

CSE 31

Computer Organization

Lecture 21 – Cache (3)

CPU Design (1)



Announcement

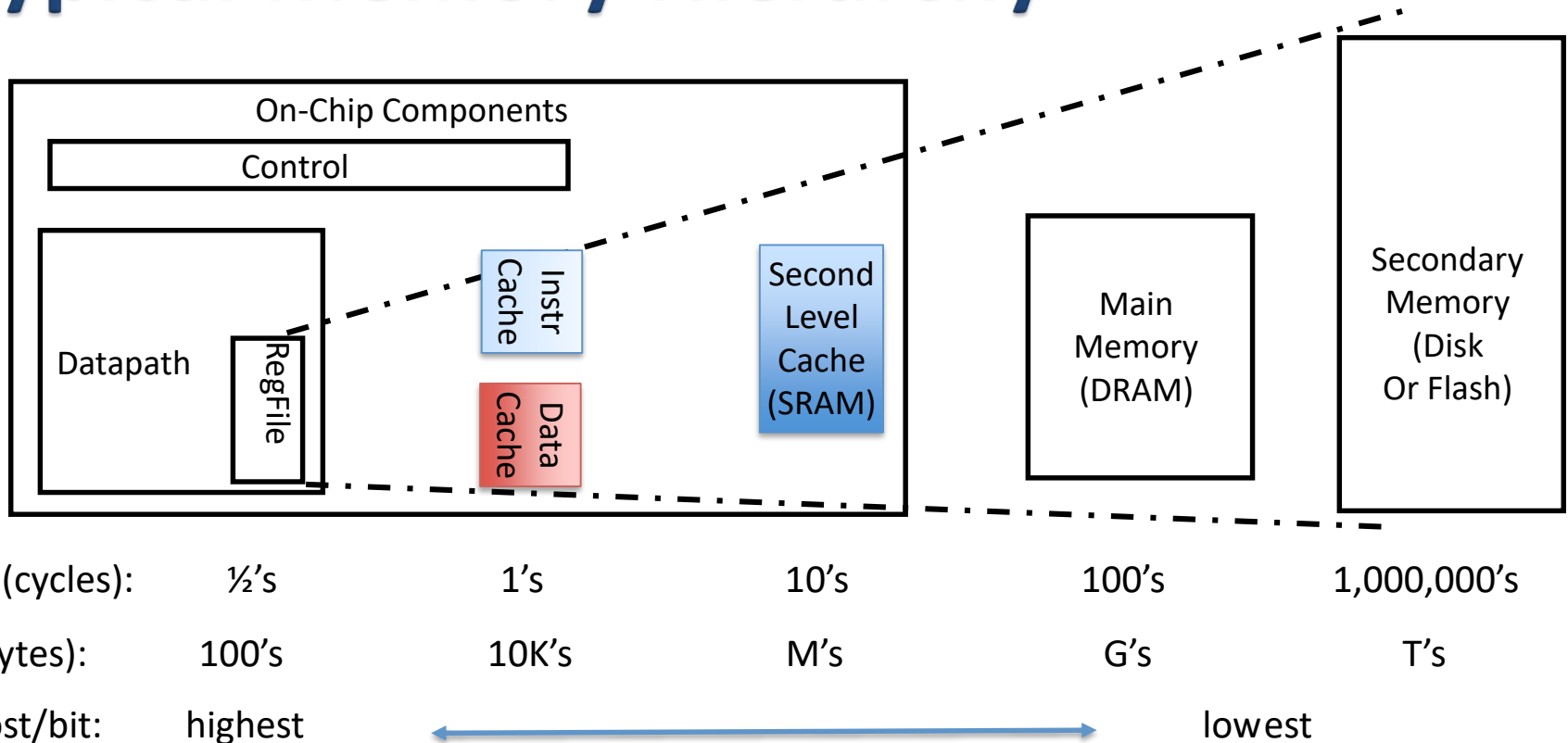
- ▶ Lab #9
 - Due in 1 week
- ▶ Project #2
 - Start working on it during lab this week
 - Due Monday (4/29)
- ▶ HW #7 in CatCourses
 - Due Monday (5/6) at 11:59pm
- ▶ Reading assignment
 - Chapter 5.1-5.6 of zyBooks
 - Make sure to do the Participation Activities
 - Due Friday (4/26) at 11:59pm
 - Chapter 5.7-5.11 of zyBooks
 - Make sure to do the Participation Activities
 - Due Friday (5/3) at 11:59pm

Announcement

▶ Midterm Exam 2

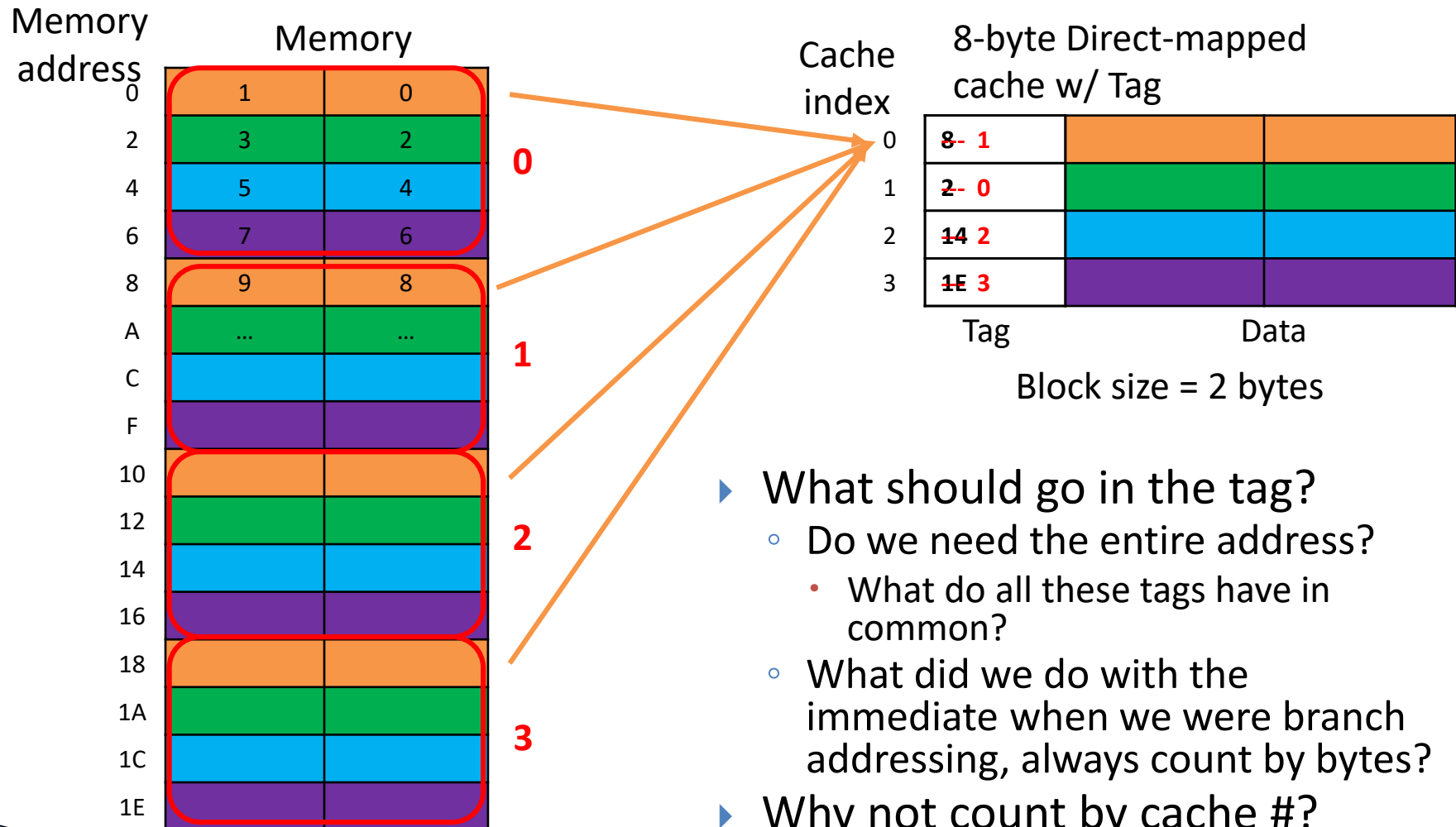
- 4/24 (Wednesday, in lecture) Not 4/17 as scheduled
- Lectures #8 - #18
- HW #2 - #6
- Practice exam in CatCourses
- Closed book
- 1 sheet of note (8.5" x 11")
- MIPS reference sheet will be provided
- Review: Tomorrow (4/23) 12-2pm, COB2 390

Typical Memory Hierarchy



- ▶ **Principle of locality + memory hierarchy** presents programmer with as much memory as is available in the *cheapest* technology at the speed offered by the *fastest* technology

Direct-Mapped Cache (4/4)



- ▶ What should go in the tag?
 - Do we need the entire address?
 - What do all these tags have in common?
 - What did we do with the immediate when we were branch addressing, always count by bytes?
- ▶ Why not count by cache #?
 - It's useful to draw memory with the same width as the block size

Issues with Direct-Mapped

- ▶ Since multiple memory addresses map to same cache index, how do we tell which one is in there?
- ▶ What if we have a block size > 1 byte?
 - Answer: divide memory address into three fields

t t	i i i i i i i i i i i	o o o o
---	-----------------------	---------

Tag to check if it has the correct block

Index to select block

Byte offset within block

Direct Mapped Cache

Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

0000 0001 0010 0011 0100 0011 0100 1111

0 miss

00	Mem(0)

1 miss

00	Mem(0)
00	Mem(1)

2 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)

3 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

3 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

15 miss

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

11

15

- 8 requests, 6 misses

Taking Advantage of Spatial Locality

Let cache block hold more than one word

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

0 miss

00	Mem(1)	Mem(0)

1 hit

00	Mem(1)	Mem(0)

2 miss

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

3 hit

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

01 5 4 miss 4

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

3 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

4 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

15 miss

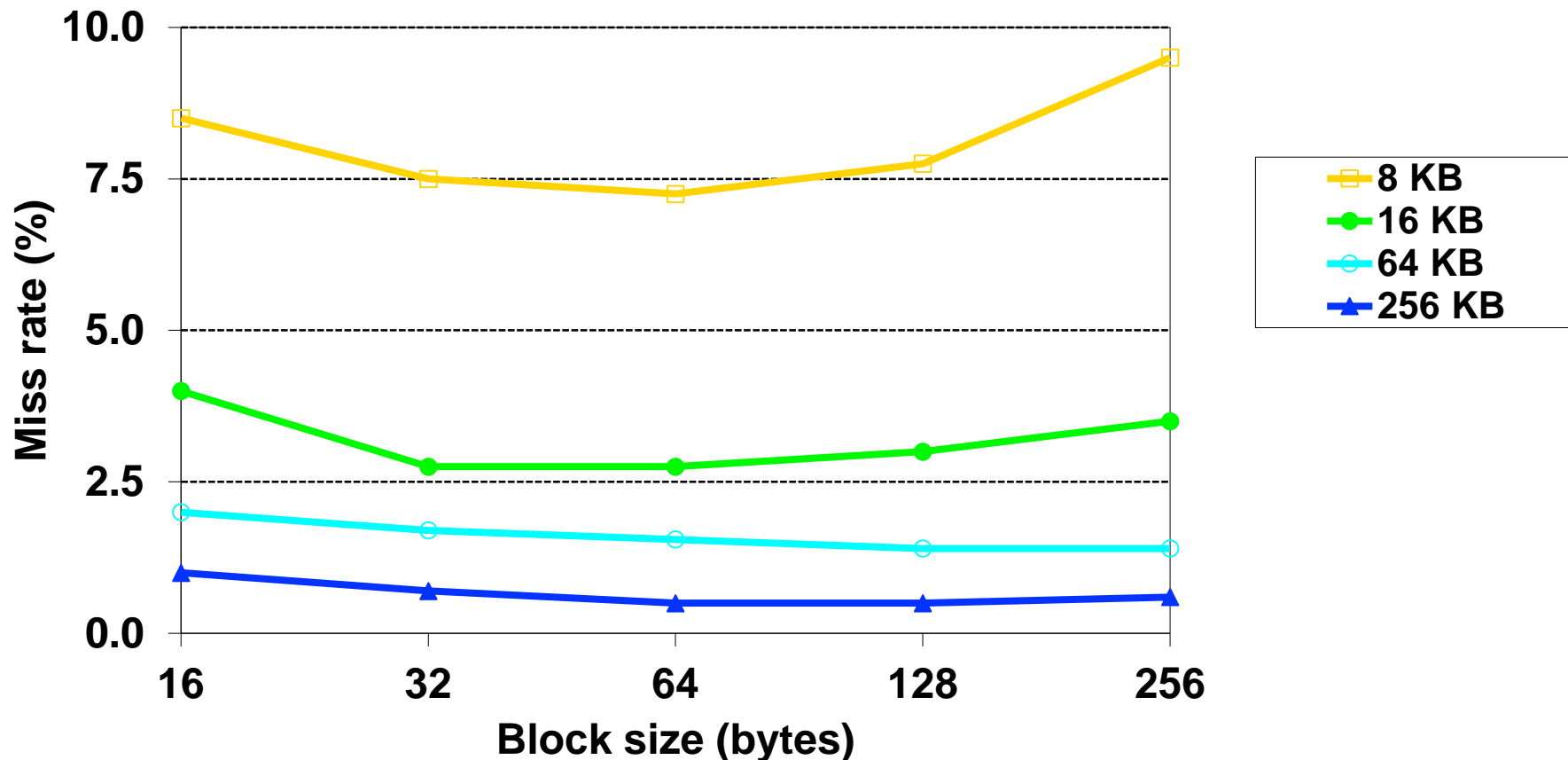
11

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

15 14

- 8 requests, 4 misses

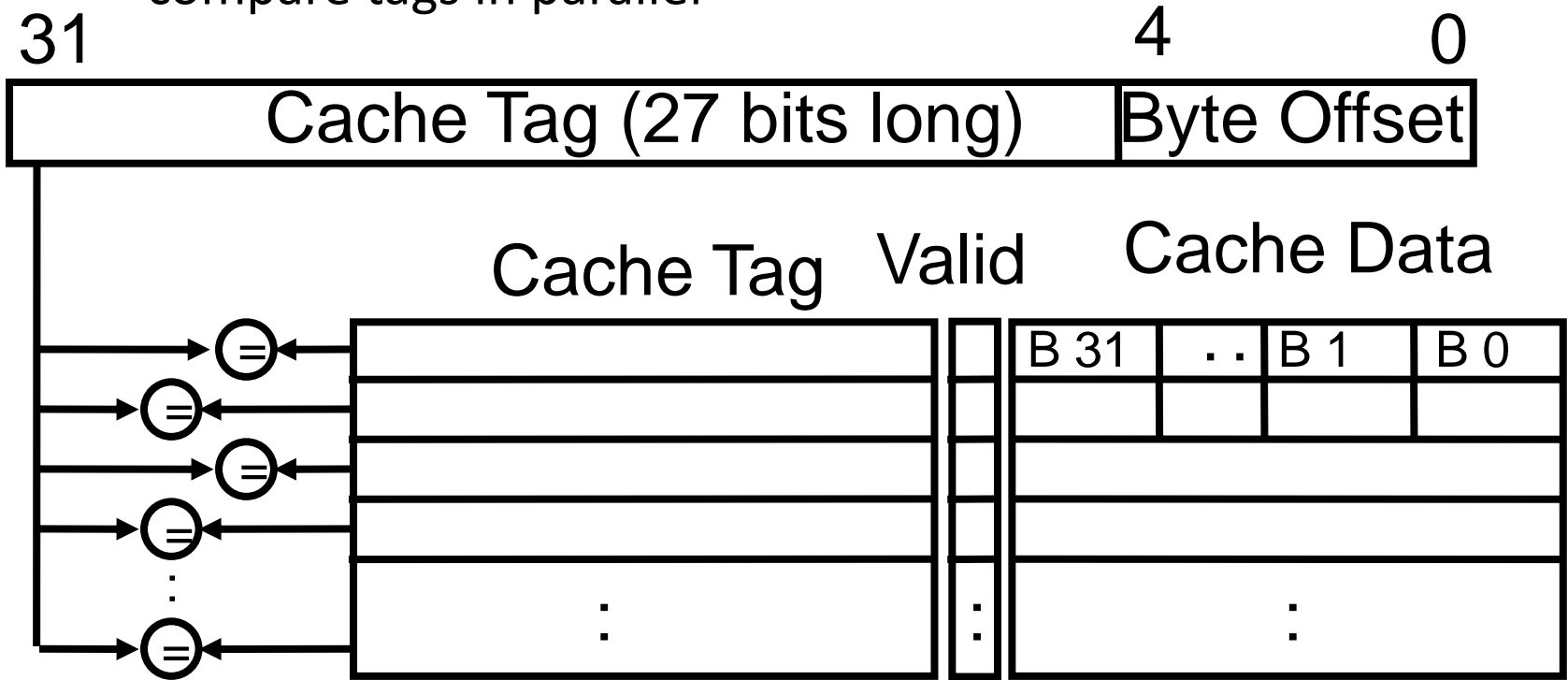
Miss Rate vs Block Size vs Cache Size



Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing capacity misses)

Fully Associative Cache (2/3)

- ▶ Fully Associative Cache (e.g., 32 B block)
 - compare tags in parallel



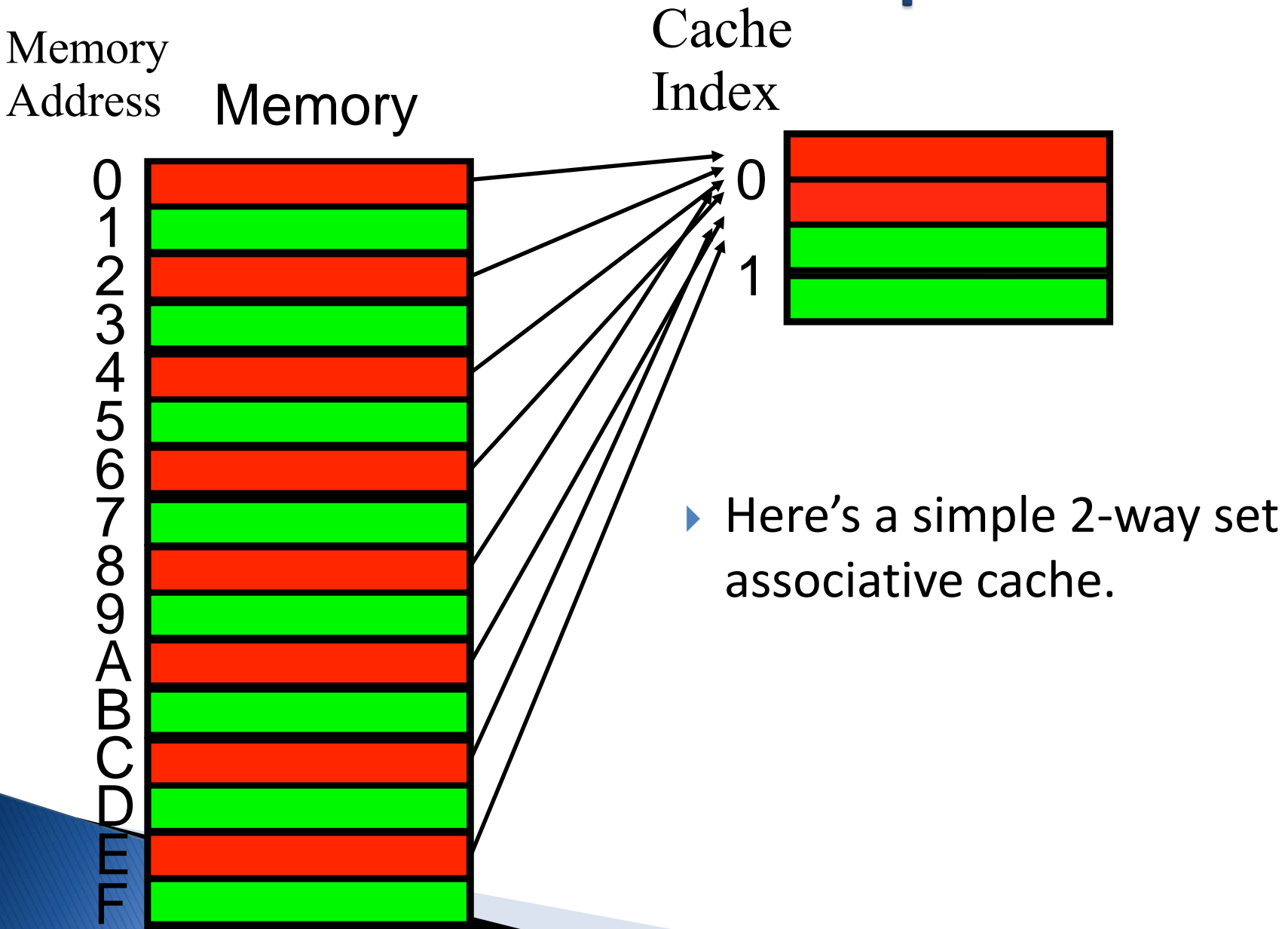
Fully Associative Cache (3/3)

- ▶ Benefit of Fully Assoc. Cache
 - No Conflict Misses (since data can go anywhere)
- ▶ Drawbacks of Fully Assoc. Cache
 - Need hardware comparator for every single entry: if we have a 64KB of data in cache with 4B entries, we need 16K comparators: infeasibly high cost

N-Way Set Associative Cache (1/3)

- ▶ Memory address fields:
 - **Tag**: same as before
 - **Offset**: same as before
 - **Index**: points us to the correct “row” (called a **SET** in this case)
- ▶ So what’s the difference?
 - each set contains multiple blocks
 - once we’ve found correct set, must compare with all tags in that set to find our data
 - Hybrid of direct-mapped and fully associative

Associative Cache Example

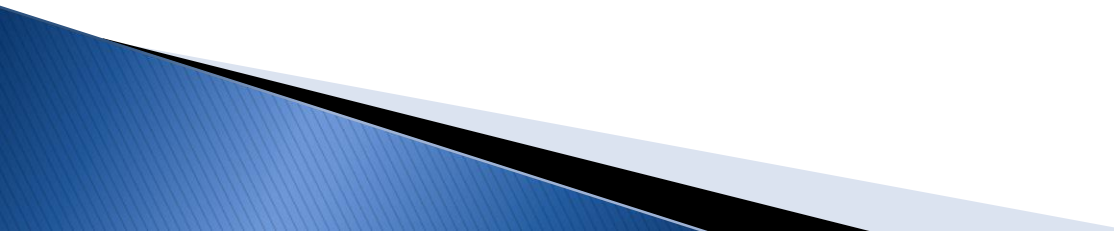


N-Way Set Associative Cache (2/3)

▶ Basic Idea

- cache is direct-mapped w/respect to sets
- each set is fully associative with N blocks in it

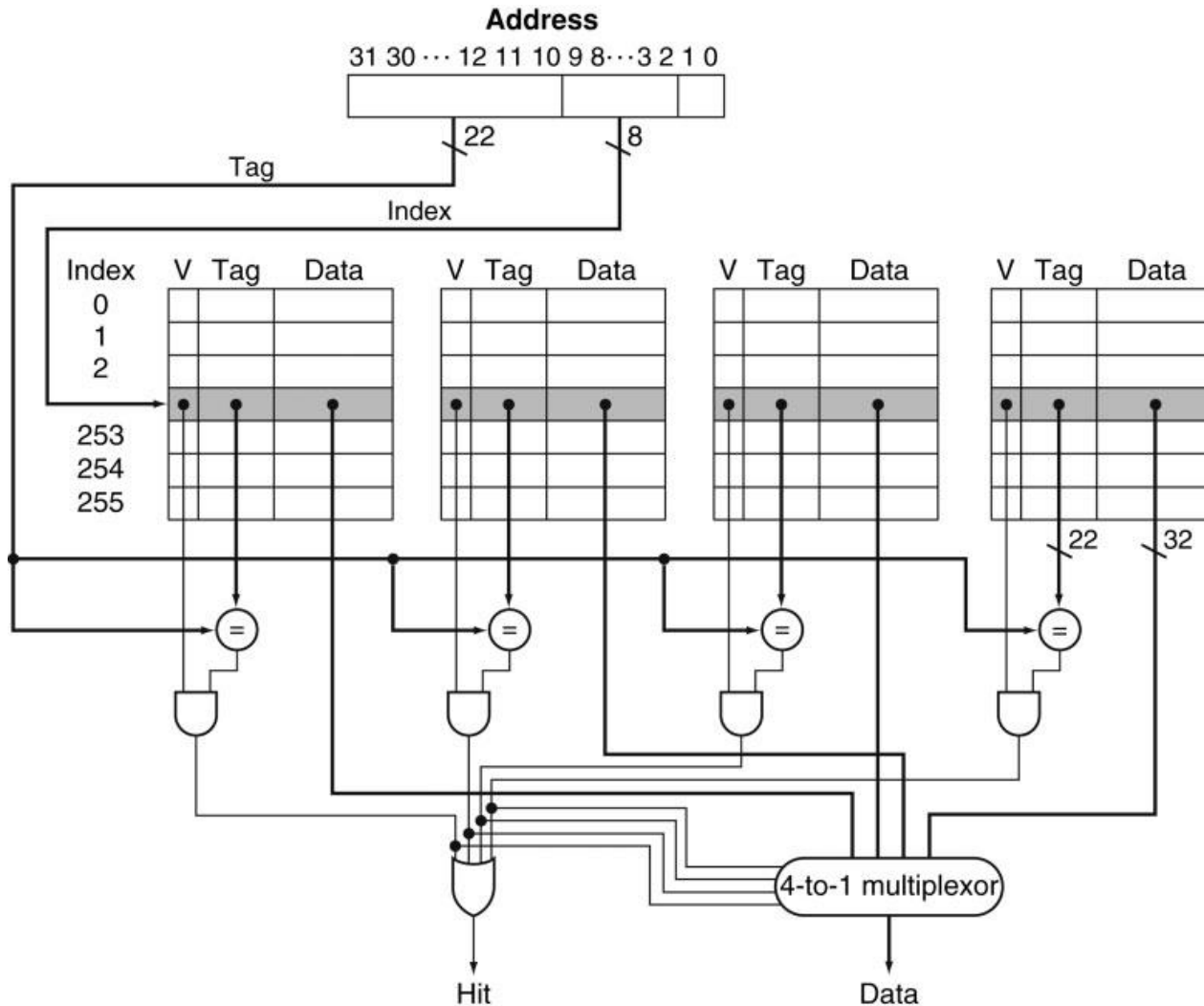
▶ Given memory address:

- Find correct set using Index value.
 - Compare Tag with all Tag values in the determined set.
 - If a match occurs, hit!, otherwise a miss.
 - Finally, use the offset field as usual to find the desired data within the block.
- 

N-Way Set Associative Cache (3/3)

- ▶ What's so great about this?
 - even a 2-way set assoc. cache avoids a lot of conflict misses
 - hardware cost isn't that bad: only need N comparators
- ▶ In fact, for a cache with M blocks,
 - it's **Direct-Mapped** if it's 1-way set assoc.
 - it's **Fully Assoc** if it's M-way set assoc.
 - so these two are just special cases of the more general set associative design

4-Way Set Associative Cache Circuit

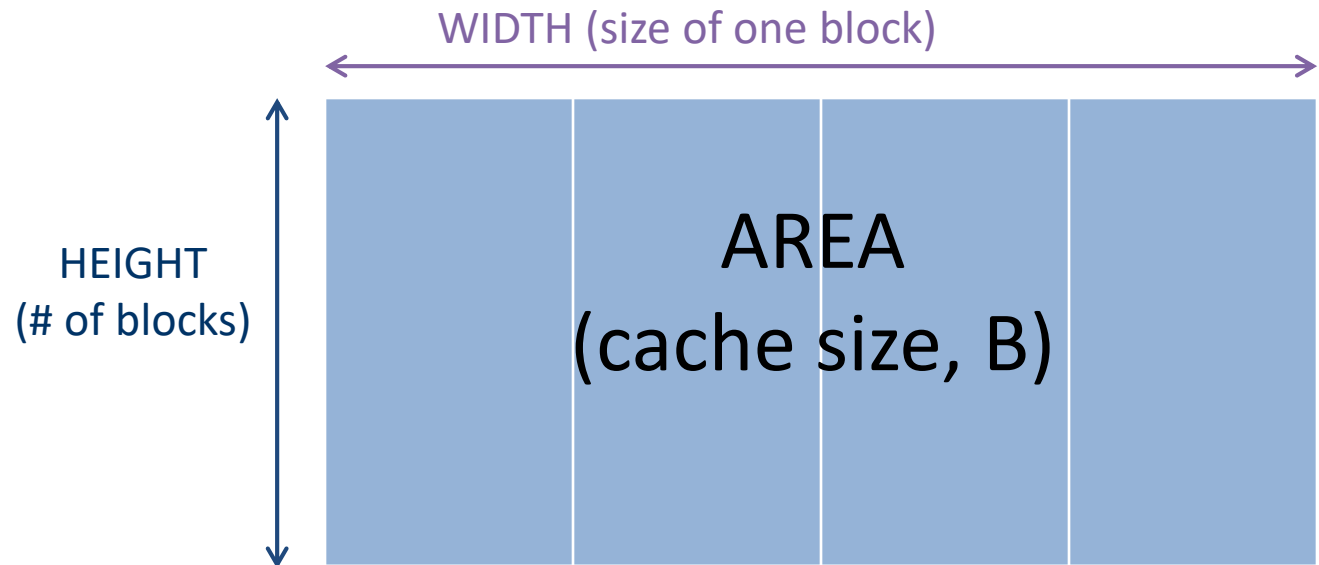




AREA (cache size, B)

= HEIGHT (# of blocks) * WIDTH (size of one block)

Tag	Index	Offset
-----	-------	--------



Block Replacement Policy

▶ Direct-Mapped Cache

- index completely specifies position which position a block can go in on a miss

▶ N-Way Set Assoc.

- index specifies a set, but block can occupy any position within the set on a miss

▶ Fully Associative

- block can be written into any position

▶ Question: if we have the choice, where should we write an incoming block?

- If there are any locations with valid bit off (empty), then usually write the new block into the first one.
- If all possible locations already have a valid block, we must pick a **replacement policy**: rule by which we determine which block gets “cached out” on a miss.

Block Replacement Policy: LRU

- ▶ LRU (Least Recently Used)
 - Idea: cache out block which has been accessed (read or write) least recently
 - Pro: **temporal locality** → recent past use implies likely future use: in fact, this is a very effective policy
 - Con: with 2-way set assoc, easy to keep track (one LRU bit); with 4-way or greater, requires complicated hardware and much time to keep track of this

Block Replacement Example

- ▶ We have a 2-way set associative cache with a four word total capacity and one word blocks. We perform the following word address accesses (ignore bytes for this problem):

0, 2, 0, 1, 4, 0, 2, 3, 5, 4

- ▶ How many hits and how many misses will there be for the LRU block replacement policy?

Block Replacement Example: LRU

0: miss, bring into set 0 (blk 0)

2: miss, bring into set 0 (blk 1)

Addresses 0, 2, 0, 1, 4, 0, ...

0: hit

1: miss, bring into set 1 (blk 0)

4: miss, bring into set 0 (blk 1, replace 2)

0: hit

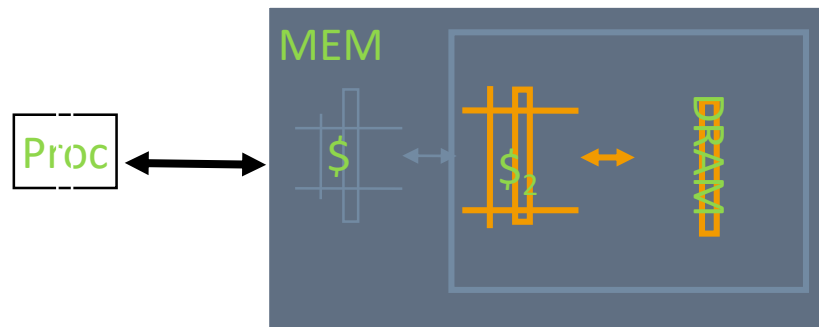
	blk 0	blk 1
set 0	0	lru
set 1		
set 0	lru 0	lru 2
set 1		
set 0	lru 0	lru 2
set 1		
set 0	0	lru 2
set 1	1	lru
set 0	lru 0	lru 4
set 1	1	lru
set 0	0	4 lru
set 1	1	lru

Big Idea

- ▶ How to choose between associativity, block size, replacement & write policy?
- ▶ Design against a performance model
 - Minimize: Average Memory Access Time
 - $\text{= Hit Time} + \text{Miss Penalty} \times \text{Miss Rate}$
 - Influenced by technology & program behavior
- ▶ Create the illusion of a memory that is large, cheap, and fast - on average
- ▶ How can we improve miss penalty?

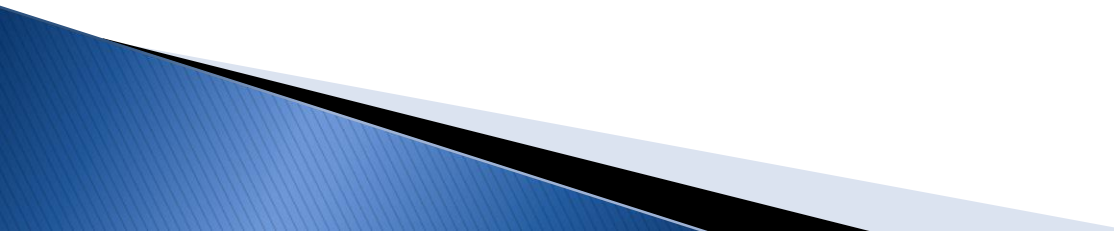
Improving Miss Penalty

- ▶ When caches first became popular, Miss Penalty
~ 10 processor clock cycles
- ▶ Today 2400 MHz Processor (0.4 ns per clock cycle) and 80 ns to go to DRAM
~ 200 processor clock cycles!

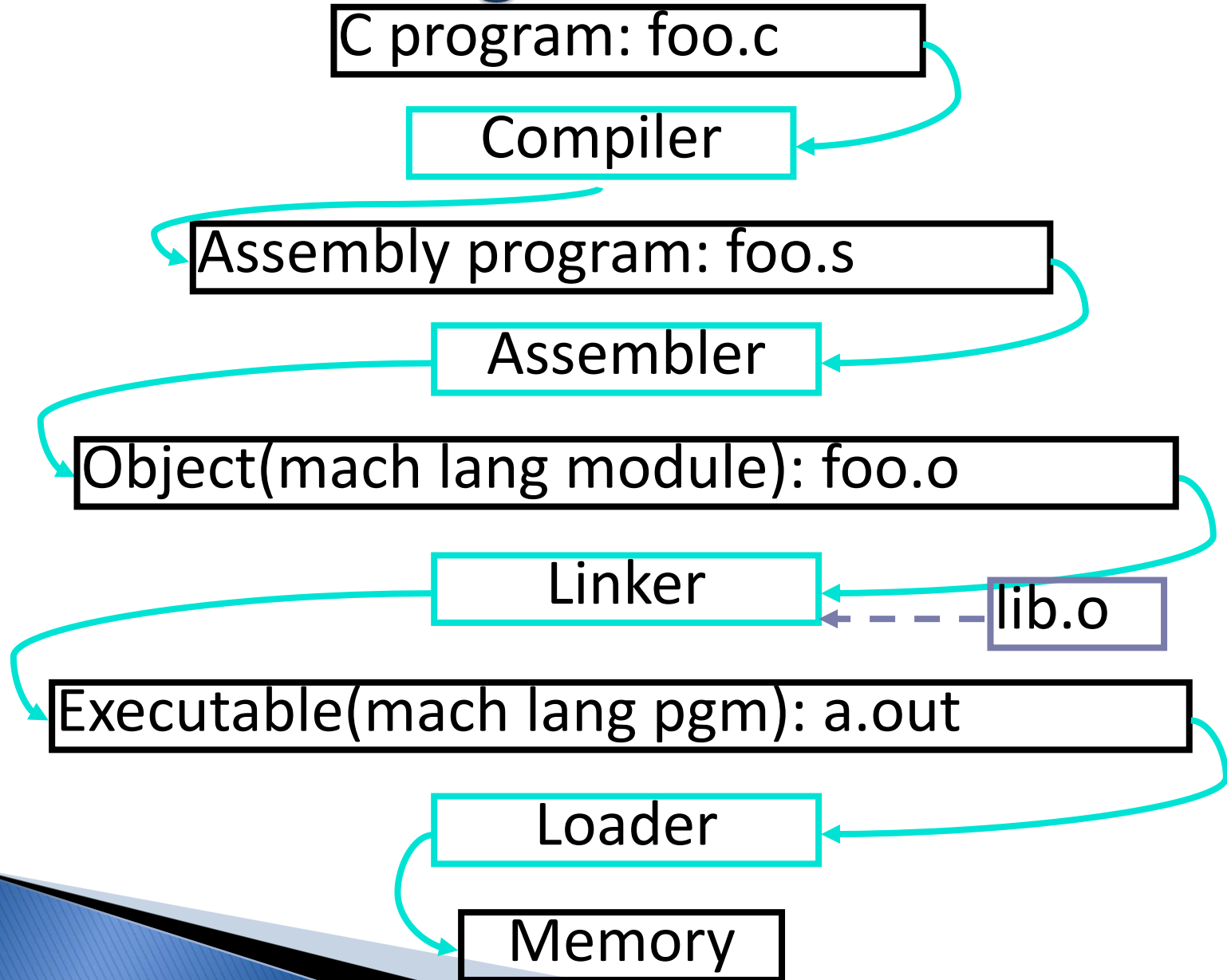


**Solution: another cache between
memory and the processor cache:
Second Level (L2) Cache**

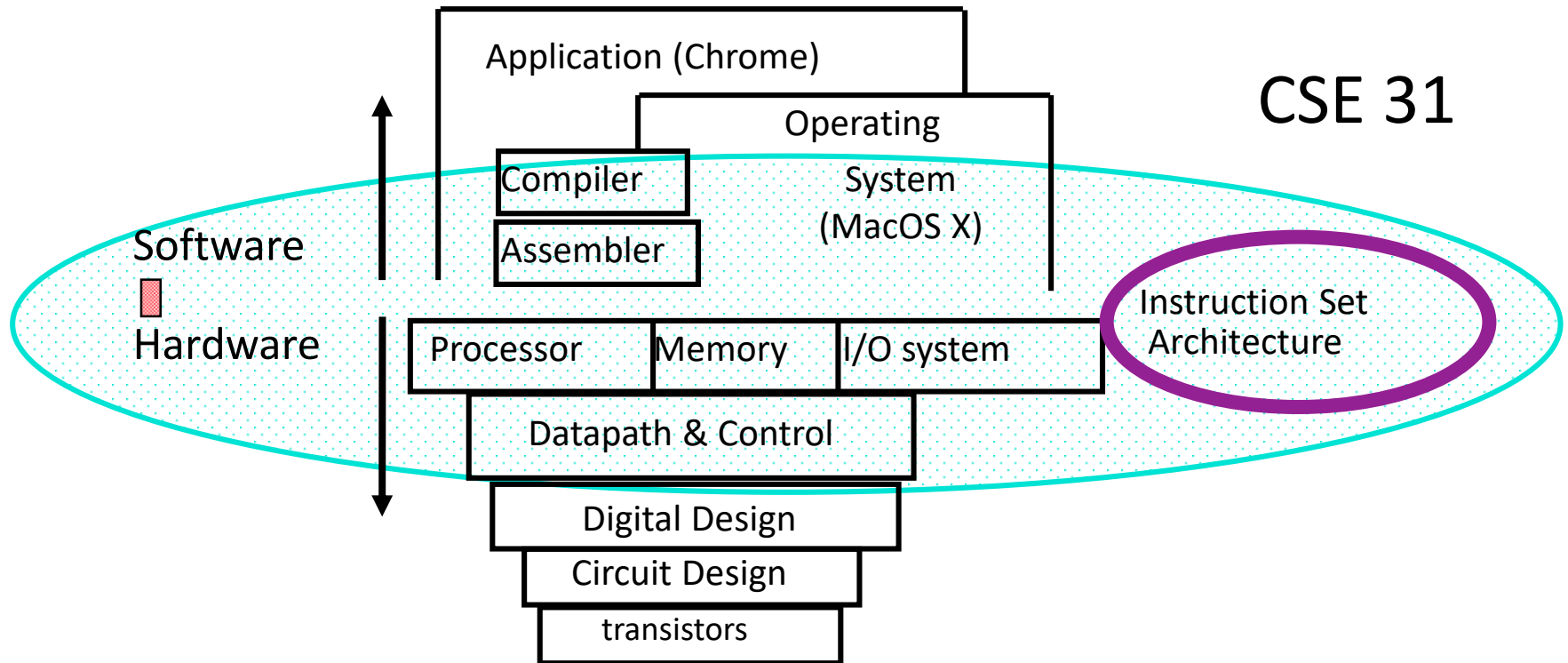
Summary

- ▶ Principle of Locality for Computer Memory
 - ▶ Hierarchy of Memories (speed/size/cost per bit) to Exploit Locality
 - ▶ Cache – copy of data lower level in memory hierarchy
 - ▶ Direct Mapped to find block in cache using Tag field and Valid bit for Hit
 - ▶ Larger caches reduce Miss rate via Temporal and Spatial Locality, but can increase Hit time
 - ▶ Larger blocks to reduces Miss rate via Spatial Locality, but increase Miss penalty
 - ▶ AMAT helps balance Hit time, Miss rate, Miss penalty
- 

Review on Program Process



What are “Machine Structures”?



Coordination of many **levels of abstraction**

ISA is an important abstraction level:
contract between HW & SW

Below the Program

- High-level language program (in C)

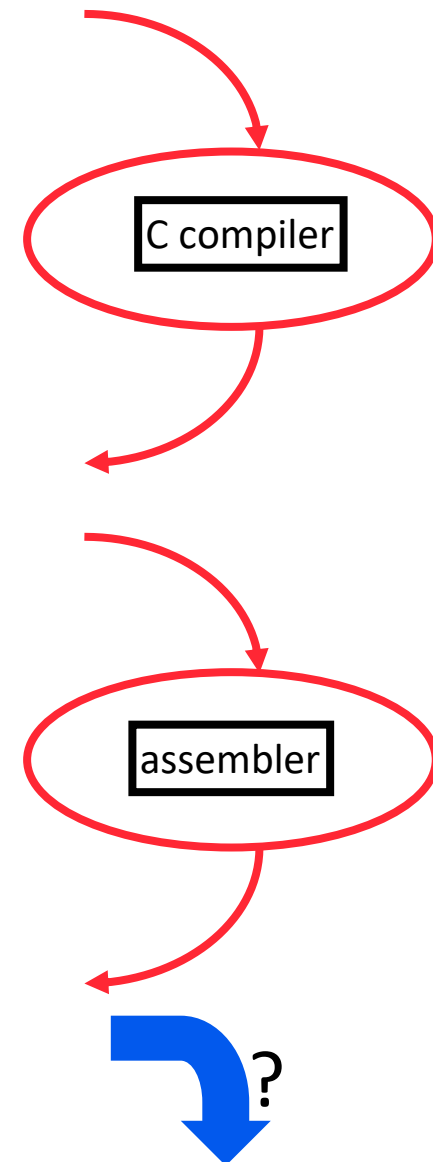
```
swap  int v[], int k){  
    int temp;  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

- Assembly language program (for MIPS)

```
swap: sll    $2, $5, 2  
      add    $2, $4, $2  
      lw     $15, 0($2)  
      lw     $16, 4($2)  
      sw     $16, 0($2)  
      sw     $15, 4($2)  
      jr     $31
```

- Machine (object) code (for MIPS)

```
000000 00000 00101 0001000010000000  
000000 00100 00010 0001000000100000 . . .
```



Synchronous Digital Systems

The hardware of a processor, such as the MIPS, is an example of a Synchronous Digital System

Synchronous:

- Means all operations are coordinated by a central clock.
 - It keeps the “heartbeat” of the system!

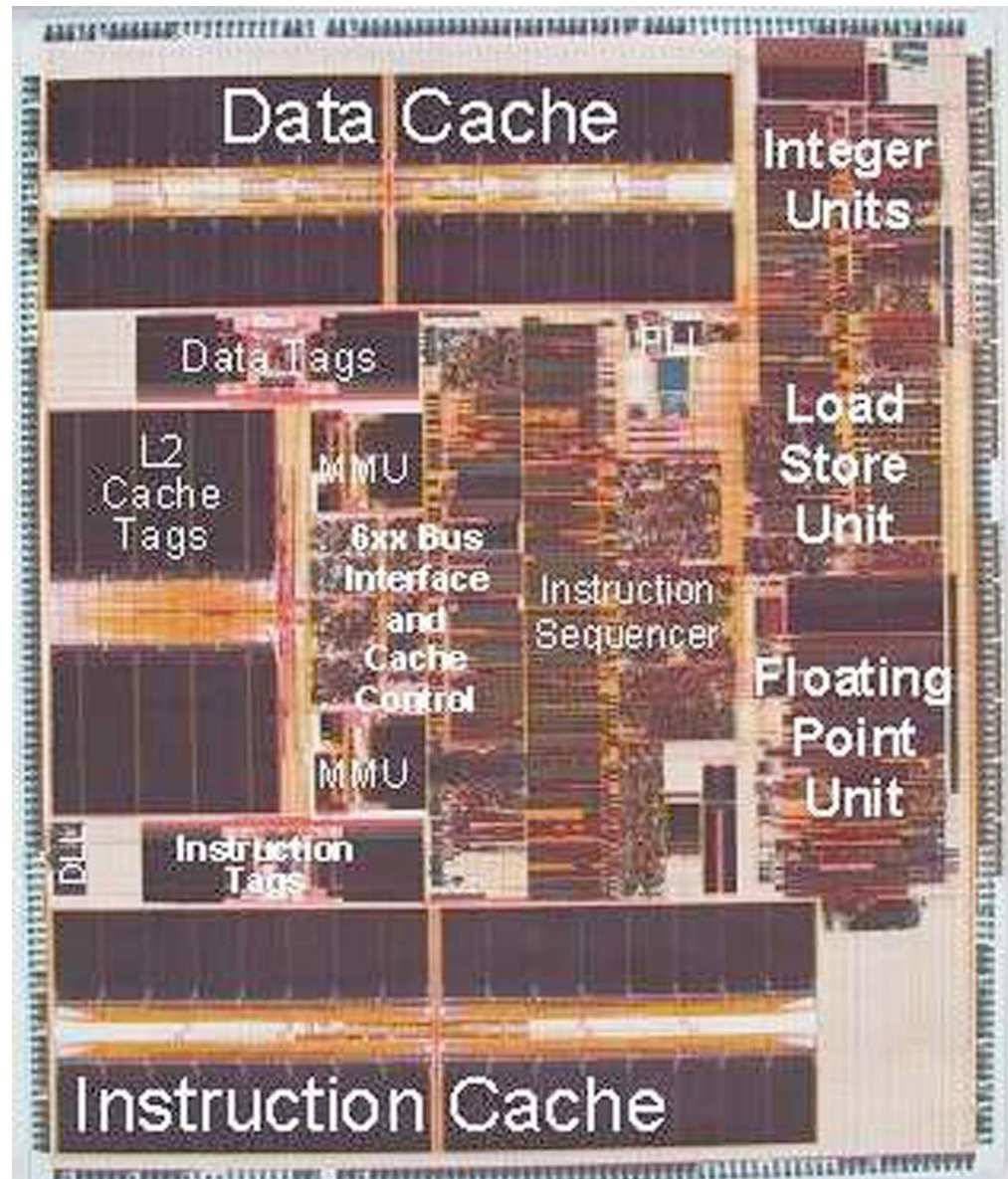
Digital:

- Mean all values are represented by discrete values
- Electrical signals are treated as 1's and 0's and grouped together to form words.

PowerPC Die Photograph

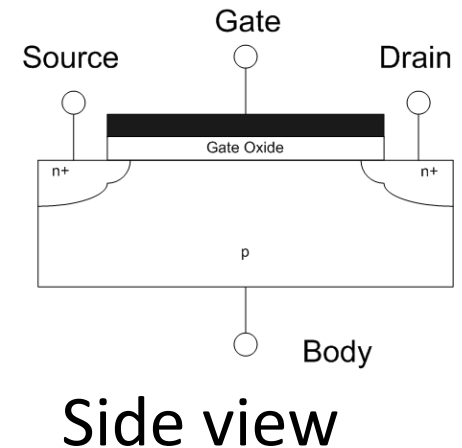
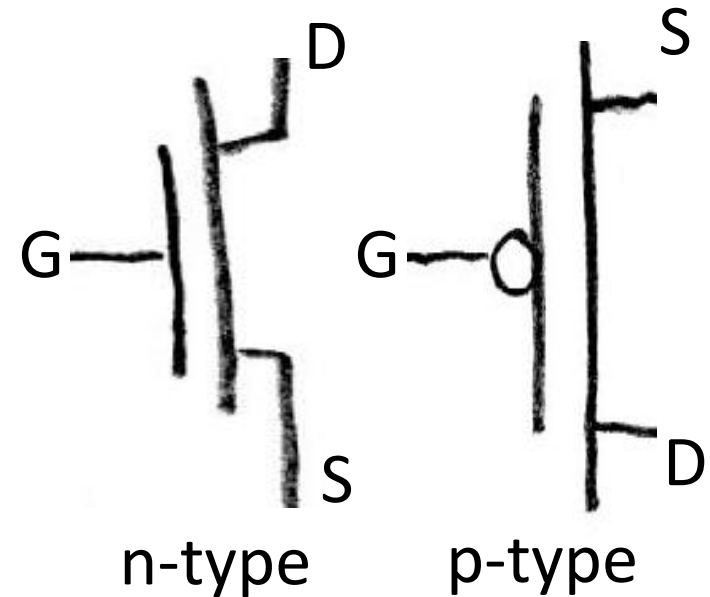


Let's look
closer...



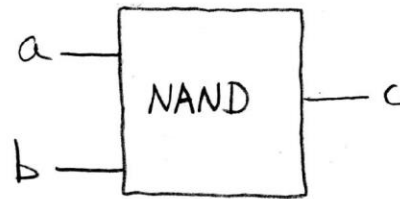
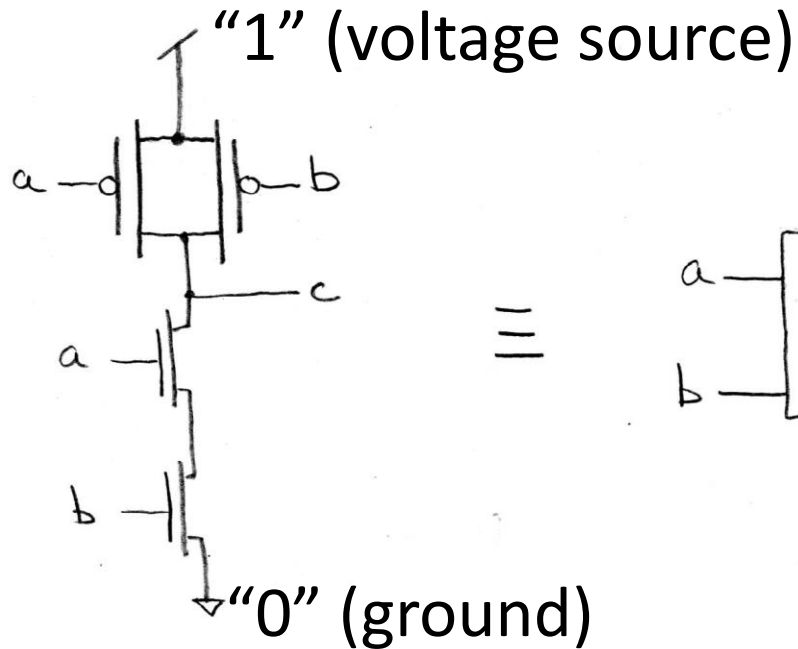
Transistors 101

- MOSFET
 - Metal-Oxide-Semiconductor Field-Effect Transistor
 - Come in two types:
 - n-type NMOSFET
 - p-type PMOSFET
- For n-type (p-type opposite)
 - If voltage not enough between G & S, transistor turns “off” (cut-off) and Drain-Source NOT connected
 - If the G & S voltage is high enough, transistor turns “on” (saturation) and Drain-Source ARE connected



Transistor Circuit Rep. vs. Block diagram

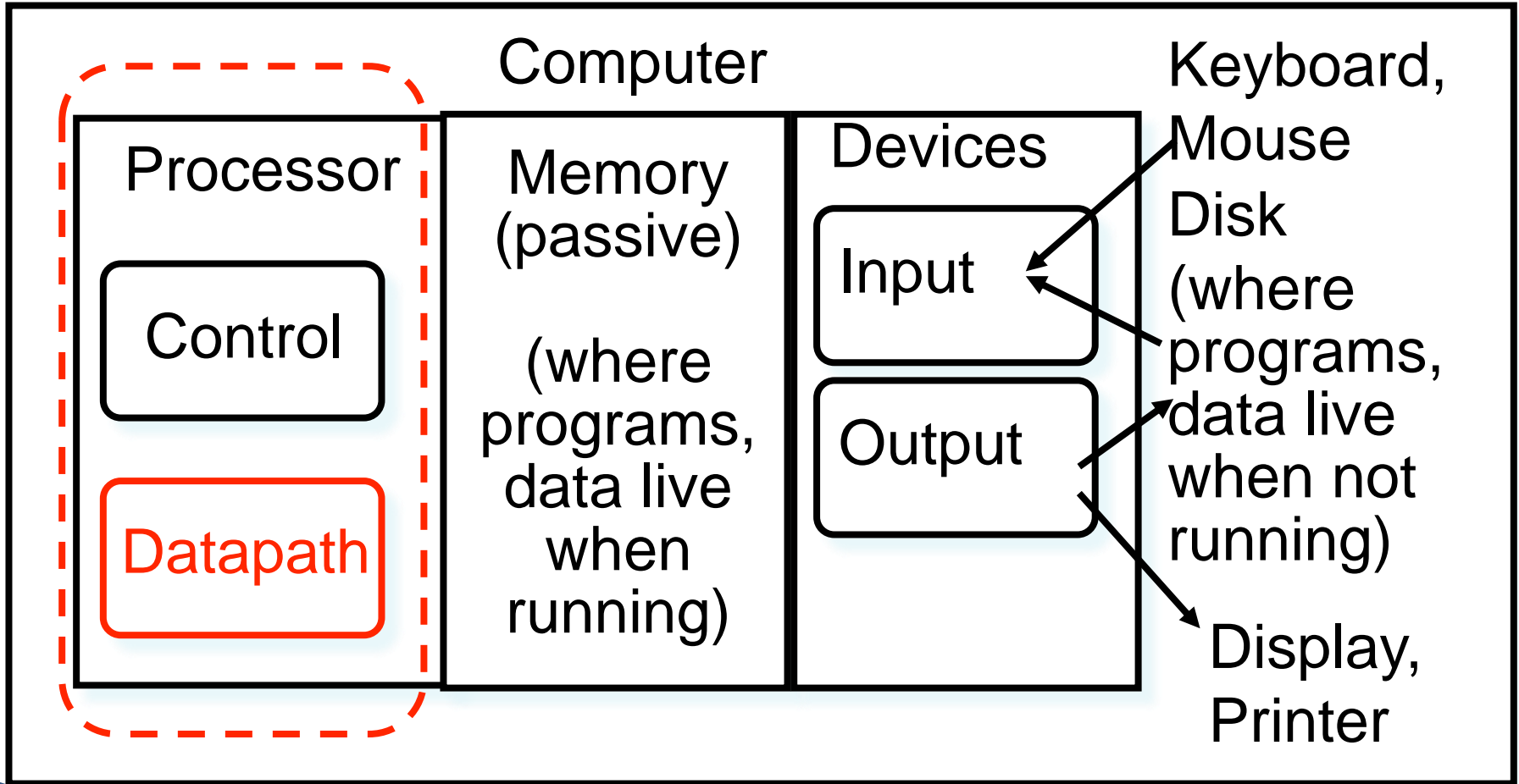
- Chips are composed of nothing but transistors and wires.
- Small groups of transistors form useful building blocks.



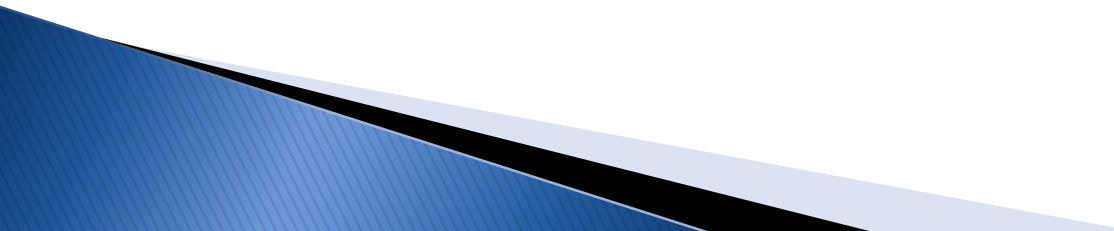
a	b	c
0	0	1
0	1	1
1	0	1
1	1	0

- Block are organized in a hierarchy to build higher-level blocks: ex: adders.

Five Components of a Computer



The CPU

- ▶ **Processor (CPU)**: the active part of the computer, which does all the work (data manipulation and decision-making)
 - ▶ **Datapath**: portion of the processor which contains hardware necessary to perform operations required by the processor (the brawn)
 - ▶ **Control**: portion of the processor (also in hardware) which tells the datapath what needs to be done (the brain)
- 

Stages of the Datapath

- ▶ Problem: a single, atomic block which “executes an instruction” (performs all necessary operations beginning with fetching the instruction) would be too bulky and inefficient
- ▶ Solution: break up the process of “executing an instruction” into **stages**, and then connect the stages to create the whole **datapath**
 - smaller stages are easier to design
 - easy to optimize (change) one stage without touching the others

Stages of the Datapath (1/5)

- ▶ There is a wide variety of MIPS instructions: so what general steps do they have in common?
- ▶ Stage 1: **Instruction Fetch**
 - no matter what the instruction type is, the 32-bit instruction word must first be fetched from memory (the cache-memory hierarchy)
 - also, this is where we **Increment PC**
(that is, $PC = PC + 4$, to point to the next instruction: byte addressing so + 4)

Stages of the Datapath (2/5)

► Stage 2: Instruction Decode

- upon fetching the instruction, we next gather data from the fields (decode all necessary instruction data)
- first, read the `opcode` to determine instruction type and field lengths
- second, read in data from all necessary registers
 - for `add`, read two registers (R-format)
 - for `addi`, read one register (I-format)
 - for `jal`, no reads necessary (J-format)

Stages of the Datapath (3/5)

► Stage 3: ALU (Arithmetic-Logic Unit)

- the real work of most instructions is done here: arithmetic (+, -, *, /), shifting, logic (&, |), comparisons (slt)
- what about loads and stores, do they need this stage?
 - lw \$t0, 40(\$t1)
 - the address we are accessing in memory = the value in \$t1 PLUS the value 40
 - so we do this addition in this stage

Stages of the Datapath (4/5)

▶ Stage 4: Memory Access

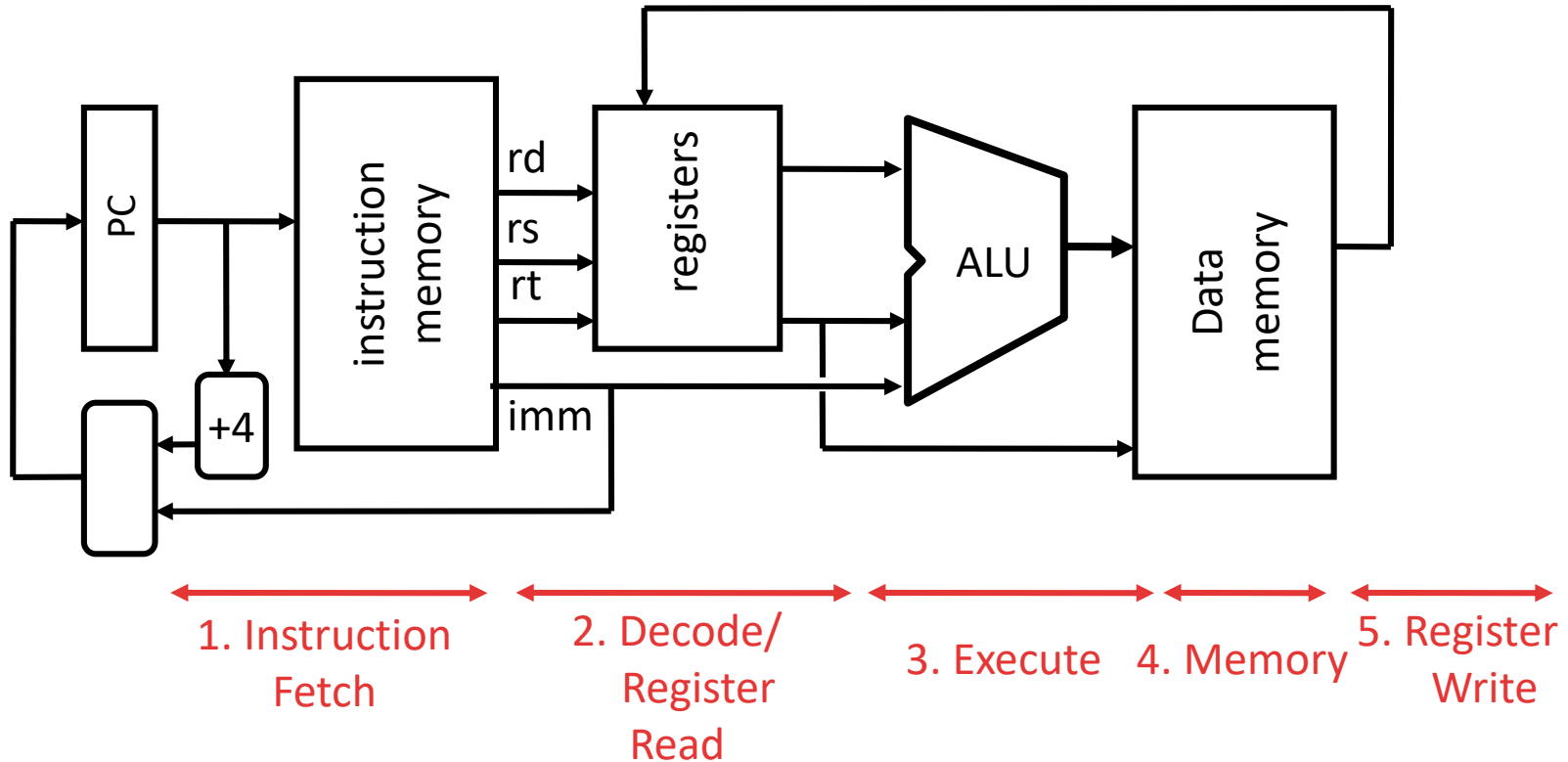
- actually only the load and store instructions do anything during this stage; the others remain idle during this stage or skip it all together
- since these instructions have a unique step, we need this extra stage to account for them
- as a result of the cache system, this stage is expected to be fast

Stages of the Datapath (5/5)

► Stage 5: Register Write

- most instructions write the result of some computation into a register
- examples: arithmetic, logical, shifts, loads, slt
- what about stores, branches, jumps?
 - don't write anything into a register at the end
 - these remain idle during this fifth stage or skip it all together

Generic Diagram of Datapath



Summary

- ▶ CPU design involves Datapath, Control
 - Datapath in MIPS involves 5 CPU stages
 1. Instruction Fetch
 2. Instruction Decode & Register Read
 3. ALU (Execute)
 4. Memory
 5. Register Write