COMP311: *COMPUTER ORGANIZATION!*

Lecture 23: Intro to Memory Hierachy

tinyurl.com/comp311-sp26

Logistics Update!

- Written Assignment 6
  - *Due tomorrow*
- Lab 5 due LDOC (4/27)
- Office Hours Ending LDOC
- Final Review Session on First Reading Day

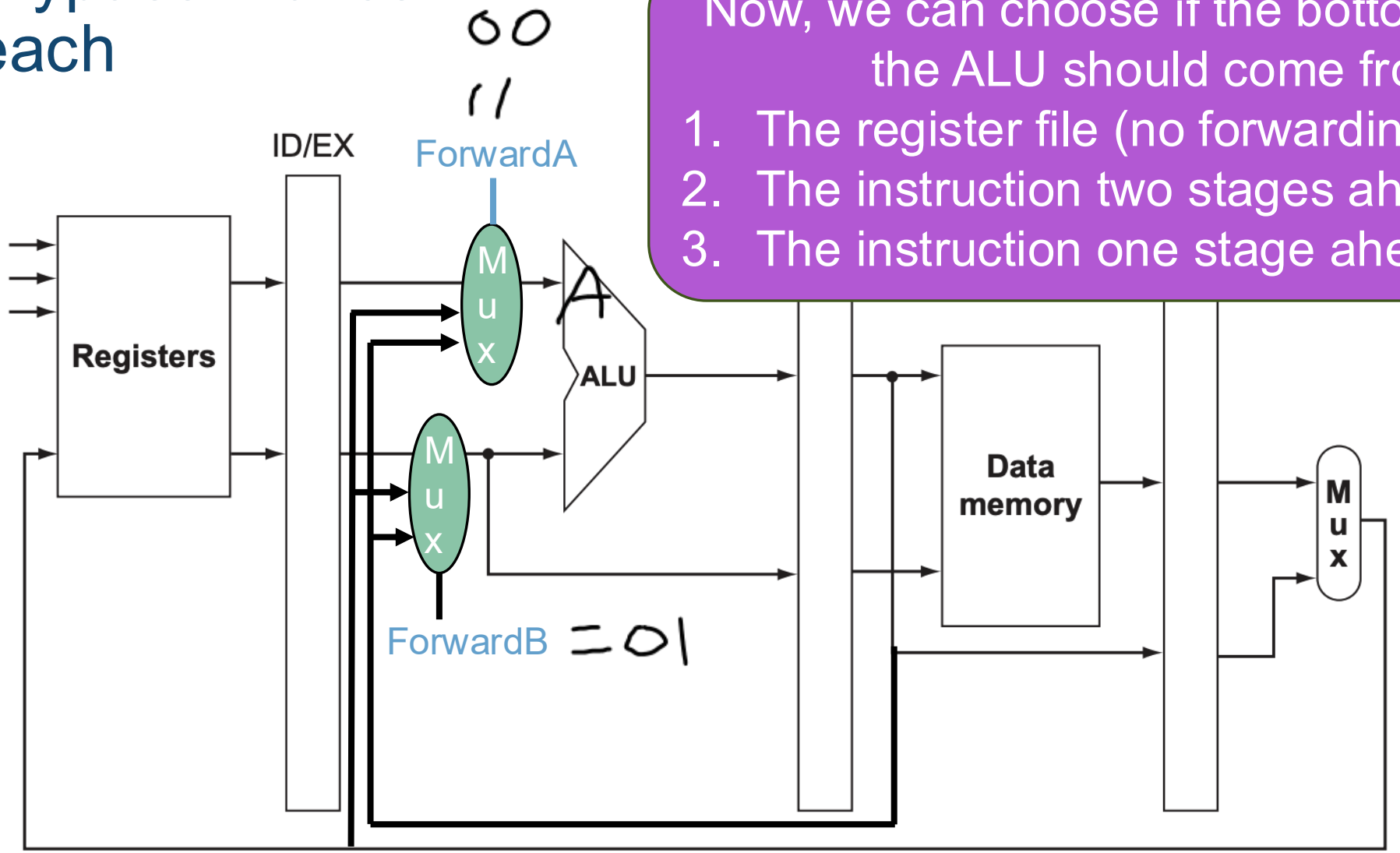
(4/28)

REVIEW: SOURCE BYPASSING

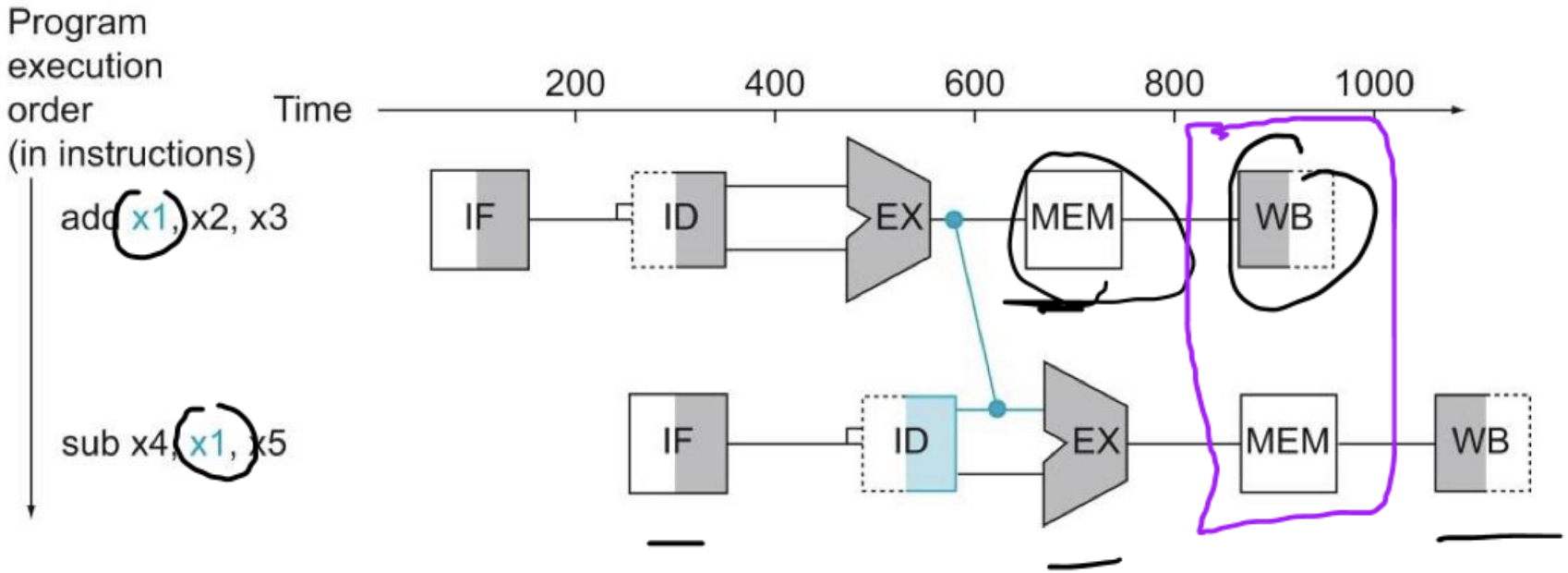
Source Bypass Muxes: 3 case each

Now, we can choose if the bottom input of the ALU should come from

1. The register file (no forwarding) -- 00
2. The instruction two stages ahead -- 01
3. The instruction one stage ahead -- 10



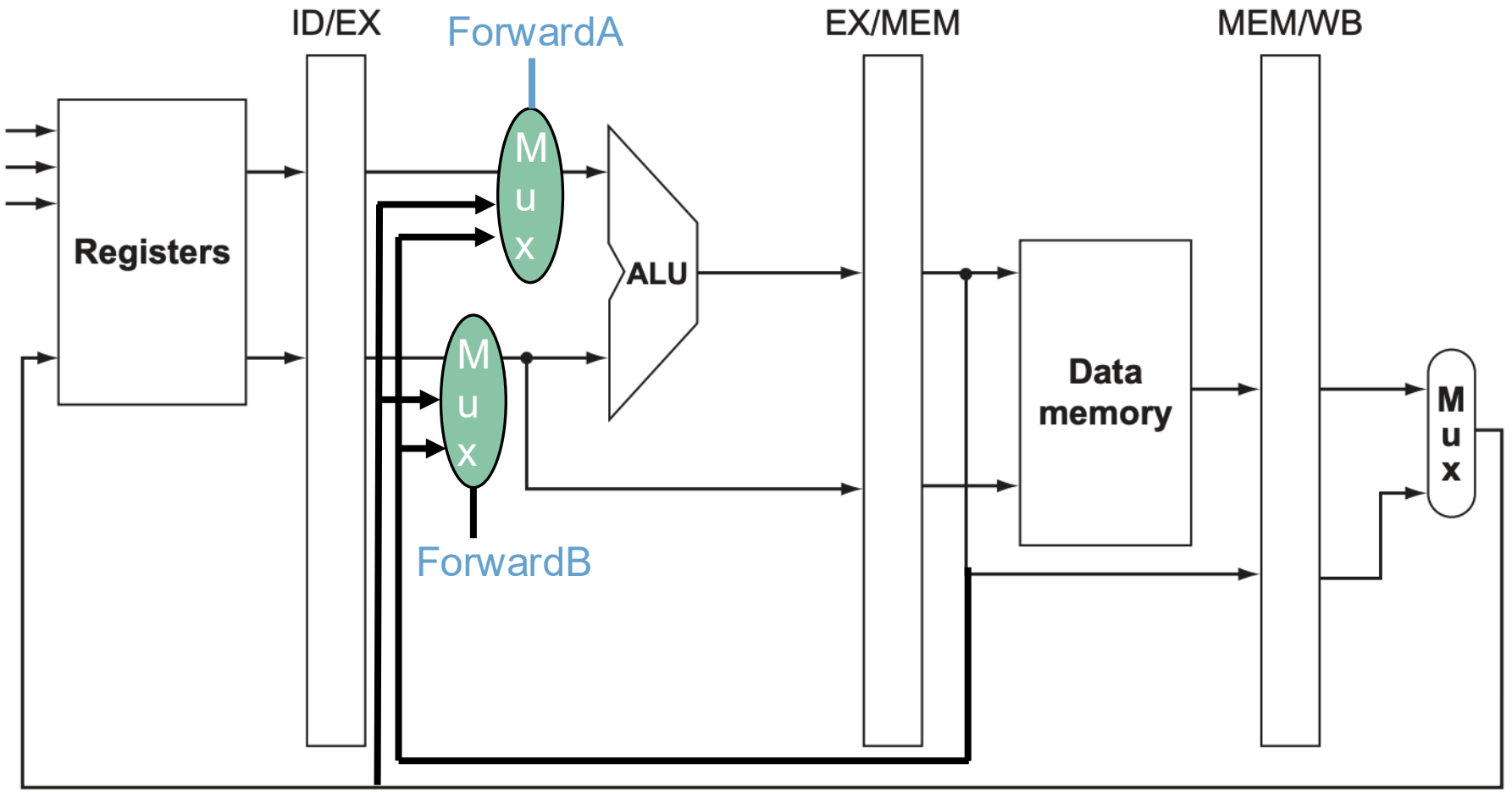
Graphical Representation of Forwarding



Source: https://csweb.wooster.edu/jmusgrave/cs210/Chapter_04.pdf

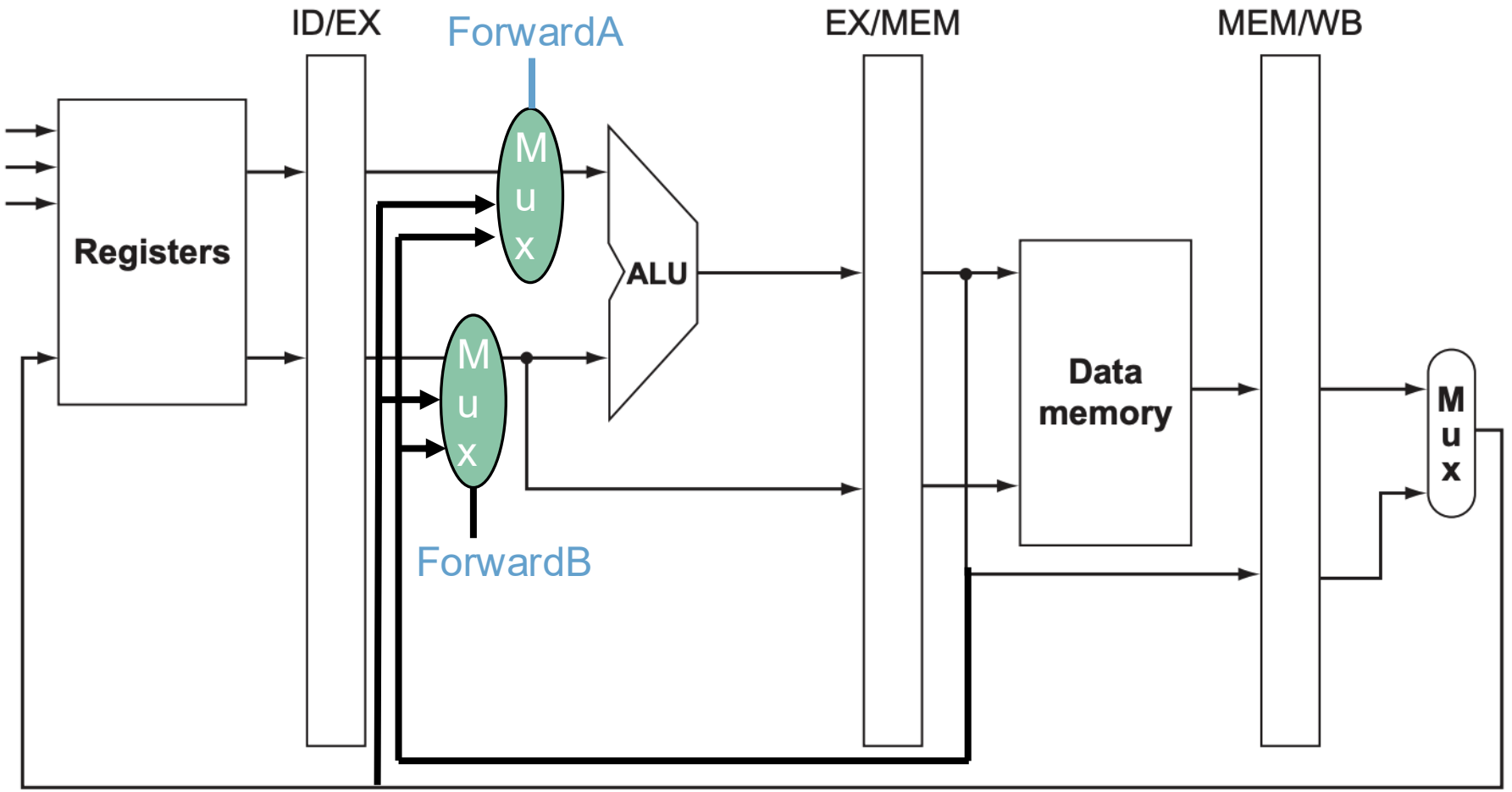
LOAD-USE HAZARD

Load-Use Data Hazard



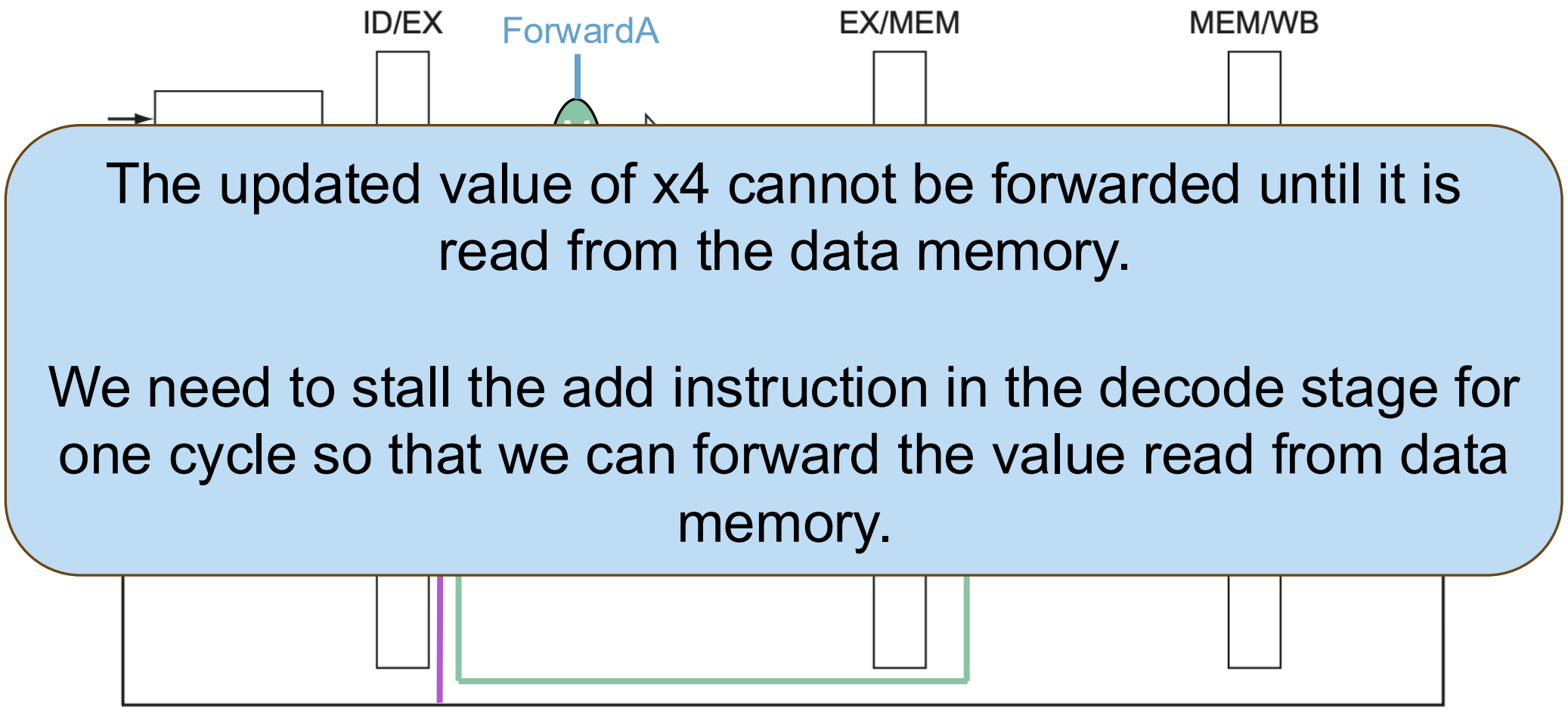
add x5, x4, x3 lw x4, 0(x9)

Load-Use Data Hazard



add x5, x4, x3 lw x4, 0(x9)

Load-Use Data Hazard

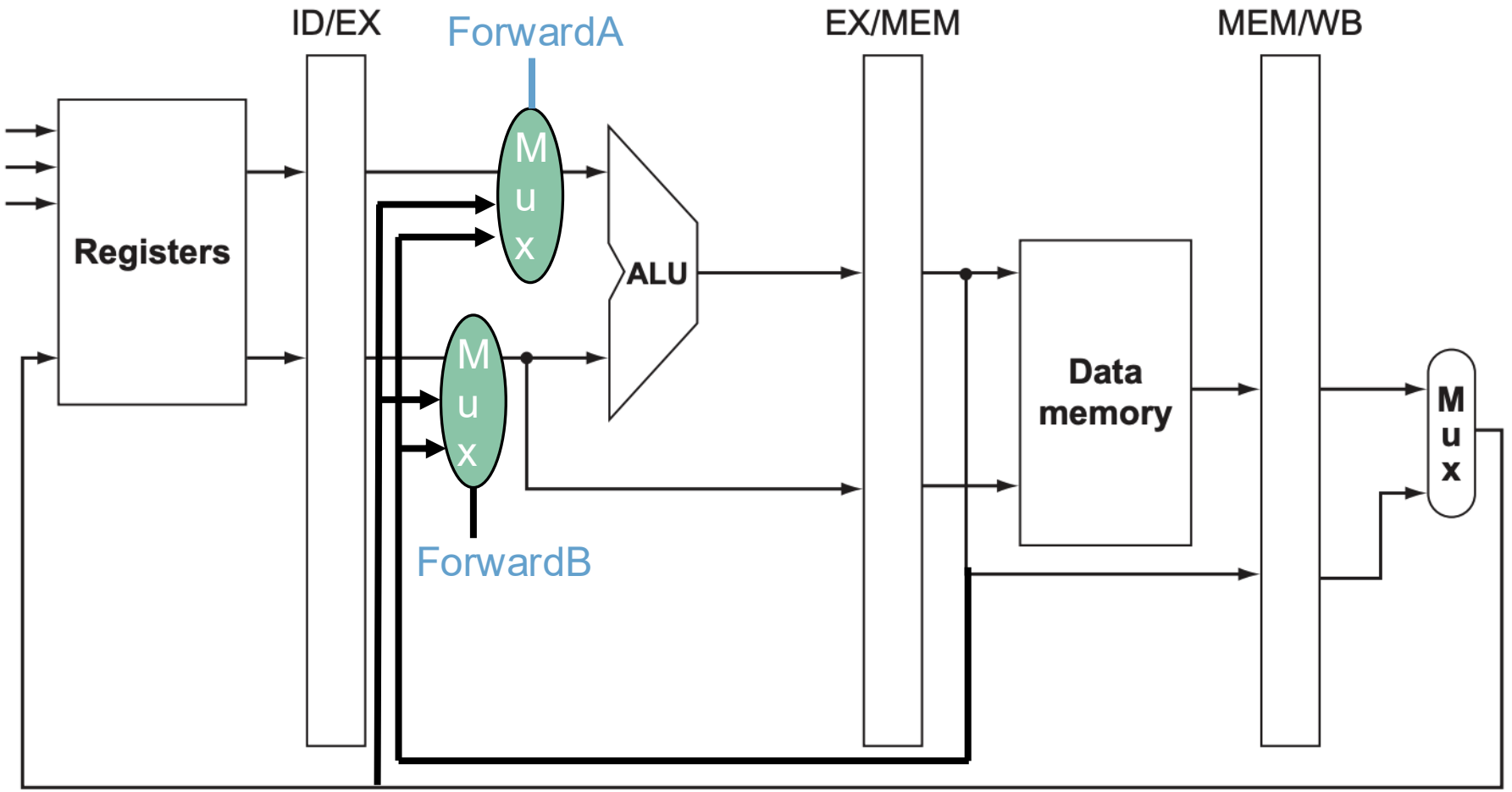


The updated value of x4 cannot be forwarded until it is read from the data memory.

We need to stall the add instruction in the decode stage for one cycle so that we can forward the value read from data memory.

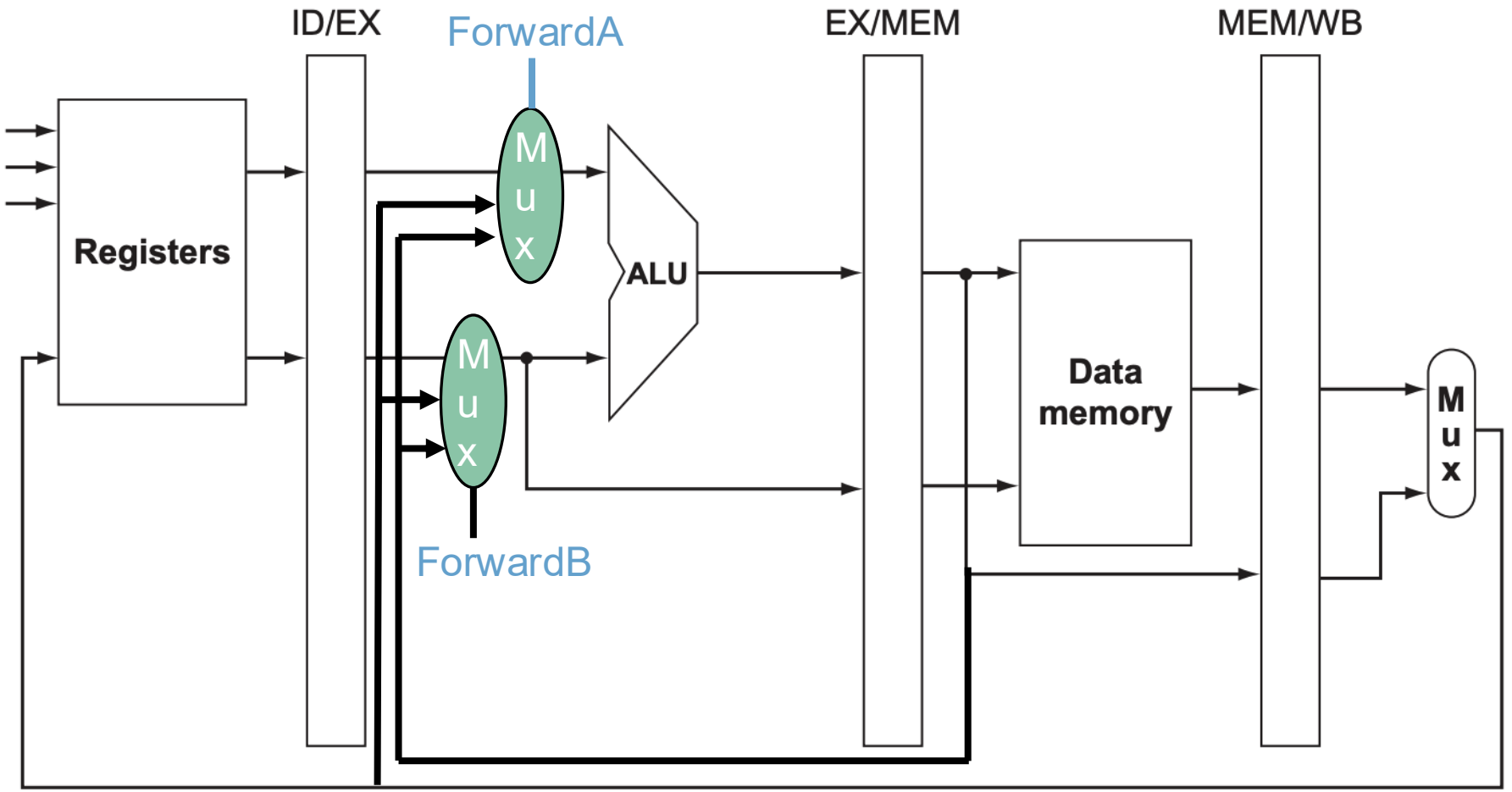
add x5, x4, x3 lw x4, 0(x9)

Load-Use Data Hazard



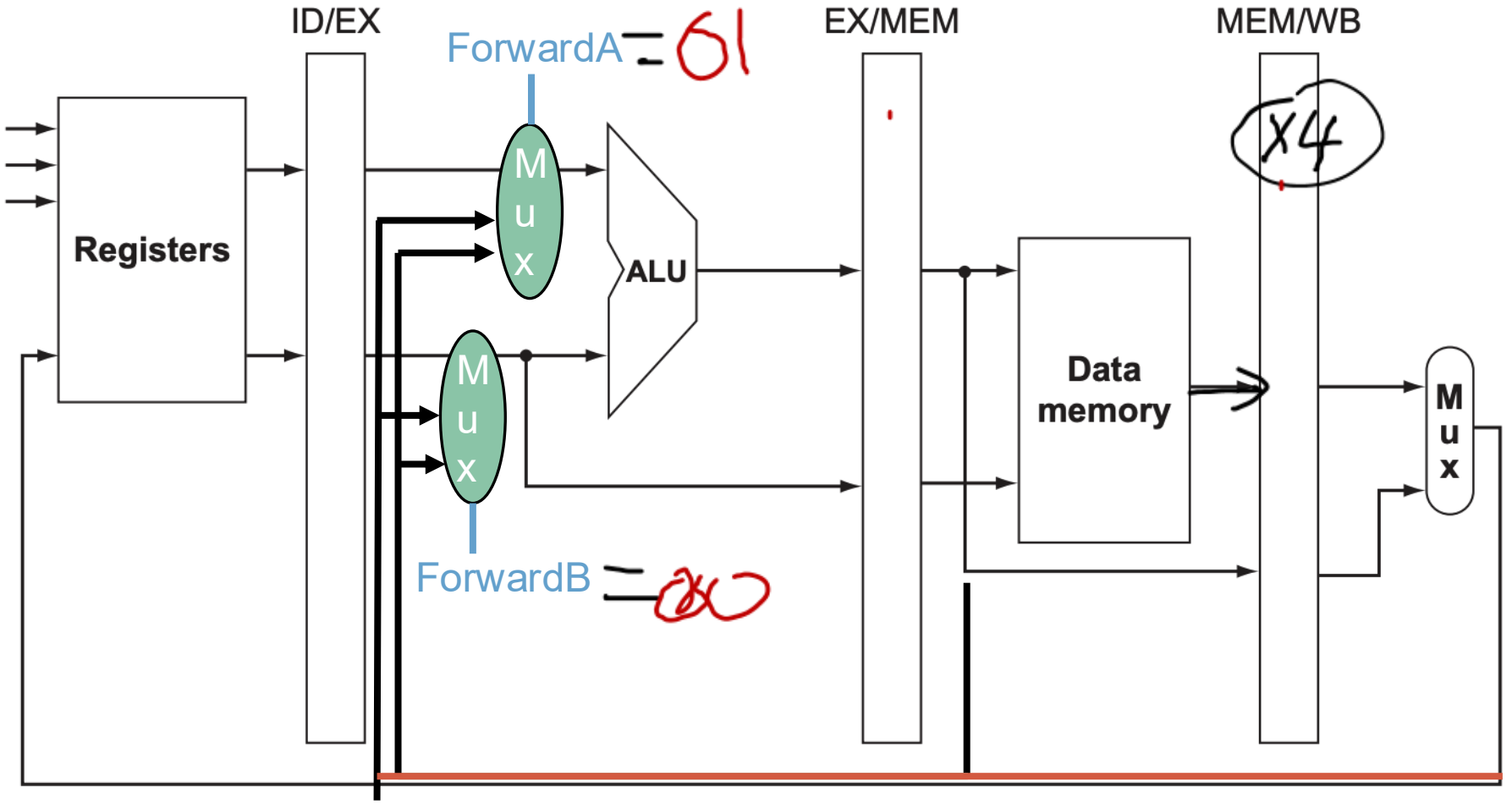
add x5, x4, x3 lw x4 , 0(x9)

Load-Use Data Hazard



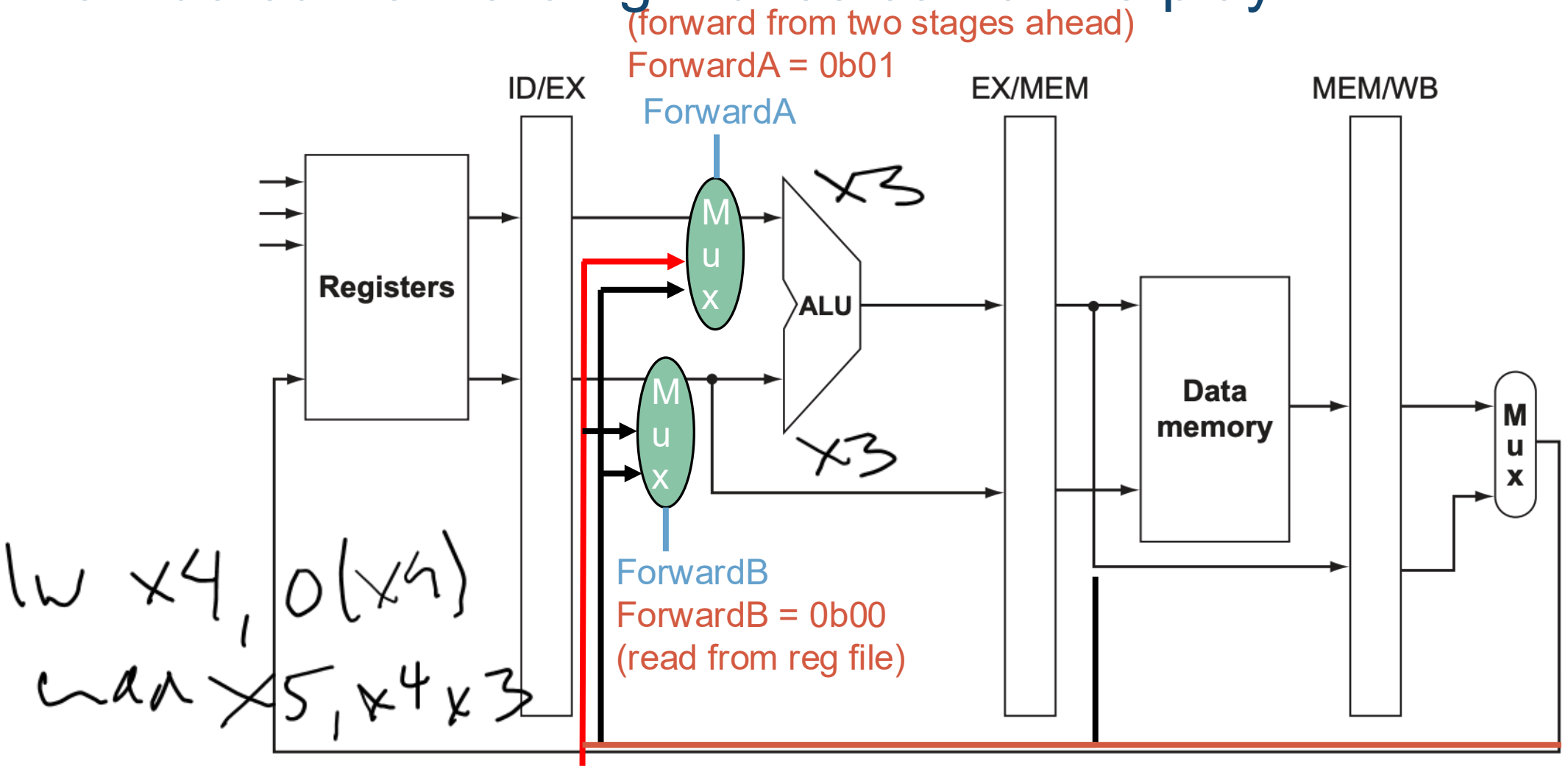
add x5, x4, x3 *stall* lw x4 , 0(x9)

How do our forwarding muxes come into play?



add x5, x4, x3 *stall* lw x4, 0(x9)

How do our forwarding muxes come into play?



add x5, x4, x3 *stall* lw x4, 0(x9)

Pipeline Diagram

00 - reg file
10 - 1 stage
01 - 2 stages

Hint: the only stalls should be because of Load-Use hazards!

1. Fill out the pipeline diagram for the following set of instructions.
2. What should ForwardA and ForwardB be set to on each cycle?
   - If the instruction in the execute stage is a stall, set ForwardA and ForwardB to XX.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lw x3, 0(x8)	F	D	X	M	W										
add x11, x3, x3		F	D	X	M	W									
lw x4, 0(x11)			F	D	X	M	W								
add x9, x11, x9				F	D	X	M	W							
sub x10, x4, x9					F	D	X	M	W						

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ForwardA	XX	XX	00	XX	01	00	10	01	XX	XX					
ForwardB	XX	XX	00	XX	01	00	00	10	XX	XX					

Forwarding Explanation
Cycle 0-1:
No instruction in EX.
(ForwardA, ForwardB) = (XX, XX)

Cycle 2: lw x3, 0(x8) in EX
Load computes address only.
Operands come from register file.
(00, 00)

Cycle 3: Stall (load-use hazard)
No real instruction in EX.
(XX, XX)

Cycle 4: add x11, x3, x3 in EX
Needs x3 from lw.
Load result is in MEM/WB at start of cycle.
Forward both operands from MEM/WB.
(01, 01)

Cycle 5: lw x4, 0(x11) in EX
Load computes address only.
Operands come from register file.
(00, 00)

Cycle 6: add x9, x11, x9 in EX
Needs x11 from previous add.
At start of cycle, value is in EX/MEM.
Forward from EX/MEM for operand A.
Operand B from register file.
(10, 00)

Cycle 7: sub x10, x4, x9 in EX
Two dependencies:
x4 from lw → value in MEM/WB → forward (01)
x9 from add → value in EX/MEM → forward (10)
(01, 10)

Forwarding Codes:
00 = register file
10 = EX/MEM (1 stage ahead)
01 = MEM/WB (2 stages ahead)
XX = don’t care (no instruction in EX)
Key Rule:
Forward based on where the value is at the start of the cycle, not the end.

Another Explanation of Previous Slide

- Cycle 0-1: Nothing to be executed yet
  - *(XX, XX)*
- Cycle 2: Instruction_1: LW in the execution stage → A load in EX only computes an address and does not need forwarded ALU operands. Both ALU inputs come from the register file through the ID/EX register.
  - *(00, 00)*
- Cycle 3: Stall bc of Load-Use hazard!
  - *(XX, XX)*
- Cycle 4: Instruction_2: Add in the execution stage. The add instruction needs x3, which was loaded by Instruction 1. Load results are available at the end of the MEM stage. At this moment, Instruction 1 is in MEM and its loaded value will be placed in MEM/WB this cycle. Instruction 2 needs the value now in EX, so it must forward from the MEM/WB pipeline register. This is considered two stages ahead.
  - *(01, 01)*
- Cycle 5: Instruction_3: LW in execution stage → use ALU operands, so both inputs should come from the regfile.
  - *(00, 00)*
- Cycle 6: Instruction_4: Add in execution stage → This instruction needs x11, which was produced by Instruction 2. At the start of this cycle, Instruction 2's ALU result is sitting in the EX/MEM pipeline register, because it completed EX one cycle earlier. Forwarding from EX/MEM corresponds to "one stage ahead." The other operand (x9) comes from the register file.
  - *(10, 00)*
- Cycle 7: instruction_5: Sub in execution stage → This instruction has two dependencies.
  x4 was produced by Instruction 3 (a load). Load values become available at the end of MEM. At the start of cycle 7, Instruction 3 is in WB and its load result is sitting in MEM/WB. That requires forwarding from two stages ahead.
  x9 was produced by Instruction 4 (an ALU instruction). At the start of this cycle, Instruction 4 is in MEM, so its ALU result is in EX/MEM. That requires forwarding from one stage ahead.
  - *(01, 10)*

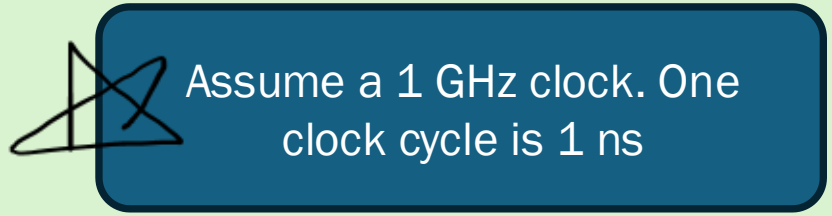
Pipeline Diagram

1. Fill out the pipeline diagram for the following set of instructions.
2. What should ForwardA and ForwardB be set to on each cycle?
   - If the instruction in the execute stage is a stall, set ForwardA and ForwardB to XX.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lw x3, 0(x8)	F	D	X	M	W										
add x11, x3, x3		F	D	D	X	M	W								
lw x4, 0(x11)			F	F	D	X	M	W							
add x9, x11, x9					F	D	X	M	W						
sub x10, x4, x9						F	D	X	M	W					

ForwardA	XX	XX	00	XX	01	00	10	01	XX	XX					
ForwardB	XX	XX	00	XX	01	00	00	10	XX	XX					

Performance Analysis + Pipelining



1×10^{-9}

- Assume a pipelined implementation where:
  - *Taken branches take 2 cycles.*
  - *Loads and stores take 2 cycles.*
- Assuming approximately 10% of instructions executed are branches, and of those 80% of the time they are taken, and 15% of instructions executed are loads or stores, what sort of speed-up do we expect?

Performance Analysis + Pipelining

- Assume a pipelined implementation where:
  - *Taken branches take 2 cycles.*
  - *Loads and stores take 2 cycles.*
- Assuming approximately 10% of instructions executed are branches, and of those 80% of the time they are taken, and 15% of instructions executed are loads or stores, what sort of speed-up do we expect?

Ideal impl: 100 inst x 1 cycle = 100 cycles
→ 100 ns

Performance Analysis + Pipelining

goal: CPI → 1

Ideal impl: 100 inst x 1 cycle = 100 cycles
→ 100 ns

Pipelined

Branches → .10(100) → 10 inst
80% taken → 8 taken (8 x 2) + (2 x)
20% nx → 2 nx = 18 cycles

loads / stores
.15(100) → 15 inst x 2 → 30 cycles

other: 75% left → 75 cycles

Total: 18 + 30 + 75 = 123 ns

100 / 123 ≈ 0.81

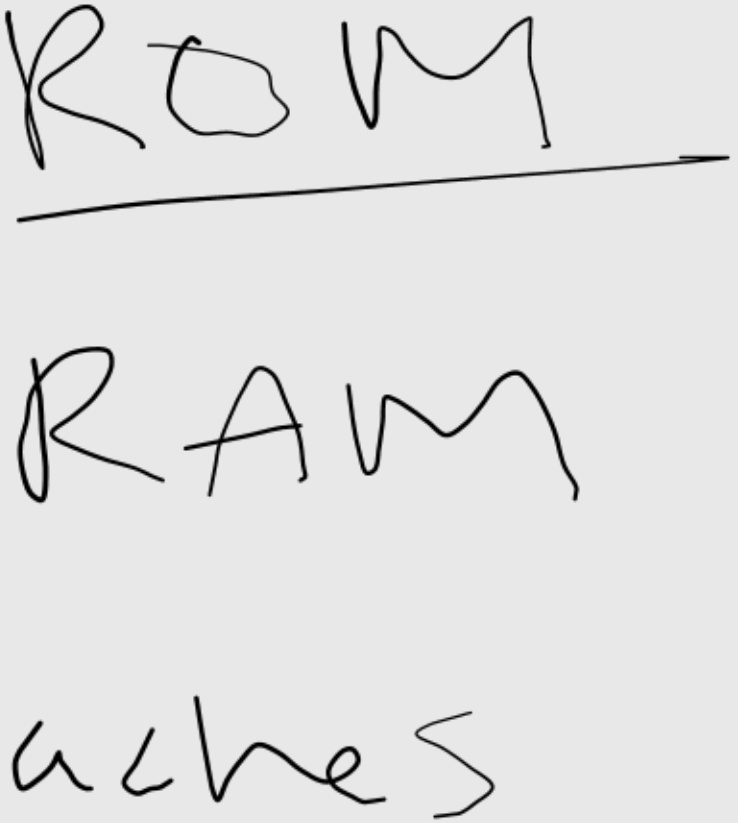
Some takeaways...

- Design strategy: Focus our attention on placing pipelining registers around the slowest circuit elements (bottlenecks!)
- You can pipeline an RISC-V CPU even more. There exist implementations with 7, 8, and 9 pipeline stages. But the overhead of bypass paths and the number of stall cases increase .
- Memory access time is our real bottleneck!

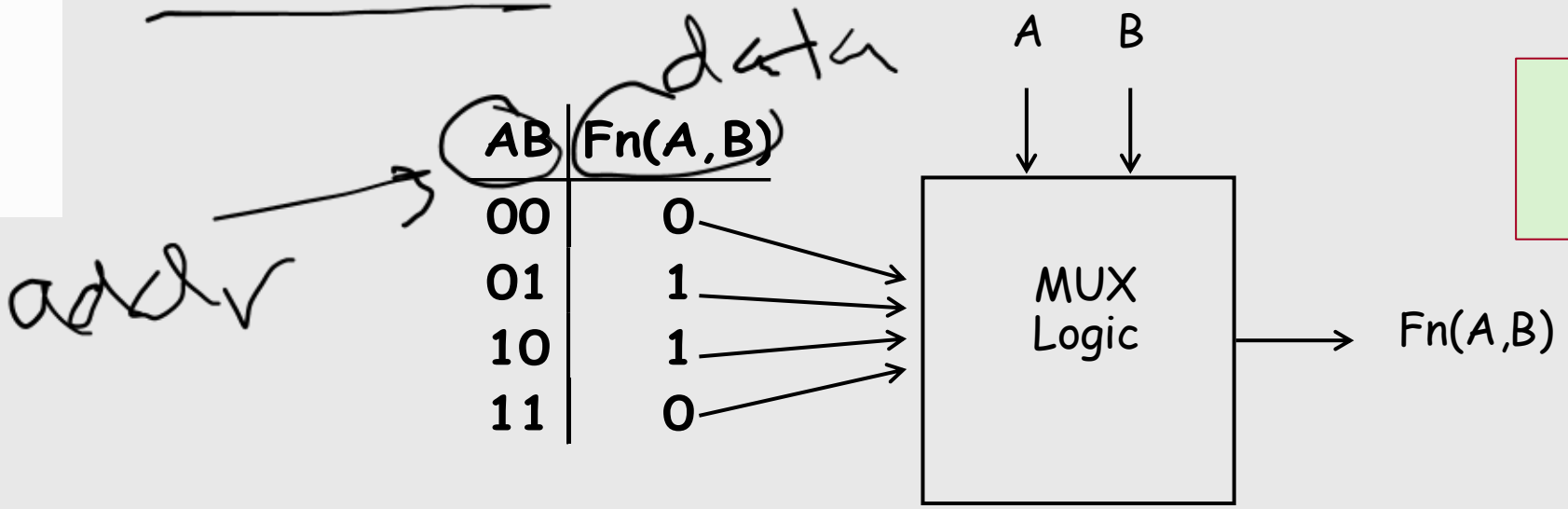
MEMORY: AN OVERVIEW

Topics

- Memory Arrays
- Transparent Latches
- Edge-Triggered Registers
- Finite State Machines



Read-Only Memory = a Lookup Table

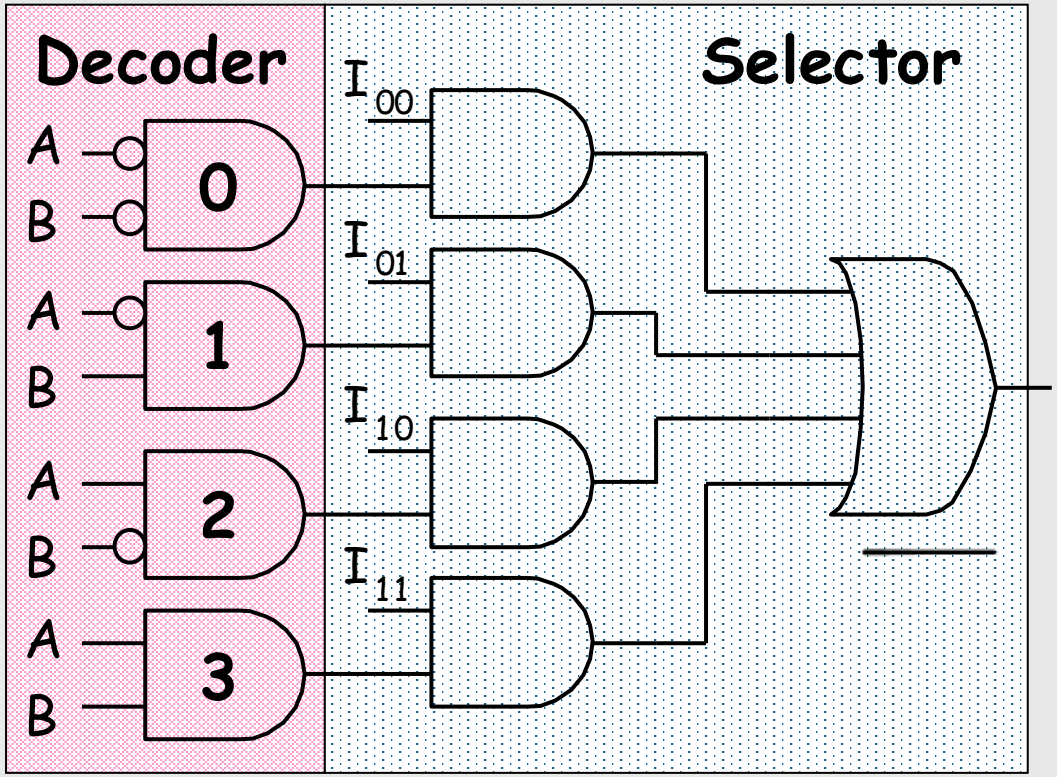


Read-Only Memory (ROM)

- A multiplexer can implement a lookup table:
  - *for an N-input function we need a 2^N input multiplexer*
  - *simply select one row out of 2^N*
  - *2^N values go into the mux, only one comes out*
  - *N-bit select input is now the "address" of data being accessed*
- Limitations:
  - *mux can get really big: 10-bit address → 1024 rows!*
  - *read-only; we also would like to write!*

Takeaway: ROM → "hardcoded truth table"
MUX → Hardware that selects correct output based on the input

A Mux's Guts: Decoder + Selector



A decoder generates all possible product terms for a set of inputs

Multiplexers can be partitioned into two sections.

A DECODER that identifies the desired input,and

a SELECTOR that enables that input onto the output.

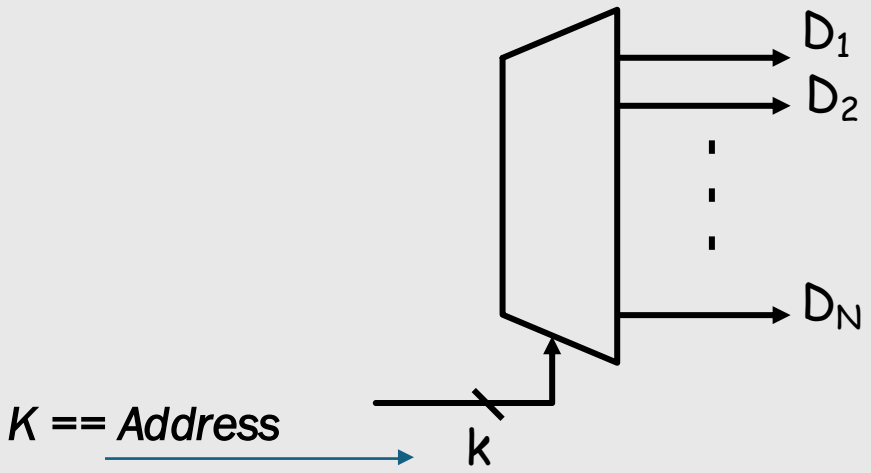
addressing

Key idea behind building ROMs: by **sharing the decoder** part of the logic MUXs could be adapted to make lookup tables with **any number of outputs**

Decoder

4-to-1 mux

1-to-4 mux



DECODER:

k SELECT inputs,

N = 2^k DATA OUTPUTs.

Selected D_j HIGH; all others LOW.

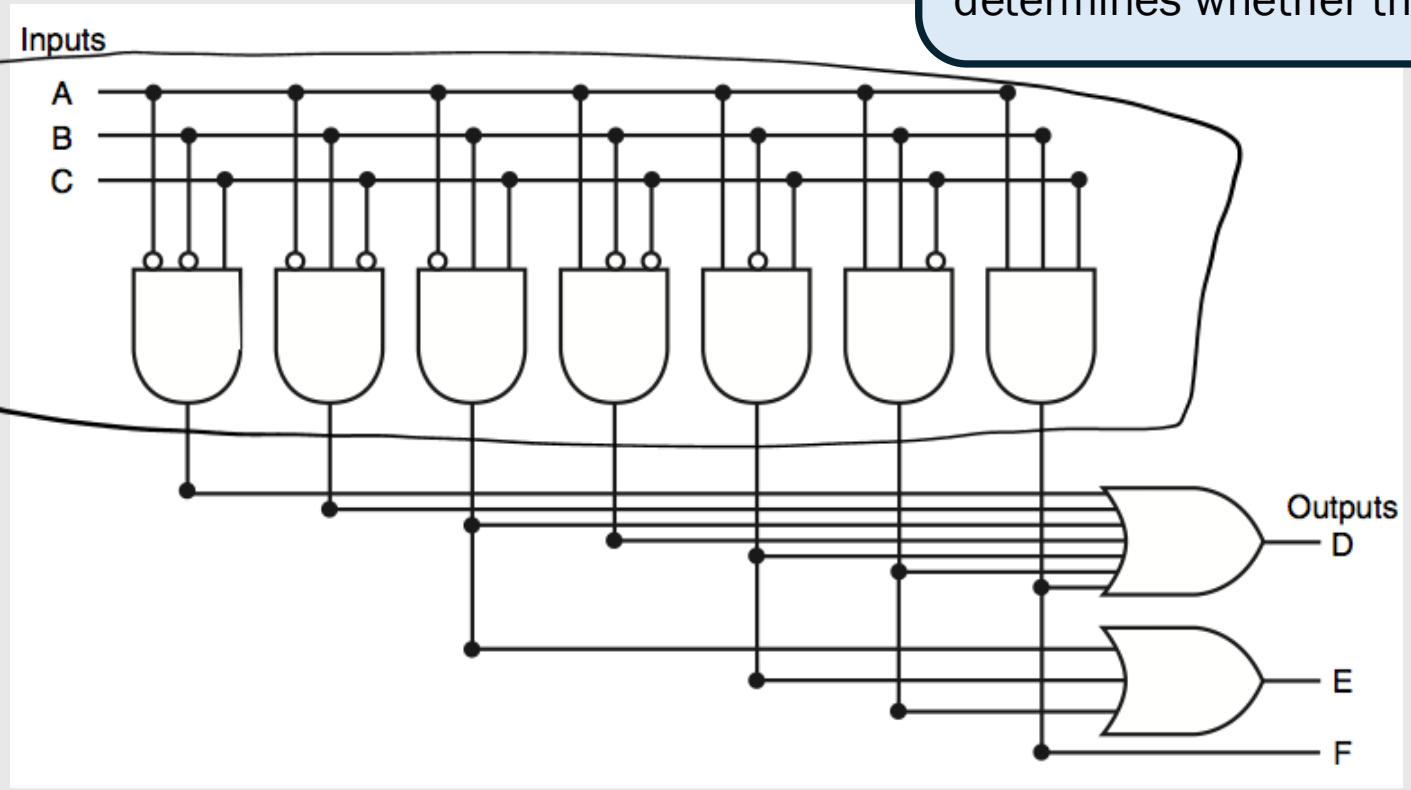
One-hot signal to identify row we are working with

NOW, we are well on our way to building a general purpose table-lookup device.

We can build a 2-dimensional ARRAY of decoders and selectors as follows by arranging them in a grid…

Shared Decoding Logic

This structure is exactly what a ROM does: the decoder selects a row (the address), and each output corresponds to a column that determines whether that row produces a 1 or 0.



We can build a general purpose “table-lookup” device called a Read-Only Memory (ROM), from which we can implement any truth table and, thus, any combinational device

Logic According to ROMs

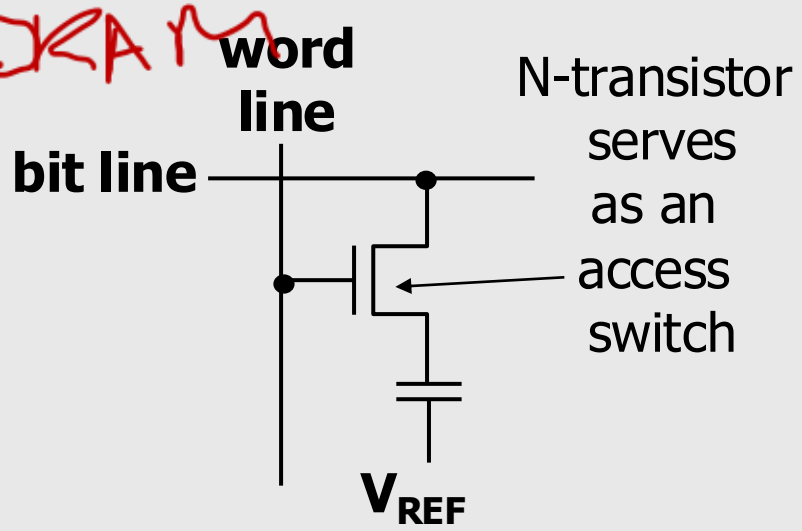
Store fixed values to be read

- Different ROM implementation technologies
  - *Mask (old) → ROM*
    - read-only memory
  - *Fuses (old) → PROM*
    - programmable read-only memory
  - *Erasable → EPROM*
    - erasable programmable read-only memory
  - *Electrically erasable → EEPROM*
    - electrically-erasable programmable read-only memory
    - → today called **FLASH**! Used everywhere!
- Model: LOOK UP value of function in truth table…
  - *Inputs: "ADDRESS" of a truth table entry*
  - *ROM SIZE = # truth table entries…*
    - … for an N-input boolean function, size = $2^N \times \#\text{outputs}$

2^N

Read-Write Memories *(RAM)*

- Read-Write often called **random-access memories (RAM)**
- Dynamic RAM uses analog storage:
  - *encode information using voltages*
  - *from physics: we can "store" a voltage as "charge" on a capacitor*

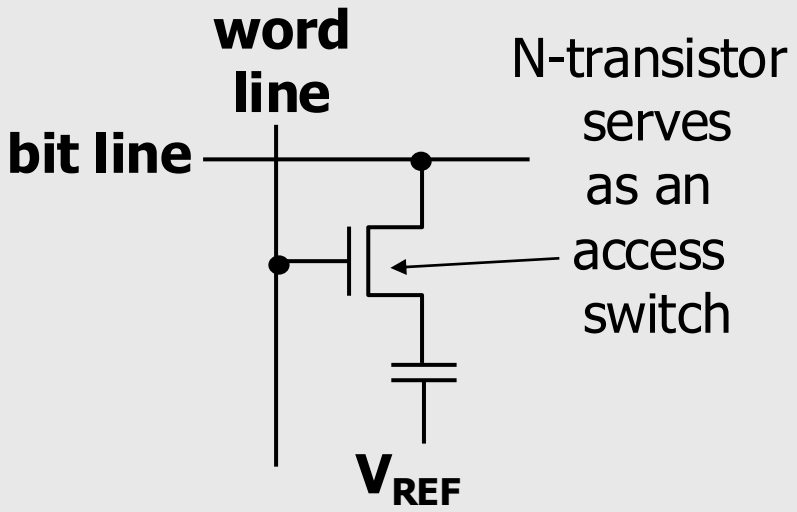


To write:
Drive bit line, turn on access transistor, force storage cap to new voltage

To read:
precharge bit line, turn on access trans., detect (small) change in bit line voltage

Read-Write Memories *(RAM)*

- Dynamic RAM uses analog storage:
  - *encode information using voltages*
  - *from physics: we can "store" a voltage as "charge" on a capacitor*
  - *Pros:*
    - compact!
  - *Cons:*
    - it leaks! -> refresh
    - complex interface
    - reading a bit, destroys it
      - *(you have to rewrite the value after each read)*
    - it's NOT a digital circuit



To write:
Drive bit line, turn on access transistor, force storage cap to new voltage

To read:
precharge bit line, turn on access trans., detect (small) change in bit line voltage

This storage circuit is the basis for commodity DRAMs