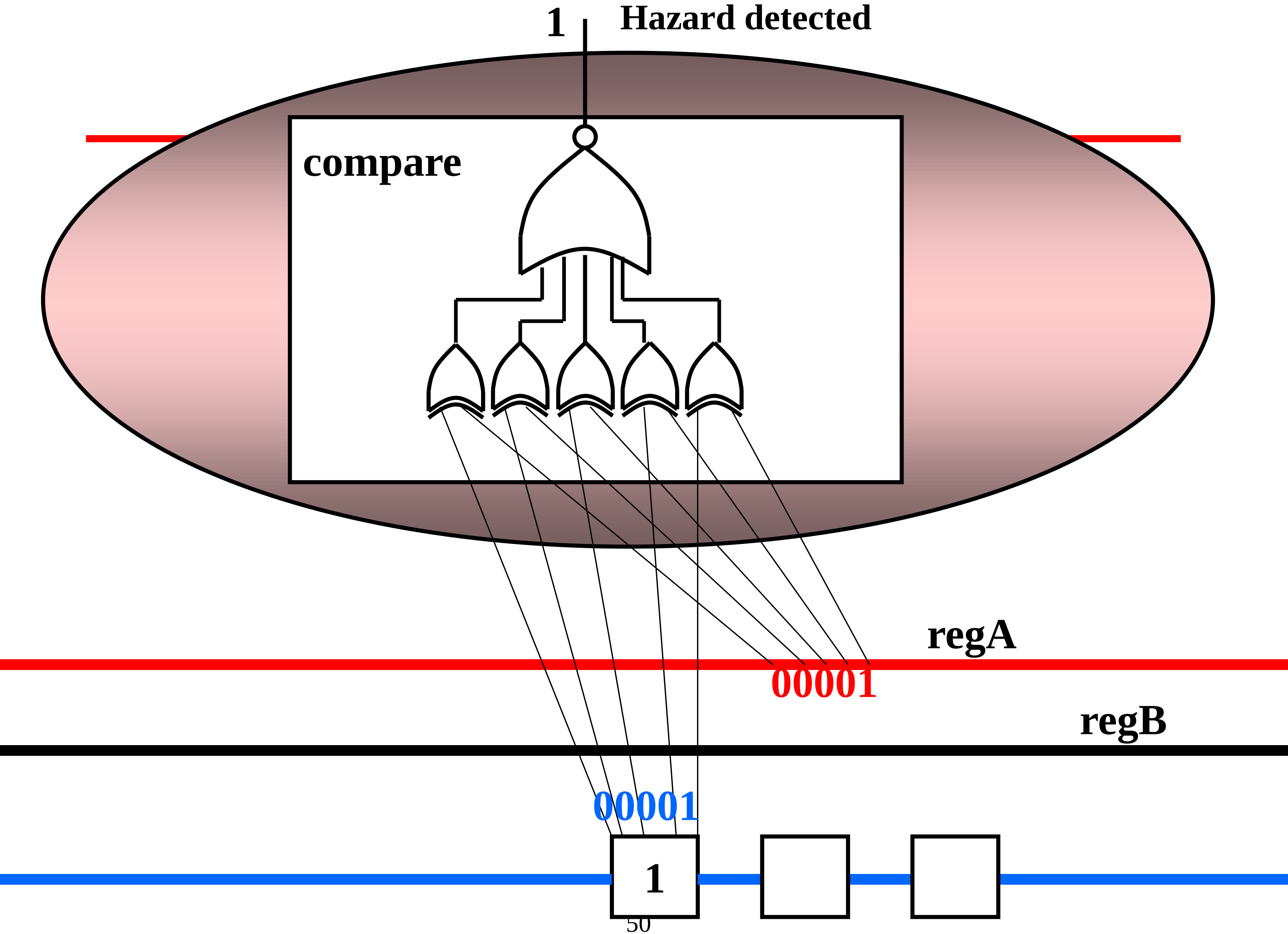
add $1,$2,$3

sub $4,$1,$5

First half of cycle 3




Hazard
detected
compare
compare
1
regA
compare
compare
regB
REG
file
1
IF/
ID
ID/
EX

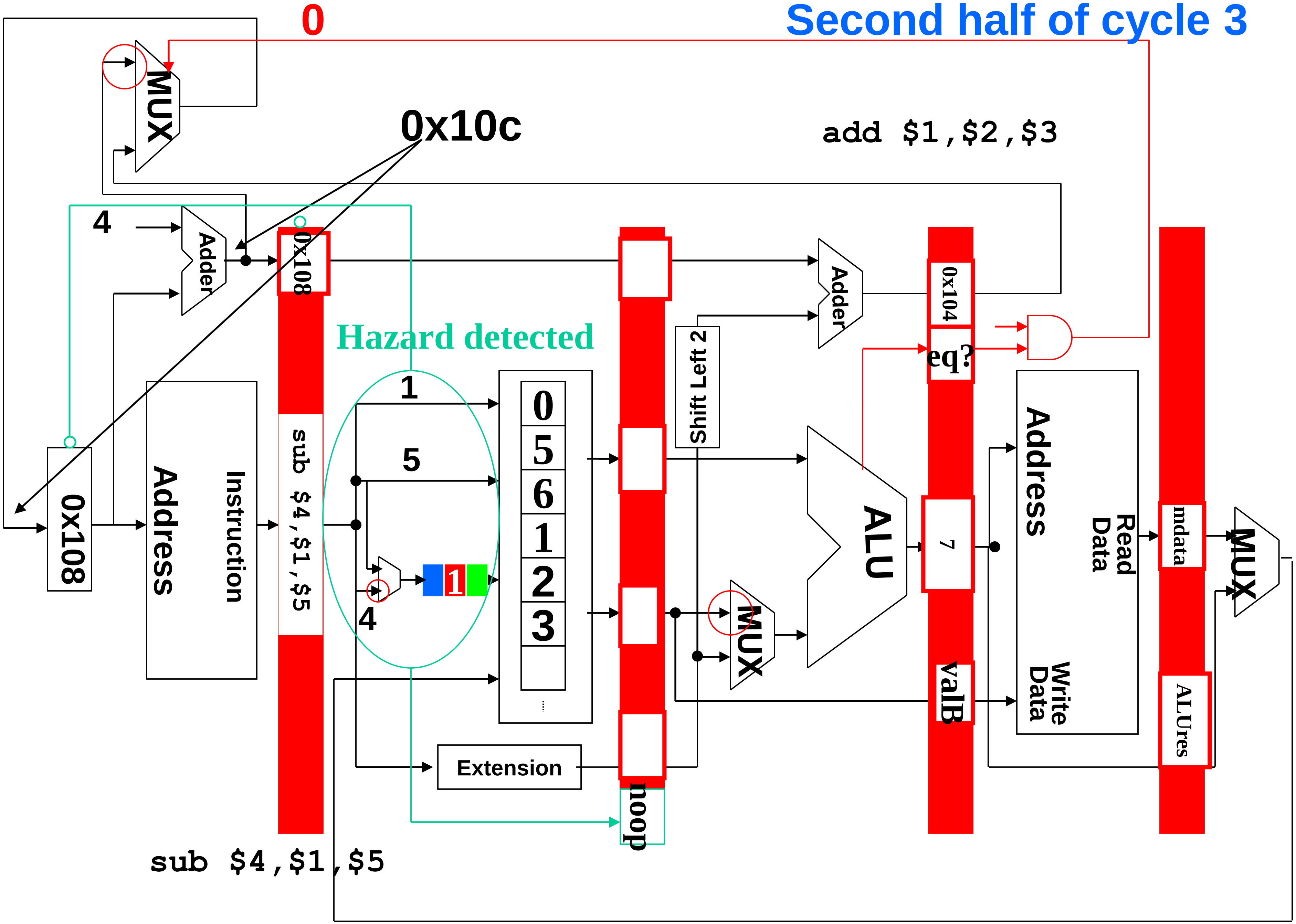

1
Hazard detected
compare
regA
00001
regB
00001
1

What Next?

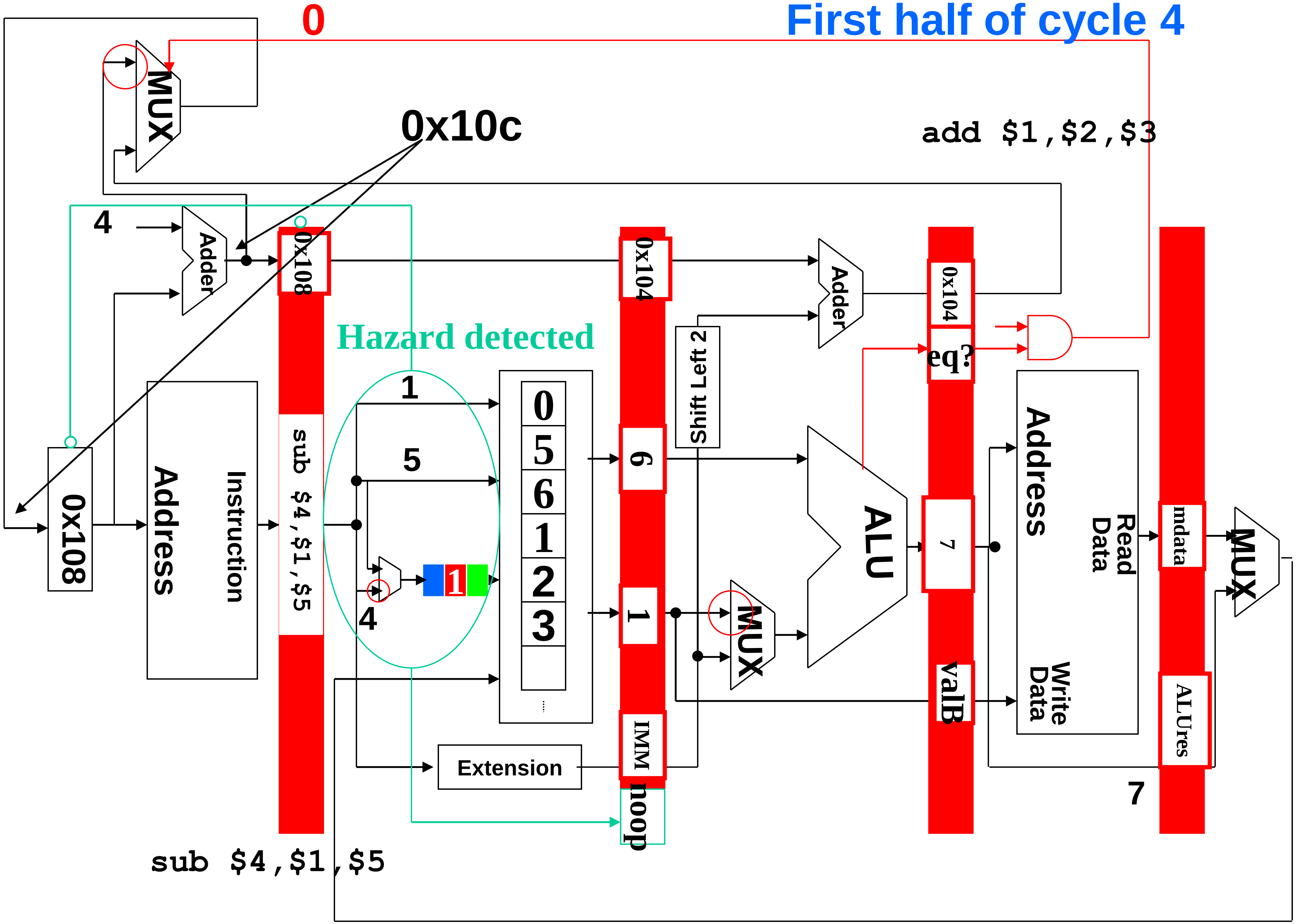
- Detect:
  - Compare regA with previous DestRegs
    - 5 bit operand fields
  - Compare regB with previous DestRegs
    - 5 bit operand fields

- Stall:
  - Keep current instructions in fetch and decode
  - Pass a noop to execute

Second half of cycle 3



First half of cycle 4

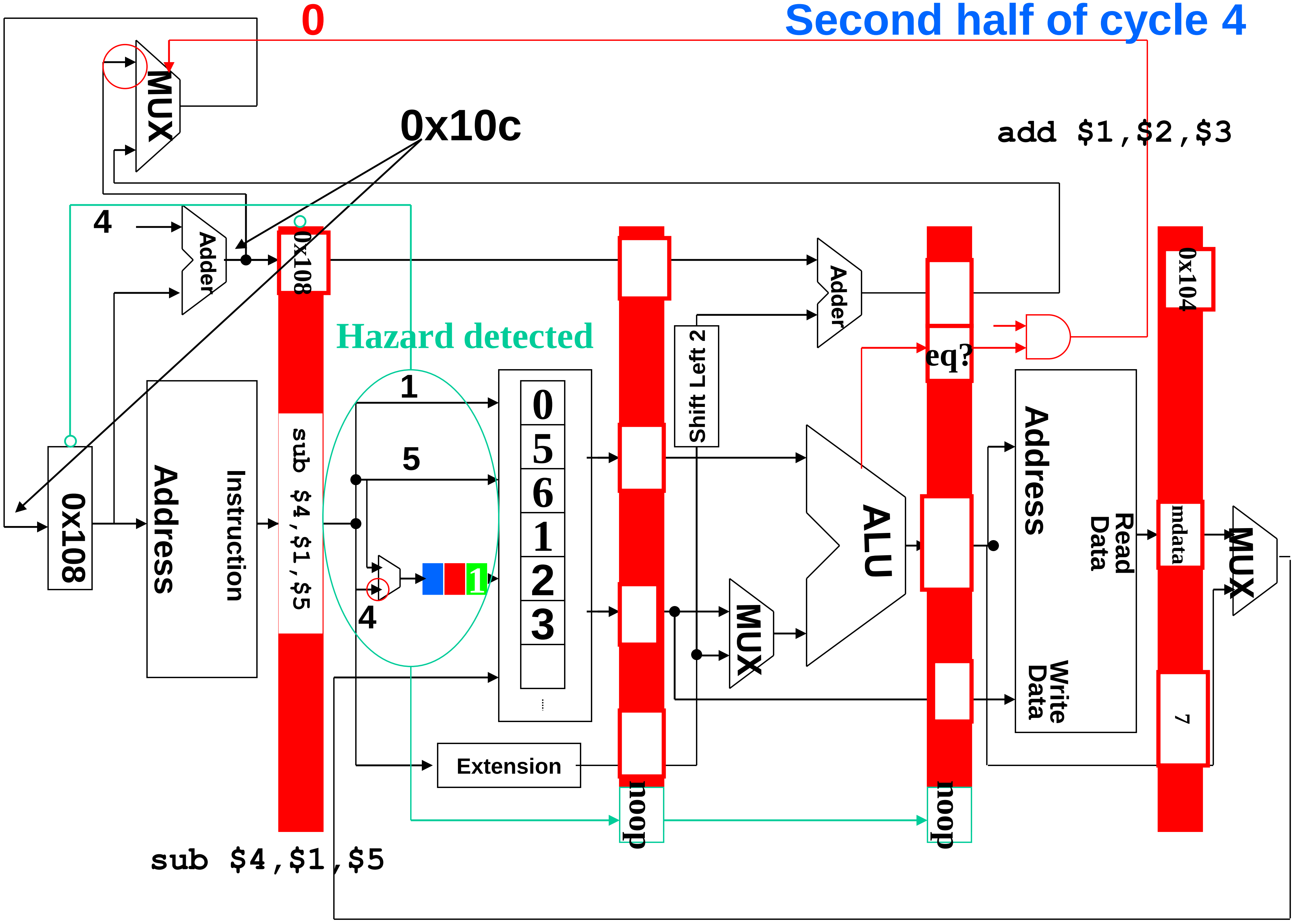


Hazard detected

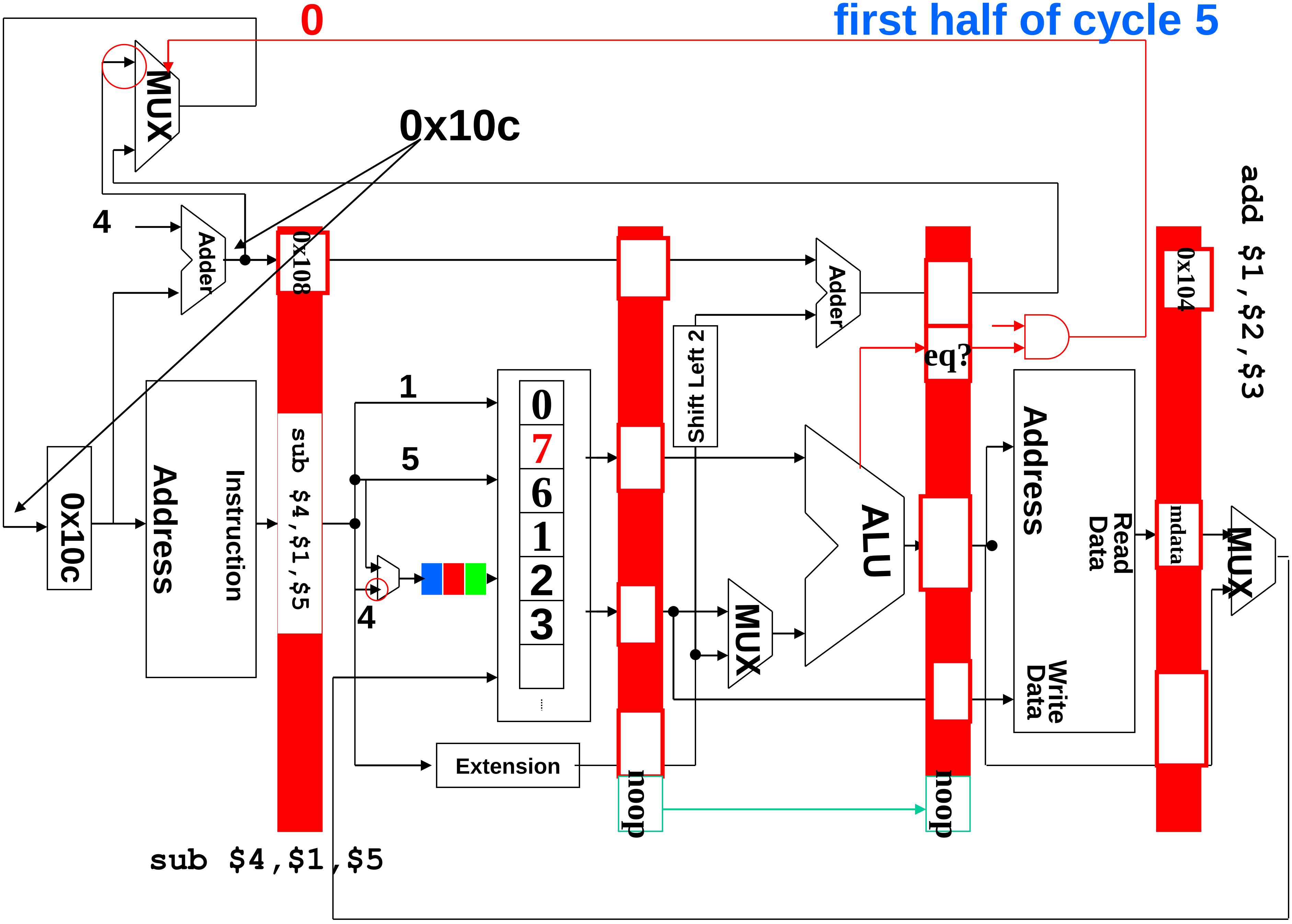
add $1,$2,$3

sub $4,$1,$5

Second half of cycle 4

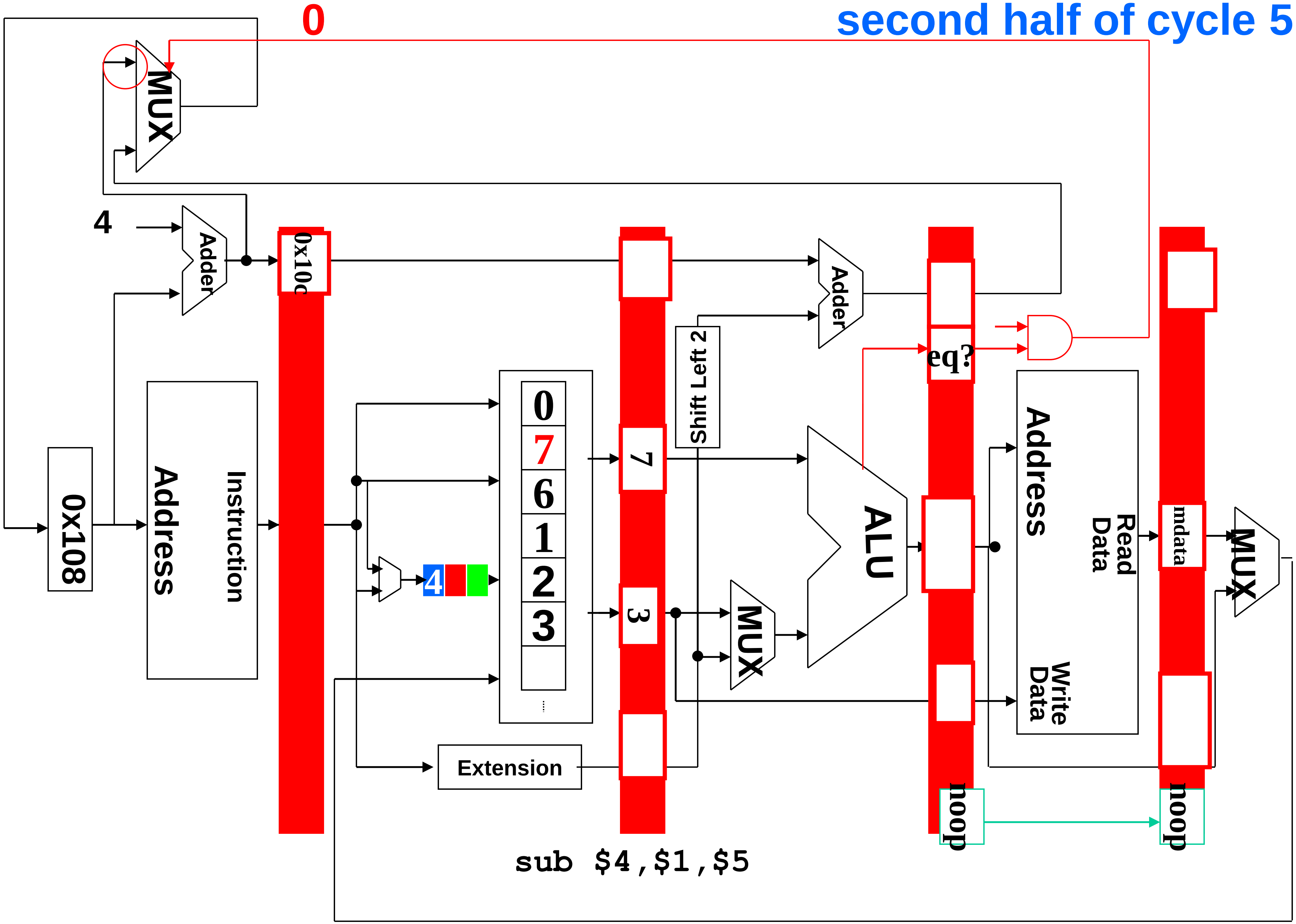


first half of cycle 5



sub $4,$1,$5

second half of cycle 5



sub $4,$1,$5

Timing graph

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13
`add $1,$2,$3`	IF	ID	EX	ME	WB								
`Sub $4,$1,$5`		IF	no op	no op	ID	EX	ME	WB					
`add $6,$1,$7`					IF	ID	EX	ME	WB				
`lw $6,10($8)`						IF	ID	EX	ME	WB			
`sw $6,13($1)`							IF	no op	no op	ID	EX	ME	

Problems with the second solution

- Still CPI is the same as before, no improvement in performance

- The only improvement is in the code size, and no longer compiler is responsible to detect the data hazards

- In fact, now the system runs slower
  - Why?

The third solution

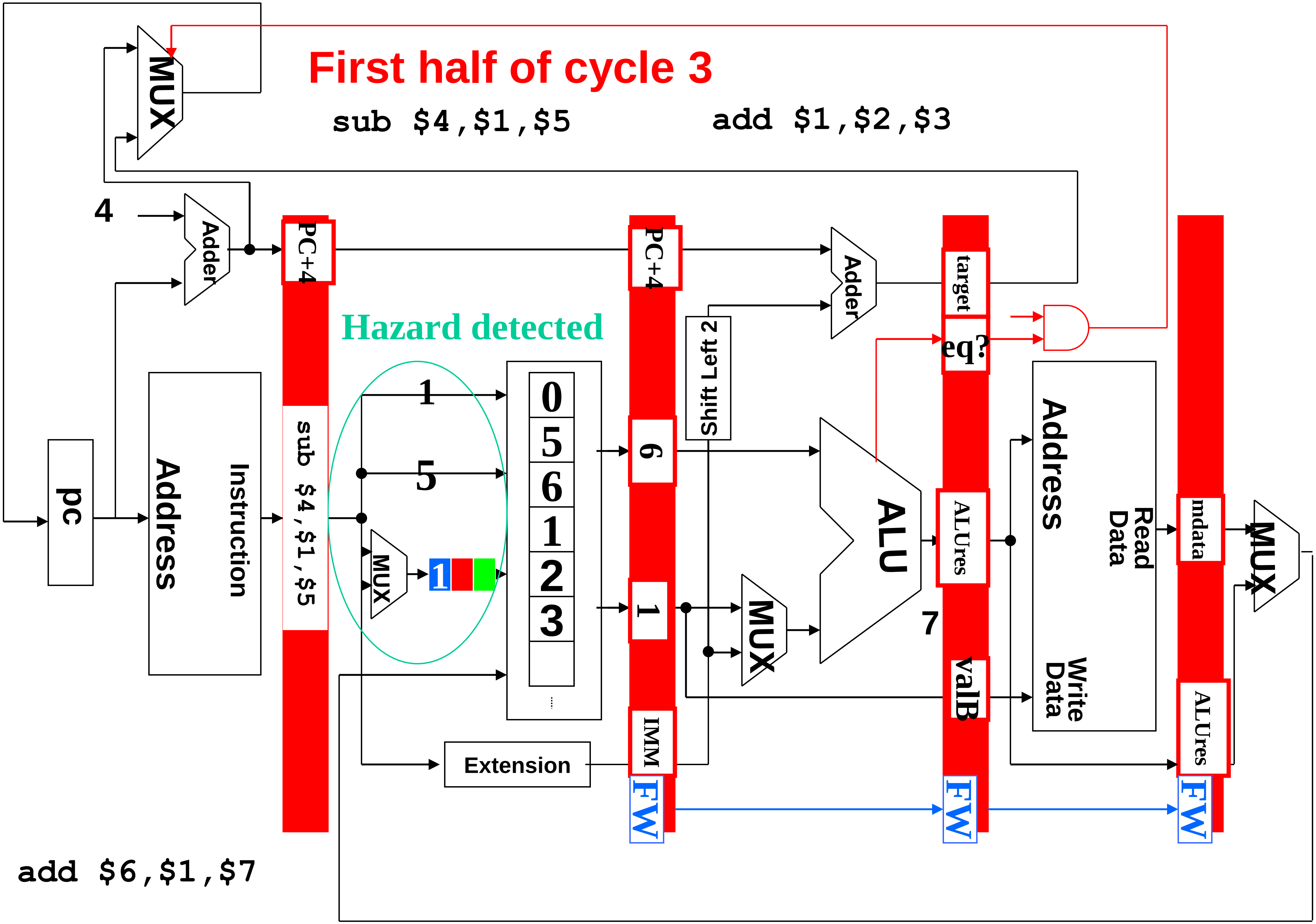
- Detect the data hazard
- Add instruction calculated the result in the execution cycle
- Forward the result to the decode stage of the sub instruction
- Therefore
  - sub does not need to wait until the result is written back into register file
  - And more control is needed; place the result somewhere else rather than register file

The third solution

- Detect: same as detect and stall
  - Except that all 4 hazards are treated differently
    - i.e., you can’t logical-OR the 4 hazard signals
- Forward:
  - New bypass datapaths route computed data to where it is needed
  - New MUX and control to pick the right data
- Beware: Stalling may still be required even in the presence of forwarding

First half of cycle 3

`sub $4,$1,$5`   `add $1,$2,$3`

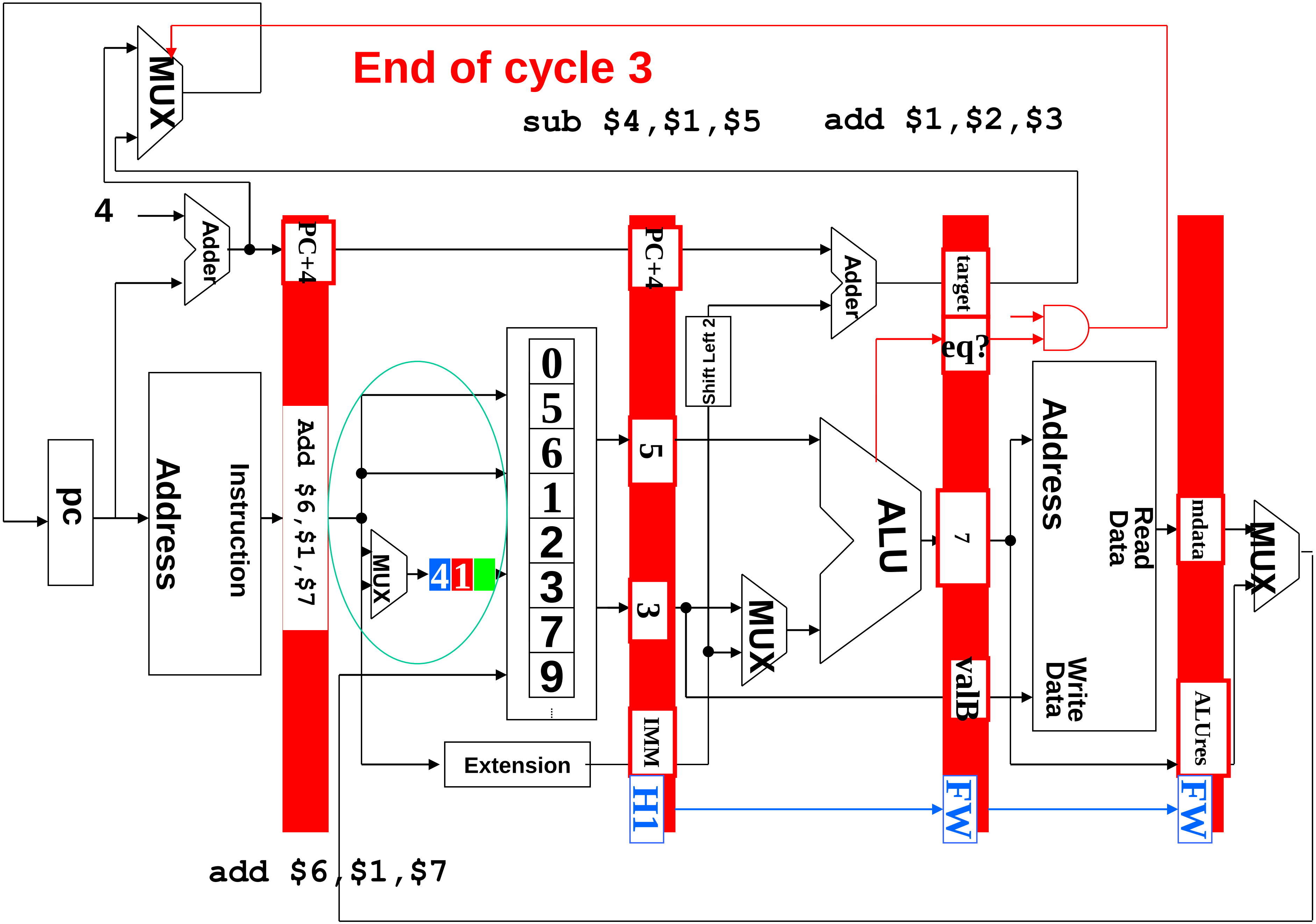


Hazard detected

`add $6,$1,$7`

End of cycle 3

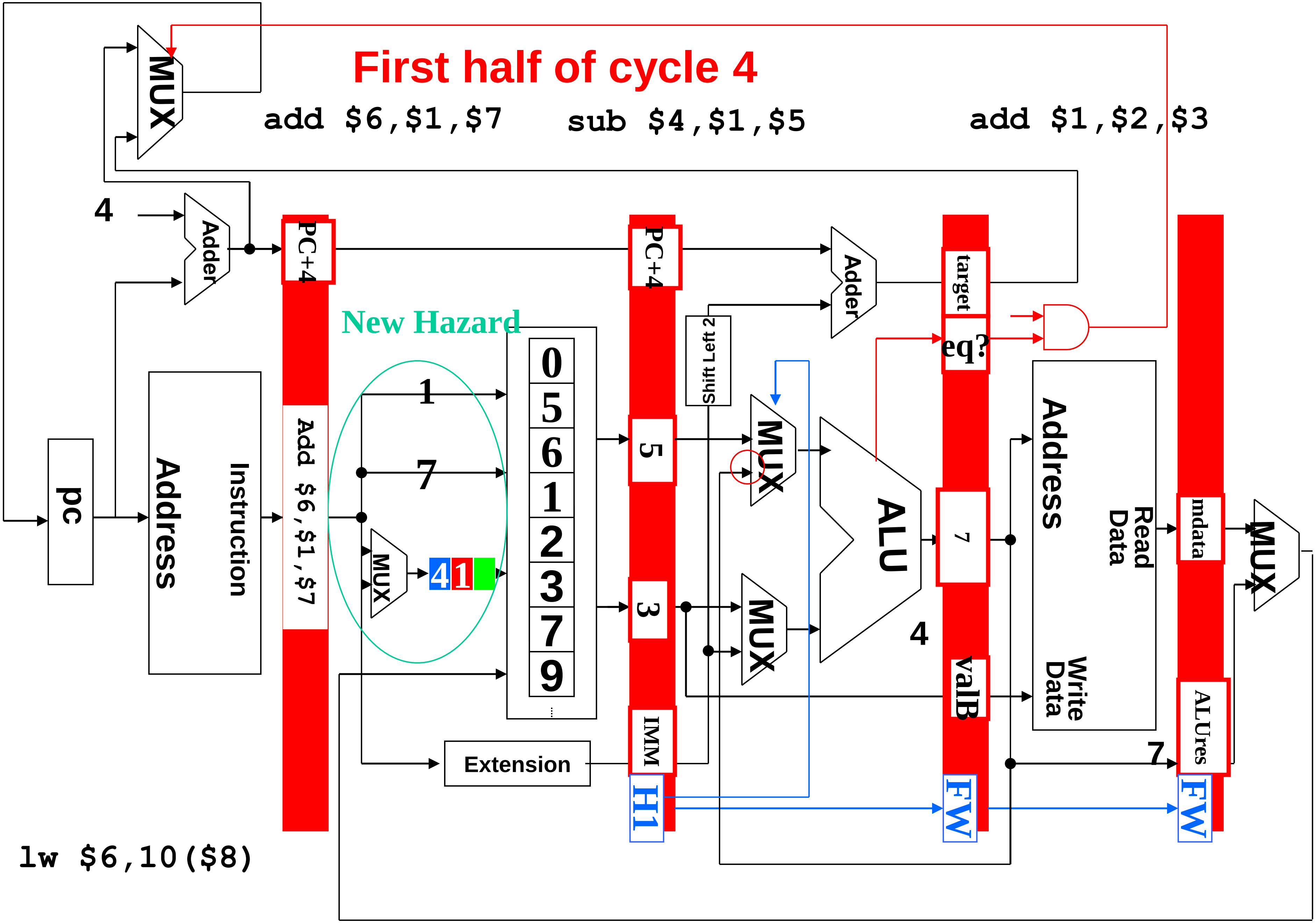
sub $4,$1,$5   add $1,$2,$3



add $6,$1,$7

First half of cycle 4

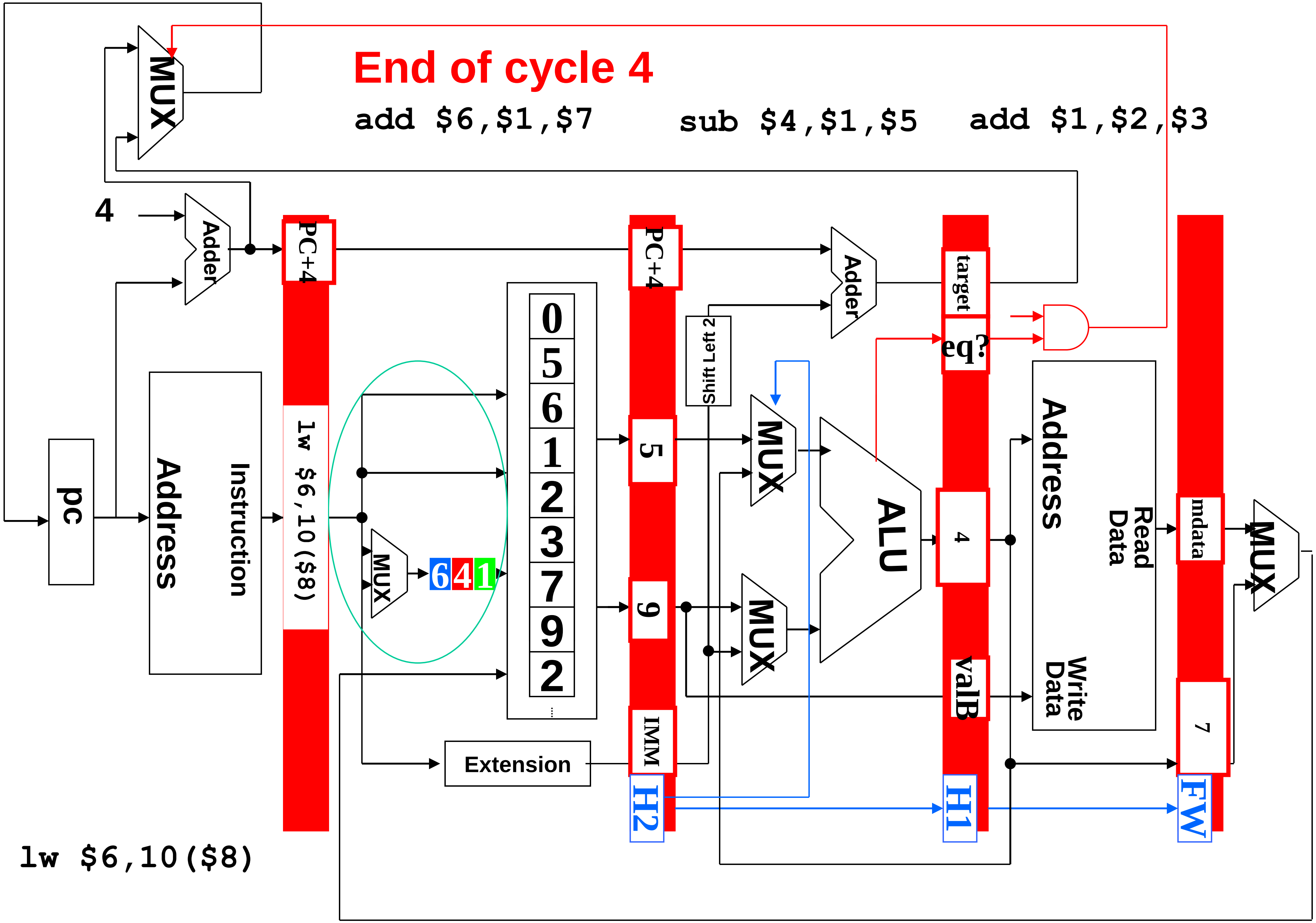
add $6,$1,$7 sub $4,$1,$5 add $1,$2,$3



lw $6,10($8)

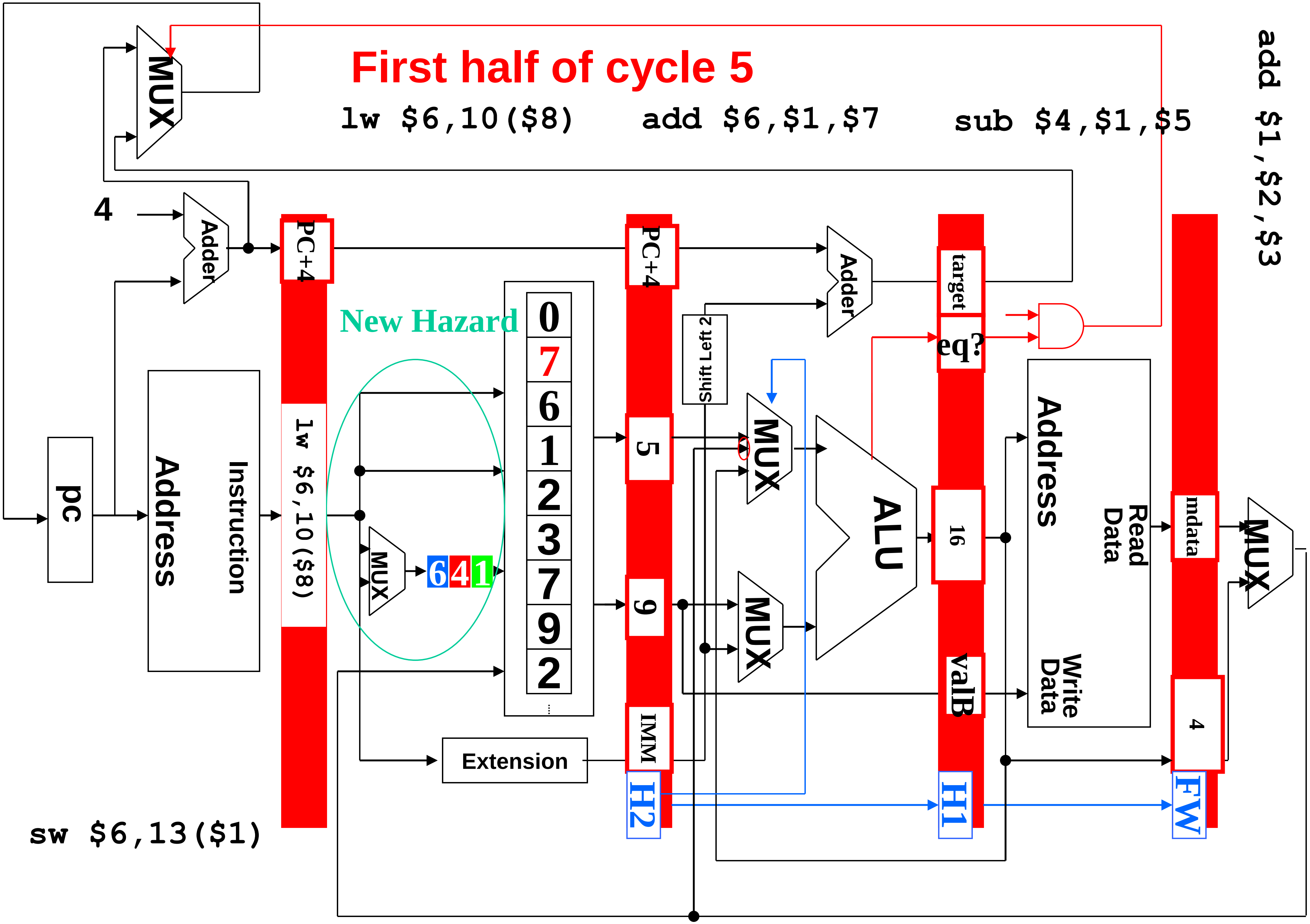
End of cycle 4

add $6,$1,$7 sub $4,$1,$5 add $1,$2,$3



lw $6,10($8)

First half of cycle 5



sw $6,13($1)   lw $6,10($8)   add $6,$1,$7   sub $4,$1,$5   add $1,$2,$3

What else can go wrong in our pipelined CPU?

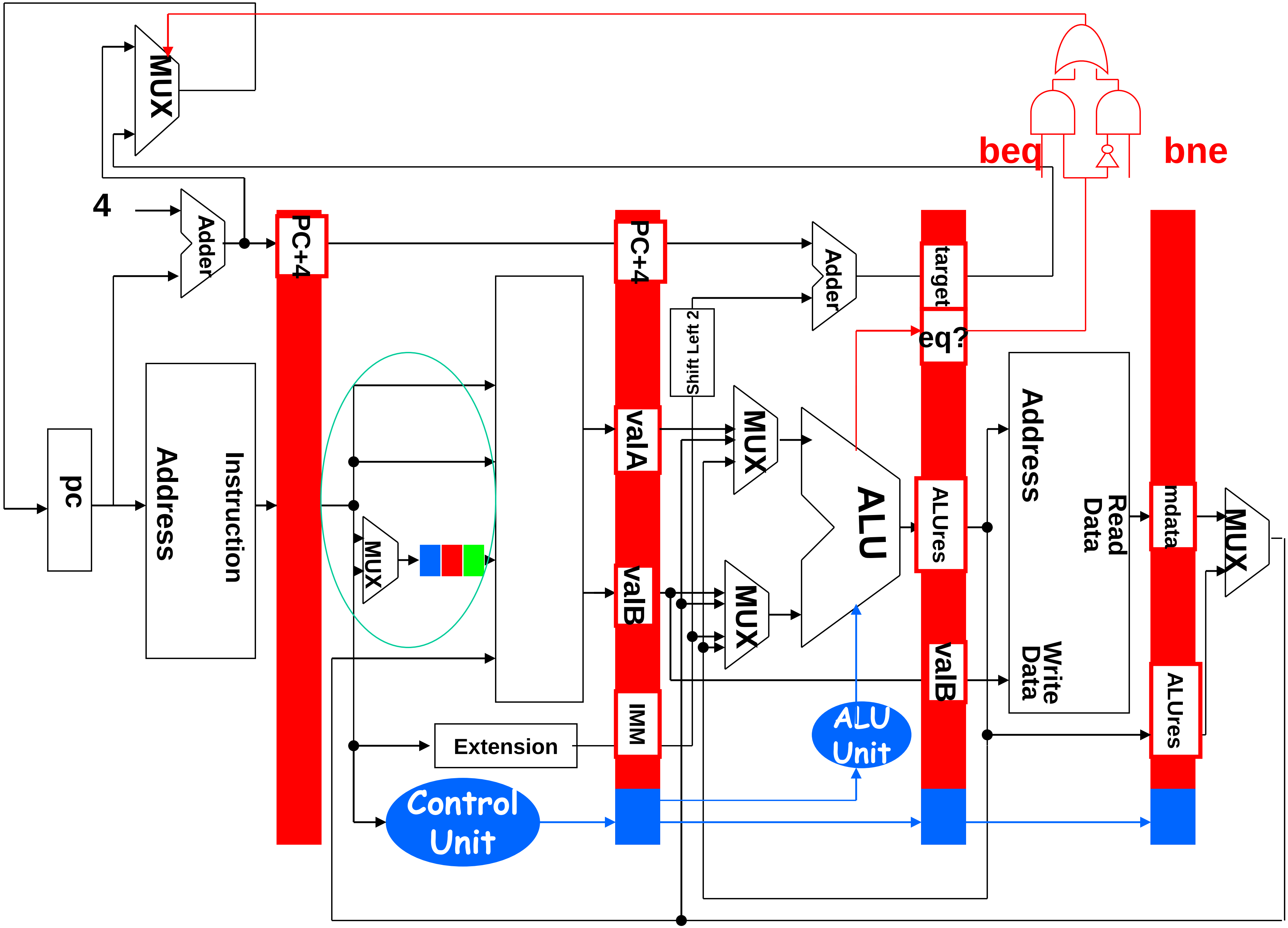
- Control hazards

- Exceptions: First of all, what are exceptions? And, how do you handle exceptions in a pipelined processor with 5 instructions in flight?

Control Hazard

- What is a control hazard?

- How does the pipelined CPU handle control hazards?


MUX
beq
bne
4
Adder
PC+4
PC+4
Adder
target
eq?
Shift Left 2
pc
Address
Instruction
MUX
valA
valB
IMM
MUX
MUX
ALU
ALUres
valB
Address
Read Data
Write Data
mdata
ALUres
MUX
Extension
ALU Unit
Control Unit

What happens in executing BEQ?

- Fetch: read instruction from memory
- Decode: read source operands from reg
- Execute: calculate target address and test for equality
- Memory: **Send target to PC** if test is equal
- Writeback: Nothing left to do

Example

```
y=y*2;
x=0;
for(j=100;j>0;j--){
      x++;
      z--;
}
y--;
x=x*3;
z=z+x;
```

```
100 add  $3,$3,$3
104 add  $2,$0,$0
108 li   $5,100
112 addi $2,$2,1
116 addi $4,$4,-1
120 addi $5,$5,-1
124 bne  $5,$0,112
128 addi $3,$3,-1
132 add  $5,$2,$0
136 add  $2,$2,$2
140 add  $2,$2,$5
144 add  $4,$4,$2
```

What do you observe from the example?

- How many times the branch is taken?
- How many times is not taken?
- What happens each time that the branch instruction is executed?
- What happens next?

Surprise!

```
112 addi $2,$2,1
           .  .  .
124 bne  $5,$0,-4
128 addi $3,$3,-1
132 add  $5,$2,$0
136 add  $2,$2,$2
```

124	IF	ID	EXE	MEM	WB				
128		IF	ID	EXE	MEM	WB			
132			IF	ID	EXE	MEM	WB		
136				IF	ID	EXE	MEM	WB	
112					IF	ID	EXE	MEM	WB

Solutions

- Avoid
  - Make sure there are no hazards in the code
- Detect and Stall
  - Delay fetch until branch resolved.
- Speculate and Squash-if-Wrong
  - Go ahead and fetch more instruction in case it is correct, but stop them if they shouldn’t have been executed

Avoid

- Don’t have branch instructions!
  - Maybe a little impractical ☺

- Delay taking branch:

  dbeq R1,R2,offset

  dbne R1,R2,offset

  Instructions at PC+4, PC+8, etc will execute before deciding whether to fetch from PC+4+offset. (If no useful instructions can be placed after dbeq, noops must be inserted.)

Consider our example again

```
100 add  $3,$3,$3
104 add  $2,$0,$0
108 li   $5,100
112 addi $2,$2,1
116 addi $4,$4,-1
120 addi $5,$5,-1
124 bne  $5,$0,-4
128 addi $3,$3,-1
132 add  $5,$2,$0
136 add  $2,$2,$2
140 add  $2,$2,$5
144 add  $4,$4,$2
```

```
100 add  $3,$3,$3
104 add  $2,$0,$0
108 li   $5,100
112 addi $2,$2,1
116 addi $4,$4,-1
120 addi $5,$5,-1
124 bne  $5,$0,-4
128 noop
132 noop
136 noop
140 addi $3,$3,-1
144 add  $5,$2,$0
148 add  $2,$2,$2
152 add  $2,$2,$5
156 add  $4,$4,$2
```

Can we do better?

```
100 add  $3,$3,$3
104 add  $2,$0,$0
108 li   $5,100
112 addi $5,$5,-1
116 dbne $5,$0,-2
120 addi $4,$4,-1
124 addi $2,$2,1
128 noop
132 addi $3,$3,-1
136 add  $5,$2,$0
140 add  $2,$2,$2
144 add  $2,$2,$5
148 add  $4,$4,$2
```

```
100 add  $3,$3,$3
104 add  $2,$0,$0
108 li   $5,100
112 dbne $5,$0,-1
116 addi $5,$5,-1
120 addi $4,$4,-1
124 addi $2,$2,1
128 addi $3,$3,-1
132 add  $5,$2,$0
136 add  $2,$2,$2
140 add  $2,$2,$5
144 add  $4,$4,$2
```

This code generates wrong results.

Problems with this solution

- Old programs (legacy code) may not run correctly on new implementations
  - Longer pipelines need more instuctions/noops after delayed beq
- Programs get larger as noops are included
  - Especially a problem for machines that try to execute more than one instruction every cycle
  - Intel EPIC: Often 25% - 40% of instructions are noops
- Program execution is slower
  - CPI equals 1, but some instructions are noops

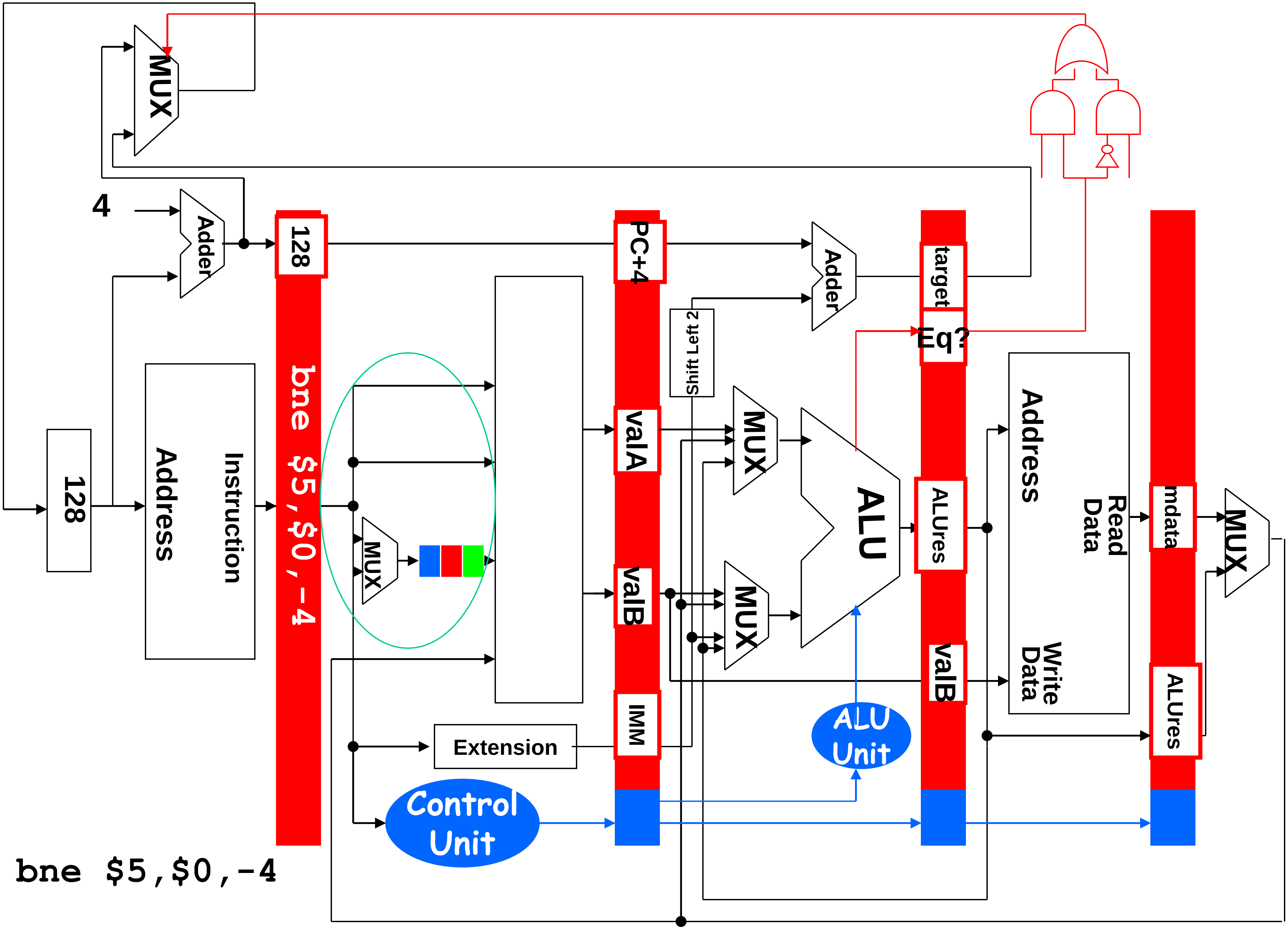
Detect and Stall (hardware approach)

- Detection:
  - Must wait until decode
  - Compare opcode to beq
  - Alternately, this is just another control signal

- Stall:
  - Keep current instructions in fetch
  - Pass noop to **decode** stage (not execute!)

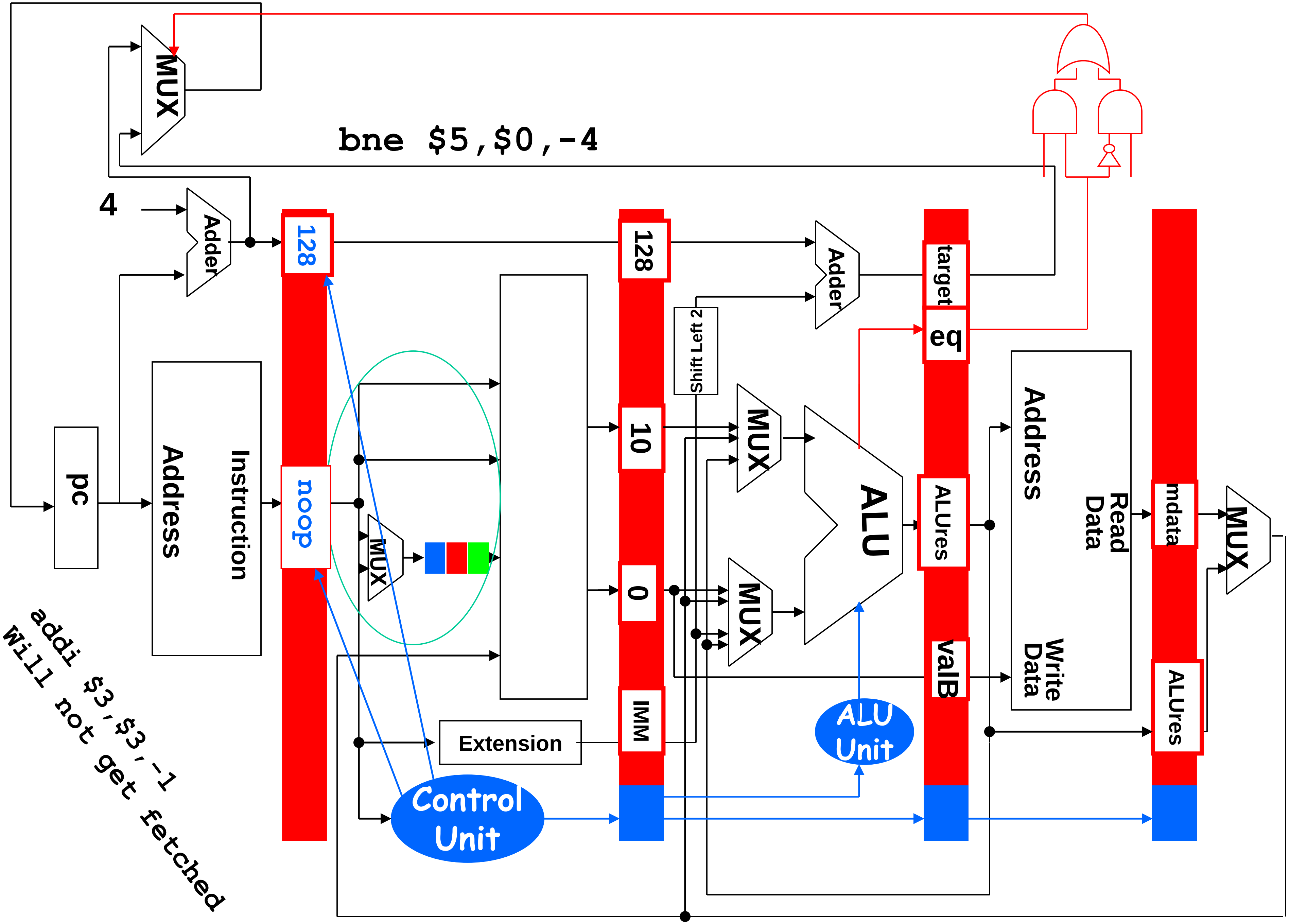
Our example again

```
100 add  $3,$3,$3
104 add  $2,$0,$0
108 li   $5,100
112 addi $2,$2,1
116 addi $4,$4,-1
120 addi $5,$5,-1
124 bne  $5,$0,-4
128 addi $3,$3,-1
132 add  $5,$2,$0
136 add  $2,$2,$2
140 add  $2,$2,$5
144 add  $4,$4,$2
```

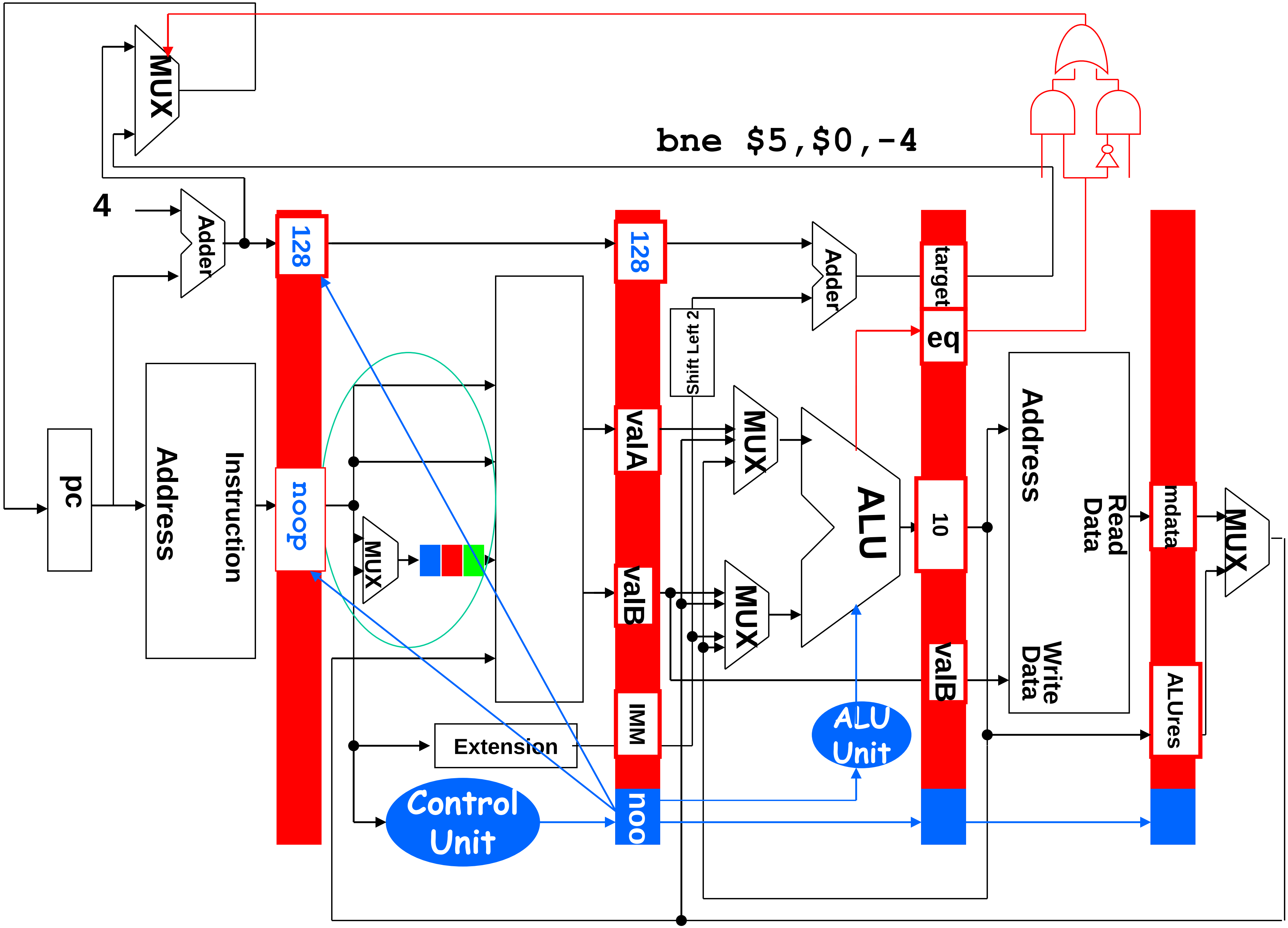

MUX
4
Adder
128
128
Address
Instruction
bne $5,$0,-4
MUX
Extension
Control Unit
PC+4
valA
valB
IMM
Shift Left 2
Adder
MUX
MUX
ALU
ALU Unit
target
Eq?
ALUres
valB
Address
Read Data
Write Data
mdata
ALUres
MUX

bne $5,$0,-4

bne $5,$0,-4

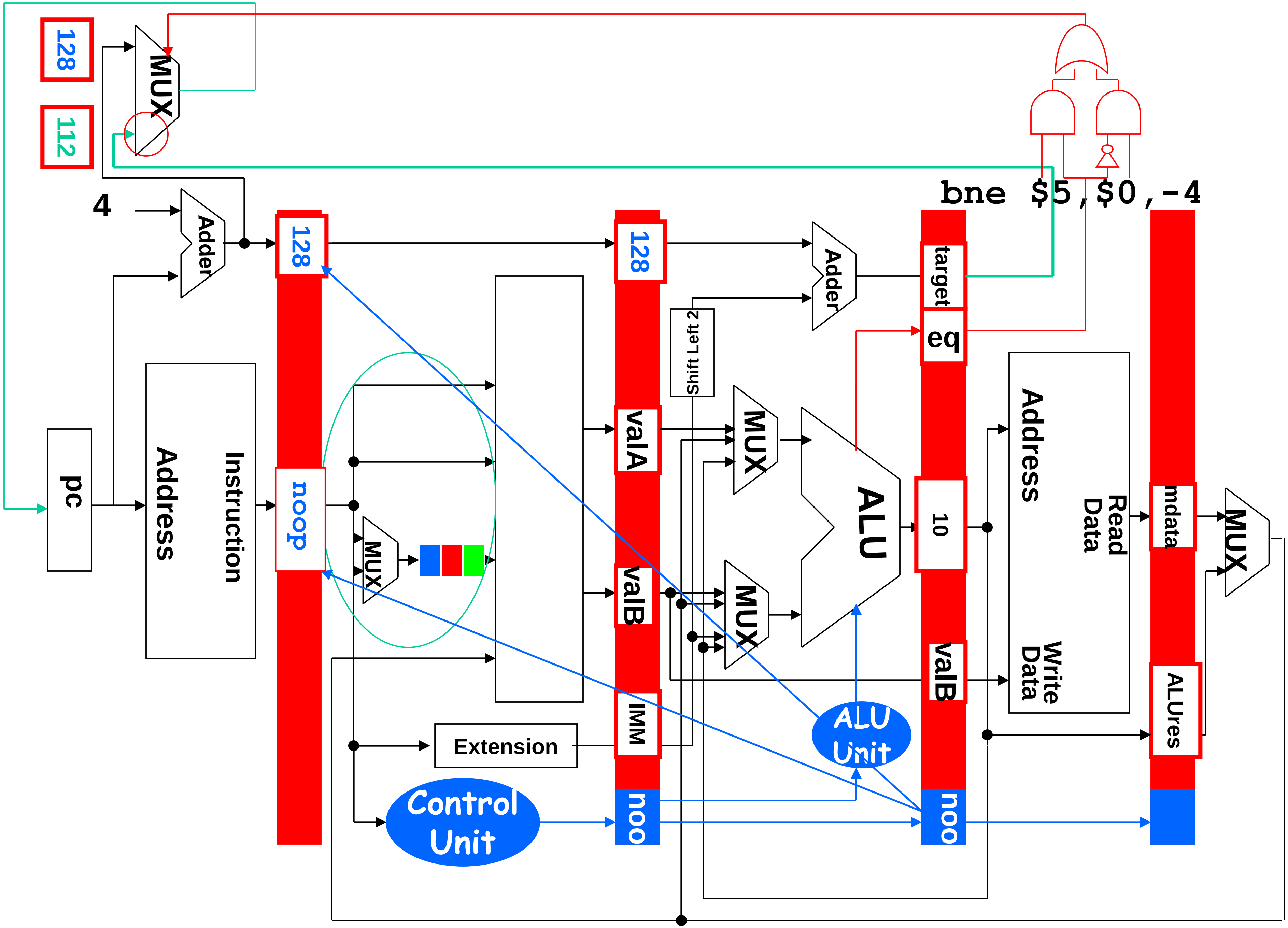


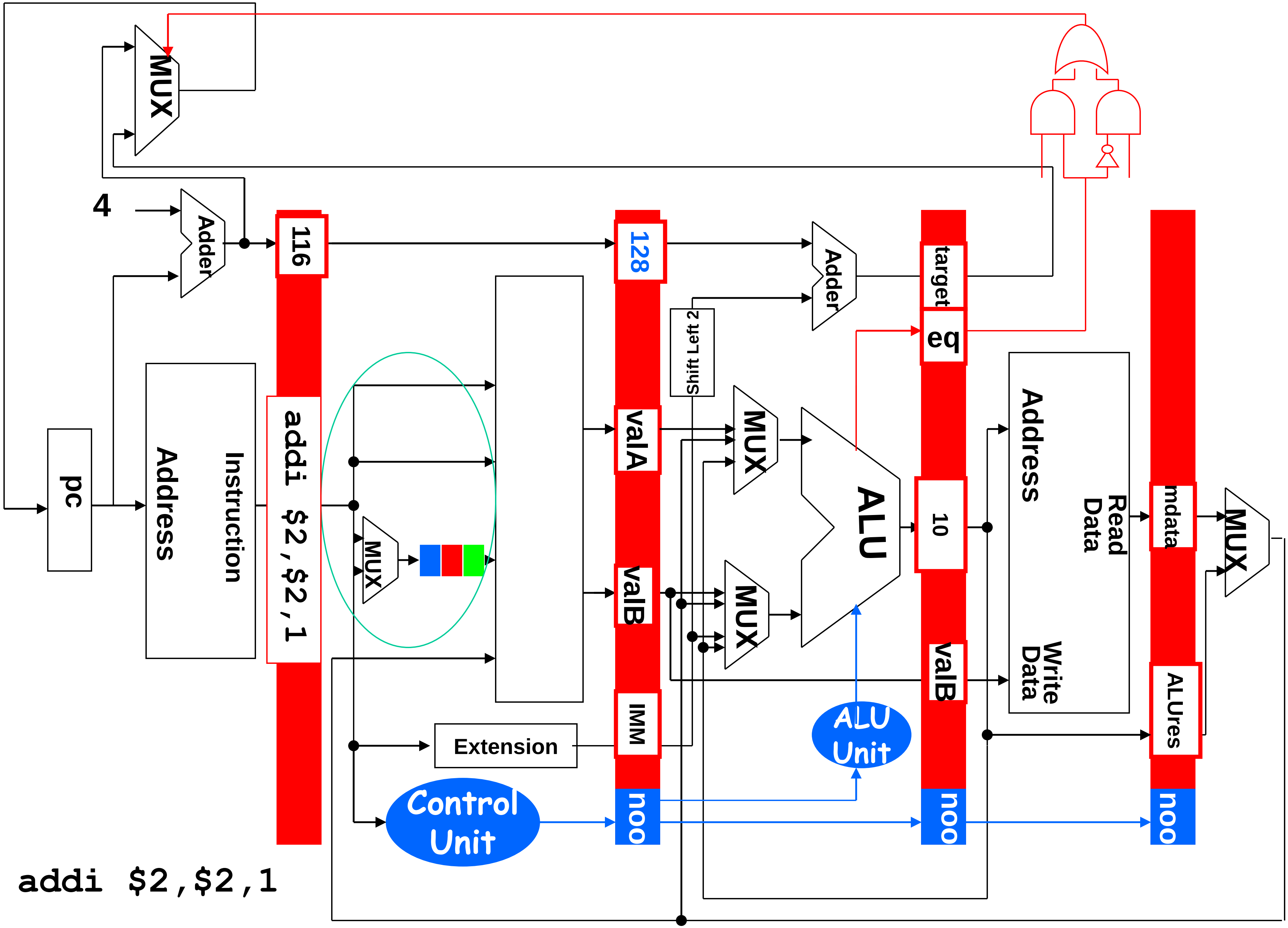
addi $3,$3,-1
Will not get fetched


MUX
4
Adder
pc
Address
Instruction
128
noop
MUX
Extension
Control Unit
128
valA
valB
IMM
noo
Shift Left 2
MUX
MUX
ALU
Adder
target
eq
10
valB
ALU Unit
Address
Read Data
Write Data
mdata
ALUres
MUX

bne $5,$0,-4

bne $5,$0,-4


128
112
MUX
4
Adder
pc
Address
Instruction
128
noop
MUX
128
valA
valB
IMM
noo
Extension
Control Unit
Shift Left 2
Adder
MUX
MUX
ALU
target
eq
10
valB
noo
ALU Unit
Address
Read Data
Write Data
mdata
ALUres
MUX


MUX
4
Adder
116
128
Adder
target
eq
Shift Left 2
pc
Address
Instruction
addi $2,$2,1
MUX
valA
valB
MUX
MUX
ALU
10
Address
Read Data
Write Data
mdata
MUX
valB
ALUres
Extension
IMM
ALU Unit
Control Unit
noo
noo
noo

addi $2,$2,1

What seems to be the problem?

- CPI increases every time a branch is detected!

- Is that necessary? Not always!
    - Only about ½ of the time is the branch taken
        - Let’s assume that it is NOT taken…
            - In this case, we can ignore the beq or bne (treat them like a noop)
            - Keep fetching PC + 4
        - What if we are wrong?
            - OK, as long as we do not COMPLETE any instructions we mistakenly executed (i.e. don’t perform writeback)

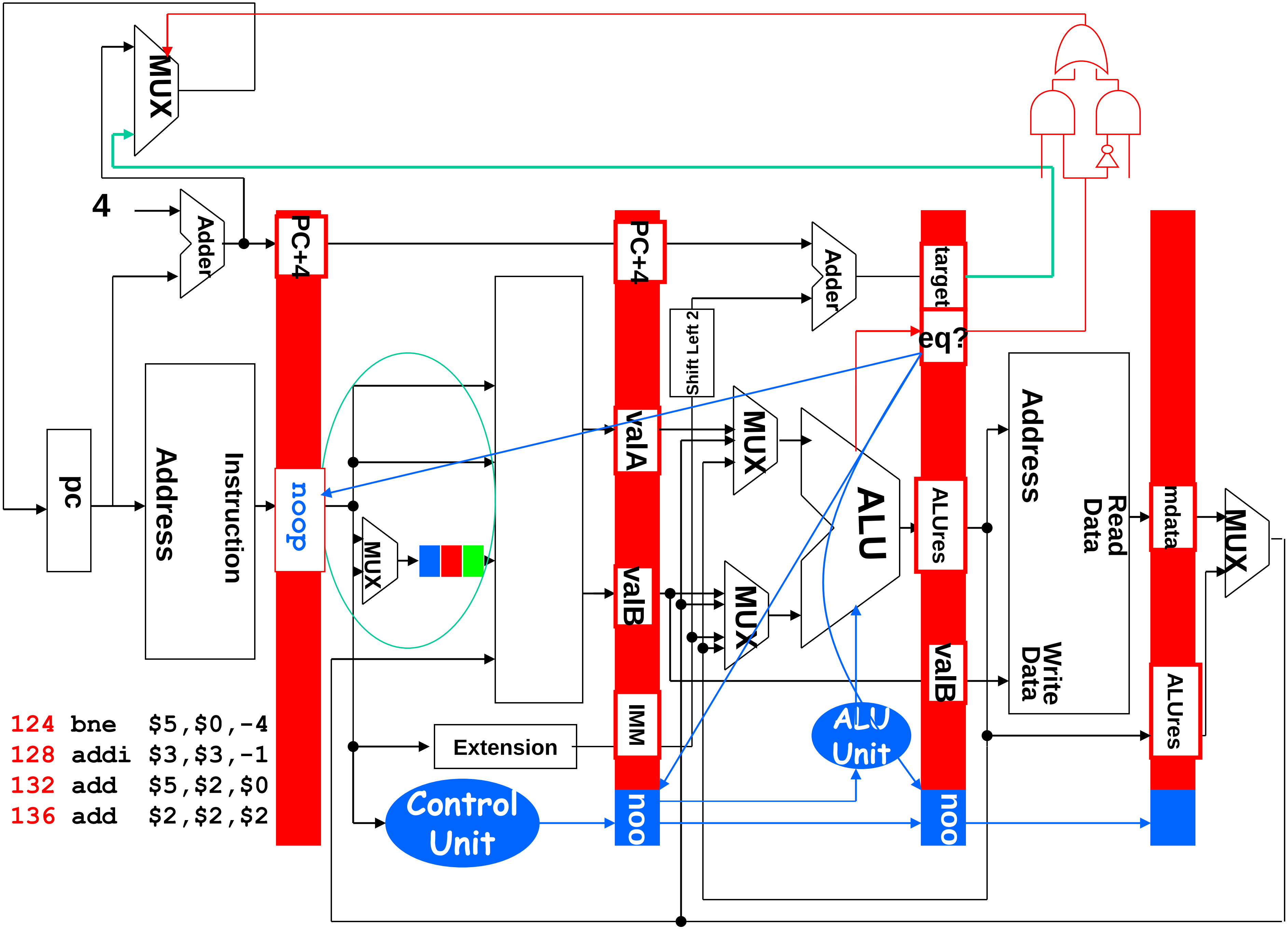
Speculate and Squash

- Speculate: assume not equal
  - Keep fetching from PC+4 until we know that the branch is really taken

- Squash: stop bad instructions if taken
  - Send a noop to:
    - Decode, Execute and Memory
  - Send target address to PC

Our example again

```
100 add  $3,$3,$3
104 add  $2,$0,$0
108 li   $5,100
112 addi $2,$2,1
116 addi $4,$4,-1
120 addi $5,$5,-1
124 bne  $5,$0,-4
128 addi $3,$3,-1
132 add  $5,$2,$0
136 add  $2,$2,$2
140 add  $2,$2,$5
144 add  $4,$4,$2
```



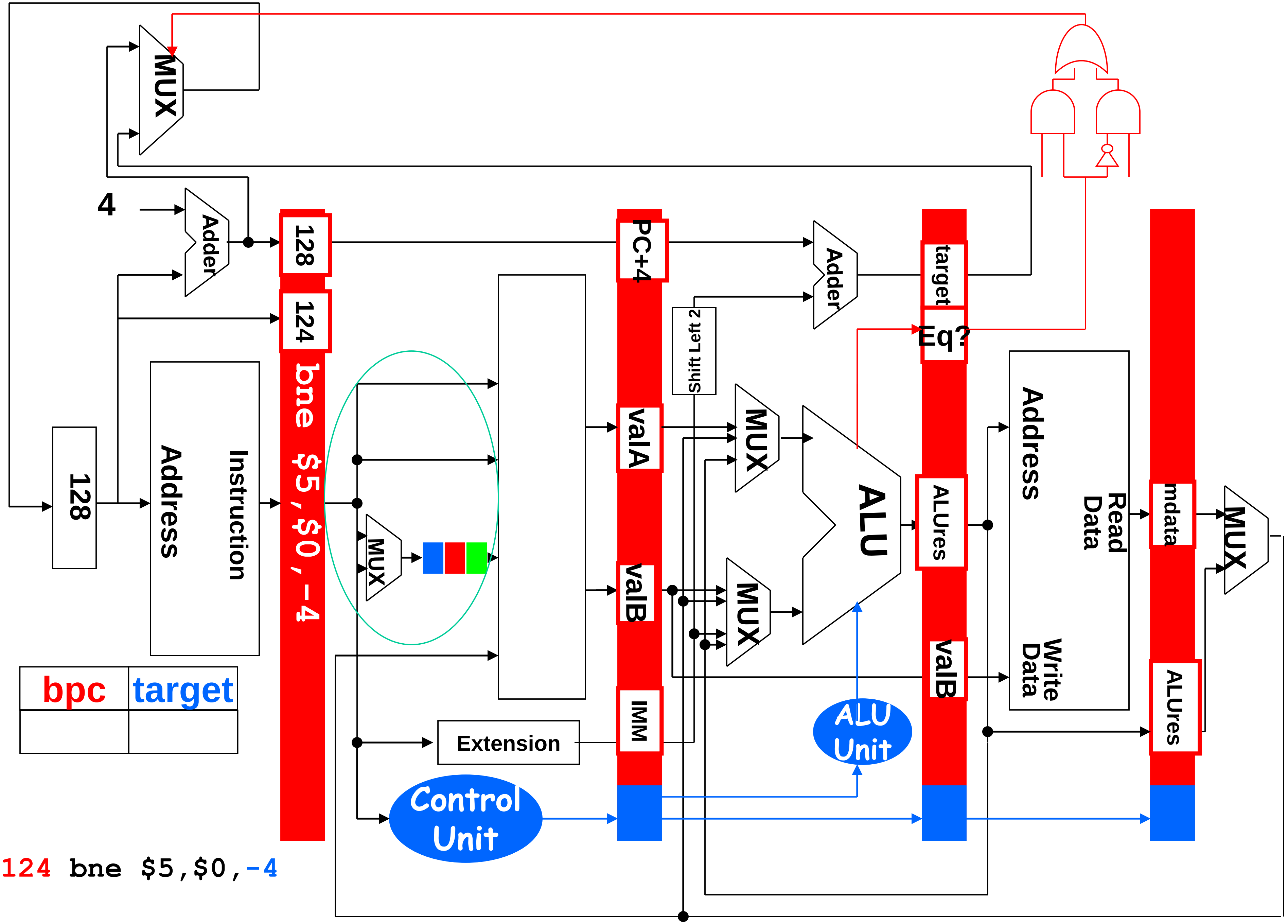
```
124 bne   $5,$0,-4
128 addi  $3,$3,-1
132 add   $5,$2,$0
136 add   $2,$2,$2
```

Performance problem, again

- CPI increases every time a branch is taken!
  - About ½ of the time

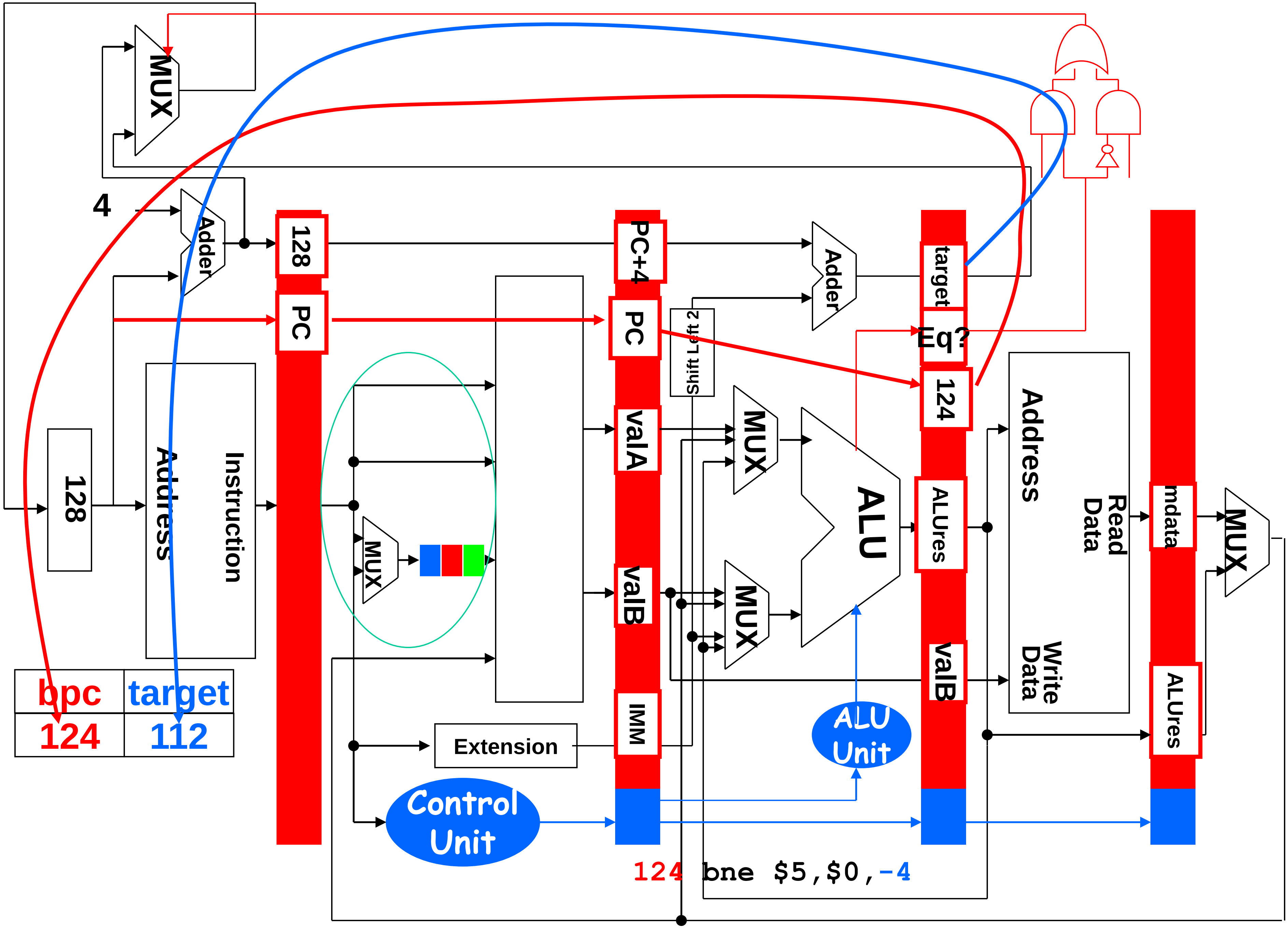
- Is that necessary?

No!, but how can you fetch from the target before you even know the previous instruction is a branch – much less whether it is taken???



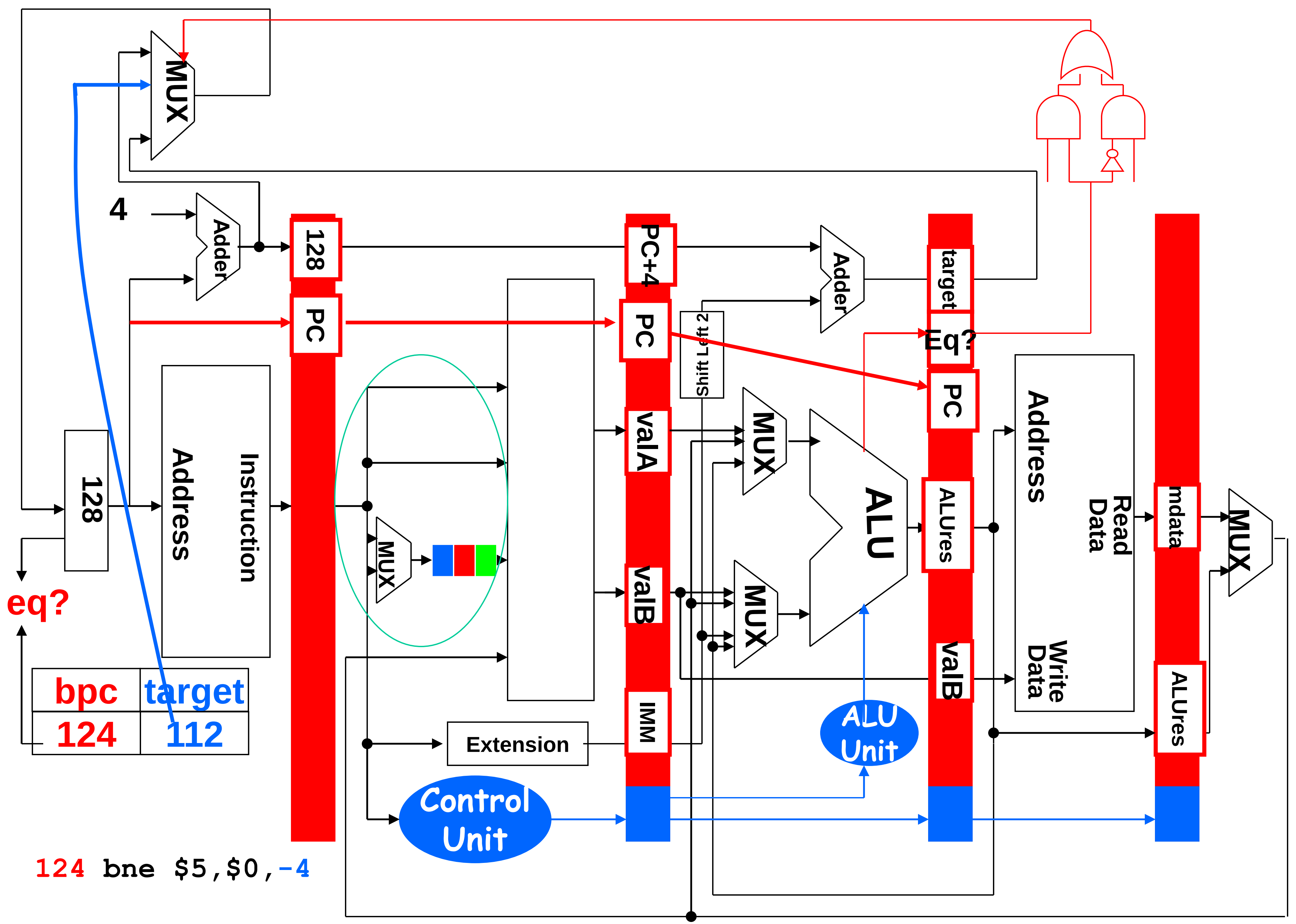
bpc	target

124 bne $5,$0,-4



bpc	target
124	112

```
124 bne $5,$0,-4
```



bpc	target
124	112

124 bne $5,$0,-4

Branch Prediction

• Predict not taken:	~50% accurate
• Predict backward taken:	~65% accurate
• Predict same as last time:	~80% accurate
• Pentium:	~85% accurate
• Pentium Pro:	~92% accurate
• Best paper designs:	~96% accurate