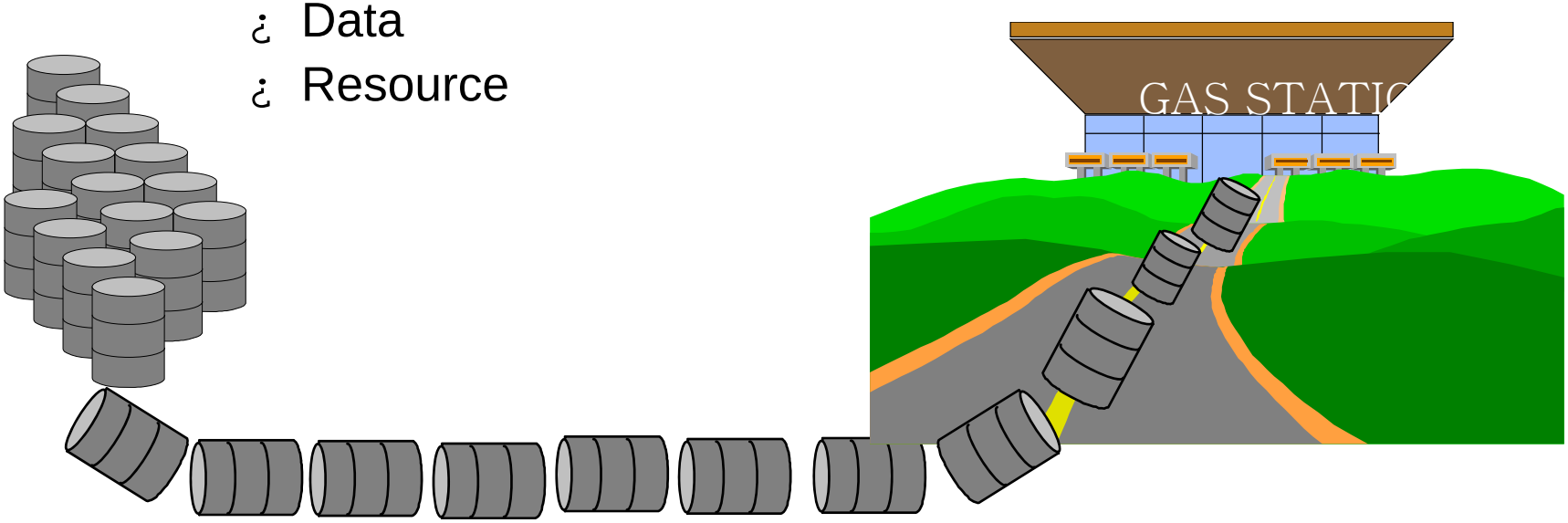
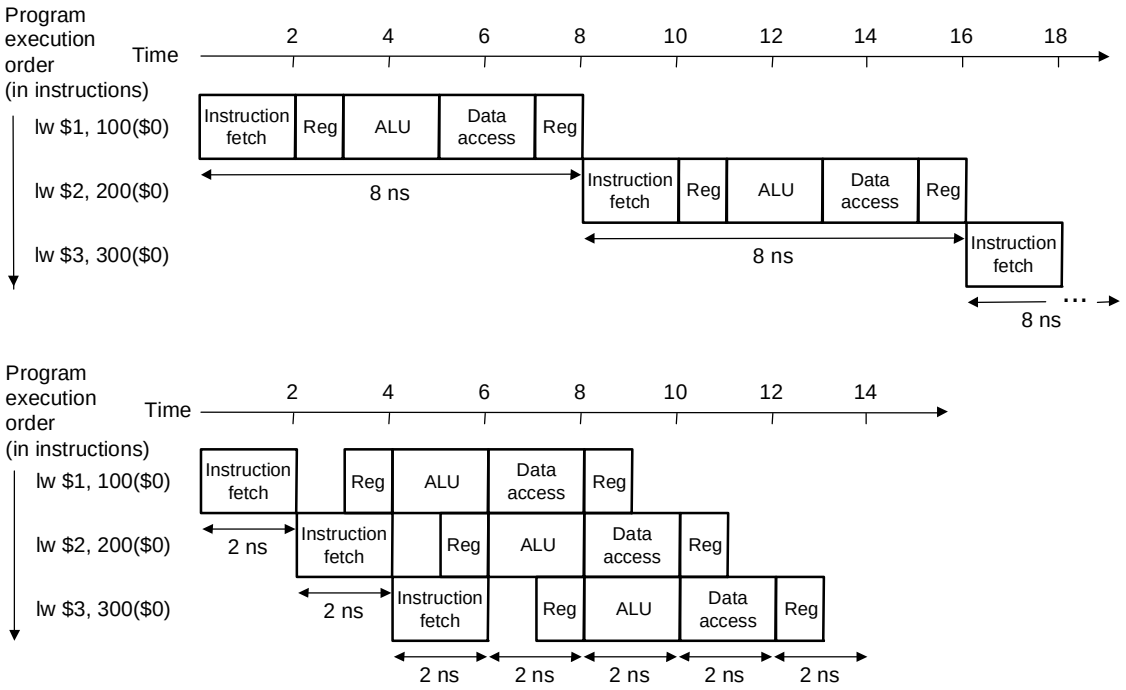
Pipelined Processor Design

- Pipelining
  - ¿ Static In-order Pipeline
    - ∅ Dynamic out-of-order Pipeline
    - ∅ Multiple pipeline
- Hazards
  - ¿ Control
  - ¿ Data
  - ¿ Resource



Pipelining

- Improve perfomance by increasing instruction throughput



Ideally, $\text{Speedup}_{\text{pipeline}}$ = Pipeline Depth

after start-up to fill up the pipe

Pipelining

- What makes it easy
  - ¿ all instructions are the same length
  - ¿ just a few instruction formats
  - ¿ memory operands appear only in loads and stores
- What makes it hard?
  - ¿ structural hazards: suppose we had only one memory
  - ¿ control hazards: need to worry about branch instructions
  - ¿ data hazards: an instruction depends on a previous instruction

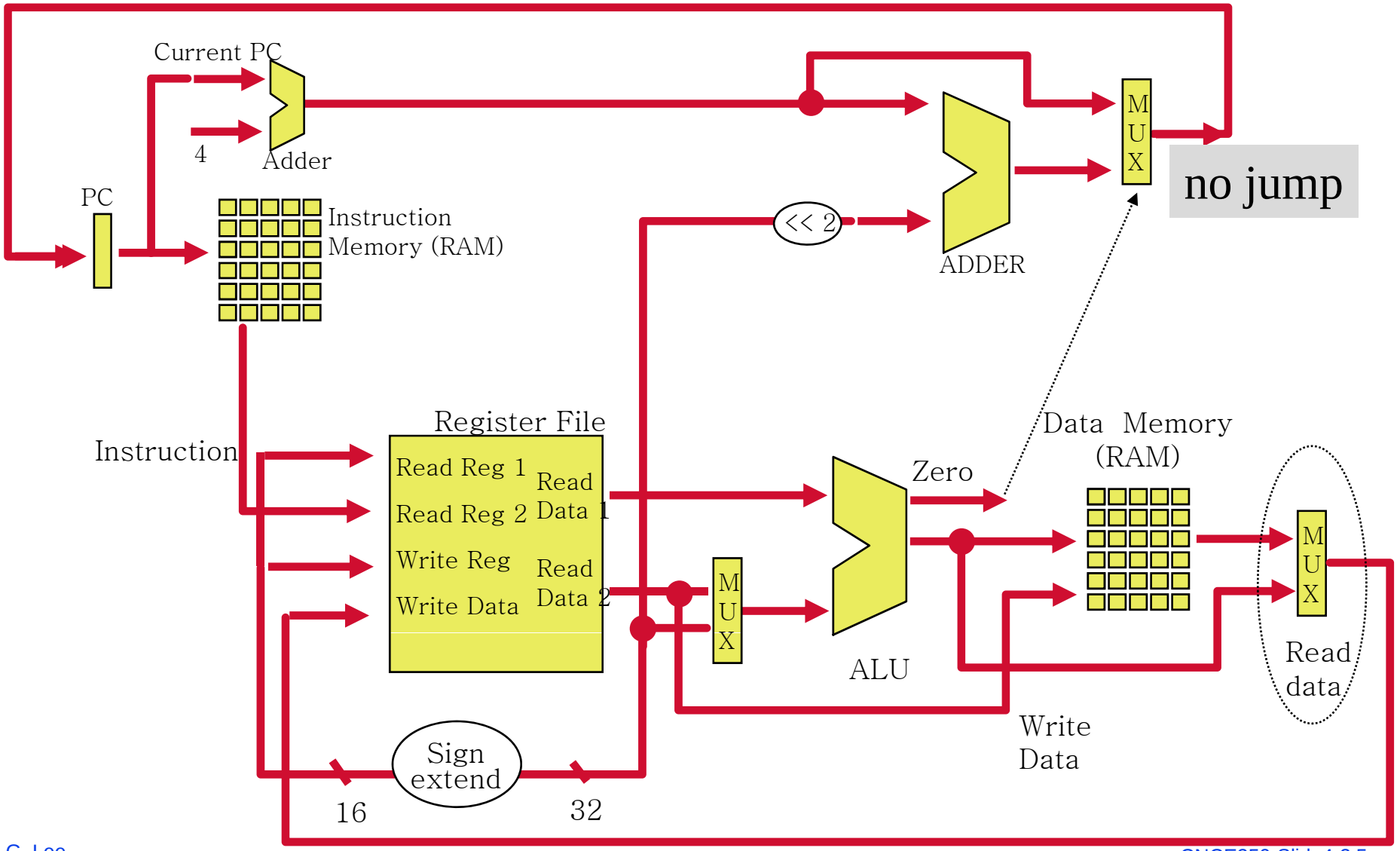
MIPS Pipeline Stages

- Stage 1: Instruction Fetch (IF)
- Stage 2: Instruction Decode (ID)
- Stage 3: Execute (EX)
- Stage 4: Memory Access (M)
- Stage 5: Write Back (to register file) (WB)

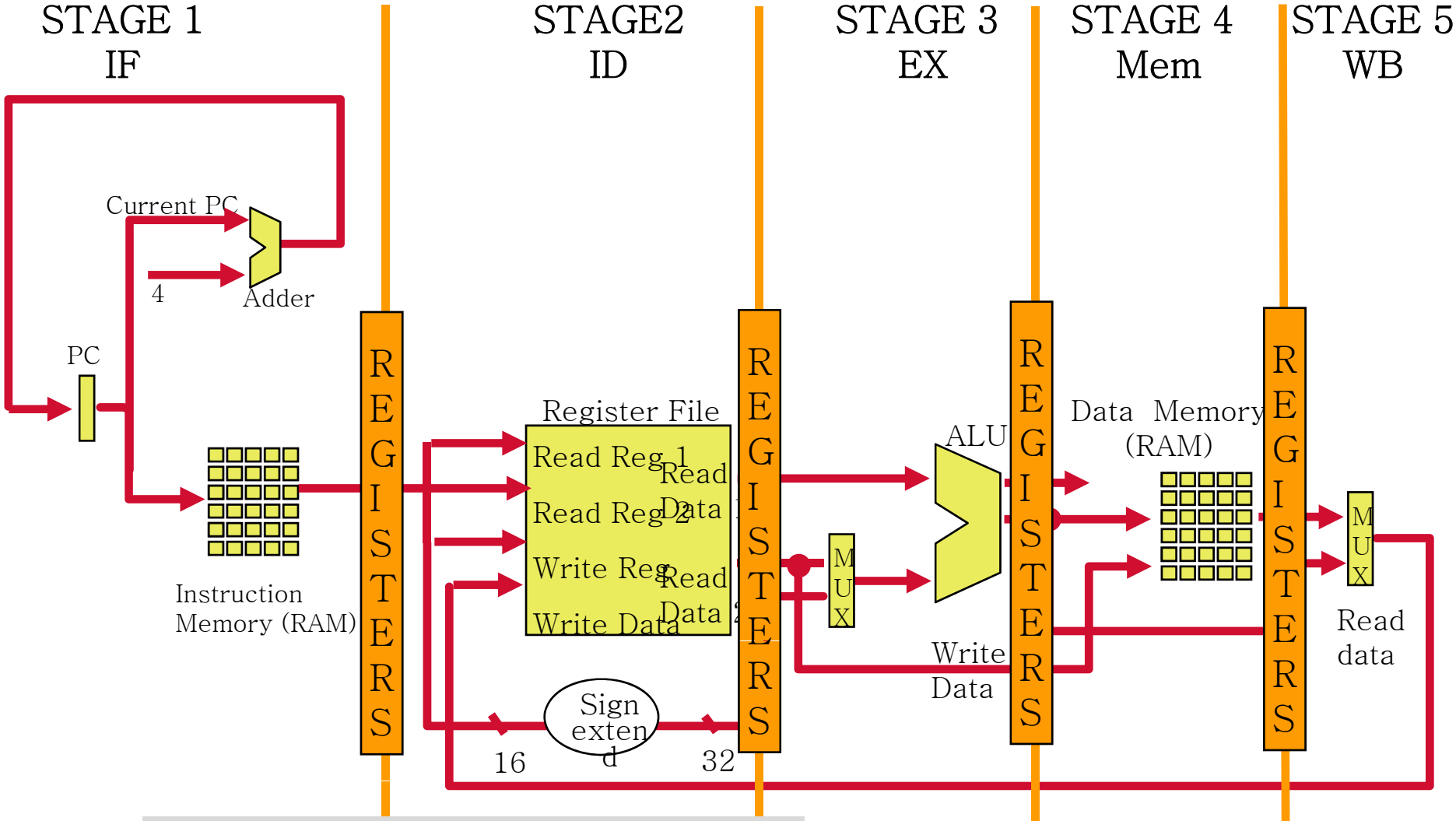
NOTE: slight change in stages

•Dividing data path with intermediate storage
•Generating control signal
•Taking care of conflicts and dependency (hazards)

Recall: Complete Single-cycle Datapath

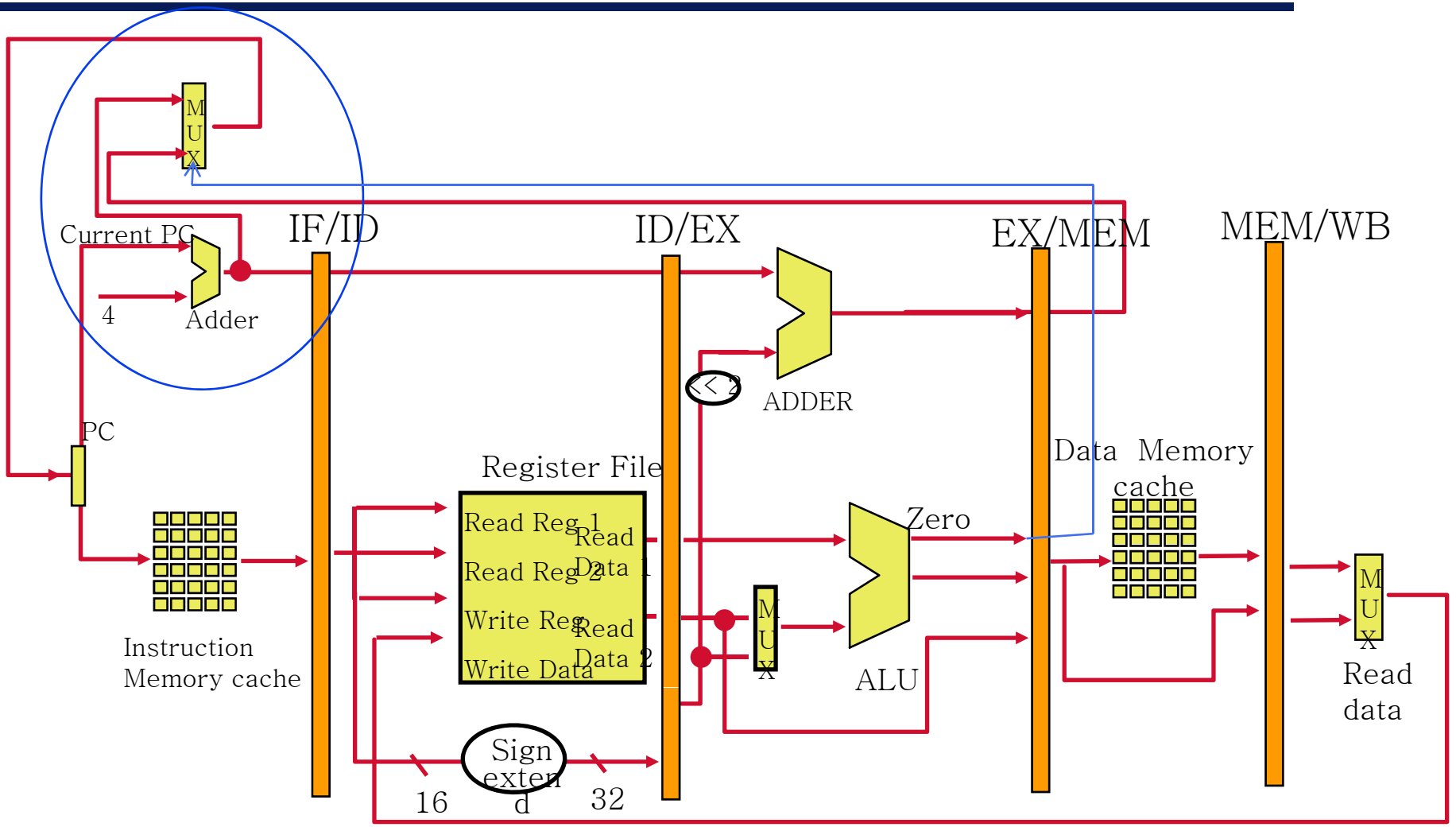


Partition the Datapath & Separate the Different Stages

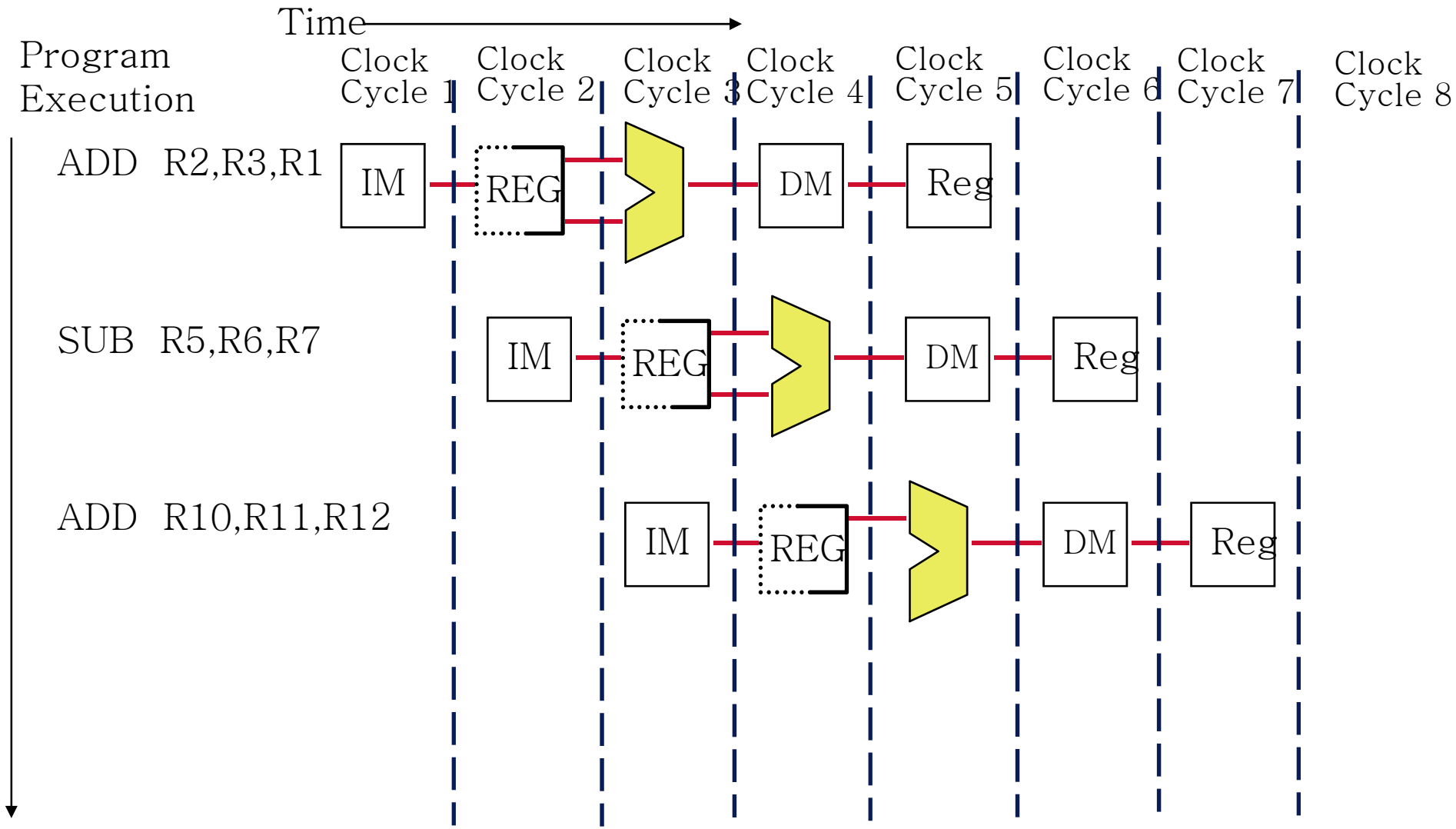


based on single cycle data path

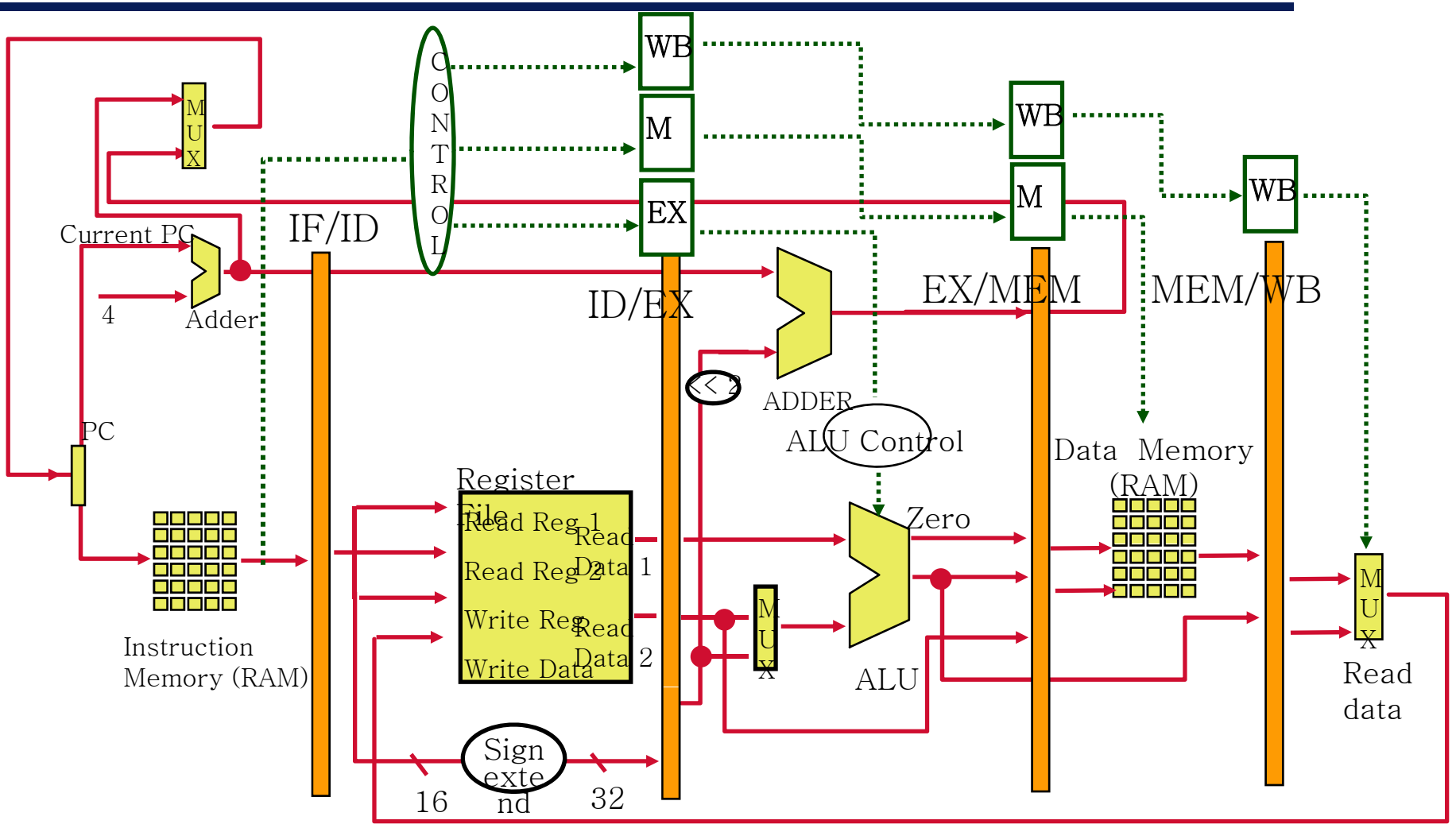
5 Stage Pipeline with BEQ



Flow of Instructions Through Pipeline



Pipeline Control

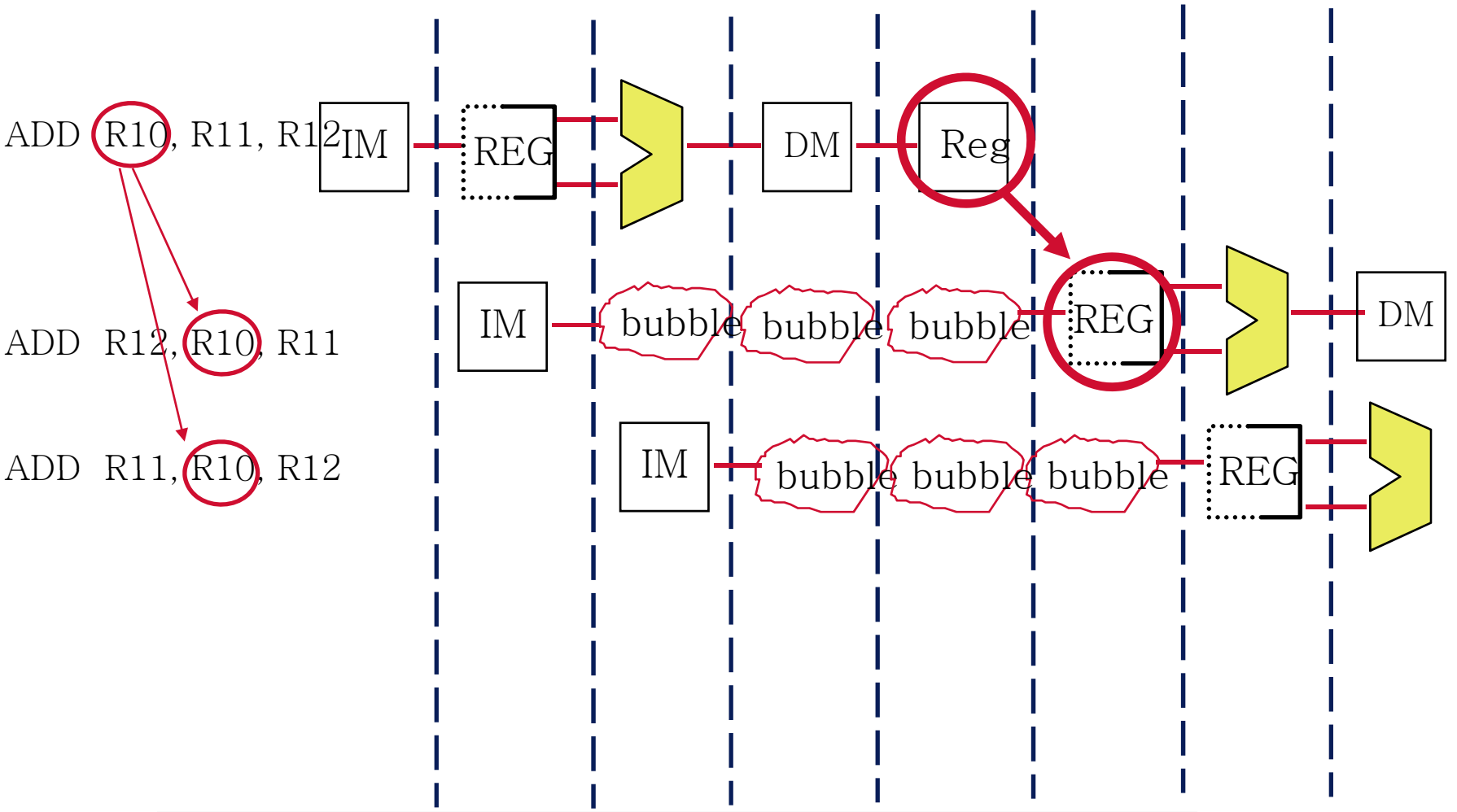


carry instruction itself + PC + control signals

Data Hazards

- Programs assume instructions are executed sequentially with one instruction completing before the next one begins
  - ¿ Usually the compiler assumes the single machine model
- Pipelining violates this assumption
- Dependencies can occur between instructions executing concurrently within the pipeline - if the dependencies are based on data requirements, we call them Data Hazards
- Types of data hazards
  - ¿ Read-after-write (RAW)
    - Ø A true dependency
  - ¿ Write-after-read (WAR)
    - Ø Artificial dependency due to register assignment
  - ¿ Write-after-write (WAW)
    - Ø Artificial dependency due to register assignment

Solution 1 : Stall



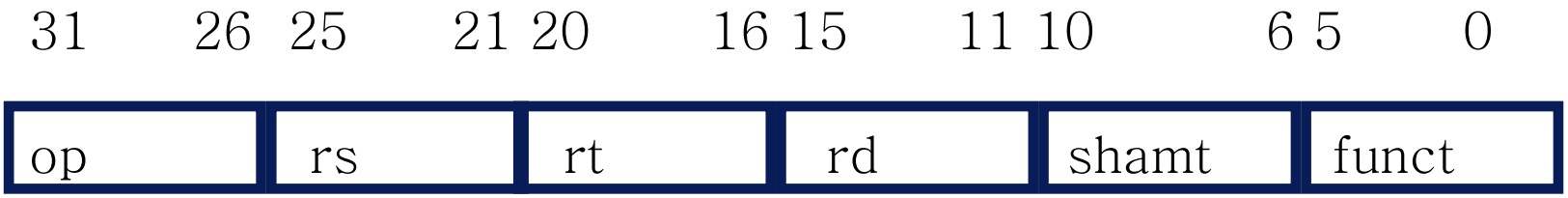
bubble: delay slot – nop or actual delay or something else

Stall Conditions

- Need to detect data hazard
  - ¿ Occurs when one instruction tries to read result from previous instruction that hasn't completed yet.
    - ℓ Block depending instruction to go further
    - Ø Stop IF
  - ¿ Specifically,
    - Ø When Instruction in Execute stage tries to read a register that an instruction in the MemAcc or WB stages will write back to the Register File
  - ¿ Textbook Notation
    - Ø ID/EX.RegisterRs refers to the number of the first source register found in the pipeline register ID/EX.
    - Ø ID/EX. RegisterRt refers to the number of the second source register found in the pipeline register ID/EX.

Recall What an Instruction Looks Like

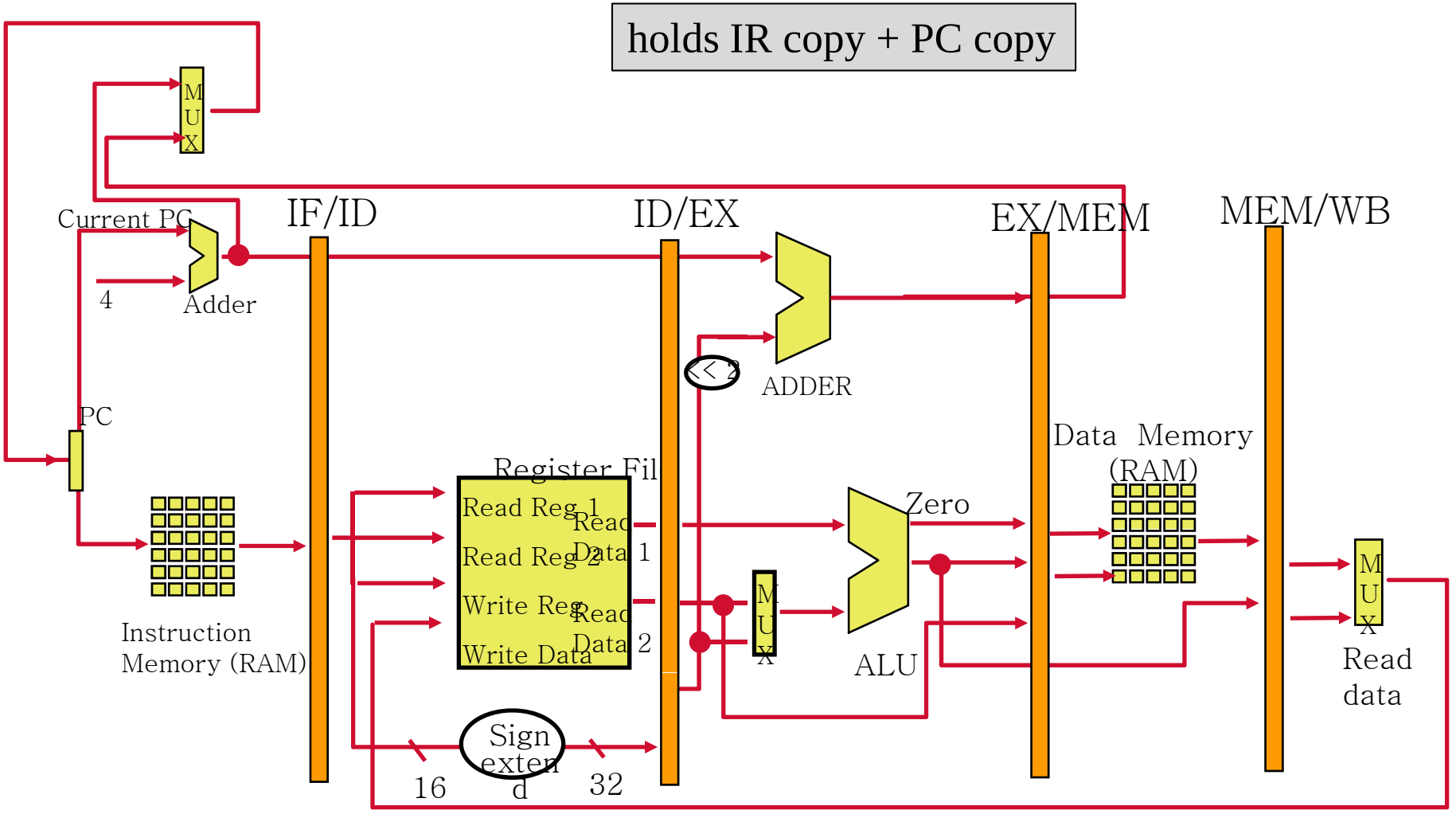
- add R8, R17, R18
  - ¿ is stored in binary format as
    - ø 00000010 00110010 01000000 00100000
  - ¿ MIPS lays out instructions into “fields”



- ¿ op operation of the instruction
- ¿ rs first register source operand
- ¿ rt second register source operand
- ¿ rd register destination operand
- ¿ shamt shift amount
- ¿ funct function (select type of operation)

g1

Remember the Registers In Between Each Stage



g1 at IF/ID a fecthed instruction resides, while at ID/EX a decoded and precalculated 16-bit immediate extended to 32-bit and register numbers reside along with instruction itself and its PC+4

ghlee, 2013-04-15

Data Hazard Stall Conditions

Hazard Detection:

1a	EX/MEM. RegisterRd	==	ID/EX.RegisterRs
1b	EX/MEM.RegisterRd	==	ID/EX. RegisterRt
2a	MEM/WB. RegisterRd	==	ID/EX. RegisterRs
2b	MEM/WB. RegisterRd	==	ID/EX. RegisterRt

Note:

1. the idea is comparing the instruction at ID stage with previous instructions; if matches, stall the pipe.
2. Multiple IRs(Instruction registers) = registers in between

The instruction passed the ID stage is called "**instruction issued**".

So, rephrasing 1 can be: compare the instruction at decode stage with the instructions issued.

Example

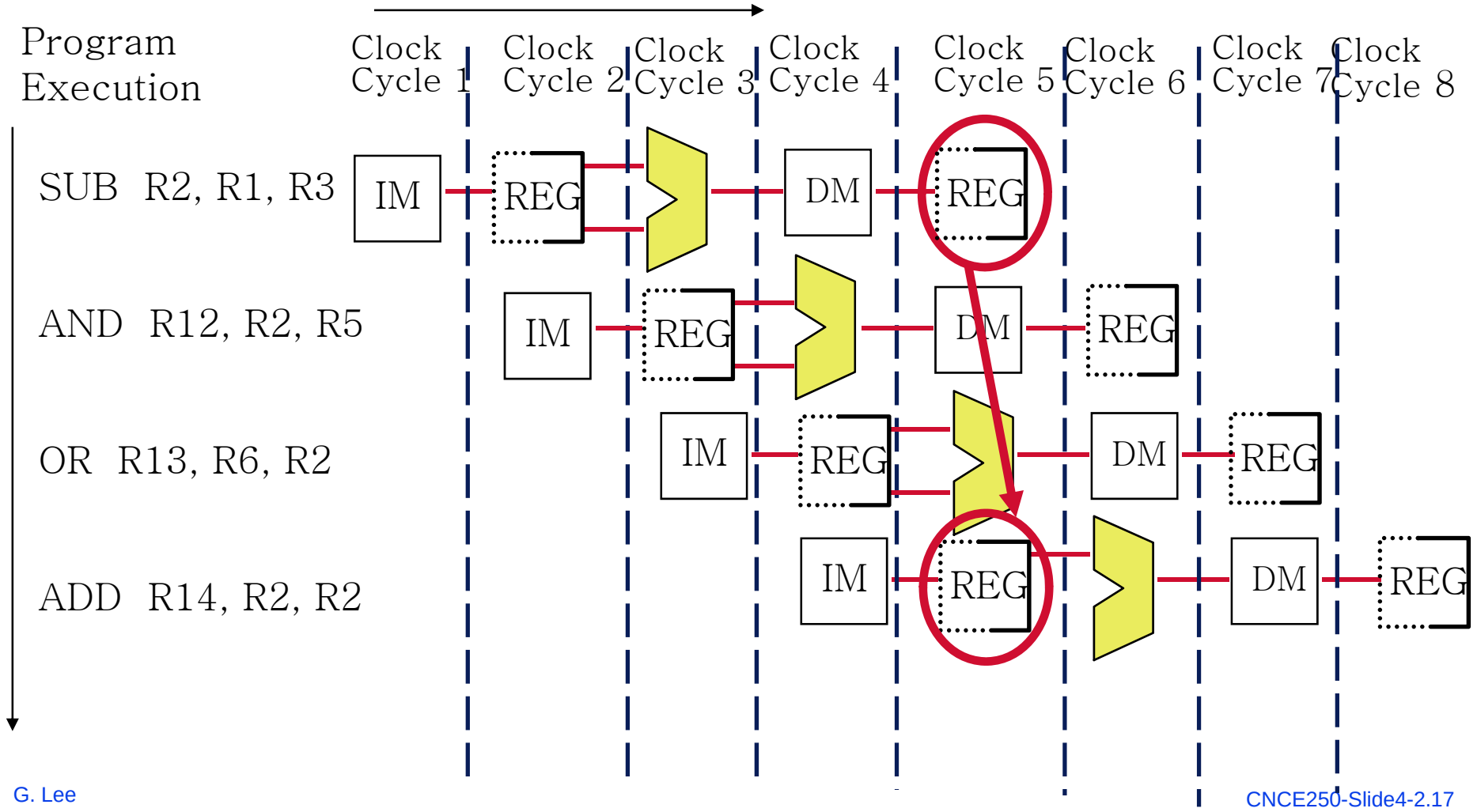
sub	R2, R1, R3	Rd = R2	Rs = R1	Rt = R3
and	R12, R2, R5	Rd = R12	Rs = R2	Rt = R5
or	R13, R6, R2	Rd = R13	Rs = R6	Rt = R2
add	R14, R2, R2	Rd = R14	Rs = R2	Rt = R2
sw	R15, 100(R2)	Rd = R15	Rs = R2	Rt = XX

- SUB-AND Hazard
  - ¿ EX/MEM.RegisterRd == ID/EX. RegisterRs == R2
- SUB-OR Hazard
  - ¿ MEM/WB.RegisterRd == ID/EX. RegisterRt == R2

- Do we care about the interaction between sub (instruction 1) and add (instruction 4)?

No Dependence Between Instruction 1 and 4

assume write and read to the same register in the same cycle



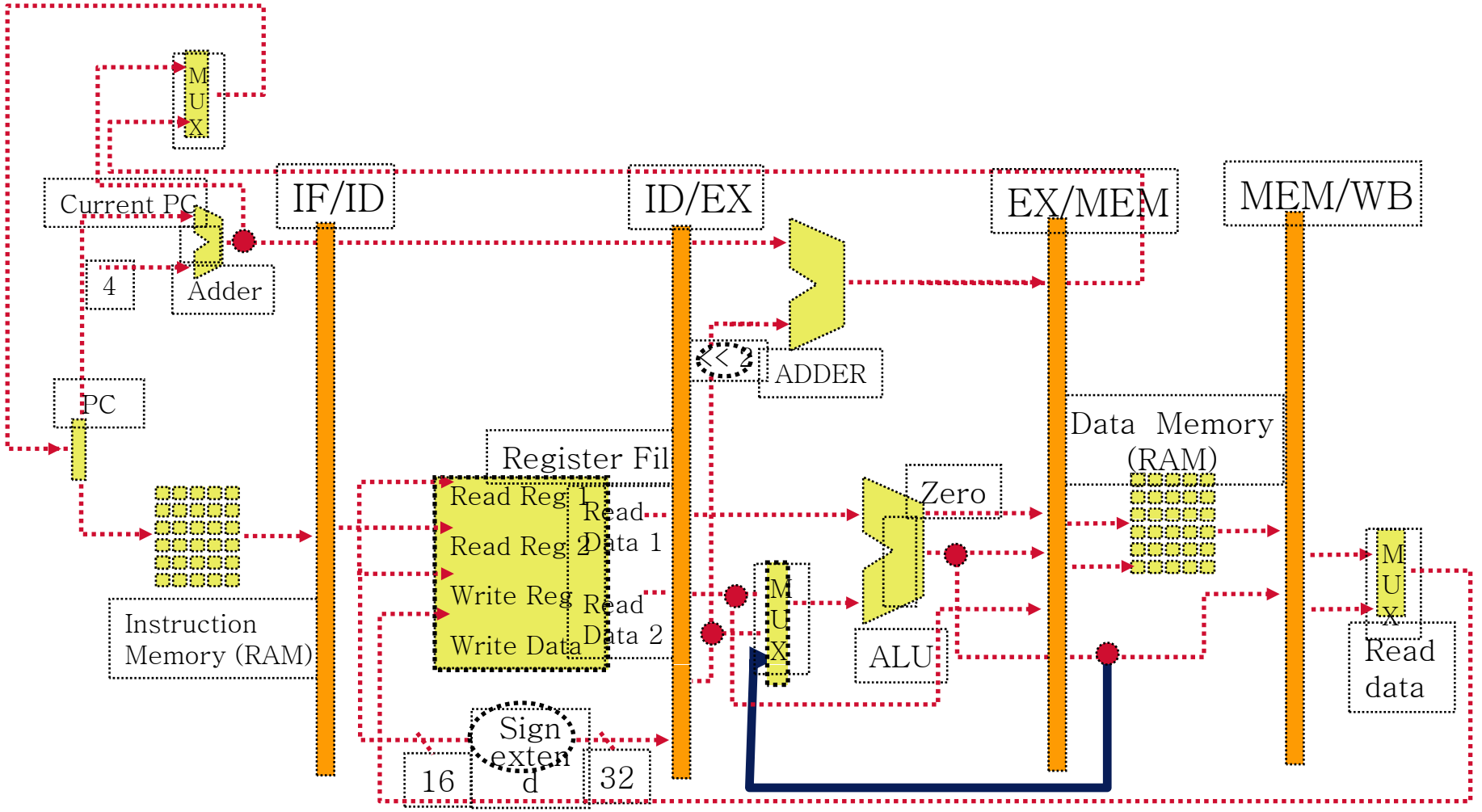
How Do We Stall the Pipeline?

- Compiler can insert NOPS
- Hardware Can Simulate NOPS
  - ¿ Do not activate the IF stage until hazard condition is removed
  - ¿ Forward NOP, i.e. no active control signals for WB or PC-write or Memory-write

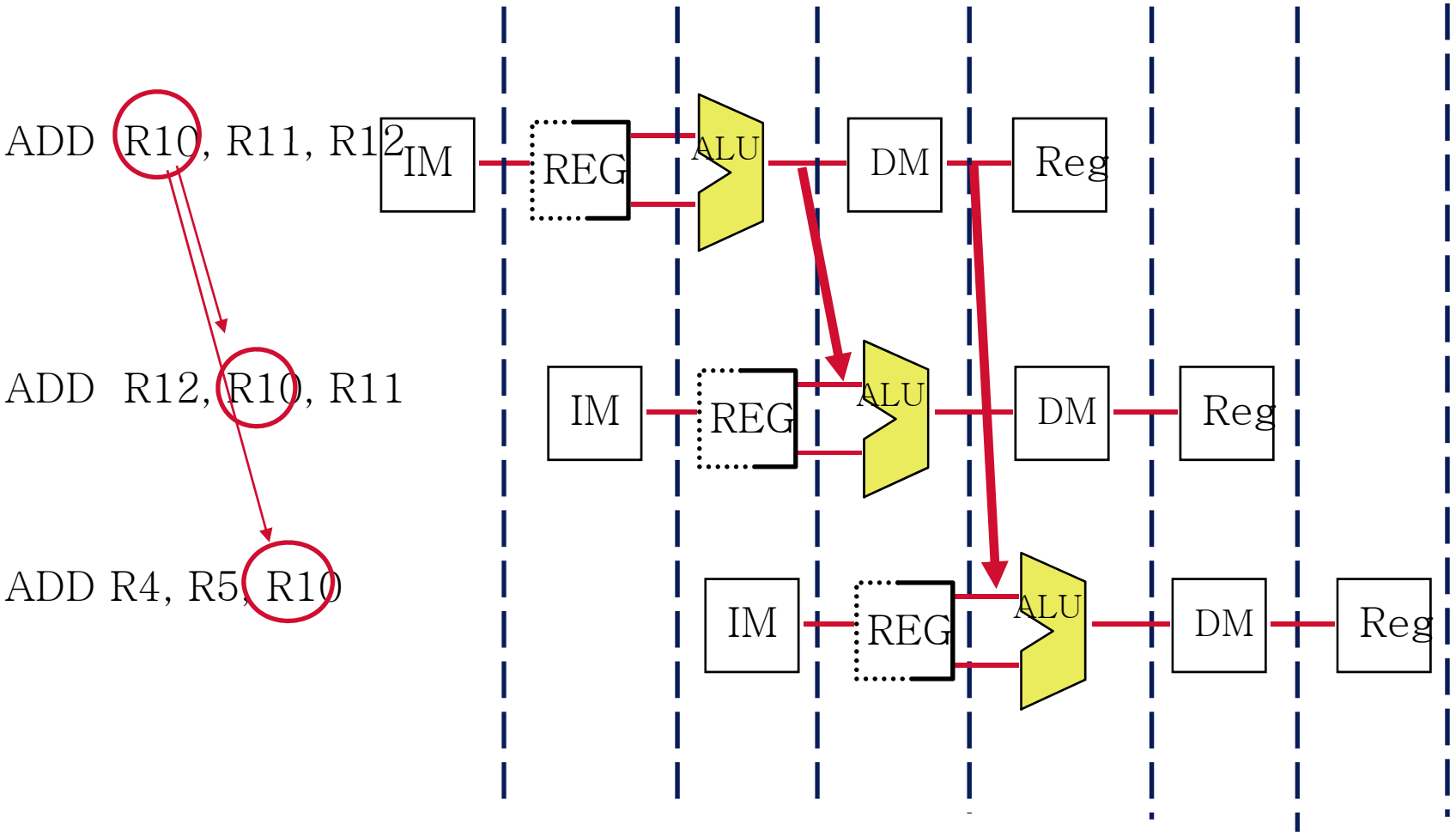
Reducing Data Hazards: Forwarding

- Data may be already computed
  - just not in the Register File yet
- Bypass register file and directly forward to where needed
  - new bypass data path

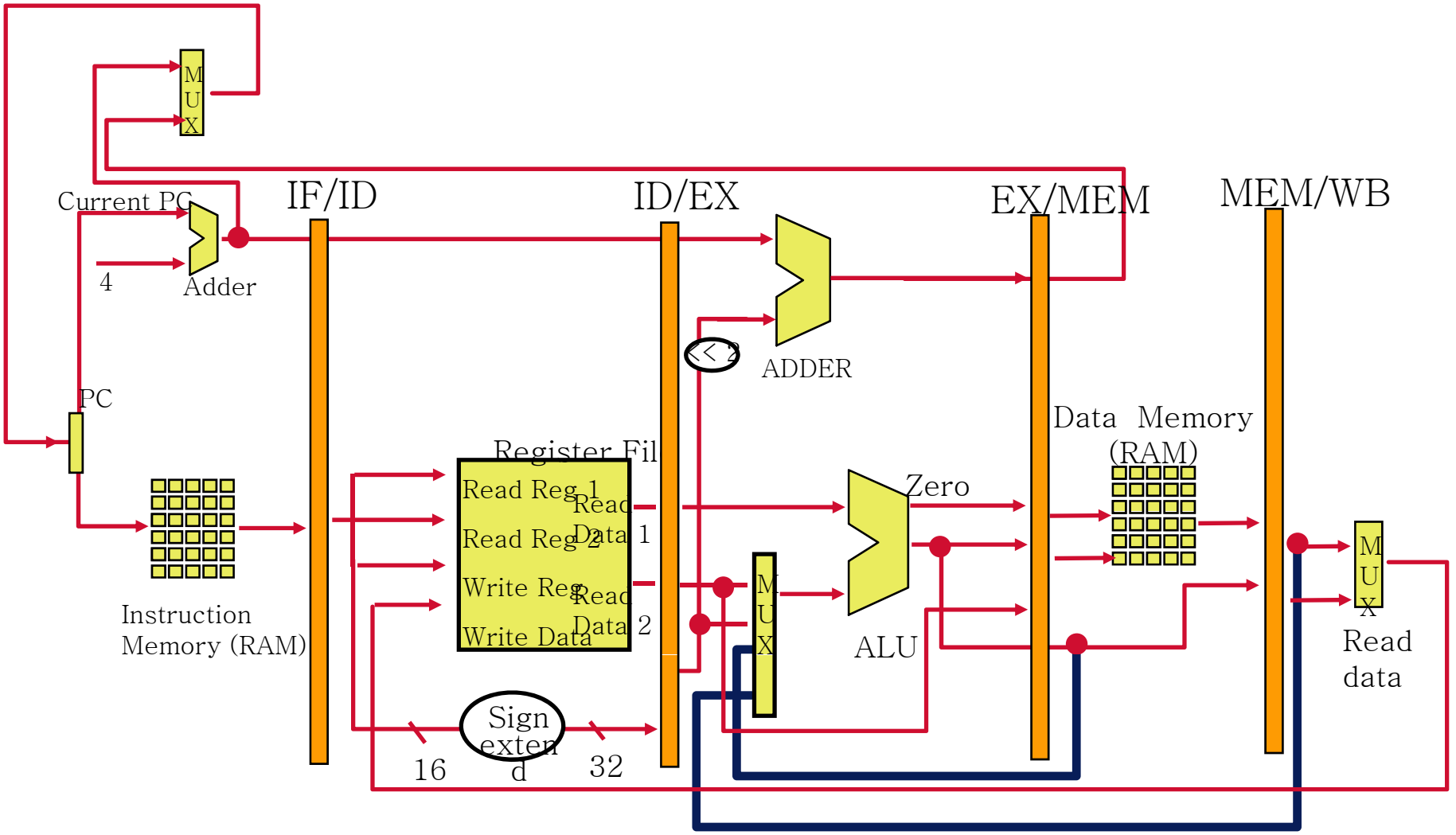
Additions to the Datapath for Forwarding



Forwarding Continued

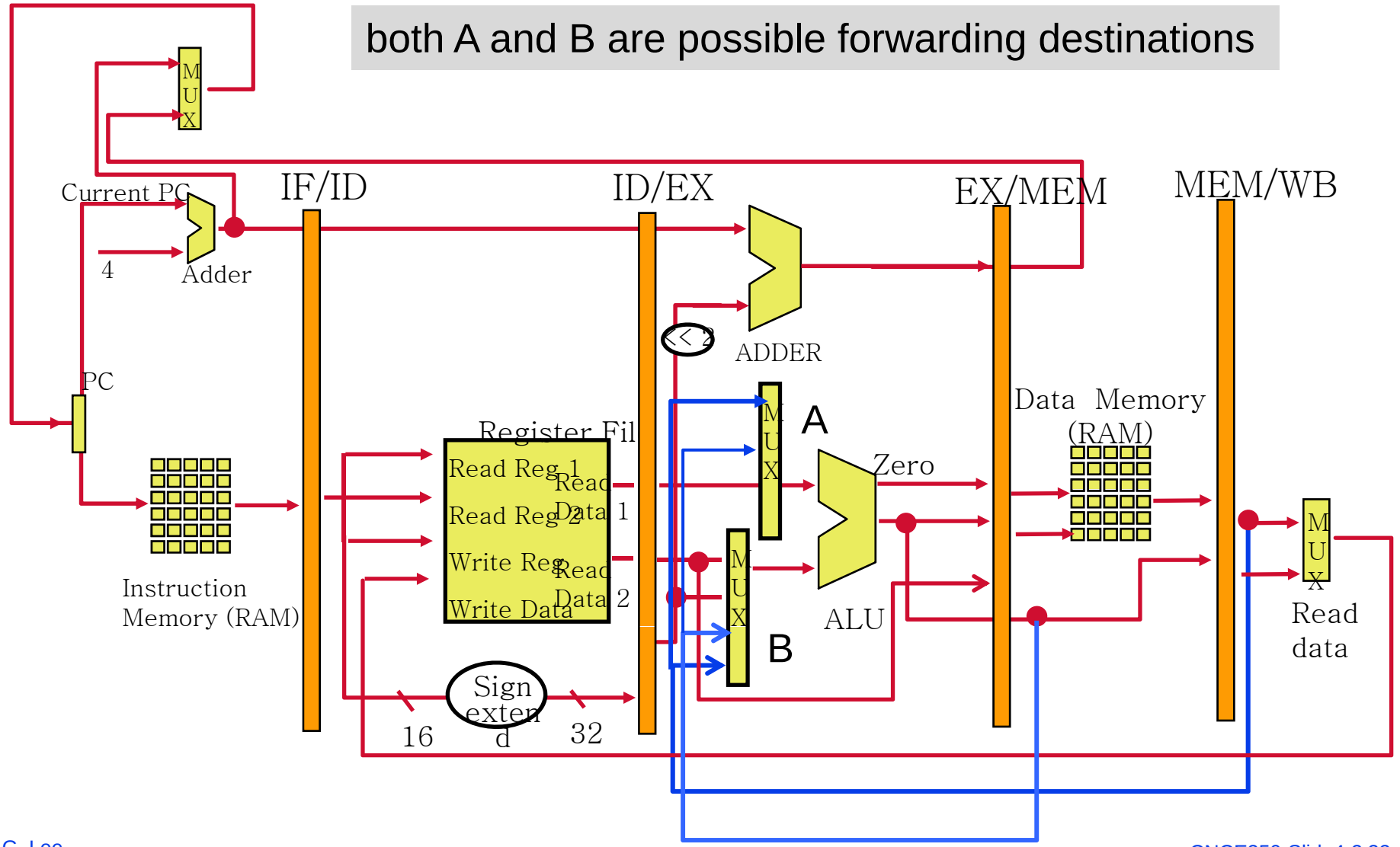


More Additions to the Datapath

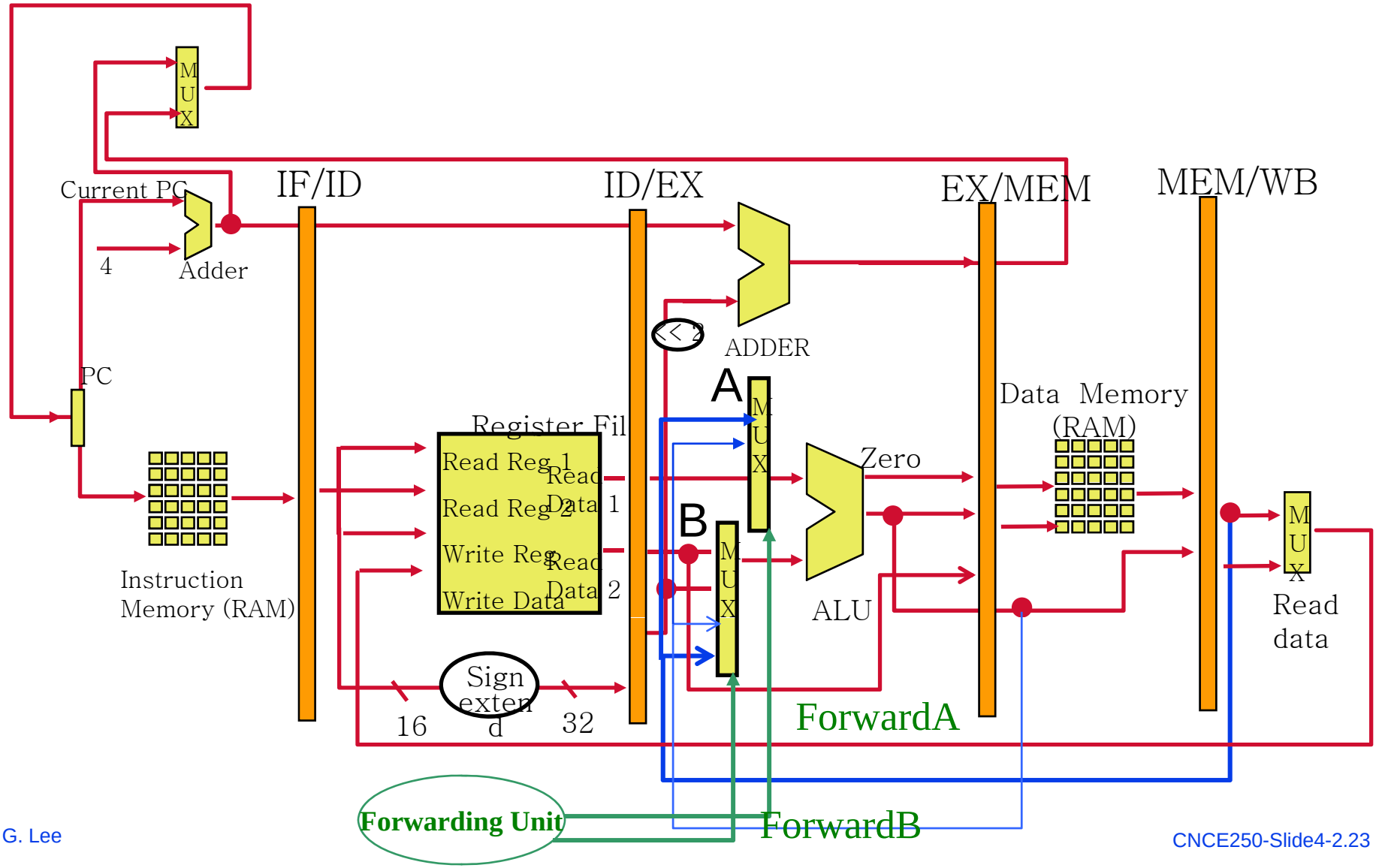


More Additions to the Datapath

both A and B are possible forwarding destinations



Add forwarding unit for proper control



Add control part for forwarding

- Two cases – at EX and at Mem
  - ¿ At Ex :

```
add r1, r2, r3                or      add r1, r2,r3
sub r4, r1, r5                        or  r6, r7, r1
```

when sub/or instruction reads operands of r1, the producer(add) is in EX stage. The value of r1 can be read from EX/MEM register.

```
if (EX/MEM.RegWrite
    and (EX/MEM.RegisterRd != 0)
   and (Ex/MEM.RegisterRd = ID/EX.RegisterRs)
ForwardA = 2;

if (EX/MEM.RegWrite
    and (EX/MEM.RegisterRd != 0)
   and (Ex/MEM.RegisterRd = ID/EX.RegisterRt)
ForwardB = 2;
```

Add control part for forwarding(cont')

- At Mem :

```
add r1, r2, r3                    add r1, r2,r3
add r8, r3, r4                    add r8, r3, r4
sub r4, r1, r5                    or  r6, r7, r1
```

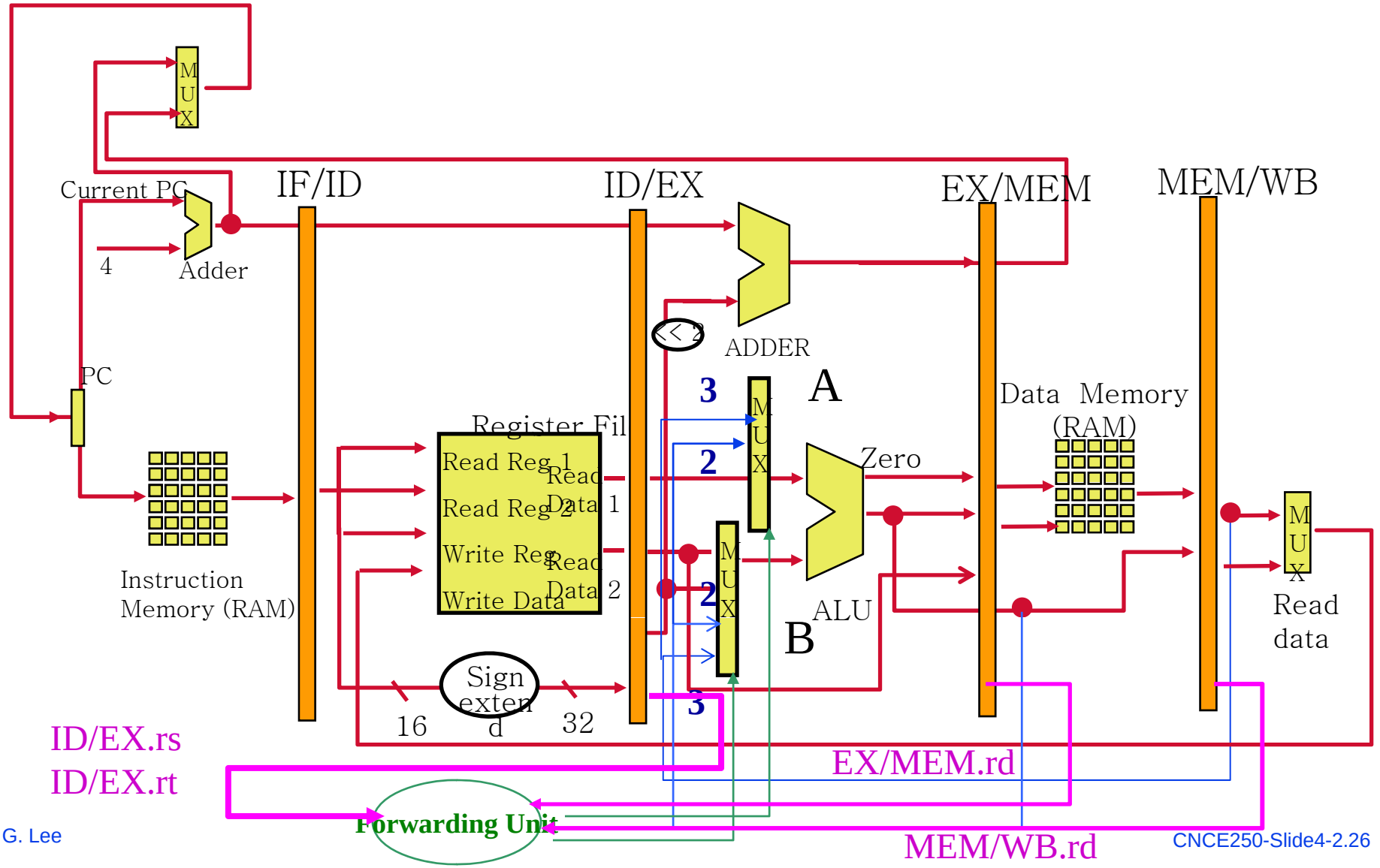
or

when sub/or instruction reads operands of r1, the producer(add) is in MEM stage. The value of r1 can be read from MEM/WB register.

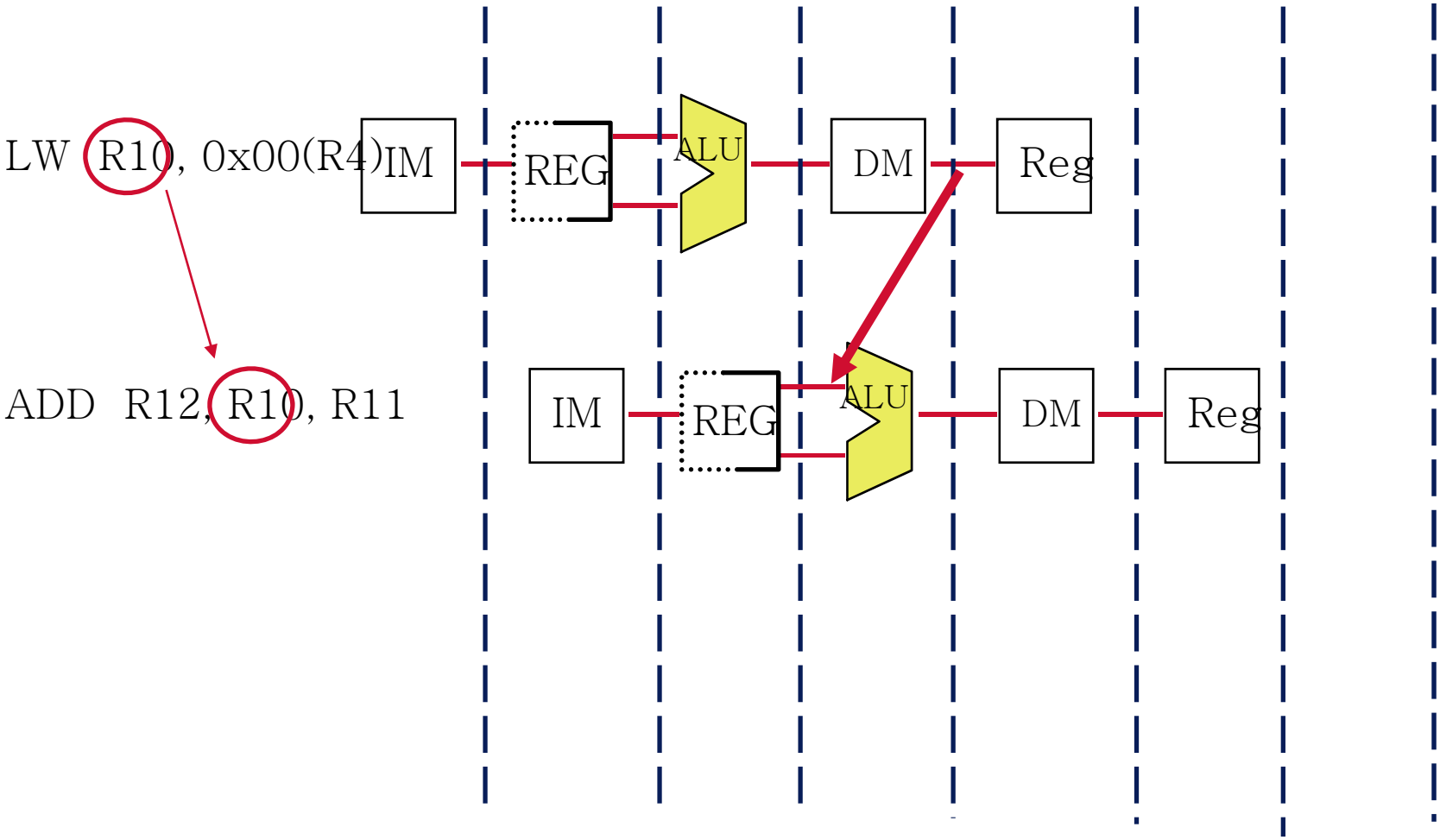
```
if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd != 0)
   and (MEM/WB.RegisterRd = ID/EX.RegisterRs)
ForwardA = 3;

if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd != 0)
   and (MEM/WB.RegisterRd = ID/EX.RegisterRt)
ForwardB = 3;
```

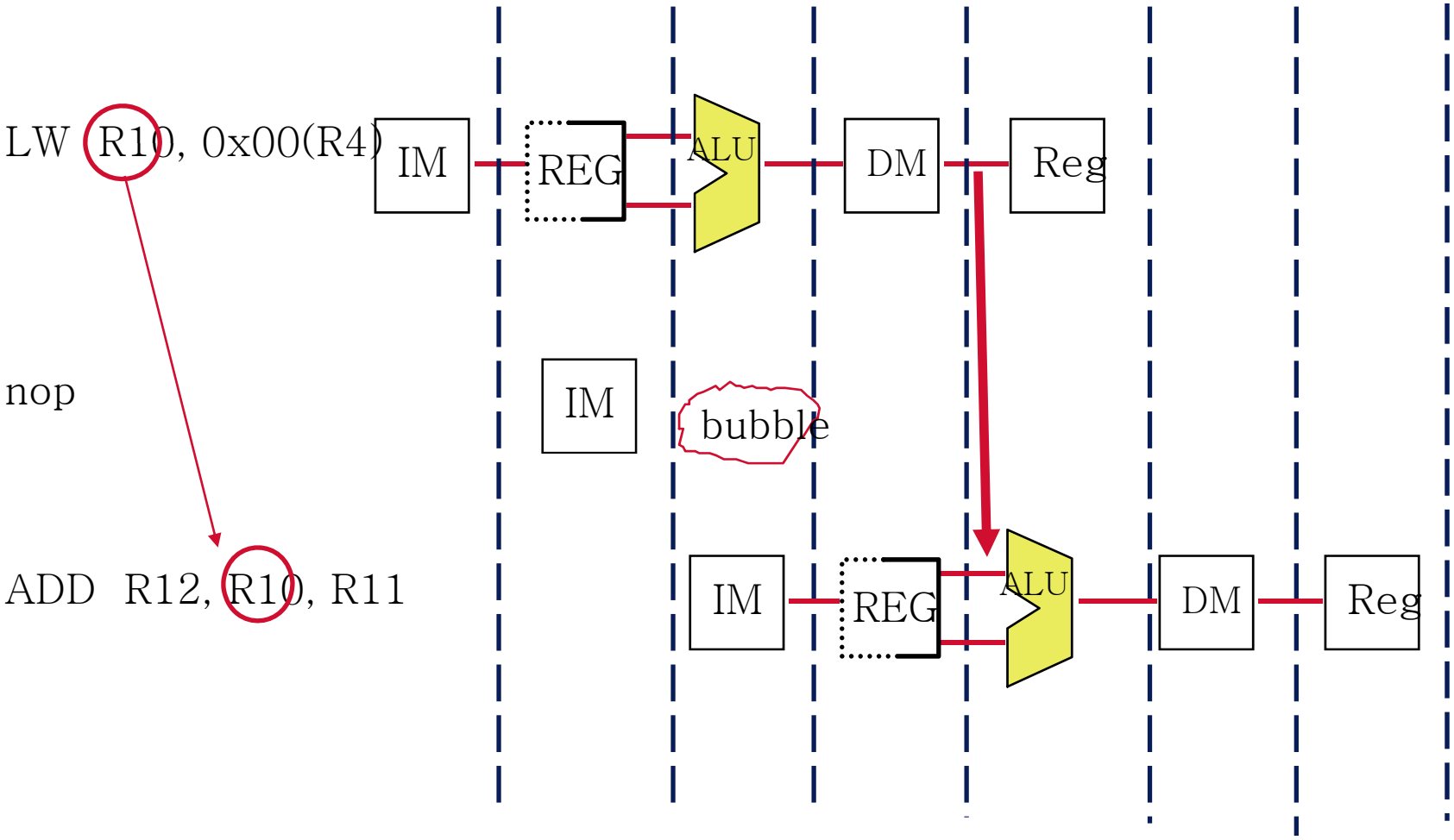
Complete forwarding unit



Forwarding Doesn't Always Work



Loads and Stores Require a Load Delay Slot



MIPS Load-Delay Slot

- MIPS exposed the load-delay slot to the compiler
  - ¿ This makes it part of the architecture, not just an implementation detail
  - ¿ Therefore, it's up to the compiler (or assembly code writer) to make sure that the instruction after a load does not depend on the result of the load
- An alternative would have been to force the hardware to detect the data hazard and stall the pipeline
  - ¿ Most of today's architectures detect the hazard and stall

Example of Forwarding and Load Delay

- Rewrite the code assuming a machine without forwarding (by inserting nops).

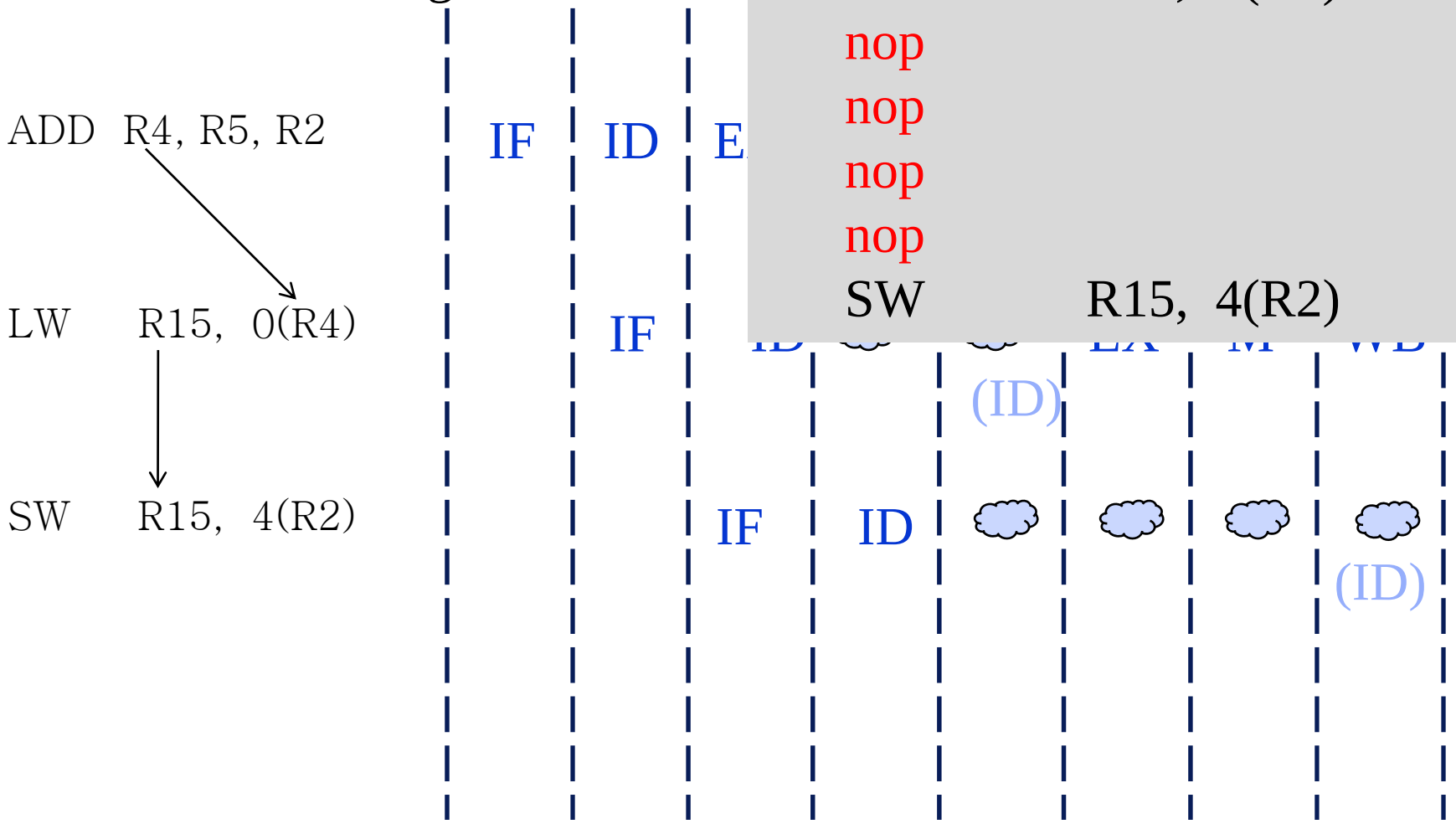
```
ADD     R4, R5, R2
LW      R15,  0(R4)
SW      R15,  4(R2)
```

- Rewrite the code assuming forwarding

Solution Templete

```
ADD     R4, R5, R2
nop
nop
LW      R15,  0(R4)
nop
nop
nop
nop
SW      R15,  4(R2)
```

Without Forwarding



Solution Templete

```
ADD     R4, R5, R2
LW      R15,  0(R4)
nop
ADD     R5,  R15, R6
```

With Forwarding