CSCI 210: Computer Architecture
Lecture 37: Last Class!

Cynthia Taylor

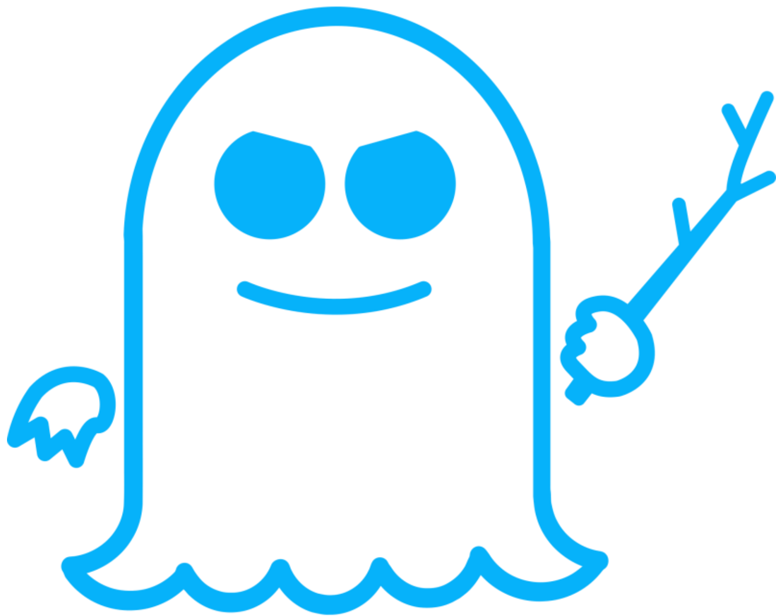
Oberlin College

May 10th, 2024

CS History: Cache Side Channel Attacks

- Flush + Reload (2014), Evict + Reload (2015)
- Depends on shared memory (e.g., a shared crypto library) being in the cache
- Attacker causes the cache to be emptied, either by calling flush or by filling it with new data
- Victim uses shared memory to do something (e.g., encrypt a message)
- Attacker references different bytes of shared memory and measures how long it takes: if it's fast, byte is in the cache, which means the victim referenced it. If it's slow, not in the cache, so victim didn't reference it
- This lets the attacker trace the code the victim is running
- Researchers used it to extract private encryption keys, even across virtual machines
- Takes advantage of L3 cache being shared across cores!

CS History: Spectre & Meltdown (2018)



- Combines speculative execution with cache side channel attacks
- Create a program with a pattern of references that will cause the branch predictor to predict taken
- Put some code you don't have access to in the branch – it will be run by the branch predictor, but then rolled back
- Now the memory it referenced is in the cache!
- Use cache side channel attack to trace the code
- Meltdown does a similar thing, but accessing memory in the OS kernel
- Affects every Intel processor since 1995, also ARM processors, all OSes
- Fixes aren't cheap and tend to slow processors down a lot

Cache Final Project

- Double check that the skeleton code provided includes support for the command line flag --hit-time (or –h) for setting the cache hit time
  - If it doesn't apply the patch in my BlackBoard announcement

```
Options:
  -b, --block-size <BLOCK_SIZE>        Set the block size in bytes. Must be a power of two [default: 32]
  -a, --associativity <ASSOCIATIVITY>  Set the associativity. Must be a power of two [default: 1]
  -s, --size <SIZE>                    Set the cache size in kilobytes (kB). Must be a power of two [default: 32]
  -h, --hit-time <HIT_TIME>            Set the cache hit time [default: 4]
  -m, --miss-penalty <MISS_PENALTY>    Set the cache miss penalty [default: 34]
      --help                           Print help
  -V, --version                        Print version
```

Questions on Cache Final Project?

We have a 8 byte block size, direct-mapped, 16 kB cache. For a given byte address, to find the offset

A. Mod by 8

B. Mod by 16

C. Divide by 8

D. Divide by 16

E. None of the above

If we are simulating a cache and not actually accessing data, do we need the offset?

A. Yes, we need it to access the right cache block

B. No, we can ignore it entirely and use those bits as part of the index

C. No, but we still need to divide by the block size to get the block address

We have an 8 byte block size, direct-mapped, 16 kB cache. How many rows will this cache have?

A. 16

B. 2*1024

C. 8*1024

D. 16*1024

*Recall a kB is 1024 bytes

We have an 8 byte block size, direct-mapped, 16 kB cache. How can we find the index from an address?

A. Divide by 8

B. Divide by (16*1024)

C. Mod by 16*1024

D. Divide by 8, then mod by (2*1024)

E. None of the above

We have an 8 byte block size, 2-way set associative, 16 kB cache. How can we find the index from an address?

A. Divide by 8, then mod by 1024

B. Divide by 8, then mod by (2*1024)

C. Divide by 8, then mod by (2*2*1024)

D. Divide by 16, then mod by (2*1024)

E. None of the above

We have an 8 byte block size, direct mapped, 16 kB cache. How can we find the tag of an address?

A. Divide by 8

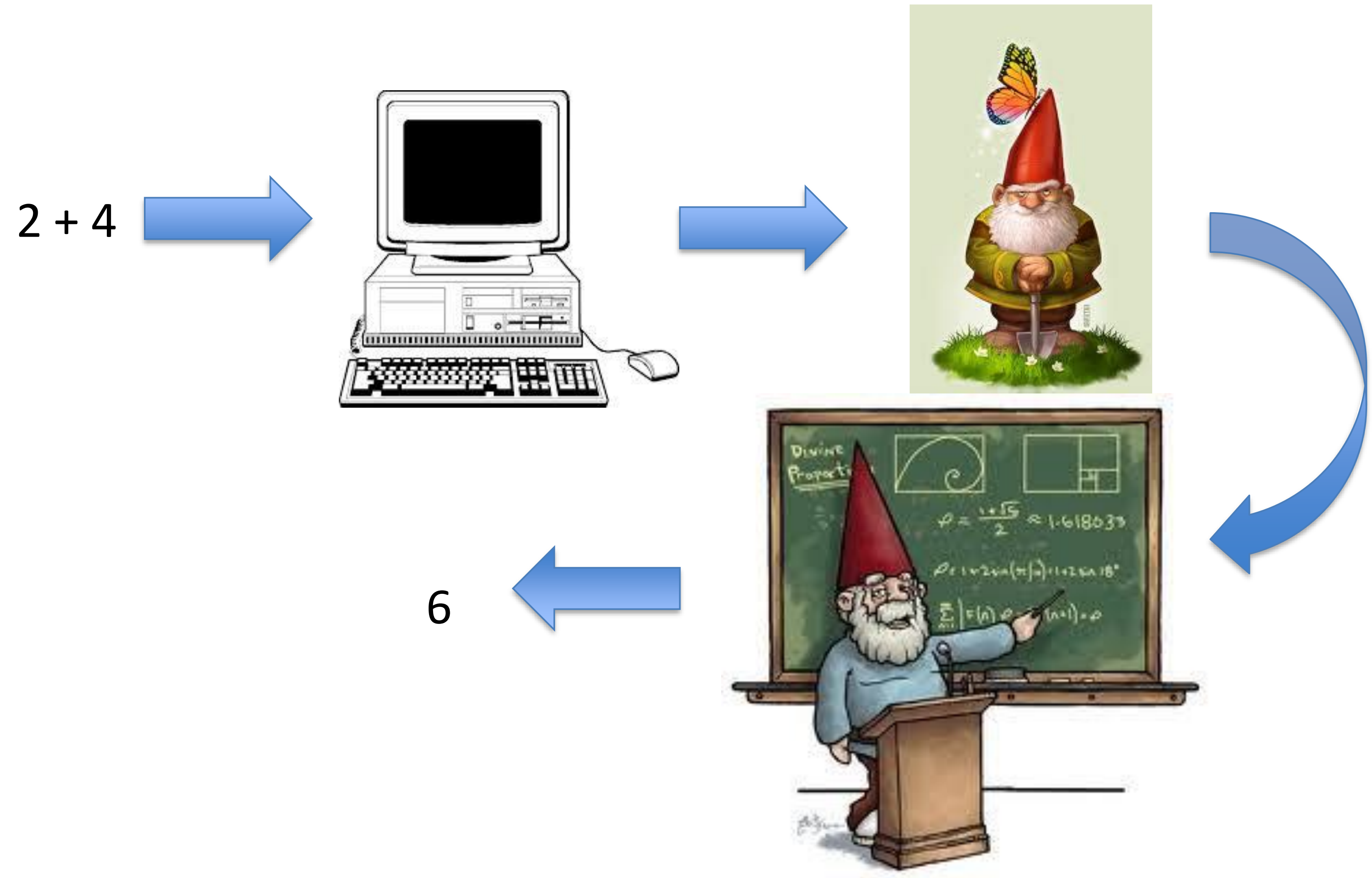
B. Divide by 8 * 1024

C. Divide by 16*1024

D. None of the above

Questions on Final Project?

Previous Conceptions of How Computers Work



Actually Assembly

High Level:

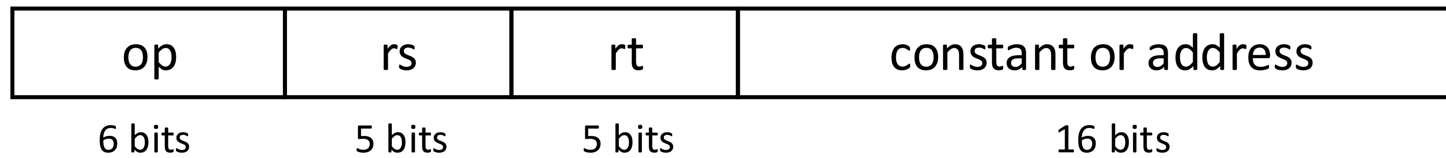
```
x = 2 + 4
```

Assembly (assuming we have a mem address for x in $s0):

```
li      $t1,  2
addi  $t1, $t1, 4
sw    $t1, 0($s0)
```

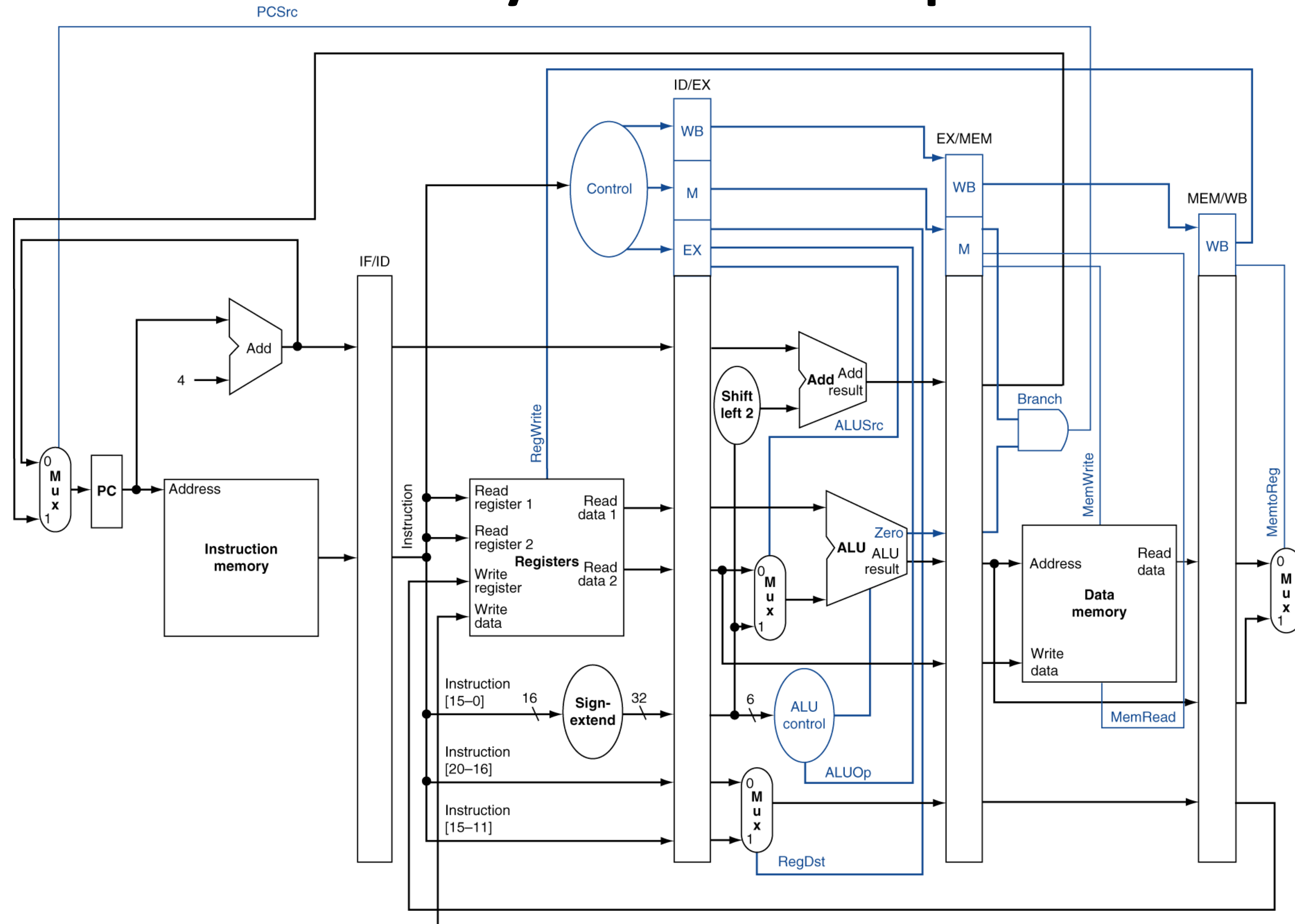
Actually Machine Instructions

```
addi $t1, $t1 5
```

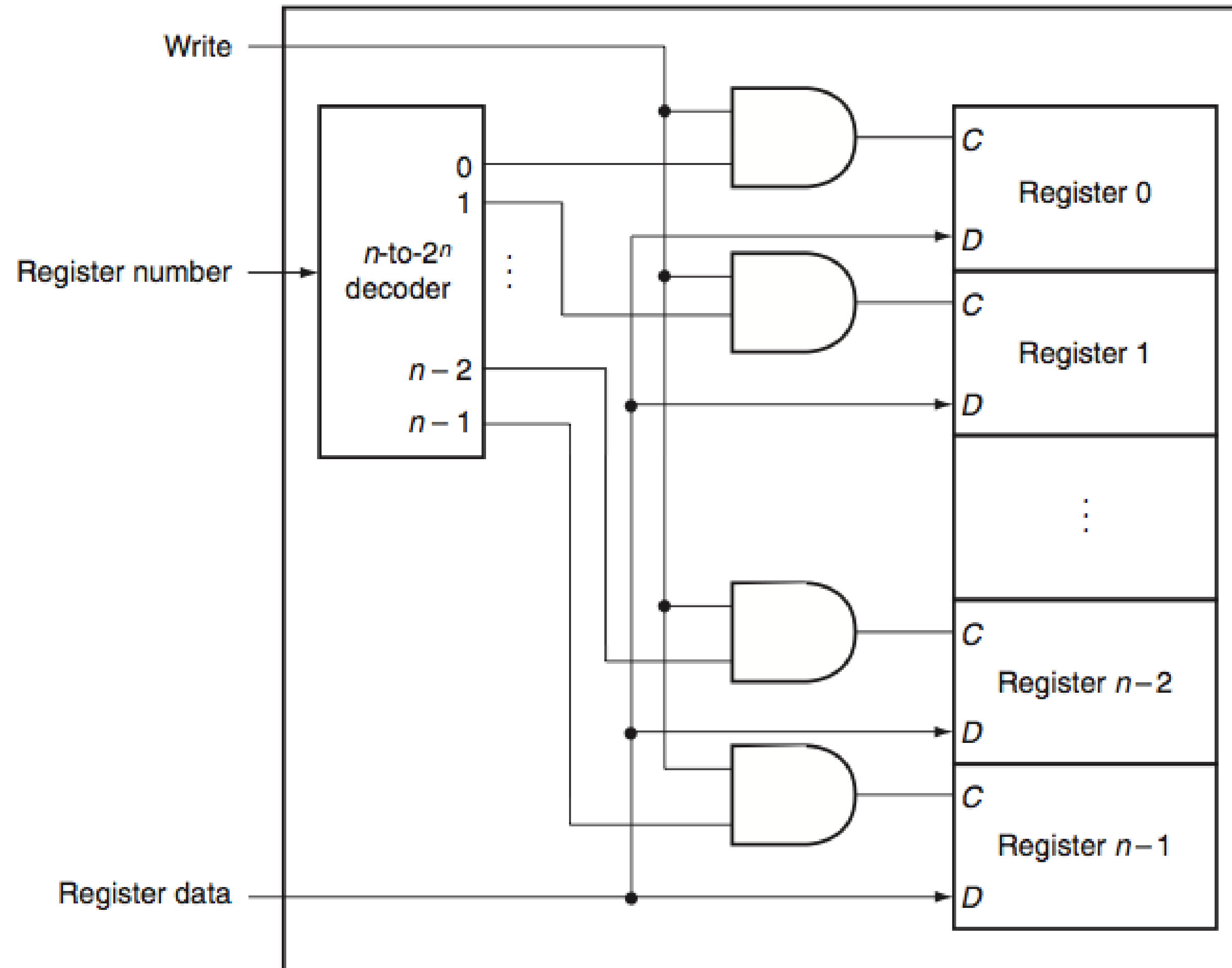



```
00100001001010010000000000001001
```

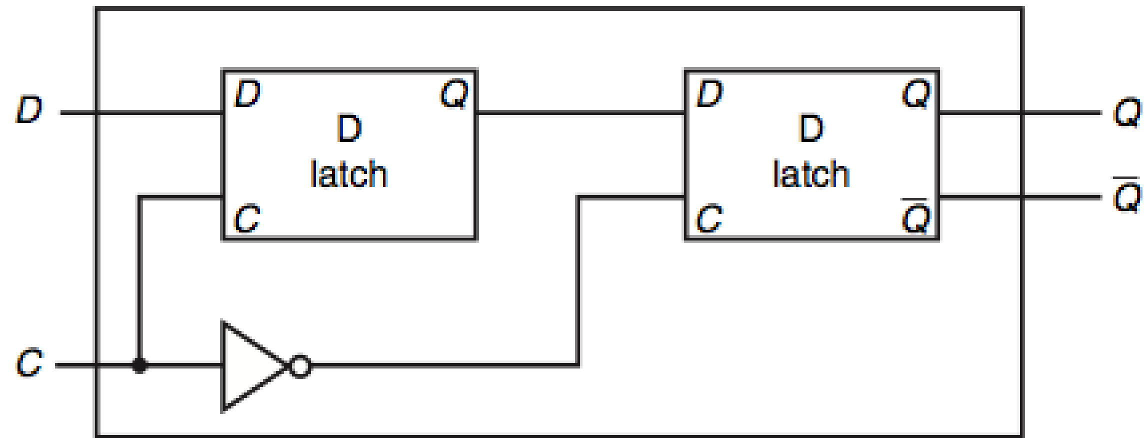
Actually The Datapath



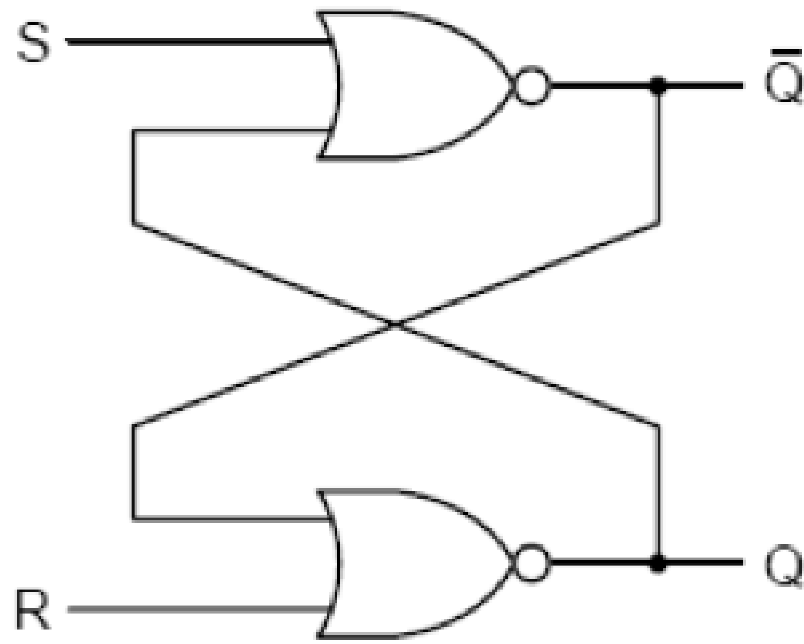
Actually Registers



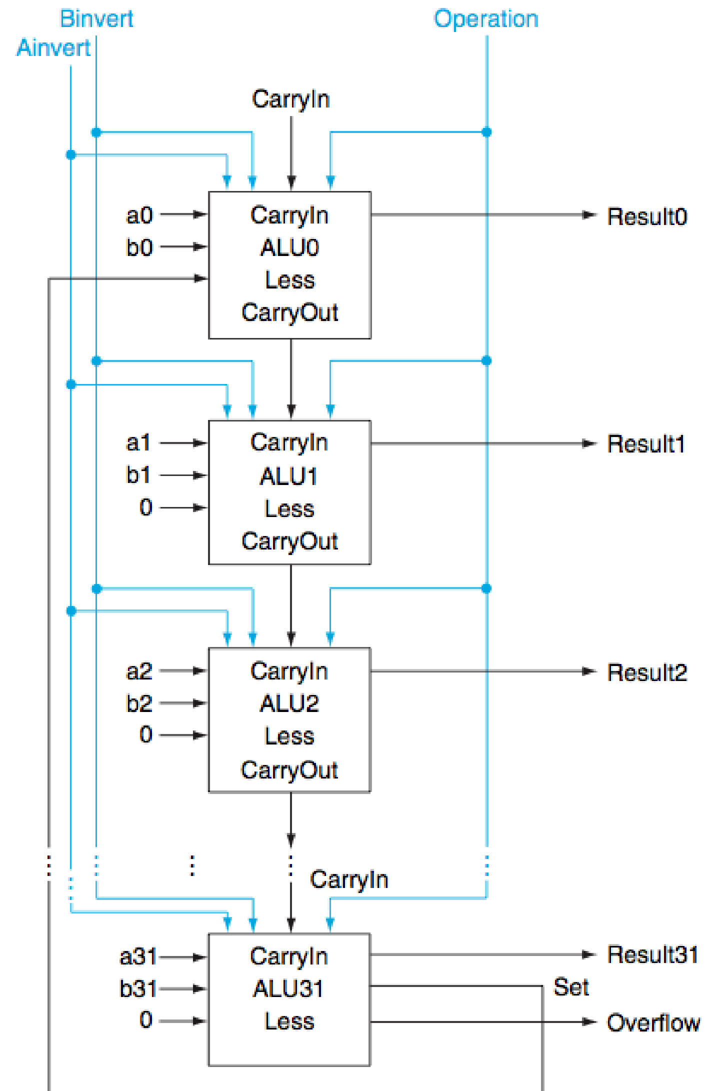
Actually Flip-flops



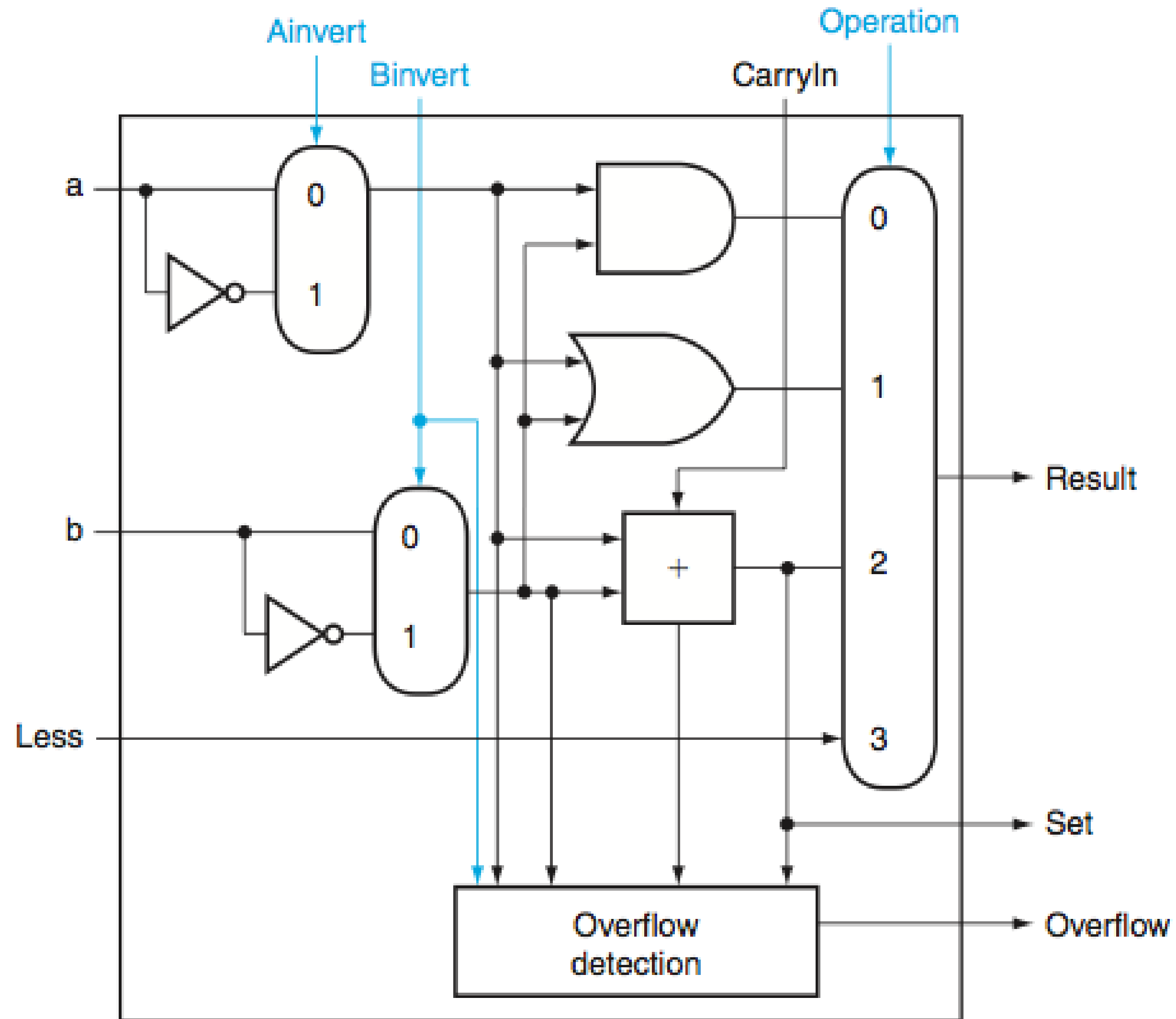
Actually Latches



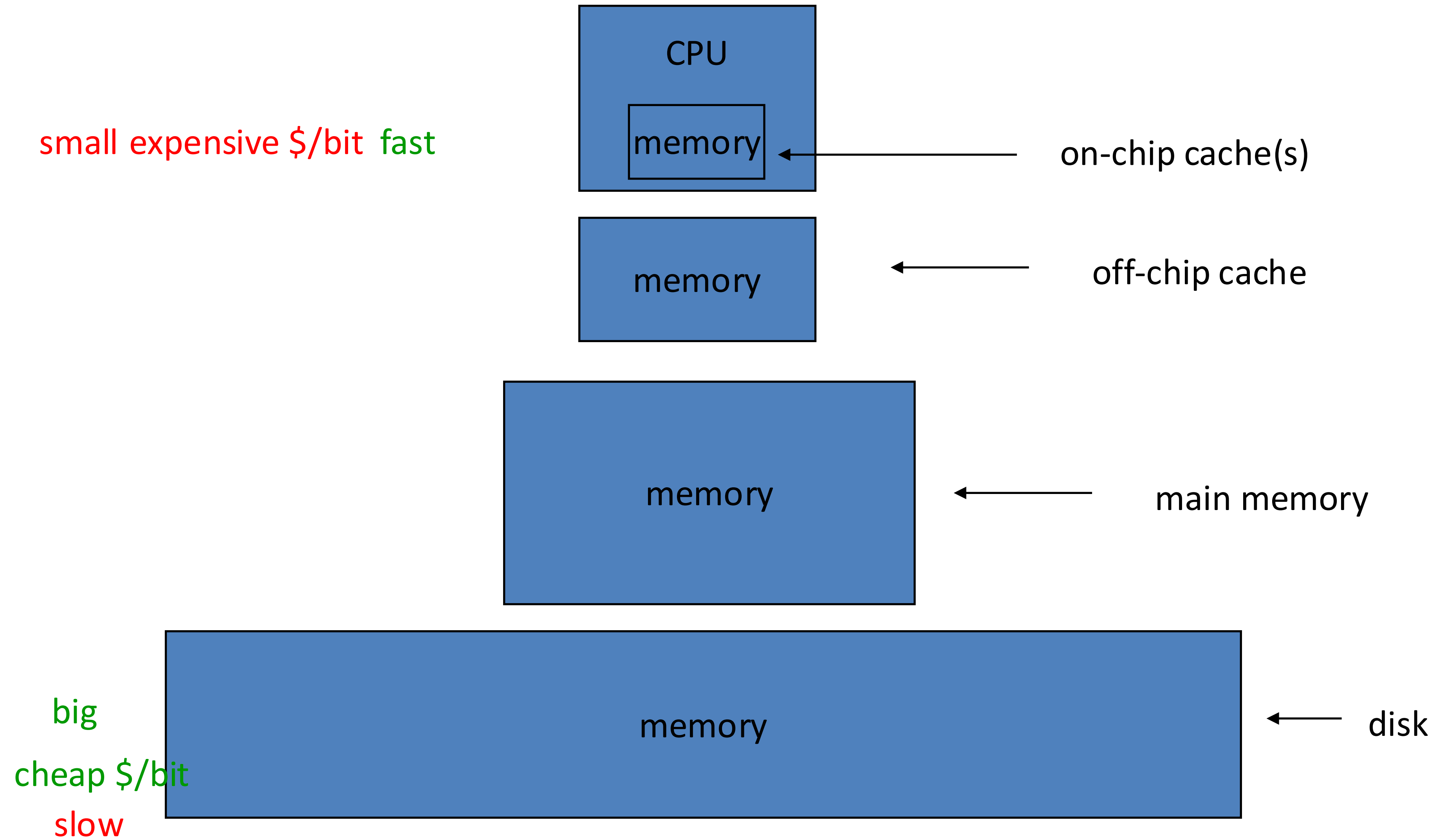
Actually the ALU



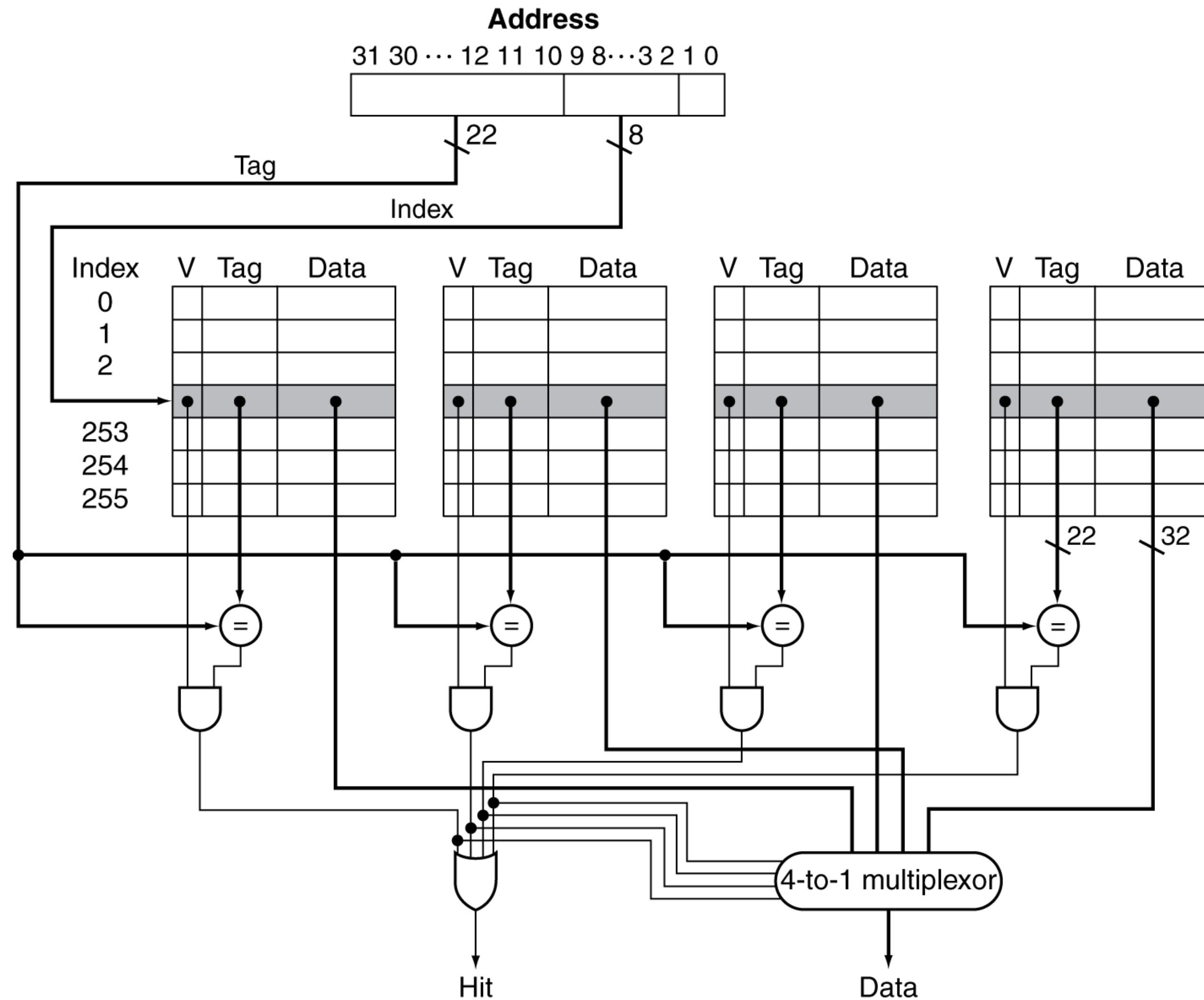
Actually the ALU



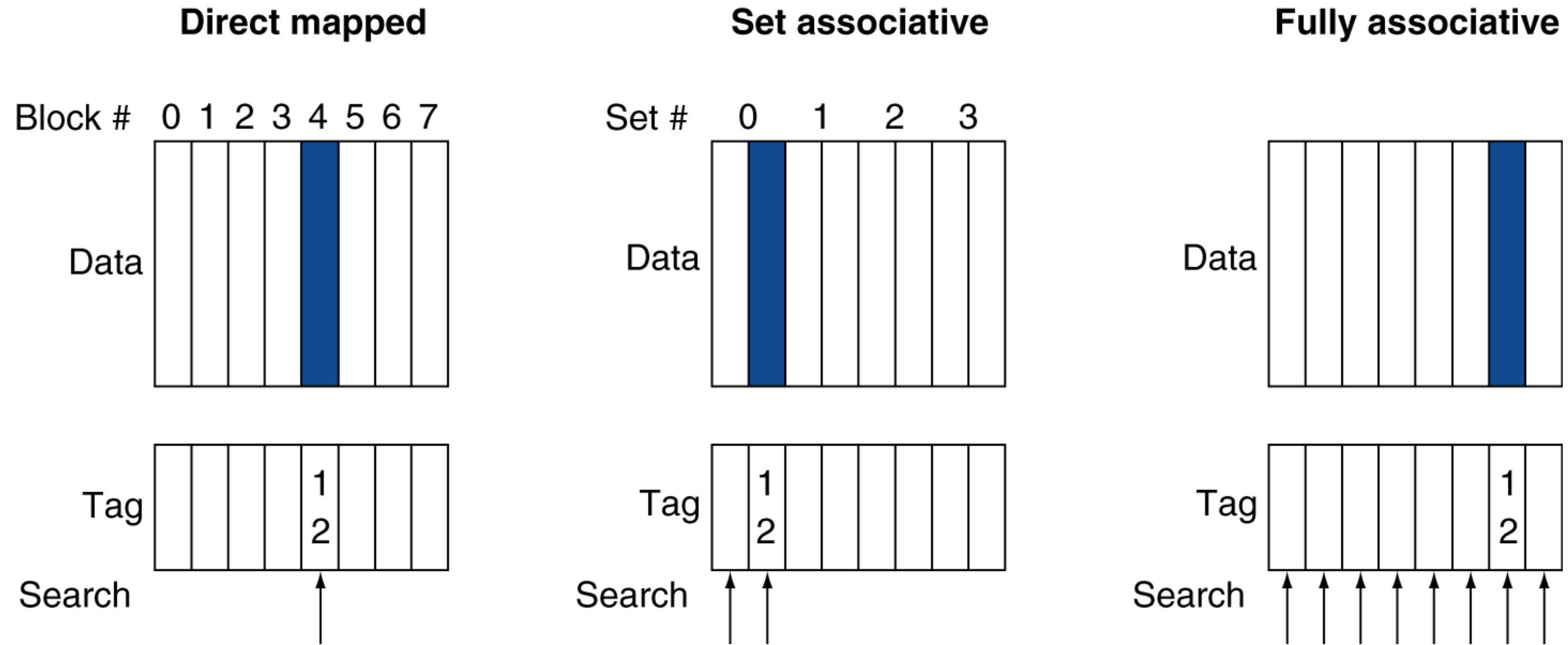
Actually Memory



Actually Caches



Actually LOTS of Caches



But wait, what about?

- Negative Numbers
- Floating Point
- All that other stuff . . .

Computers are Complicated

- But now, you know how they work. Kinda.

- I appreciate all the hard work you’ve done for this class.
- Have a great break!
- …and fill out course evals!