

Pipelined Datapath

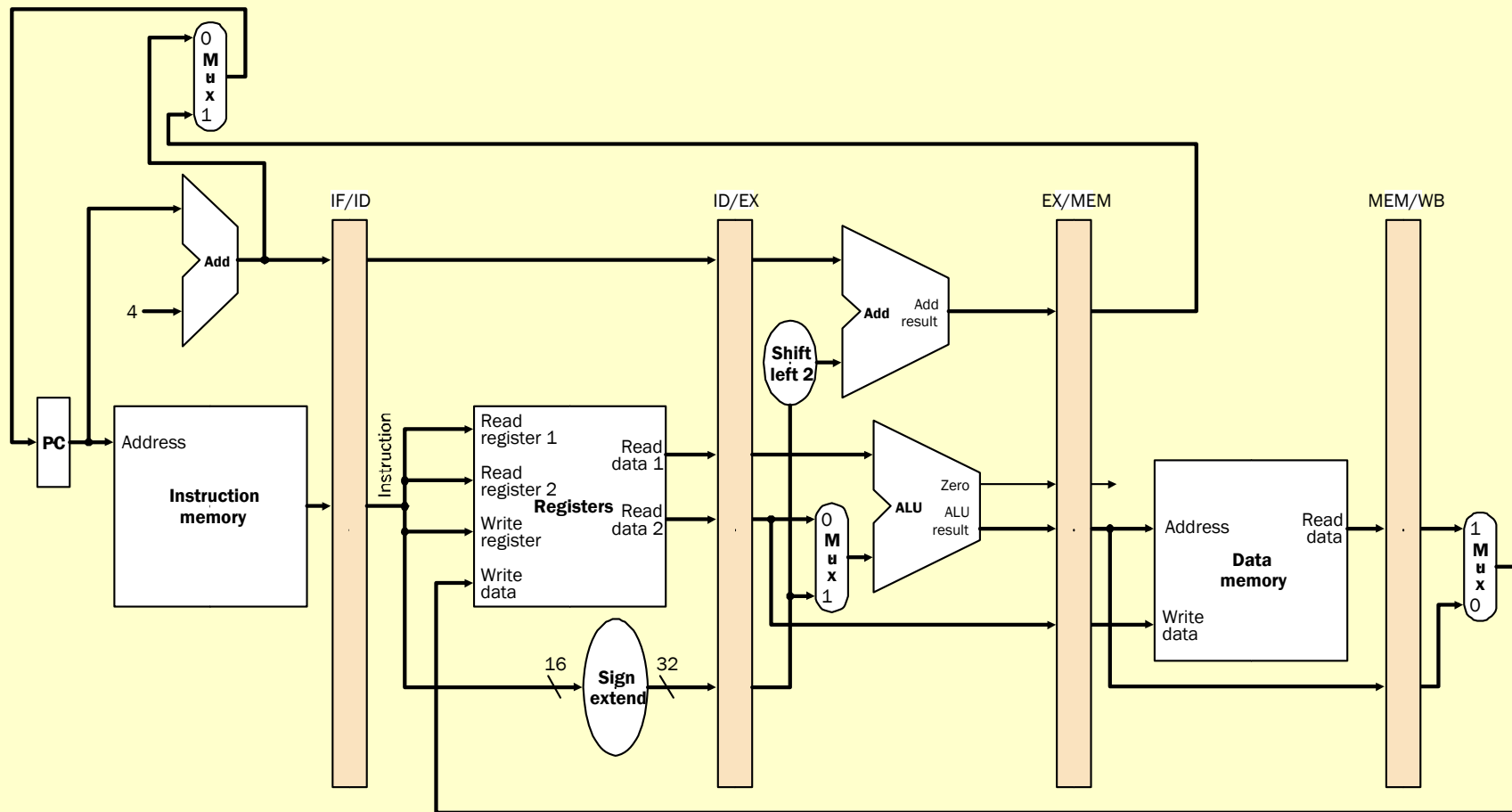
Instruction Fetch

Instruction Decode/
Register Fetch

Execute/
Address Calculation

Memory Access

Write Back



The Pipeline in Execution

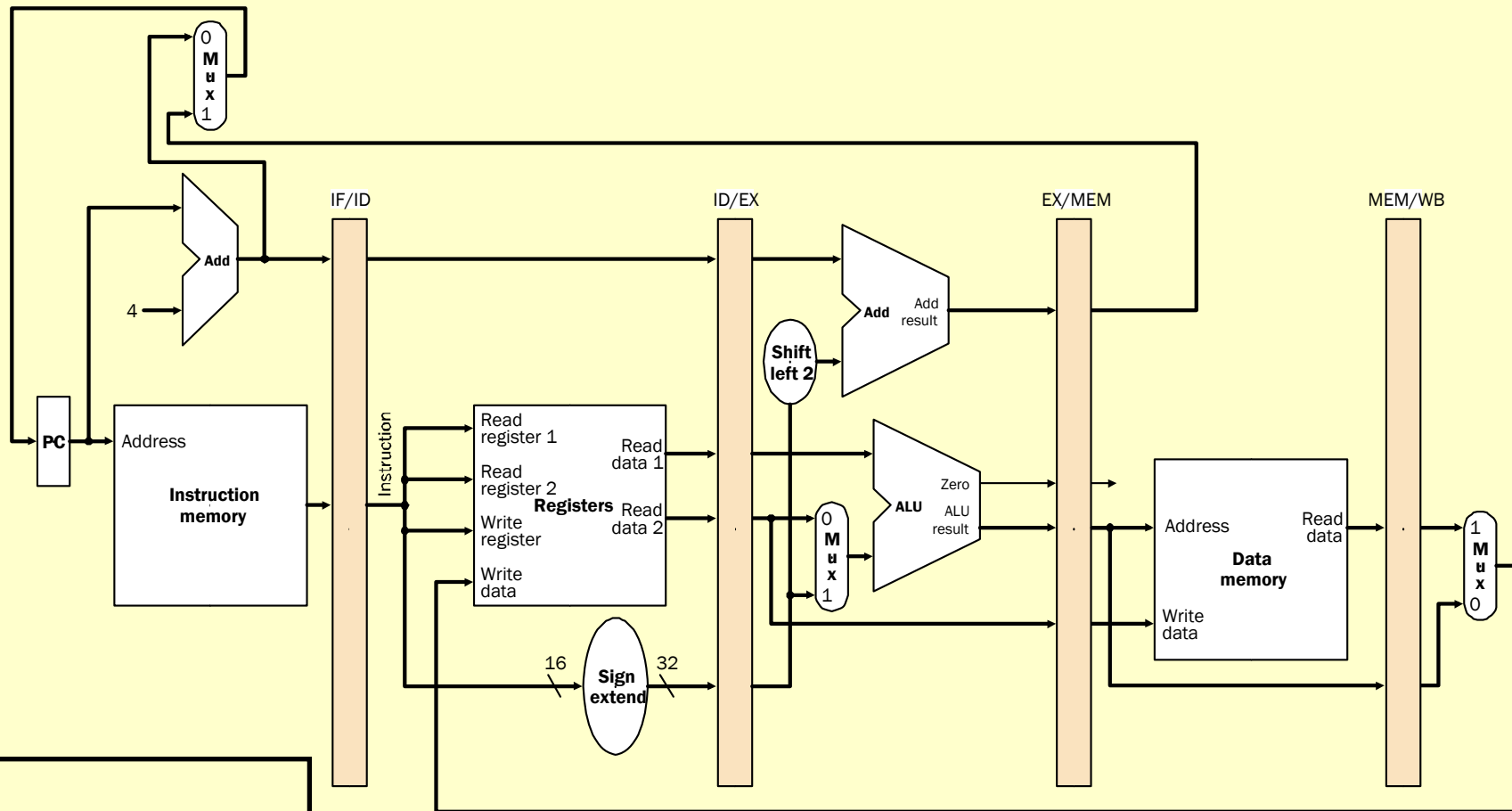
add \$10, \$1, \$2

Instruction Decode/
Register Fetch

Execute/
Address Calculation

Memory Access

Write Back



add \$10, \$1, \$2
lw \$12, 1000(\$4)
sub \$15, \$4, \$1

The Pipeline in Execution

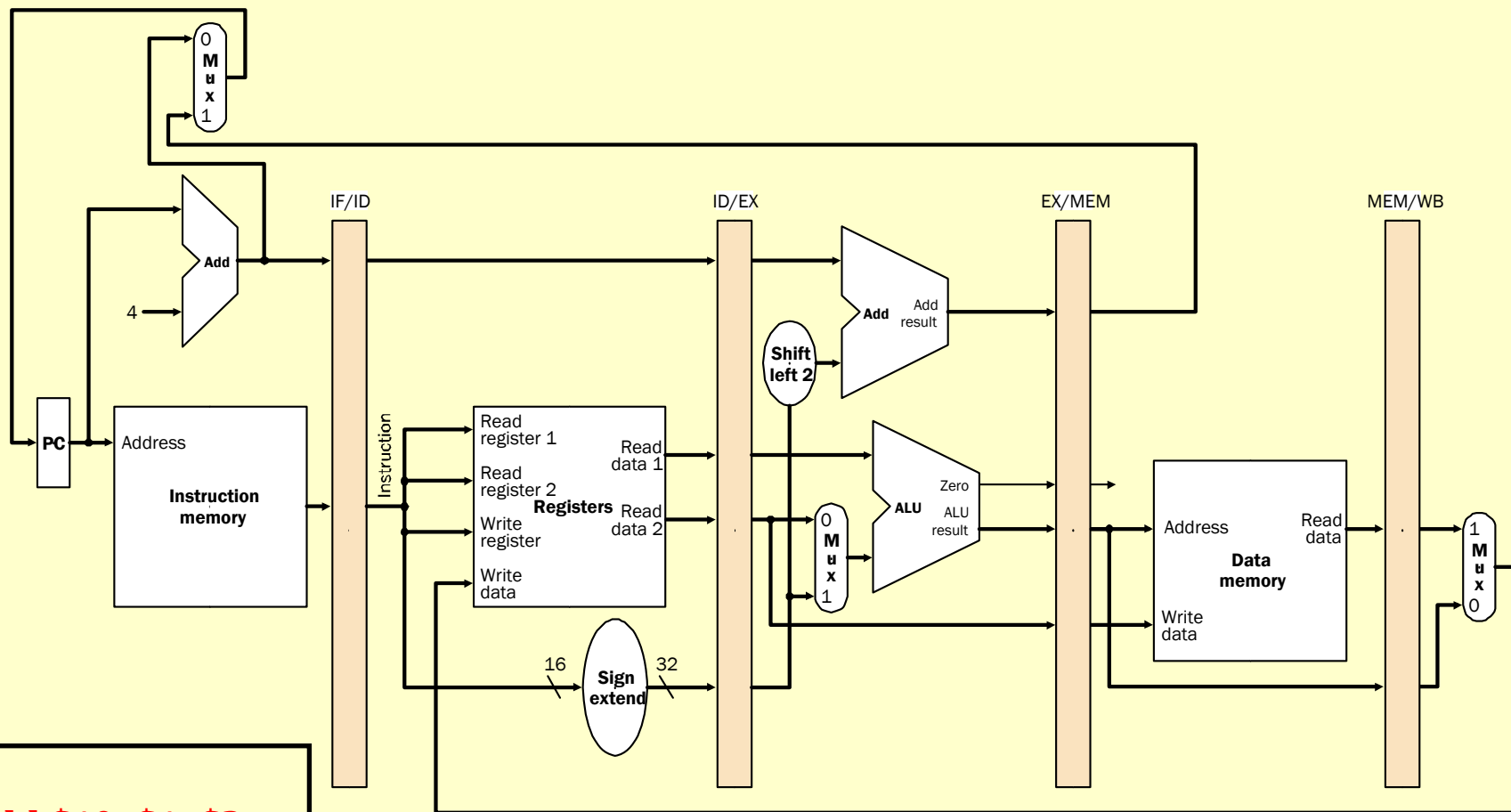
lw \$12, 1000(\$4)

add \$10, \$1, \$2

Execute/
Address Calculation

Memory Access

Write Back



add \$10, \$1, \$2
lw \$12, 1000(\$4)
sub \$15, \$4, \$1

The Pipeline in Execution

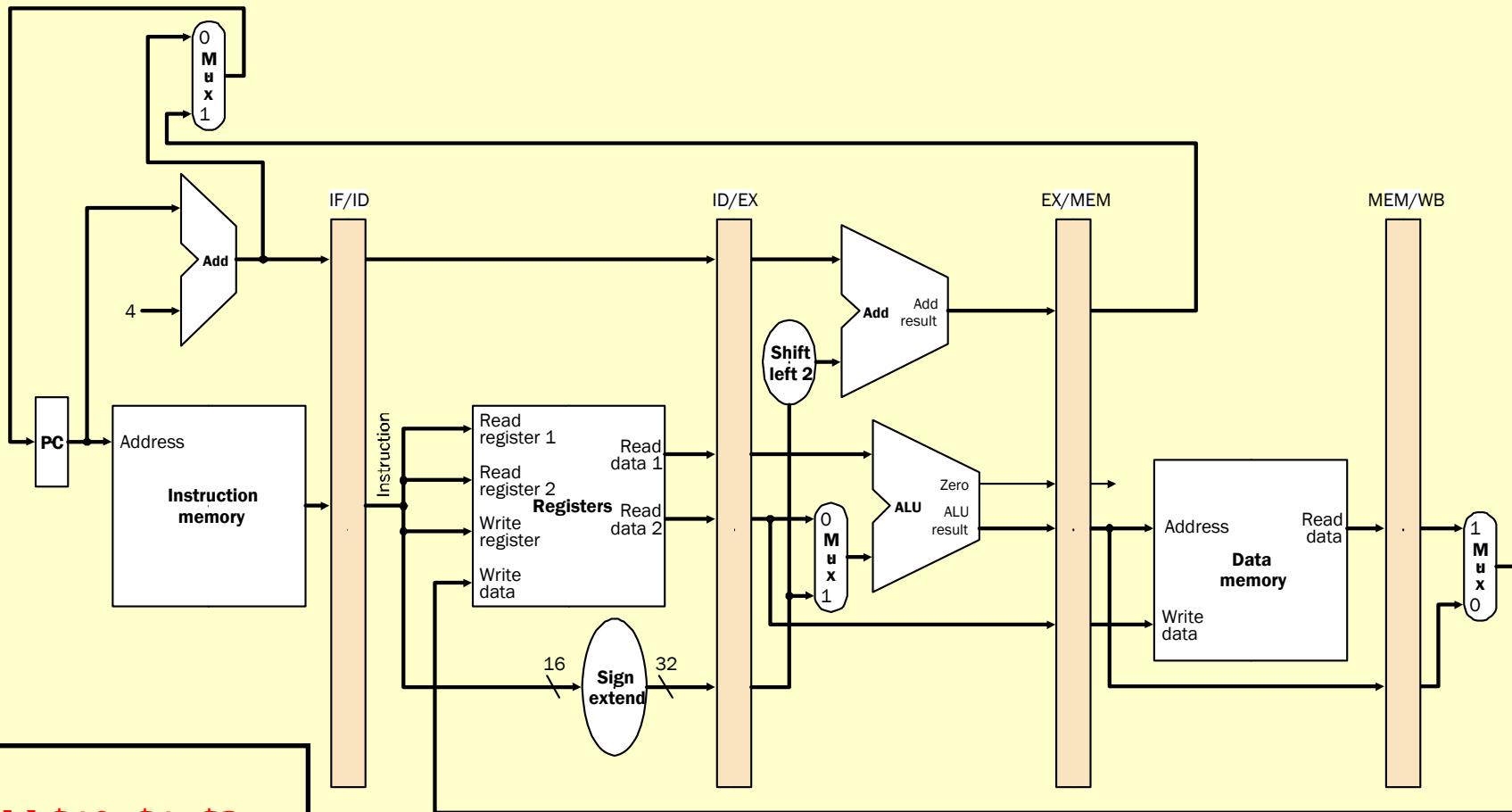
sub \$15, \$4, \$1

lw \$12, 1000(\$4)

add \$10, \$1, \$2

Memory Access

Write Back



add \$10, \$1, \$2
lw \$12, 1000(\$4)
sub \$15, \$4, \$1

The Pipeline in Execution

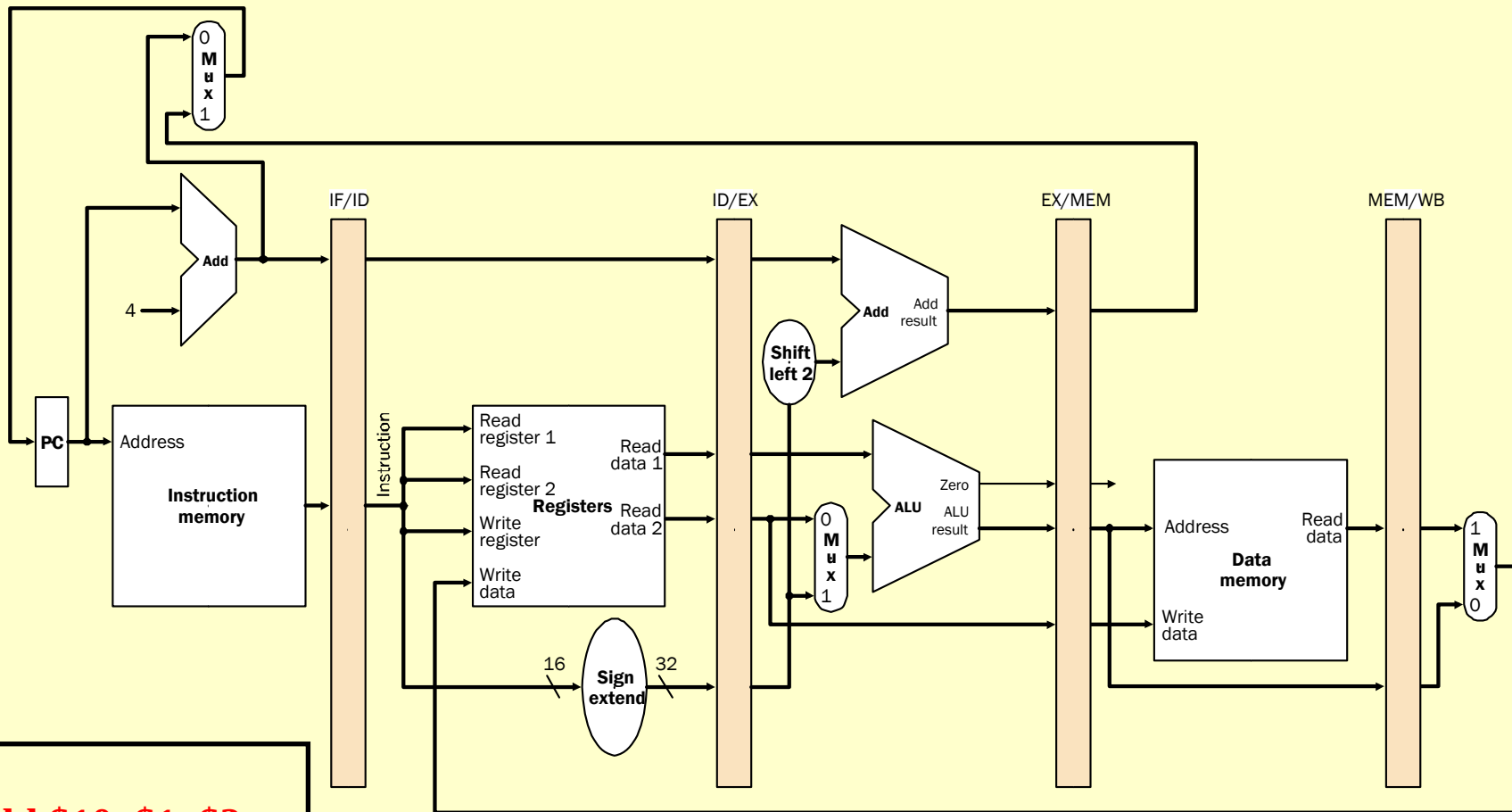
Instruction Fetch

sub \$15, \$4, \$1

lw \$12, 1000(\$4)

add \$10, \$1, \$2

Write Back



add \$10, \$1, \$2
lw \$12, 1000(\$4)
sub \$15, \$4, \$1

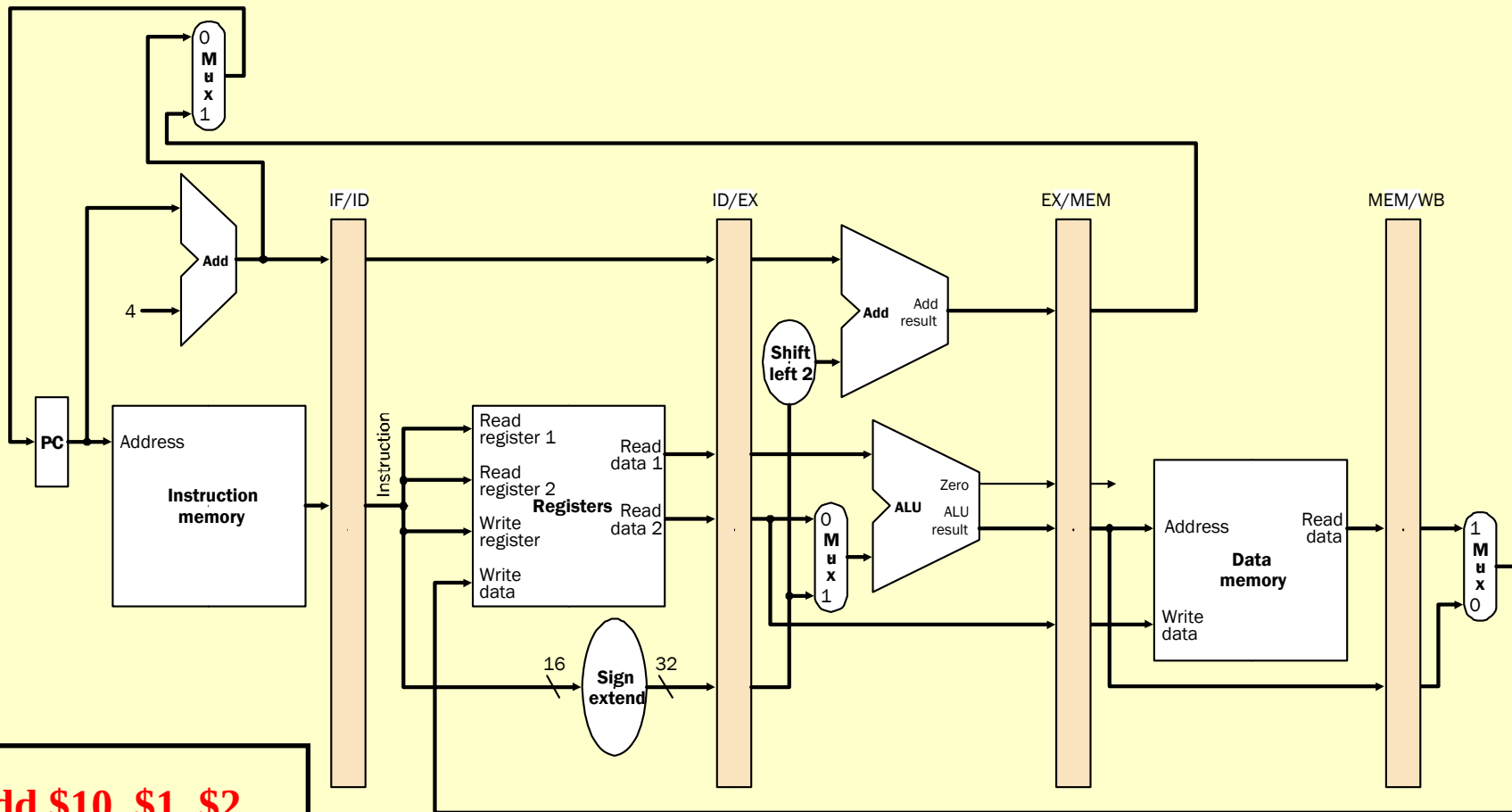
The Pipeline in Execution

Instruction Fetch

Instruction Decode/
Register Fetch

sub \$15, \$4, \$1

lw \$12, 1000(\$4) add \$10, \$1, \$2



add \$10, \$1, \$2
lw \$12, 1000(\$4)
sub \$15, \$4, \$1
sll \$0, \$0, 0

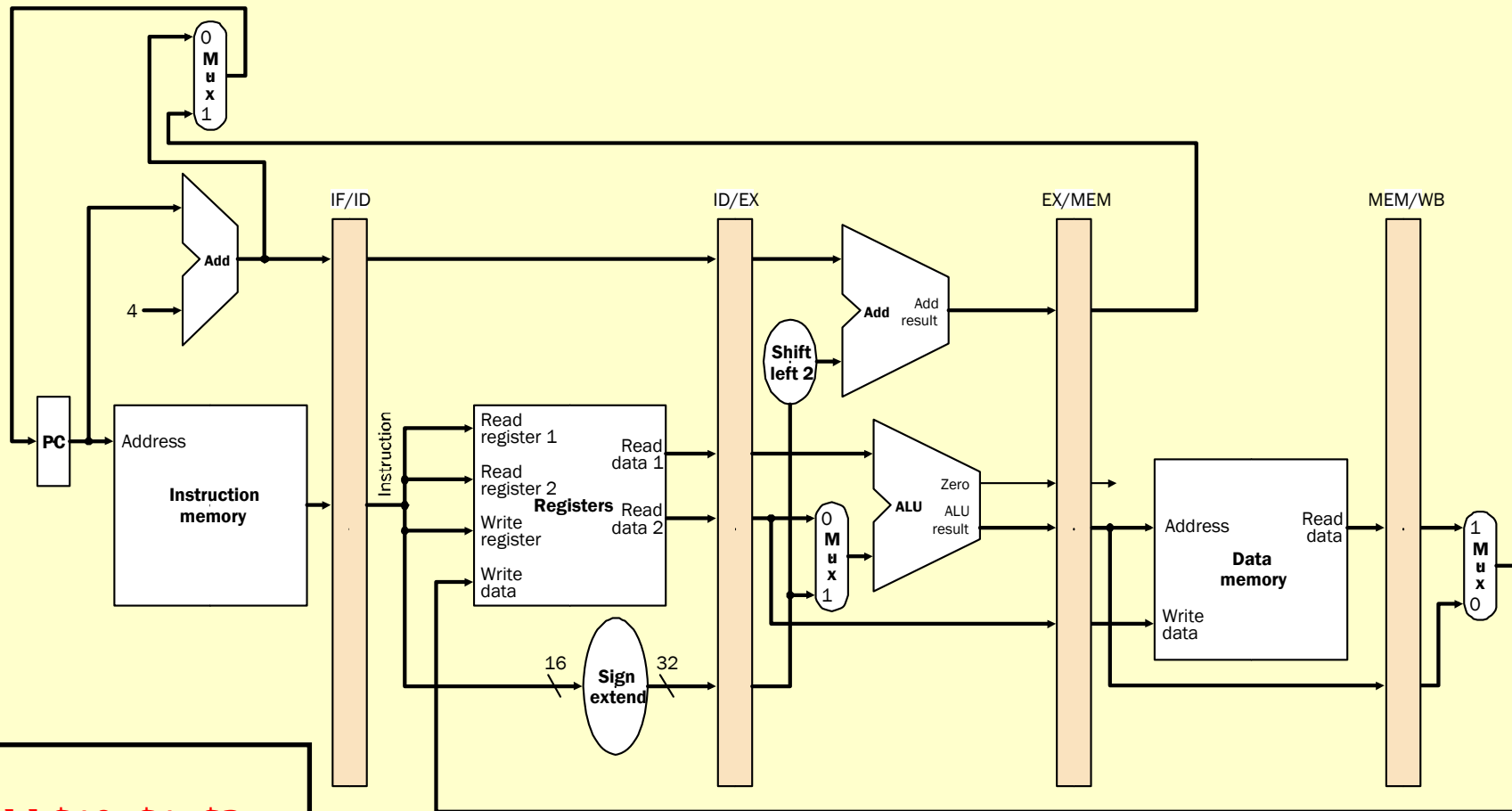
The Pipeline in Execution

Instruction Fetch

Instruction Decode/
Register Fetch

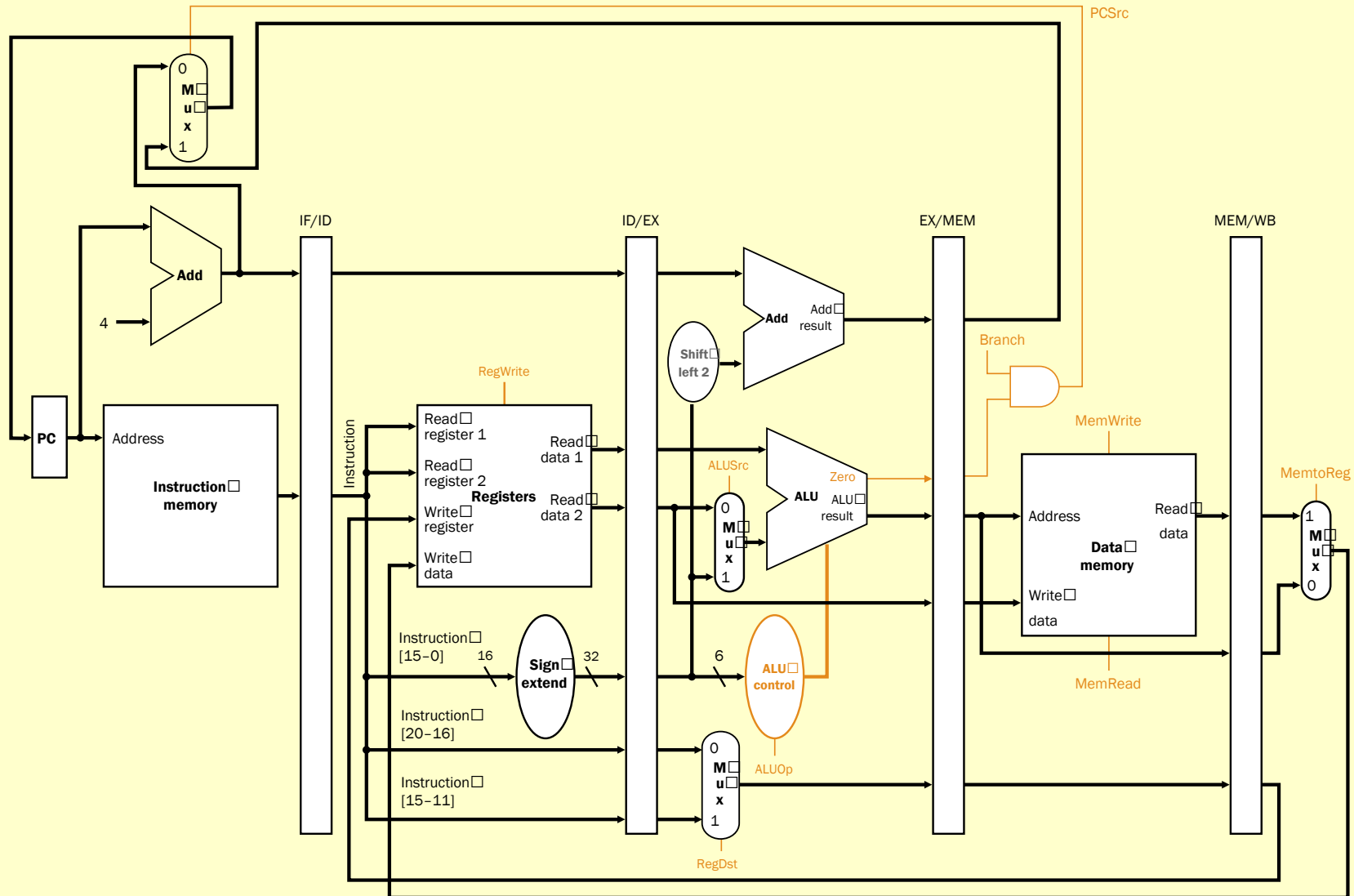
Execute/
Address Calculation

sub \$15, \$4, \$1 lw \$12, 1000(\$4)



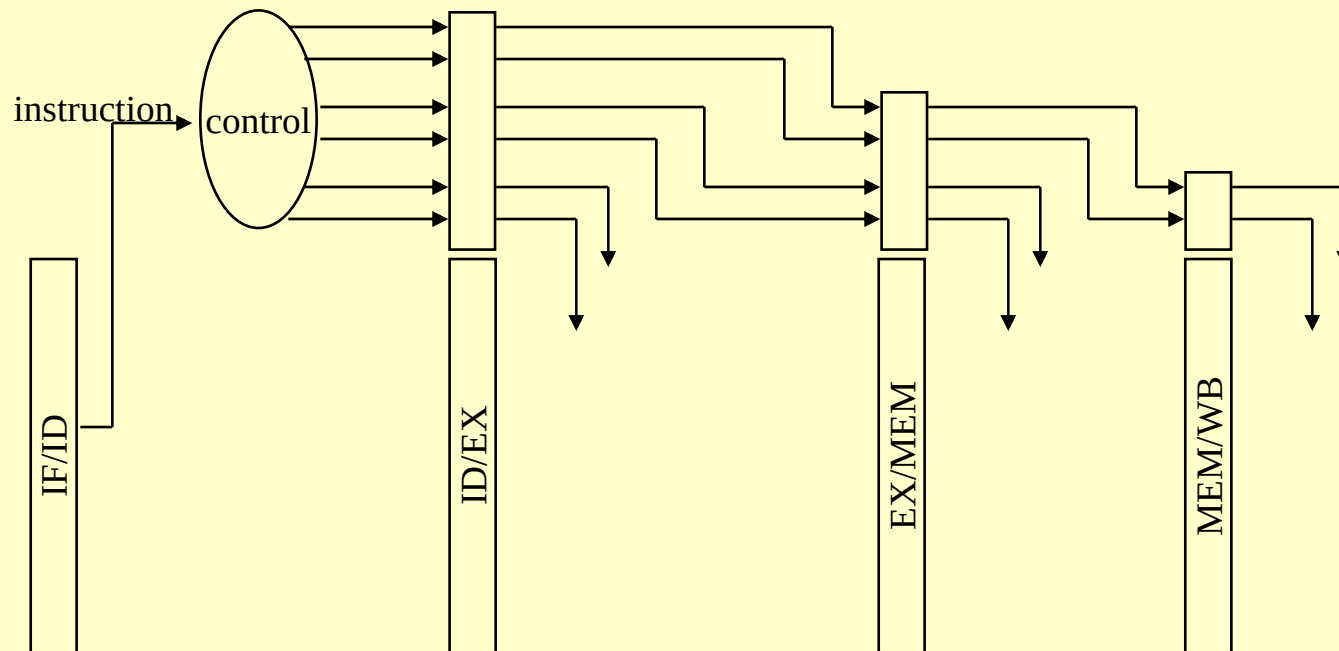
add \$10, \$1, \$2
lw \$12, 1000(\$4)
sub \$15, \$4, \$1

The Pipeline in Execution, with controls



Pipelined Control

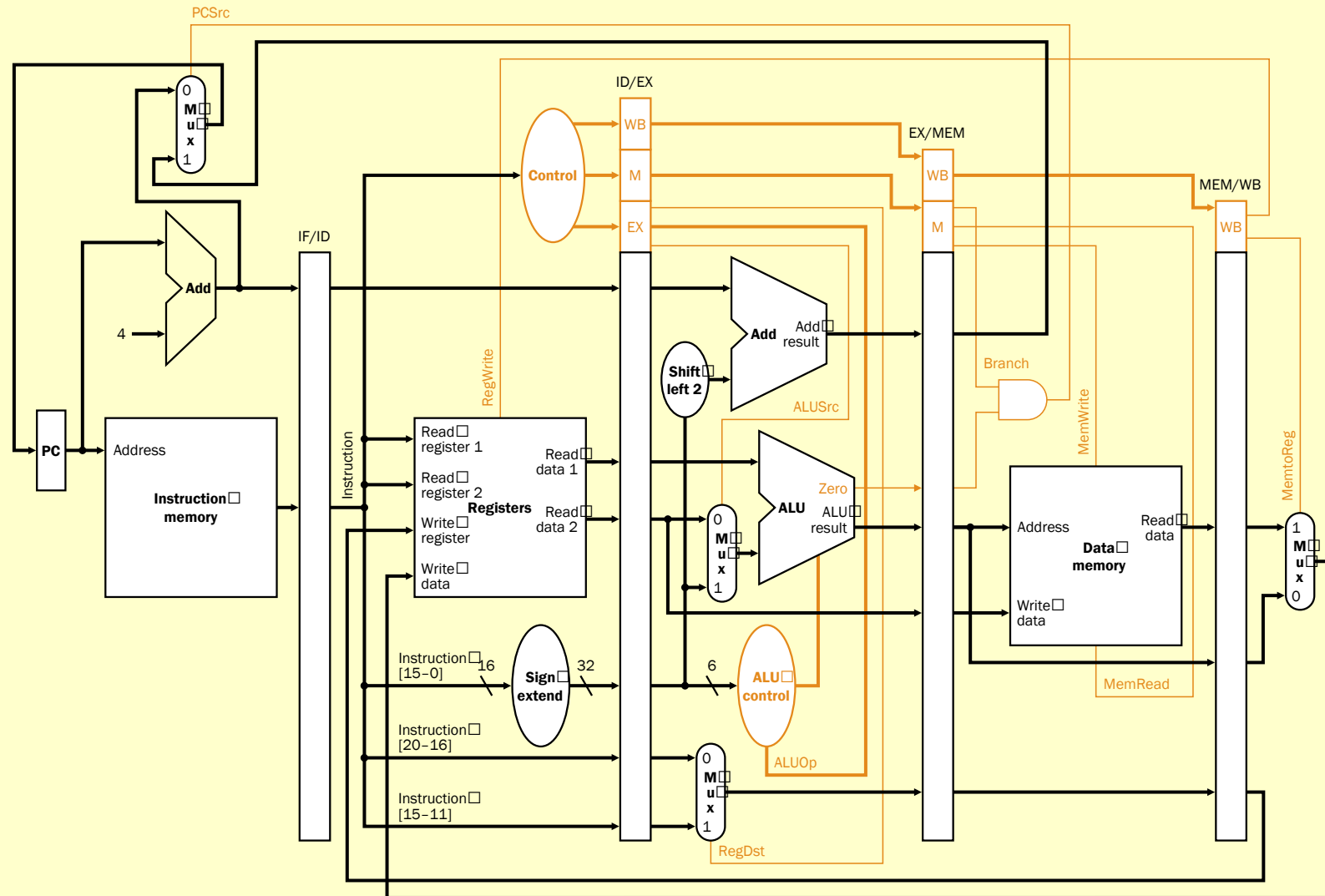
- FSM isn't really appropriate
- Combinational Logic (like single-cycle design)!



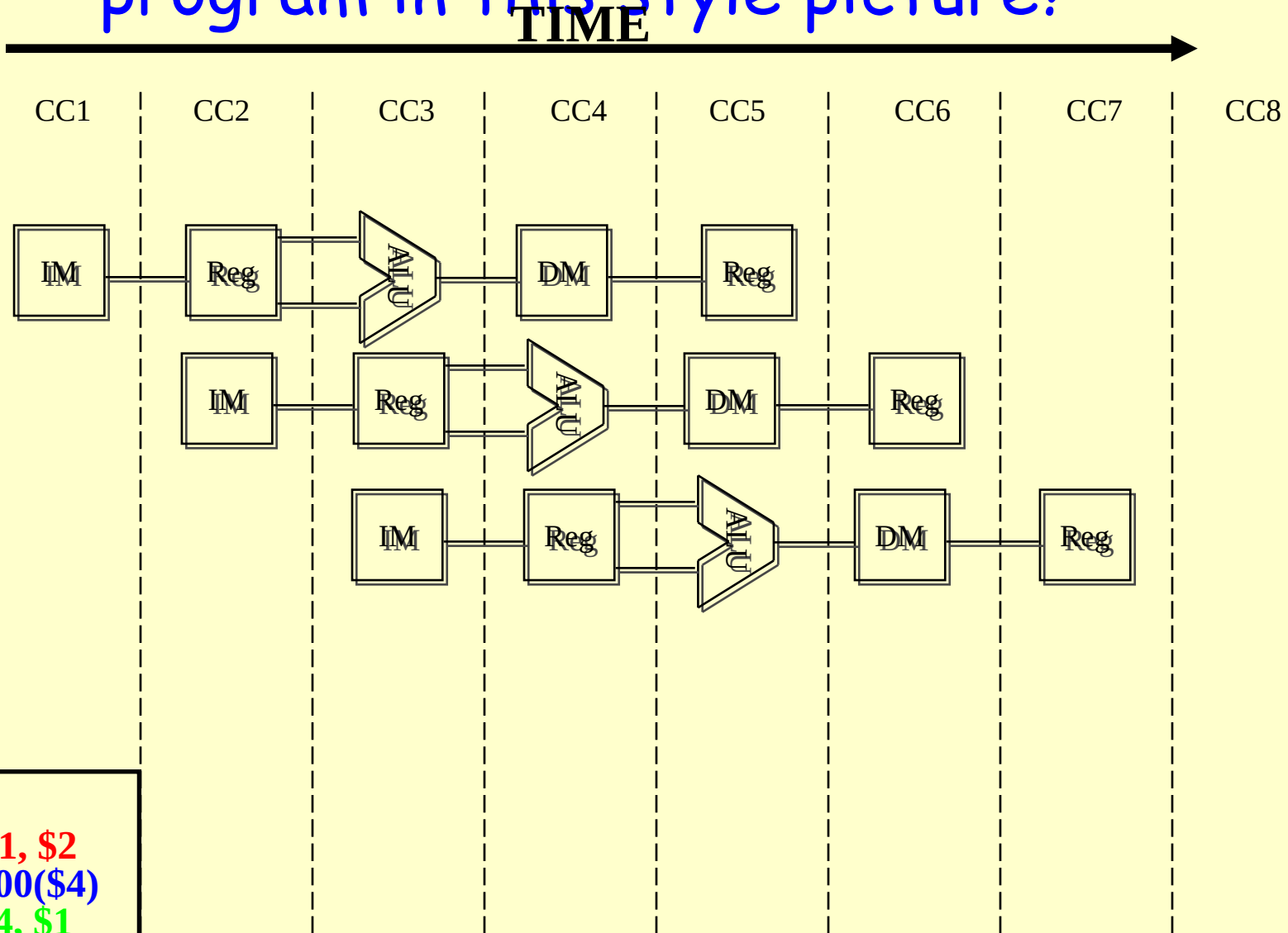
Self Check

- Why do we only need one register file?
It is USED in two different stages of the pipeline...

The Pipeline with Control Logic



Translation: How do we show that program in this style picture?



add \$10, \$1, \$2
lw \$12, 1000(\$4)
sub \$15, \$4, \$1

Is it really that easy?

- Suppose initially, for all registers
 - register i holds the number 2^i
- What happens when we execute this program...

```
add $3, $10, $11  
lw $8, 50($3)  
sub $11, $8, $7
```

The Pipeline in Execution

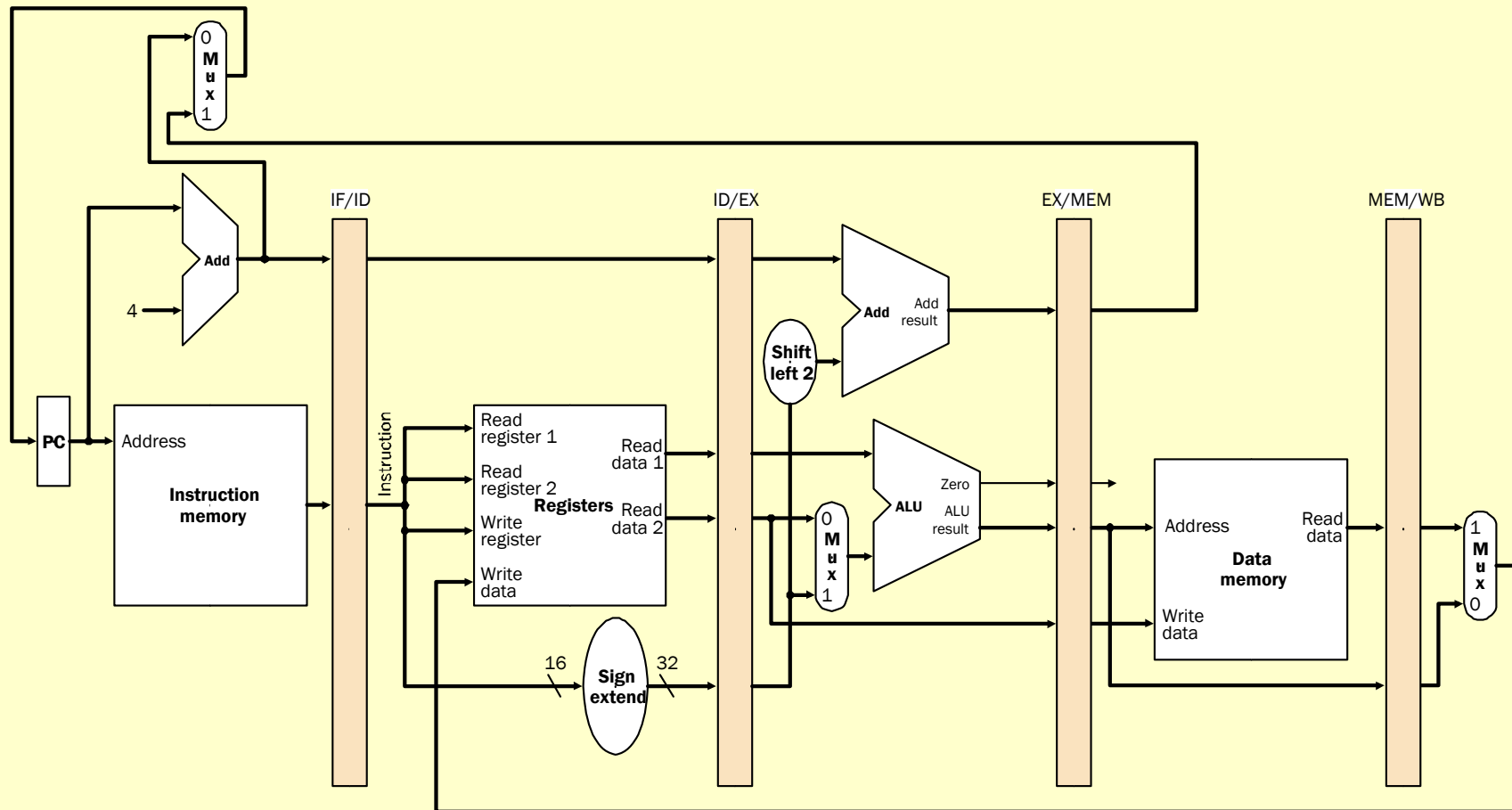
lw \$8, 50(\$3)

add \$3, \$10, \$11

Execute/
Address Calculation

Memory Access

Write Back



The Pipeline in Execution

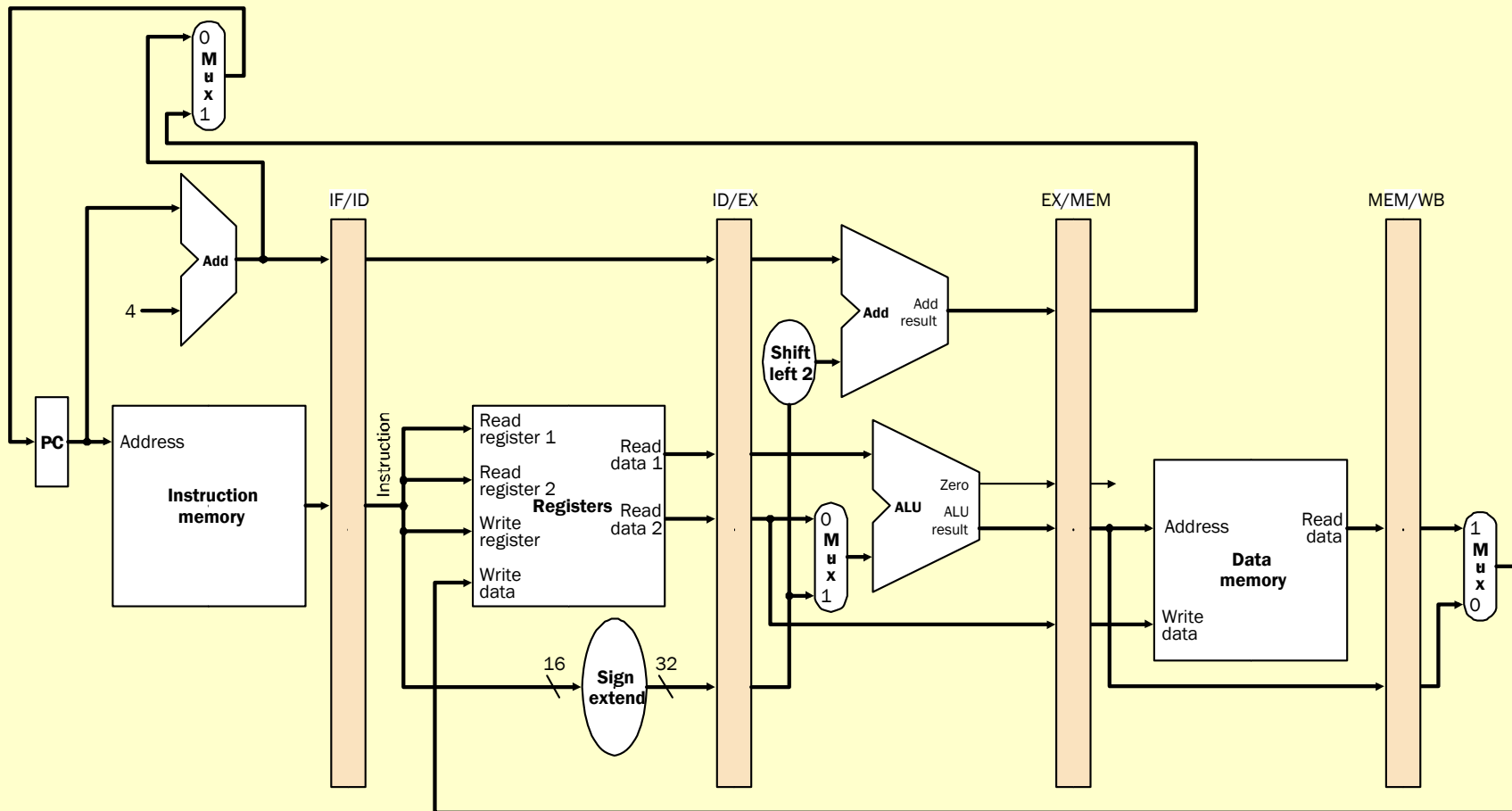
sub \$11, \$8, \$7

lw \$8, 50(\$3)

add \$3, \$10, \$11

Memory Access

Write Back



The Pipeline in Execution

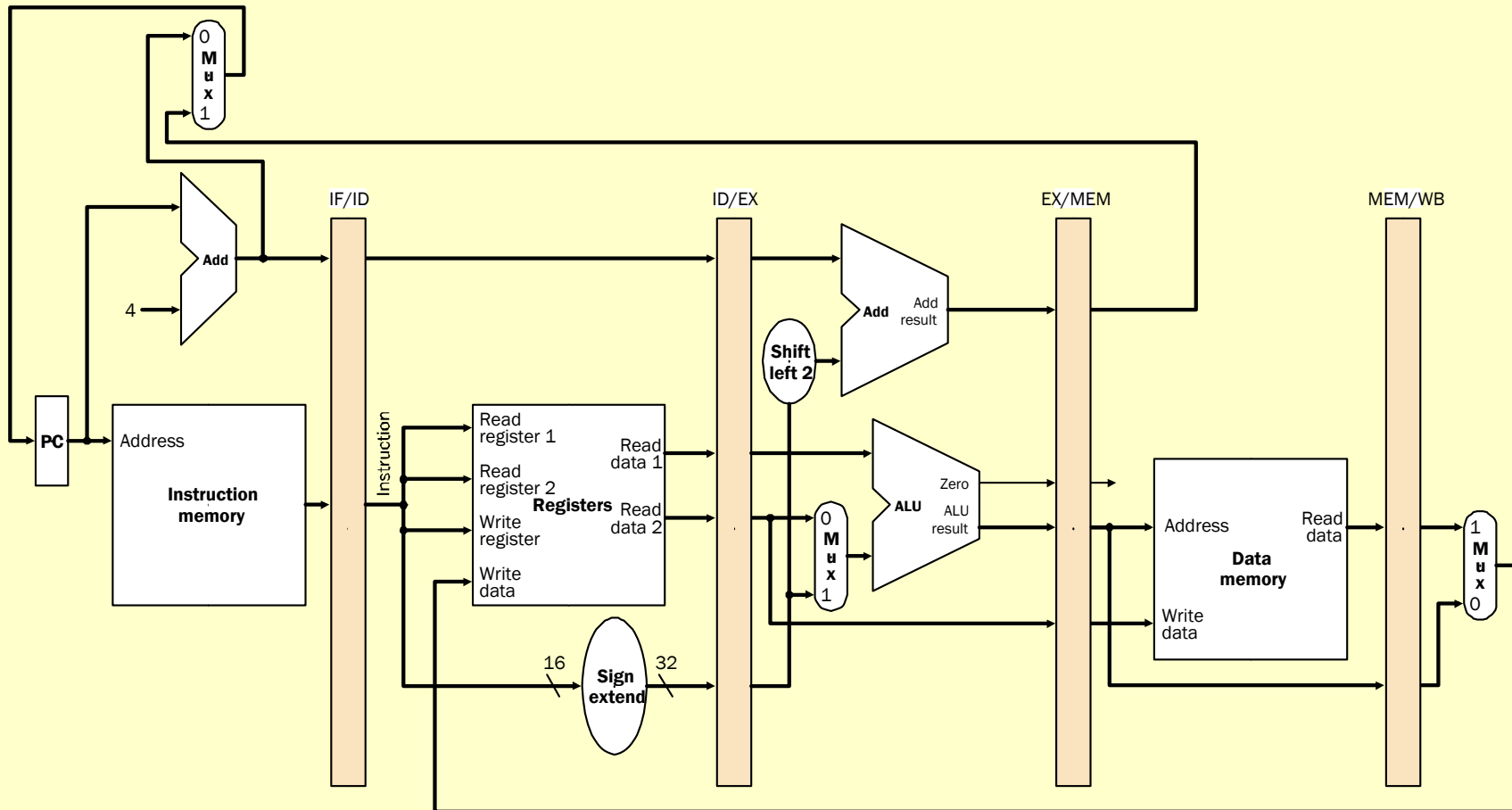
add \$10, \$1, \$2

sub \$11, \$8, \$7

lw \$8, 50(\$3)

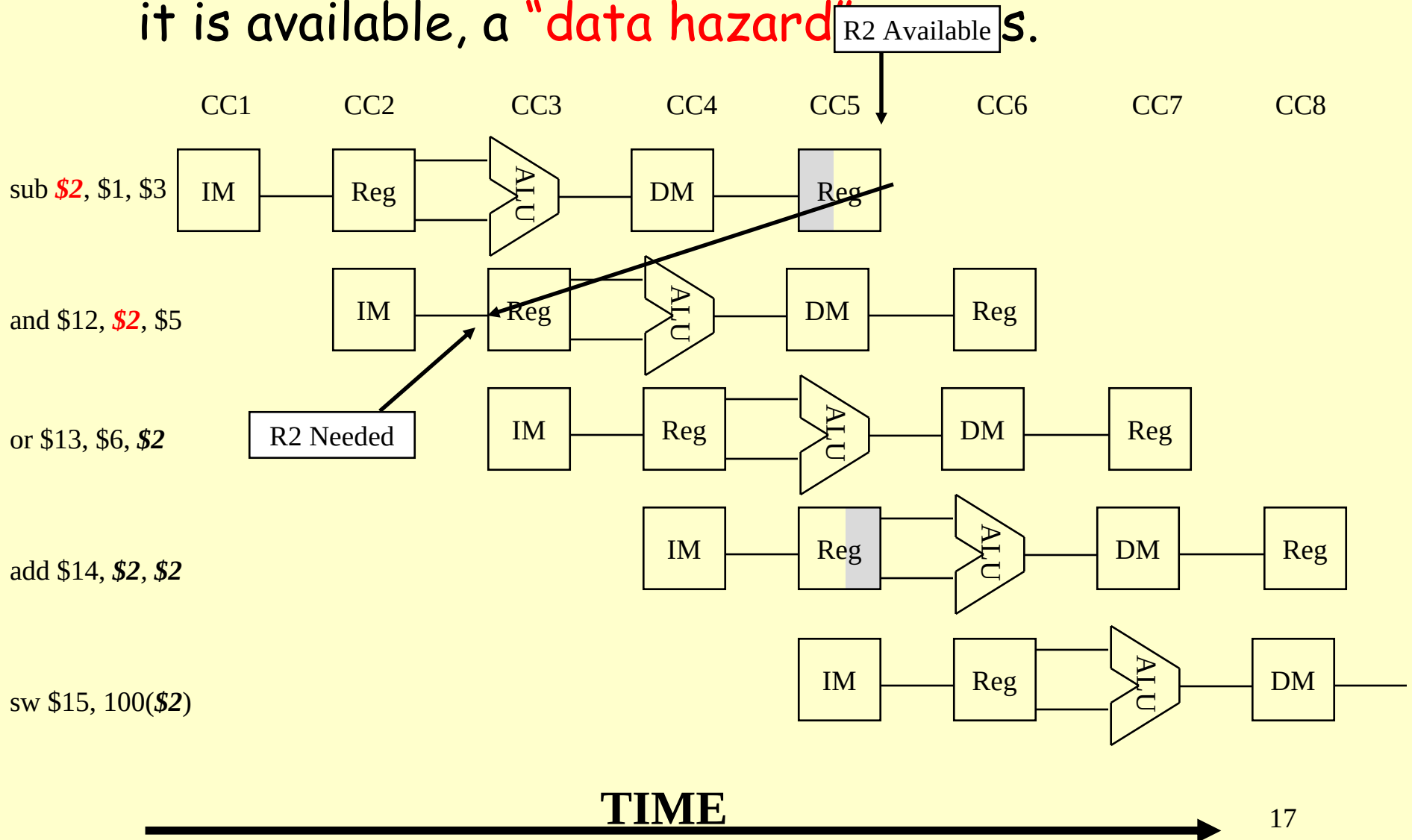
add \$3, \$10, \$11

Write Back



Data Hazards

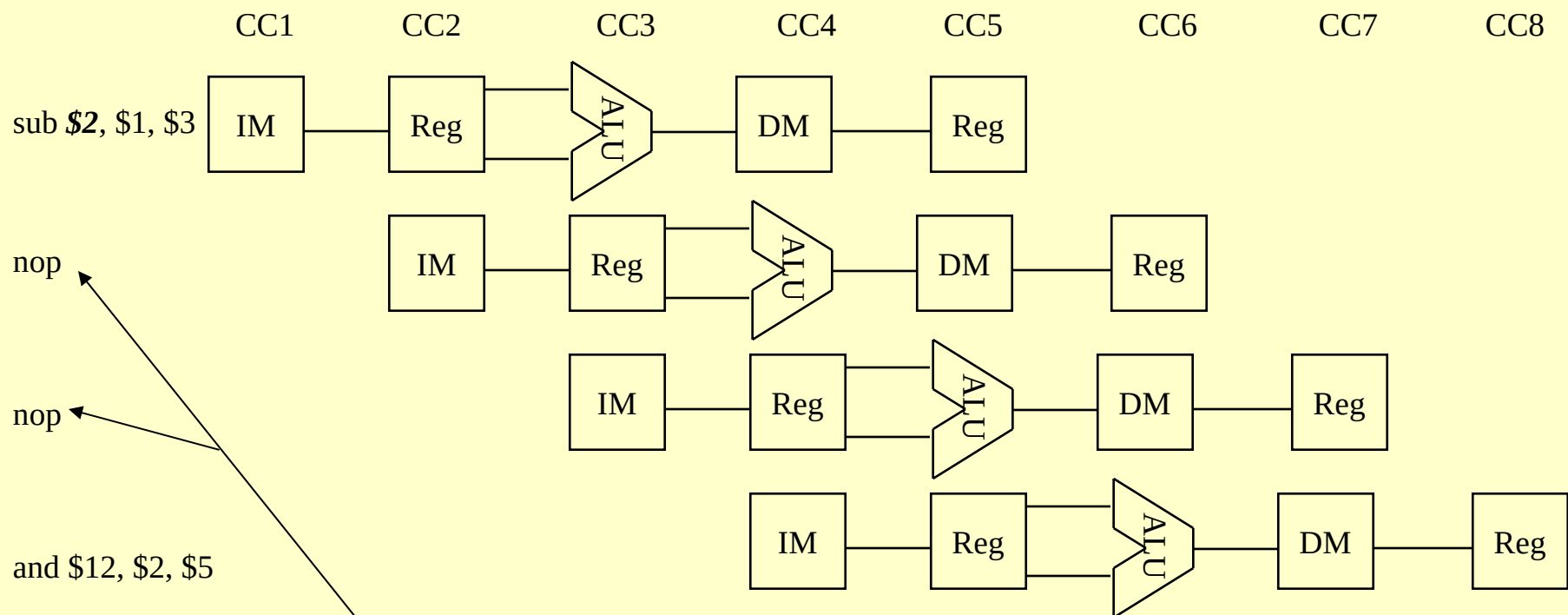
- When a result is needed in the pipeline before it is available, a "data hazard" exists.



Three ways to handle remaining Data Hazards

- In Software
 - insert independent instructions (or no-ops)
- In Hardware
 - insert bubbles (i.e. stall the pipeline)
 - data forwarding

Dealing with Data Hazards in Software



Insert enough no-ops (or other instructions that don't use register 2) so that data hazard doesn't occur,

Where are No-ops needed?

sub \$2, \$1,\$3

and \$4, \$2,\$5

or \$8, \$2,\$6

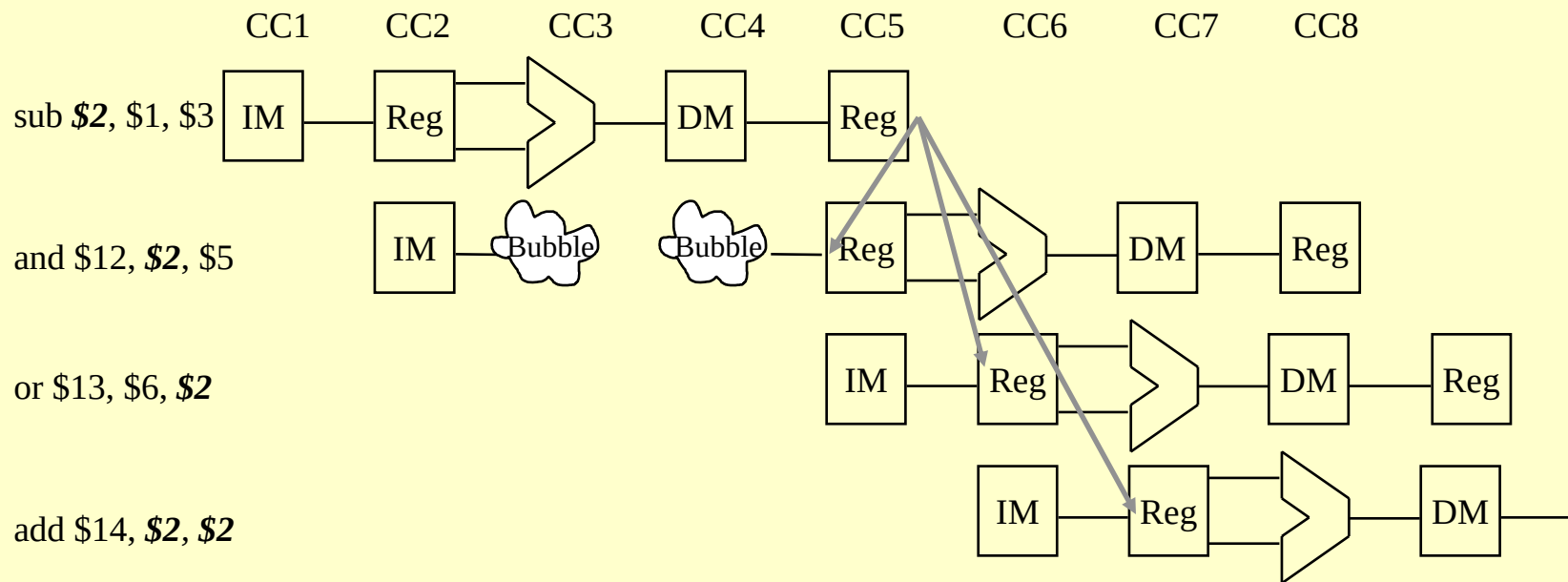
add \$9, \$4,\$2

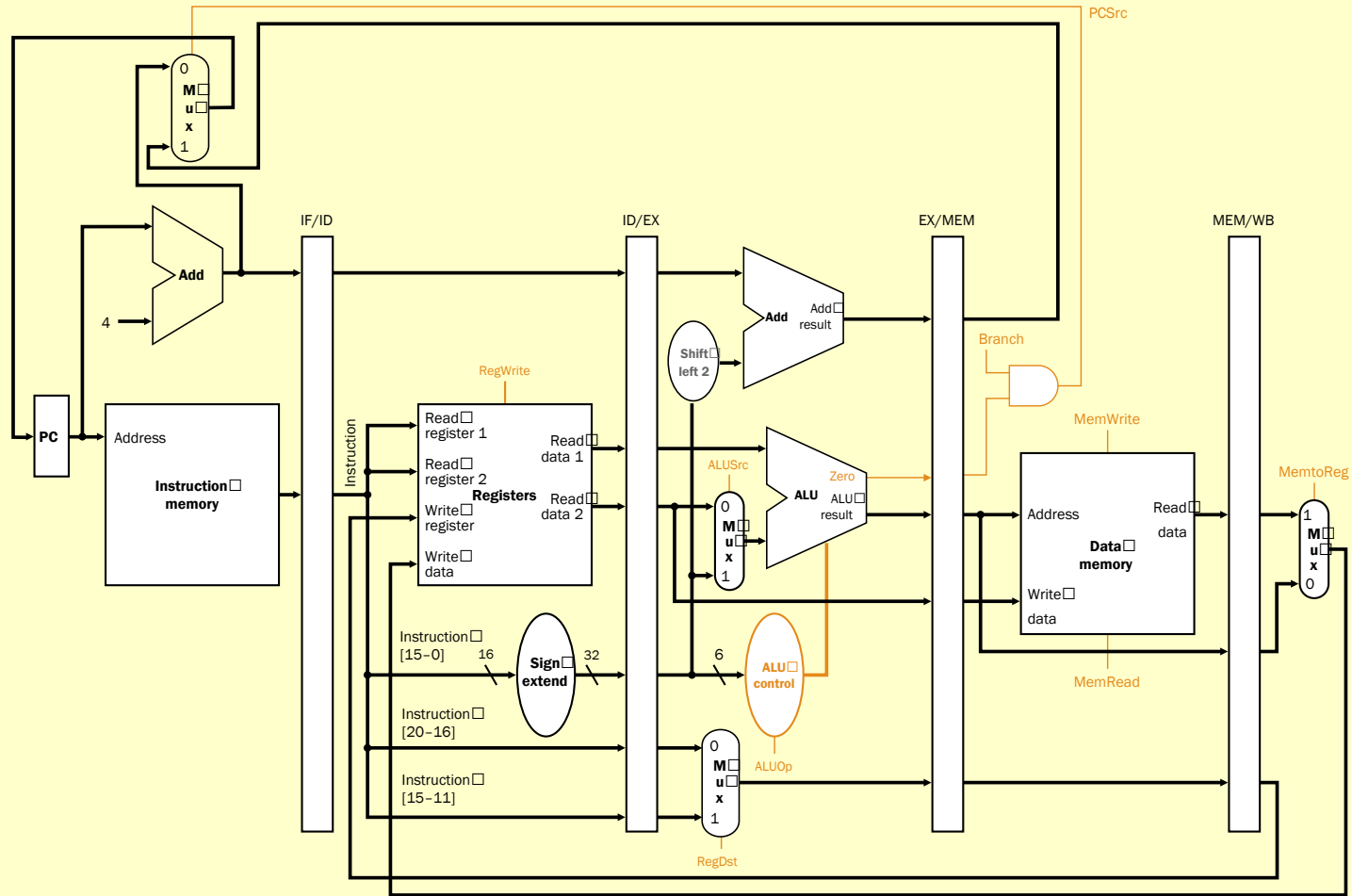
Handling Data Hazards in Hardware: Pipeline Stalls

- To insure proper pipeline execution in light of register dependences, we must:
 - Detect the hazard
 - Stall the pipeline
 - prevent the IF and ID stages from making progress
 - the ID stage because we can't go on until the dependent instruction completes correctly
 - the IF stage because we do not want to lose any instructions.
 - insert "no-ops" into later stages

Handling Data Hazards in Hardware

Stall the pipeline





- What comparisons tell us when to stall?
 - What do we need to compare for the previous instr seq?

```

opc    = Inst[31:26];    func    = Inst[31:26];
rs     = Inst[25:21];    rt      = Inst[20:16];
rd     = Inst[15:11];    sa      = Inst[10:6];
imm    = Inst[15:0];

```

```

WritesRD = (opc == 'OP_RRR)
WritesRT = ( (opc == 'OP_ADDI) || (opc == 'OP_ADDIU)
             || (opc == 'OP_ANDI) || (opc == 'OP_LBU)
             || (opc == 'OP_LHU)  || (opc == 'OP_LW)
             || (opc == 'OP_LUI)  || (opc == 'OP_ORI)
             || (opc == 'OP_SLTI) || (opc == 'OP_SLTIU));

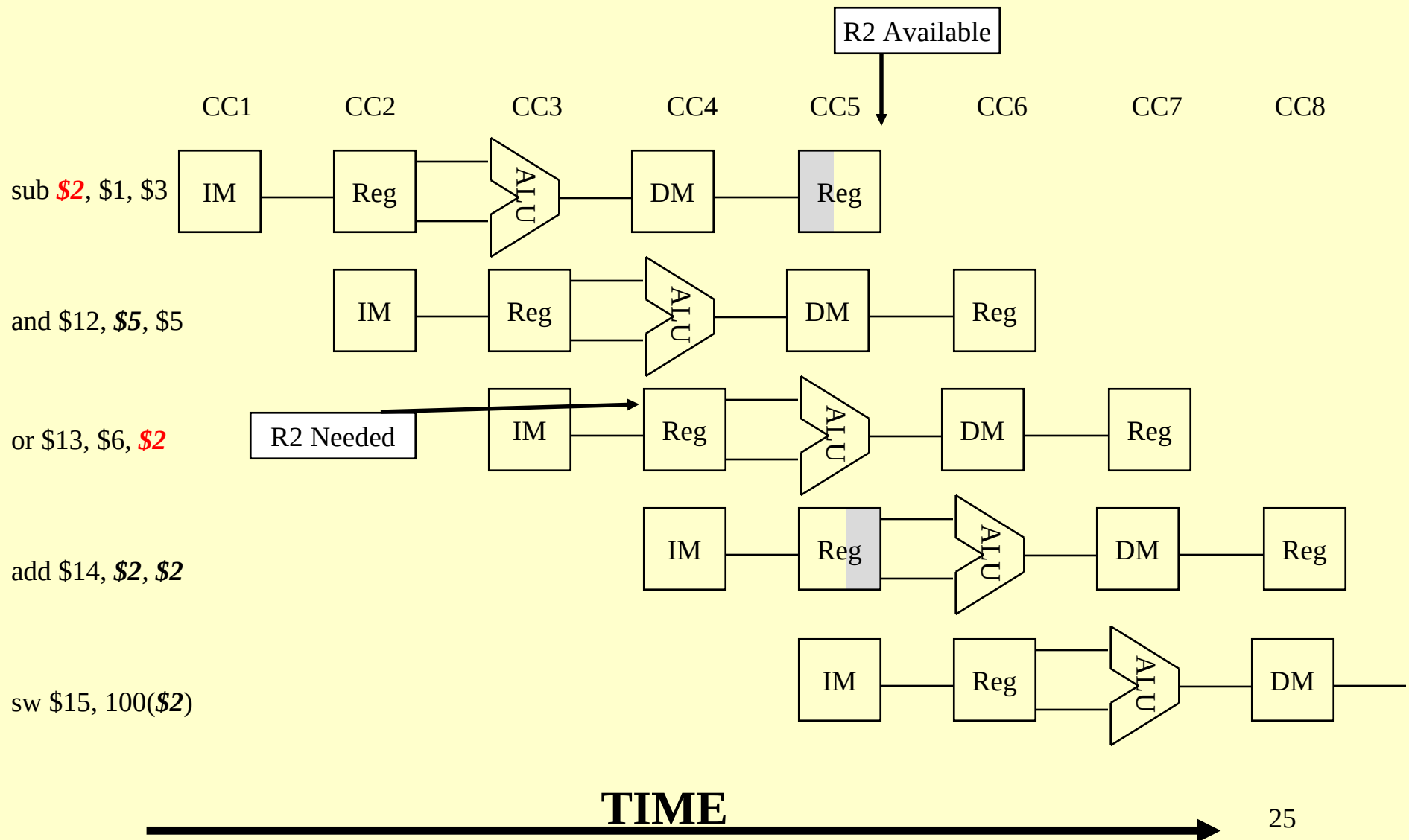
IsStore = ( (opc == 'OP_SB)  || (opc == 'OP_SH)
            || (opc == 'OP_SW));

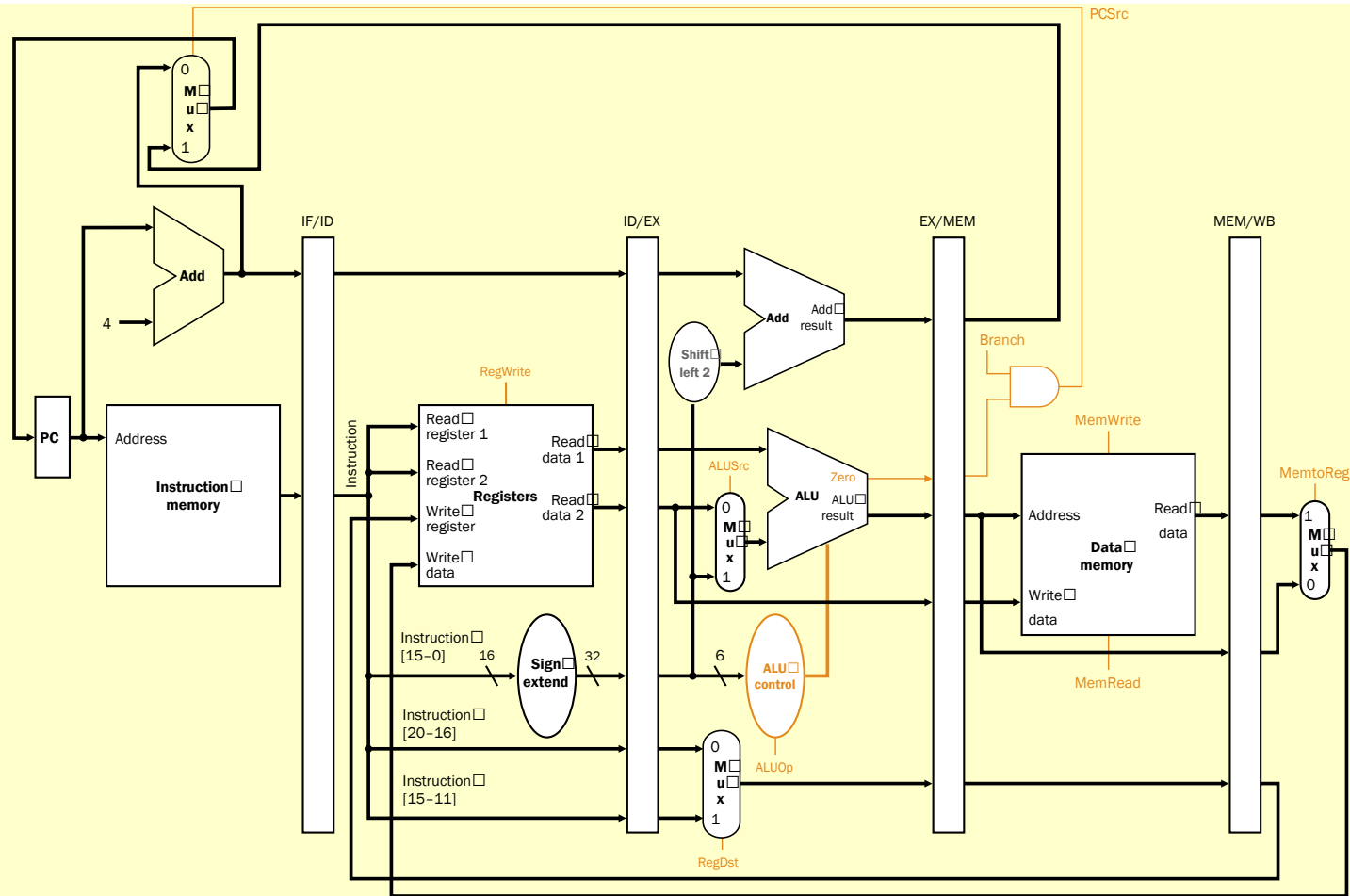
IsLoad  = ( (opc == 'OP_LBU) || (opc == 'OP_LHU)
            || (opc == 'OP_LW));

ReadsRT = ( (opc == 'OP_RRR) || (opc == 'OP_BEQ)
            || (opc == 'OP_BNE) || (IsStore);

```

Did we handle all the cases?



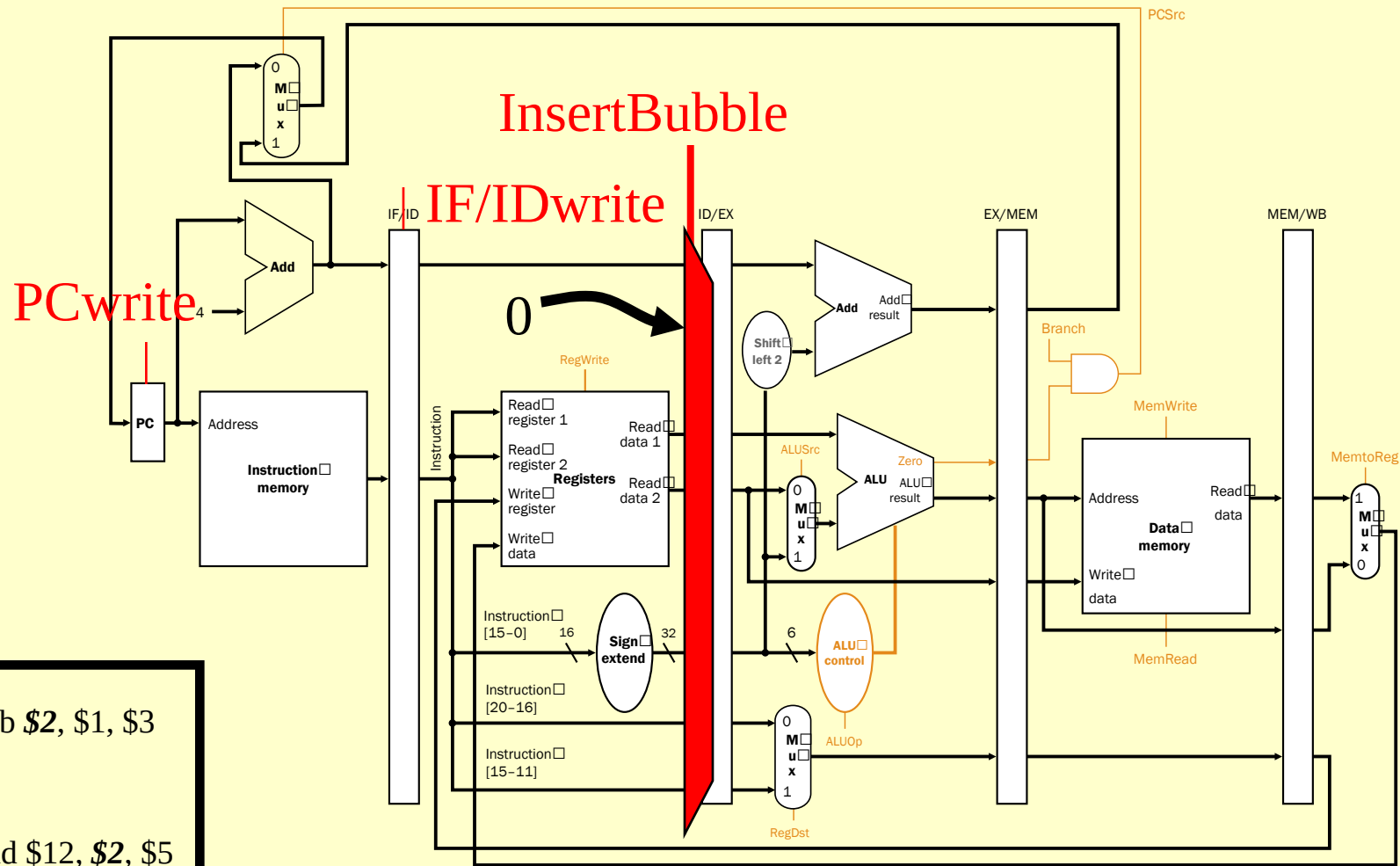


- What comparisons tell us when to stall?
 - What do we need to compare?

Actually Stalling the Pipeline

- Prevent the IF and ID stages from proceeding
 - don't write the PC (PCWrite = 0)
 - don't rewrite IF/ID register (IF/IDWrite = 0)
- Insert "nops" at decode
 - set all control signals propagating to ID/EX/MEM/WB to zero

The Pipeline



sub \$2, \$1, \$3

and \$12, \$2, \$5

or \$13, \$6, \$2

add \$14, \$2, \$2

PCWrite = 0

IF/IDWrite = 0

InsertBubble=1

Self Check!

- I've got a new design for my hardware. Due to switch to 512-bit ALU ops, my pipeline breakdown takes 7 stages:

- IF, ID, EX1, EX2, EX3, MEM, WB

- How many cycles must my hardware stall for:

Sub \$2, \$3, \$1

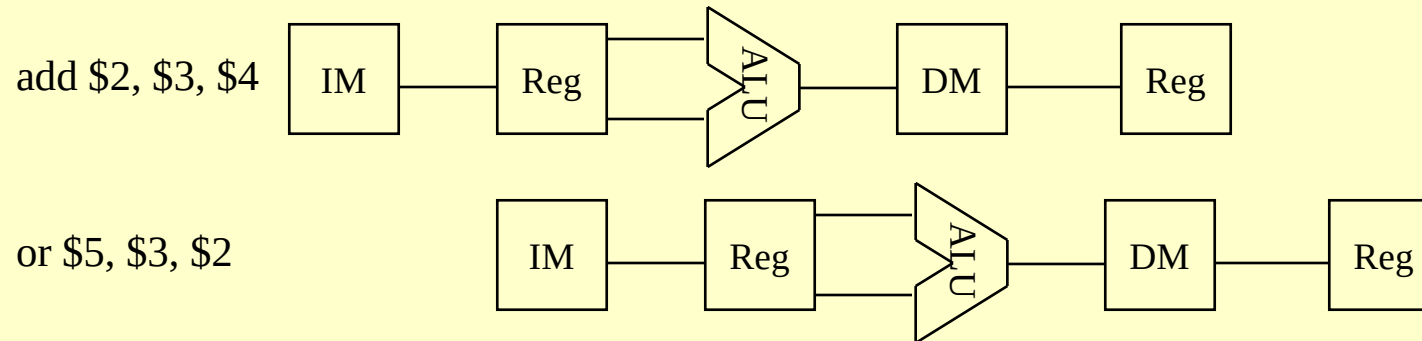
And \$12, \$2, \$5

Can you draw the all the "stall possibilities"?

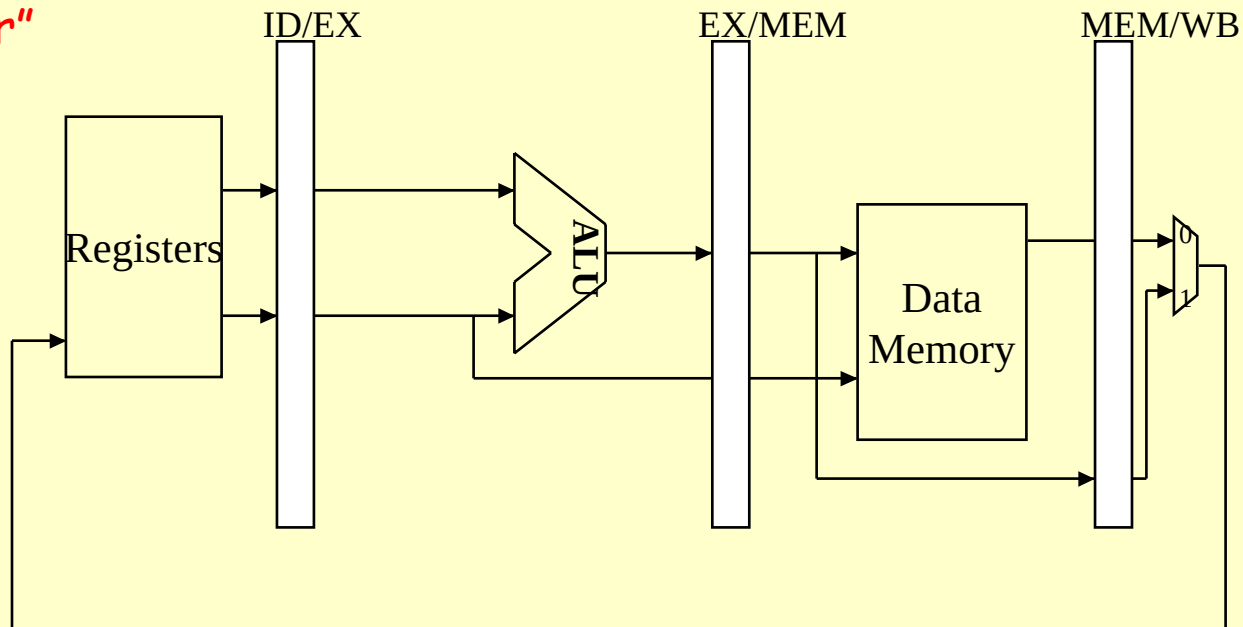
IF, ID, EX1, EX2, EX3, MEM, WB

CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
sub \$2, \$1, \$3								
and \$12, \$2, \$5								
or \$13, \$6, \$2								
add \$14, \$2, \$2								
sub \$14, \$2, \$2								
mult \$14, \$2, \$2								

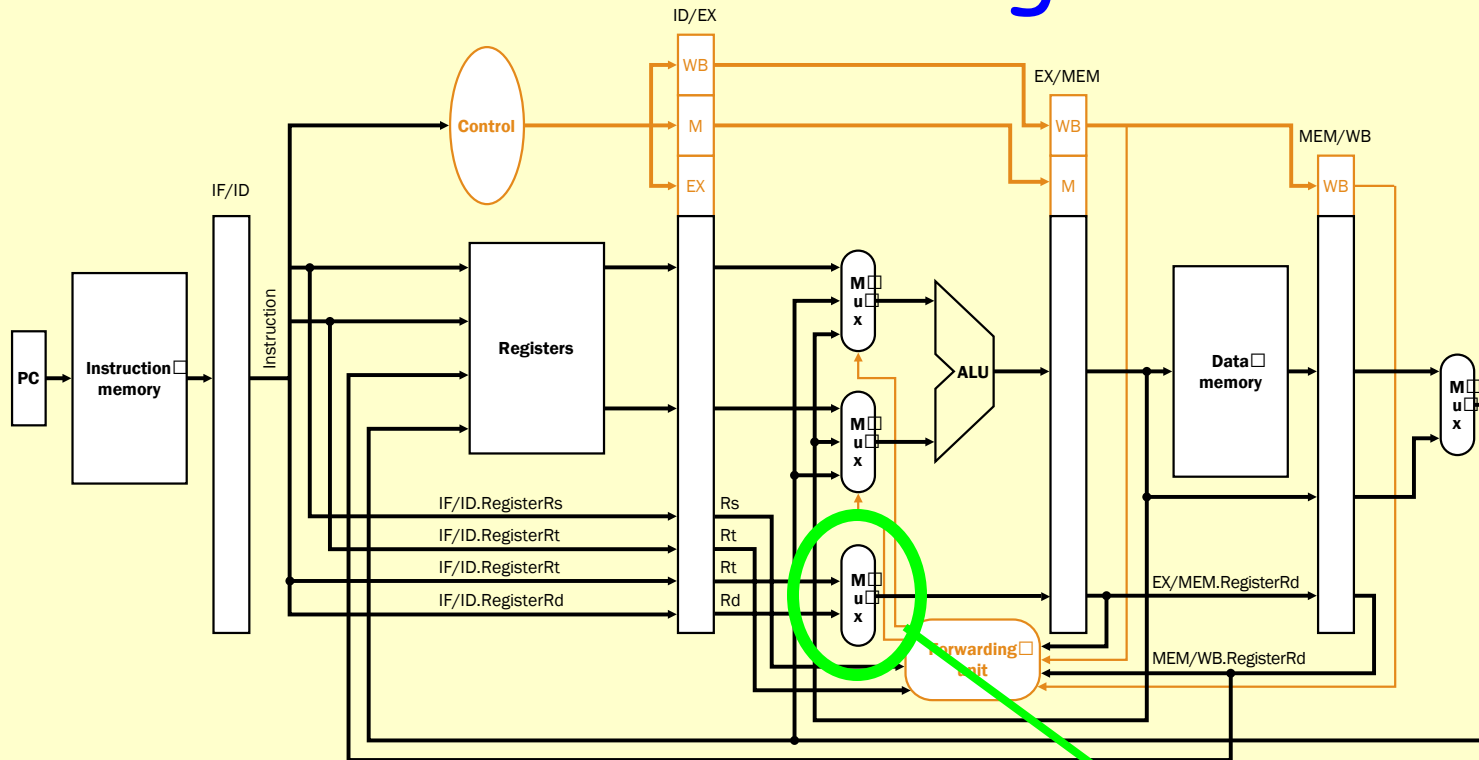
Reducing Data Hazards Through Forwarding



We could avoid stalling if we could get the ALU output from "add" to ALU input for the "or"



Reducing Data Hazards Through Forwarding



EX Hazard:

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

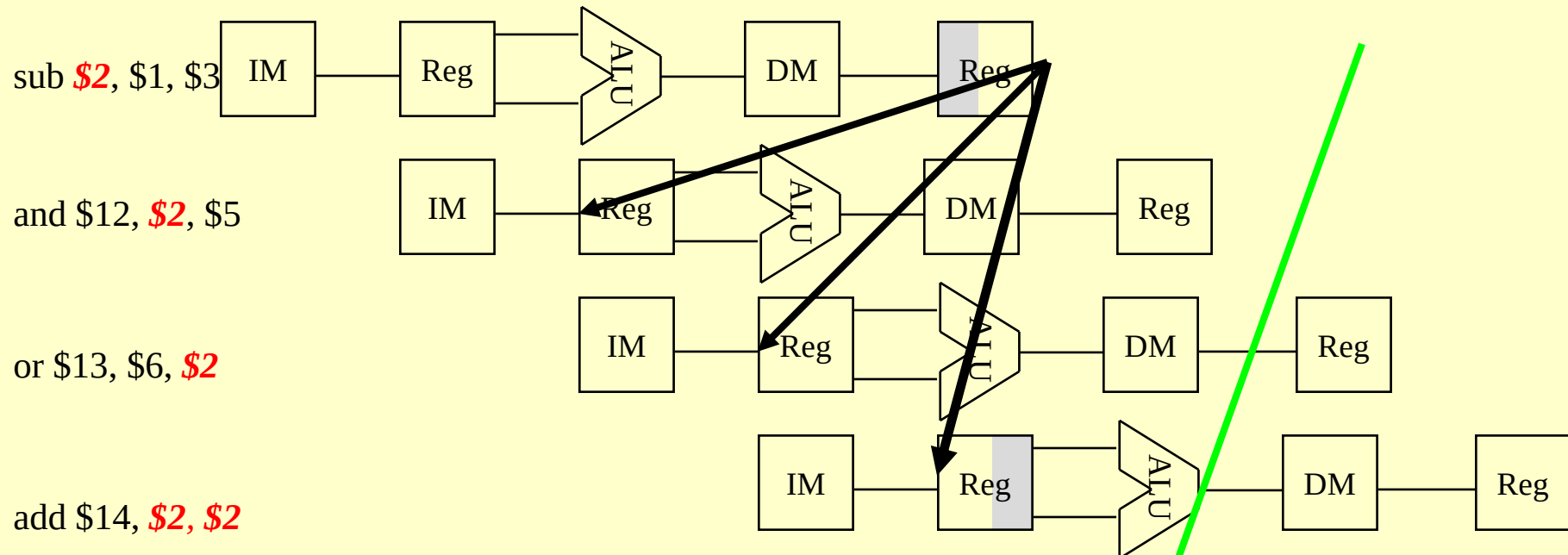
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

handles LD
RegisterRd=RegisterRt

Is That All?

- How many hazards did we have to deal with in this pipeline?

from book



Mem Hazard:

```

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
    and !((EX/MEM.RegisterRd = ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
    and !((EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
    
```

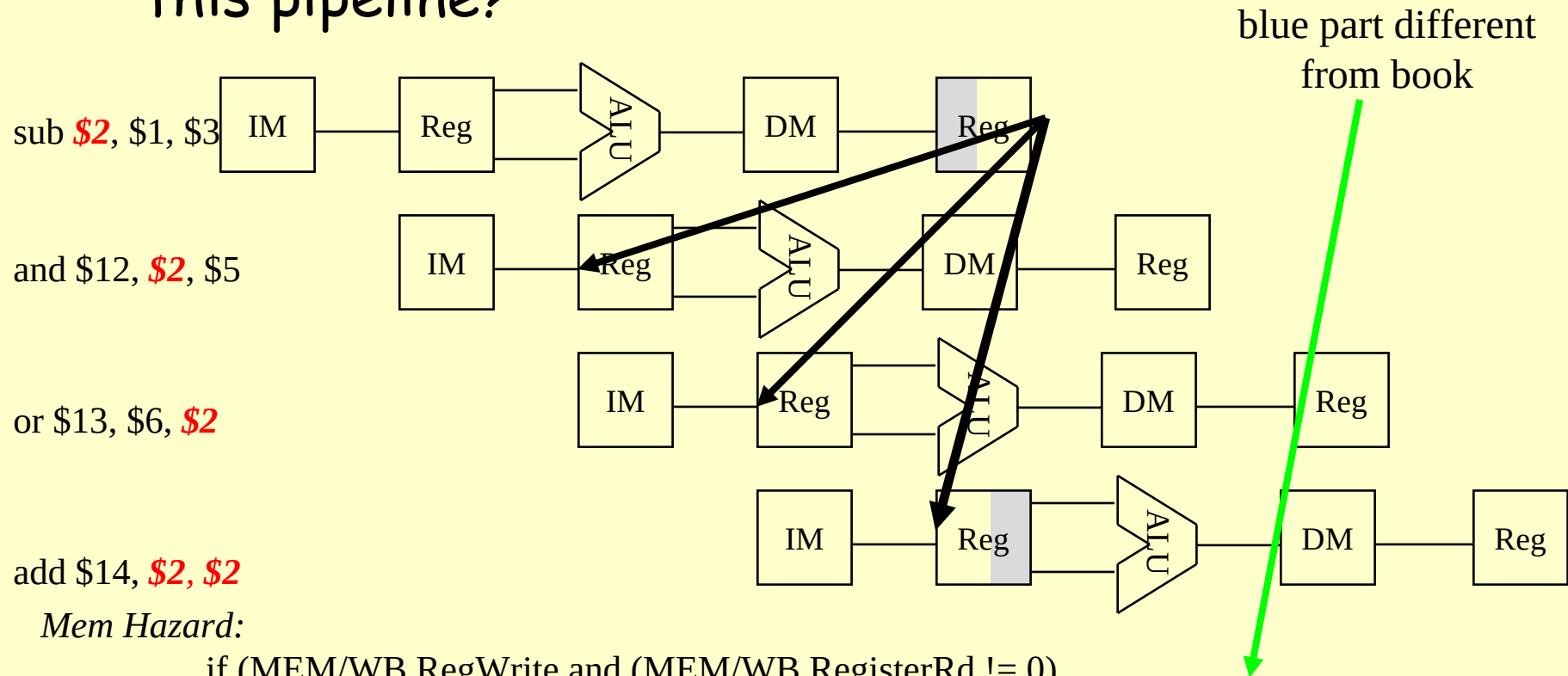
what about:

```

add $1,$4,$25
sw $1,0x800($2)
add $1,$1,$31
    
```

Is That All?

- How many hazards did we have to deal with in this pipeline?



if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0))

and !((EX/MEM.RegisterRd = ID/EX.RegisterRs) and EX/MEM.RegWrite)

and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0))

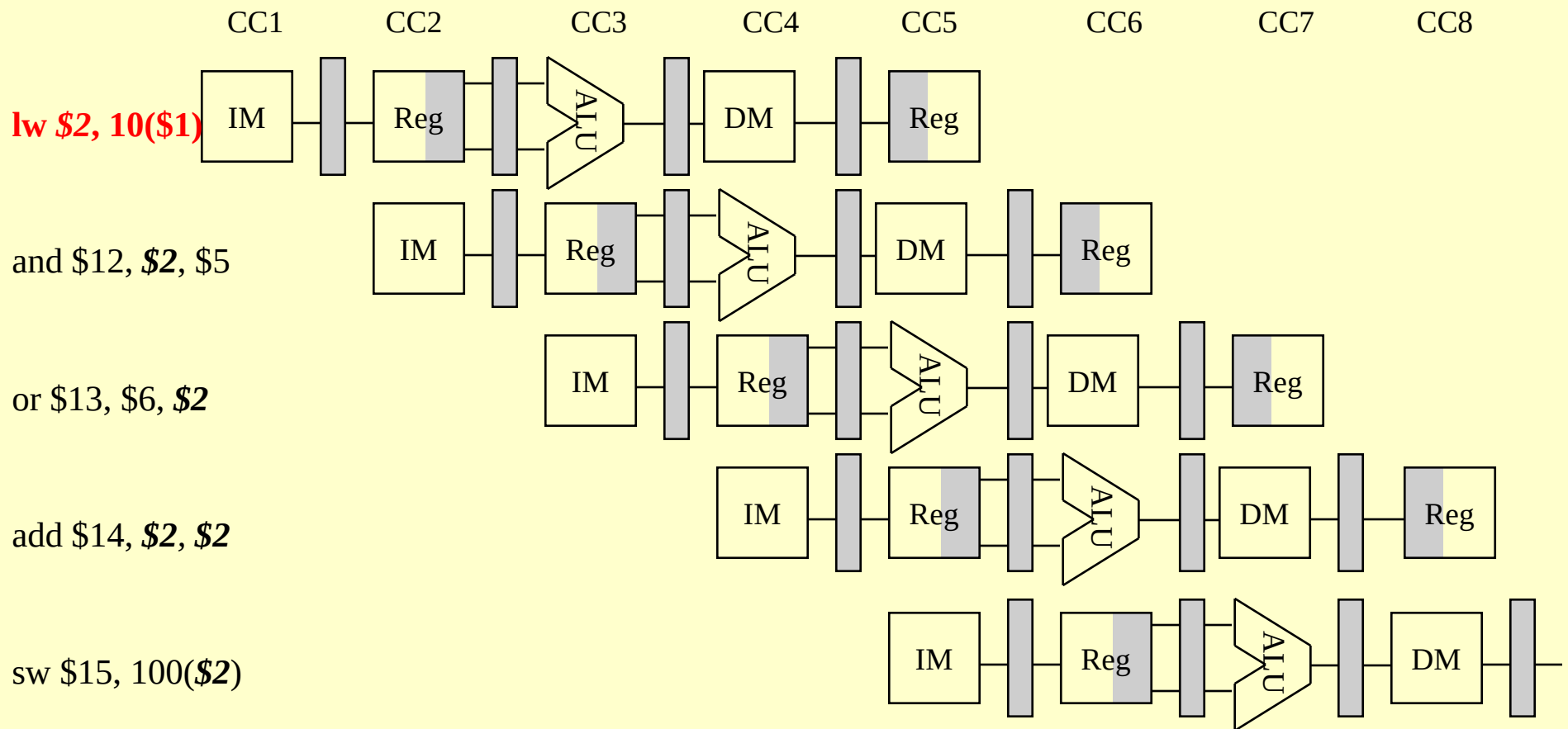
and !((EX/MEM.RegisterRd = ID/EX.RegisterRt) and EX/MEM.RegWrite)

and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

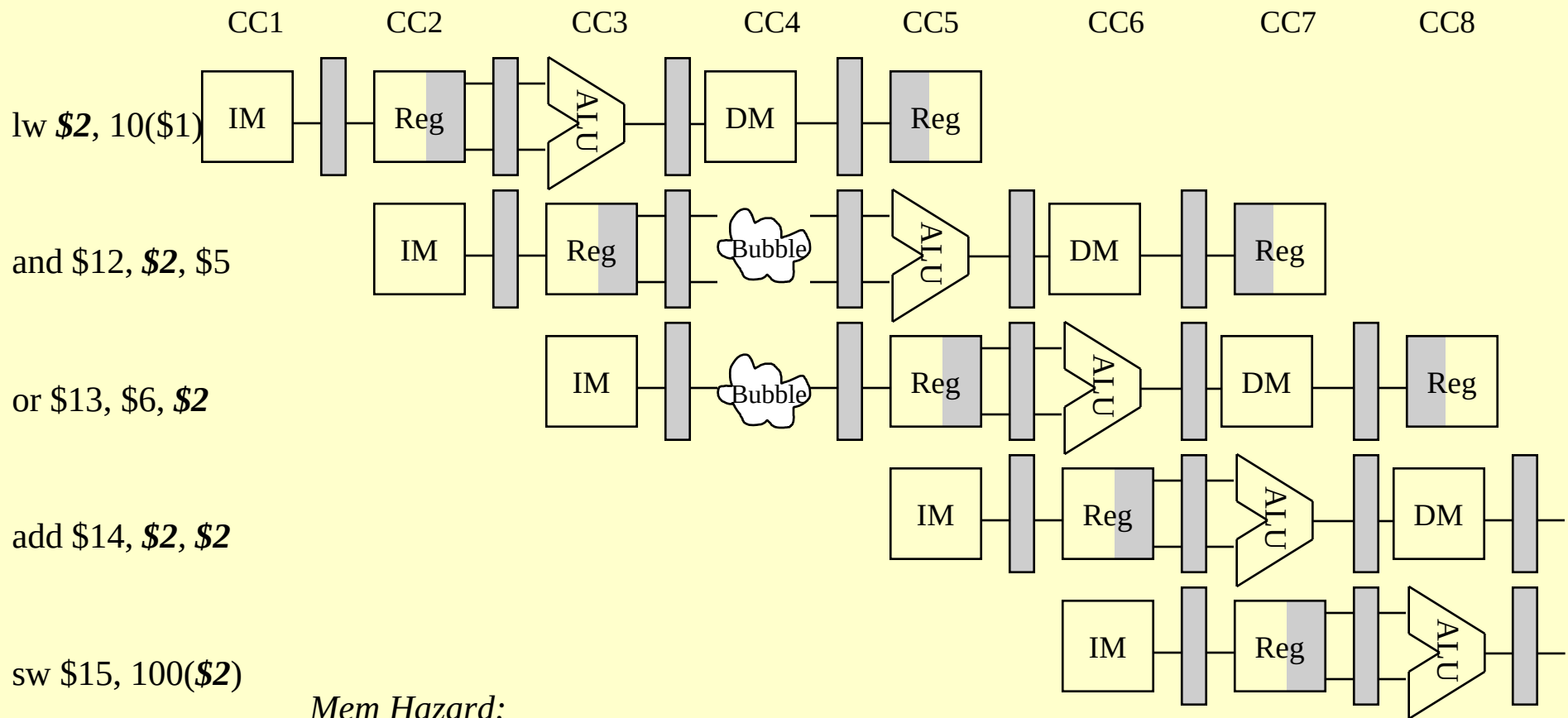
Data Forwarding

- Forwarding (just shown) handles two types of data hazards
 - EX hazard
 - MEM hazard
- We've already handled the third type (WB) hazard by using a bypassing reg file
 - if the register file is asked to read and write the same register in the same cycle, the reg file allows the write data to be forwarded to the output.

Does Forwarding eliminate all hazards??



NO! You may need to stall after loads



Mem Hazard:

if (ID/EX.MemRead and
 ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
 (ID/EX.RegisterRt = IF/ID.RegisterRt)))

stall;

Try this one...

Show stalls and forwarding for this code

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

```
lw $4, 100($3)
```

```
and $6, $4, $3
```

```
sub $7, $6, $2
```

Self Check...

- Can you write control logic for an N-stage pipeline, including bypassing and stalls?
 - Almost certainly on the final
 - practice!
 - create a pipeline
 - write the control logic

Some other specific forwarding situations

- Page 412 of book
 - Loads forwarding to stores
- AddI not in the datapath shown in book, extra mux needed for it
 - Could you figure out what extra forwarding would be needed for addi?

Data Hazard Key Points

- Pipelining provides high throughput, but does not handle data dependences easily.
- Data dependences cause *data hazards*.
- Data hazards can be solved by:
 - software (no-ops)
 - hardware stalling
 - hardware forwarding
- Our processor, and indeed all modern processors, use a combination of forwarding and stalling.

Branch Hazards in the Pipelined Processor

\$3 = \$2 + \$1

\$1 = \$2 - \$5

if (\$1 == \$3)
 \$5 = \$1 * \$3

else

\$5 = \$1 * \$2

\$10 = \$11 | \$12

...

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

sub \$1, \$2, \$5

beq \$1, \$3, yupEqual

mult \$5, \$1, \$2

jump goOn

yupEqual:

mult \$5, \$1, \$3

goOn:

or \$10,\$11,\$12

....

Dependences

- Data dependence: one instruction is dependent on another instruction to provide its operands.
- Control dependence (aka branch dependences): one instructions determines whether another gets executed or not.
- Control dependences are particularly critical with conditional branches.

add \$5, \$3, \$2

sub \$6, \$5, \$2

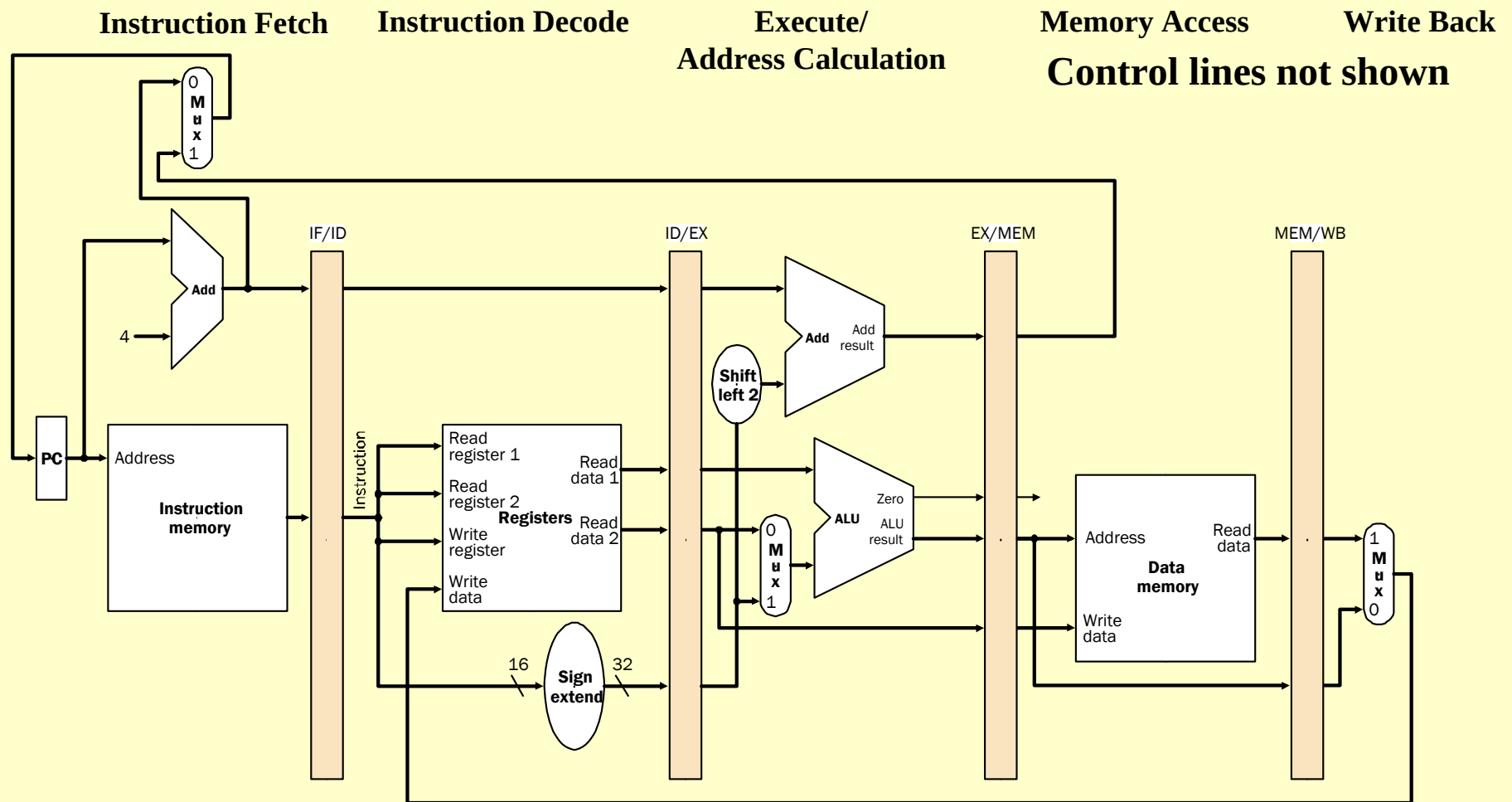
beq \$6, \$7, somewhere

and \$9, \$3, \$1

Control Hazards vs. Data Hazards

- How are control hazards and data hazards different?
- When could the pipeline "know" a data hazard existed? How about a control hazard?

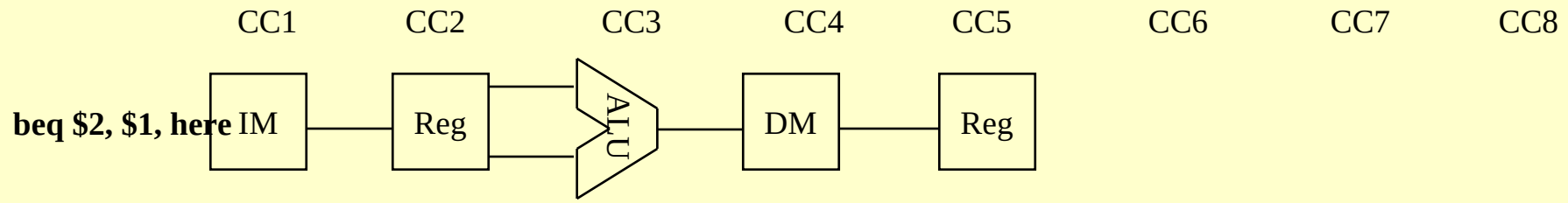
When are branches resolved?



Branch target address is put in PC during:
Correct "next" instruction is fetched during:

What is my branch hazard penalty?

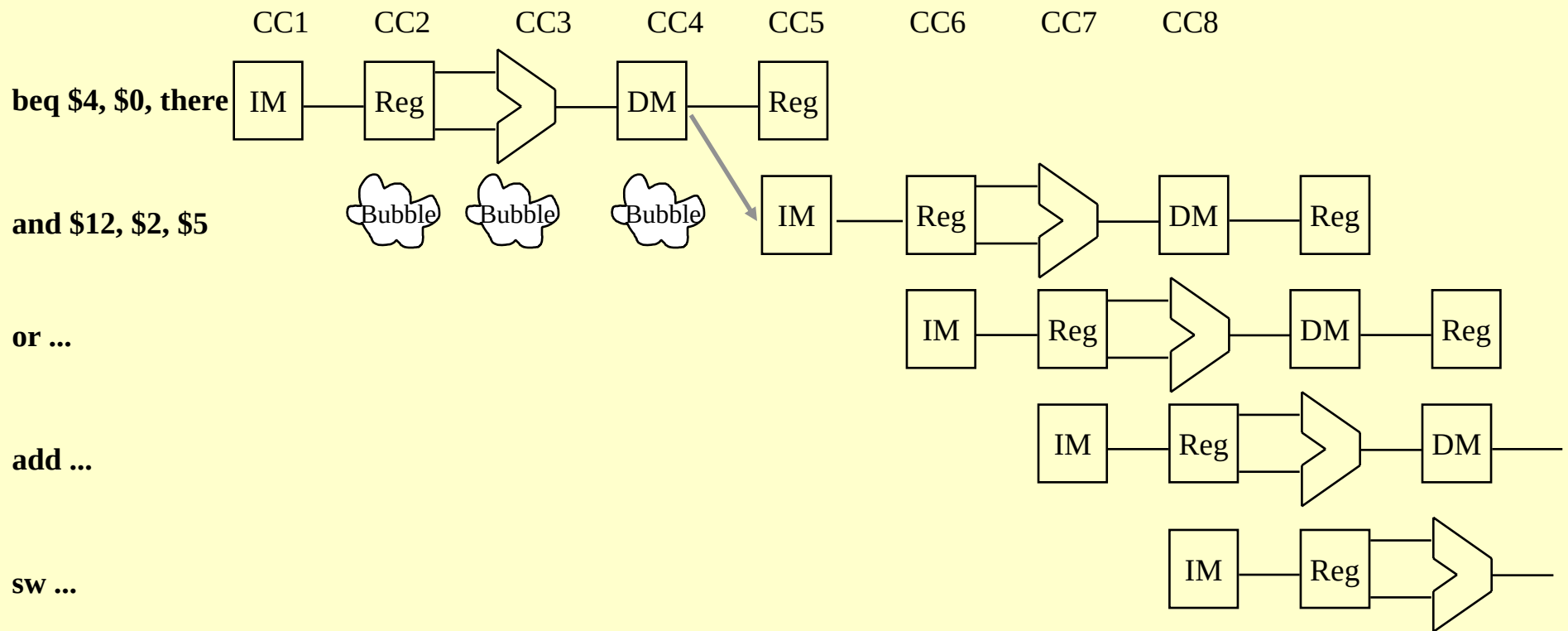
How many cycles will I possibly be fetching the *WRONG* instructions



Dealing With Branch Hazards

- Software solution
 - insert no-ops (I don't think any processors do this)
- Hardware solutions
 - stall until you know which direction branch goes
 - guess which direction, start executing chosen path (but be prepared to undo any mistakes!)
 - static branch prediction: base guess on instruction type
 - dynamic branch prediction: base guess on execution history
 - reduce the branch delay
- Software/hardware solution
 - delayed branch: Always execute instruction after branch.
 - Compiler puts something useful (or a no-op) there.

Stalling for Branch Hazards

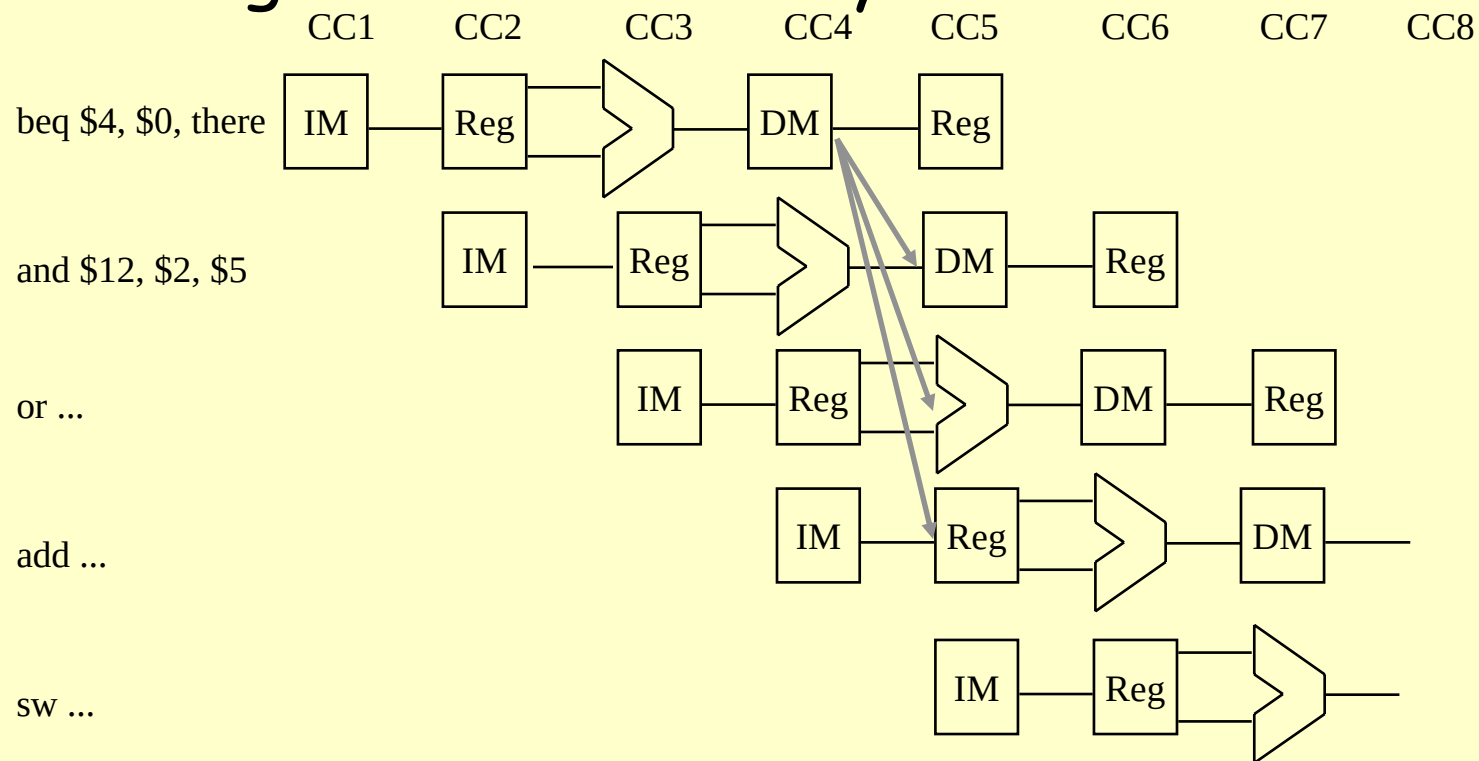


Stalling for Branch Hazards

- All branches waste 3 cycles.
 - Seems wasteful, particularly when the branch isn't taken.
- It's better to guess whether branch will be taken
 - Easiest guess is "branch isn't taken"

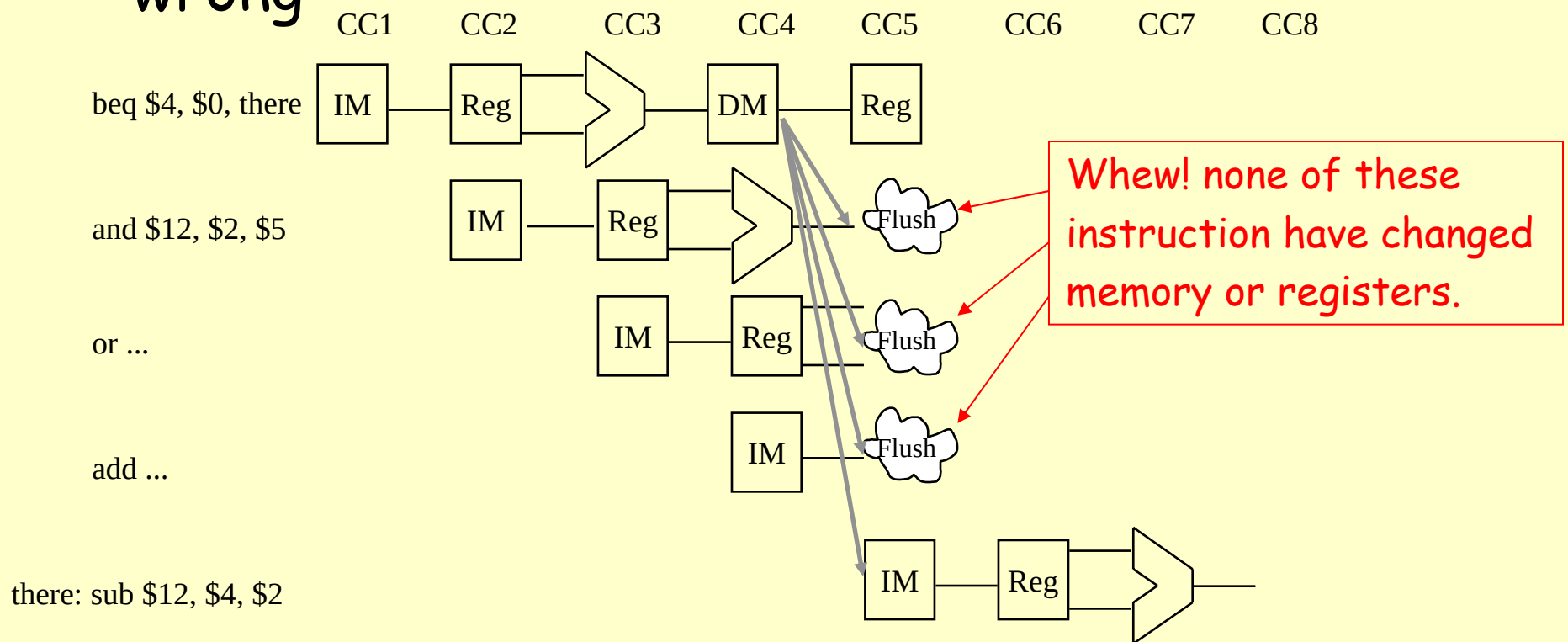
Assume Branch Not Taken

- works pretty well when you're right - no wasted cycles



Assume Branch Not Taken

- same performance as stalling when you're wrong

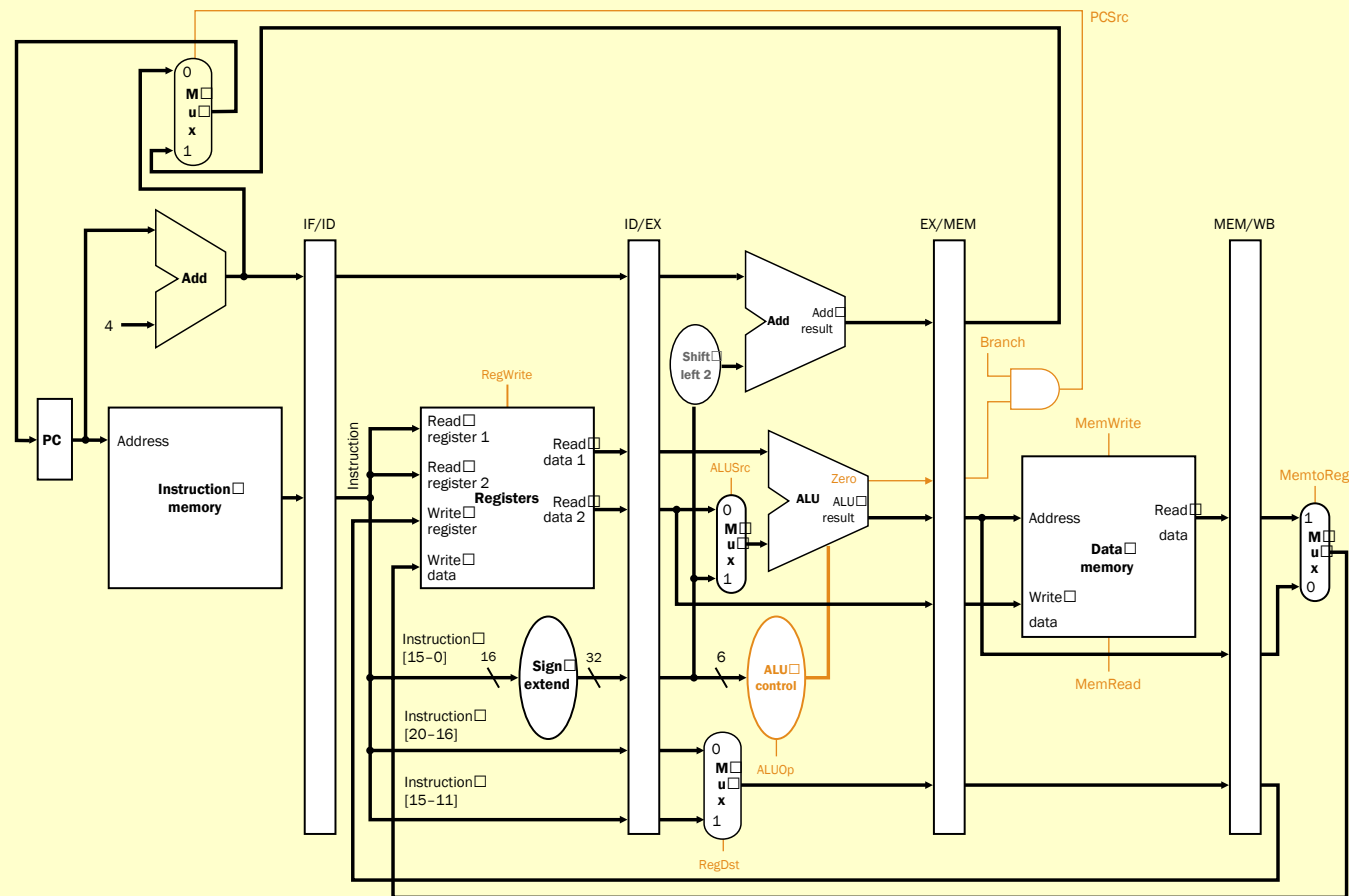


Some other static BP strategies

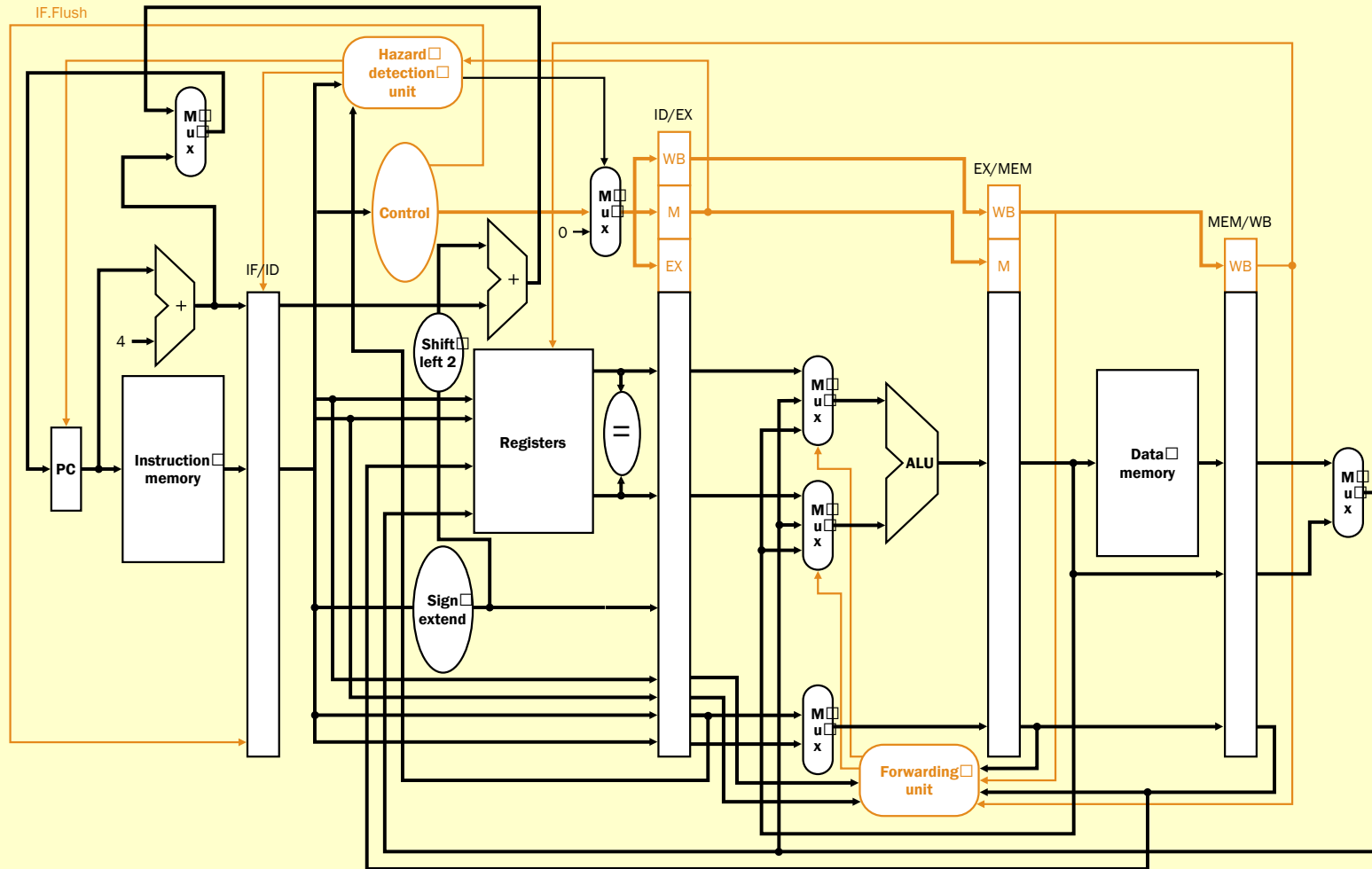
1. predict backwards taken,
forward not taken
 - "backwards" = negative displacement field
 - loops (which branch backwards) are usually executed multiple times.
 - "if-then-else" often takes the "then" (no branch) clause.
2. Compiler makes educated guess
 - sets "predict taken/not taken" bit in instruction

Reducing the Branch Delay

it's easy to reduce stall to 2-cycles



Reducing the branch delay



- Target computation & equality check in ID phase.

- This figure also shows flushing hardware.

Eliminating the Branch Stall

- There's no rule that says we have to branch immediately. We could wait an extra instruction before branching.
- The original SPARC and MIPS processors used a branch delay slot to eliminate single-cycle stalls after branches.
- The instruction after a conditional branch is always executed in those machines, whether the branch is taken or not!

Finding an instruction to eliminate the branch stall

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

sub \$1, \$2, \$5

beq \$1, \$3, yupEqual

mult \$5, \$1, \$2

jump goOn

yupEqual: mult \$5, \$1, \$3

goOn: or \$10,\$11,\$12

....