Topics

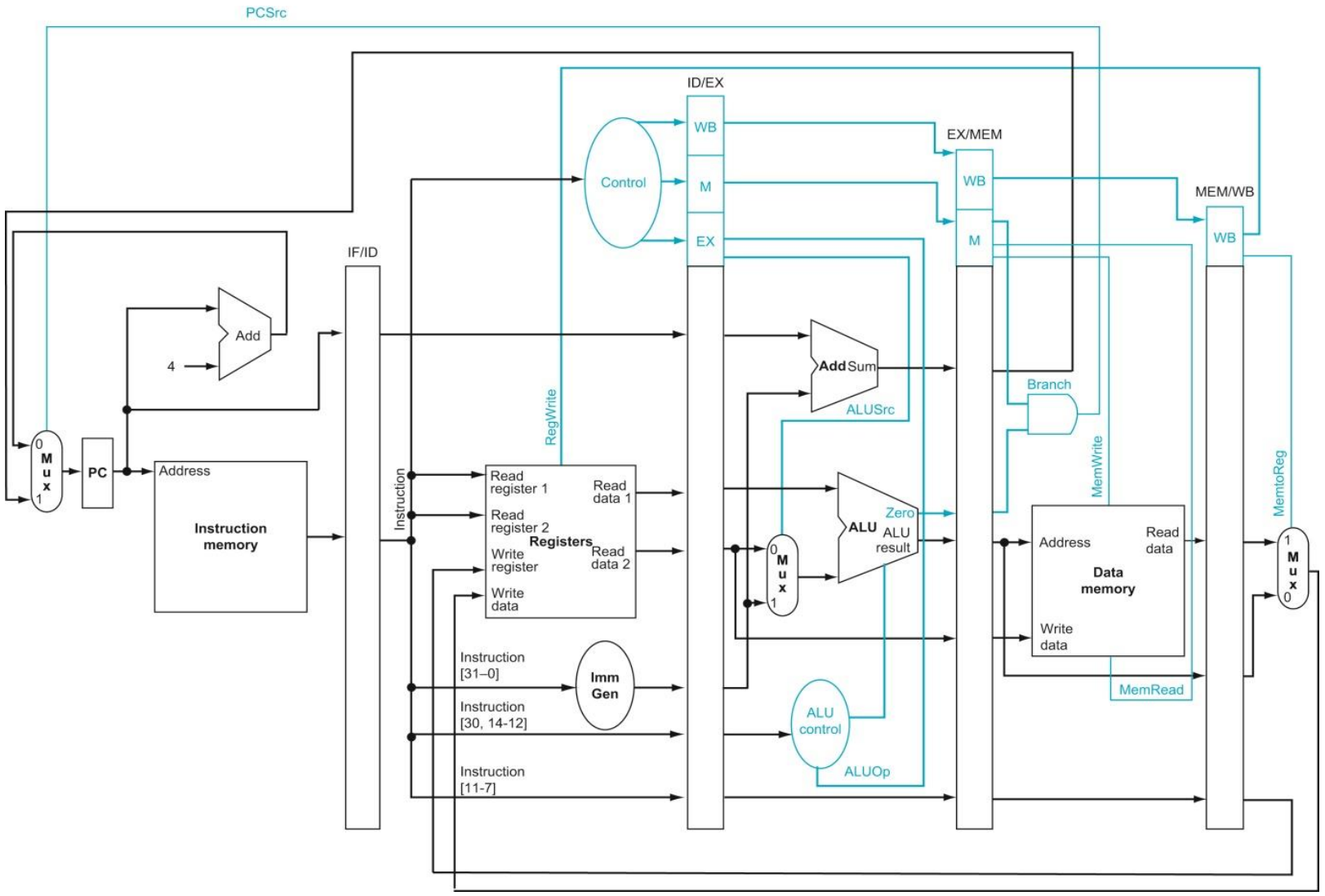
- Single Cycle CPU Design (sections 4.1-4.4)

- Pipelining
  - Overview (section 4.6)
  - Pipelined Datapath and control (section 4.7)
  - Data hazard (section 4.8)
  - ➔ Control hazard (section 4.9)

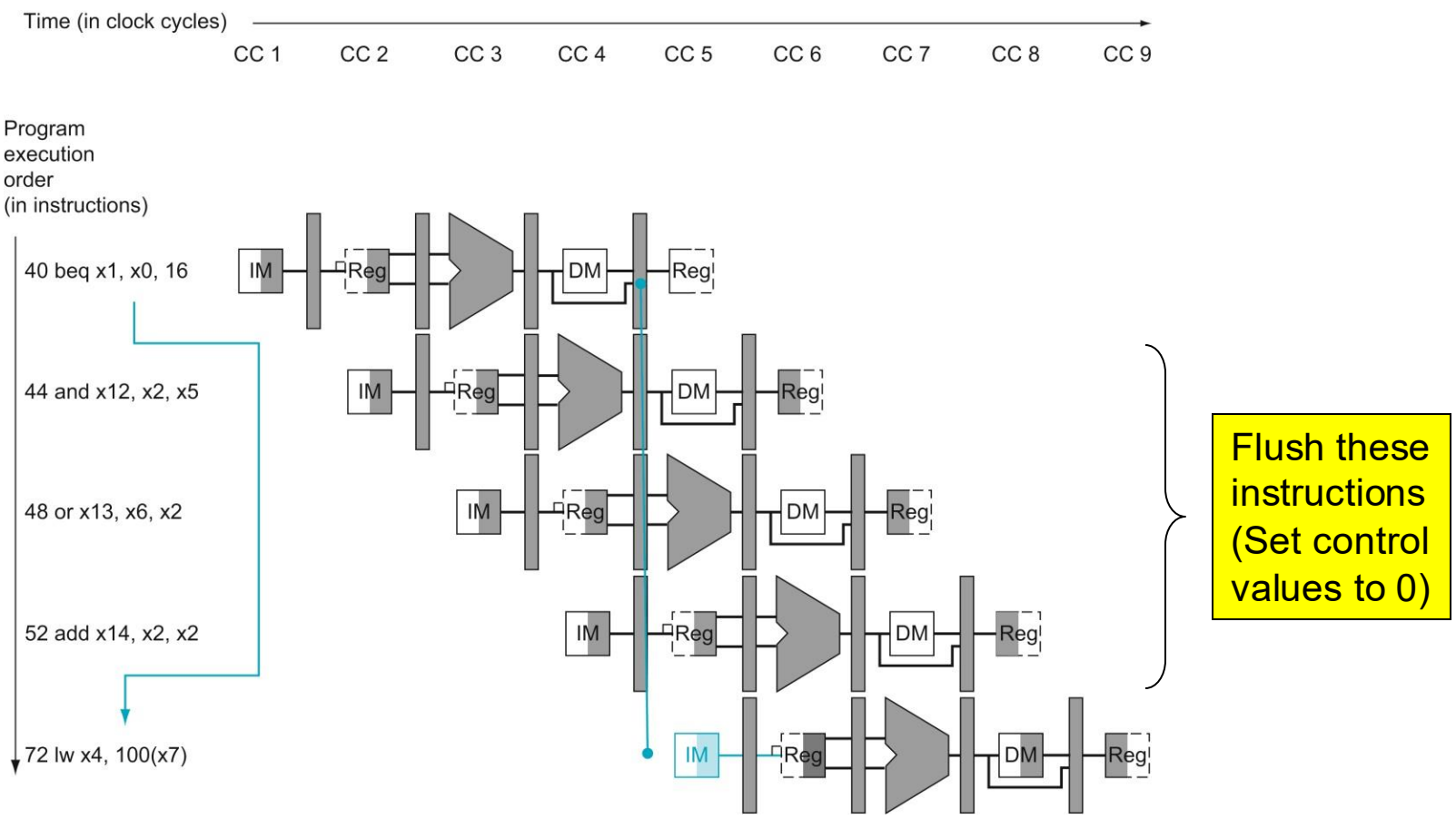
Thus times do shift,
each thing his turn does hold;
New things succeed,
as former things grow old.

Robert Herrick

Branch Logic

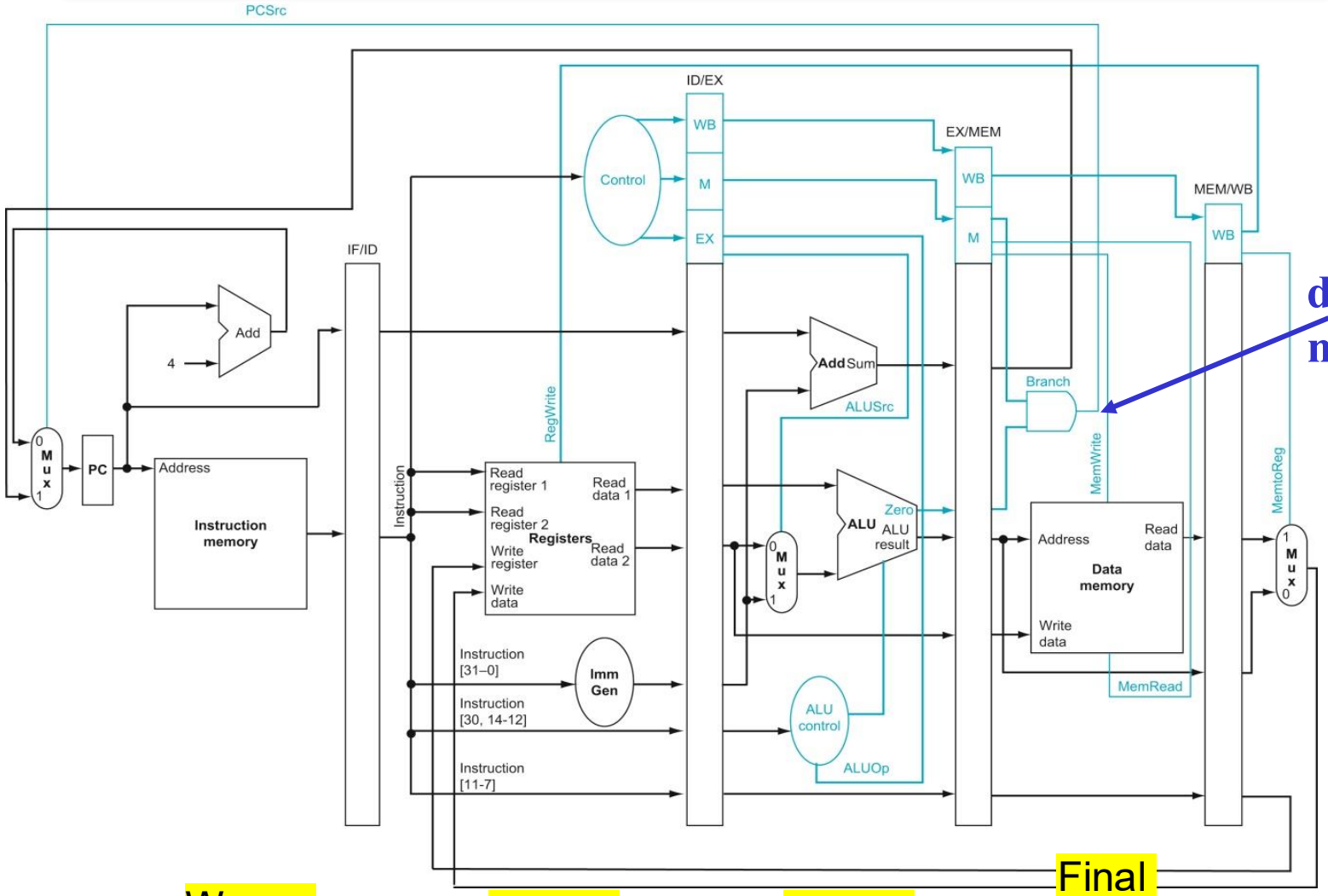


Branch Hazards



branch hazard: attempt to make a decision before condition is evaluated

Branch Hazards



Observations

- **Current design, non-optimized**
  - Branch decision does not occur until MEM stage⇒ waste 3 CCs
- **Is it possible to reduce branch delay?**
  - YES
  - How? For beq x, y, label, x xor y then or all bits
  - In EXE stage
    - Two CCs branch delay
  - In ID Stage
    - One CC branch delay
- **3 strategies to resolve branch hazard**
  - Delayed branch;
  - Static branch prediction;
  - Dynamic branch Prediction

Delayed Branch

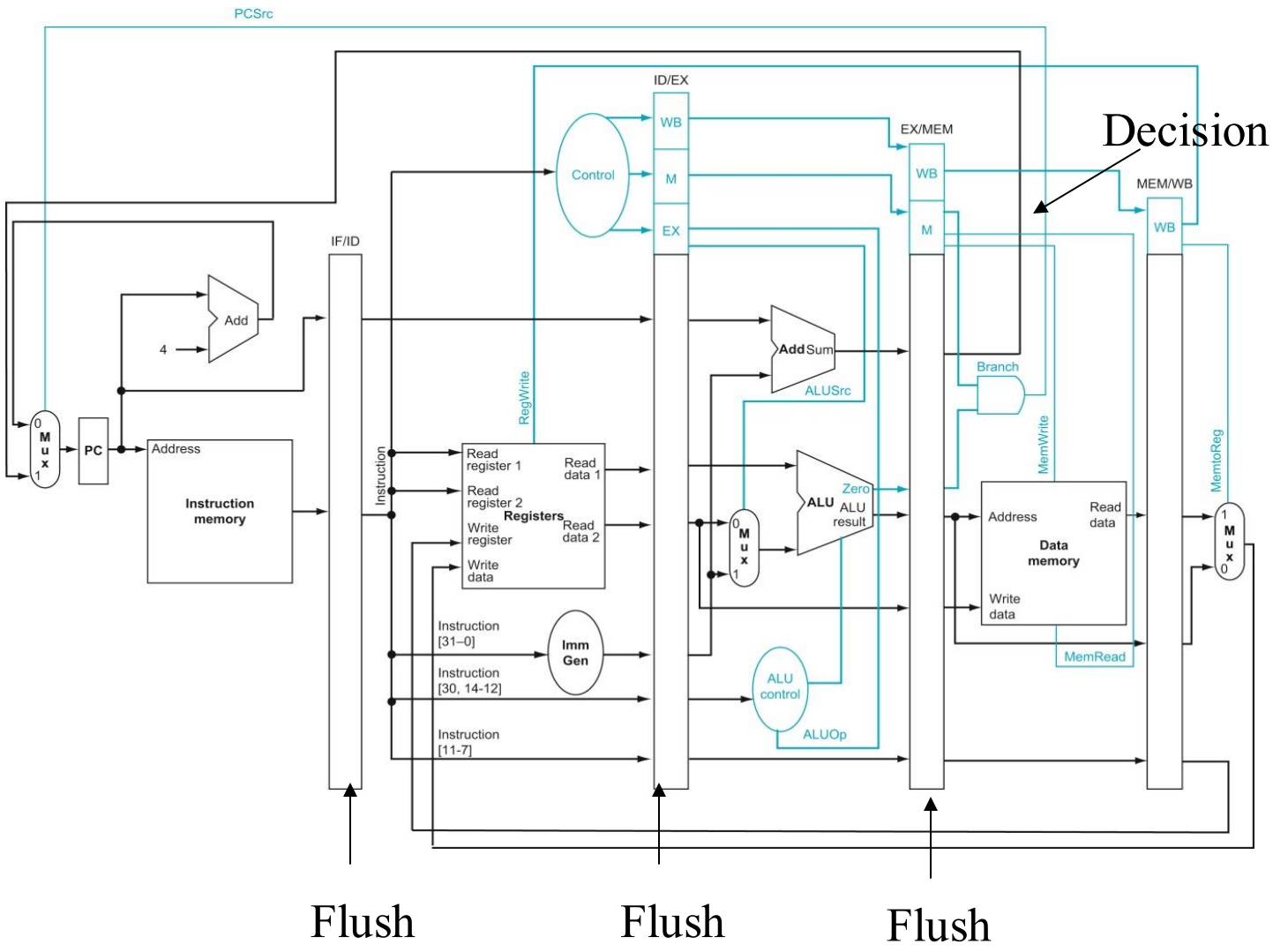
- **always execute the instruction following the branch.**
  - Only one will be executed
- **Done by compiler or assembler**
  - 50% successful rate

Static Branch Prediction

- **Assume the branch will not be taken**
- **If prediction is wrong, clear the effect of sequential instruction execution.**
  - Branch decision is made at MEM stage: instructions in IF, ID, EX stages need to be discarded.
  - Branch decision is made at ID stage: only flush IF/ID pipeline register!

- ***note, flush is not stall! … we are not repeating the flushed instruction, we are discarding it.***

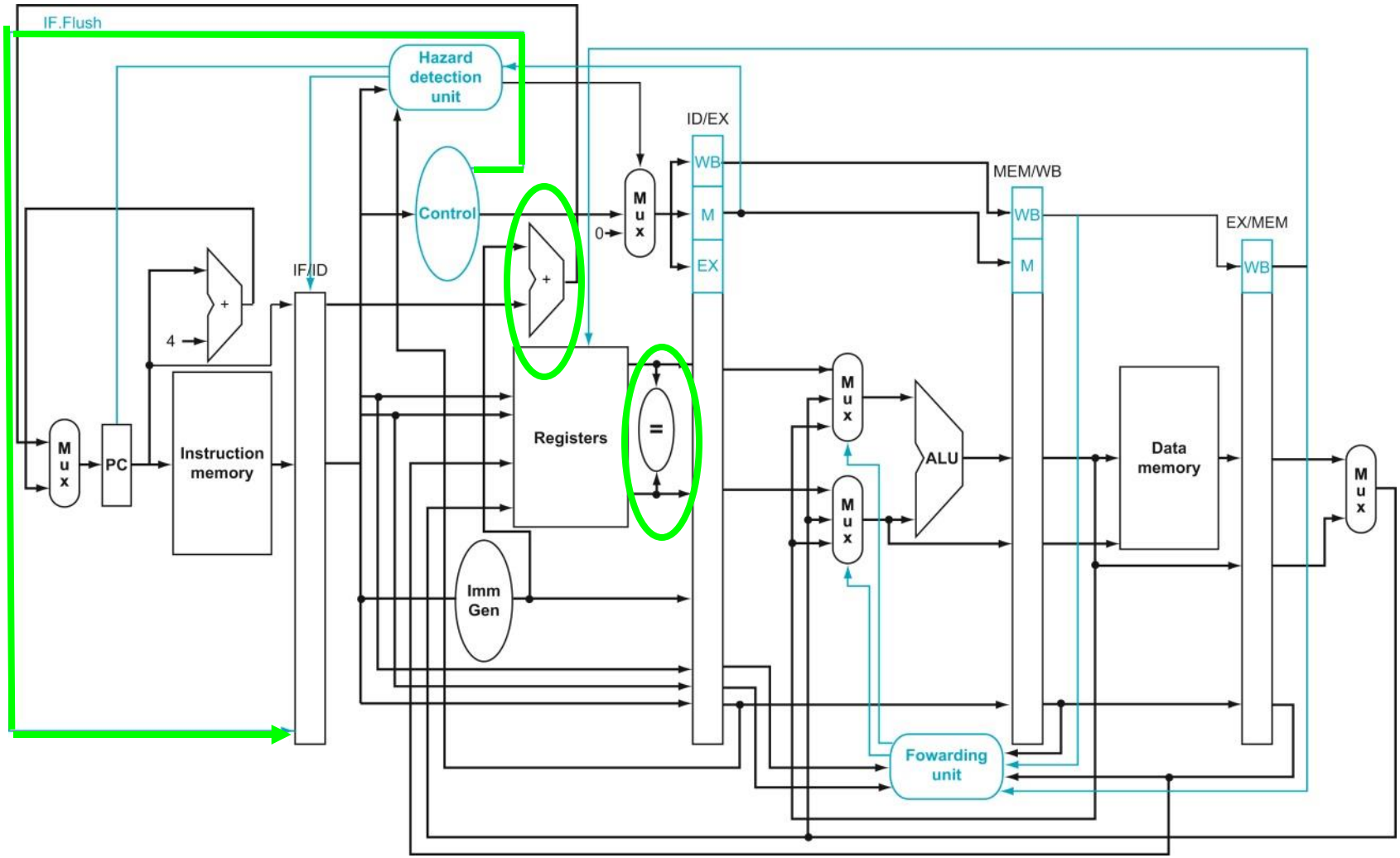
Branch Decision at the MEM Stage



Reducing Penalty

- **Move hardware to an earlier stage: ID stage**
  - Branch Target Address (BTA) adder
  - Register comparator (new!)
  - Control combines Branch and comparator for IF Flush signal, BTA selection in ID stage

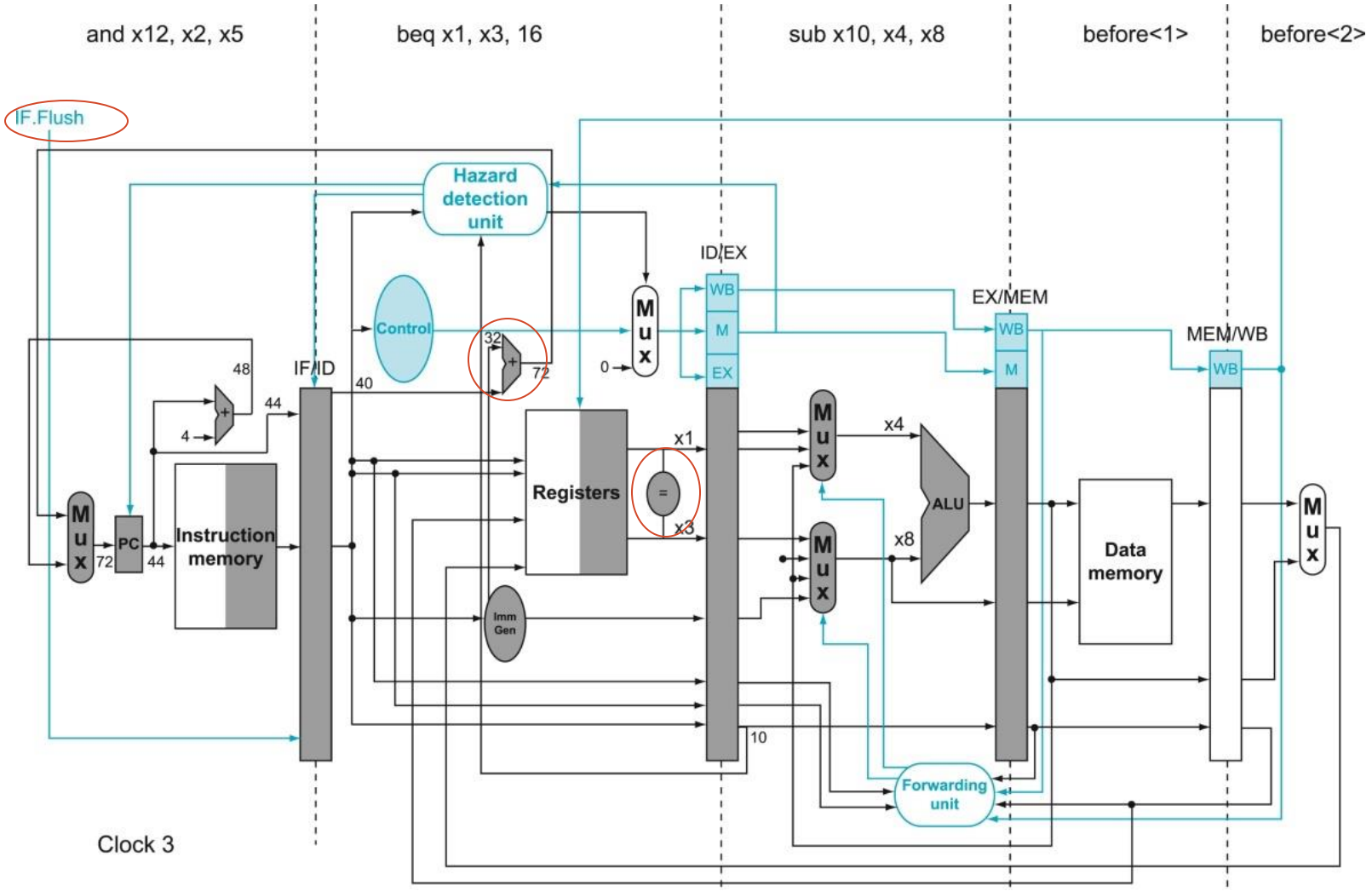
Branch Decision at the ID Stage



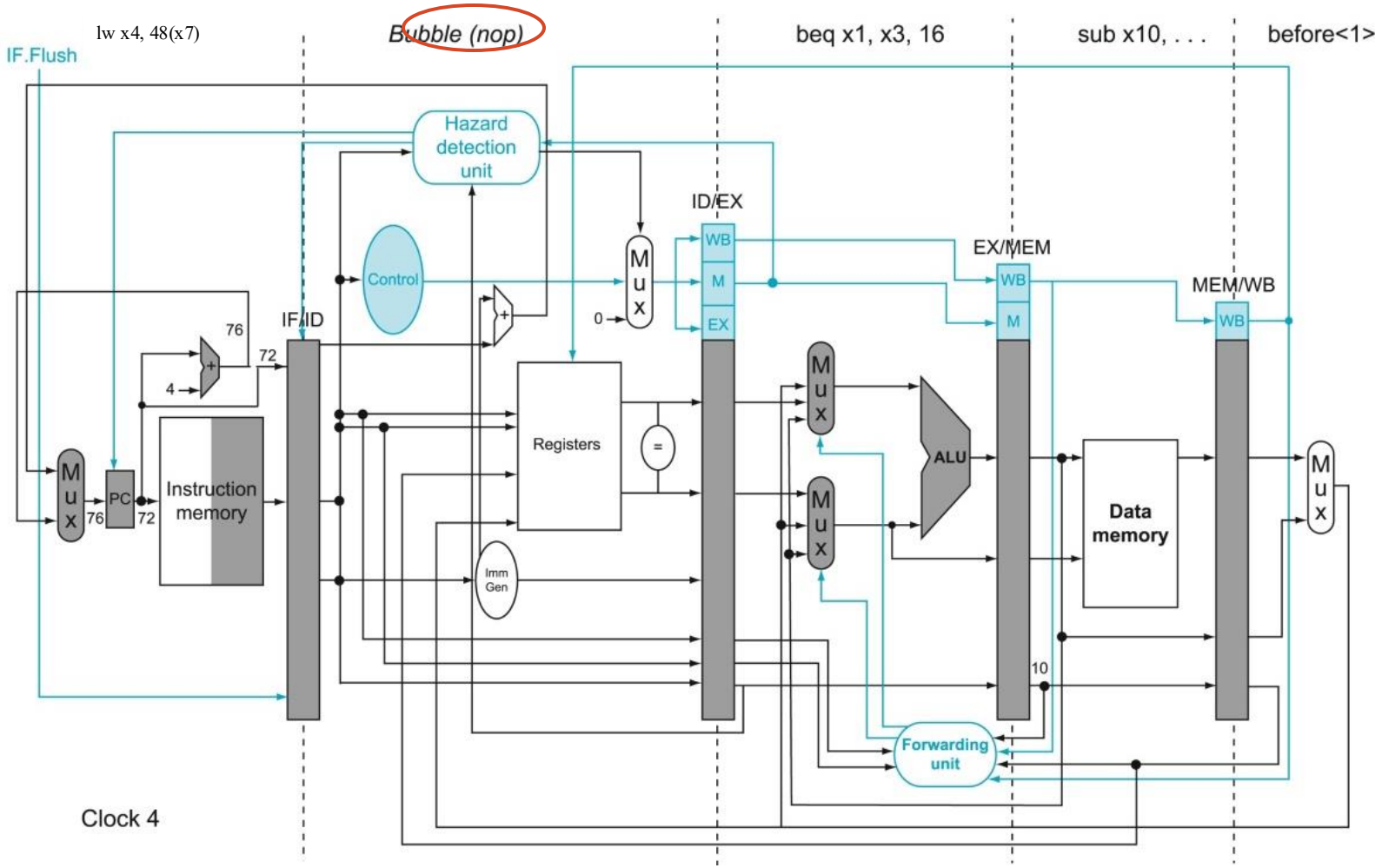
Example: Branch Taken

```
36:  sub  x10, x4, x8
40:  beq  x1,  x3, 16   # 16 is the value in the immediate
44:  and  x12, x2, x5
48:  or   x13, x2, x6
52:  add  x14, x4, x2
56:  slt  x15, x6, x7
     ...
72:  lw   x4, 48(x7)
```

Example: Branch Taken

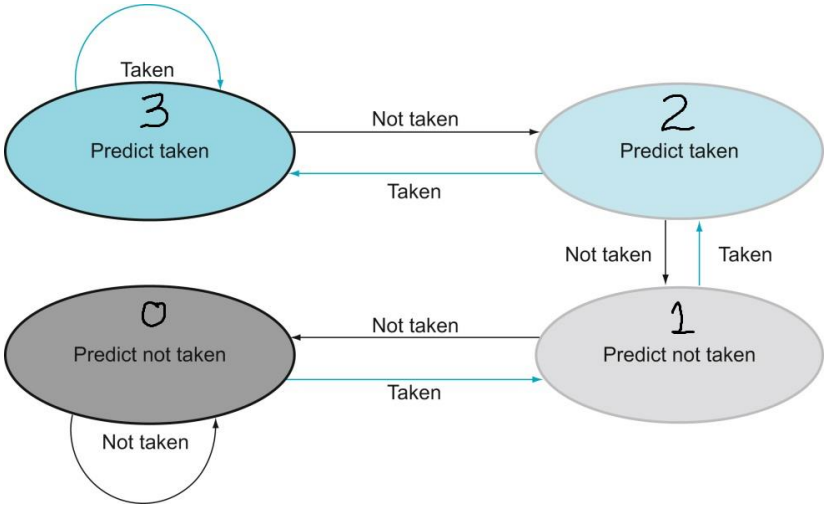


Example: Branch Taken



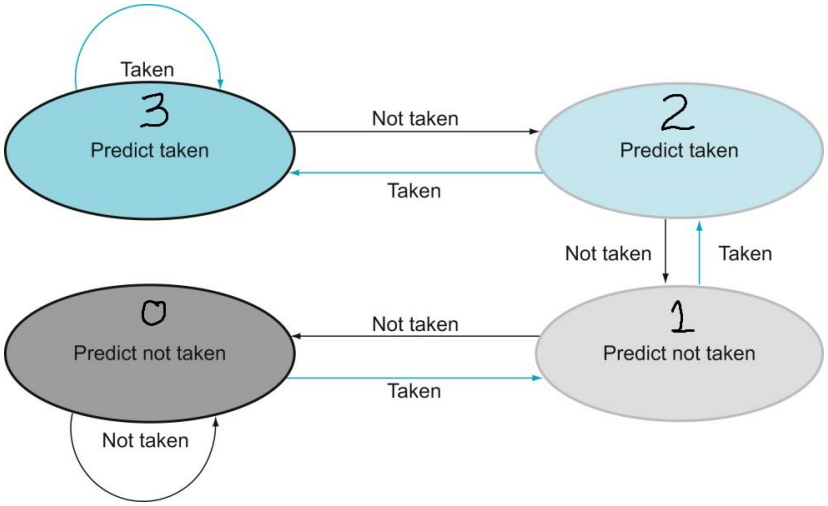
Dynamic Branch Prediction

- **Static branch prediction is crude!**
- **Take history into consideration**
  - If a branch was taken last time, then fetch the new instruction from the same place
  - Branch prediction buffer
    - contains a bit (or bits) which tells whether the branch was recently taken or not
  - Is the prediction correct? Any bad effect?
  - 1-bit prediction scheme
  - 2-bit prediction scheme



Exercise

- **Consider a loop branch that branches nine times in a row, then is not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?**
  - 1-bit prediction?
  - 2-bit prediction?



- Actual: T, T …, T, NT
- 1-bit prediction:

NT, T…T, T (80% accuracy)

- 2-bit prediction?

NT, NT, T…T, T (70% accuracy)

Exceptions

- **Exceptions: events other than branch or jump that change the normal flow of instruction**
  - Arithmetic overflow, undefined instruction, etc
  - Internal of the processor
  - Interrupts from external – I/O interrupts

- **Use arithmetic overflow as an example**
  - When an overflow is detected, we need to transfer control to the exception handling routine immediately
    - RISC-V uses location `0x 1C09 0000`
    - Detected in the EX stage
  - Similar idea as branch hazard
    - De-assert all control signals in EX and ID stages, flush IF/ID

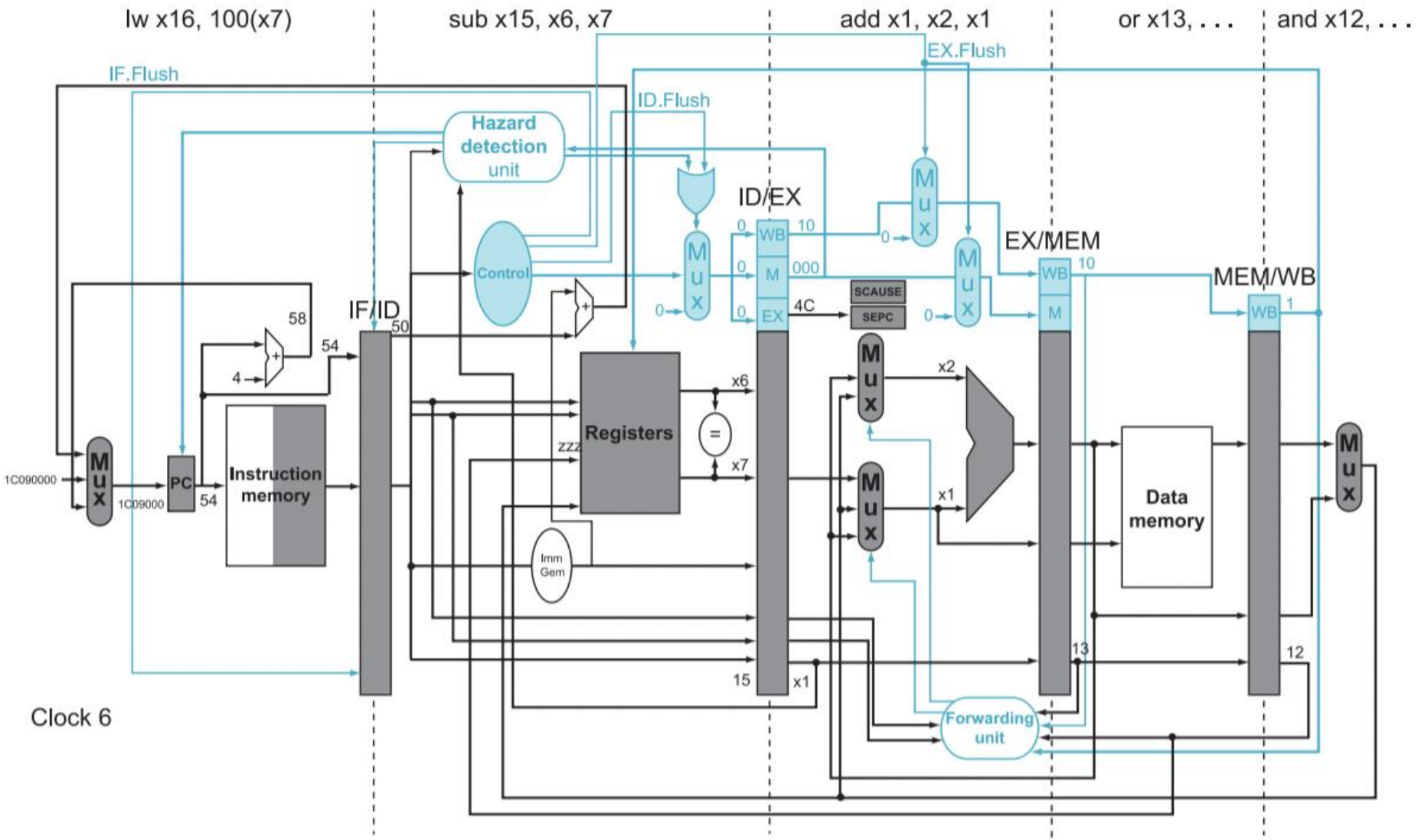
Example

```
sub    x11, x2, x4
and    x12, x2, x5
or     x13, x2, x6
add    x1, x2, x1          -- overflow occurs
sub    x15, x6, x7
lw     x16, 100(x7)
```

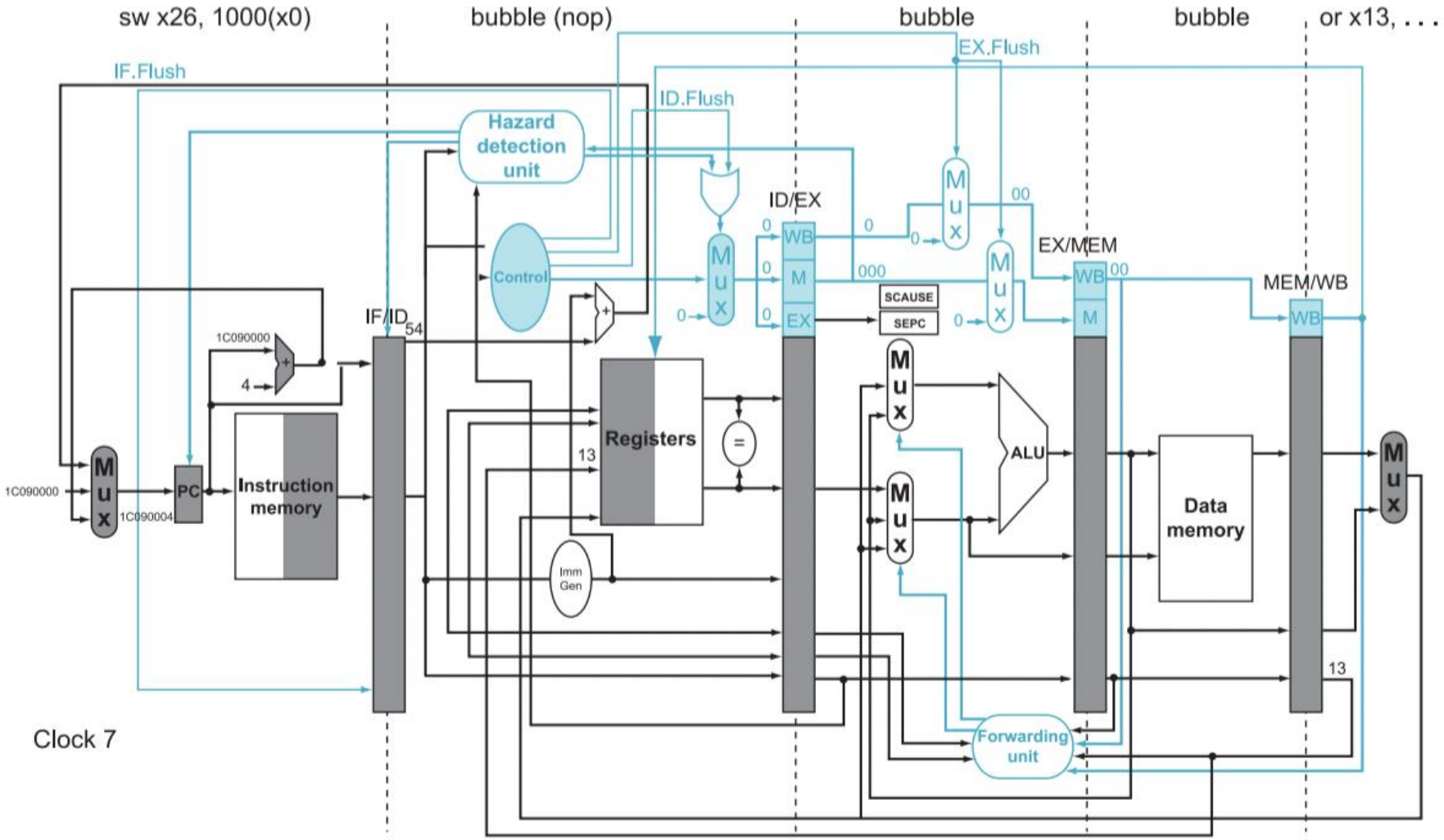
Exceptions handling routine:

```
0x 1C09 0000    sw    x25, 1000(x0)
0x 1C09 0004    sw    x26, 1004(x0)
```

Clock 6


lw x16, 100(x7)
sub x15, x6, x7
add x1, x2, x1
or x13, . . .
and x12, . . .
IF.Flush
EX.Flush
ID.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
IF/ID
WB
M
EX
SCAUSE
SEPC
4C
000
Registers
Instruction memory
PC
Imm Gem
Data memory
Forwarding unit
Mux
x1
x2
x6
x7
zzz
1C090000
1C09000
Clock 6

Clock 7


sw x26, 1000(x0)
bubble (nop)
bubble
bubble
or x13, . . .
IF.Flush
ID.Flush
EX.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
WB
M
EX
SCAUSE
SEPC
IF/ID
54
1C090000
4
PC
1C090004
Instruction memory
Registers
13
Imm Gen
ALU
Data memory
Forwarding unit
Clock 7

Performance of Pipelined Datapath

- **Functional unit time**
  - 200ps for memory access
  - 100ps for ALU operation
  - 50ps for register file access
- **Instruction frequencies**
  - 25% loads,
  - 10% stores,
  - 45% R-type
  - 20% branches,
- **For piplelined datapath, compute average CPI and CCT**
  - 50% of load are immediately followed by an instruction that uses the result
  - Branch delay on misprediction is 1 clock cycle and 25% branches are mispredicted
  - Assume branch condition evaluation is done in the ID stage

Solution

- **For pipelined design**
  - CPI for lw: 1x0.5+2x0.5=1.5
  - CPI for branch: 1x0.75+2x0.25=1.25
  - Average CPI= 1.5*25% + 1*10% + 1*45%+1.25*20% =1.175
  - CCT= 200ps

Example 1

Consider the following code:

```
        addi x10, x0, 2
loop:   addi x10, x10, -1
        addi x9, x10, -1
        bne x10, x0, loop
        add x11, x10, x9
```

For all parts, assume forwarding

a) How many cycles does this code take if there is no branch prediction, and branches are resolved (i.e., we know if they are taken or not taken) at the end of the ID stage?

b) How many cycles does this code take if branches are predicted not taken, and branches are resolved at the end of the ID stage?

c) How many cycles does this code take if branches are predicted not taken, and branches are resolved at the end of the MEM stage?

Example 1a Answer

a) **14**

Ins	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11	CC12	CC13	CC14
I1	IF	ID	EX	M	WB									
I2		IF	ID	EX	M	WB								
I3			IF	ID	EX	M	WB							
I4				IF	ID	EX	M							
I2						IF	ID	EX	M	WB				
I3							IF	ID	EX	M	WB			
I4								IF	ID	EX	M			
I5										IF	ID	EX	M	WB

Example 1b Answer

b) 13

The second branch's stall will now be avoided as I'm predicting not taken and predicting correct. So, execution time = 14 - 1 = 13 cycles.

Example 1c Answer

c) 15

The second branch will not cause any stalls as it is a

Ins	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	10	11	12	13	14	15
I1	IF	ID	EX	M	WB										
I2		IF	ID	EX	M	WB									
I3			IF	ID	EX	M	WB								
I4				IF	ID	EX	M								
I2								IF	ID	EX	M	WB			
I3									IF	ID	EX	M	WB		
I4										IF	ID	EX	M		
I5											IF	ID	EX	M	WB

Example 2

- Schedule the following code for the pipelined processor with forwarding and reduced branch delay using a single branch delay slot to minimize stall cycles:

```
loop: lw x1,0(x2)        #  x1 array element
      add x1, x1, x3     # add constant in x3
      sw x1,0(x2)        # store result array element
      addi x2, x2, -4    # decrement address by 4
      bne x2, x4, loop   # branch if x2 != x4
```

- Assuming the initial value of x2 = x4 + 40 (i.e., it loops 10 times); assuming condition is evaluated at ID stage
- What is the CPI and total number of cycles needed to run the code with and without compiler scheduling?

Example 2 Solution

- Without compiler scheduling

```
loop:  lw x1,0(x2)
       stall
       add x1, x1, x3
       sw x1,0(x2)
       addi x2, x2, -4
       stall
       bne x2, x4, loop
       stall
```

Ignoring the initial 4 cycles to fill the pipeline:

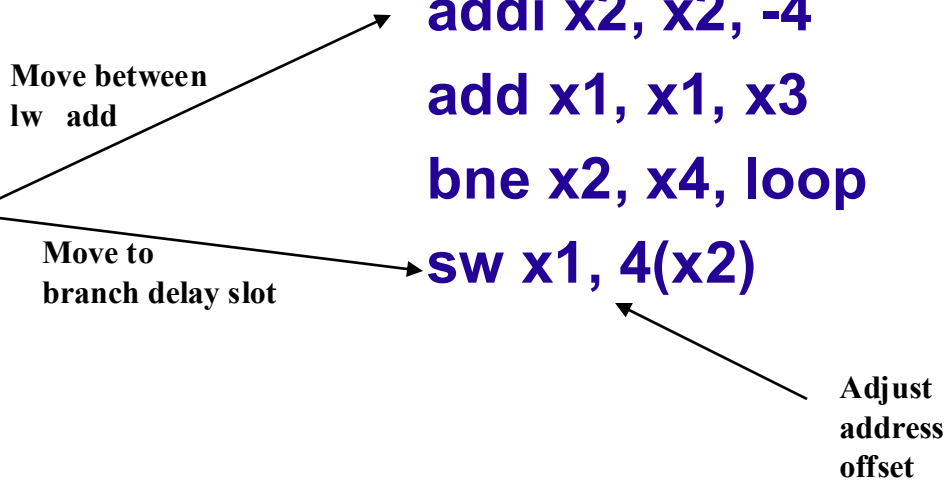
Each iteration takes = 8 cycles

CPI = 8/5 = 1.6

Total cycles = 8 x 10 = 80 cycles

- With compiler scheduling

```
loop:  lw x1,0(x2)
       addi x2, x2, -4
       add x1, x1, x3
       bne x2, x4, loop
       sw x1, 4(x2)
```



Ignoring the initial 4 cycles to fill the pipeline:

Each iteration takes = 5 cycles

CPI = 5/5 = 1

Total cycles = 5 x 10 = 50 cycles

Speedup = 80/ 50 = 1.6