COMP311: *COMPUTER ORGANIZATION!*

Lecture 27: More Pipeline Hazards, Caching + the Memory Hierarchy

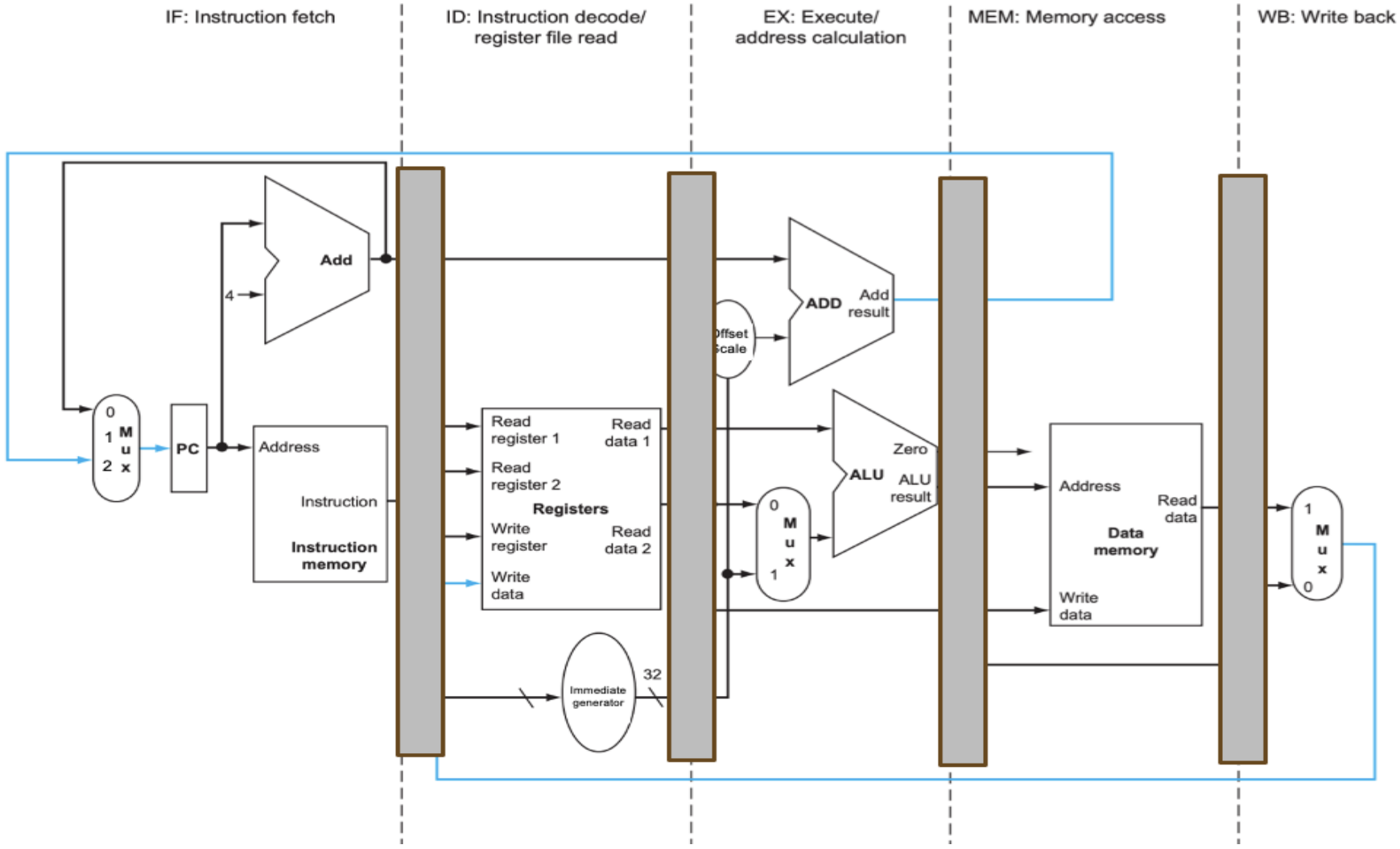
tinyurl.com/comp311-fa25

Final Exam Logistics!

- In this room! **Saturday 12/6. 12-3pm**
- **Updated assigned seats** will be posted on Canvas the day before
- Test format will be the same as the quizzes
  - *Expected length: ~2.5x quizzes*
- Exam is **cumulative**
  - *Equally distributed across all topics. Including today!*
- Final review session held by TAs 12/4.
- Final review guide + extra practice problems will be posted today
  - *Not exhaustive – just a study tool! Your awesome TAs are generous (and haven't seen the exam ☺)*
  - *Going over in-class practice & written assignments will be helpful!*
- You will get a multi-page reference sheet. I will post it tomorrow!

PIPELINE HAZARDS

Each pipeline register name is based on the stages it separates



Register Names

IF/ID

ID/EX

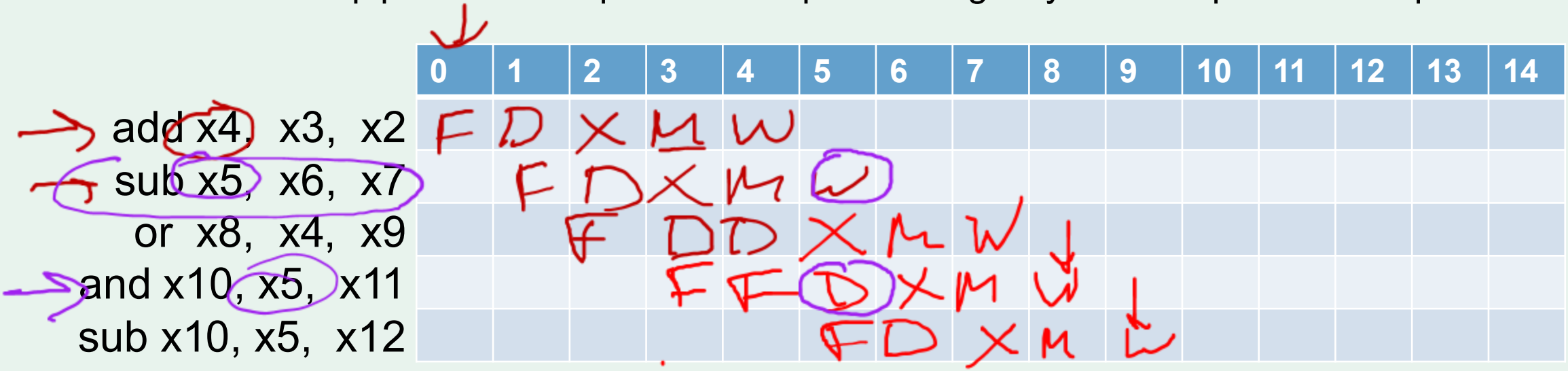
EX/MEM

MEM/WB

Pipeline Diagram with Stalls

Data hazard

1. Fill out the pipeline diagram for the following ~~set of instructions~~.
2. What is the CPI?
3. What is the speedup compared to a single-cycle datapath?
   - Assume pipelined clock period = 220ps and single-cycle clock period = 820ps



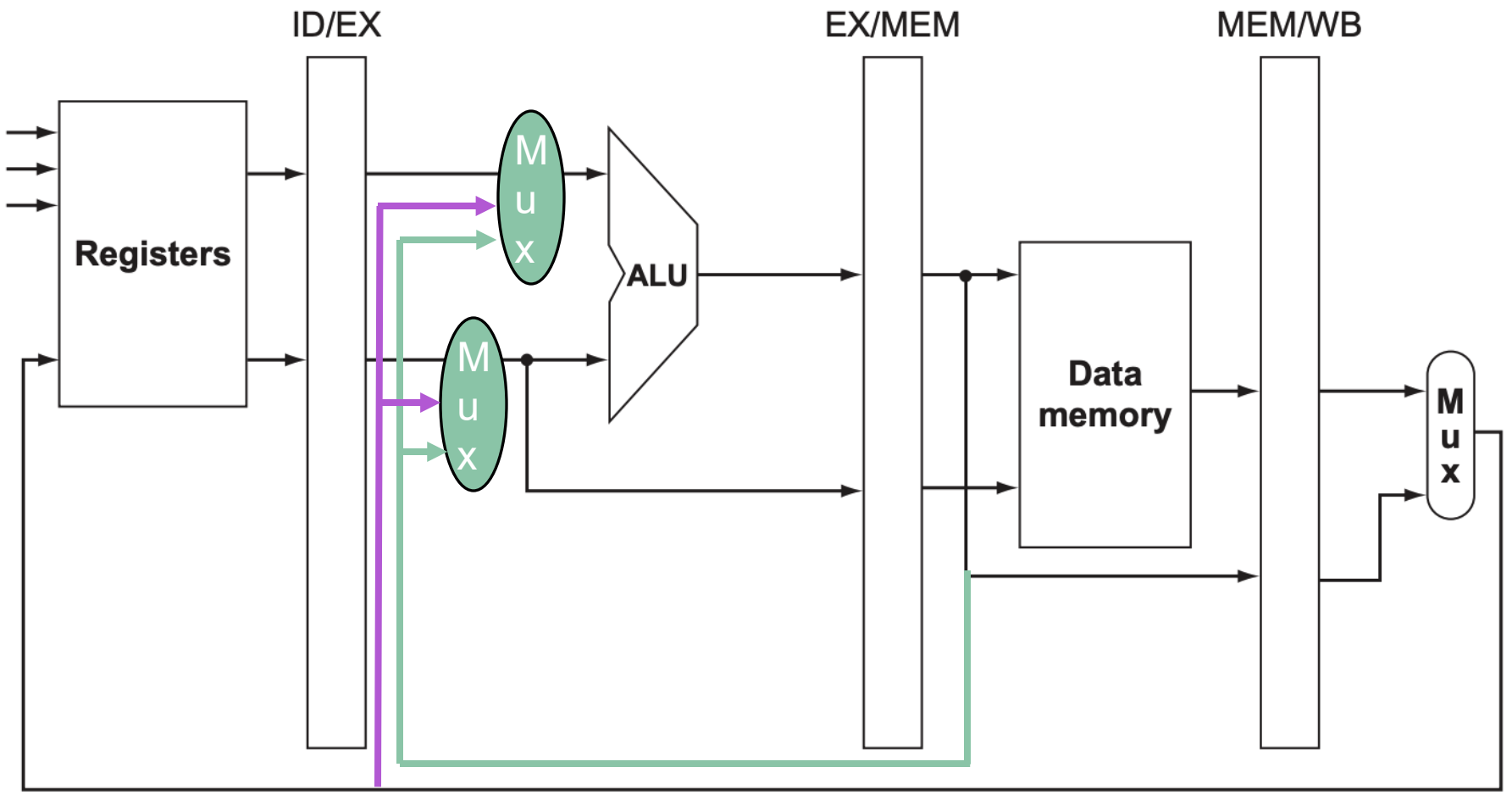
Pipeline Diagram with Stalls

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
add x4, x3, x2	F	D	X	M	W										
sub x5, x6, x7		F	D	X	M	W									
or x8, x4, x9			F	D	D	X	M	W							
and x10, x5, x11				F	F	D	X	M	W						
sub x10, x5, x12						F	D	X	M	W					

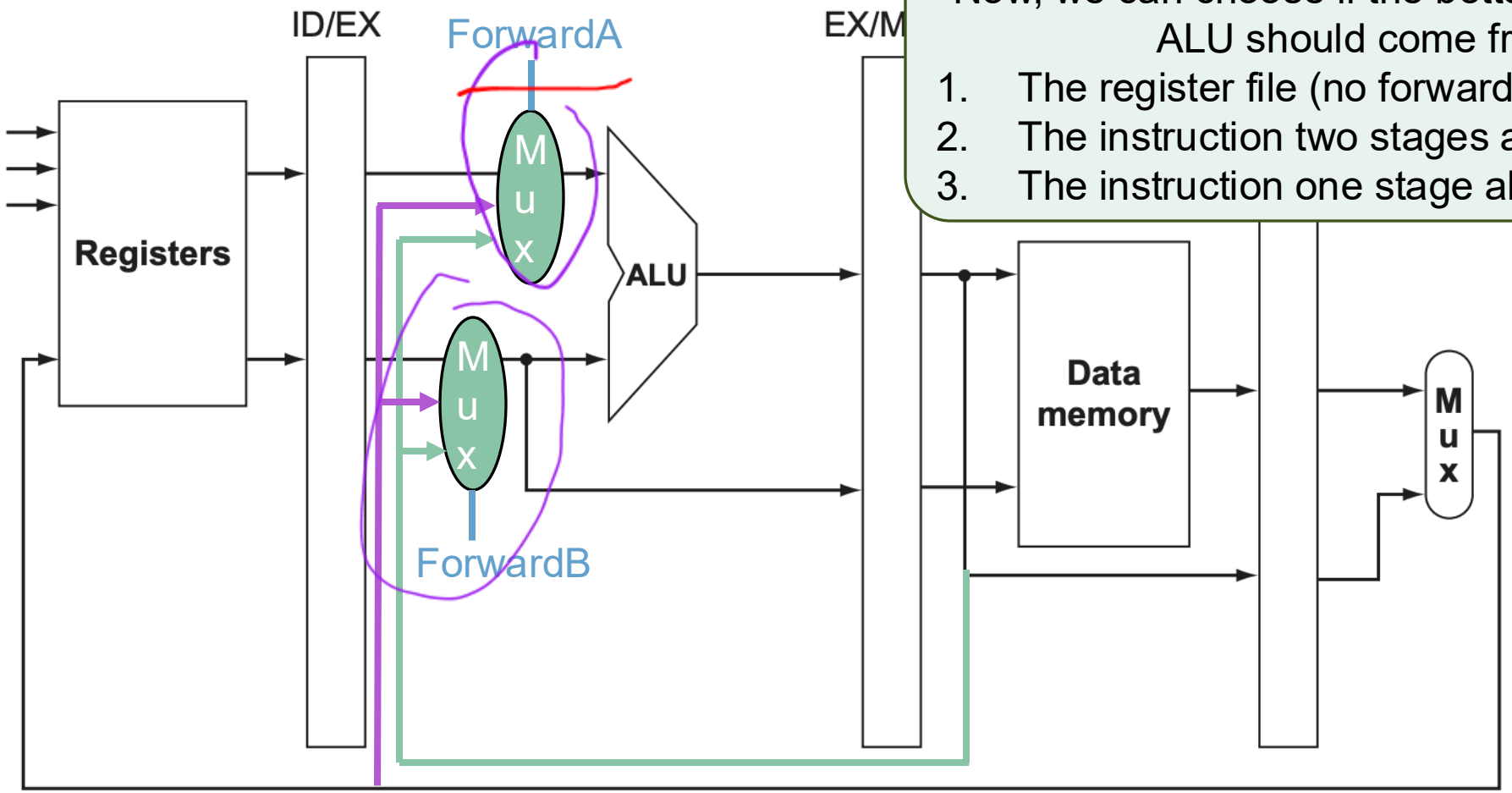
$$CPI = \frac{10}{5} = 2$$

$$Speedup = \frac{5 \cdot 820ps}{10 \cdot 220ps} = \frac{4100ps}{2200ps} = 1.86$$

Recall: How to update the Datapath to Allow for Forwarding in R-Type Instructions



Recall: How to update the Datapath to Allow for Forwarding in R-Type Instructions



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

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

New control signals!

Forwarding to Handle Data Hazards

Top ALU input

ForwardA	Source
0b00	Register file
0b10	ALU result from one stage ahead
0b01	ALU result from two stages ahead

Bottom ALU input

ForwardB	Source
0b00	Register file
0b10	ALU result from one stage ahead
0b01	ALU result from two stages ahead

Forwarding

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 there is no instruction in the execute stage, set ForwardA and ForwardB to XX

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
add x2, x3, x4	F	D	X	M	W										
or x5, x2, x7		F	D	X	M	W									
sub x8, x9, x10			F	D	X	M	W								
add x3, x5, x8				F	D	X	M	W							
ForwardA															
ForwardB															

Forwarding

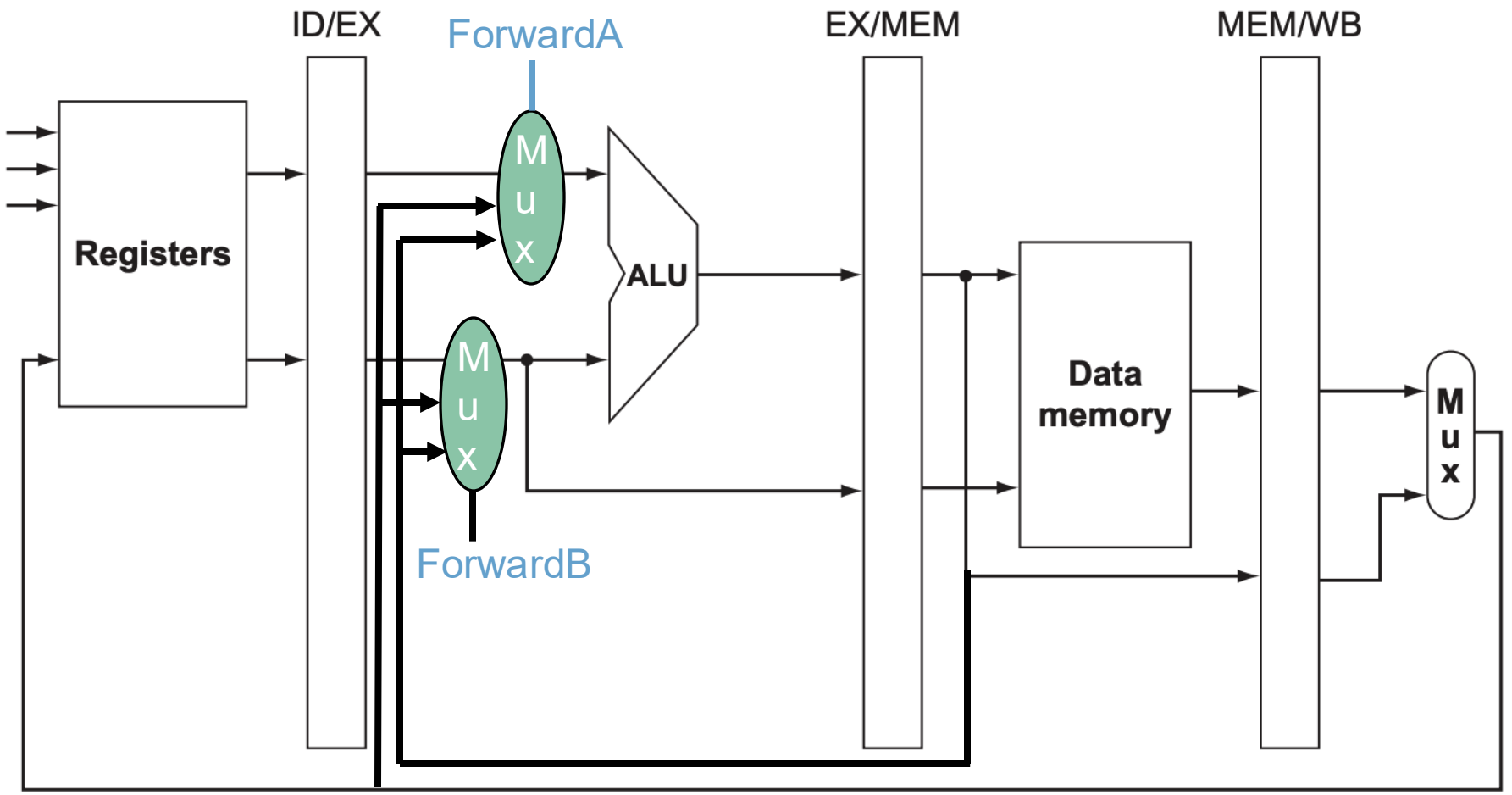
1. Fill out the pipeline diagram for the following set of instructions.
2. What should ForwardA and ForwardB be set to on each cycle?

- If there is no instruction in the execute stage, set ForwardA and ForwardB to XX

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
add x2, x3, x4	F	D	X	M	W										
or x5, x2, x7		F	D	X	M	W									
sub x8, x9, x10			F	D	X	M	W								
add x3, x5, x8				F	D	X	M	W							
ForwardA	XX	XX	00	10	00	01	XX	XX							
ForwardB	XX	XX	00	00	00	10	XX	XX							

LOAD-USE DATA HAZARD

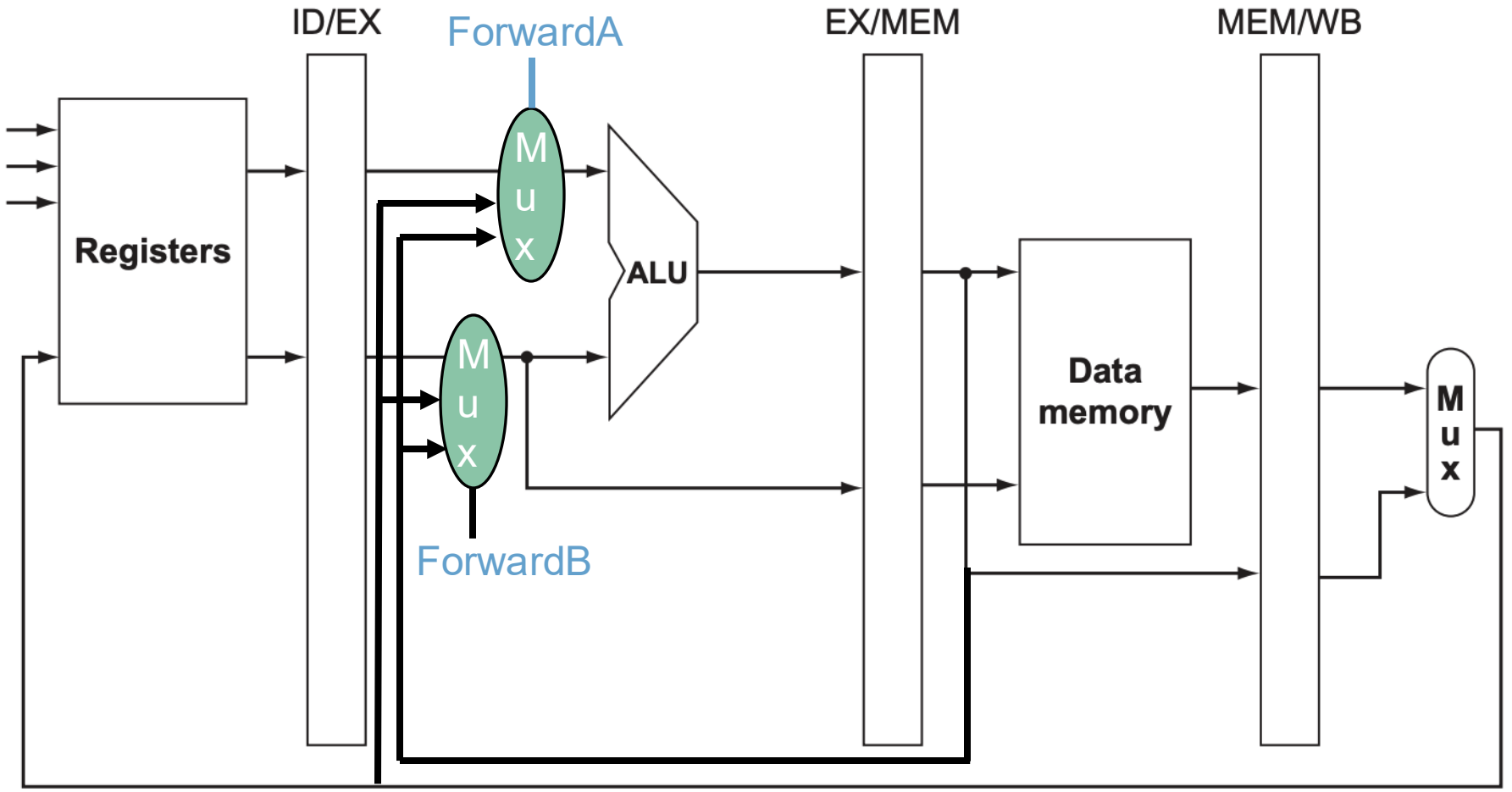
Load-Use Data Hazard



add x5, x4, x3

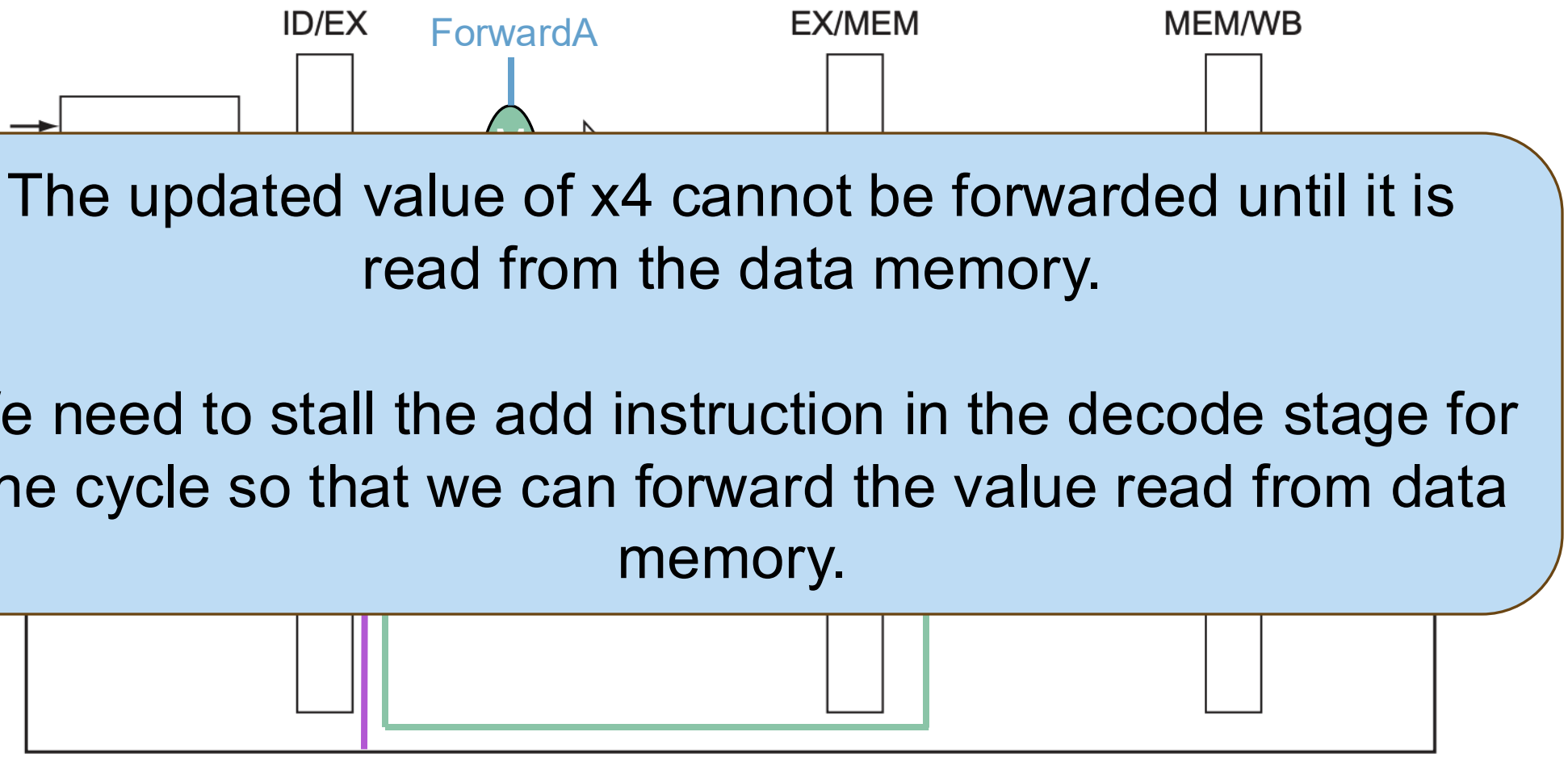
lw x4, 0(x9)

Load-Use Data Hazard



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

Load-Use Data Hazard

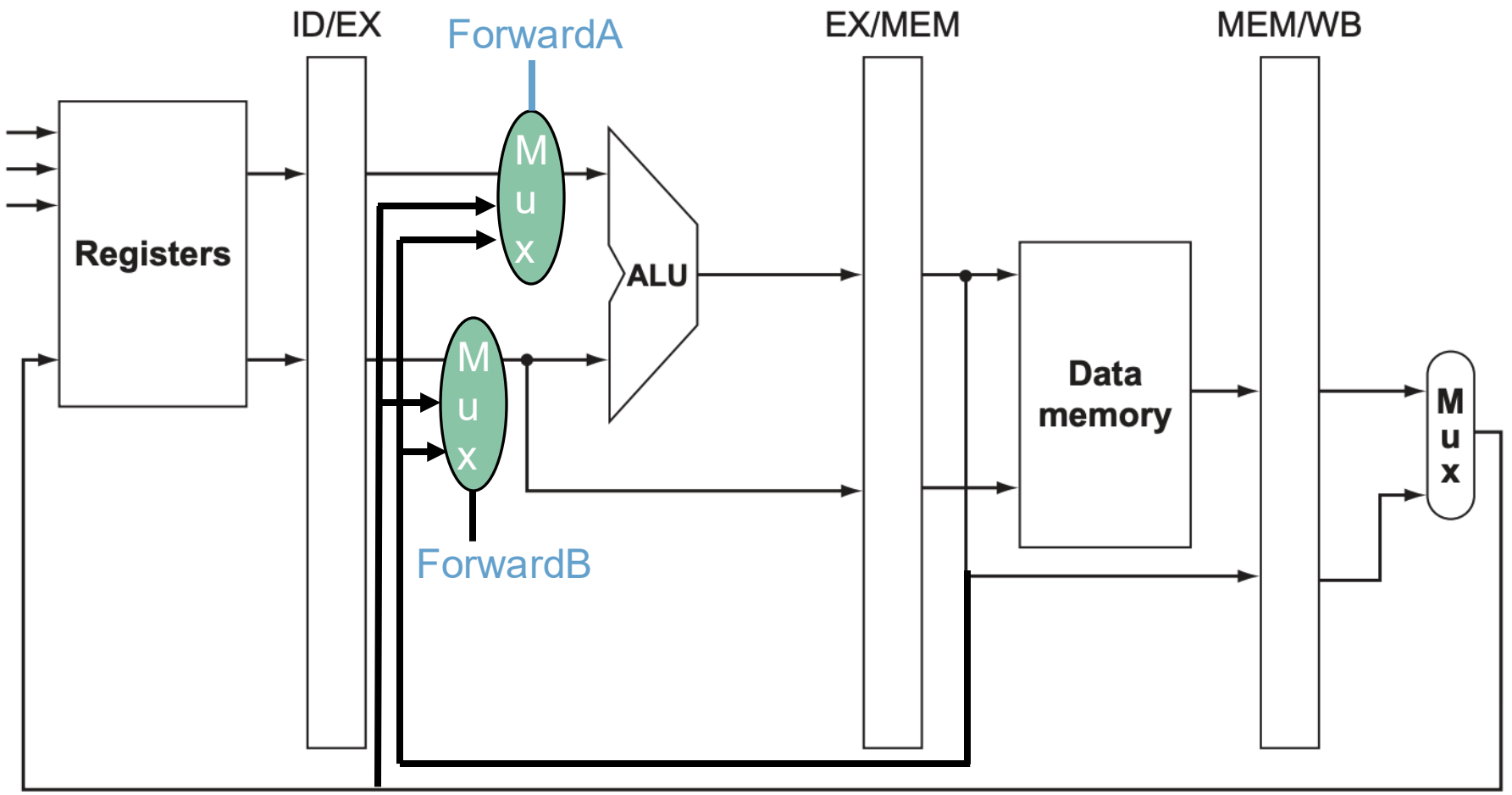


The updated value of x4 cannot be forwarded until it is read from the data memory.

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.

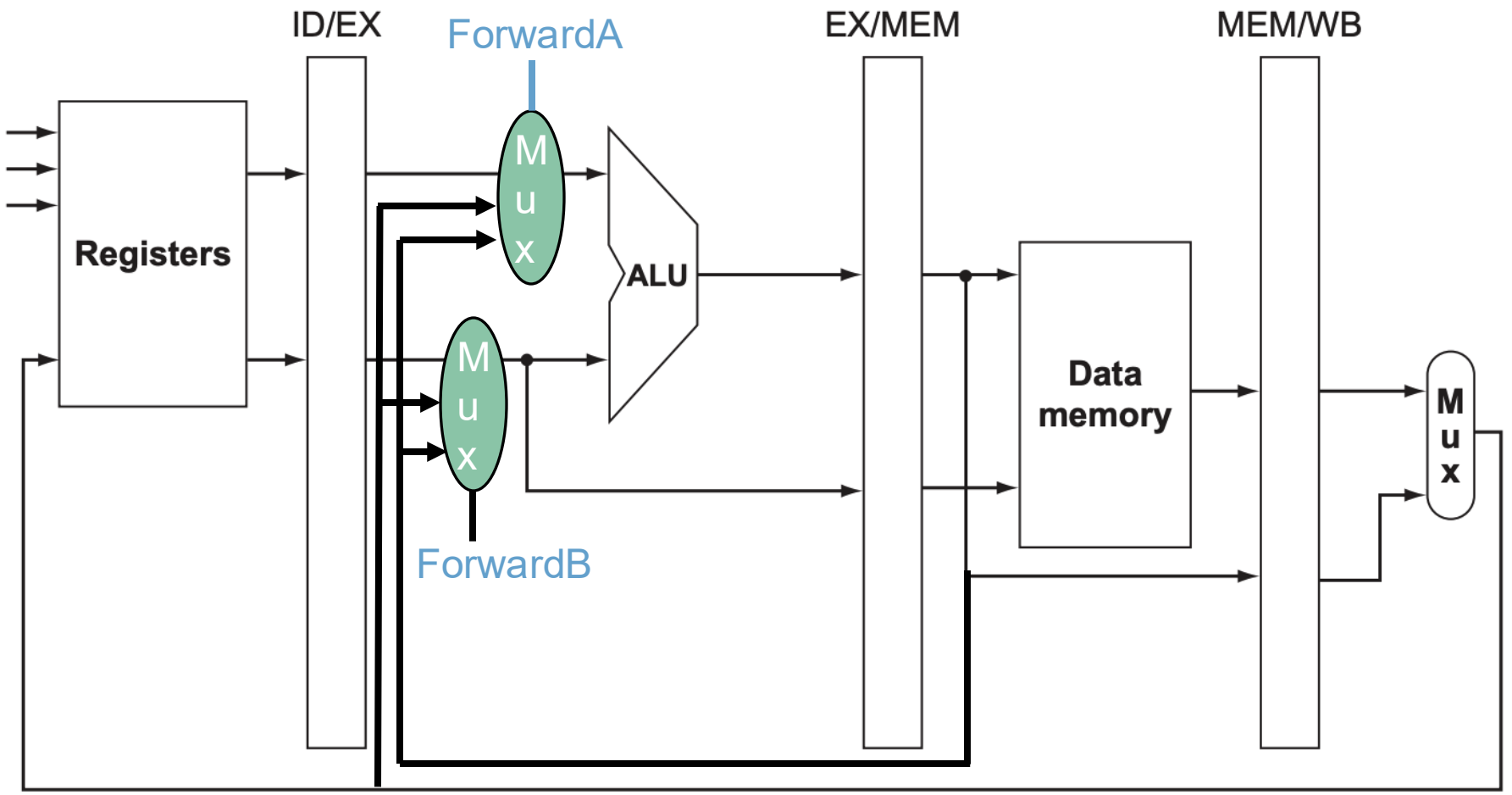
add x5, x4, x3 lw x4, 0(x9)

Load-Use Data Hazard



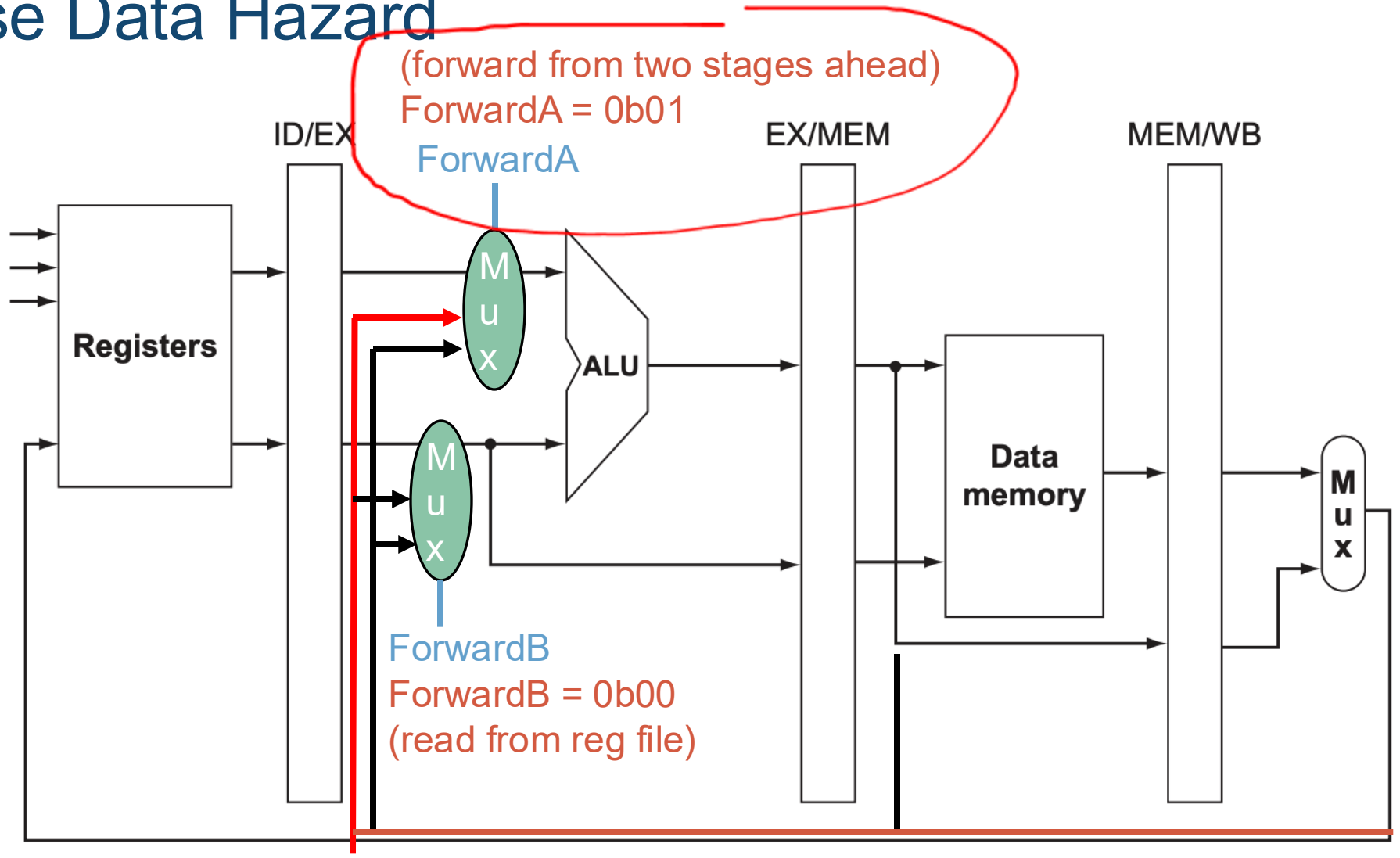
add x5, x4, x3 lw x4 , 0(x9)

Load-Use Data Hazard



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

Load-Use Data Hazard



add x5, x4, x3

stall

lw x4, 0(x9)

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)															
add x11, x3, x3															
lw x4, 0(x11)															
add x9, x11, x9															
sub x10, x4, x9															
ForwardA															
ForwardB															

Pipeline Diagram

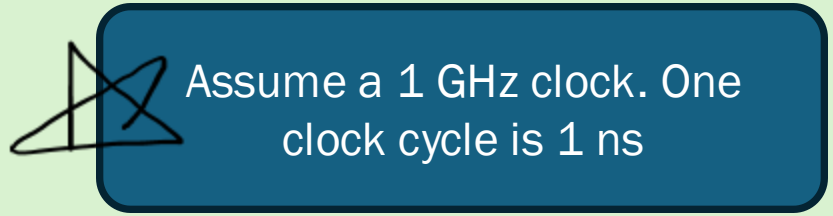
1. Fill out the pipeline diagram for the following set of instructions.
2. What should ForwardA and ForwardB be set to on each cycle?
   - If the instruction in the execute stage is a stall, set ForwardA and ForwardB to XX.

	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					

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)*

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 real 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 real speed-up do we expect?

$$Perf_{before} = (100) \times 1 = 100 \text{ Clocks} \times 2 \times 10^{-9} \text{ sec/clock} = 200 \times 10^{-9} \text{ secs}$$

$$Perf_{after} = (10)((0.8) \times 2 + (0.2) \times 1) + 15 \times 2 + 75 \times 1 = 123 \text{ Clocks}$$

$$123 \times 2/3 \times 10^{-9} \text{ sec/clock} = 82 * 10^{-9} \text{ secs}$$

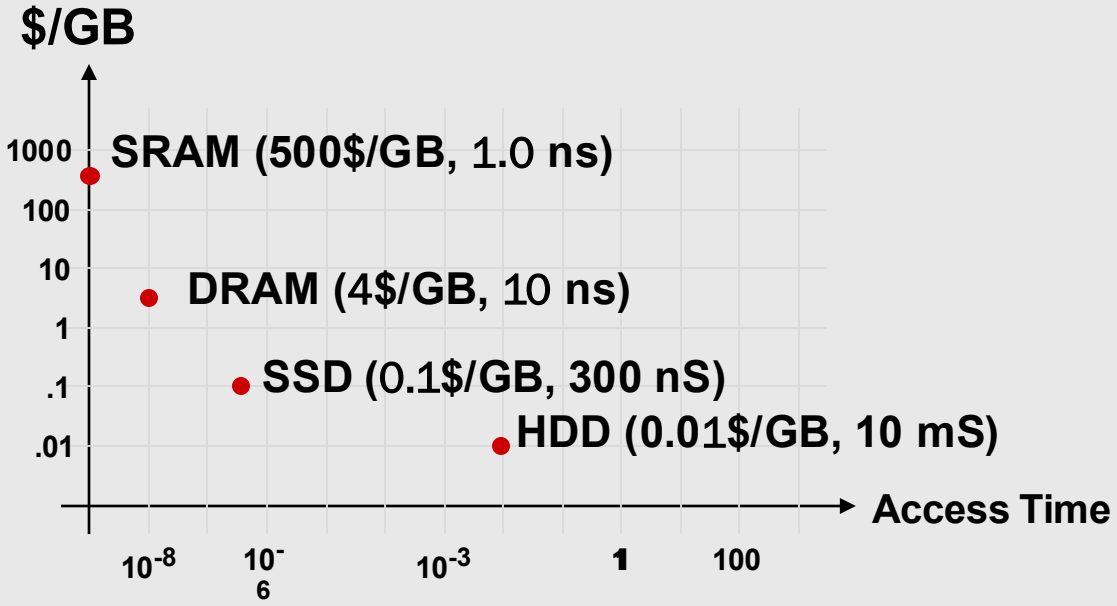
$$\text{Speedup} = \frac{Perf_{before}}{Perf_{after}} = 200/82 = 2.439 \text{ times faster}$$

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 HIERACHY & CACHING

All Memories aren't created equal



Quantity vs Speed…

Memory systems can be either:

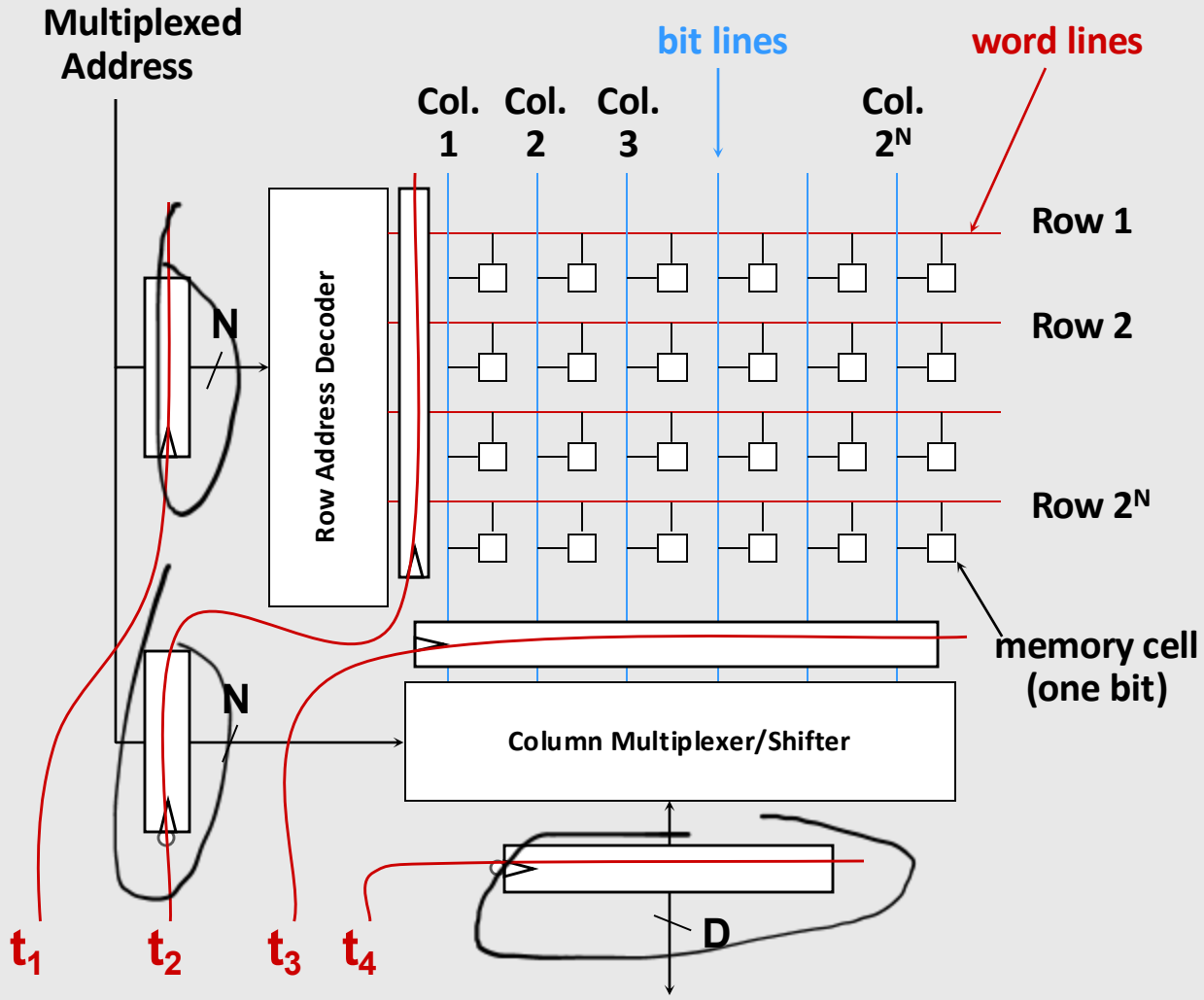
- BIG and SLOW…

or

- SMALL and FAST

Is there an ARCHITECTURAL solution to this DILEMMA?

Tricks for Increasing Throughput: Pipelining



The first thing that should pop into your mind when asked to speed up a digital design…

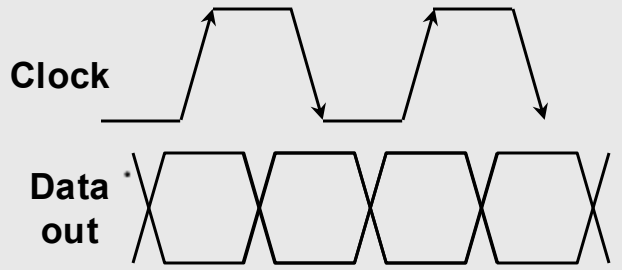
PIPELINING

Synchronous DRAM (SDRAM)

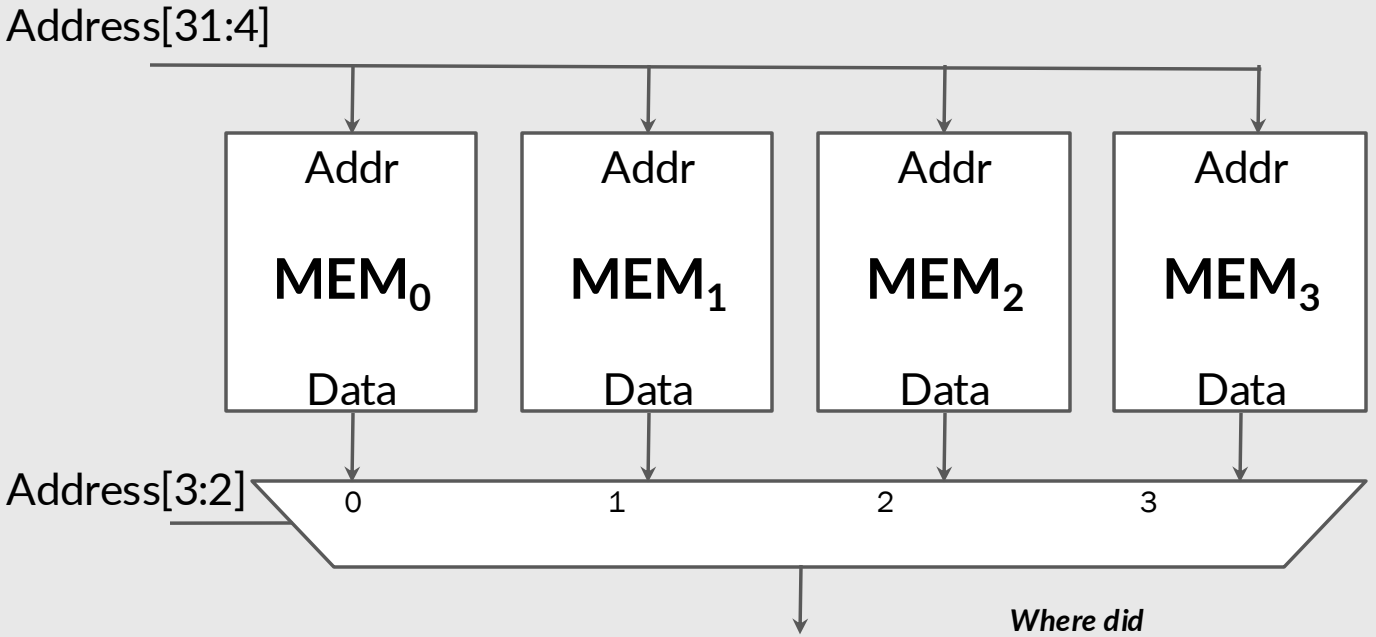
20nS reads and writes

($2.50 per Gbyte)

Double Data Rate Synchronous DRAM (DDR)



Another Trick: Interleaving



If only the lower order addresses change, we need only wait the T_{pd} of the mux.

Where did Address[1:0] go?

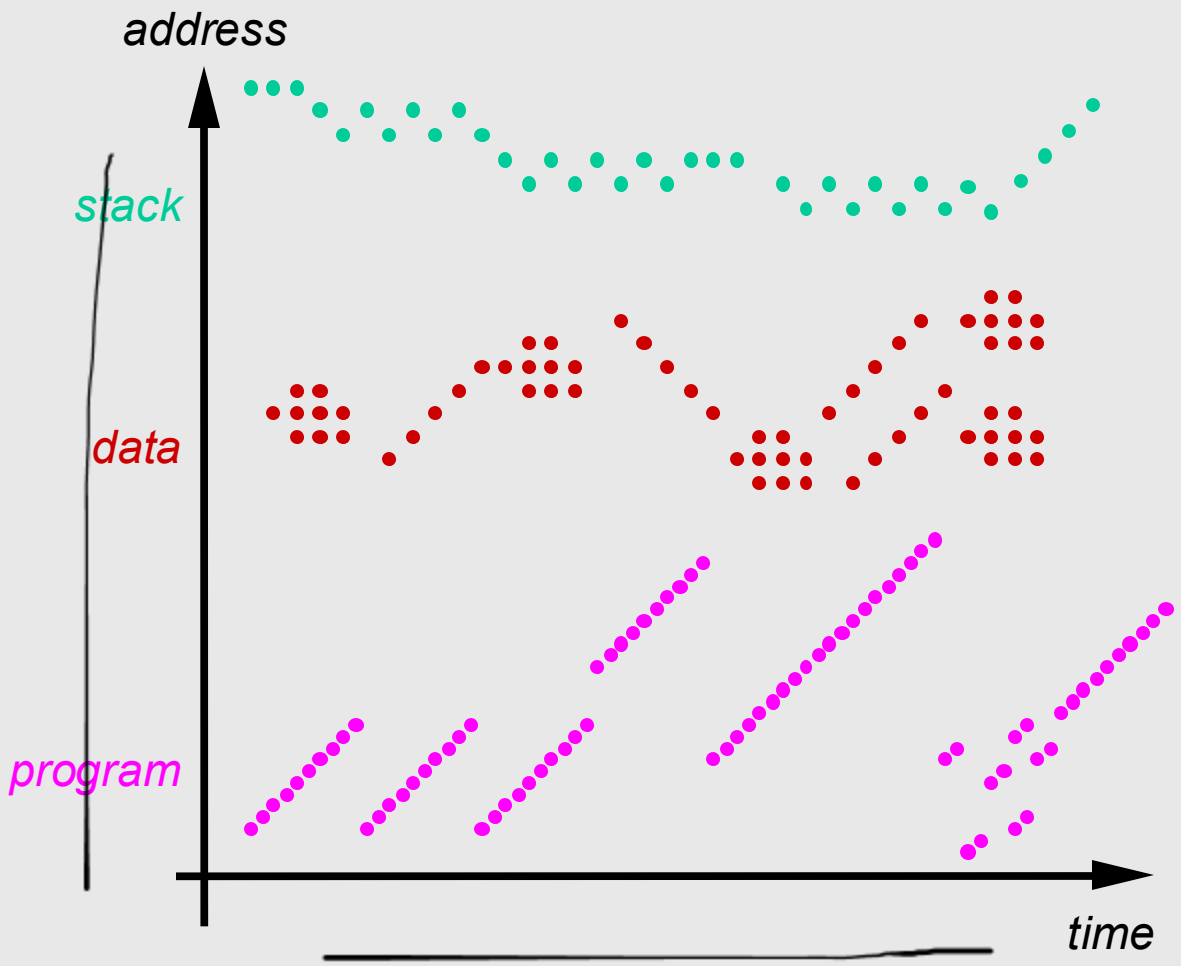
The second thing that should try when asked to speed up a digital design…

Interleaving

Accessing 4 memories at the same time has 4x the throughput. Also, every 4th word is in a different memory.

A limitation of both pipelining and interleaving is their assumption that addresses are sequential!

Typical Memory Reference Patterns



MEMORY TRACE –
A temporal sequence of memory references (addresses) from a real program.

TWO KEY OBSERVATIONS:

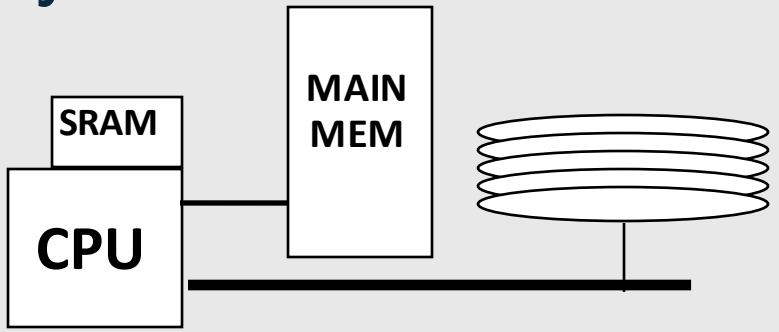
TEMPORAL LOCALITY –
If an item is referenced, it will tend to be referenced again soon

SPATIAL LOCALITY –
If an item is referenced, nearby items will tend to be referenced soon.

Exploiting the Memory Hierarchy

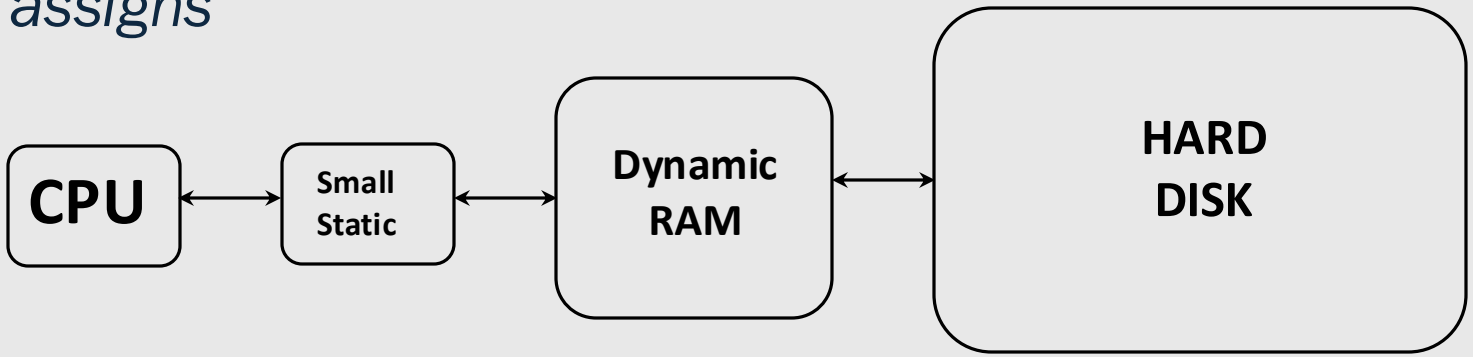
Approach 1 (Cray, DSPs, others): Expose Hierarchy

- *Registers, Main Memory, Disk each available as storage alternatives;*
- *Tell programmers: "Use them wisely"*



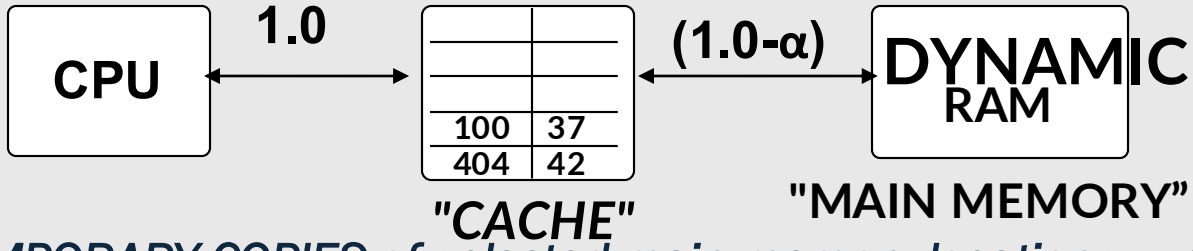
Approach 2: Hide Hierarchy

- *Programming model: SINGLE kind of memory, single address space.*
- *Machine AUTOMATICALLY assigns locations to fast or slow memory, depending on usage patterns.*



The Cache Concept

Program-Transparent Memory Hierarchy:



Cache contains TEMPORARY COPIES of selected main-memory locations... eg. Mem[100] = 37

GOALS:

1. Improve the average access time

α *HIT RATIO: Fraction of refs found in CACHE.*

(1-α) *MISS RATIO: Remaining references.*

2. Transparency (compatibility, programming ease)

$$t_{ave} = \alpha\, t_c + (1-\alpha)(t_c + t_m) = t_c + (1-\alpha)t_m$$

Challenge: Make the hit ratio, α, as high as possible.

Why, on a miss, do I incur the access penalty for both main memory and cache?

How High of a Hit Ratio?

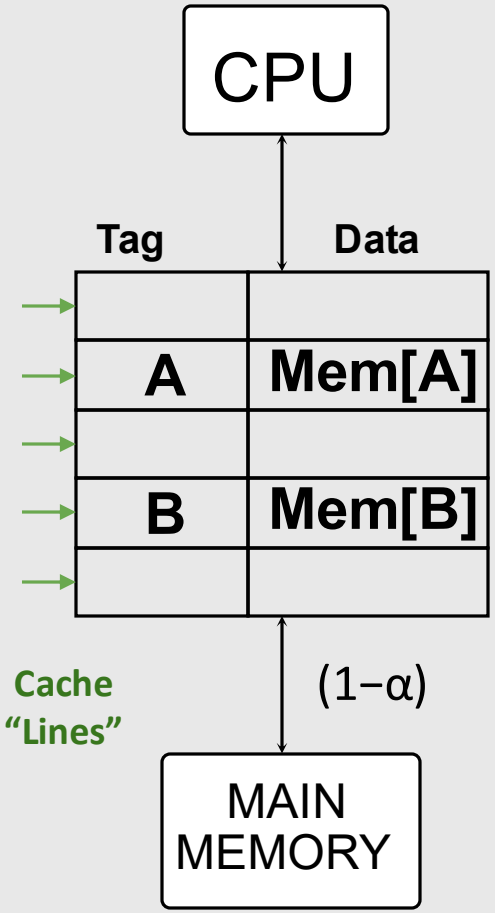
Suppose we can easily build an on-chip static memory with a 800 pS access time, but the fastest dynamic memories that we can buy for main memory have an average access time of 10 nS. How high of a hit rate do we need to sustain an average access time of 1 nS?

Solve for α: $t_{ave} = t_c + (1-\alpha)t_m$

$\alpha = 1 - (t_{ave} - t_c)/t_m = 1 - (1-0.8)/10 = 98\%$

Caches really need to be good! And they are!

Basic Cache Algorithm



Cache-lines might contain multiple sequential words from memory, thus amortizing the number of tag bits per data bits.

A Cache "tag" is part of the address necessary to determine if a cache entry contains the needed value.

"X" here is a memory address.

ON REFERENCE TO Mem[X]:
Look for X among cache tags...

HIT: X == TAG(i) , for some ***cache line i***
READ: return DATA(i)
WRITE: change DATA(i);
Start Write to Mem(X)

MISS: X not found in any TAG of the cache
REPLACEMENT SELECTION:
Select some LINE k to hold Mem[X] (Allocation)
READ: Read Mem[X]
Set TAG(k)=X, DATA(k)=Mem[X]
WRITE: Start Write to Mem(X)
Set TAG(k)=X, DATA(k)= new Mem[X]

Searching for Tags

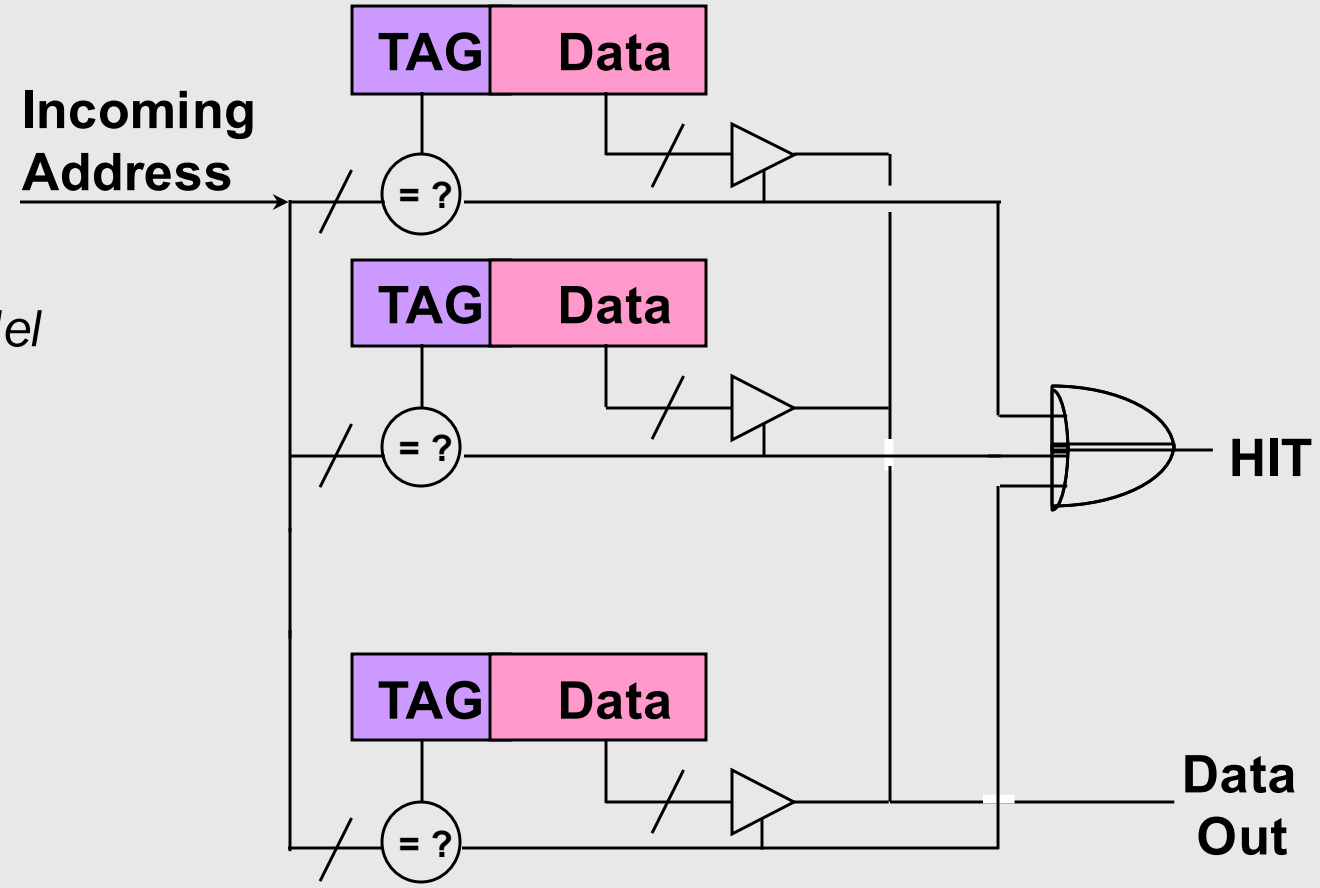
Associativity: *Degree of parallelism used to lookup tags*

Fully-Associative Cache:

The extreme in associatively:

All TAGS are searched in parallel

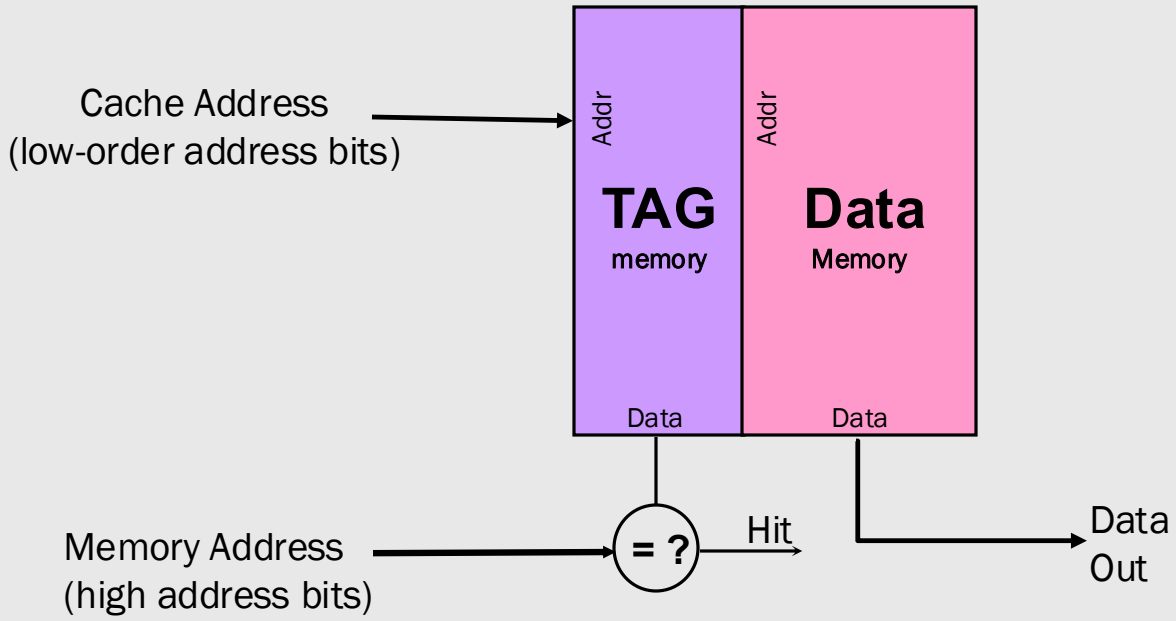
*Data items from *any* address can be located in *any* cache line*



The Other Extreme

Direct-Mapped: If it is in cache, it is in exactly one place

Non-associative or “one-way” associative. No parallelism.
Uses only one comparator and ordinary RAM for tags:



Low-Cost leader:

Direct-mapped caches require a means for translating “Memory Addresses” to “Cache Addresses”. A simple hash function.

Direct-Mapped Example

Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

With 8-byte lines, 3 low-order bits determine the byte within the line.

With 4 cache lines, the next 2 bits can be used to decide which line to use

$1024_{10} = 10000000000_2 \rightarrow \text{line} = 00_2 = 0_{10}$

$1000_{10} = 01111101000_2 \rightarrow \text{line} = 01_2 = 1_{10}$

$1040_{10} = 10000010000_2 \rightarrow \text{line} = 10_2 = 2_{10}$

Cache

	Tag	Data	
Line 0	1024	44	99
Line 1	1000	17	23
Line 2	1040	1	4
Line 3	1016	29	38

Direct-Mapped Miss

Memory

1000	17
1004	23
1008	11
1012	5
1016	29
1020	38
1024	44
1028	99
1032	97
1036	25
1040	1
1044	4

What happens when we now ask for address 1008?

$1008_{10} = 0111111\mathbf{10}000_2 \rightarrow \text{line} = 10_2 = 2_{10}$

but earlier we put 1040 there…

$1040_{10} = 1000001\mathbf{10}000_2 \rightarrow \text{line} = 10_2 = 2_{10}$

There is only one line that it can go in, so we replace 1040's contents with those of 1008.

Cache

	Tag	Data	
Line 0	1024	44	99
Line 1	1000	17	23
Line 2	**1008**	**11**	**5**
Line 3	1016	29	38

Fully-Assoc. vs. Direct-mapped

Fully-associative N-line cache:

- N tag comparators, registers used for tag/data storage ($$$)
- Location A can be stored in ANY of the N cache lines; no “collisions”
- Needs a *replacement strategy* to pick which line to use when loading new word(s) into cache

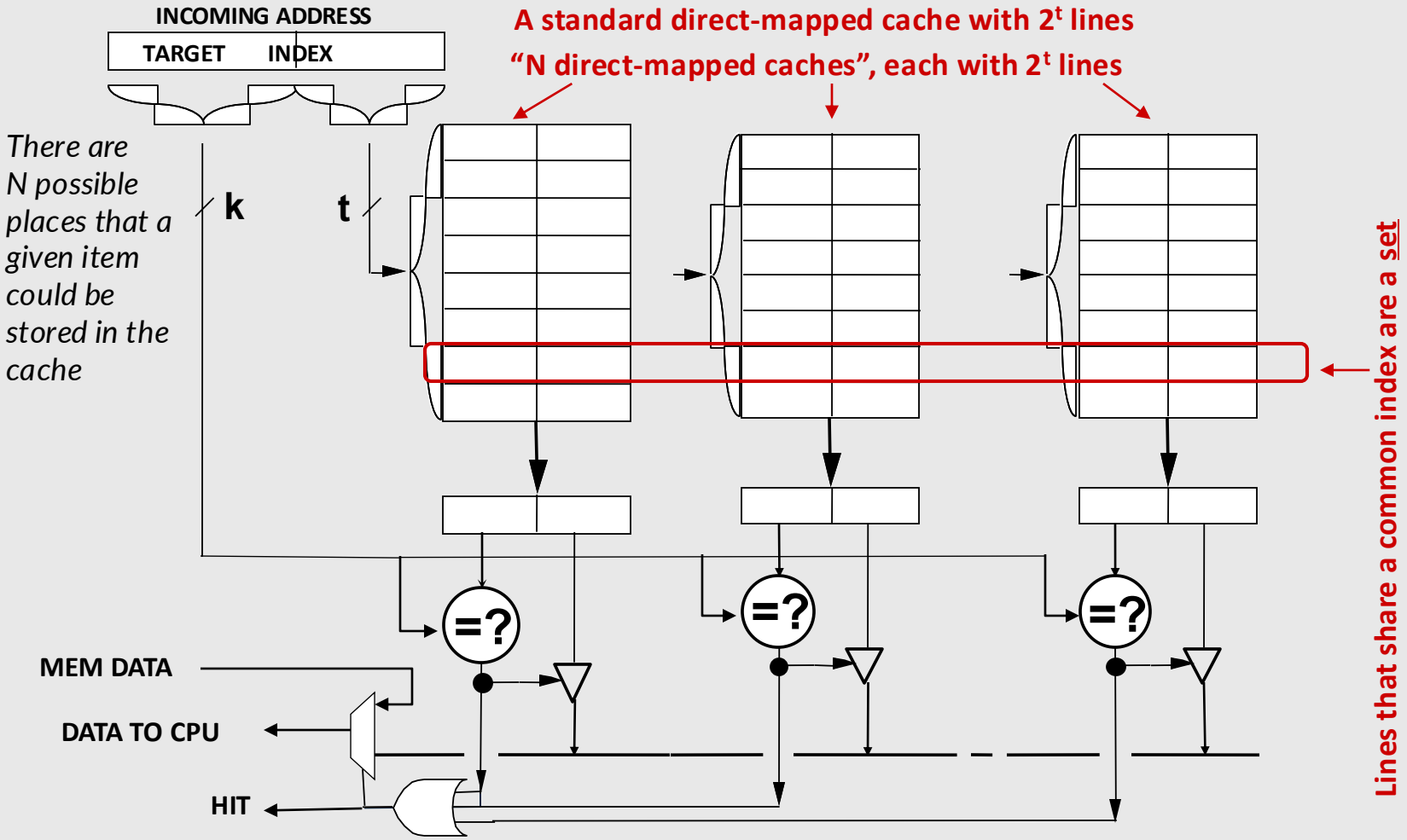
COLLISIONs occur when there are multiple items that we’d like to keep cached, we have room, but our management policies only keeps a subset of them.

Direct-mapped N-line cache:

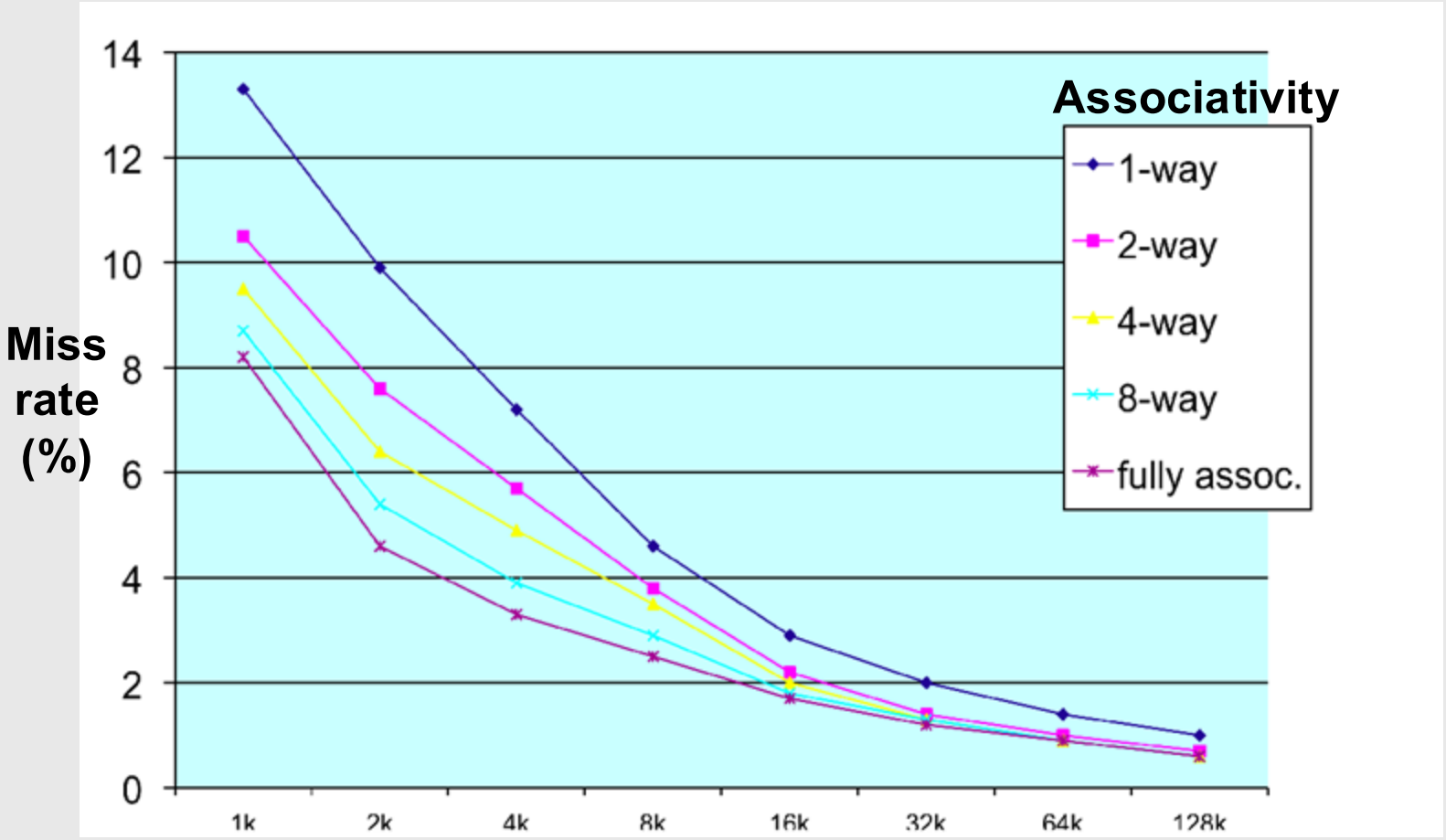
- One tag comparator, SRAM used for tag/data storage ($)
- Location A is stored in a SPECIFIC line of the cache determined by its address; address “collisions” possible
- No replacement strategy is needed; each word can only be cached at one specific cache line

Is there something in-between?

N-Way Set-Associative Cache



Associativity vs. Miss Rate



8-way is (almost) as effective as fully-associative

Handling Writes

Observation: Most (80+%) of memory accesses are READs, but writes are essential.
How should we handle writes?

Policies:

- *WRITE-THROUGH*: CPU writes are cached, but also written to main memory (stalling the CPU until write is completed). Memory always holds “the truth”.
- *WRITE-BACK*: CPU writes are cached, but not immediately written to main memory. Memory contents can become “stale”.

Additional Enhancements:

- WRITE-BUFFERS: For either write-through or write-back, writes to main memory are buffered. CPU keeps executing while writes are completed (in order) in the background.

What combination has the highest performance?

Write-Through

ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: ***X == TAG(i) , for some cache line i***

READ: return DATA[i]

WRITE: change DATA[i]; **Start Write to Mem[X]**

MISS: ***X not found in TAG of any cache line***

REPLACEMENT SELECTION:

Select some line k to hold Mem[X]

READ: Read Mem[X]

Set TAG[k] = X, DATA[k] = Mem[X]

WRITE: **Start Write to Mem[X]**

Set TAG[k] = X, DATA[k] = new Mem[X]

Write-Back

ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: ***X = TAG(i) , for some cache line I***

READ: return DATA(i)

WRITE: change DATA(i); ~~Start Write to Mem[X]~~

MISS: ***X not found in TAG of any cache line***

REPLACEMENT SELECTION:

Select some line k to hold Mem[X]

Write Back: Write Data(k) to Mem[Tag[k]]

READ: Read Mem[X]

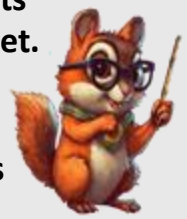
Set TAG[k] = X, DATA[k] = Mem[X]

WRITE: ~~Start Write to Mem[X]~~

Set TAG[k] = X, DATA[k] = new Mem[X]

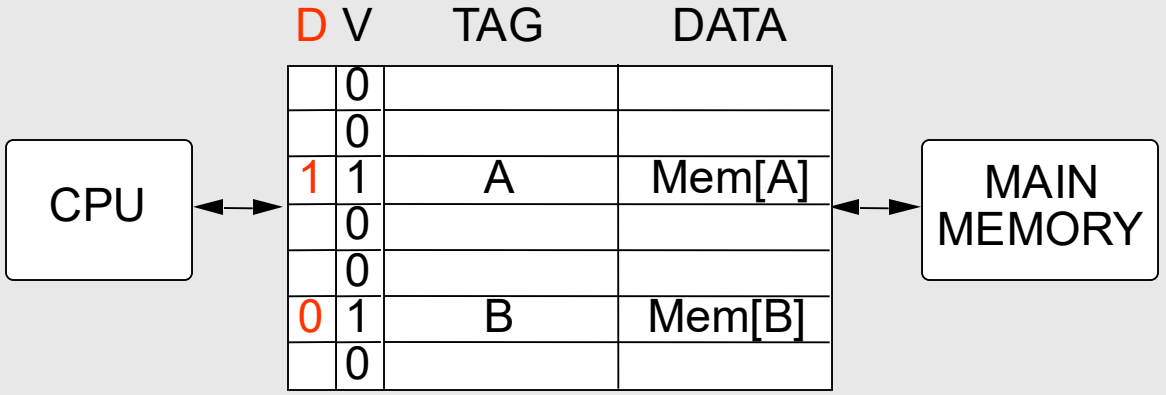
Write-Back with “Dirty” bits

Dirty and Valid bits are per line of a set.



The Dirty bit is set if the line was written, but not updated in memory.

The Valid bit is set to indicate that the cache is actually holding a value. On startup, reset and changes of programs the cache should be invalidated by clearing its Valid bits



What if the cache has a block-size larger than one?

A) If only one word in the line is modified (Dirty), we write back ALL words

B) On a MISS, we READ the line BEFORE we WRITE it.

ON REFERENCE TO Mem[X]: Look for X among tags...
HIT: ***X = TAG(i) , for some cache line I***
READ: return DATA(i)
WRITE: change DATA(i); ~~Start Write to Mem[X]~~, D[i]=1
MISS: ***X not found in TAG of any cache line***
REPLACEMENT SELECTION:
Select some line k to hold Mem[X]
If D[k] == 1 the Write Data(k) to Mem[Tag[k]]
READ: Read Mem[X]; Set TAG[k] = X, DATA[k] = Mem[X], D[k]=0
WRITE: ~~Start Write to Mem[X]~~
Read Mem[X]
Set TAG[k] = X, DATA[k] = new Mem[X], D[k]=1

Cache Design Summary

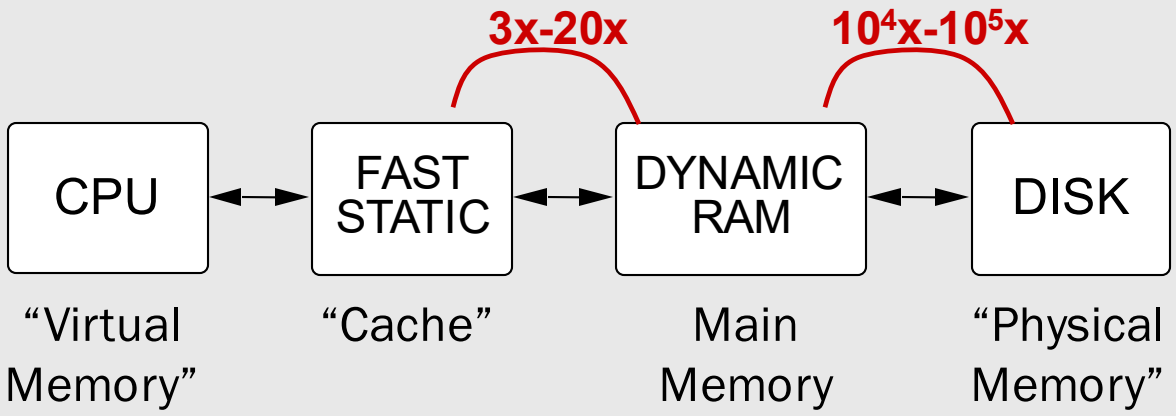
Various design decisions the affect cache performance

- Block size, exploits spatial locality, saves tag H/W, but, if blocks are too large you can load unneeded items at the expense of needed ones
- Write policies
- Write-through – Keeps memory and cache consistent, but leads to high memory traffic
- Write-back – allows memory to become STALE, but reduces memory traffic

No simple answers, in the real-world cache designs are based on simulations using memory traces.

VIRTUAL MEMORY

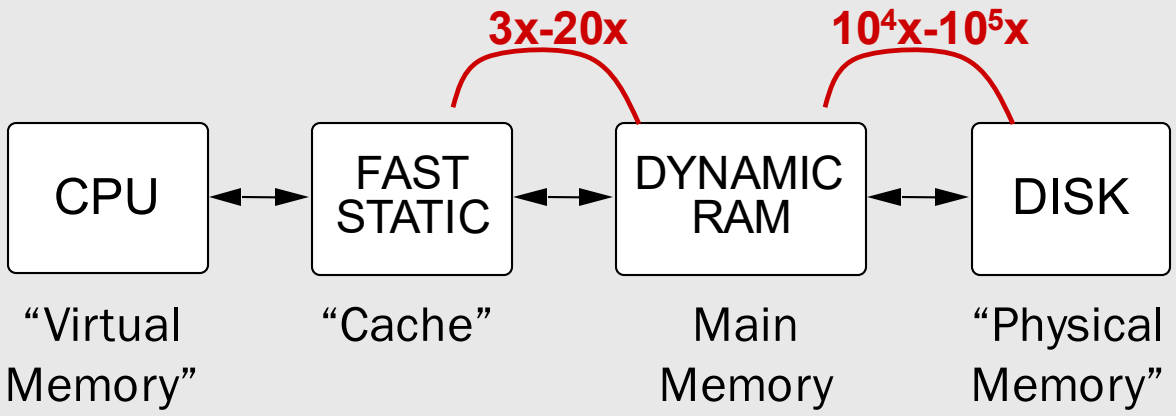
Extending the Memory Hierarchy



So far, we’ve used SMALL fast memory + BIG slow memory to fake a BIG FAST memory (caching).

Can we combine RAM and DISK to fake MUCH larger memory “size” at near RAM speeds?

Extending the Memory Hierarchy

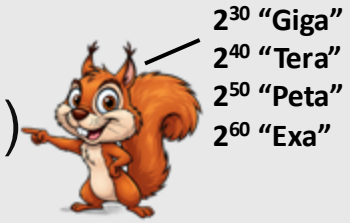


VIRTUAL MEMORY

- *Use RAM as cache to a much larger storage pool, on slower devices*
- *TRANSPARENCY - VM locations "look" the same to program whether on DISK or in RAM.*
- *ISOLATION of actual RAM size from software.*
- *Support for MULTIPLE, SIMULTANEOUS ADDRESS SPACES*

Virtual Memory

ILLUSION: Huge memory
(2^{32} (4G) bytes? 2^{64} (18E) bytes?)



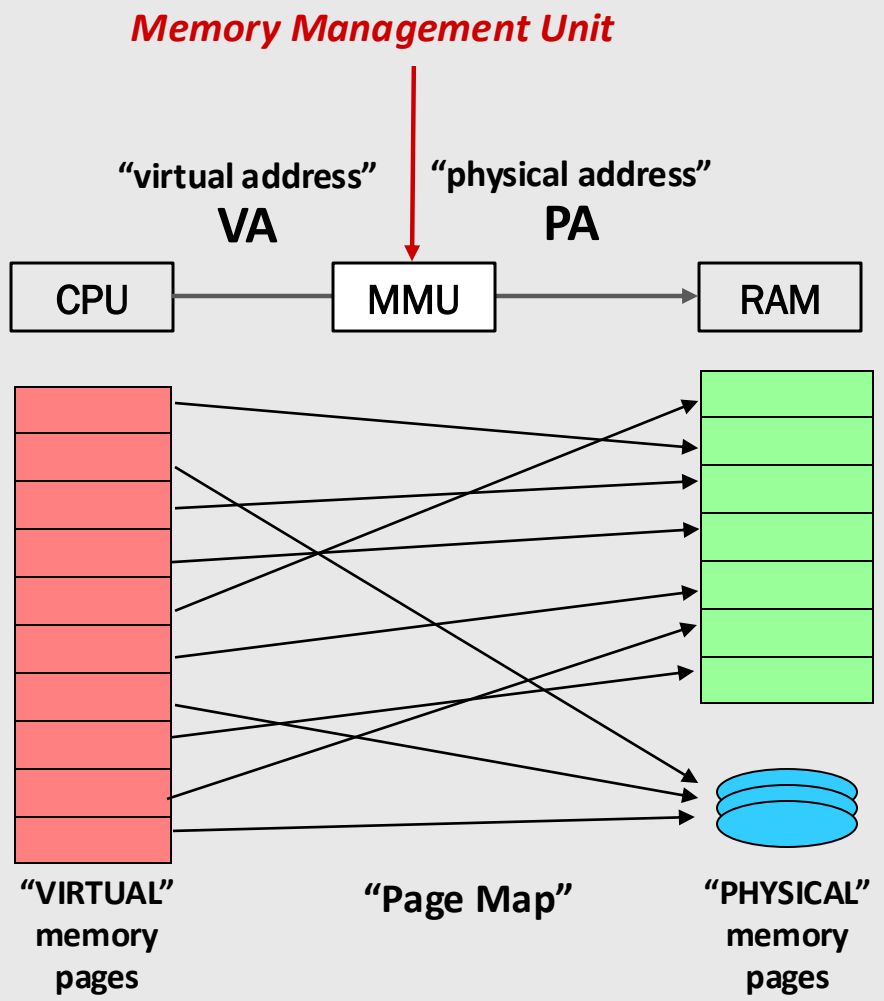
ACTIVE USAGE: small fraction
(2^{28} bytes?)

Actual HARDWARE:

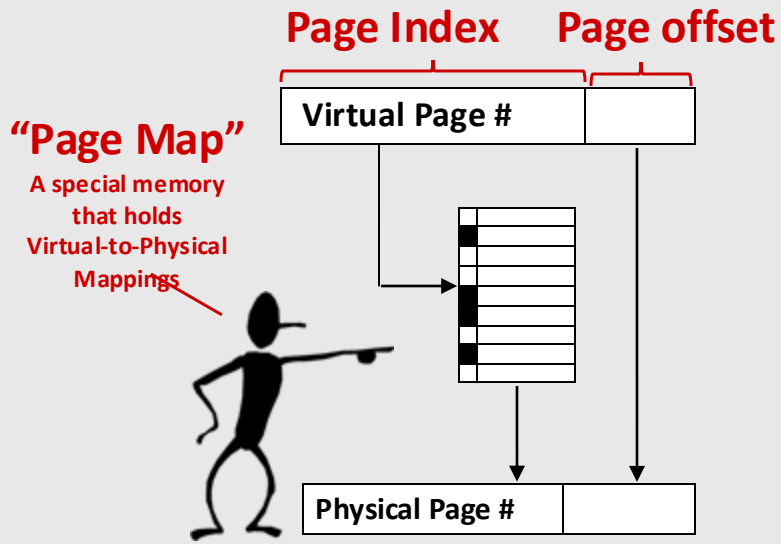
- 2^{33} (8G) bytes of RAM
- 2^{39} (500G) bytes of DISK...
  ... maybe more, maybe less!

ELEMENTS OF DECEIT:

- Partition memory into manageable chunks--"Pages" (4K-8K-16K-64K)
- MAP a few to RAM, assign others to DISK
- Keep "HOT" pages in RAM.



A Simple Page Map Design

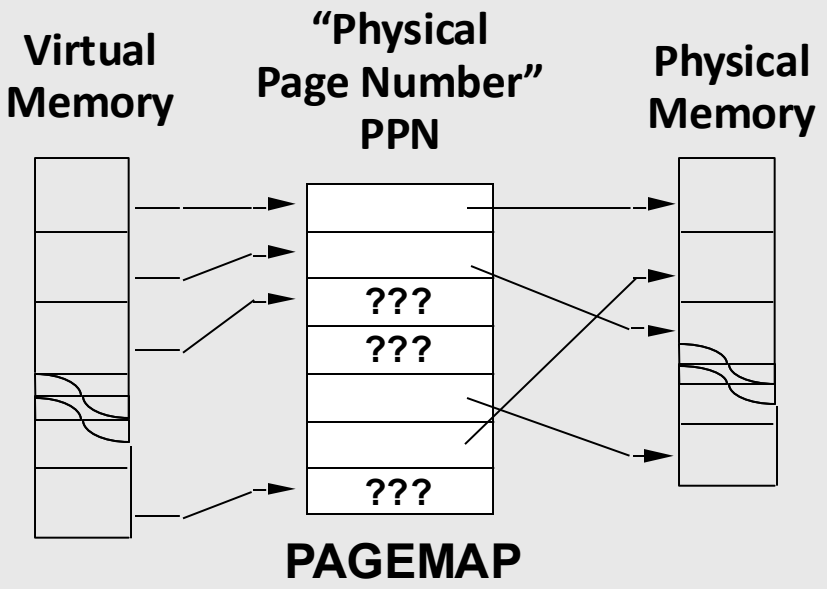


FUNCTION: Given Virtual Address,

- **Map to PHYSICAL address**

OR

- **Cause PAGE FAULT allowing page replacement**



Why use HIGH address bits to index pages?

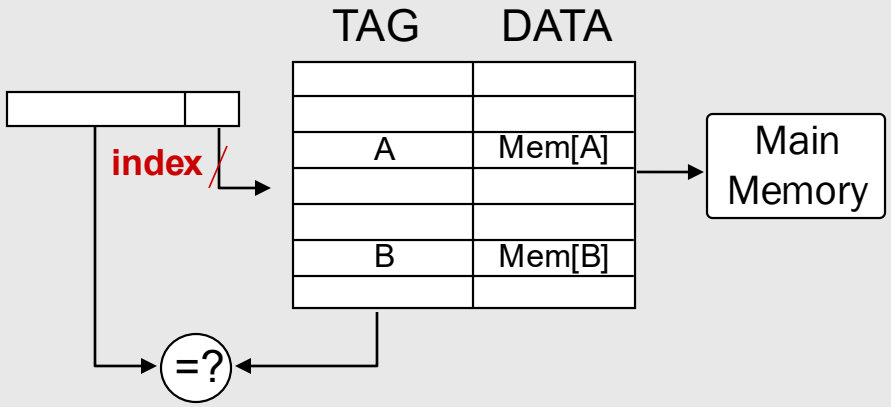
... LOCALITY.
Keeps related data on same page.

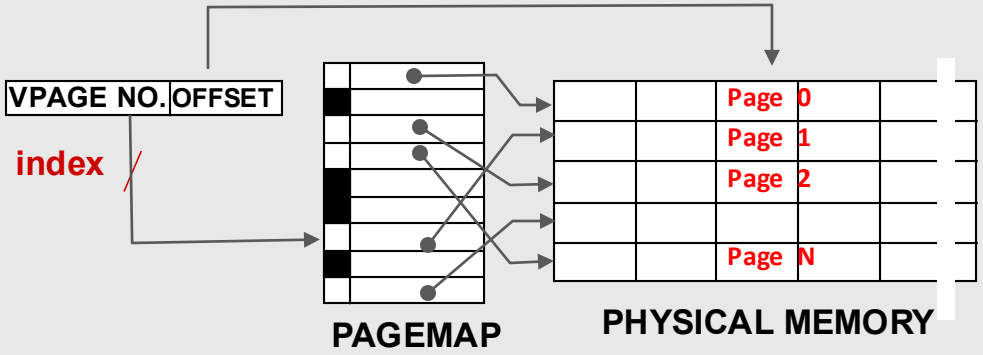
Why use LOW address bits to index cache lines?

... LOCALITY.
Keeps related data from competing for same cache lines.

Virtual Memory vs. Cache

Key idea: cache handles speed, virtual memory handles capacity & protection. Both rely on locality! ☺





CACHE:

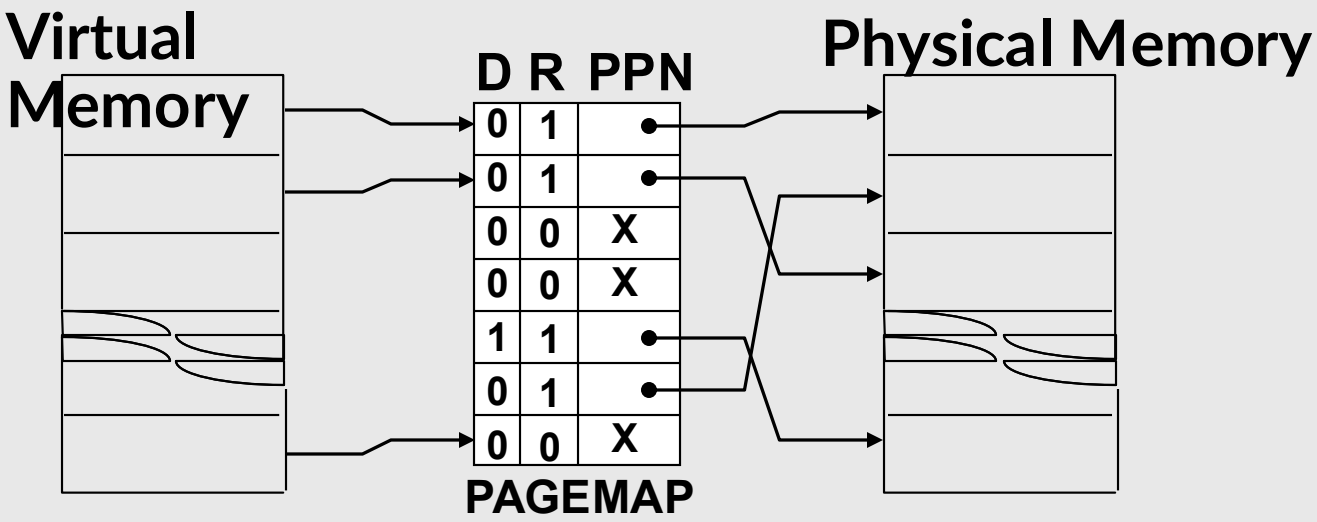
- Relatively small blocks (16-64 bytes)
- Few lines: scarce resource
- Miss time: 3x-20x hit time

VIRTUAL MEMORY:

- Disk or SSD: long latency, fast xfer
  - miss time: ~10^5 x hit time
  - write-back essential!
  - large pages in RAM
- Lots of lines: one for each page
- Vpage mapping is determined by an index (i.e. “direct-mapped” w/o tag) data in physical memory

Virtual Memory: A H/W view

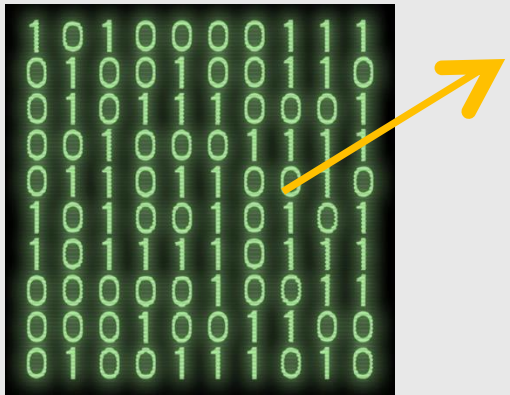
Take OS to learn more!! ☺

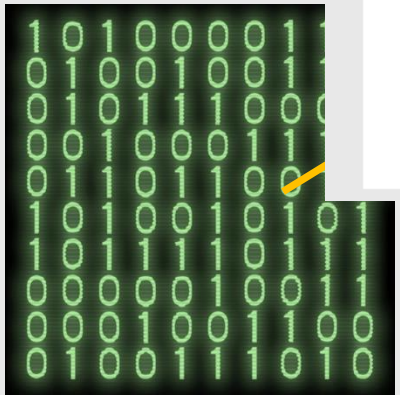


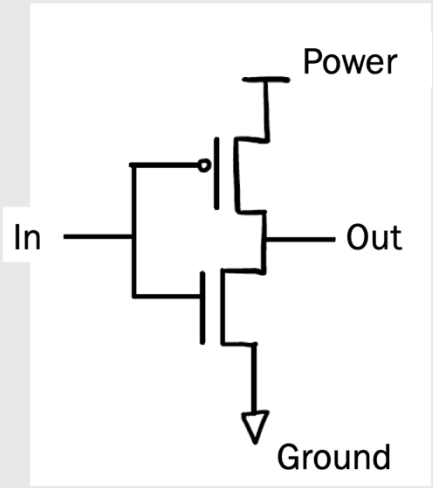
Ideal Pagemap Characteristics:

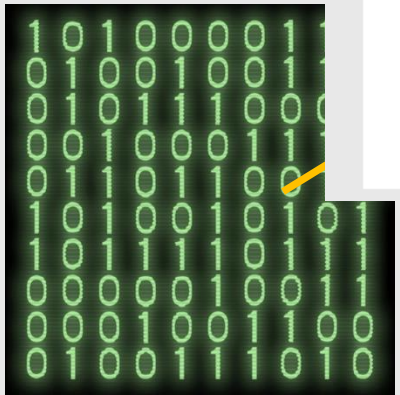
- *One entry per virtual page!*
- *Contains PHYSICAL page number (PPN) of each resident page*
- *RESIDENT bit = 1 for pages stored in RAM, or 0 for non-resident (on disk or unallocated). Page fault when R = 0.*
- *DIRTY bit says the page has changed since loading it from disk (and therefore need to write it back to disk when it's replaced)*

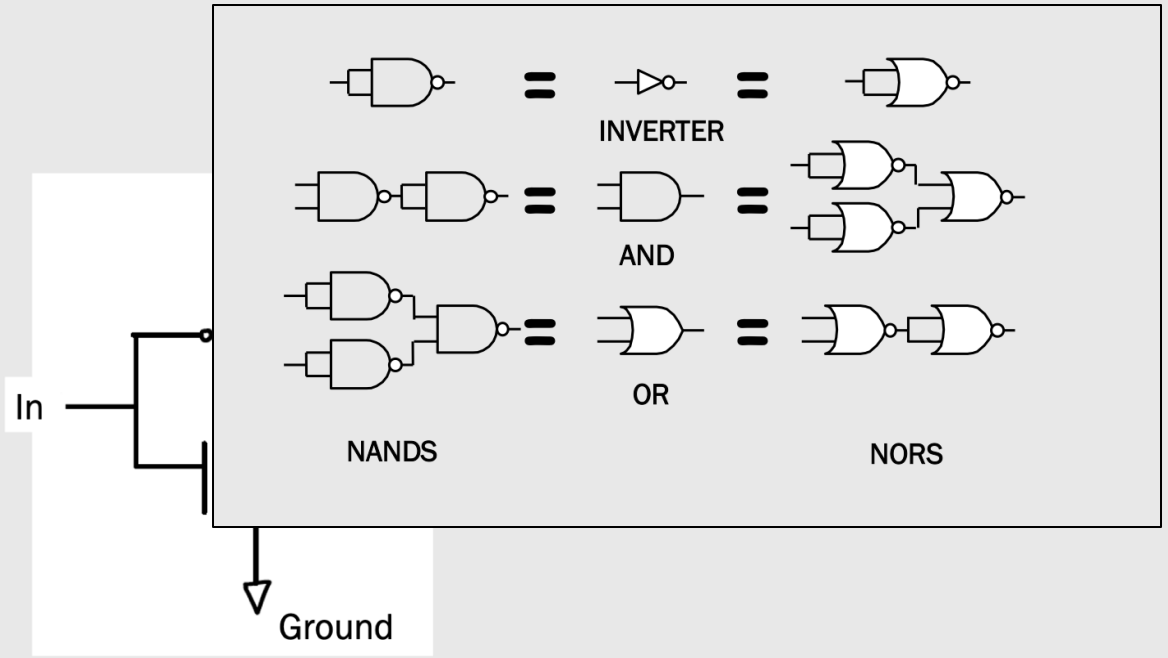
COURSE SUMMARY!

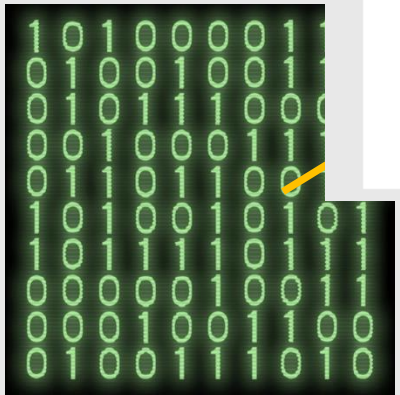


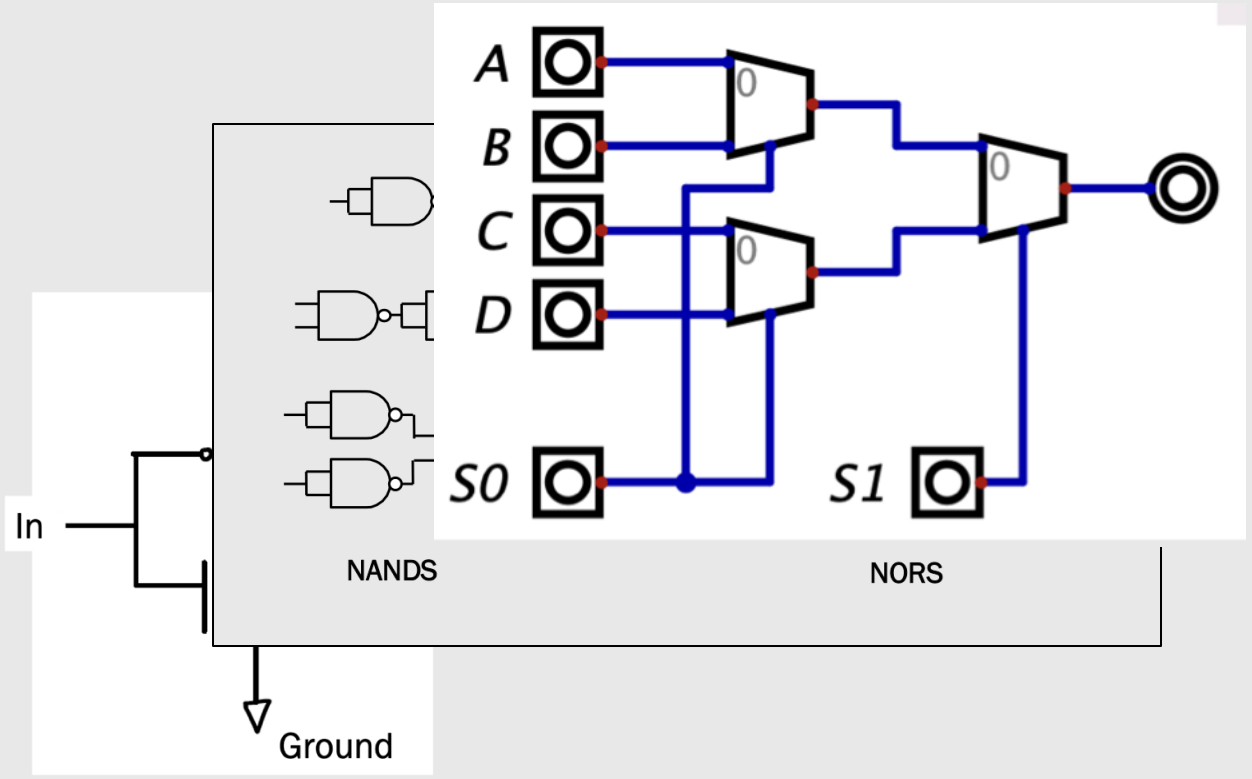


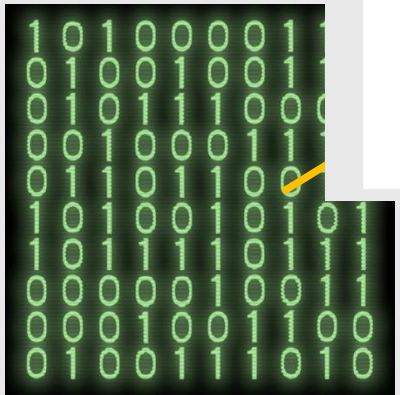

Power
In
Out
Ground

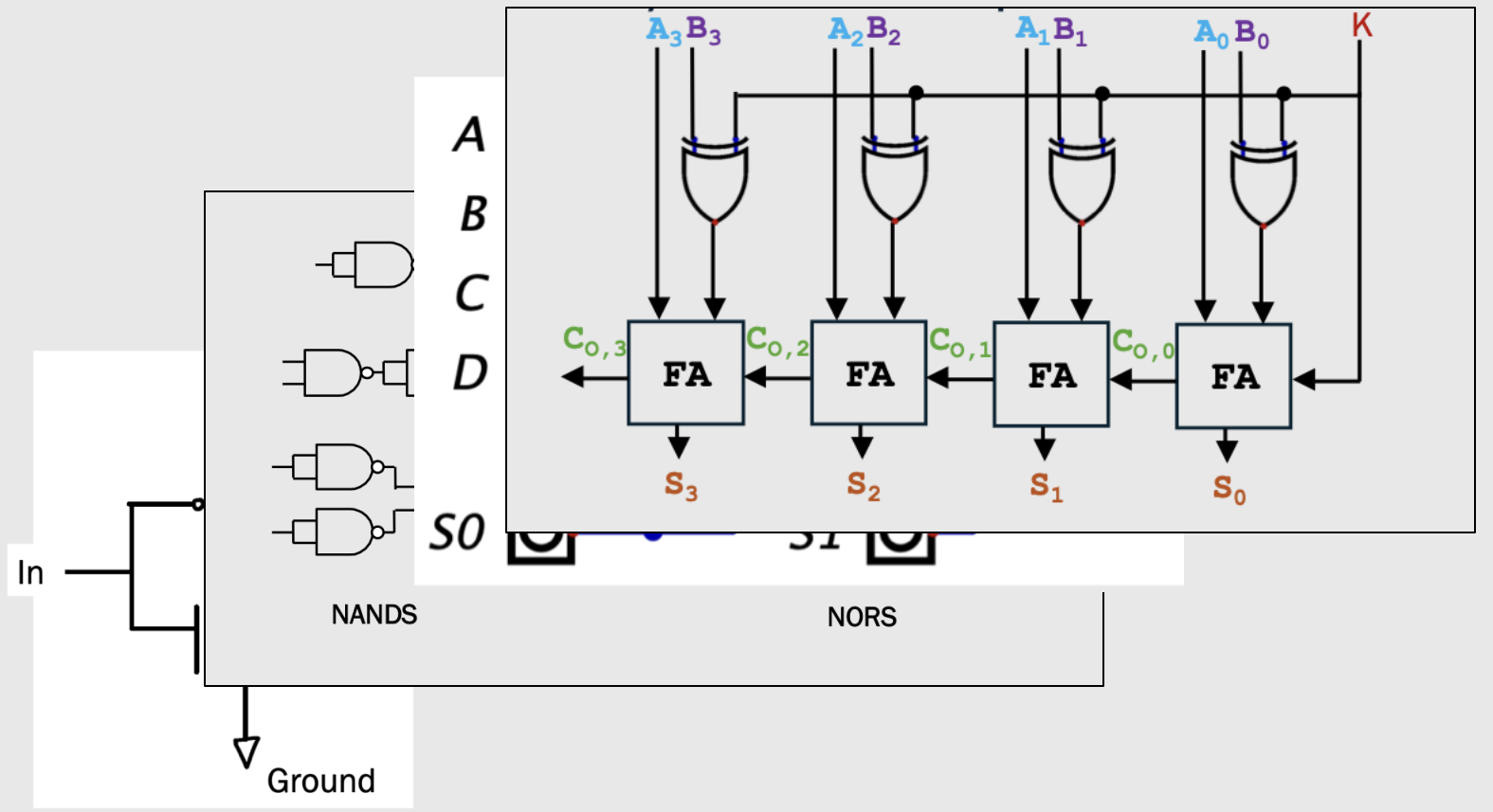


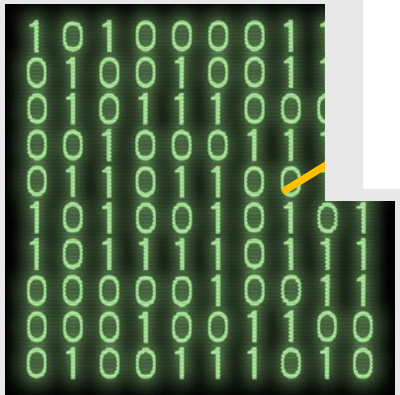

INVERTER
AND
OR
NANDS
NORS
In
Ground

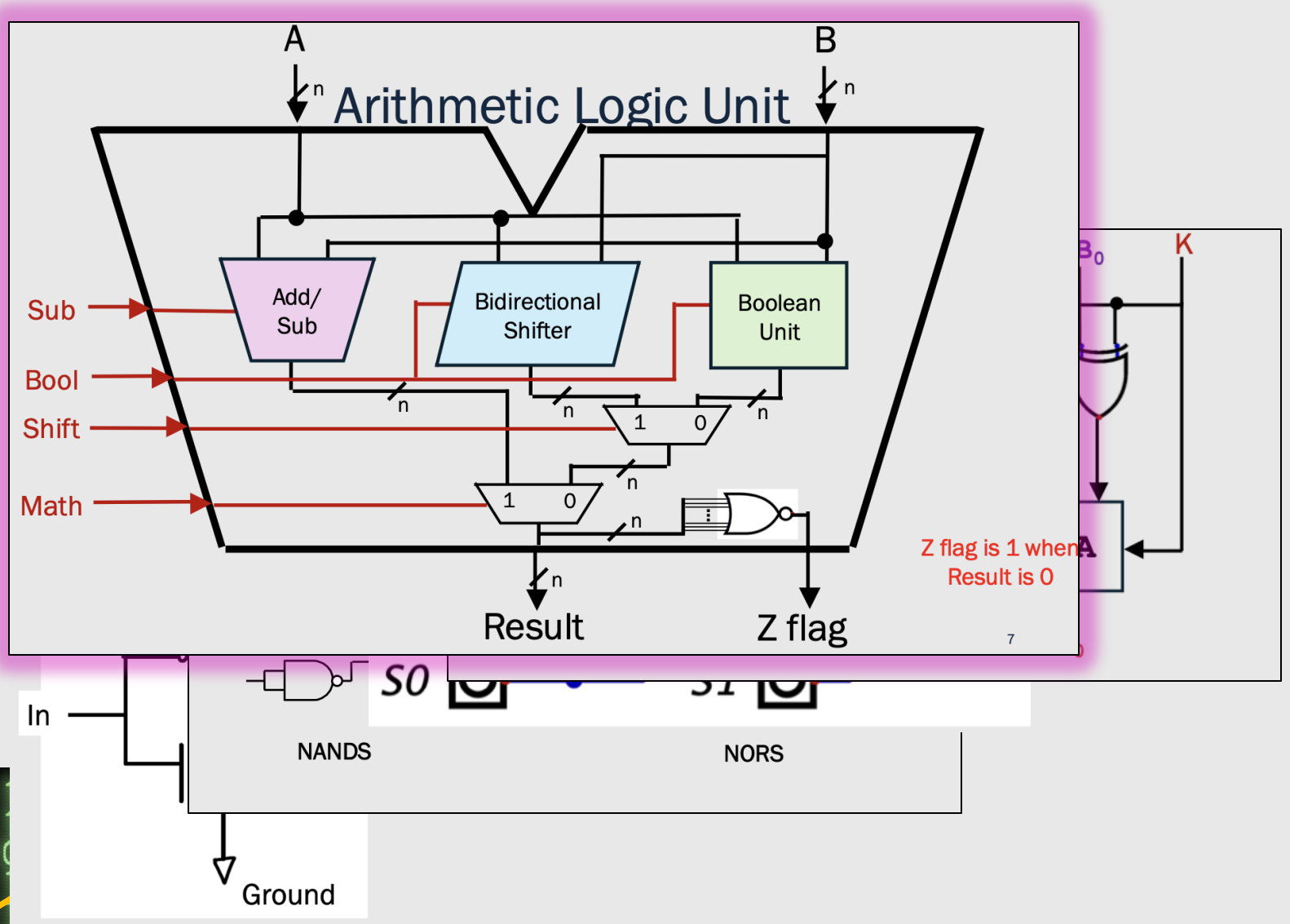


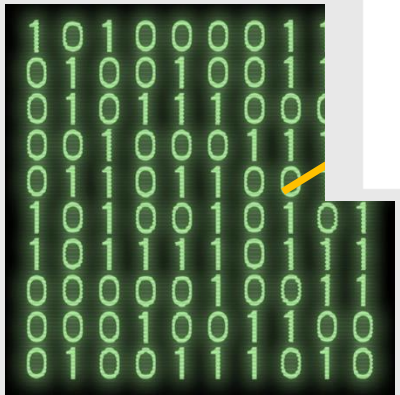

A
B
C
D
S0
S1
0
0
0
In
NANDS
NORS
Ground

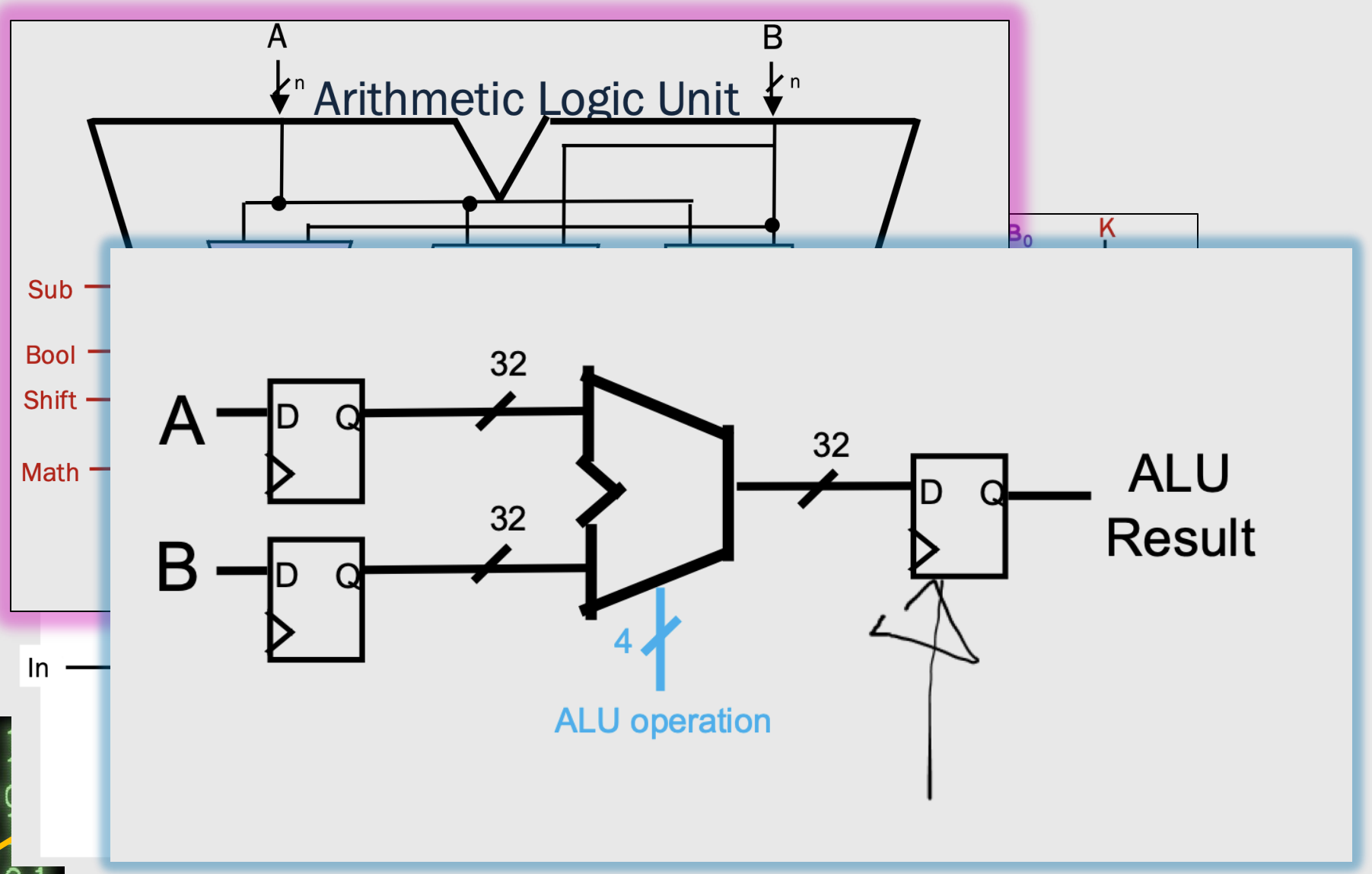


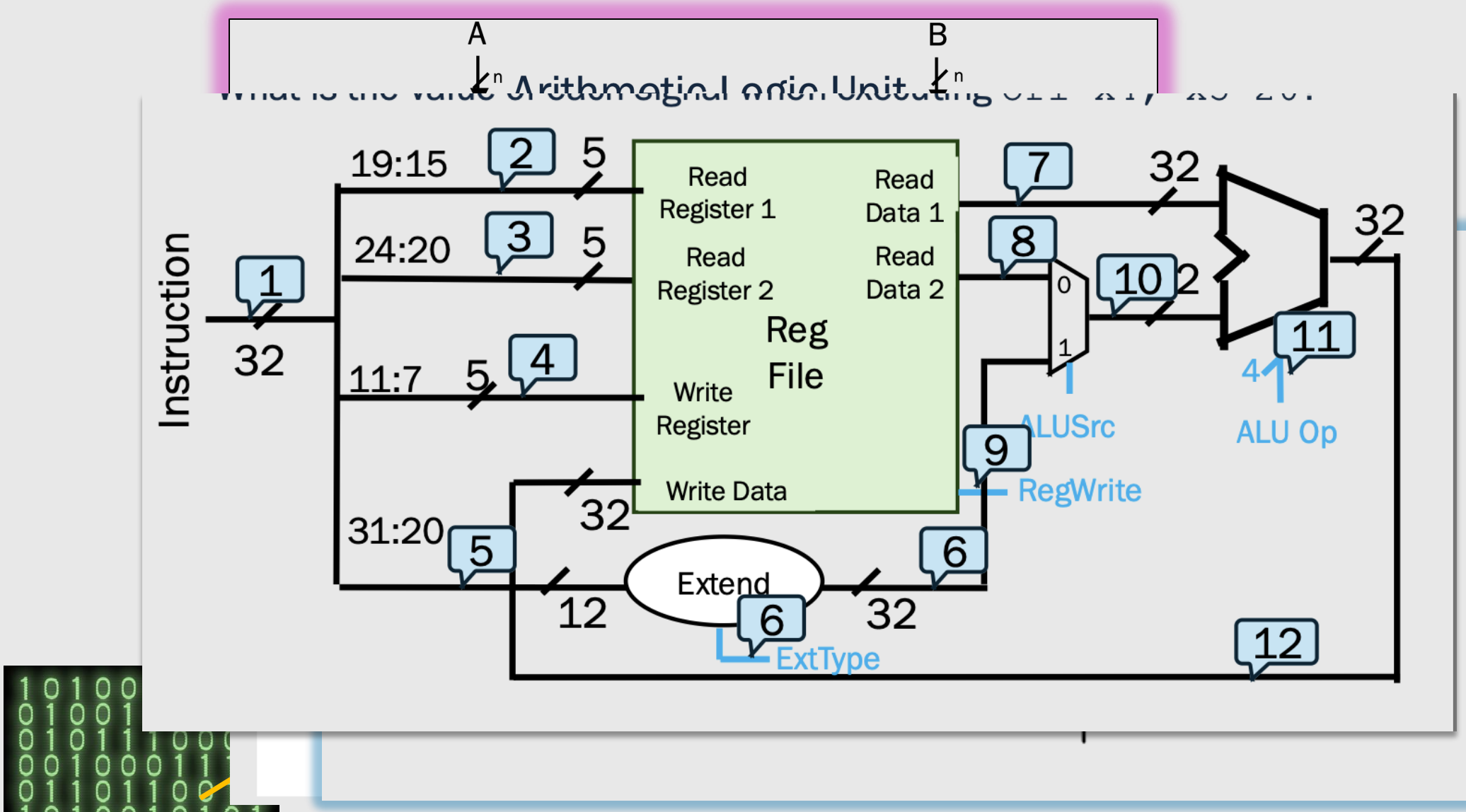

A
B
C
D
S0
In
NANDS
NORS
Ground
A3B3
A2B2
A1B1
A0B0
K
FA
FA
FA
FA
Co,3
Co,2
Co,1
Co,0
S3
S2
S1
S0

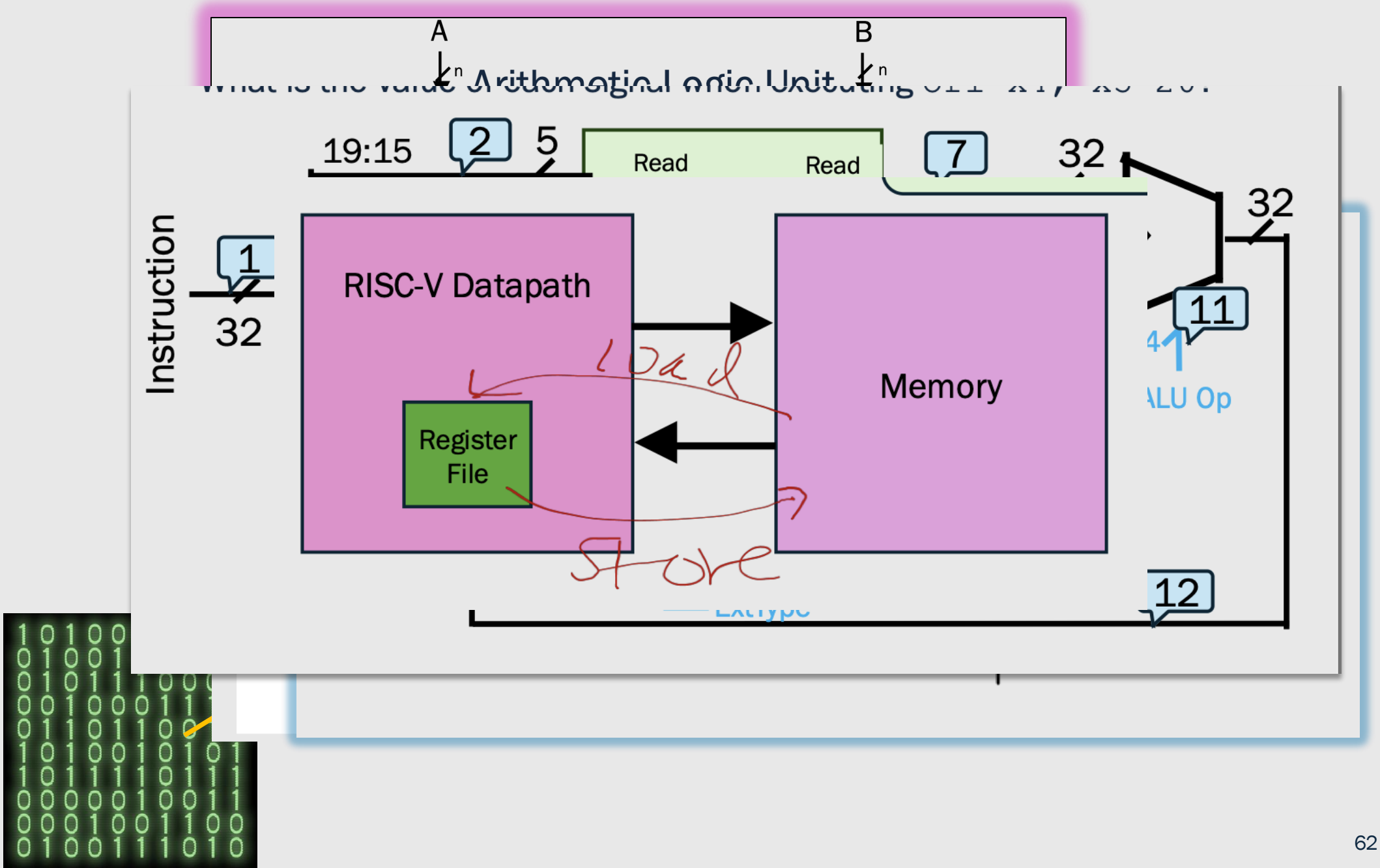


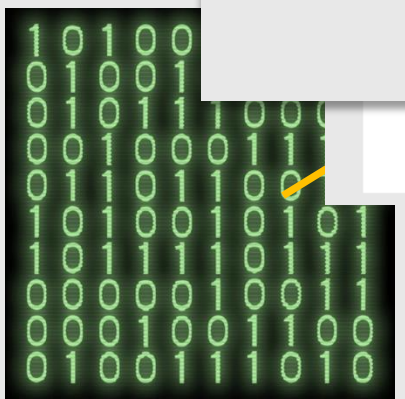

A
B
n
n
Arithmetic Logic Unit
Sub
Bool
Shift
Math
Add/
Sub
Bidirectional
Shifter
Boolean
Unit
n
n
n
1
0
n
1
0
n
n
n
Result
Z flag
Z flag is 1 when
Result is 0
7
K
In
S0
NANDS
NORS
Ground




A
n
B
n
Arithmetic Logic Unit
Sub
Bool
Shift
Math
In
A
B
D
Q
32
32
32
4
ALU operation
ALU
Result


A
n
B
n
Arithmetical and Unit
Instruction
32
1
19:15
2
5
24:20
3
5
11:7
4
5
31:20
5
12
Read
Register 1
Read
Register 2
Reg
File
Write
Register
Write Data
Read
Data 1
Read
Data 2
32
32
Extend
6
32
7
8
0
1
ALUSrc
9
RegWrite
10
2
32
11
4
ALU Op
6
ExtType
12


A
n
B
n
19:15
2
5
Read
Read
7
32
32
Instruction
1
32
RISC-V Datapath
Memory
11
ALU Op
Register
File
Load
Store
12



If you enjoyed this class…

- **Yay! Me too! Thank you all for taking it with me!**
- Consider taking…
  - *COMP541! (Digital Logic & Design! I will be teaching it in Fall 2026!)*
    - Build a single-cycle CPU using a hardware description language (Verilog)
    - End of course project is programming it in assembly! (
  - *COMP520 (Compilers!)*
    - Learn about the SW programs that take high-level code to machine code
    - Implement a compiler for a subset of C. Compiling to RISC-V!
  - *COMP530 (Operating Systems!)*
    - Learn more about the memory hierarchy, computer performance & the HW/SW interface
- Consider joining the undergrad systems reading group!
  - *Info is on my website, or you can email me!*