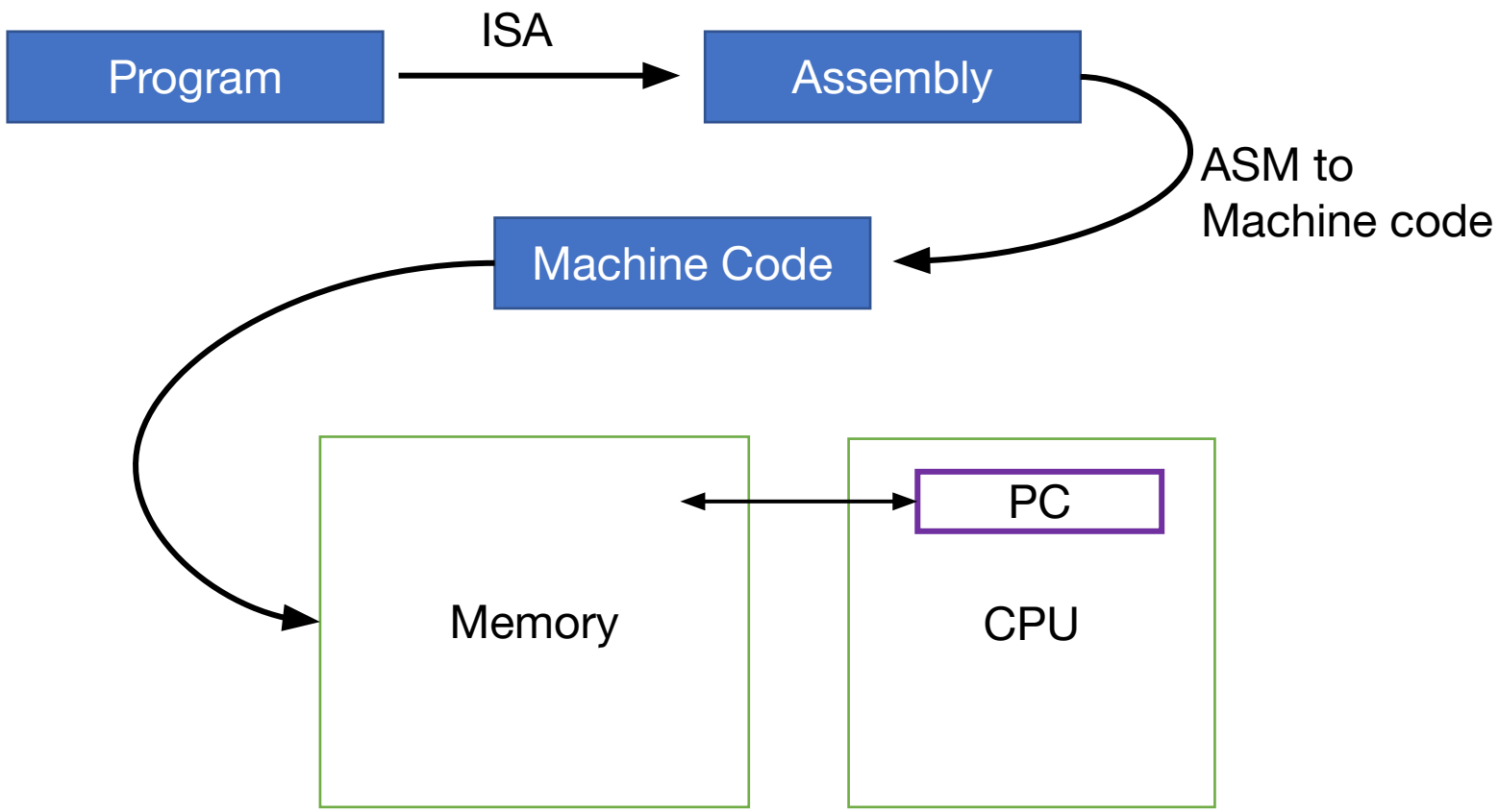
Module 2 – Preliminaries

(ECE 209) Secure and Advanced Computer Architecture

Nader Sehatbakhsh
Department of Electrical and Computer Engineering
University of California, Los Angeles

Stored-Program Computer



Role of the ISA

- First step in the design process!

Role of the ISA

- First step in the design process!
- A contract between software and hardware
  - Abstracting things away!
  - Software and hardware can be developed separately. One doesn't need to care about the internals as long as all components adhere to ISA.

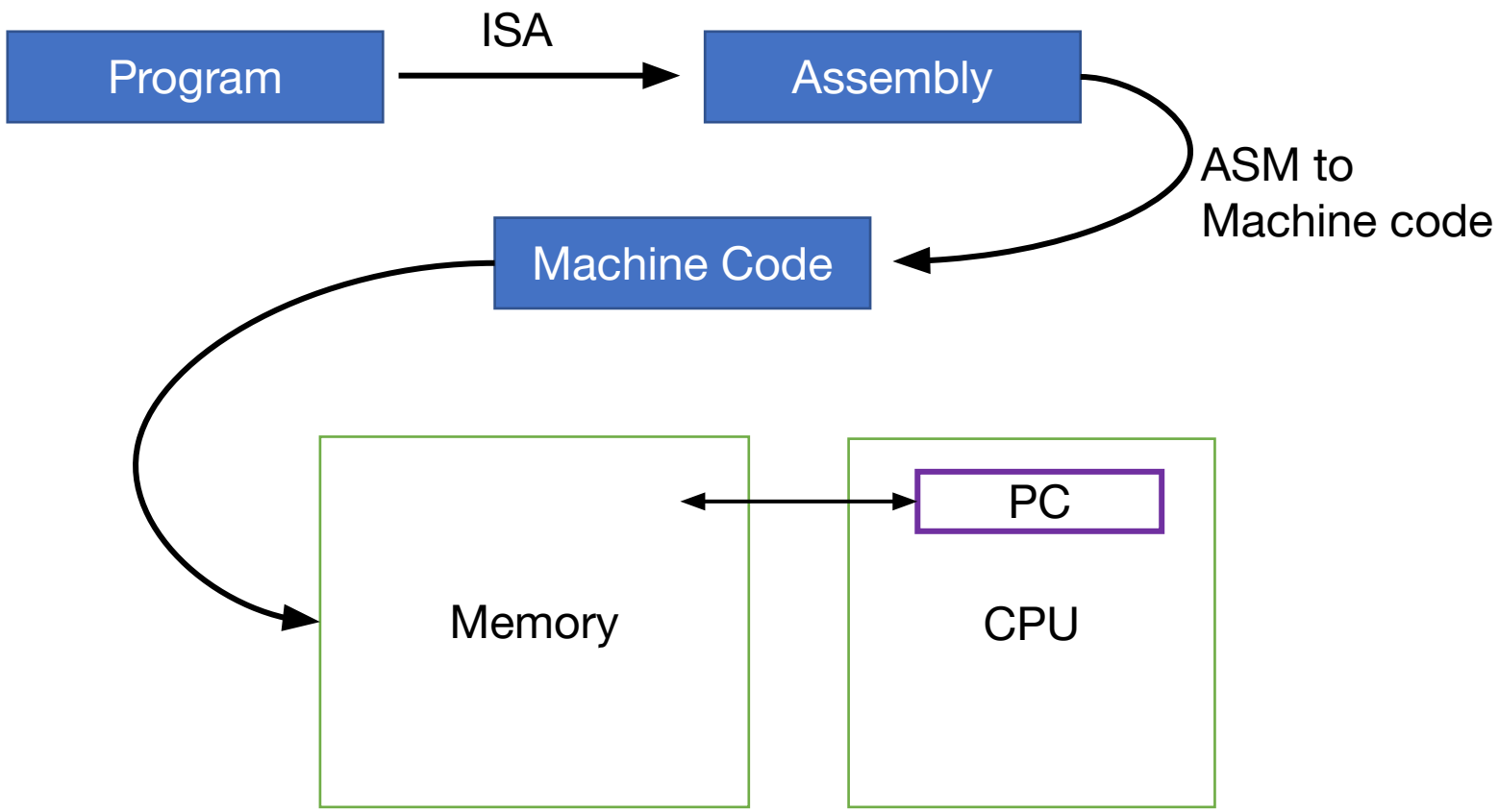
RISC-V ISA

- Fixed size, RISC based
- Simple structure
  - Command
  - Operand(s) - source and destination → they can be registers, memory, or numbers (immediates)

RISC-V ISA

Instruction	Syntax	Description	Execution
LUI	`lui rd, luiConstant`	Load Upper Immediate	`reg[rd] <= luiConstant « 12`
JAL	`jal rd, label`	Jump and Link	`reg[rd] <= pc + 4` `pc <= label`
JALR	`jalr rd, offset(rs1)`	Jump and Link Register	`reg[rd] <= pc + 4` `pc <= {(reg[rs1] + offset)[31:1], 1'b0}`
BEQ	`beq rs1, rs2, label`	Branch if =	`pc <= (reg[rs1] == reg[rs2]) ? label: pc + 4`
BNE	`bne rs1, rs2, label`	Branch if ≠	`pc <= (reg[rs1] != reg[rs2]) ? label: pc + 4`
BLT	`blt rs1, rs2, label`	Branch if < (Signed)	`pc <= (reg[rs1] <s reg[rs2]) ? label: pc + 4`
BGE	`bge rs1, rs2, label`	Branch if ≥ (Signed)	`pc <= (reg[rs1] >=s reg[rs2]) ? label: pc + 4`
BLTU	`bltu rs1, rs2, label`	Branch if < (Unsigned)	`pc <= (reg[rs1] <u reg[rs2]) ? label: pc + 4`
BGEU	`bgeu rs1, rs2, label`	Branch if ≥ (Unsigned)	`pc <= (reg[rs1] >=u reg[rs2]) ? label: pc + 4`
LB	`lb rd, offset(rs1)`	Load Byte	`reg[rd] <= signExtend(mem[addr])`
LH	`lh rd, offset(rs1)`	Load Half Word	`reg[rd] <= signExtend(mem[addr + 1: addr])`
LW	`lw rd, offset(rs1)`	Load Word	`reg[rd] <= mem[addr + 3: addr]`
LBU	`lbu rd, offset(rs1)`	Load Byte (Unsigned)	`reg[rd] <= zeroExtend(mem[addr])`
LHU	`lhu rd, offset(rs1)`	Load Half Word (Unsigned)	`reg[rd] <= zeroExtend(mem[addr + 1: addr])`
SB	`sb rs2, offset(rs1)`	Store Byte	`mem[addr] <= reg[rs2][7:0]`
SH	`sh rs2, offset(rs1)`	Store Half Word	`mem[addr + 1: addr] <= reg[rs2][15:0]`
SW	`sw rs2, offset(rs1)`	Store Word	`mem[addr + 3: addr] <= reg[rs2]`
ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <s constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <u constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, shamt`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « shamt`
SRLI	`srli rd, rs1, shamt`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »u shamt`
SRAI	`srai rd, rs1, shamt`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »s shamt`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2][4:0]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <s reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <u reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »u reg[rs2][4:0]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »s reg[rs2][4:0]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

Stored-Program Computer



ISA to Machine Code

- Using a standard table

RV32I Base Instruction Set (MIT 6.004 subset)

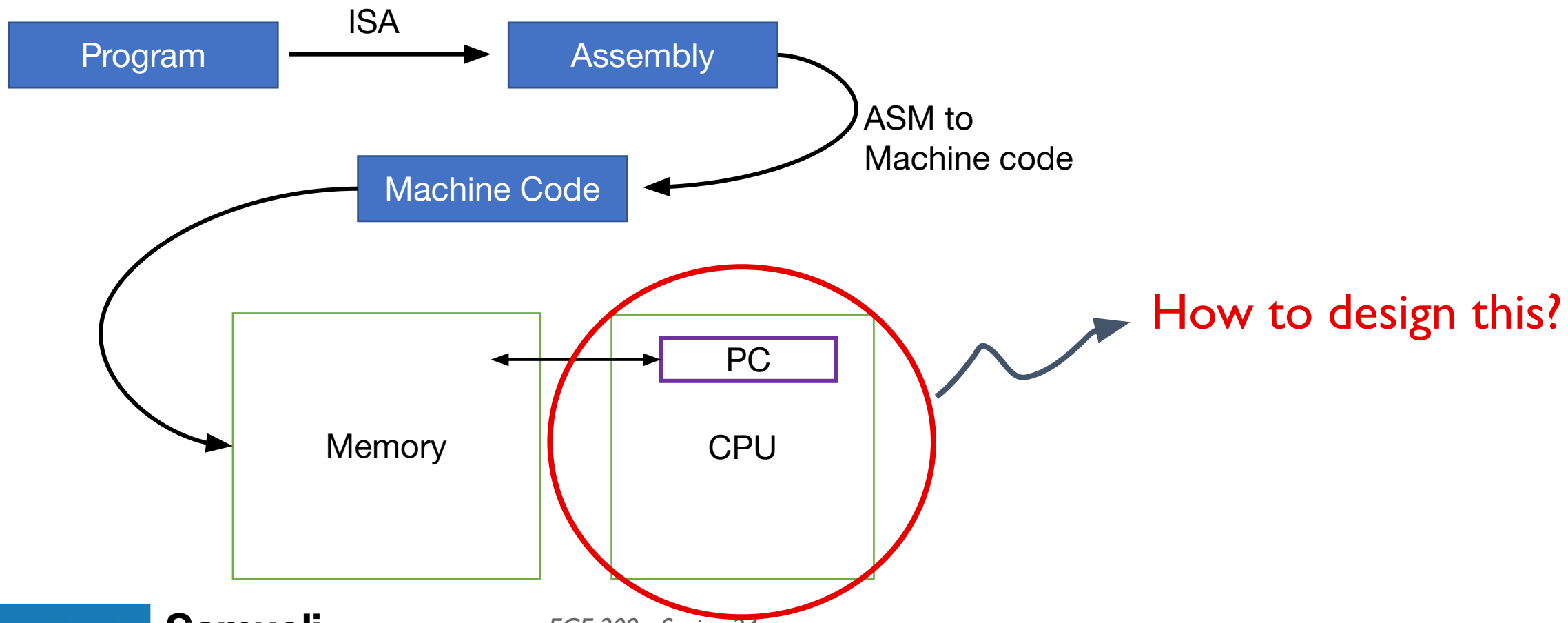
imm[31:12]				rd	0110111	LUI
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

Why this table?

- Various things to consider:
  - Size
  - Decoding complexity and speed

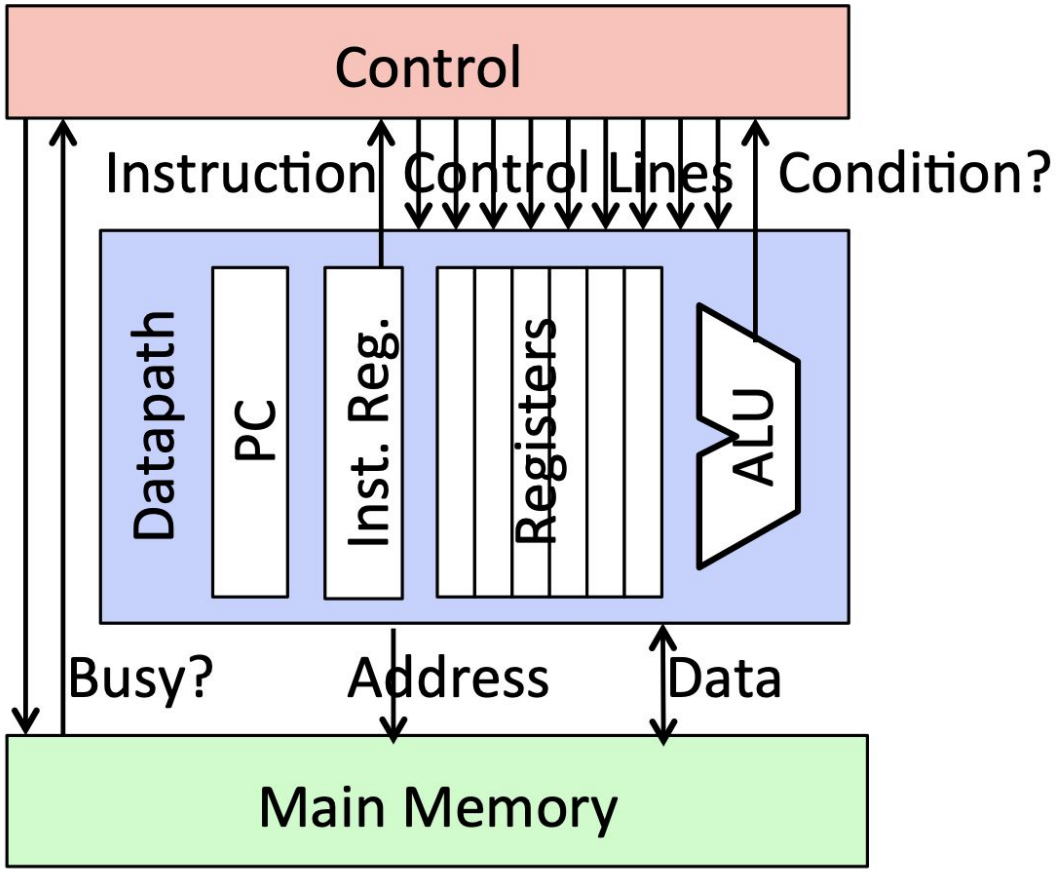
Execution Model



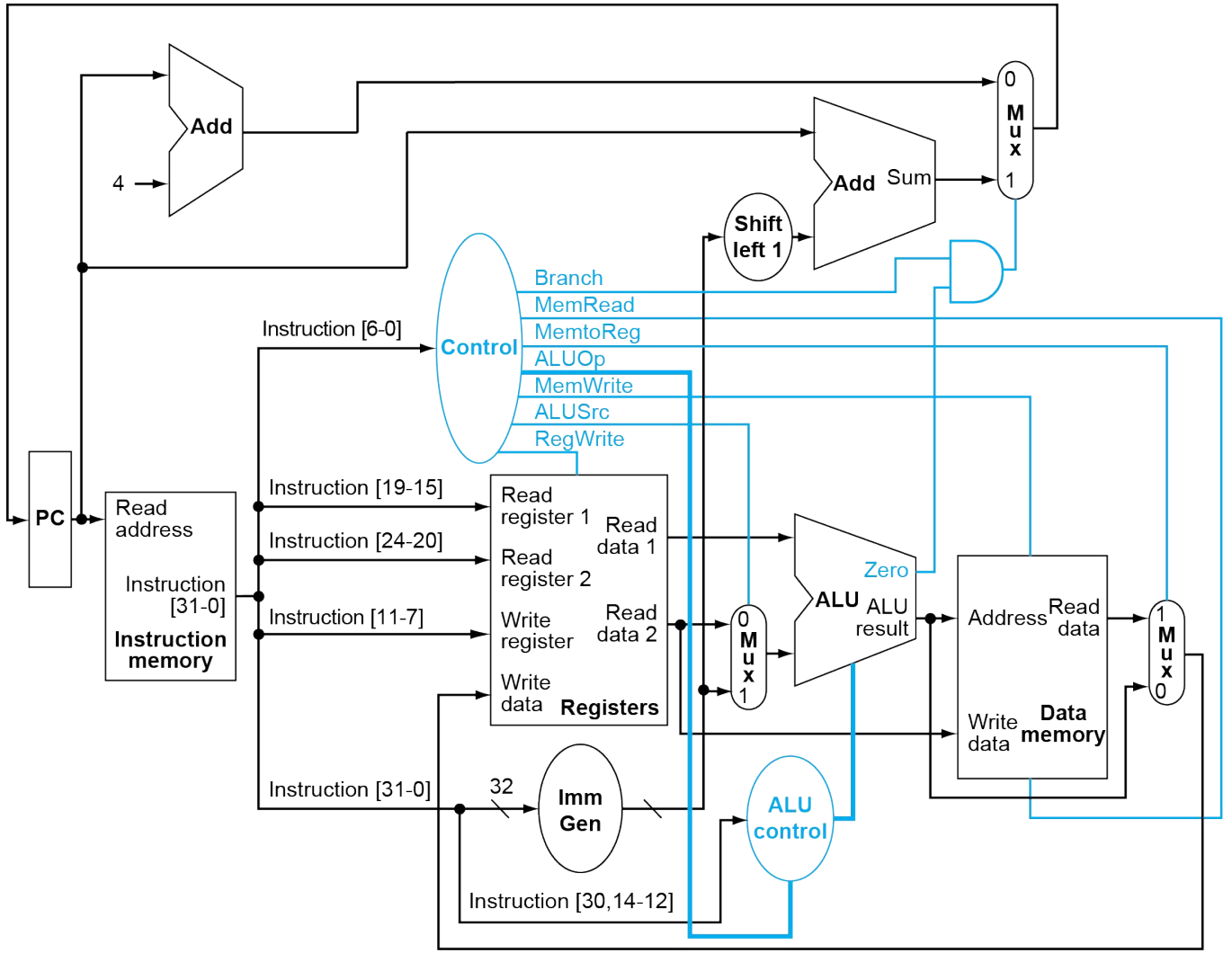
Microarchitecture

- Specific implementation of ISA.
- One ISA might have many different microarchitecture implementations.
- Things to consider:
  - Performance
  - Power
  - Area
  - Security????

Datapath + Controller

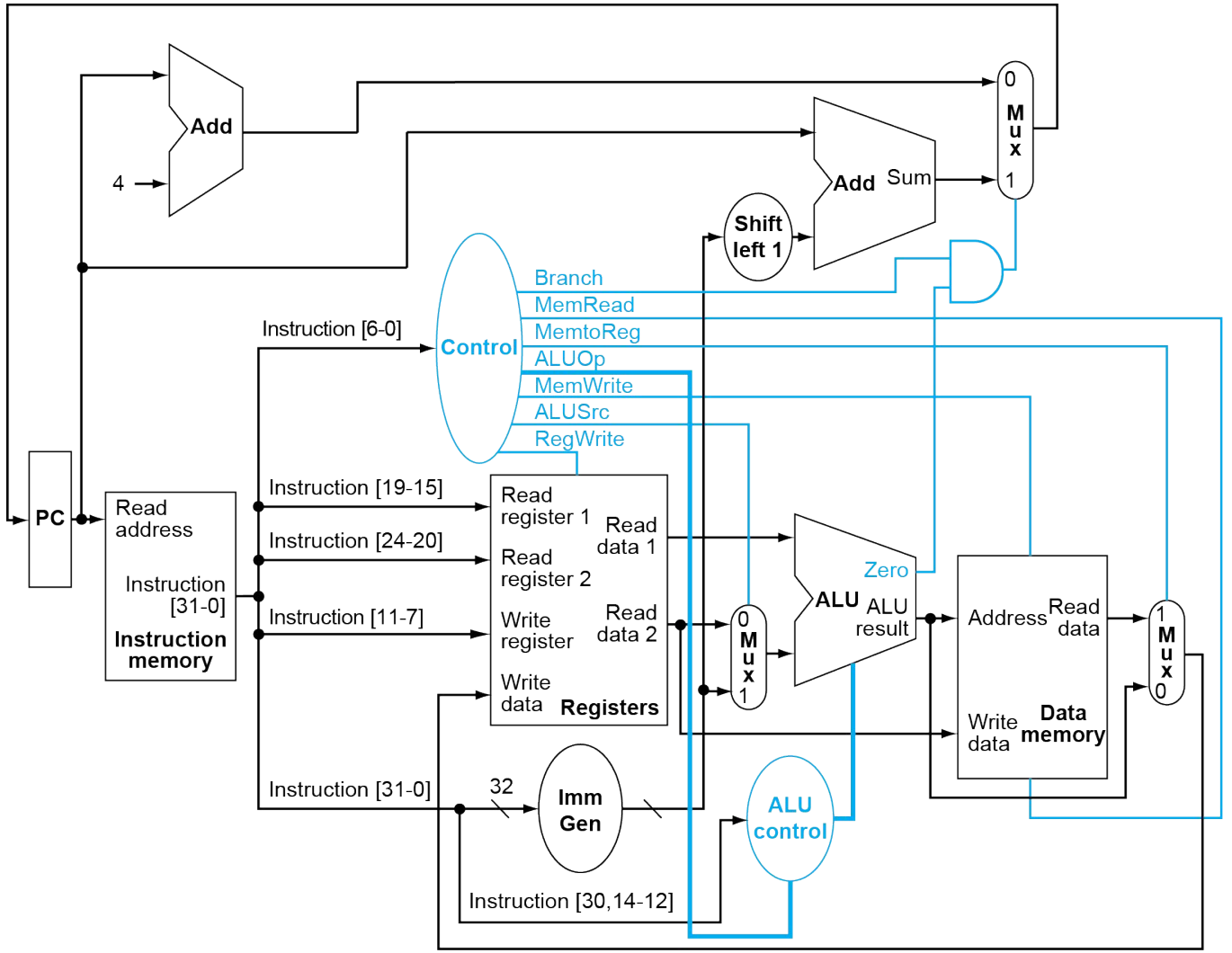


Design


Add
4
Add
Sum
0
M
u
x
1
Shift
left 1
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [6-0]
PC
Read
address
Instruction
[31-0]
Instruction
memory
Instruction [19-15]
Instruction [24-20]
Instruction [11-7]
Read
register 1
Read
register 2
Write
register
Write
data
Read
data 1
Read
data 2
Registers
Zero
ALU
ALU
result
Address
Read
data
Write
data
Data
memory
1
M
u
x
0
Instruction [31-0]
32
Imm
Gen
ALU
control
Instruction [30,14-12]

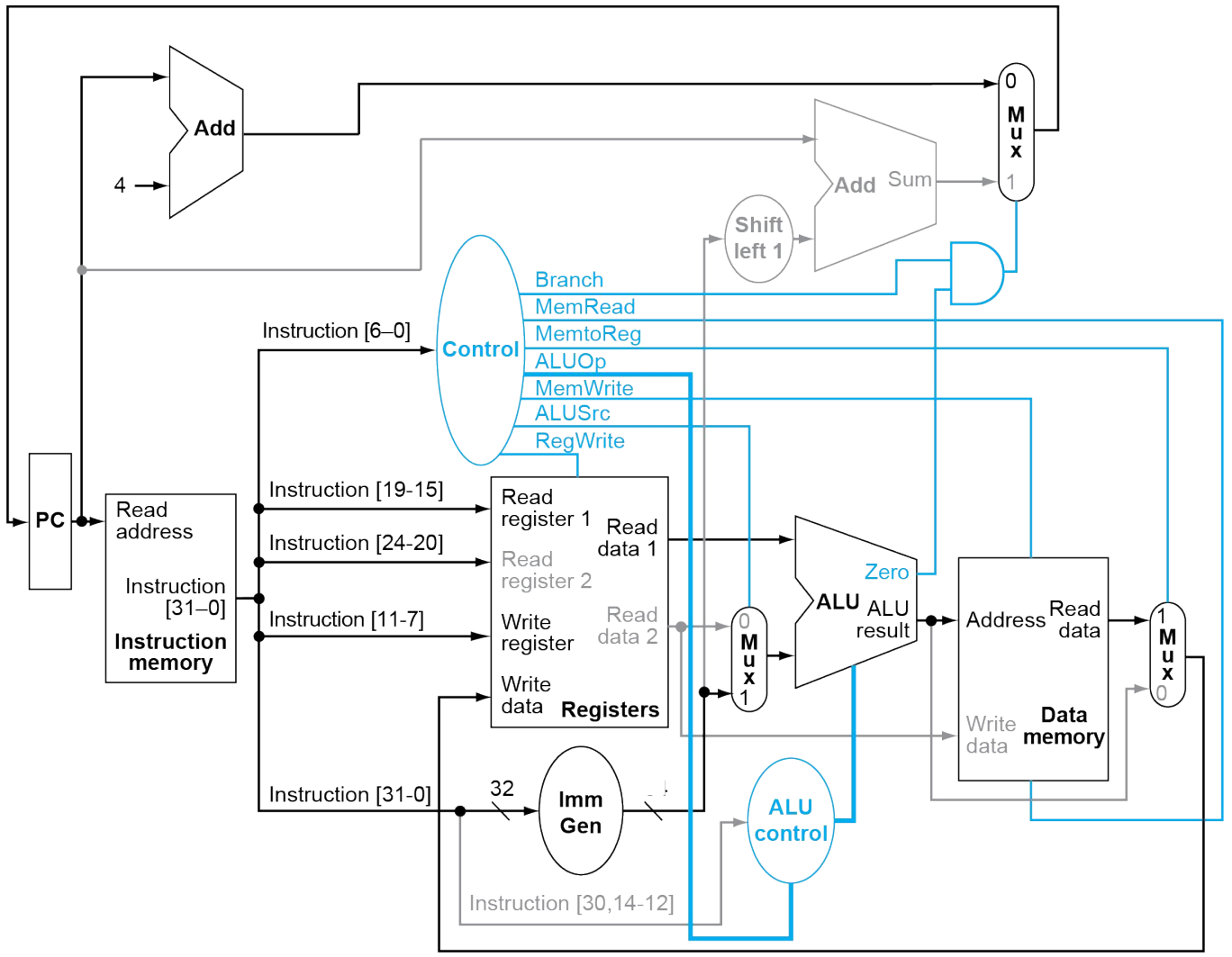
Design

Adding new components for new instructions?



Dataflow

LW



*Images were taken from Hennessy Patterson Book [1].

Main Metric?

- What is the most important metric in designing a processor?

Main Metric?

- What is the most important metric in designing a processor?
- Don't be fooled, it's not security!
- How to measure performance?

“Iron Law” of Processor Performance

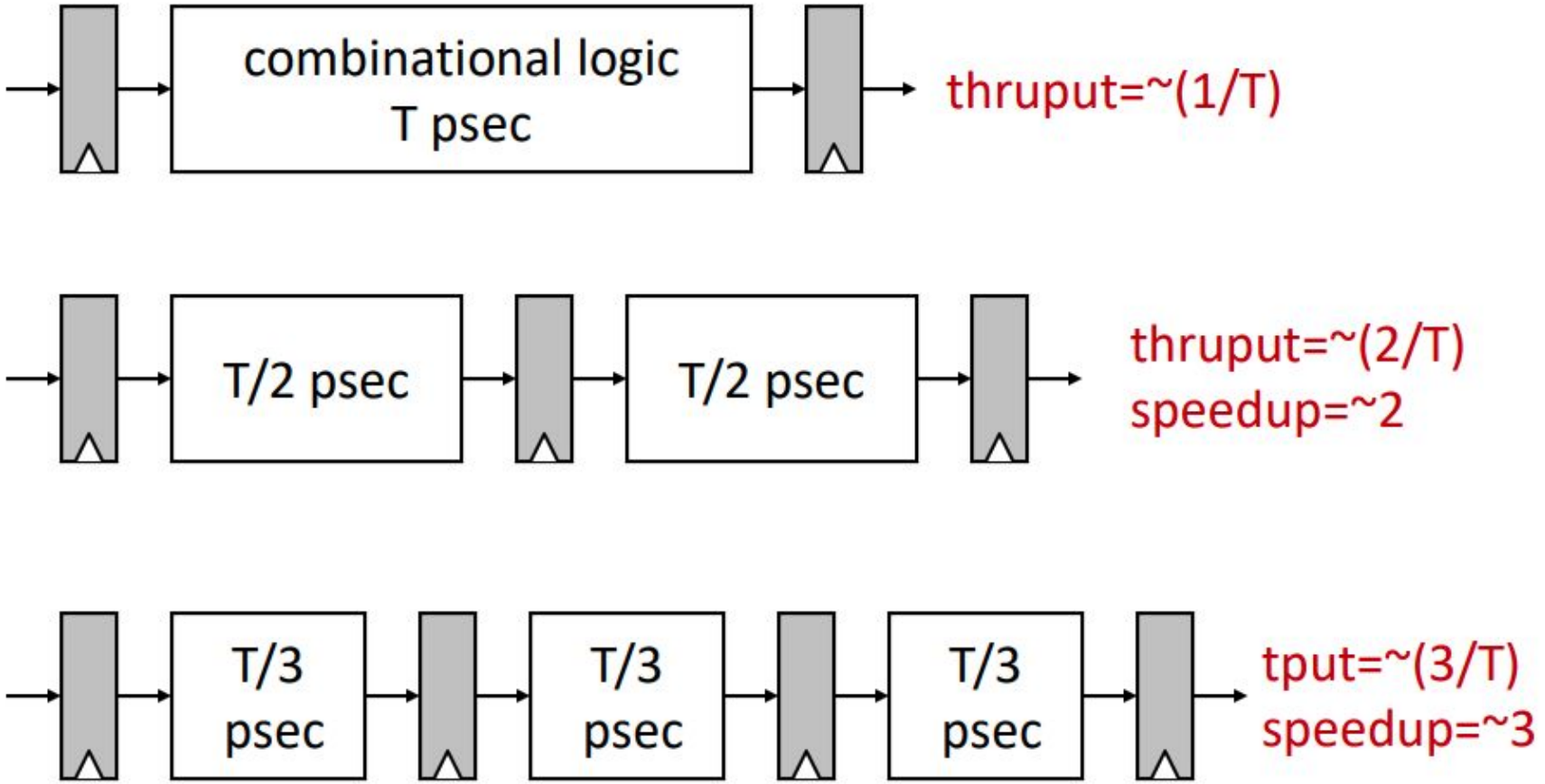
$$CPU\ Time = InstructionCount \times CyclePerInstruction \times CycleTime$$

“Iron Law” of Processor Performance

$$CPU\ Time = InstructionCount \times CyclePerInstruction \times CycleTime$$

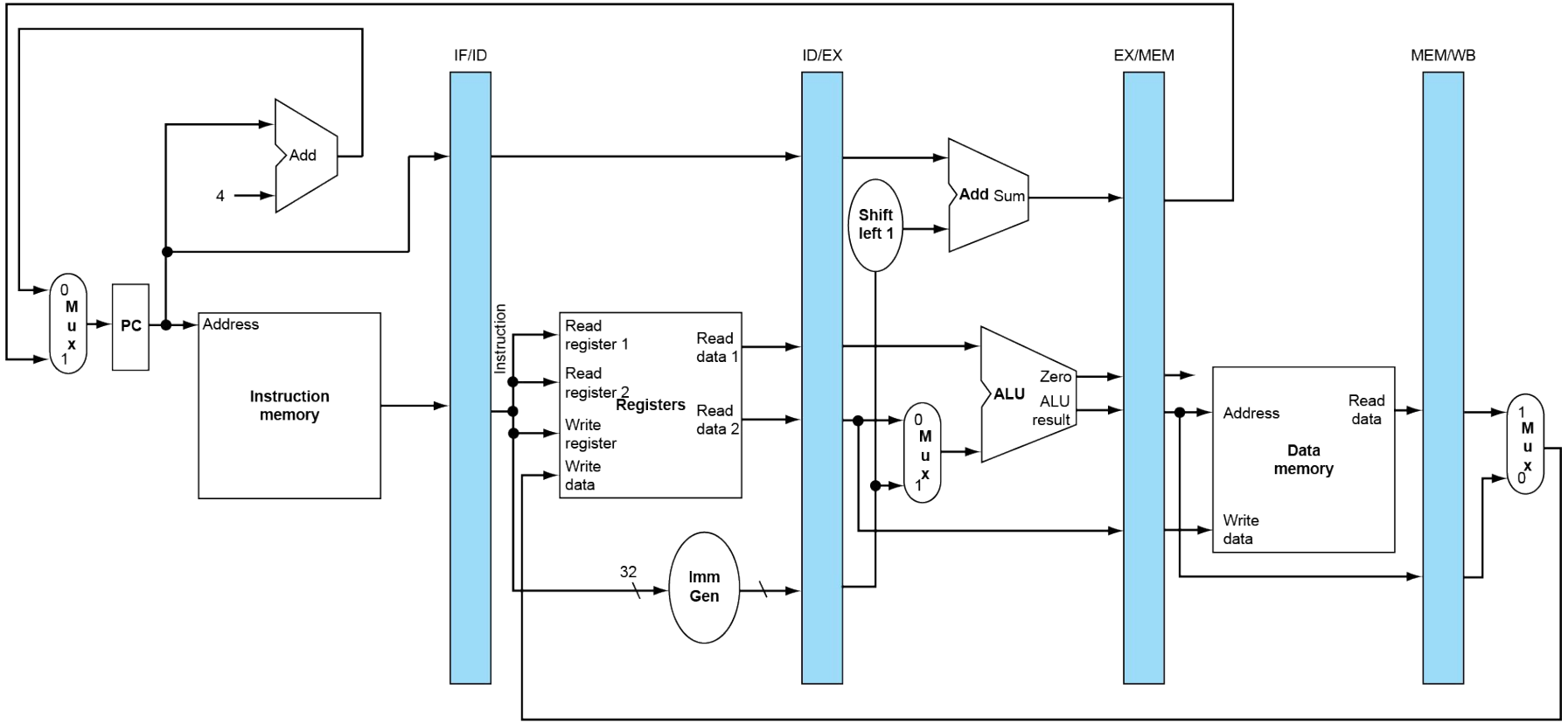
- How to improve each?

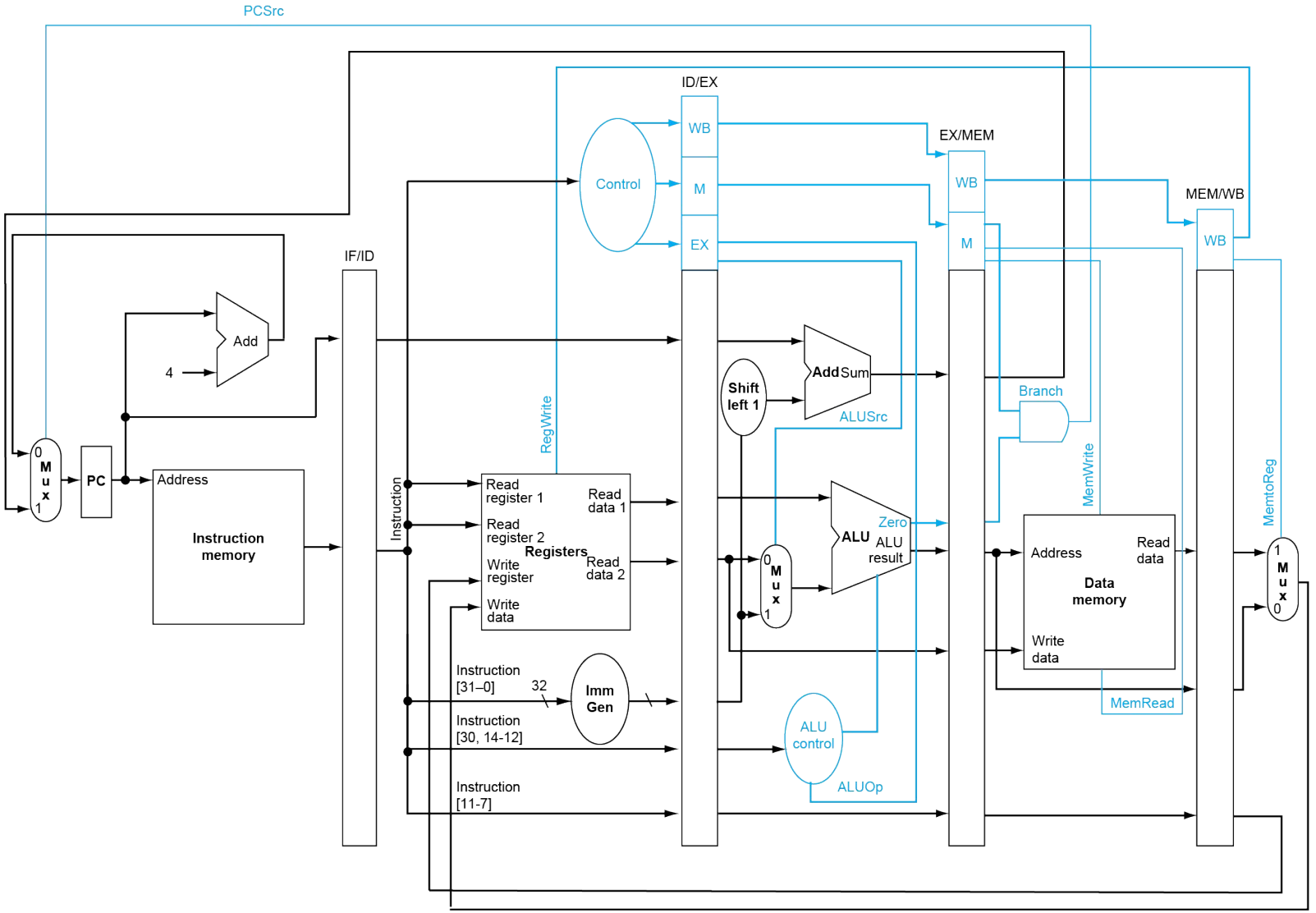
Let's add overlaps!



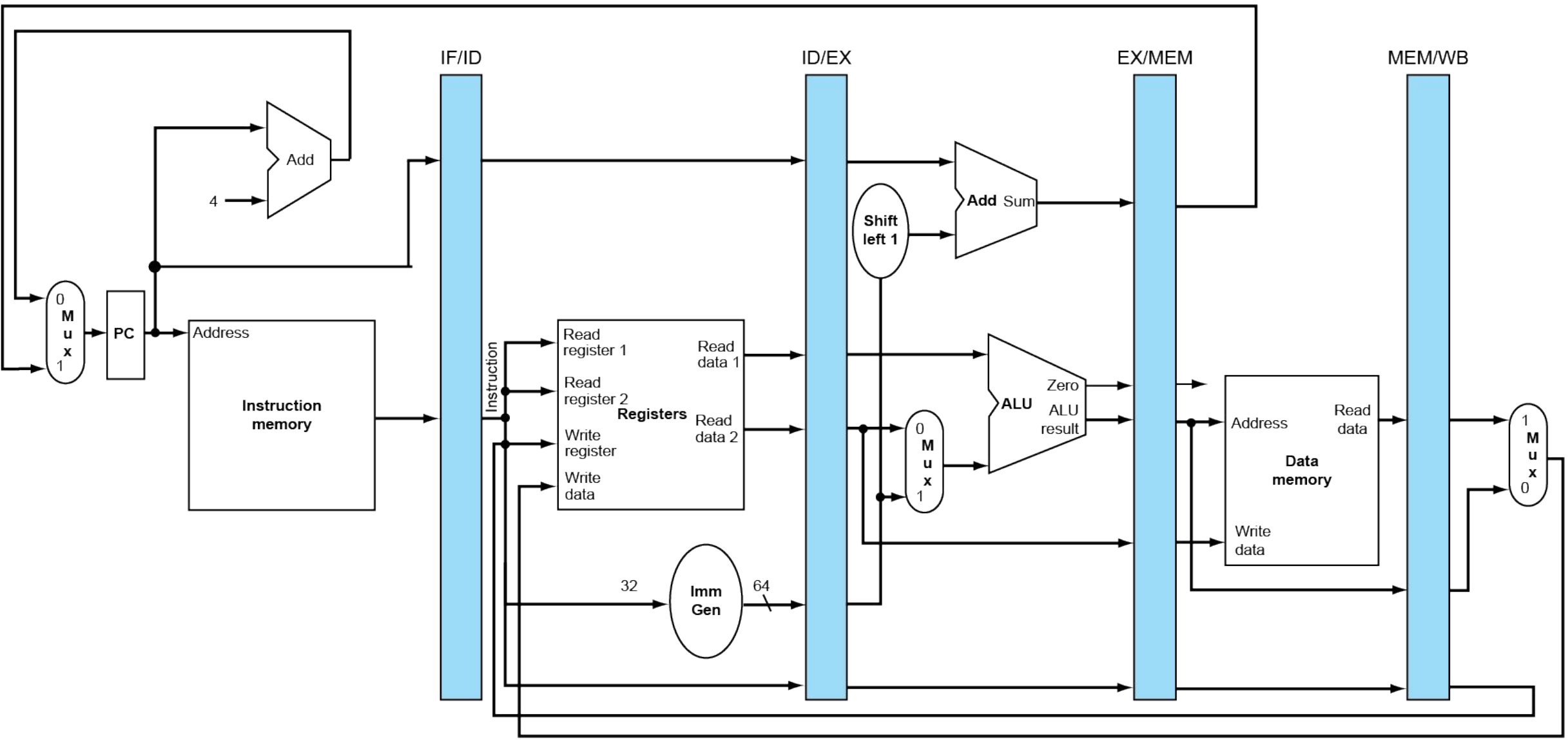
from 18-447,
James Hoe, CMU

Pipeline Design




PCSrc
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
IF/ID
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 1
Add
Sum
ALUSrc
Branch
MemWrite
MemtoReg
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
MemRead
Instruction [31–0]
32
Imm Gen
Instruction [30, 14-12]
Instruction [11-7]
ALU control
ALUOp
1
M
u
x
0

add x14, x5, x6	ld x13, 48(x1)	add x12, x3, x4	sub x11, x2, x3	ld x10, 40(x1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back



How many pipeline stages to add?

How many pipeline stages to add?

- Diminishing returns.

- Remember: there is always a tradeoff!

Now what?

- There is more to it!
- Pipelining causes correctness issues!

Data Hazard

read after write (RAW)

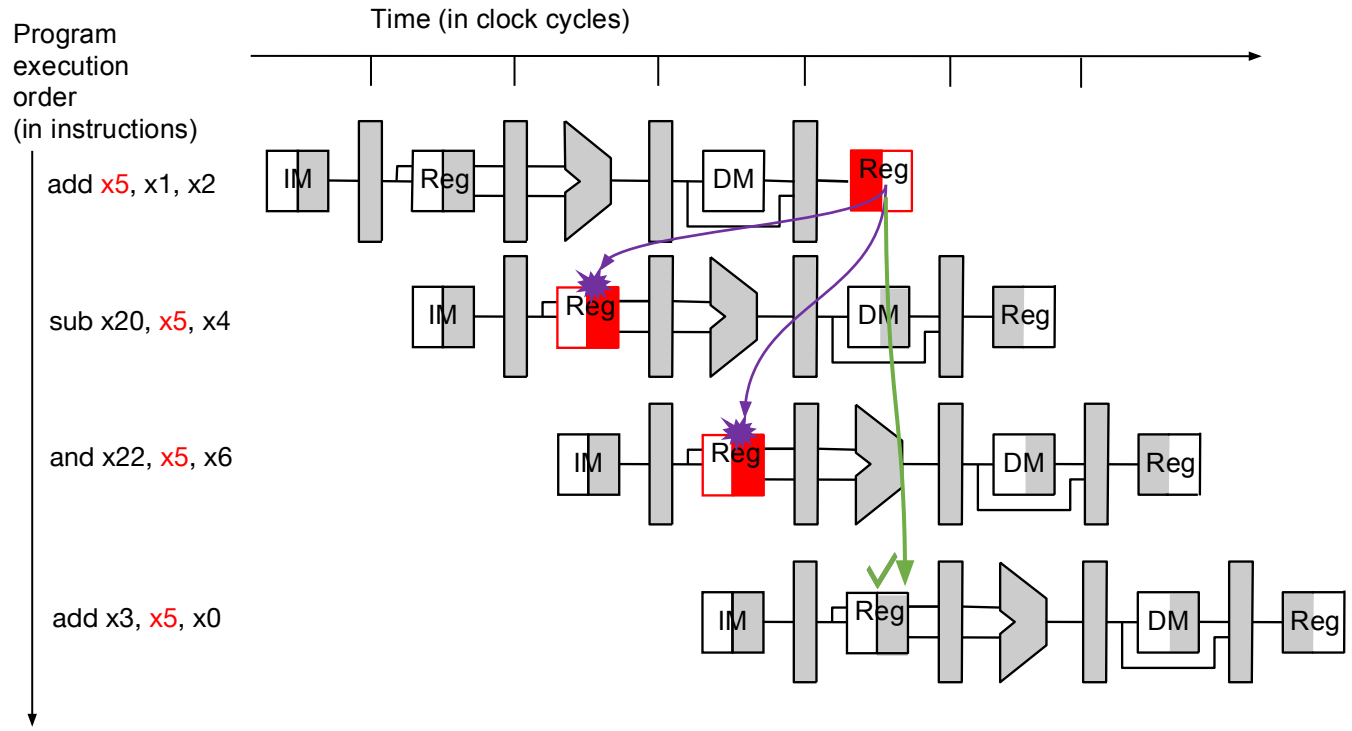
- Writing to a register *(rd)* and using it (*rs1 or rs2*) **before** the writing is finished (i.e., *rd* reaches to the *WB* stage).

```
add x5, x1, x2
sub x20, x5, x4
```

Data Hazard

read after write (RAW)

– For how many cycles we might have RAW hazard?



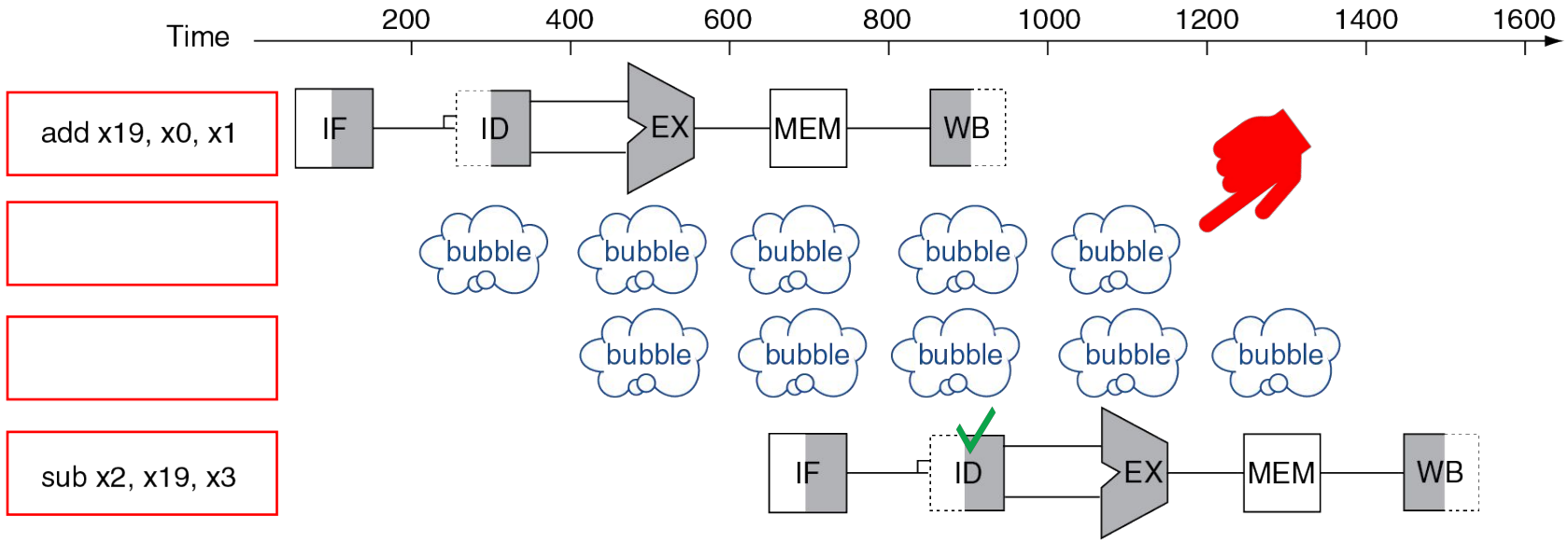
– 2 cycles

How do we fix RAW hazards?

- Two options:
  - Stalling
  - Forwarding

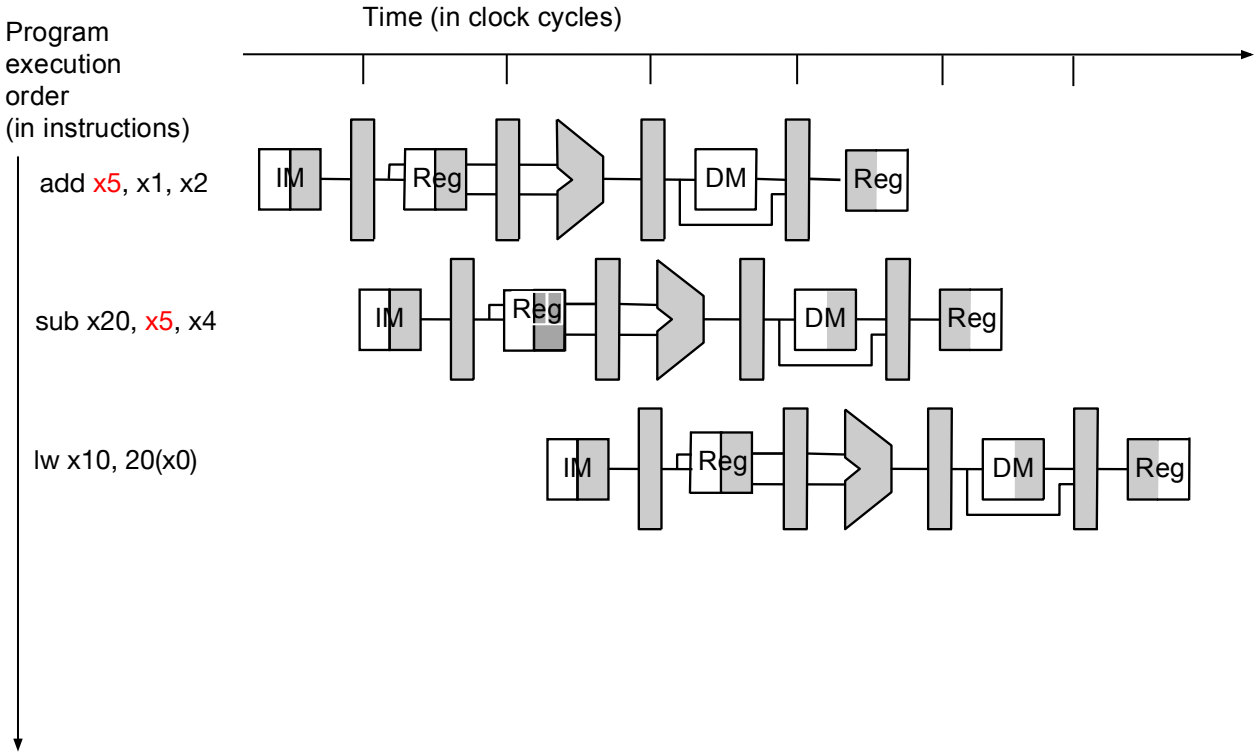
Option 1: Stall

Option 1: Stall



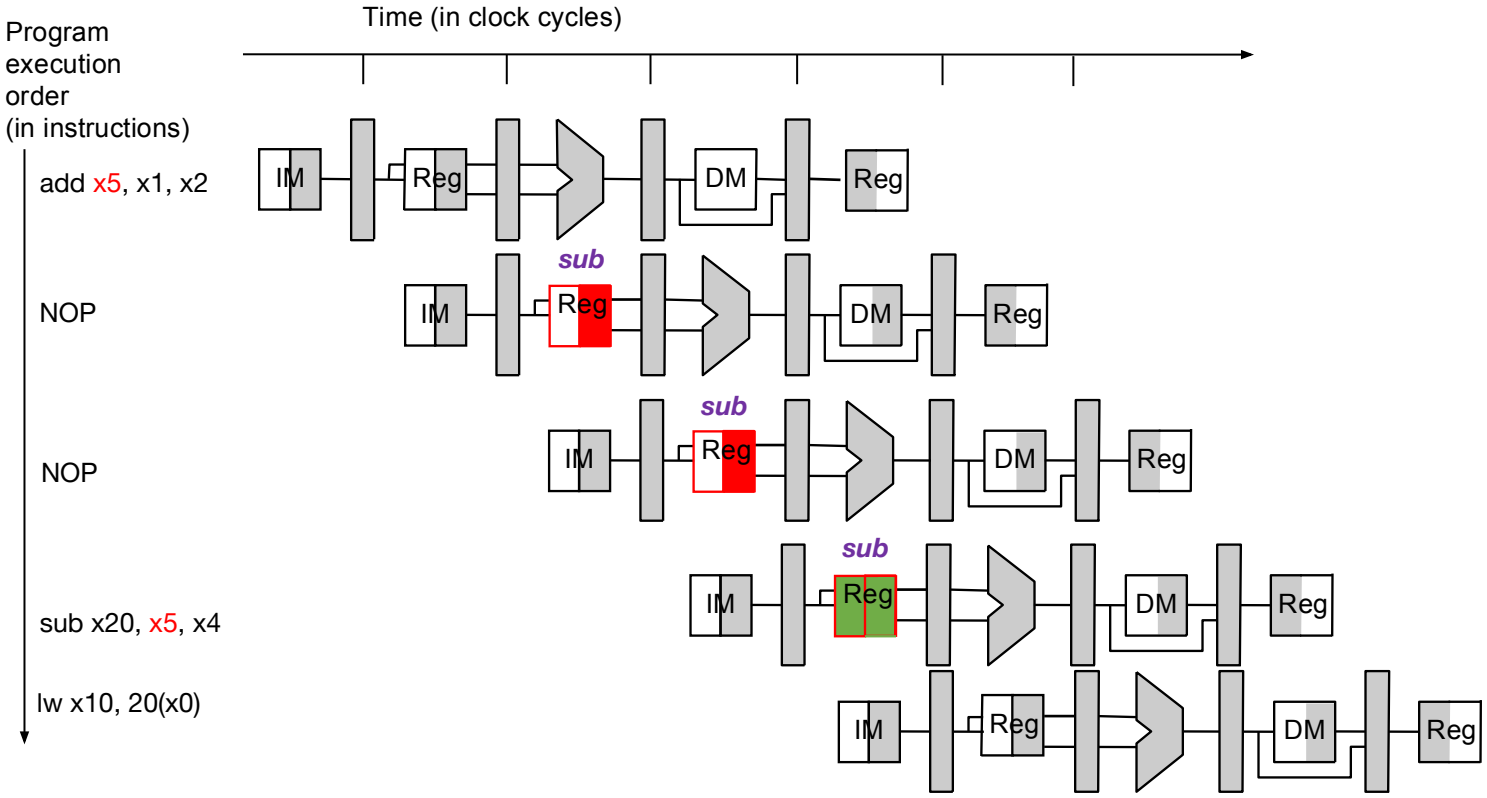
Data Hazard

"bubble" instruction

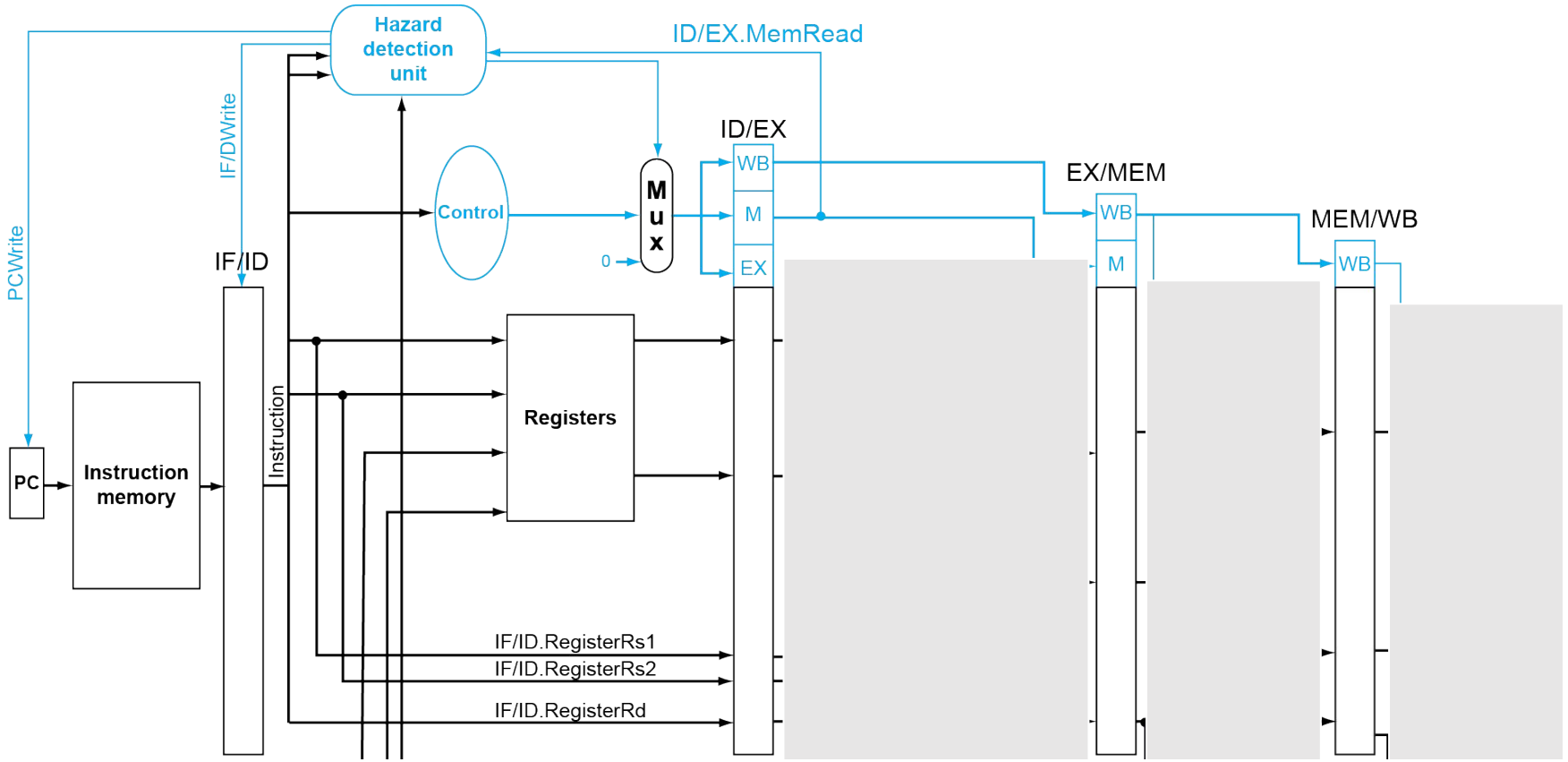


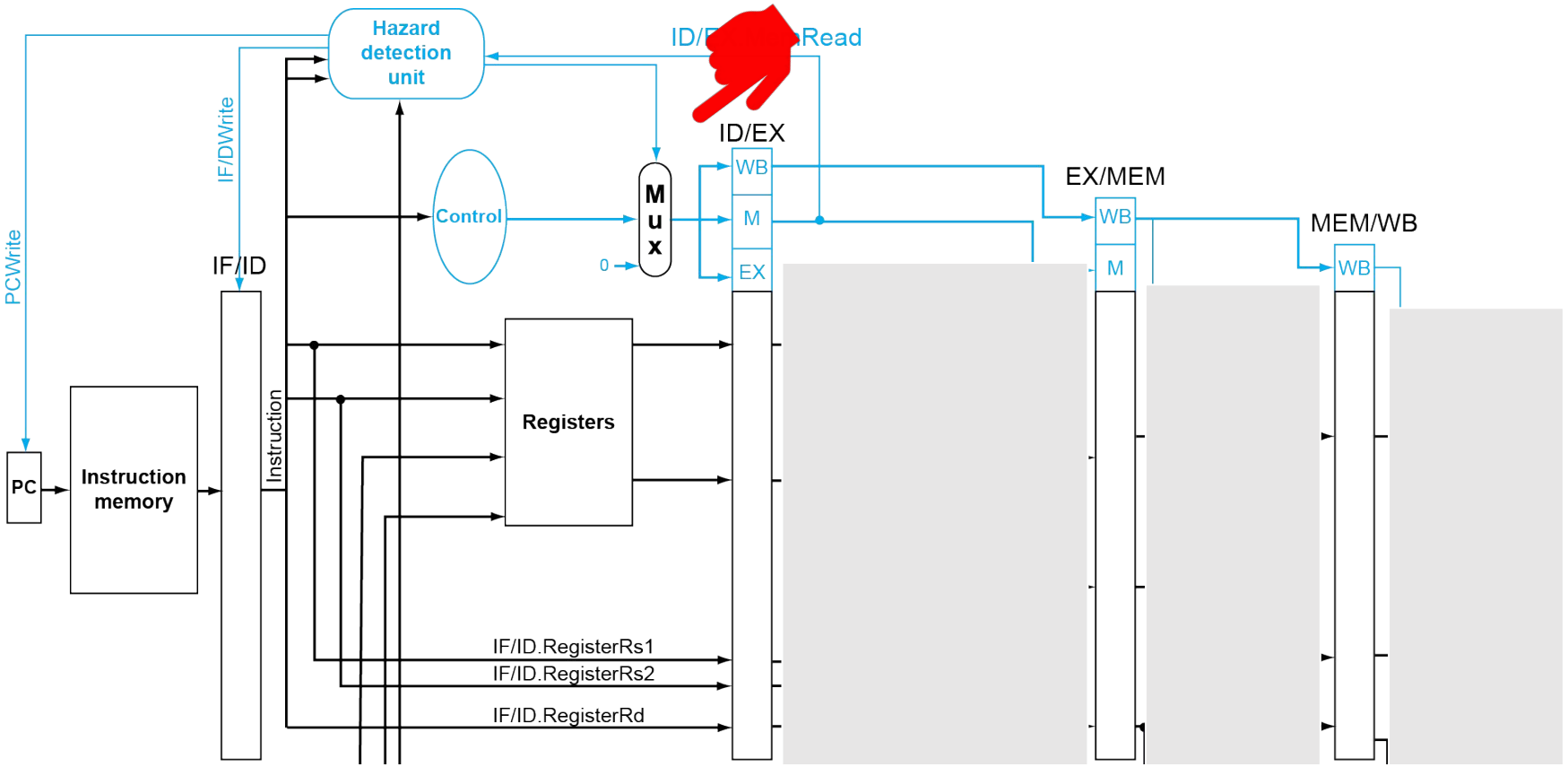
Data Hazard

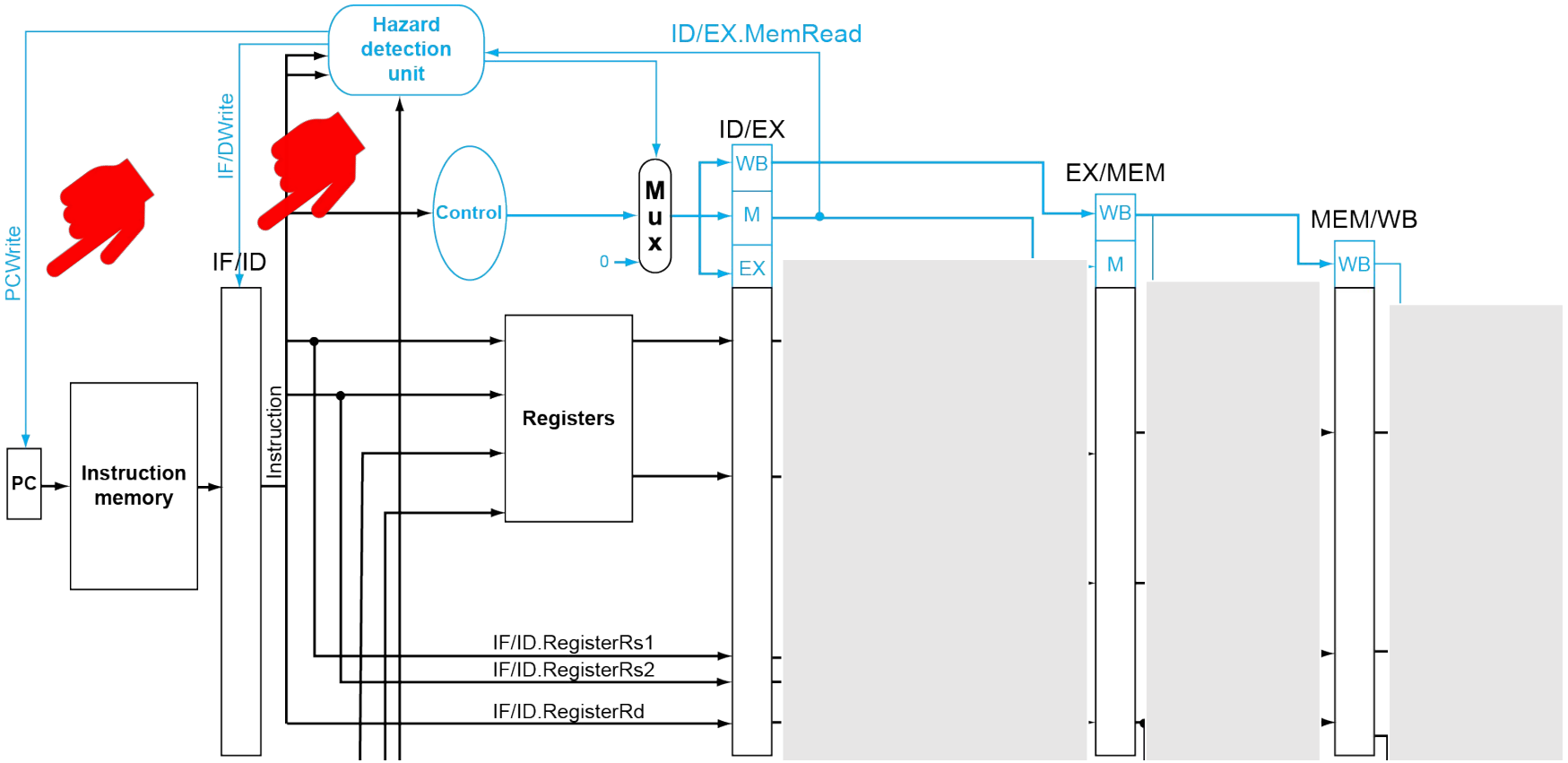
"bubble" instruction



How to implement this?


Hazard detection unit
ID/EX.MemRead
IF/DWrite
PCWrite
Control
M u x
0
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
IF/ID
Instruction
PC
Instruction memory
Registers
IF/ID.RegisterRs1
IF/ID.RegisterRs2
IF/ID.RegisterRd

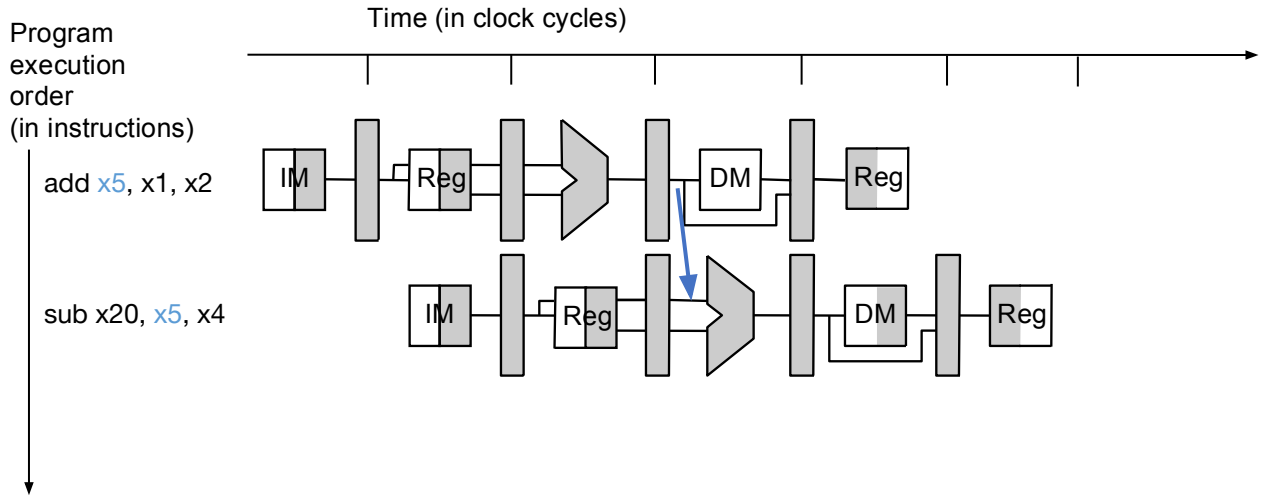

Hazard detection unit
ID/EX
Read
IF/DWrite
PCWrite
Control
M u x
0
WB
M
EX
EX/MEM
MEM/WB
IF/ID
Instruction
PC
Instruction memory
Registers
IF/ID.RegisterRs1
IF/ID.RegisterRs2
IF/ID.RegisterRd


Hazard detection unit
ID/EX.MemRead
IF/DWrite
PCWrite
Control
M u x
0
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
IF/ID
Instruction
PC
Instruction memory
Registers
IF/ID.RegisterRs1
IF/ID.RegisterRs2
IF/ID.RegisterRd

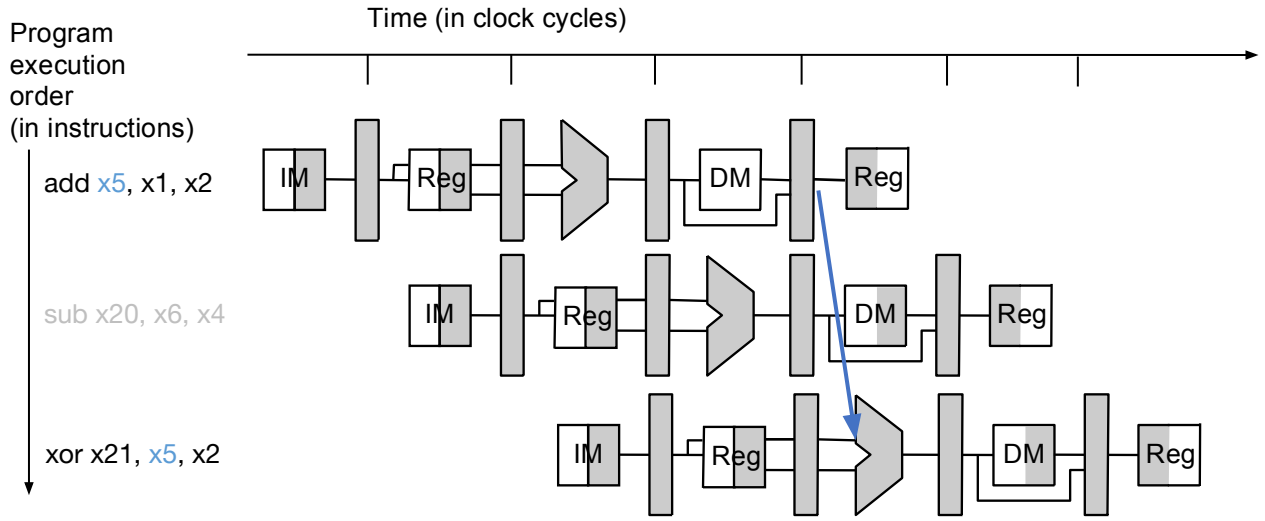
How do we fix RAW hazards?

- Two options:
  - Stalling
  - Forwarding

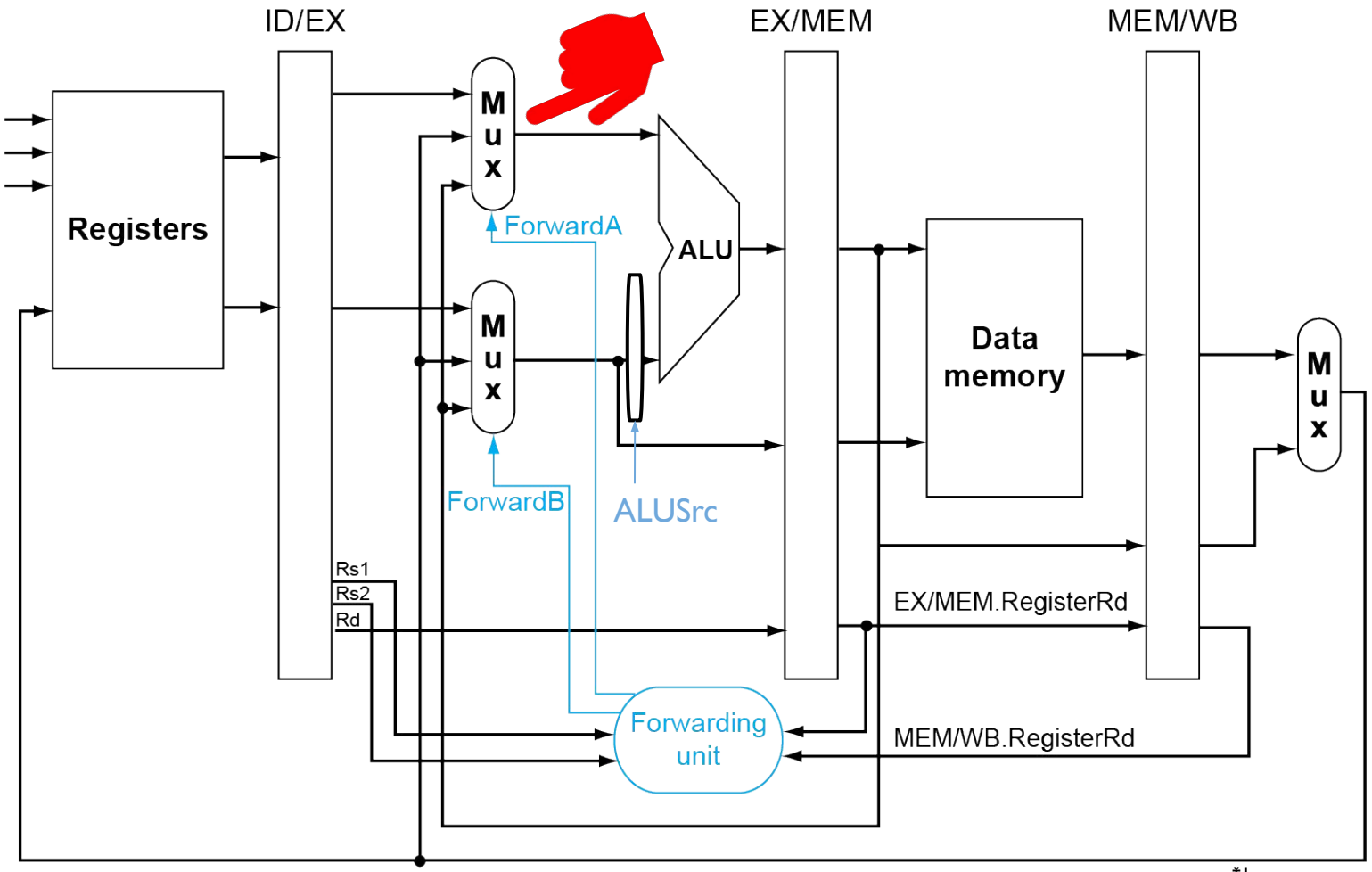
Forwarding (Bypassing)



Forwarding (Bypassing)

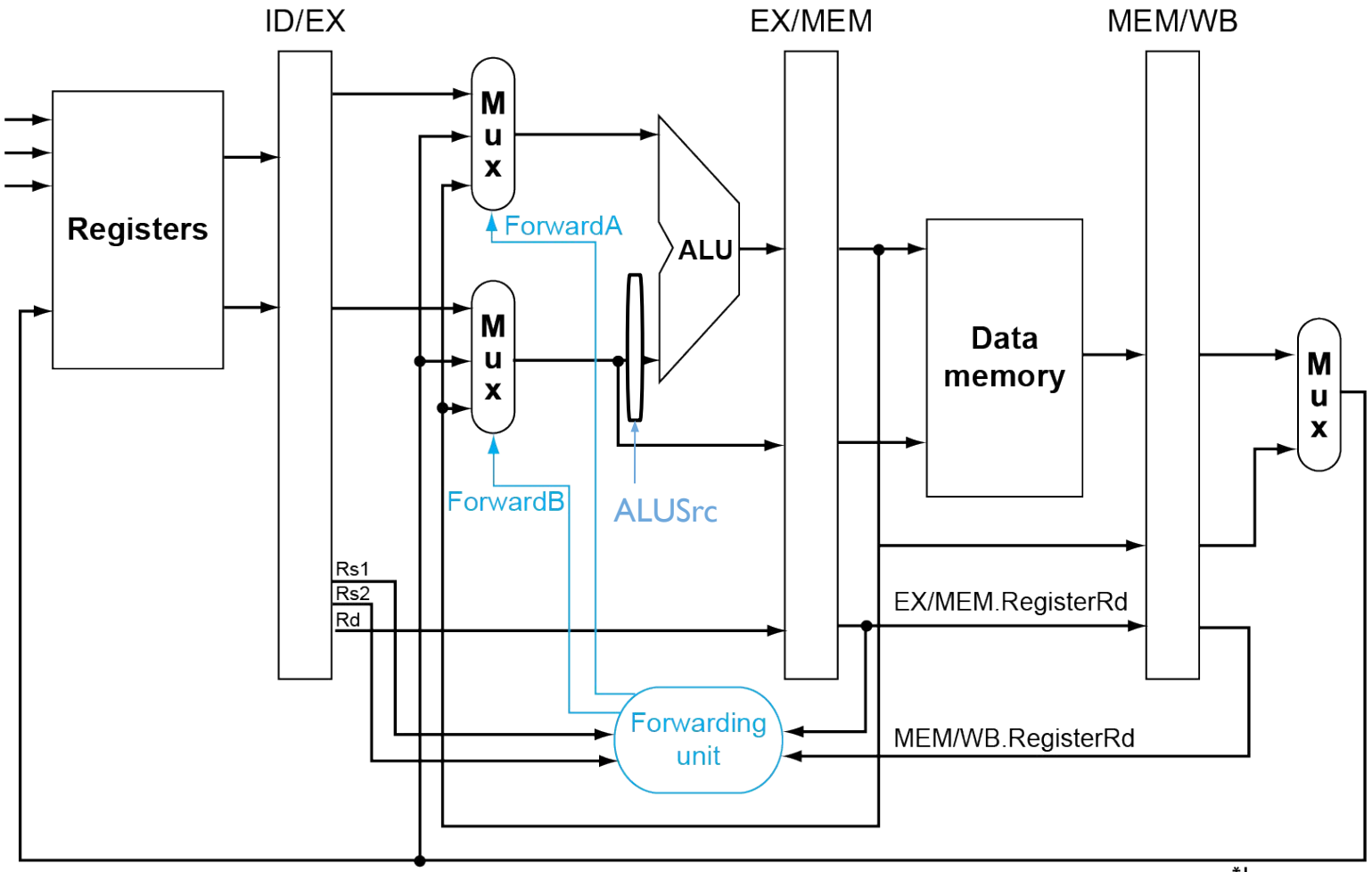


Datapath with *RAW Forwarding*



*Images were taken from Hennessy Patterson Book [1].

Datapath with *RAW Forwarding*



- Forward either from MEM or WB
- Forward to RS1 or RS2 or both!

*Images were taken from Hennessy Patterson Book [1].

How to detect when to forward?

RAW

- Data hazards when:

1a. EX/MEM. Rd = ID/EX. Rs1
1b. EX/MEM. Rd = ID/EX. Rs2
AND EX/MEM.RegWrite = 1

→ *Forward from EX/MEM pipeline reg*

2a. MEM/WB. Rd = ID/EX. Rs1
2b. MEM/WB. Rd = ID/EX. Rs2
AND MEM/WB.RegWrite = 1

→ *Forward from MEM/WB pipeline reg*

- Watch out for x0!

How do we fix RAW hazards?

- Two options:
  - Stalling
  - Forwarding

➔ Forwarding helps to prevent stalls!

How do we fix RAW hazards?

- Two options:
  - Stalling
  - Forwarding

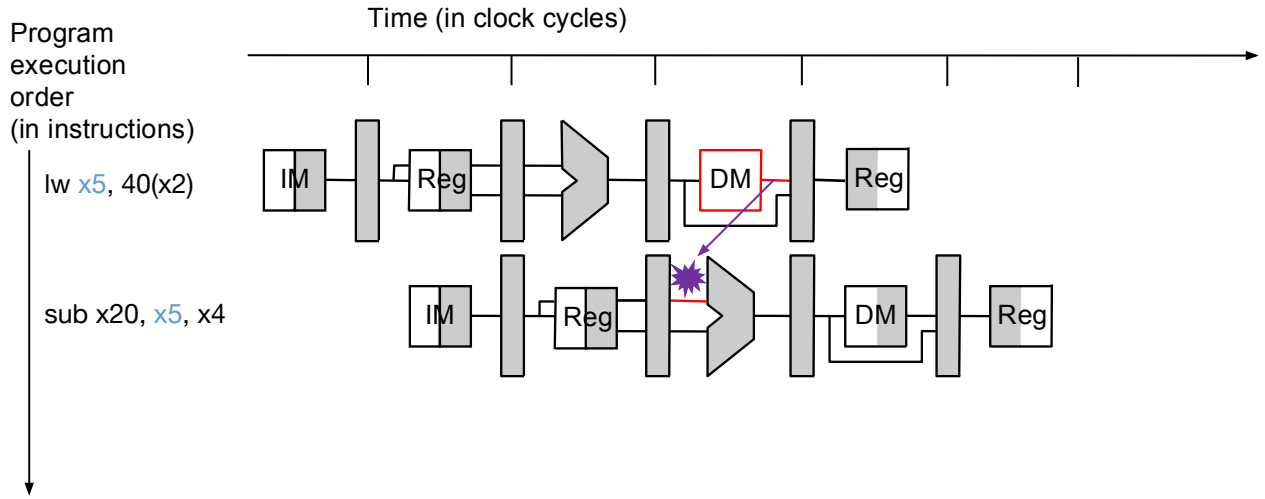
➔ Forwarding helps to prevent stalls!

but is this always true?

Forwarding for LW instructions?

- Forwarding alone won't help for LW!

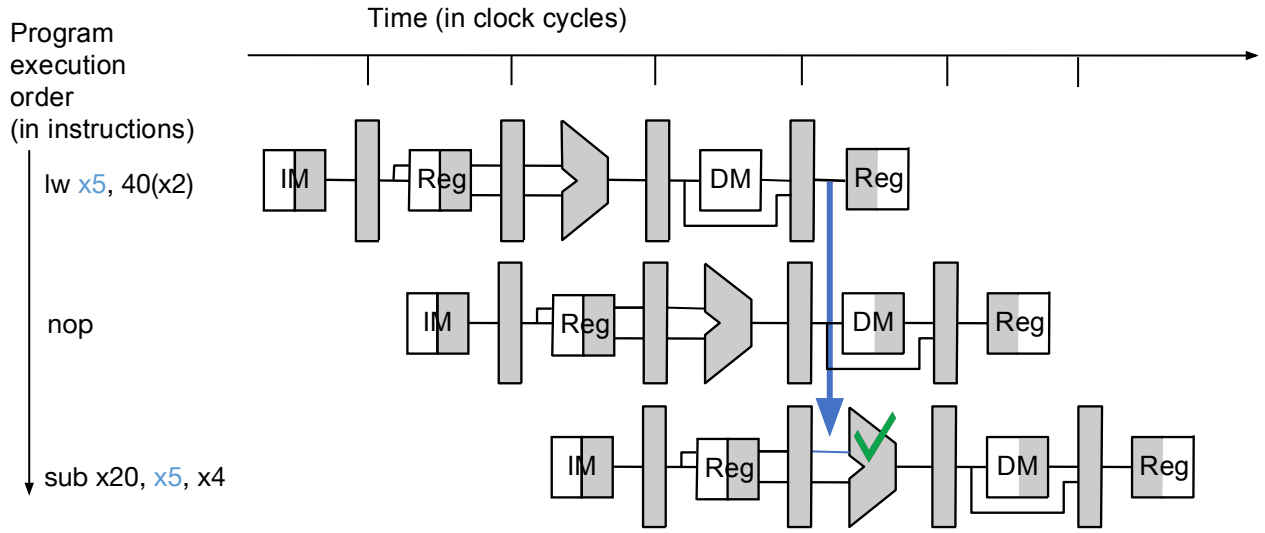
Load-Use Data Hazard



- Unlike in *reg-to-reg* forwarding, in this case data is ***not yet ready*** in *Mem* stage to be forwarded to *EX* stage.
- Half cycle vs. full cycle!

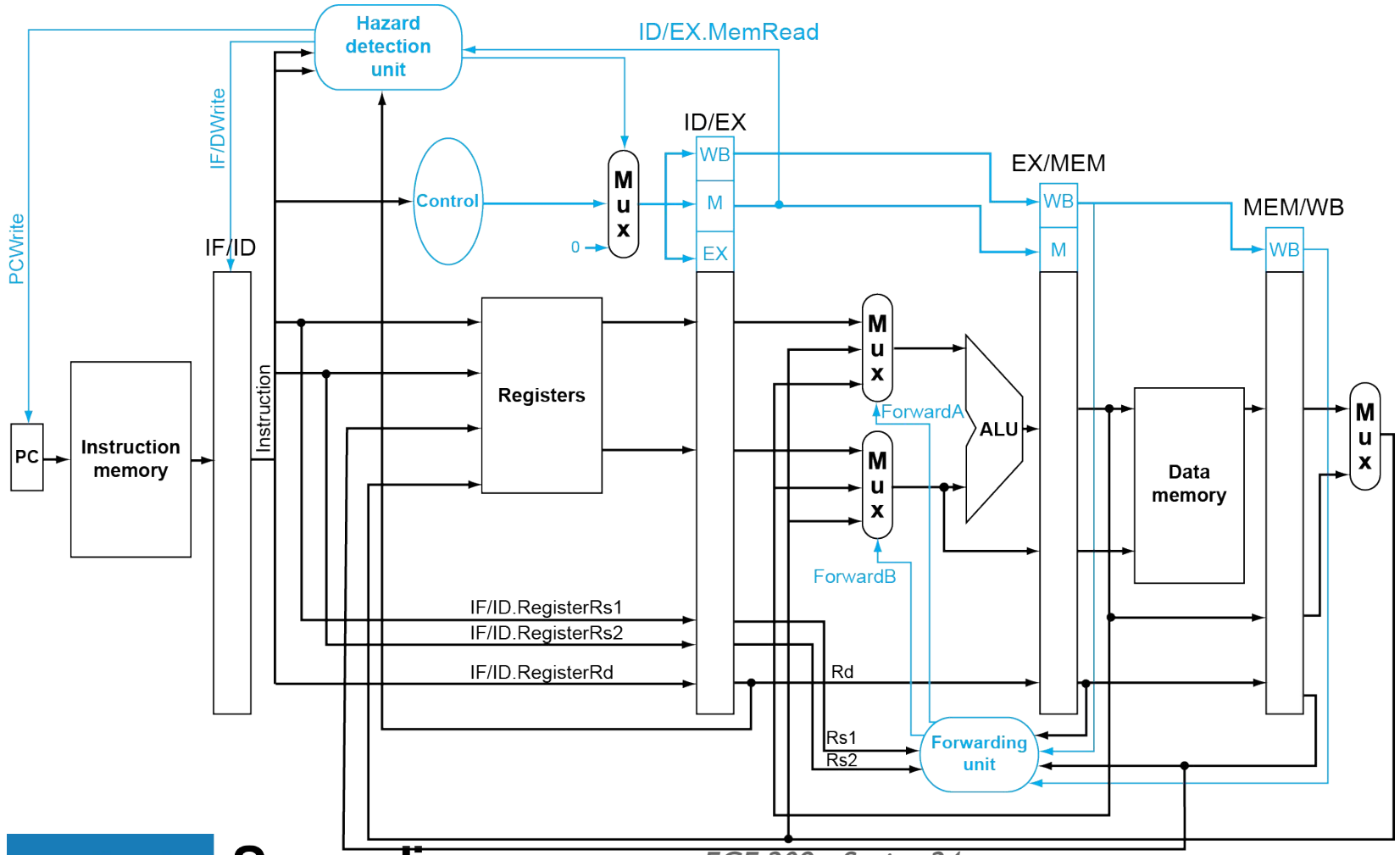
Load-Use Data Hazard

Forwarding AND Stalling

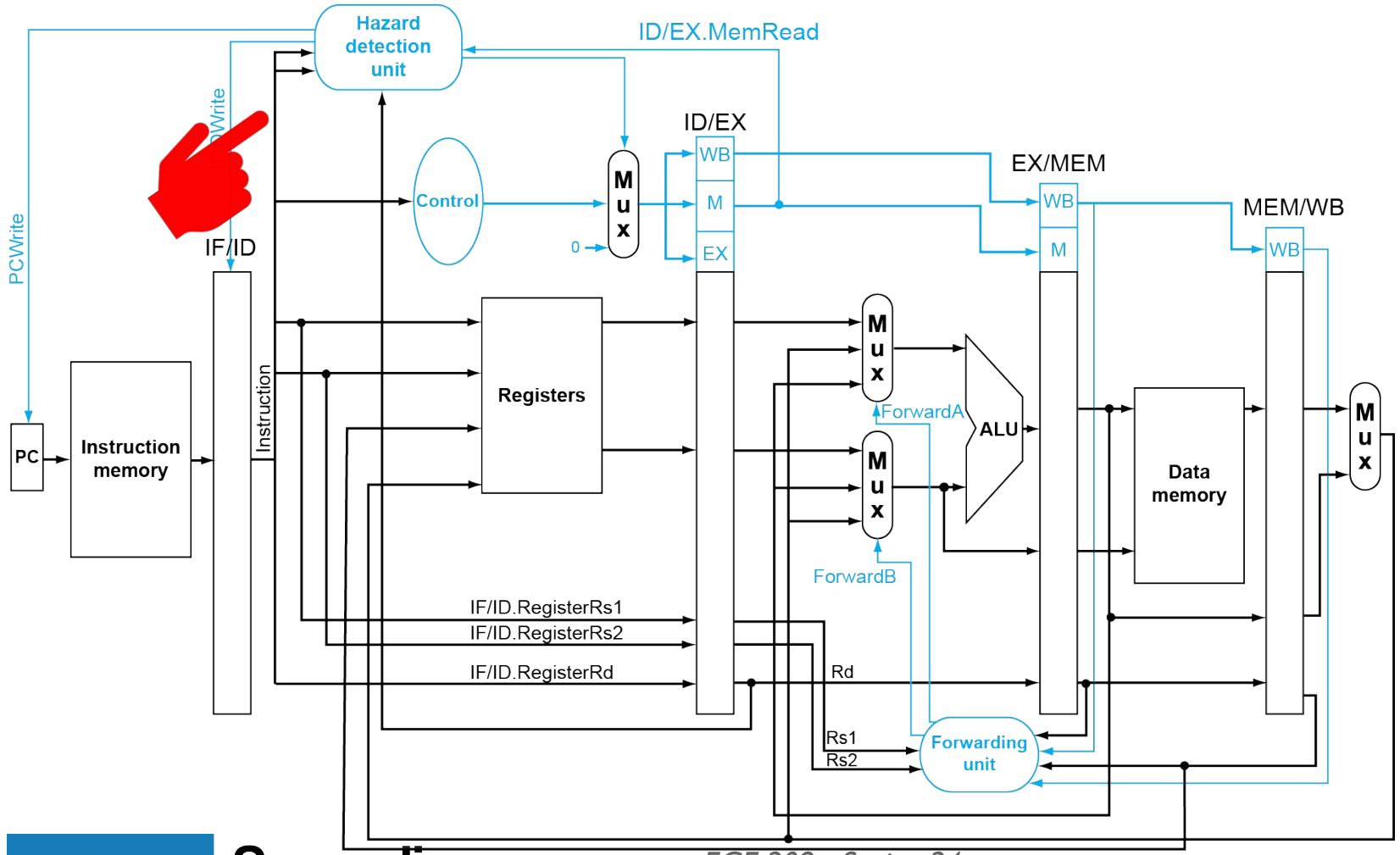


Note that, we still do the forwarding, but we have to stall for one cycle.

Forwarding AND Stalling

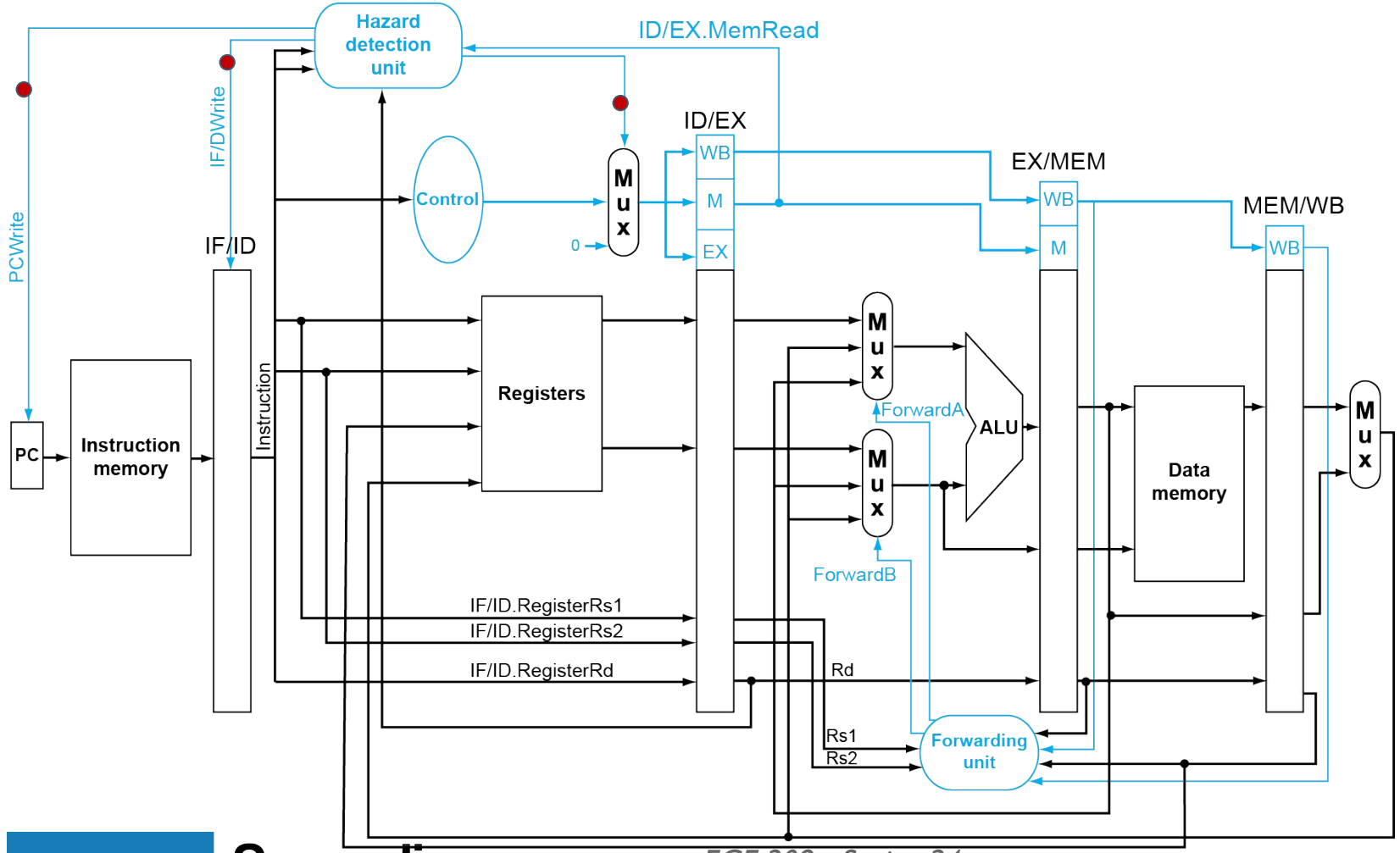


Forwarding AND Stalling



1a. ID/EX. Rd = IF/ID. Rs1
2b. ID/EX. Rd = IF/ID.Rs2
AND ID/EX.MemRead = 1

Forwarding AND Stalling



1a. ID/EX. Rd = IF/ID. Rs1

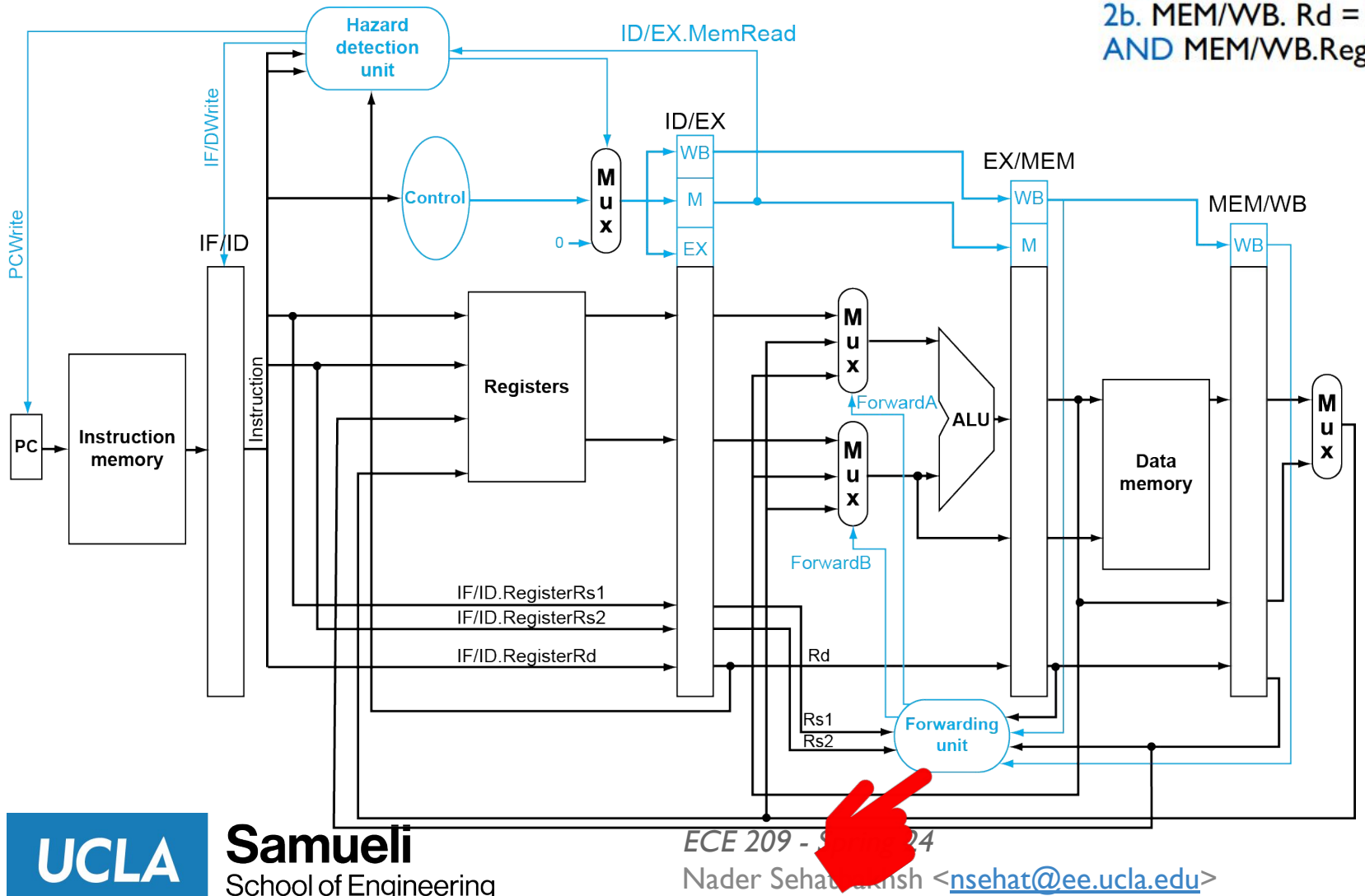
2b. ID/EX. Rd = IF/ID.Rs2

AND ID/EX.MemRead = 1

Forwarding AND Stalling

1a. EX/MEM. Rd = ID/EX. Rs1
1b. EX/MEM. Rd = ID/EX. Rs2
AND EX/MEM.RegWrite = 1
→ *Forward from EX/MEM pipeline reg*

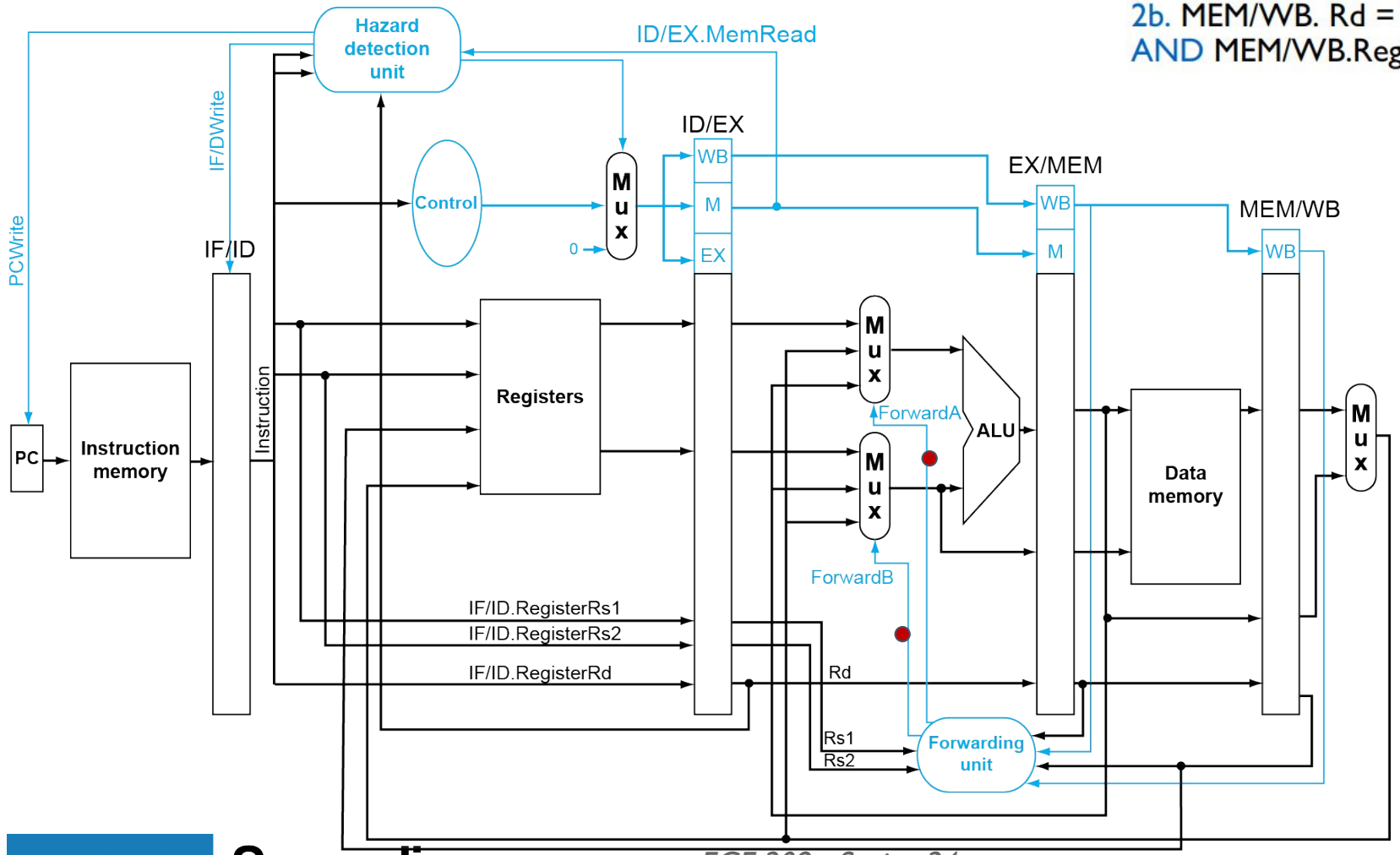
2a. MEM/WB. Rd = ID/EX. Rs1
2b. MEM/WB. Rd = ID/EX. Rs2
AND MEM/WB.RegWrite = 1
→ *Forward from MEM/WB pipeline reg*



Forwarding AND Stalling

1a. EX/MEM. Rd = ID/EX. Rs1
1b. EX/MEM. Rd = ID/EX. Rs2
AND EX/MEM.RegWrite = 1
→ *Forward from EX/MEM pipeline reg*

2a. MEM/WB. Rd = ID/EX. Rs1
2b. MEM/WB. Rd = ID/EX. Rs2
AND MEM/WB.RegWrite = 1
→ *Forward from MEM/WB pipeline reg*



What else?

UCLA Samueli School of Engineering

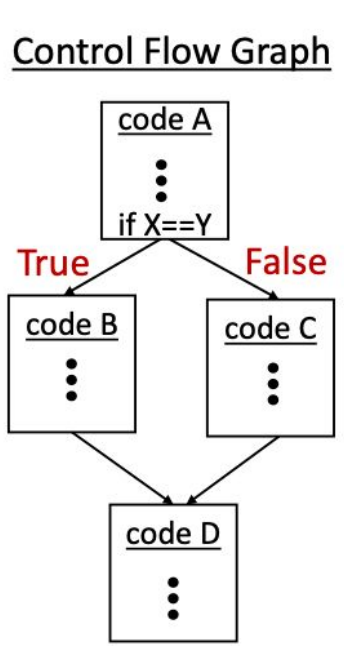
Control Hazard

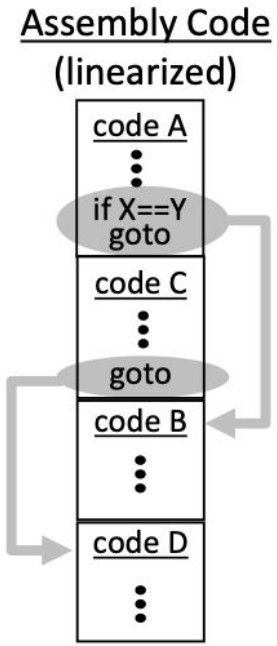
- Do we always know what is the next instruction?
  - Yes for straight line code
  - No for jumps/branches! Correctness issue!

What about branches?

- *What instruction should be fetched after a branch?*

```
// Code A
...
if(x==y)
{
  // B
}
else {
 // C
}
// Code D
...
```





- We need to "skip" in some cases!
- (but we don't know when!)

How to fix this?

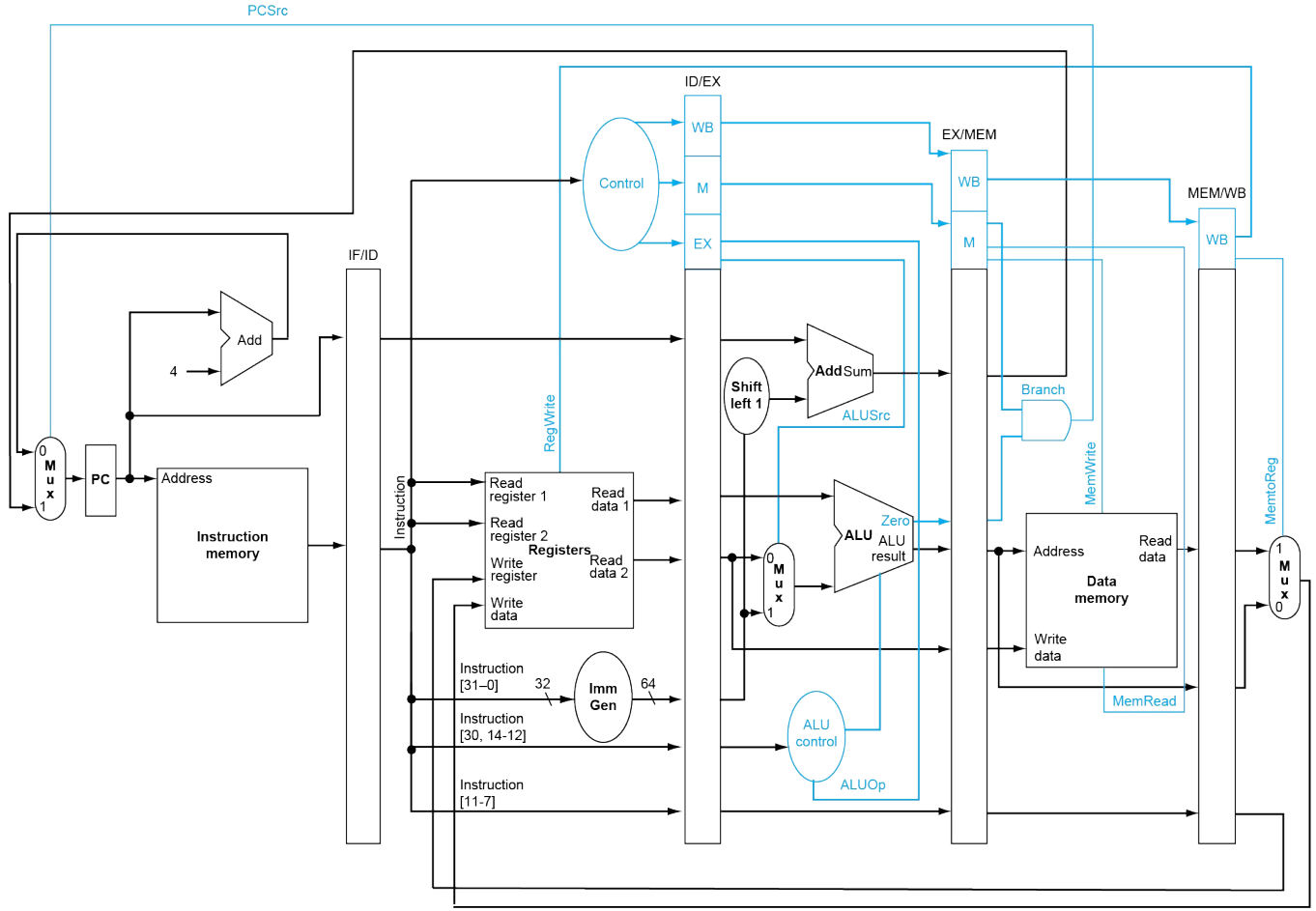
- Again two options:
  - Stalling
  - Speculation (not forwarding!)

Option 1: Stalling

- How long should we stall?

Stall: 3 cycles!

- ***When:***
  - End of *ID* stage
- ***Where:***
  - End of *EX* or Beg. *Mem* stage
- ***Whether:***
  - End of *Mem* stage



Option 2: Speculation/Prediction

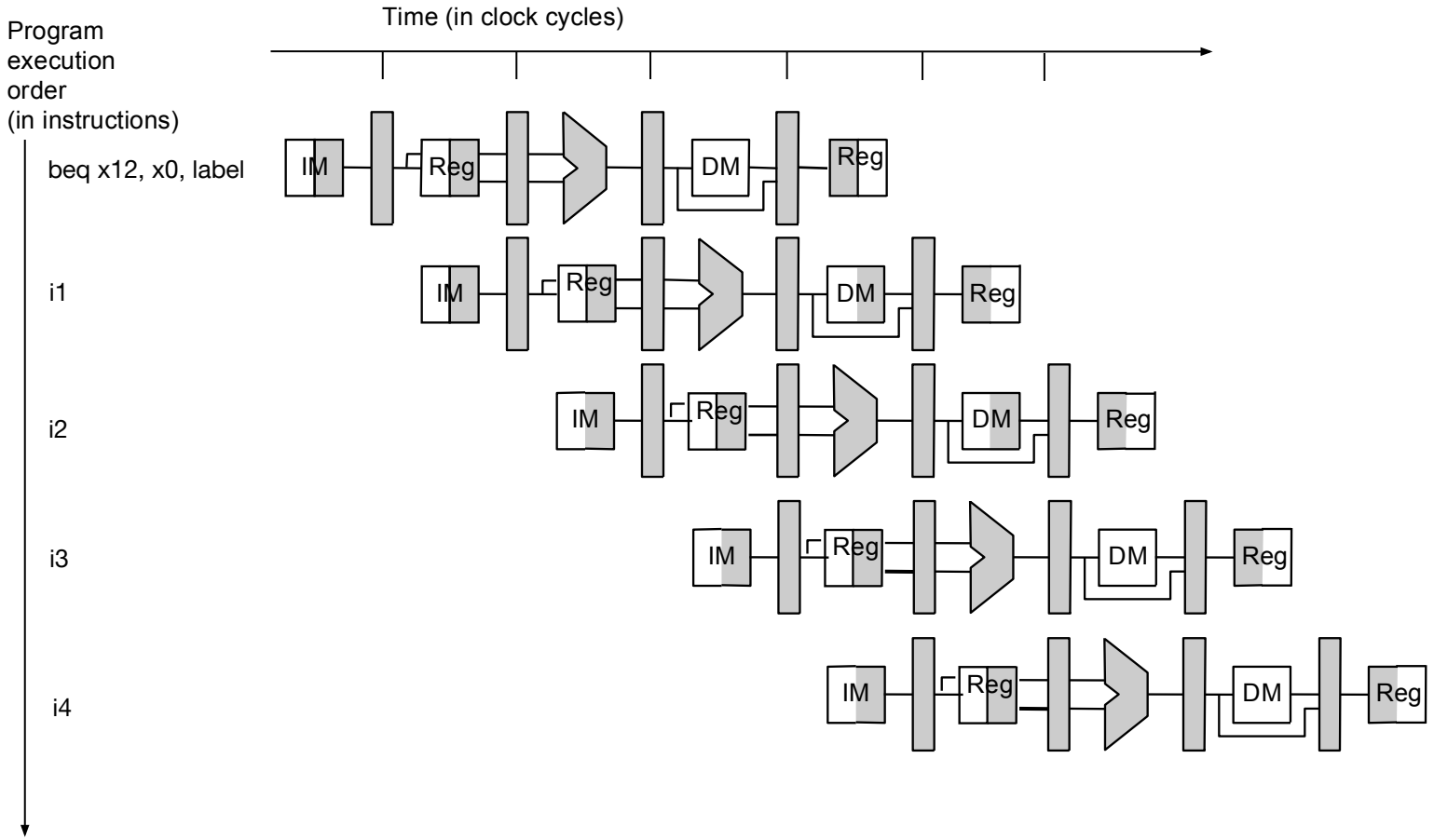
- What to predict?
  - Two options: we either predict always taken or always not taken!
  - Which one is easier?
    - *Not taken!*
    - Why?
      - Because:
        a. We don't need branch addresses, not taken means that next address is PC+4
        b. Most instructions are not branch so they are not taken too!

Option 2: Speculation/Prediction

- What to predict?
  - Two options: we either predict always taken or always not taken!
  - Which one is easier?
    - *Not taken!*
    - Why?
      - Because:
        a. We don't need branch addresses, not taken means that next address is PC+4
        b. Most instructions are not branch so they are not taken too!
- One minor issue: what to do if we mispredict?

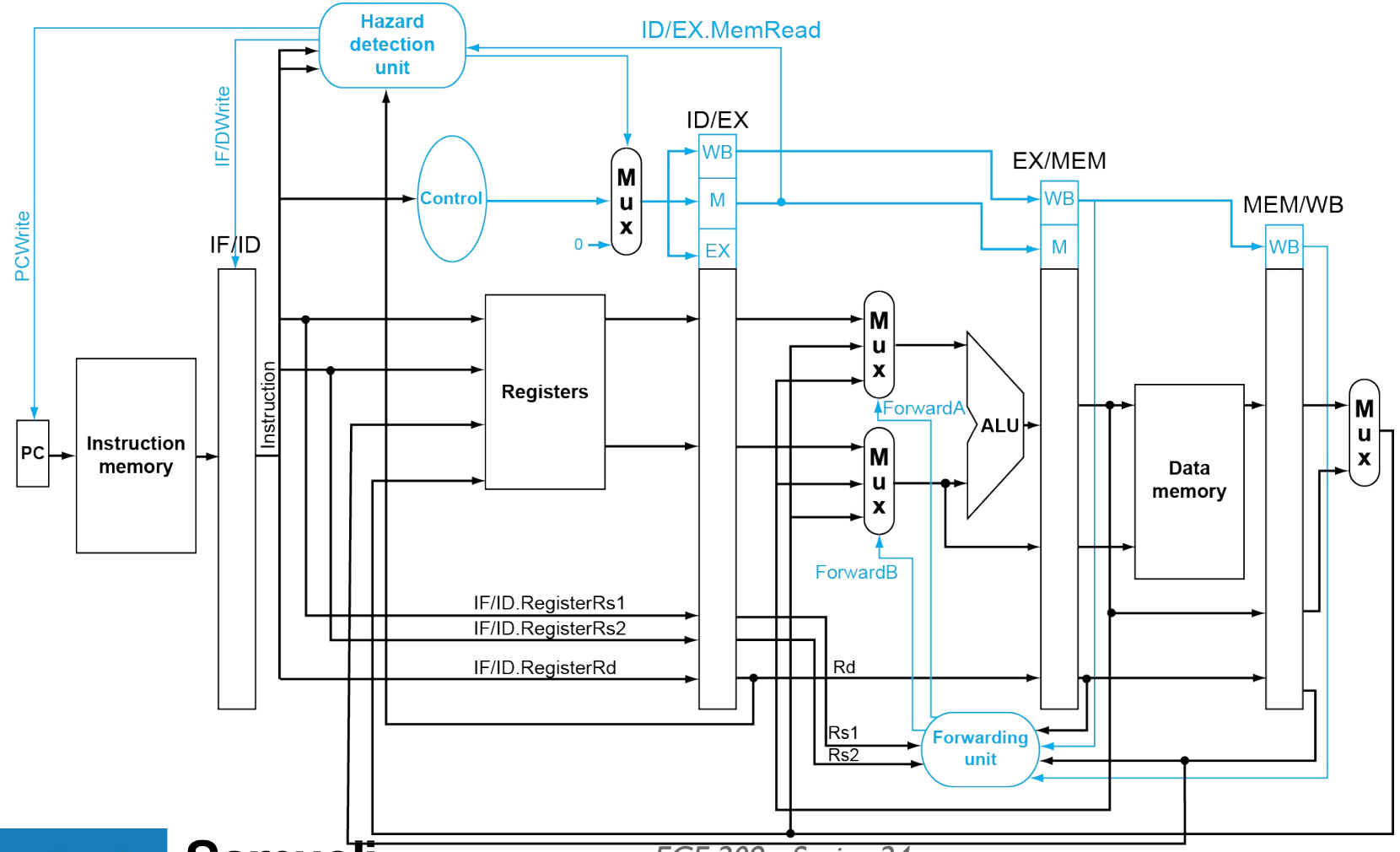
➔ We have to flush!

Flushing Instructions

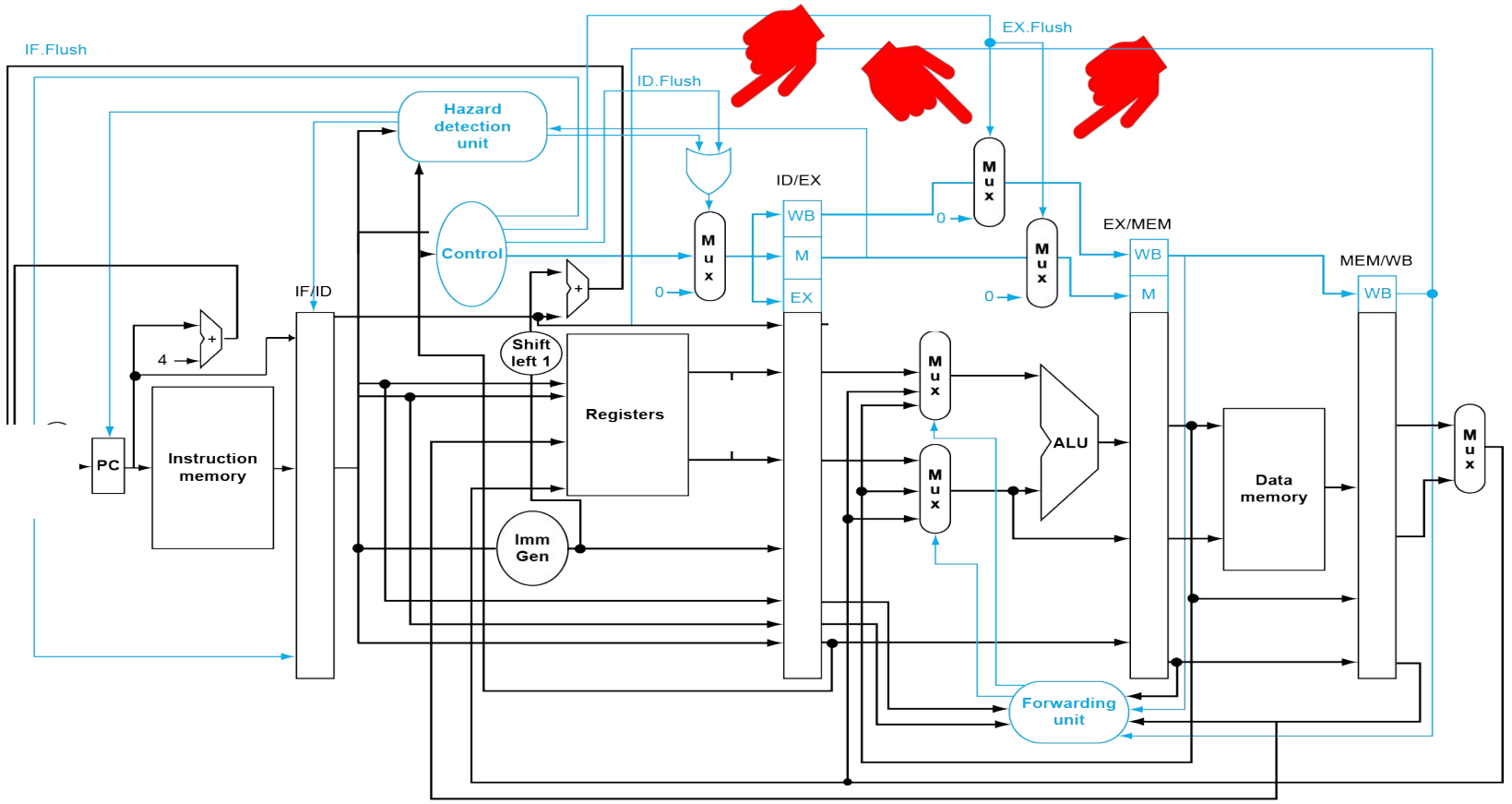


- We need to flush i1-i3 in this example.

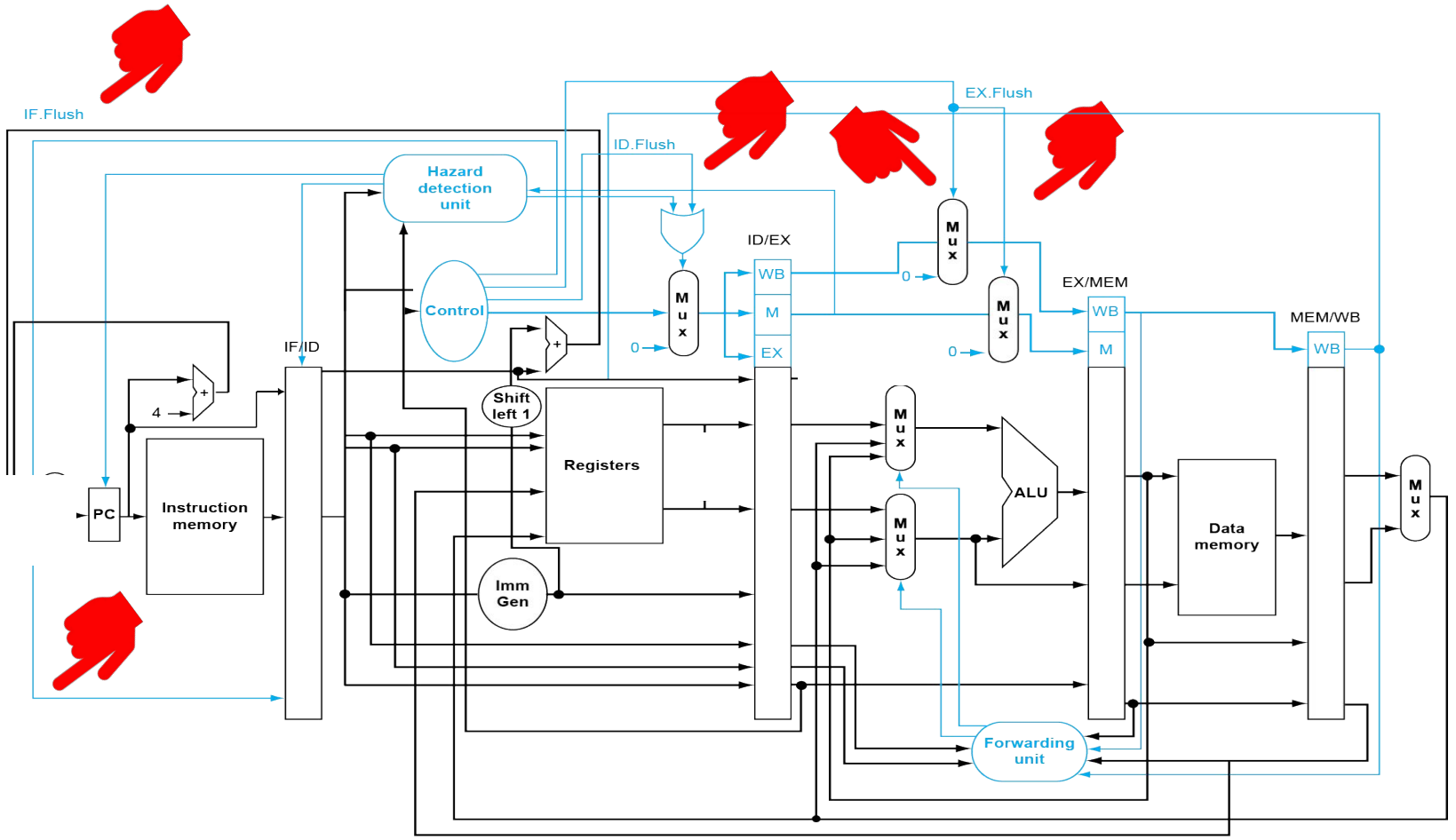
How to flush?


Hazard detection unit
ID/EX.MemRead
IF/DWrite
PCWrite
Control
0
M u x
ID/EX
WB
M
EX
EX/MEM
MEM/WB
IF/ID
Instruction
PC
Instruction memory
Registers
ForwardA
ALU
Data memory
ForwardB
IF/ID.RegisterRs1
IF/ID.RegisterRs2
IF/ID.RegisterRd
Rd
Rs1
Rs2
Forwarding unit

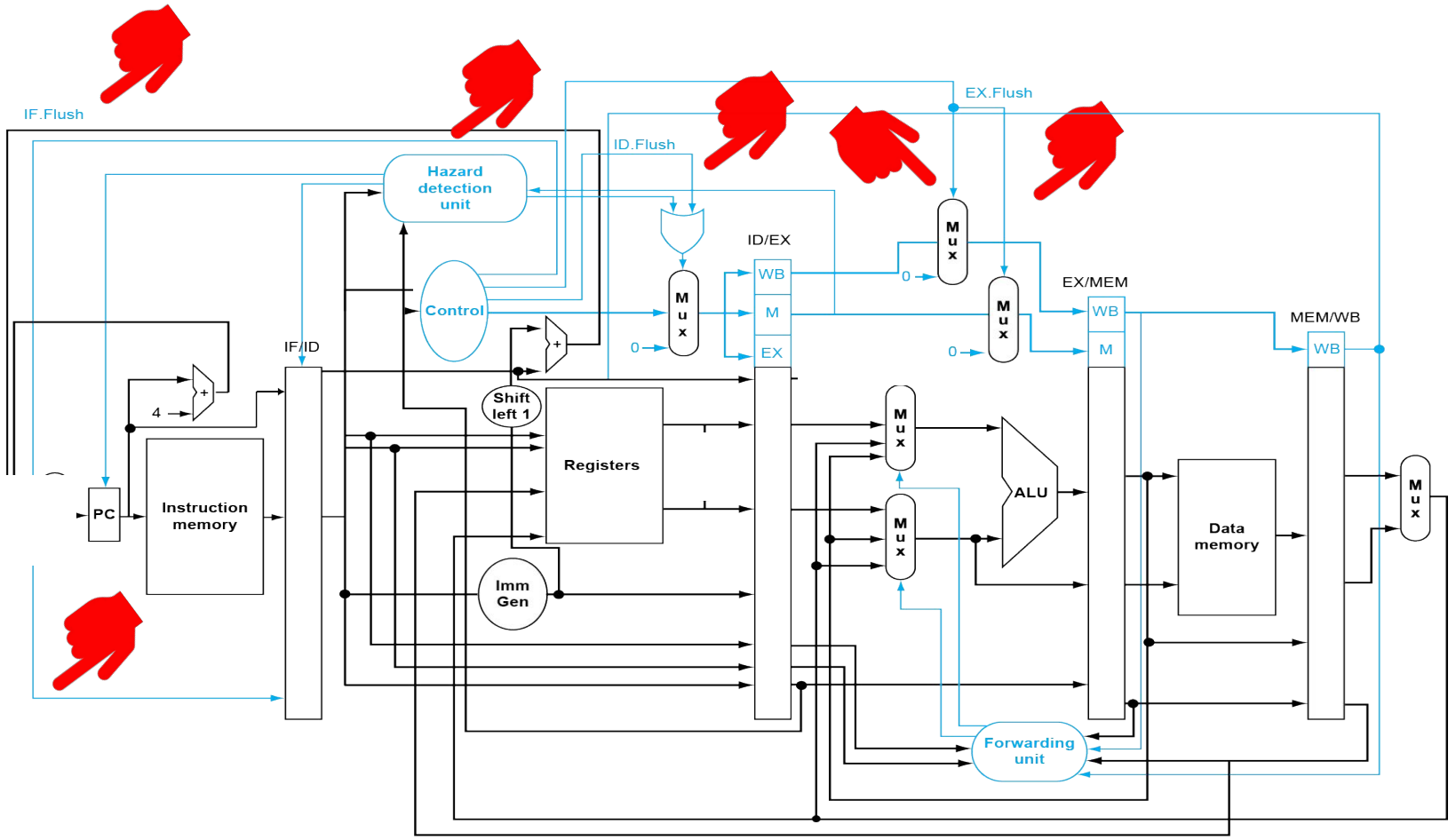
How to flush?


IF.Flush
ID.Flush
EX.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Mux
0
IF/ID
4
Shift left 1
Registers
PC
Instruction memory
Imm Gen
ALU
Data memory
Forwarding unit

How to flush?


IF.Flush
ID.Flush
EX.Flush
Hazard detection unit
Control
IF/ID
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Mux
0
4
Shift left 1
Registers
PC
Instruction memory
Imm Gen
ALU
Data memory
Forwarding unit

How to flush?


IF.Flush
ID.Flush
EX.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Mux
0
IF/ID
4
Shift left 1
Registers
PC
Instruction memory
Imm Gen
ALU
Data memory
Forwarding unit

Option 2: Speculation/Prediction

- What to predict?
  - Two options: we either predict always taken or always not taken!
  - Which one is easier?
    - *Not taken!*
    - Why?
      - Because:
        a. We don't need branch addresses, not taken means that next address is PC+4
        b. Most instructions are not branch so they are not taken too!
- One minor issue: what to do if we mispredict?

➔ We have to flush! → *three cycles penalty*

Option 2: Speculation/Prediction

- What to predict?
  - Two options: we either predict always taken or always not taken!
  - Which one is easier?
    - *Not taken!*
    - Why?
      - Because:
        a. We don't need branch addresses, not taken means that next address is PC+4
        b. Most instructions are not branch so they are not taken too!
- One minor issue: what to do if we mispredict?

➔ We have to flush! → *three cycles penalty*

★ *Can we predict sooner? → DE*

Resolving the Branch Sooner

- *How soon?*
  - *Decode* stage!

- What needs to be changed?
  - When ✓
  - Where ✗
  - Whether ✗

Resolving the Branch Sooner
"Where"

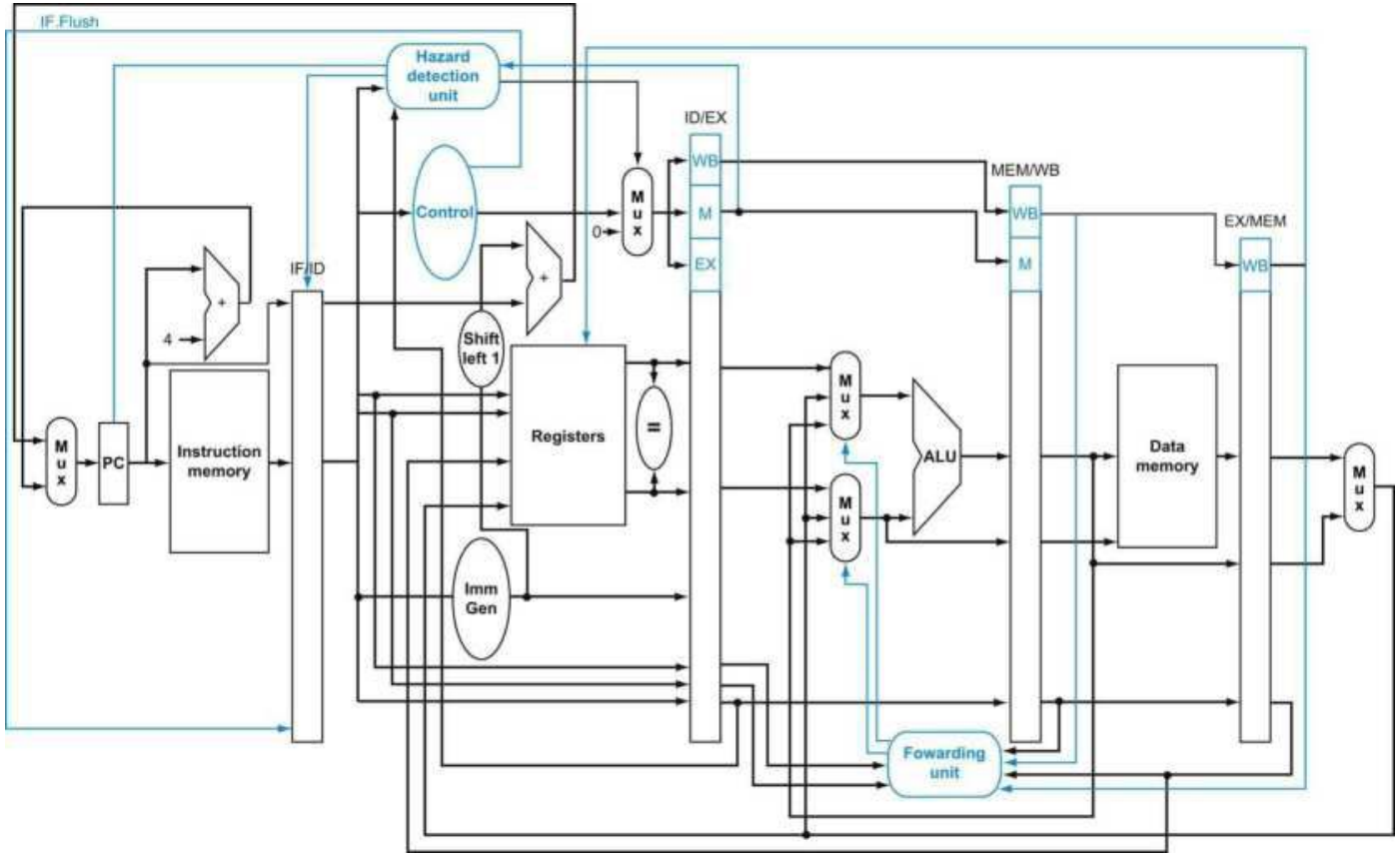
- Offset is available in DE stage.
  - Move nextPC calculation to DE
  - Use ALU for PC+offset in DE stage

Resolving the Branch Sooner
"Whether"

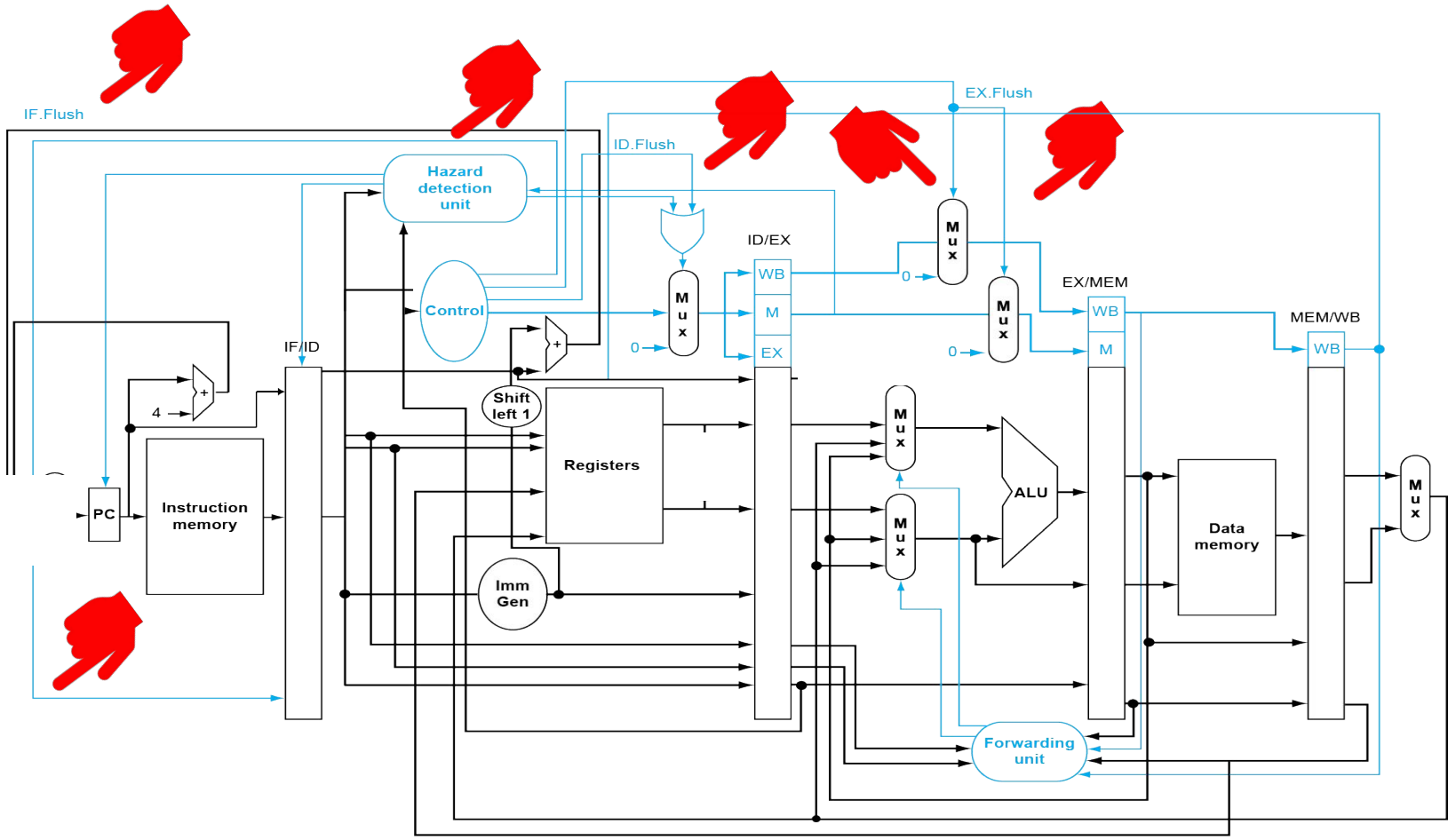
- Offset is available in DE stage.
  - Move nextPC calculation to DE
  - Use ALU for PC+offset in DE stage

- rs1 and rs2 are also available in DE stage (for *Zero*).
  - Move the comparison between rs1 and rs2 after reading from the register file.

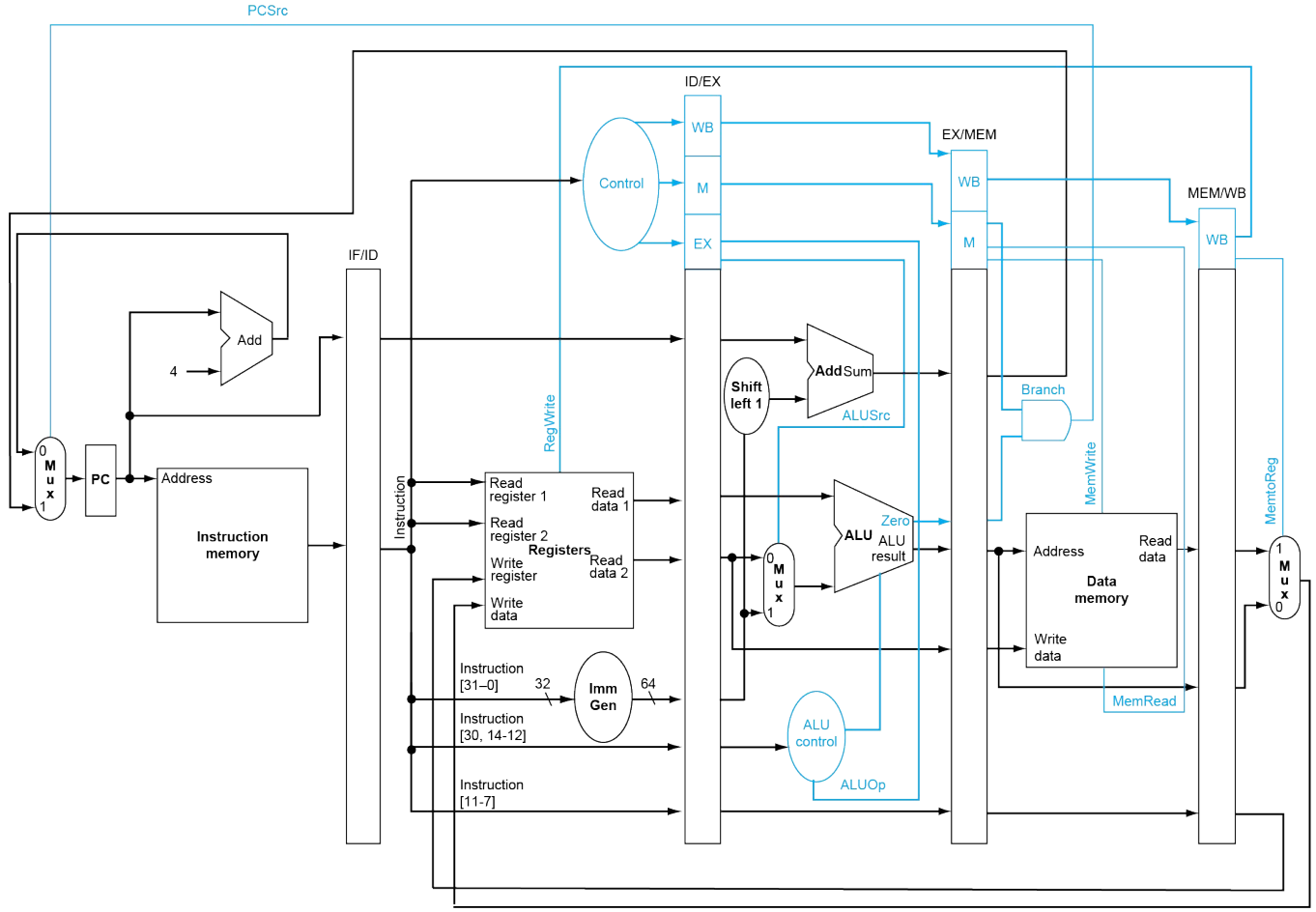
Forwarding, Stalling, and Branch Prediction (Not Taken at DE)



Resolve at MEM vs. Resolve at DE



Baseline (no forwarding or branching)



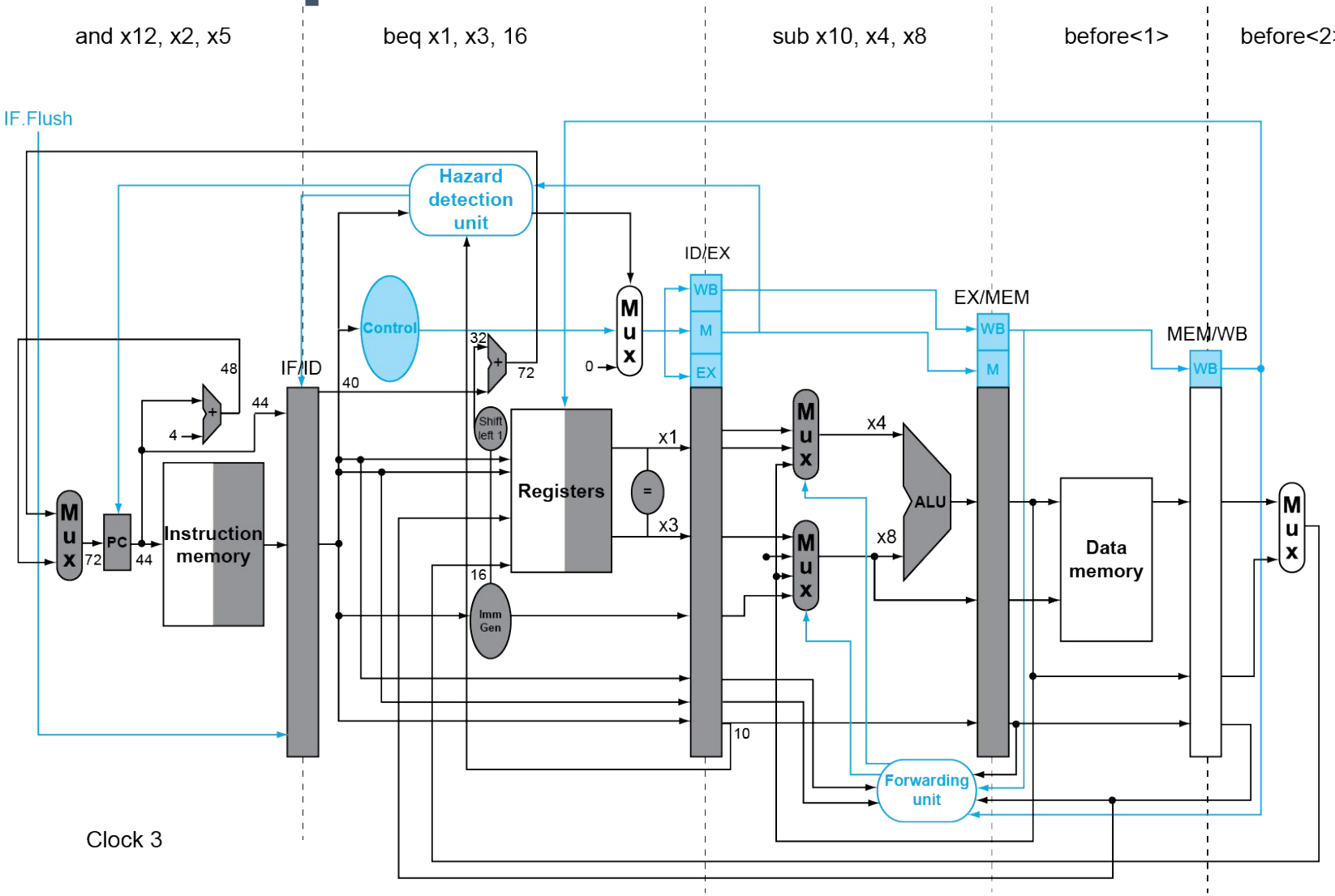
How to improve the prediction?

- We use branch predictors?
- How? → we will talk about it next week

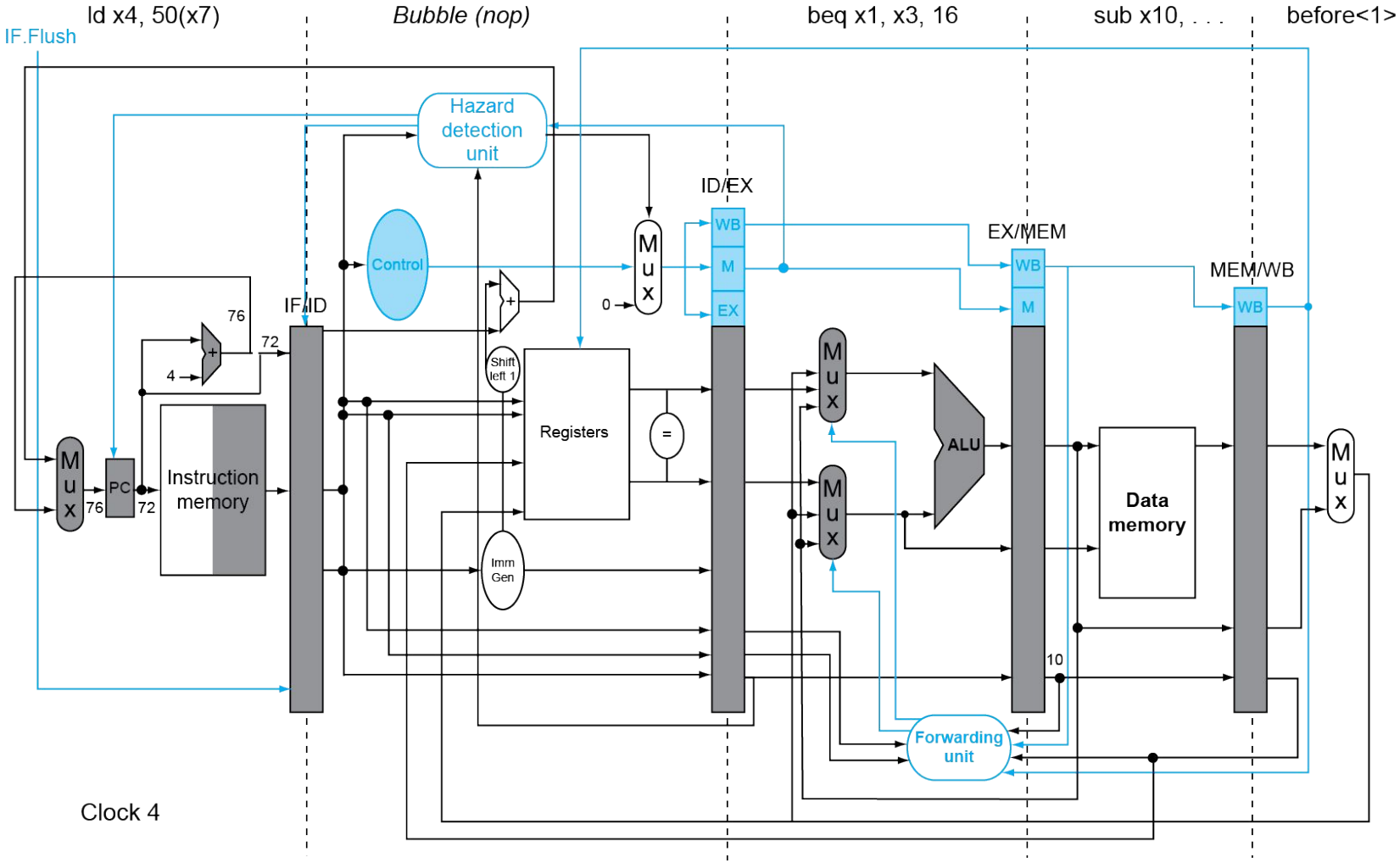
An Example

```
36:  sub  x10, x4, x8
40:  beq  x1,  x3, 16  // PC-relative branch
                       // to 40+16*2=72
44:  and  x12, x2, x5
48:  orr  x13, x2, x6
52:  add  x14, x4, x2
56:  sub  x15, x6, x7
      ...
72:  ld   x4, 50(x7)
```

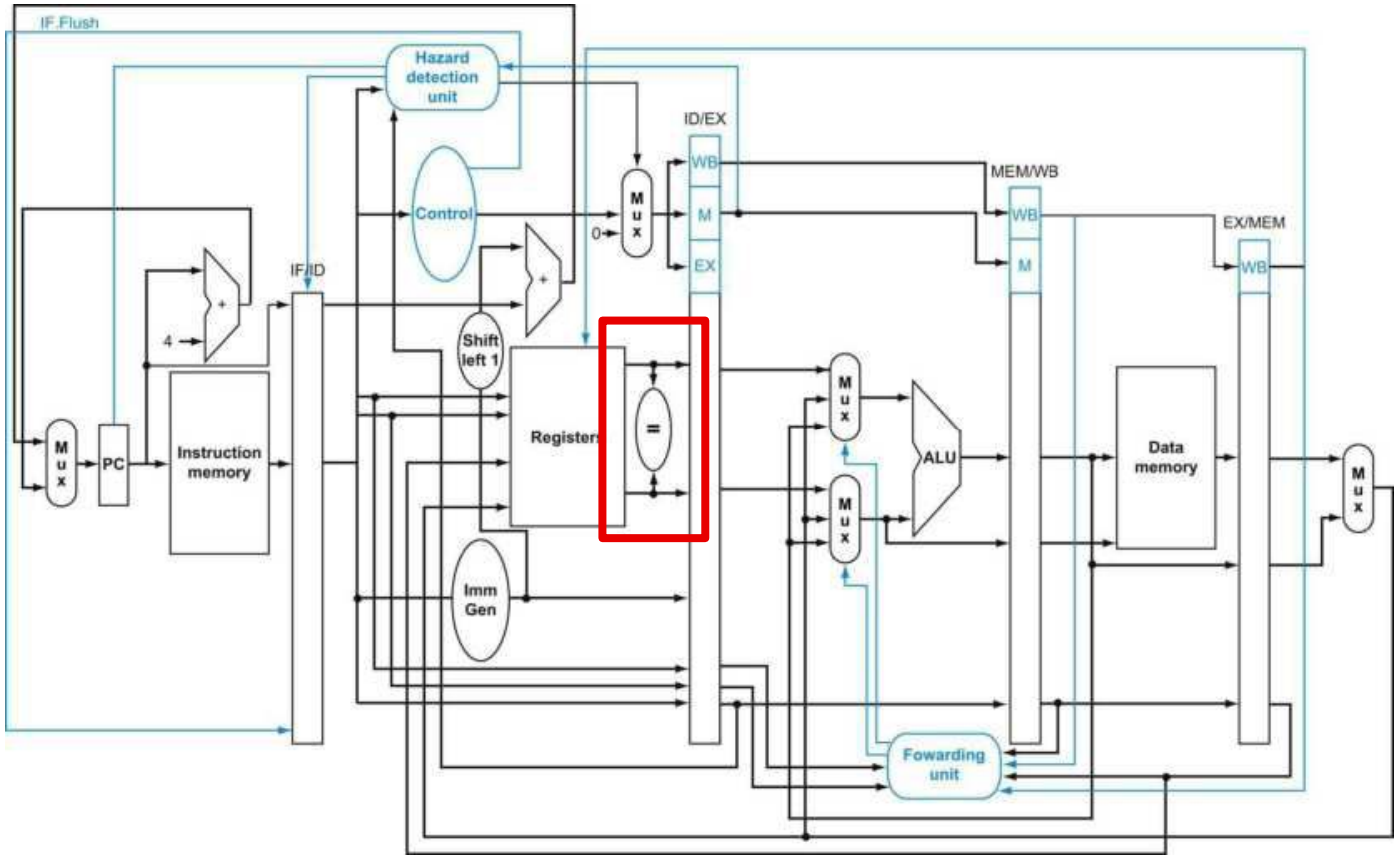
Example: Branch Taken



Example: Branch Taken



RAW Hazards for Branch Comparison?



EXAMPLE:

```
add x5, x2, x1
beq x4, x5, label
```

Role of the Compiler

- Reordering the instruction
  - Delay slot*: moving an independent instruction between the branch and the next instruction.*
- Increasing the chance of not-taken
- Loop unrolling
- …

Anything else?

- Memory hierarchy → we will get to this later!
- Multi-core?
- Advanced design choices?

Key Ideas

- How to make CPUs faster?

Key Ideas

- How to make CPUs faster?
  - Parallelization → doing more things at the same time
    - Pipelining was one but we will talk about more things

Key Ideas

- How to make CPUs faster?
  - Prediction/Speculation → Remove the need for stalling

Key Ideas

- How to make CPUs faster?
  - Improving Prediction/Speculation → remove the need for flushing

Key Ideas

- How to make CPUs faster?
  - Out-of-order execution → not waiting for unnecessary things!

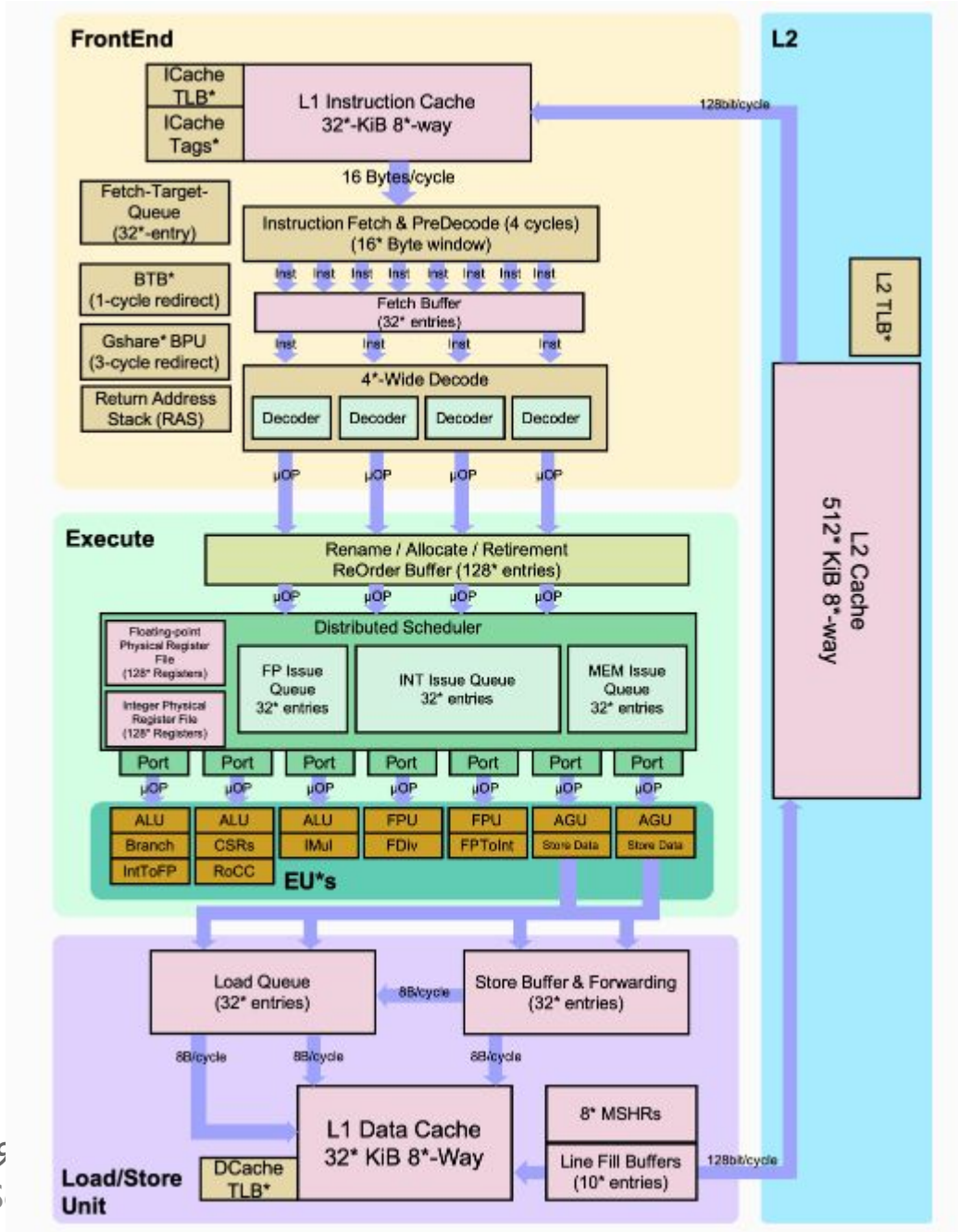
Key Ideas

- How to make CPUs faster?
  - Faster read/write from/to memory
    - Cache
    - Prefetch
    - Prediction

Nader Sehatbakhsh <nsehat@ee.ucla.edu>

BOOM Core

- Out-of-order execution



Acknowledgement

- This course is partly inspired by the following courses created by my colleagues:
  - 6.888 MIT (Yan)
  - CS 598 UIUC (Fletcher)
  - (list will be updated)

End of Presentation