CS61C - Final Review Session

Hosted by Anokhi Shah and Varun Bharadwaj

Disclaimer

Disclaimer: Topics covered in this review session may not be fully comprehensive in the representation of the topics covered in or of the difficulty of the exam. Content is pulled from pulled from previous semester materials and what is listed on the course website. Review Sessions presenters are presenting as members of HKN, regardless of relation to the course. Course staff is the source of truth for any differences in content.

Slides are posted on Ed

Topics

1. RISC-V Single Cycle Datapath
2. Pipelining/Hazards
3. Parallelism
4. Caches & Cache Coherency
5. Virtual Memory

This is not necessarily an exhaustive list of things to study! Check out the official details on Ed and on cs61c.org

RISC-V - Datapath

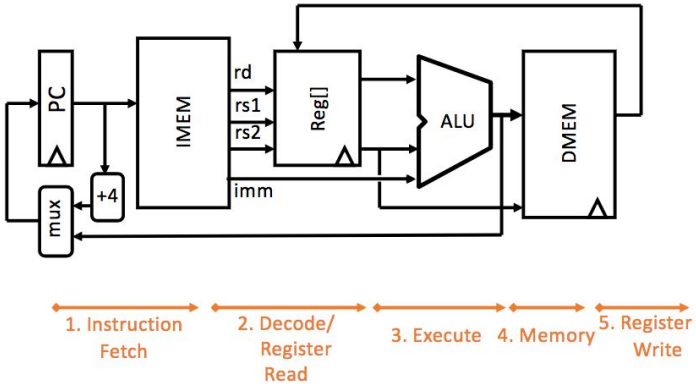
What do the the inventors of RISC do when they need to meet up?

They hold an assembly.

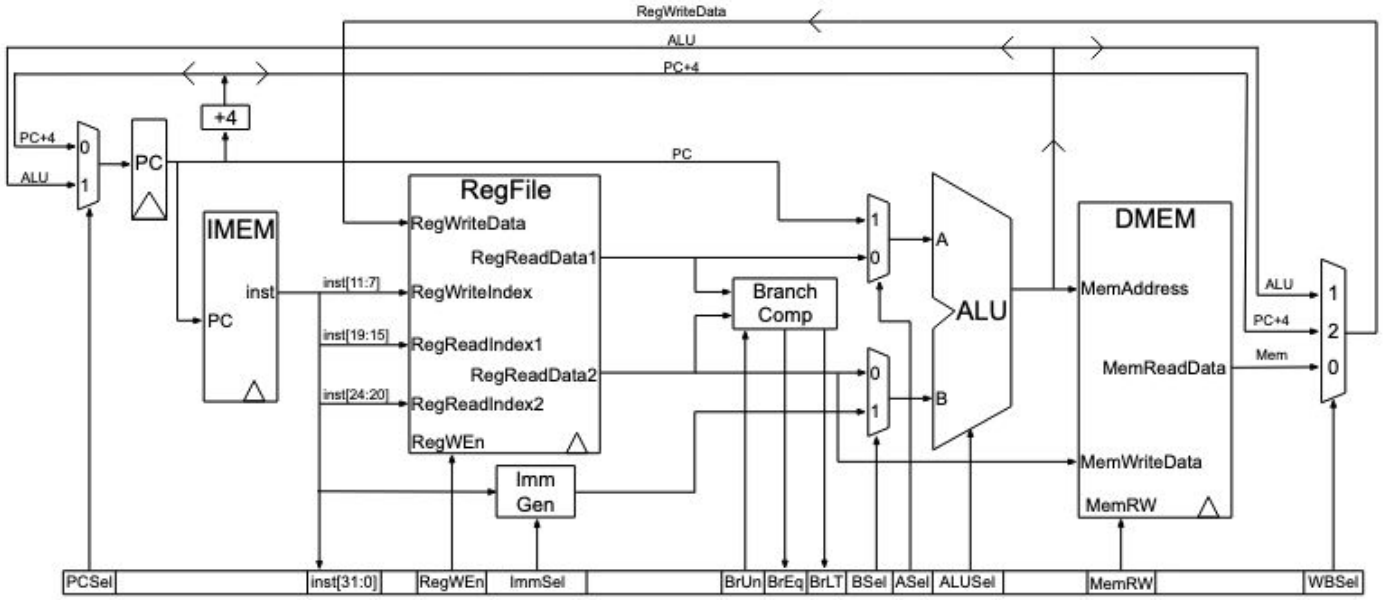
RISC-V Datapath

Five stages:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute (EX)
4. Memory Access (MEM)
5. Writeback (WB)

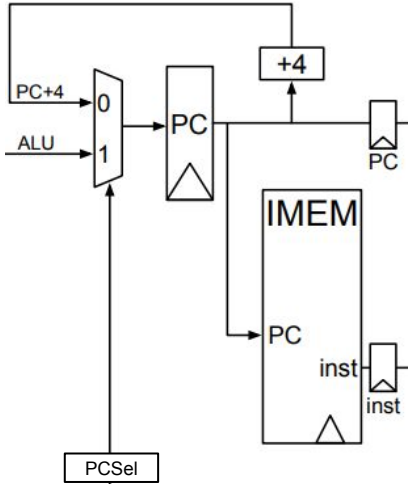


RISC-V Single Cycle Datapath



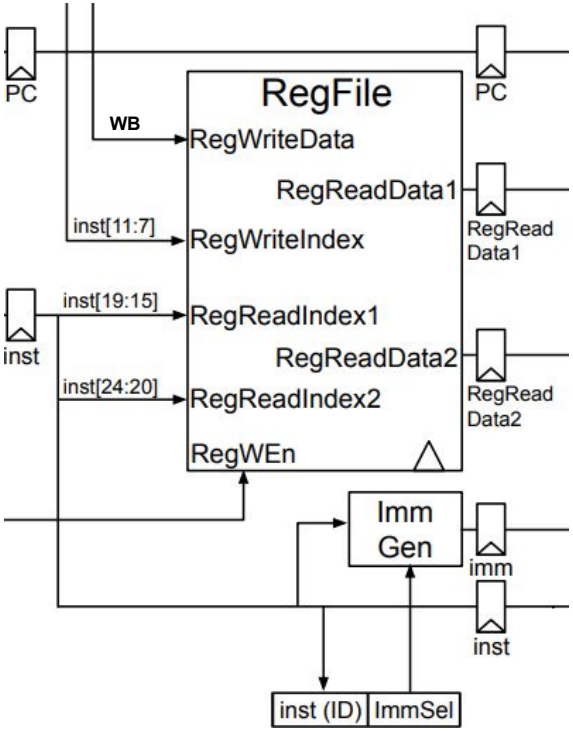
Instruction Fetch (IF)

1. Fetch the instruction from memory.
2. Increment PC by 4 (or change to new address in the case of a branch or jump).



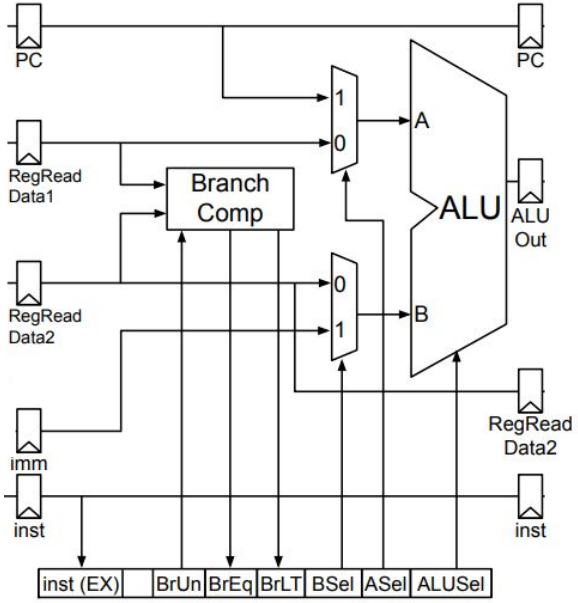
Instruction Decode (ID)

- Decode instruction into necessary fields
  - opcode, func3, func7
  - get registers (rs1, rs2, rd)
  - address, immediate (varies by instruction)
- All information is decoded but only certain pieces are used.
- Generates immediate (ImmGen, ImmSel)



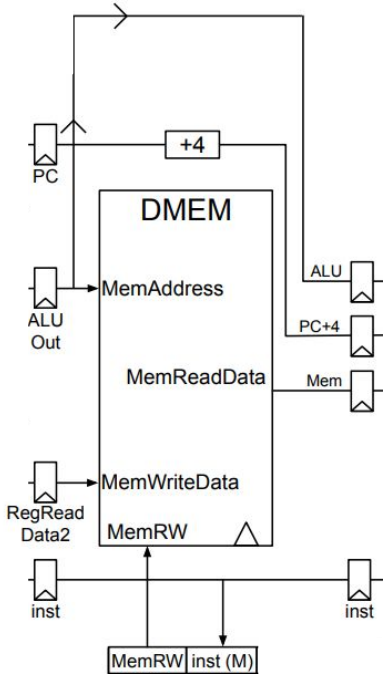
Execute (EX)

- Most calculations done here (adding, shifting, comparing, etc.)
  - For loads and stores, memory address computation is done here
- Determines whether branch needs to be taken.



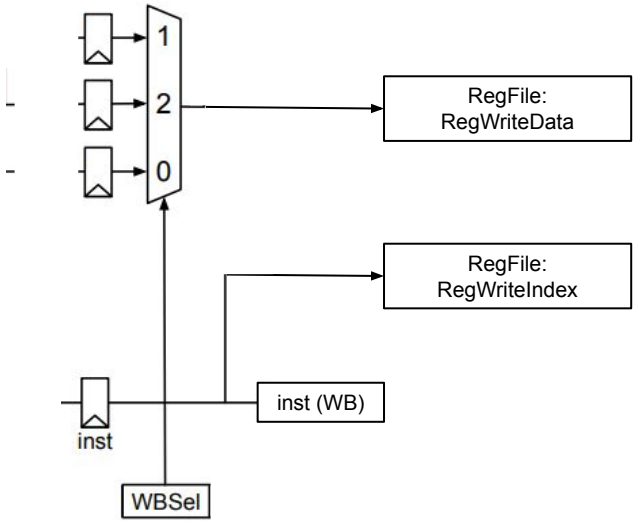
Memory Access (MEM)

- For loads and stores, write or read contents to/from memory address
- For all other instructions, idle

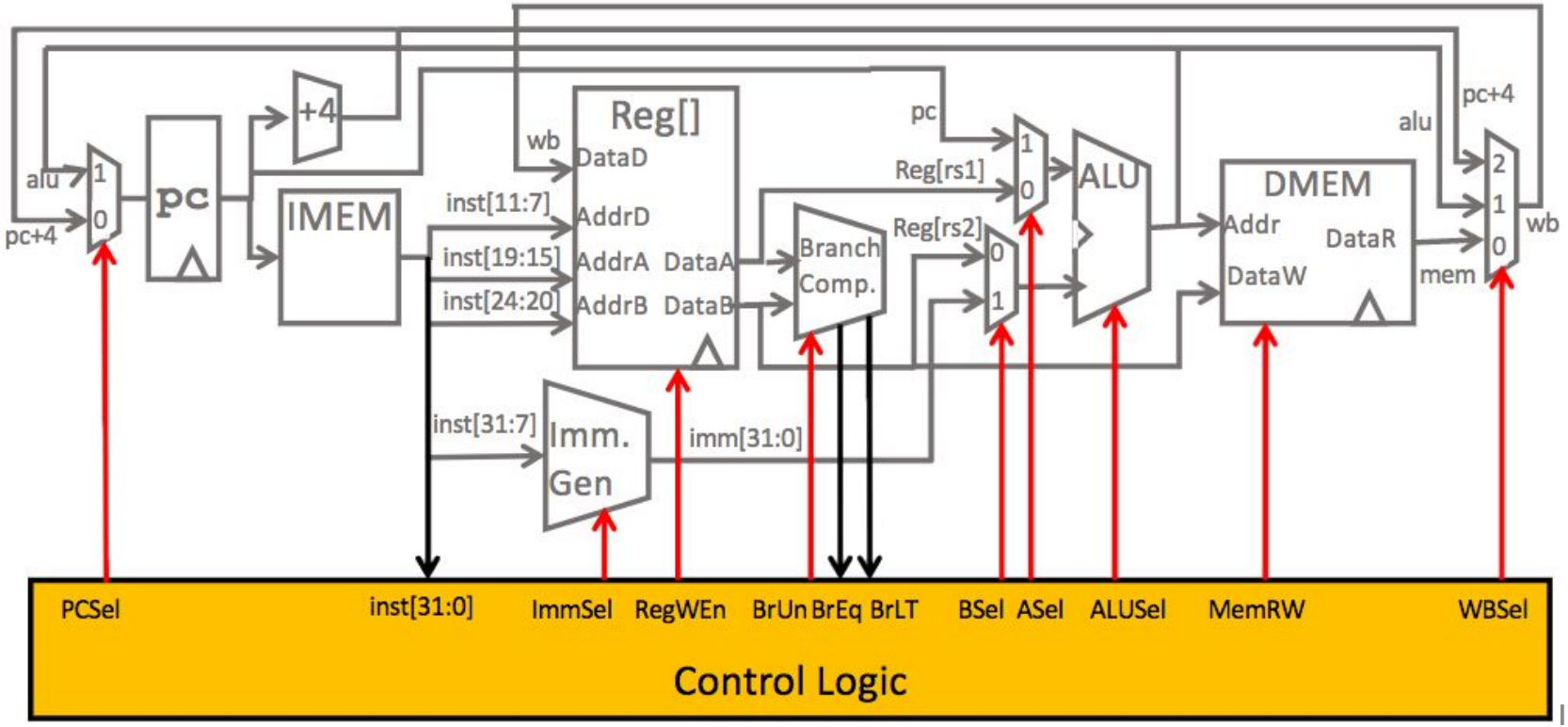


Writeback (WB)

- Write the appropriate value from the computation into the register
  - o not used for stores, branches



Control Signals



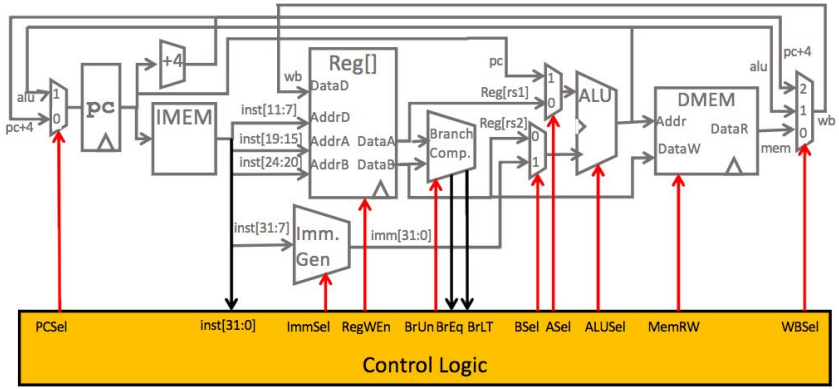
Control Signals

`PCSel`: choose value to store in PC

`ImmSel`: tells the `ImmGen` the type of immediate to generate

`ASel`: input into ALU, either Reg[rs1] or pc

`BSel`: input into ALU, either Reg[rs2] or imm/offset



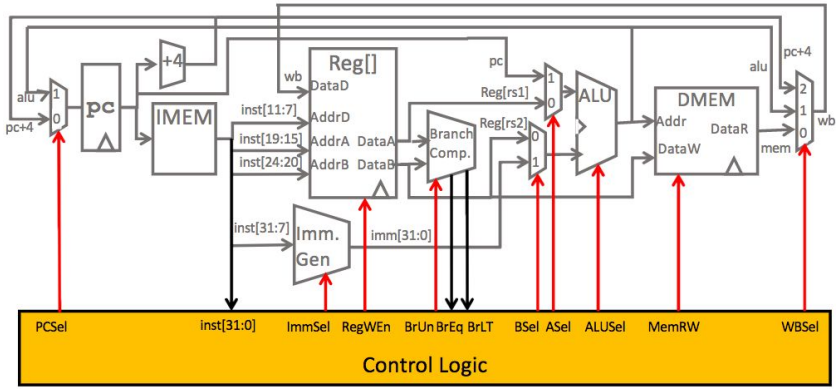
Control Signals

`BrUn:` unsigned comparison or not (bltu)

`BrEq:` branch if equal, affects PCSel

`BrLT:` branch if less than, affects PCSel

`ALUSel:` choose operation

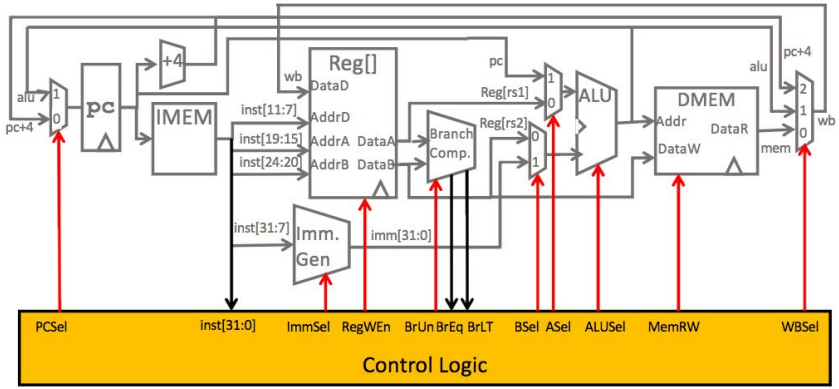


Control Signals

`MemRW`: memory read/write, read from (lw) or write to (sw) memory, default to read when not storing

`WBSel`: selects value stored in destination register

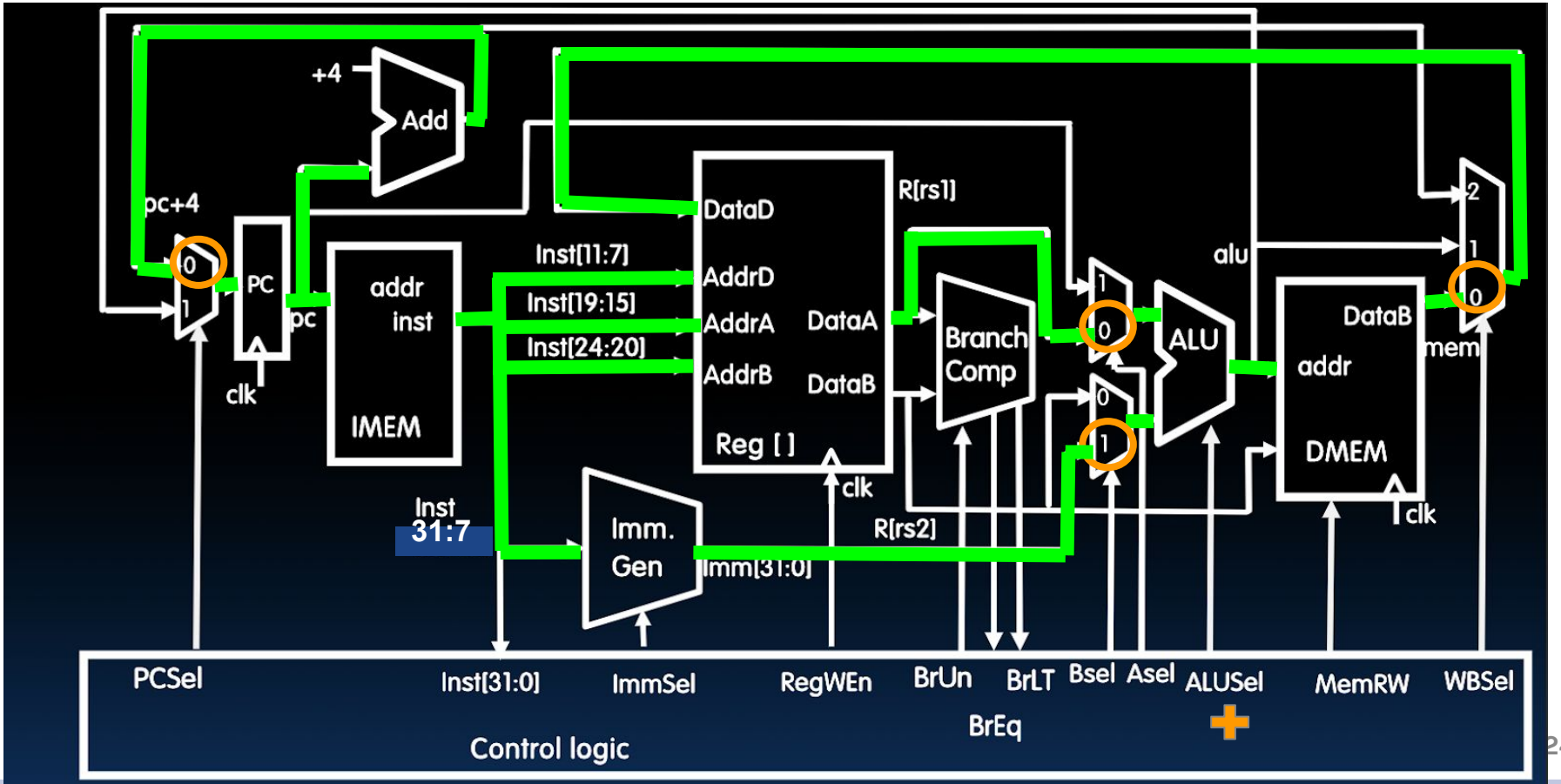
`RegWEn`: writes to destination register or not



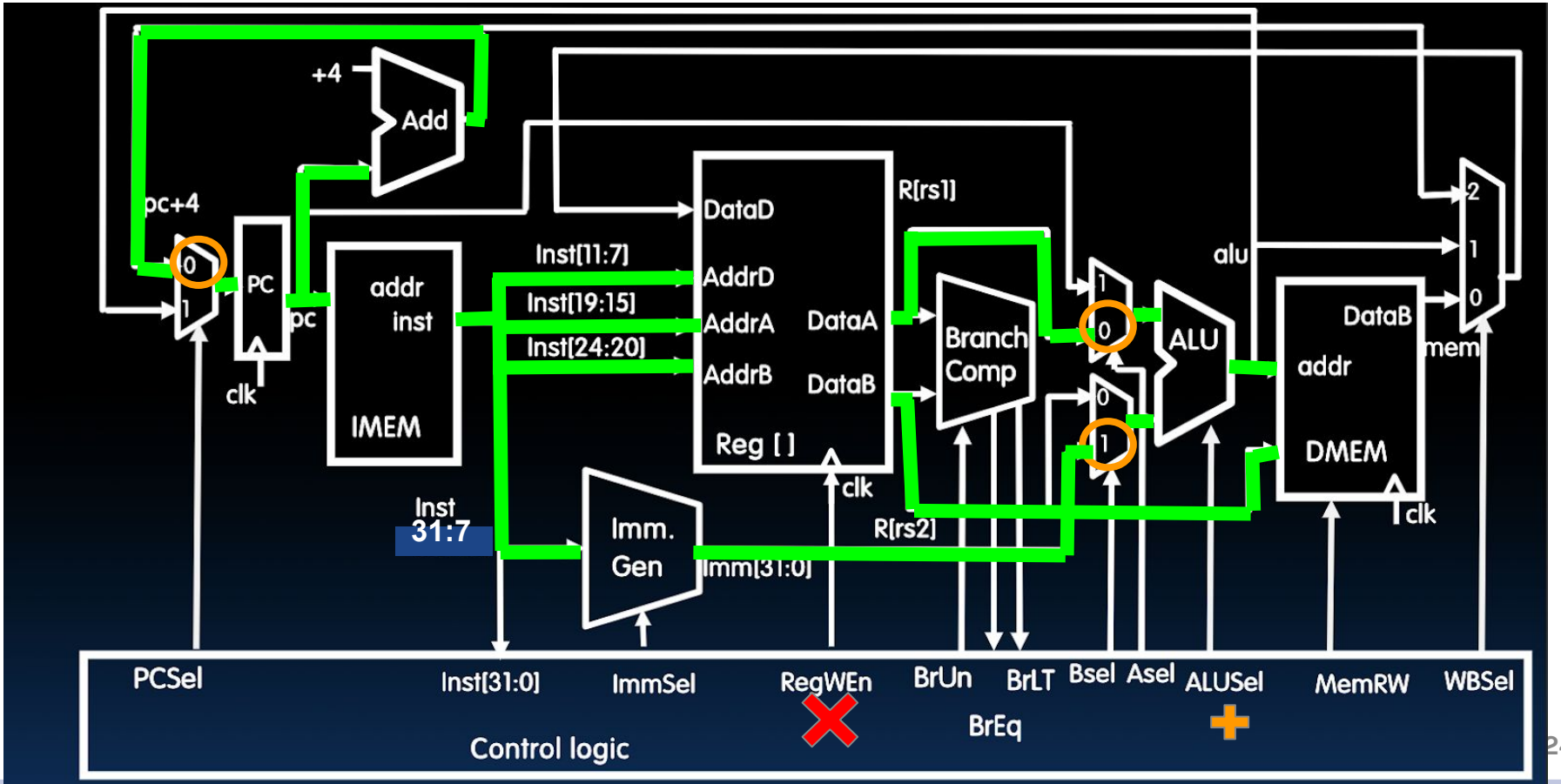
What signals to choose?

Control Bit(s)	Question	Yes	No
BrEq, BrLt	Is this a branch inst?	0/1 (depends on if branch condition was satisfied)	*
PCSel	Are we jumping/branching?	ALUOut for jump, PC+4/ALUOut for branch (depends on if branch condition was satisfied)	PC+4
ImmSel	Is this an R inst?	*	I/S/SB/UJ/J
BrUn	Are we branching?	1 if unsigned, 0 otherwise	*
ASel	Are we jumping/branching?	PC	Reg
BSel	Is this an R inst?	Reg	Imm
MemRW	Are we storing in memory?	1	0
RegWEn	Are we modifying a register?	1	0
ALUSel	What math operation are we using? Check green sheet. (Will usually be “add”, unless we are definitely not adding, like “mul”.)		
WBSel	Is RegWEn 1?	PC+4 when we are storing PC+4; ALU output when we are storing the output of a math operation; Memory output when we are loading something from memory	*

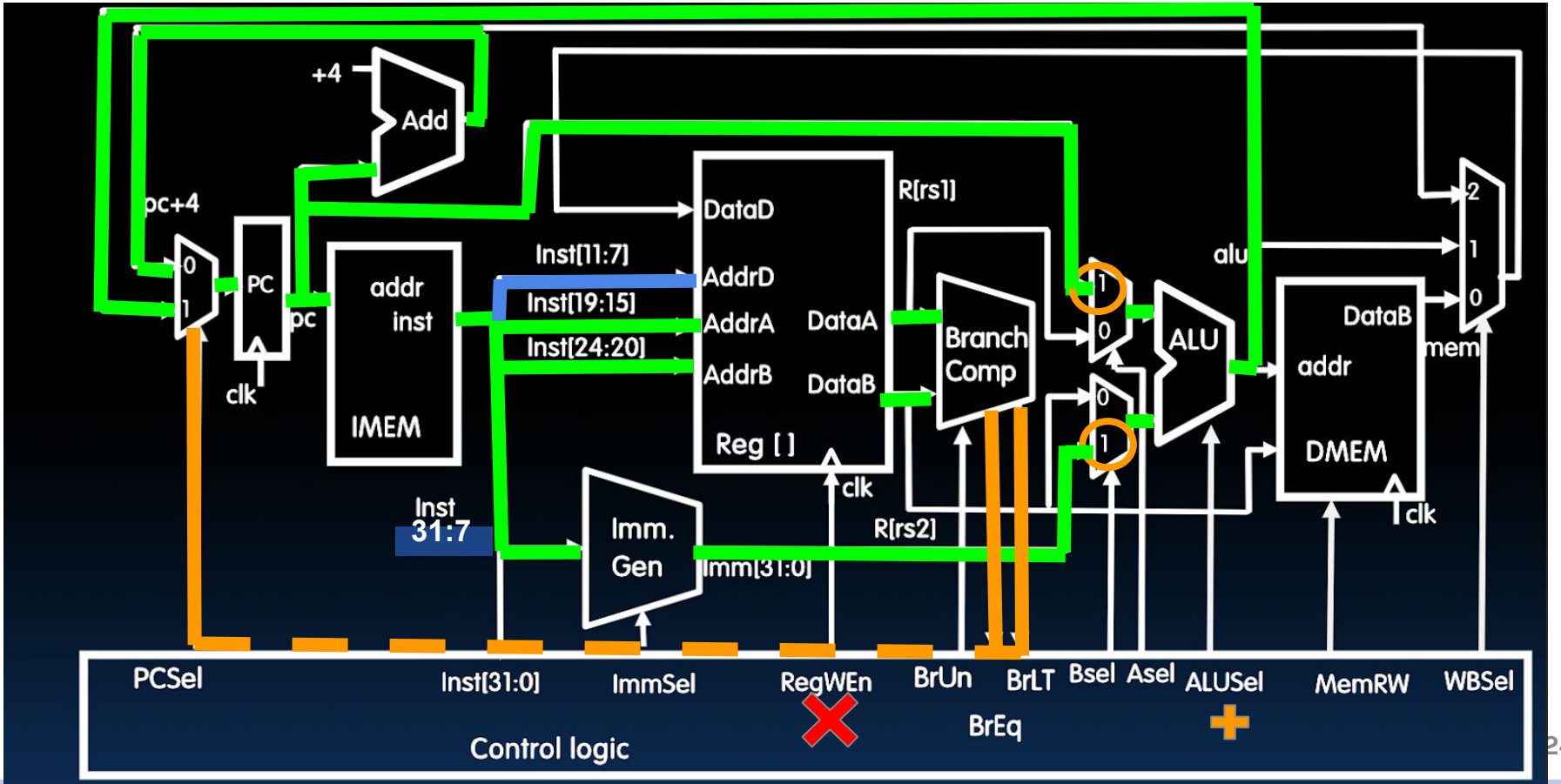
load


+4
Add
pc+4
PC
clk
pc
addr
inst
IMEM
Inst[11:7]
Inst[19:15]
Inst[24:20]
Inst
31:7
DataD
AddrD
AddrA
AddrB
DataA
DataB
Reg [ ]
clk
R[rs1]
R[rs2]
Imm.
Gen
Imm[31:0]
Branch
Comp
alu
ALU
DataB
addr
DMEM
clk
mem
PCSel
Inst[31:0]
ImmSel
RegWEn
BrUn
BrLT
Bsel
Asel
ALUSel
MemRW
WBSel
BrEq
Control logic

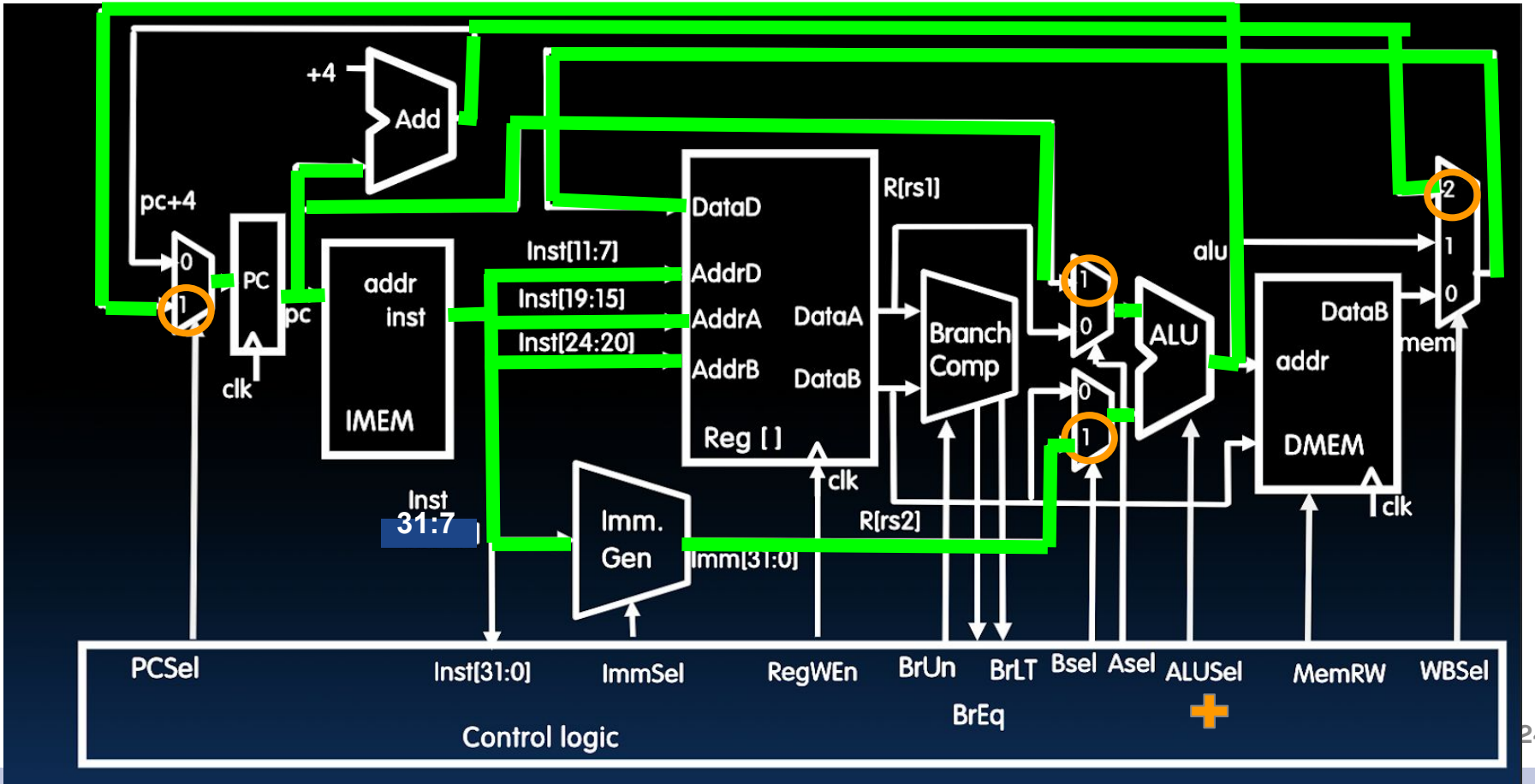
store


+4
Add
pc+4
PC
clk
pc
addr
inst
IMEM
Inst[11:7]
Inst[19:15]
Inst[24:20]
DataD
AddrD
AddrA
AddrB
DataA
DataB
Reg [ ]
clk
R[rs1]
R[rs2]
Branch
Comp
ALU
alu
DataB
addr
DMEM
clk
mem
Inst
31:7
Imm.
Gen
Imm[31:0]
PCSel
Inst[31:0]
ImmSel
RegWEn
BrUn
BrLT
Bsel
Asel
ALUSel
MemRW
WBSel
BrEq
Control logic

branch


+4
Add
pc+4
0
1
PC
clk
pc
addr
inst
IMEM
Inst[11:7]
Inst[19:15]
Inst[24:20]
Inst
31:7
DataD
AddrD
AddrA
AddrB
DataA
DataB
Reg [ ]
clk
R[rs1]
R[rs2]
Imm.
Gen
Imm[31:0]
Branch
Comp
alu
1
0
ALU
DataB
addr
DMEM
clk
2
1
0
mem
PCSel
Inst[31:0]
ImmSel
RegWEn
BrUn
BrLT
BrEq
Bsel
Asel
ALUSel
MemRW
WBSel
Control logic

jal


+4
Add
pc+4
0
1
PC
clk
pc
addr
inst
IMEM
Inst[11:7]
Inst[19:15]
Inst[24:20]
DataD
AddrD
AddrA
AddrB
DataA
DataB
Reg [ ]
clk
R[rs1]
R[rs2]
Branch
Comp
alu
1
0
ALU
DataB
addr
DMEM
clk
mem
2
1
0
Inst
31:7
Imm.
Gen
Imm[31:0]
PCSel
Inst[31:0]
ImmSel
RegWEn
BrUn
BrLT
Bsel
Asel
ALUSel
MemRW
WBSel
BrEq
Control logic

Control Logic Table (Incomplete)

Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
`add`	*	*	+4	*	*	Reg	Reg	Add	Read	1	ALU
`sub`	*	*	+4	*	*	Reg	Reg	Sub	Read	1	ALU
(R-R Op)	*	*	+4	*	*	Reg	Reg	*(Op)*	Read	1	ALU
`addi`	*	*	+4	I	*	Reg	Imm	Add	Read	1	ALU
`lw`	*	*	+4	I	*	Reg	Imm	Add	Read	1	Mem
`sw`	*	*	+4	S	*	Reg	Imm	Add	Write	0	*
`beq`	0	*	+4	B	*	PC	Imm	Add	Read	0	*
`beq`	1	*	ALU	B	*	PC	Imm	Add	Read	0	*
`bne`	0	*	ALU	B	*	PC	Imm	Add	Read	0	*
`bne`	1	*	+4	B	*	PC	Imm	Add	Read	0	*
`blt`	*	1	ALU	B	0	PC	Imm	Add	Read	0	*
`bltu`	*	1	ALU	B	1	PC	Imm	Add	Read	0	*
`jalr`	*	*	ALU	I	*	Reg	Imm	Add	Read	1	PC+4
`jal`	*	*	ALU	J	*	PC	Imm	Add	Read	1	PC+4
`auipc`	*	*	+4	U	*	PC	Imm	Add	Read	1	ALU

Single Cycle Datapath: New Instructions

We want to add a new RISC-V **I-type** instruction inc (for increment) that **takes in a register rs1** and **obtains the value stored in memory at that address**. It then **increments that value by imm** and then **stores the value back into its original place in memory** (at the address contained in rs1). Assume that the value stored in memory is a 32-bit 2's complement integer and that imm is also a 2's complement number.

The syntax is: `inc rs1, imm`

Give the description for this instruction (i.e. the description for the instruction `add rd, rs1, rs2` is `R[rd] = R[rs1] + R[rs2]; PC = PC + 4`):

Single Cycle Datapath: New Instructions

The syntax is: `inc rs1, imm`

Give the description for this instruction (i.e. the description for the instruction `add rd, rs1, rs2` is `R[rd] = R[rs1] + R[rs2]; PC = PC + 4`):

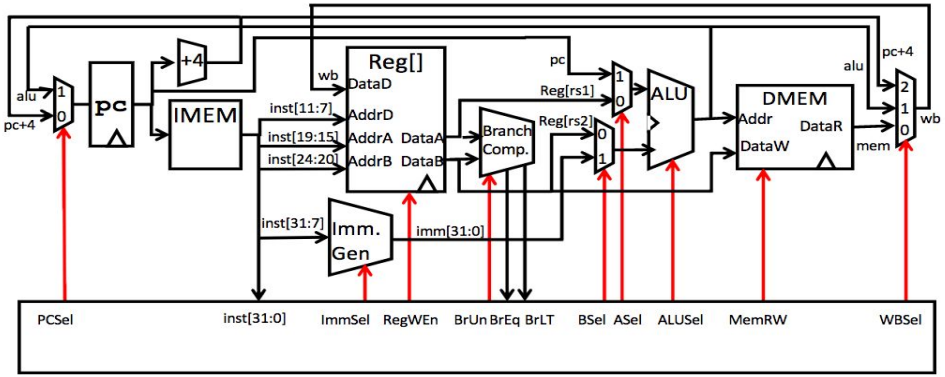
`M[R[rs1]] = M[R[rs1]]+ SignExtImm; PC = PC + 4`

Single Cycle Datapath: New Instructions

Given the instruction description:

`M[R[rs1]] = M[R[rs1]]+ SignExtImm; PC = PC + 4`

What changes do you need to make to the standard single-cycle datapath so that this instruction can be supported in a single cycle? **Assume the memory can support 1 asynchronous read and 1 synchronous write per cycle** (always reads, MemRW controls only writes)



Single Cycle Datapath: New Instructions

1. Add a value to the mux on ALU input A and update `ASel` to select value we READ from memory so that we can get `M[R[rs1]]` into the ALU to increment it.
2. Add a mux on the input of `Addr` on DMEM to select between output of ALU and bus from regfile as input address so that we can directly access `M[R[rs1]]` (no ALU address calc)
3. Add a mux on the input of `DataW` on DMEM to select between the old `DataW` bus and the output of the ALU (i.e. the value we just computed) so that we can store `M[R[rs1]] + SignExtImm` into DMEM at address `R[rs1]`

***These muxes can be controlled by one control signal - call it doInc: 1 means we're doing an inc instruction*

Single Cycle Datapath: New Instructions

What are the values of all the control bits?

```
PCSel: PC + 4
ImmSel:
RegWEn:
BrUn:
ASel:
BSel:
ALUSel:
MemRW:
WBSel:
```

Single Cycle Datapath: New Instructions

What are the values of all the control bits?

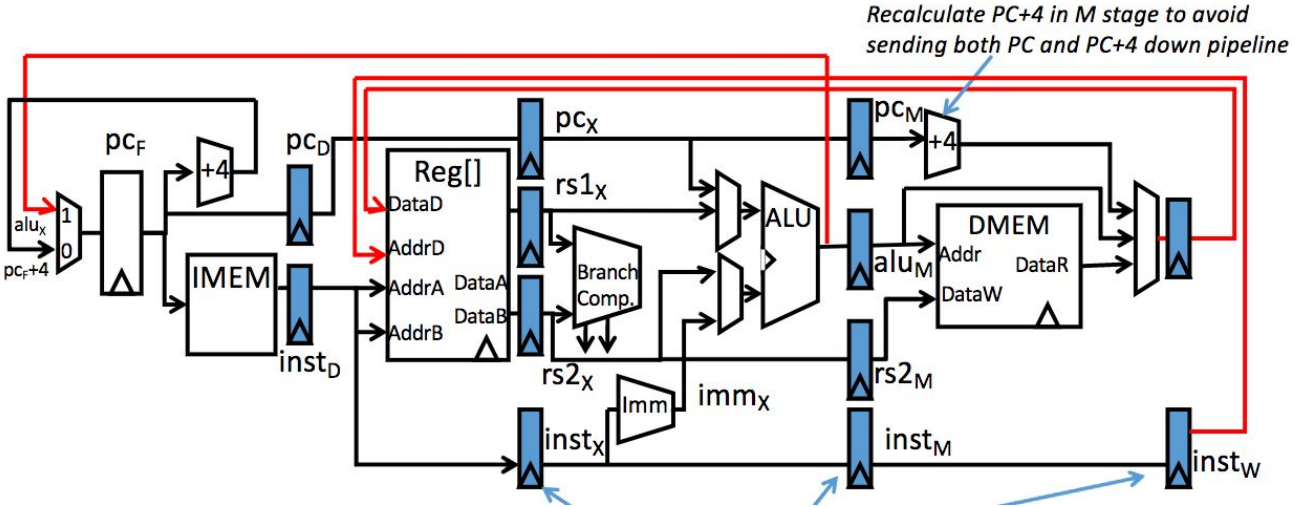
`PCSel:` `PC + 4`
`ImmSel:` `I` (Select immediate)
`RegWEn:` `0` (Don’t enable)
`BrUn:` `*`
`ASel:` `2` (new ASel value added. Will be in binary: 10_2)
`BSel:` `1`
`ALUSel:` `Add`
`MemRW:` `0, 1` (first read old data value, then write new data value)
`WBSel:` `*` (Don’t need to write to regfile and RegWEn = 0)

Pipelining

Why did the CS student call the plumber? They couldn't flush their pipeline.

Pipelined Datapath

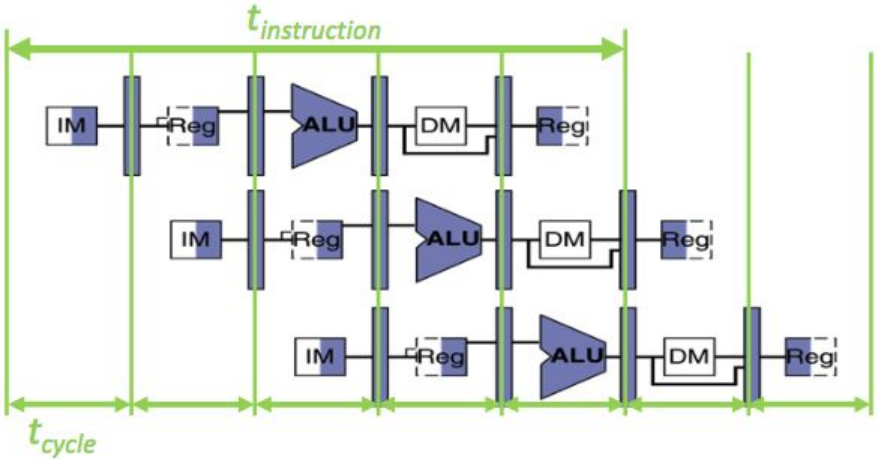
Insert registers between stages



We pipeline the instruction **with** the data to preserve the control logic at each respective step

Pipelining Intro

Increase efficiency of datapath by reusing 5 steps instead of waiting for cycle to finish to move on to next instruction



Structural Hazard

- Instructions in different steps need access to the same physical resource
- Solution 1: stalling other operations until resource becomes free to use
- Solution 2: Add more hardware, more physical resources

Data Hazard

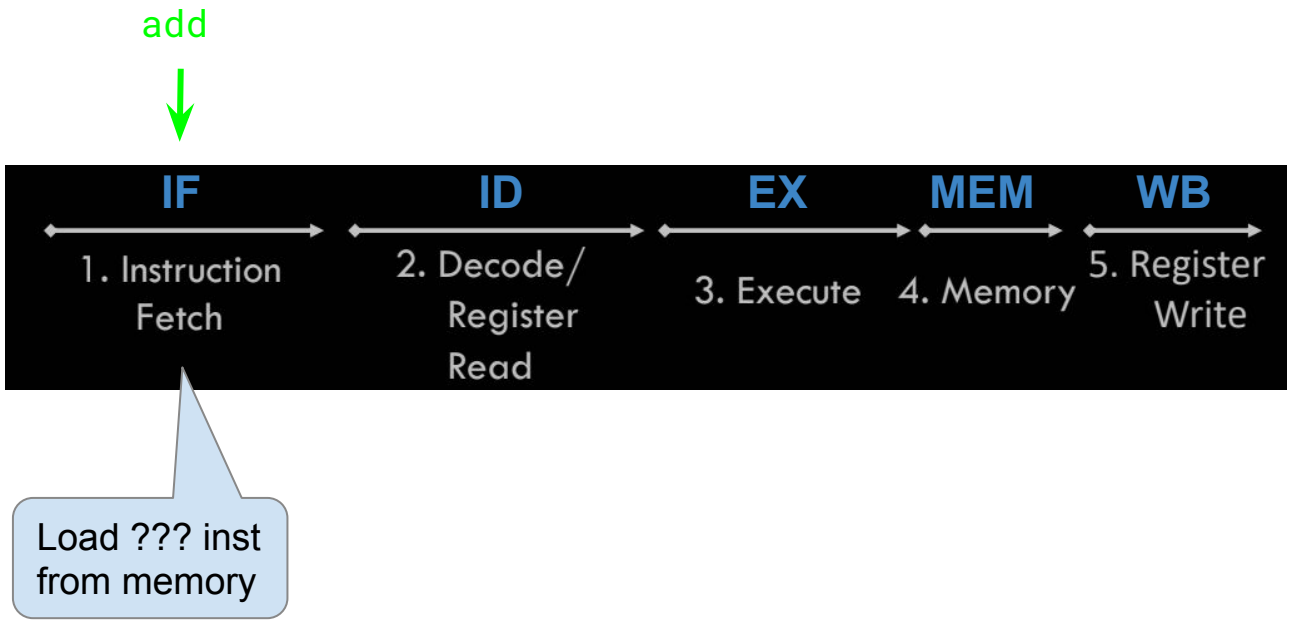
- Result from one instruction needed earlier in another instruction

```
lw x2, 4(x4)
addi x3, x2, 1
```

  x2 isn't ready in time!
- Solution 1: stall/bubble/nop (`add x0, x0, x0`)
- Solution 2: forwarding, send correct value from earlier stage of earlier instruction to later stage of later instruction (we don't wait for value to be stored in register - faster, but can't solve every hazard)

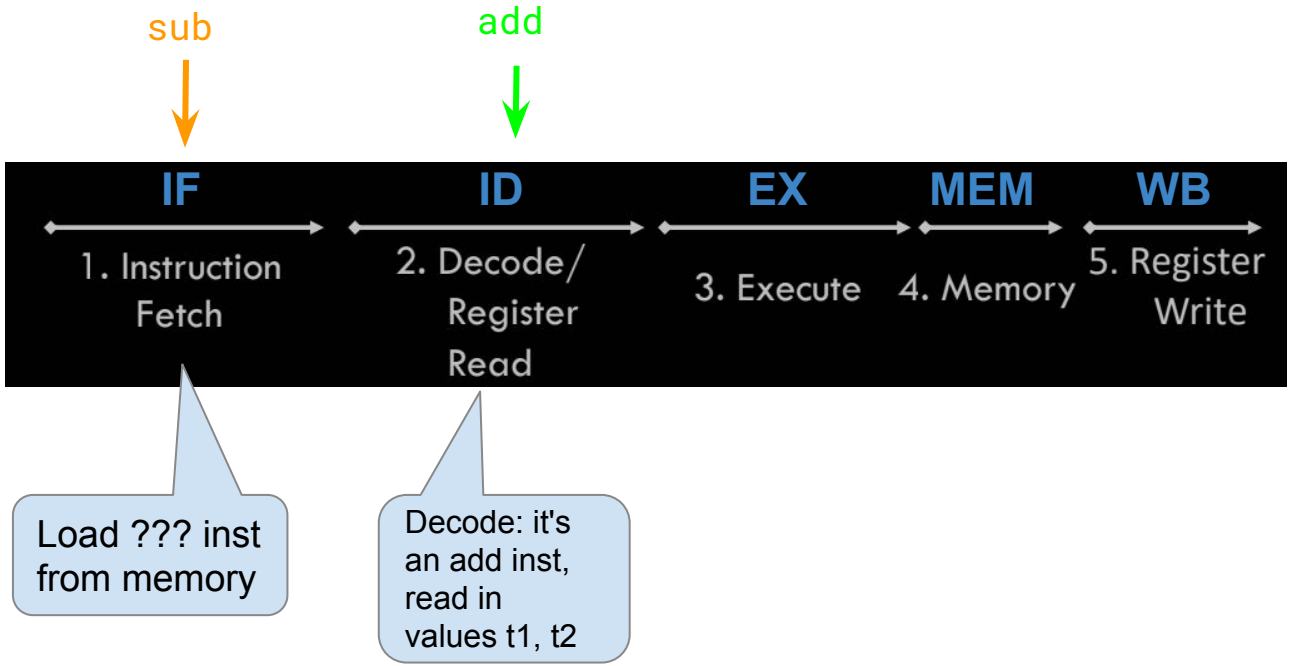
Data hazards - ALU

```
add t0, t1, t2
sub t3, t0, t5
```



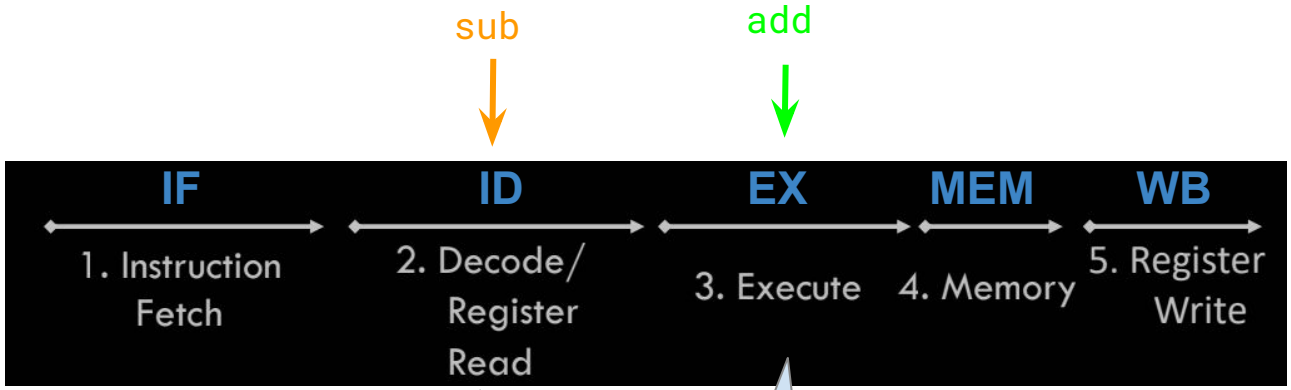
Data hazards - ALU

```
add t0, t1, t2
sub t3, t0, t5
```



Data hazards - ALU

```
add t0, t1, t2
sub t3, t0, t5
```



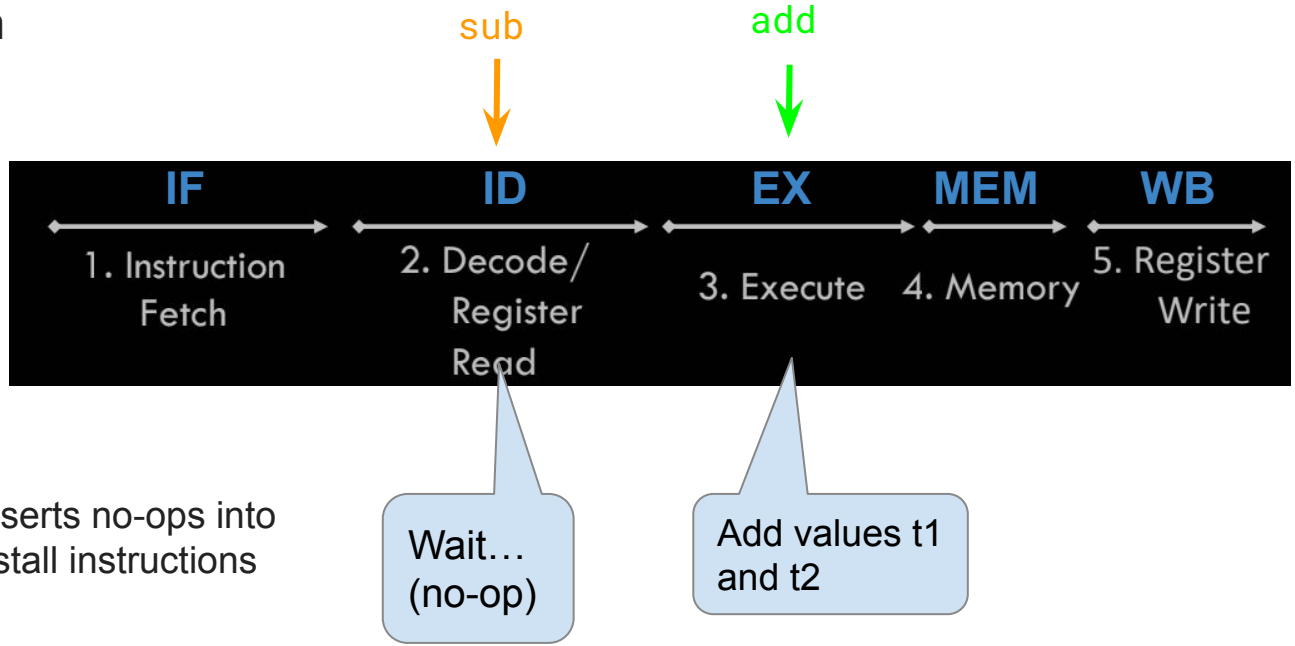
But wait - sub is supposed to use the value of t0 *after* add has already updated it, but add hasn't reached WB yet, and sub is already reading t0!

How do we fix this?

Data hazards - ALU

```
add t0, t1, t2
sub t3, t0, t5
```

Solution 1: stall the sub instruction

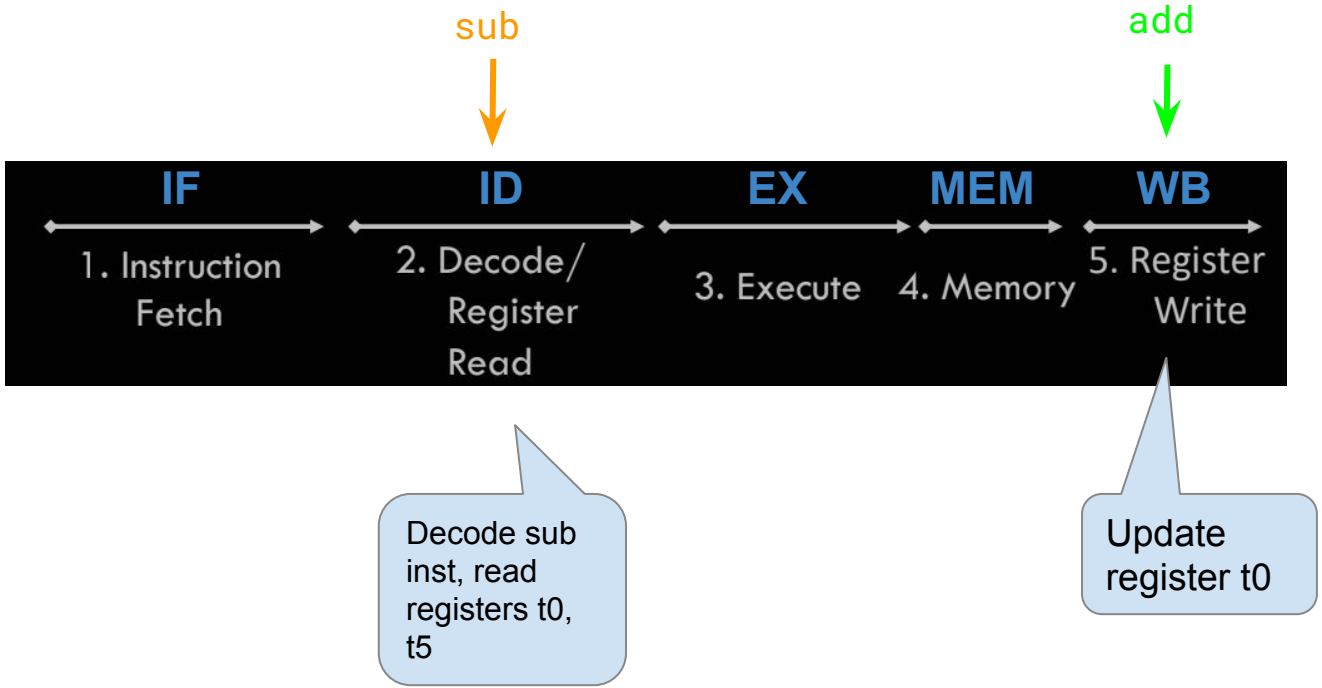


Compiler inserts no-ops into pipeline to stall instructions

Data hazards - ALU

```
add t0, t1, t2
sub t3, t0, t5
```

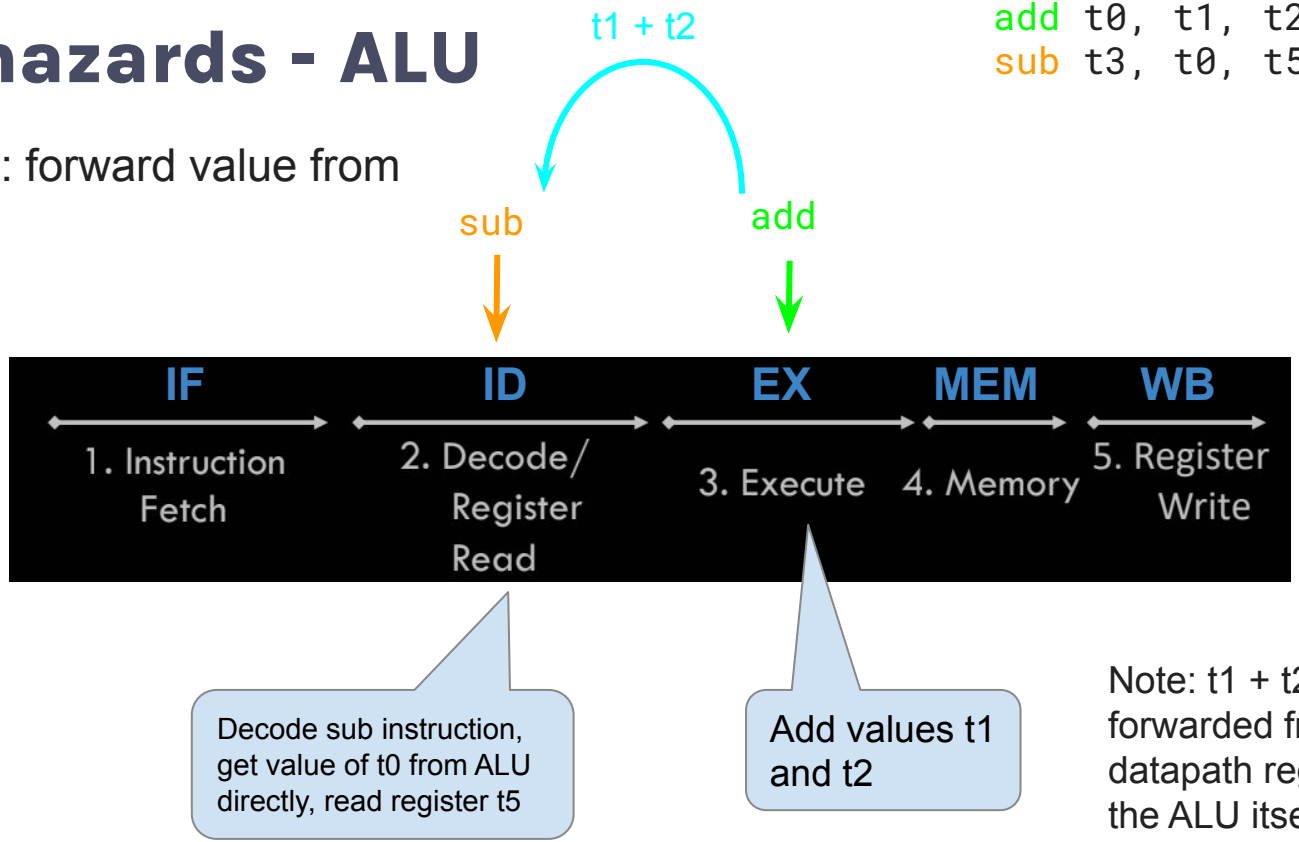
Solution 1: stall the sub instruction



Data hazards - ALU

Solution 2: forward value from ALU

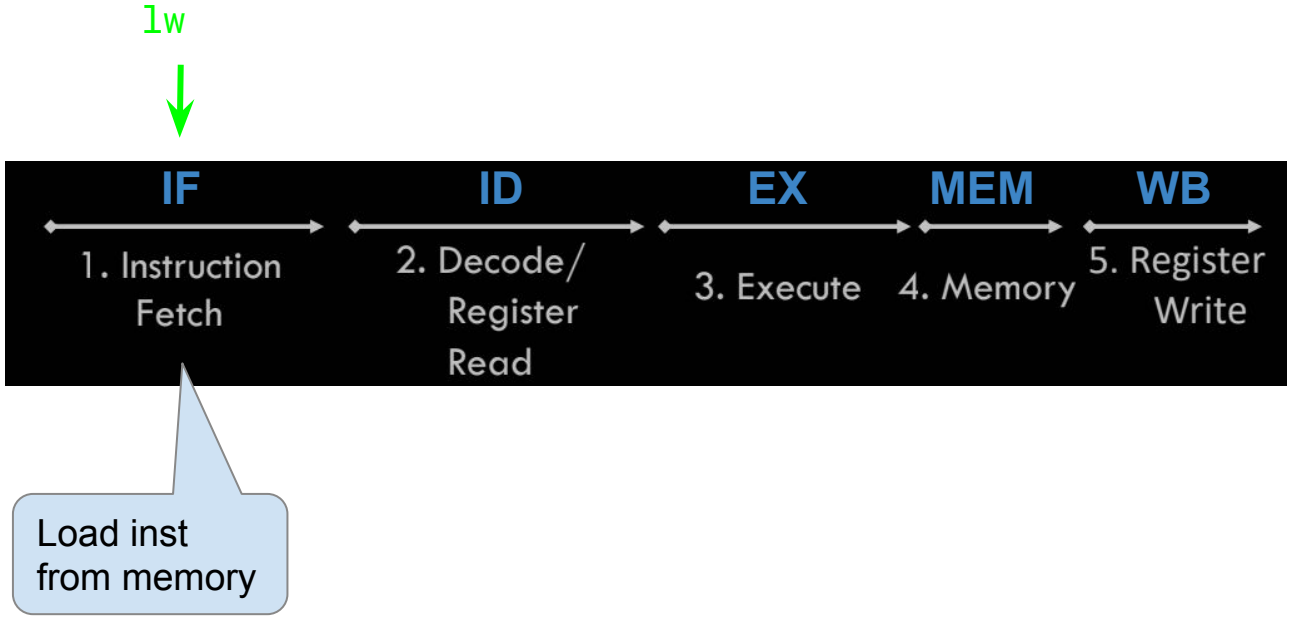
```
add t0, t1, t2
sub t3, t0, t5
```



Note: t1 + t2 value is forwarded from datapath registers, not the ALU itself

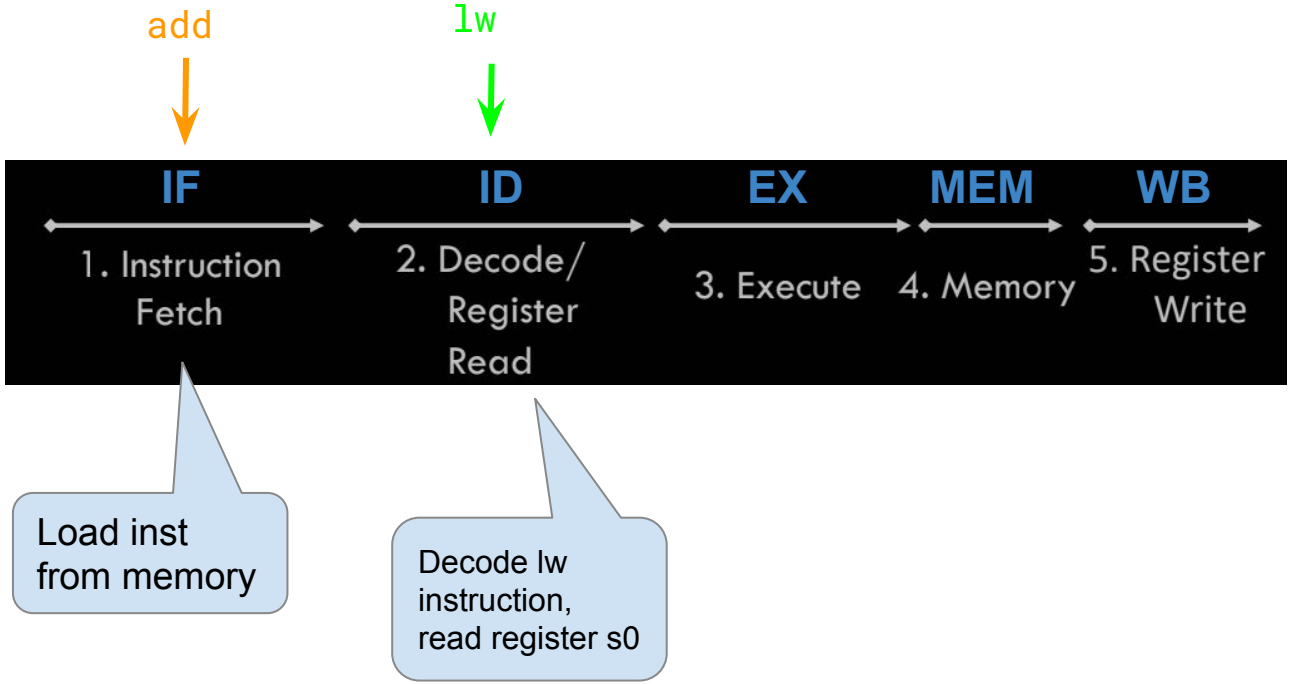
Data hazards - load

```
lw t0, 0(s0)
add t1, t0, t2
```



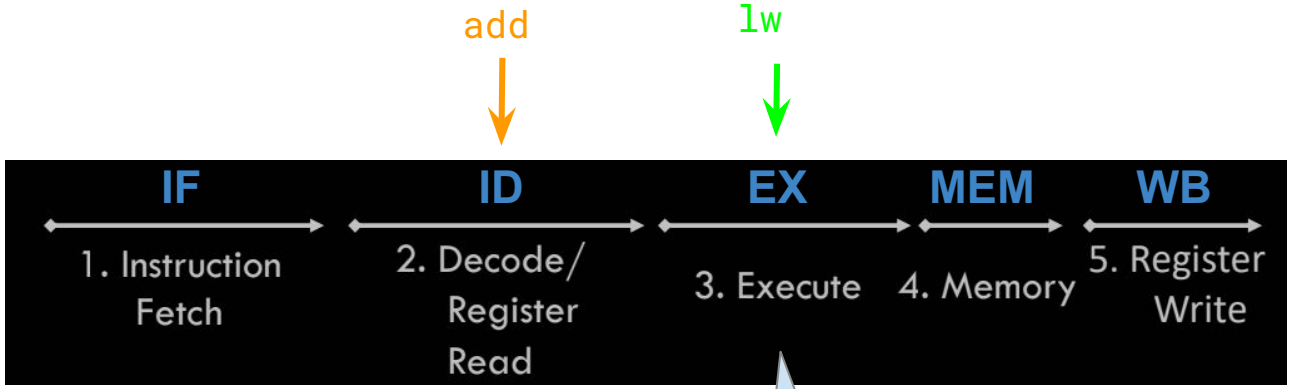
Data hazards - load

```
lw t0, 0(s0)
add t1, t0, t2
```



Data hazards - load

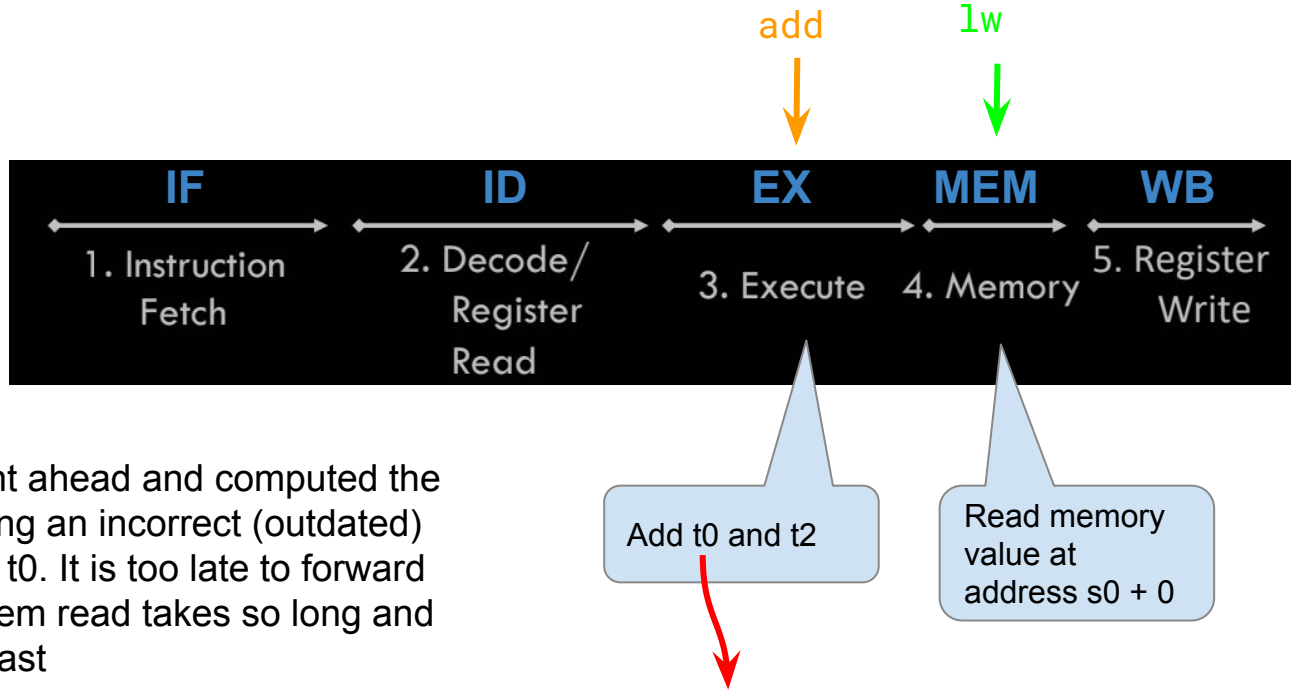
```
lw t0, 0(s0)
add t1, t0, t2
```



But wait - add is supposed to use the value of t0 *after* lw has already updated it, but lw hasn't reached MEM (let alone WB) yet, and add is already reading t0!

Data hazards - load

```
lw t0, 0(s0)
add t1, t0, t2
```

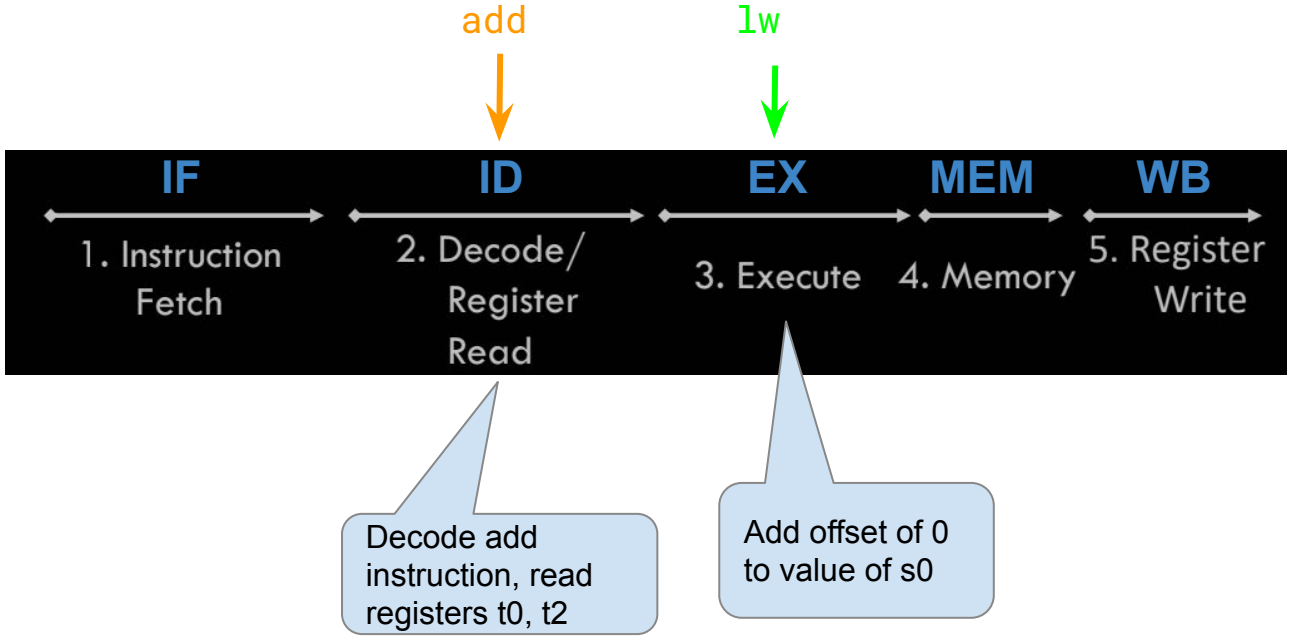


add went ahead and computed the sum using an incorrect (outdated) value of t0. It is too late to forward since mem read takes so long and ALU is fast

Data hazards - load

```
lw t0, 0(s0)
add t1, t0, t2
```

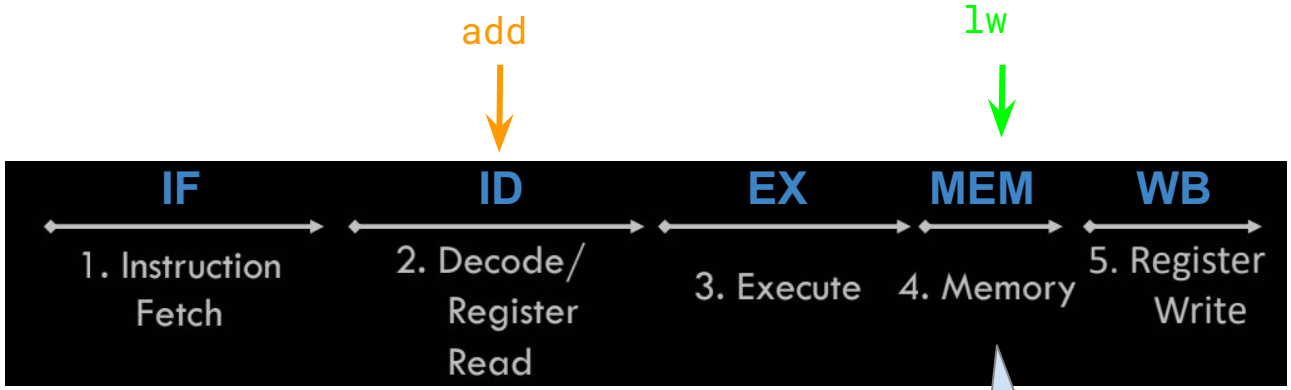
Solution: stall once and forward



Data hazards - load

```
lw t0, 0(s0)
add t1, t0, t2
sub s4, s5, s6
```

Solution: stall once and forward

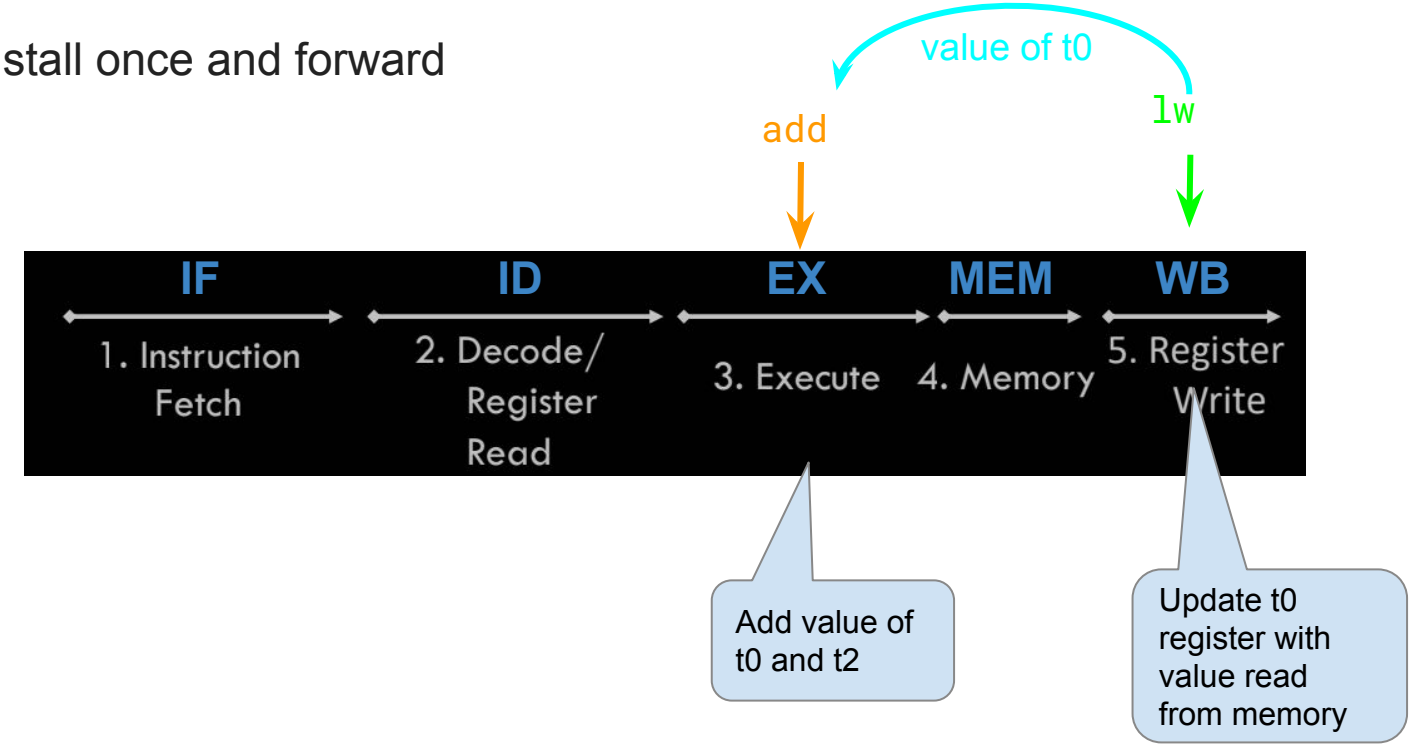


Compiler can also rearrange insts so that sub goes before add to avoid wasting stall time

Data hazards - load

```
lw t0, 0(s0)
add t1, t0, t2
```

Solution: stall once and forward

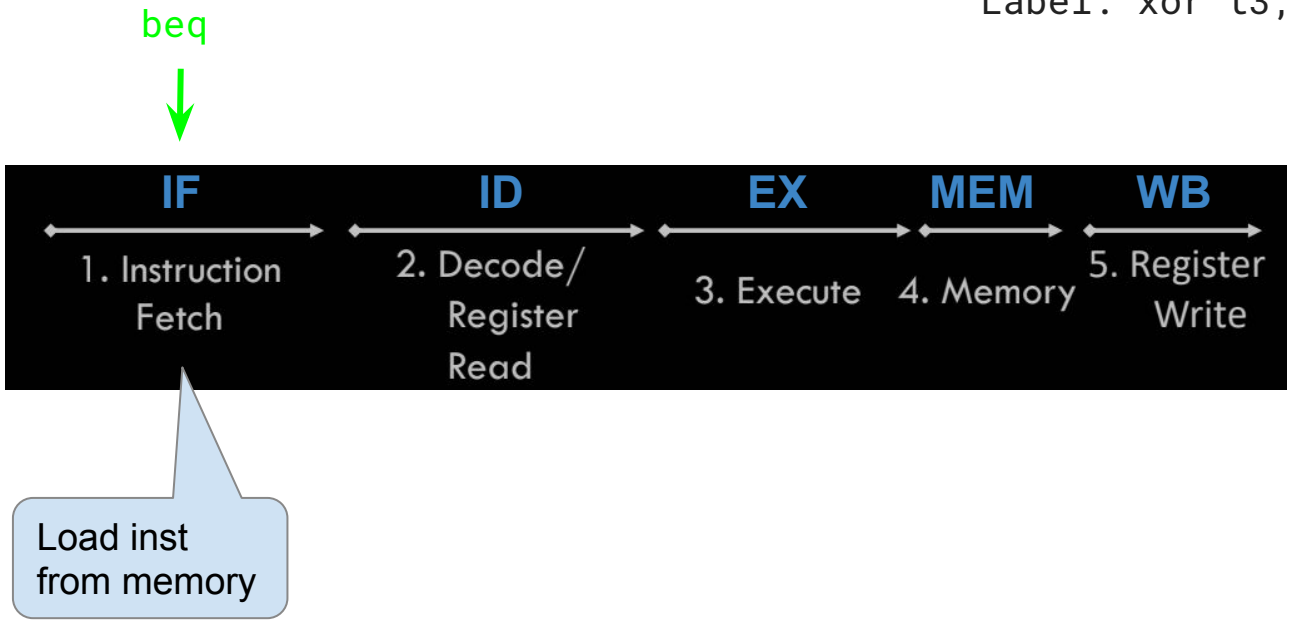


Control Hazard

- Execution order depends on previous instructions
- Branches, jumps
- Solution: branch prediction (e.g. assume conditional always false) and flush instructions if incorrect (nop)

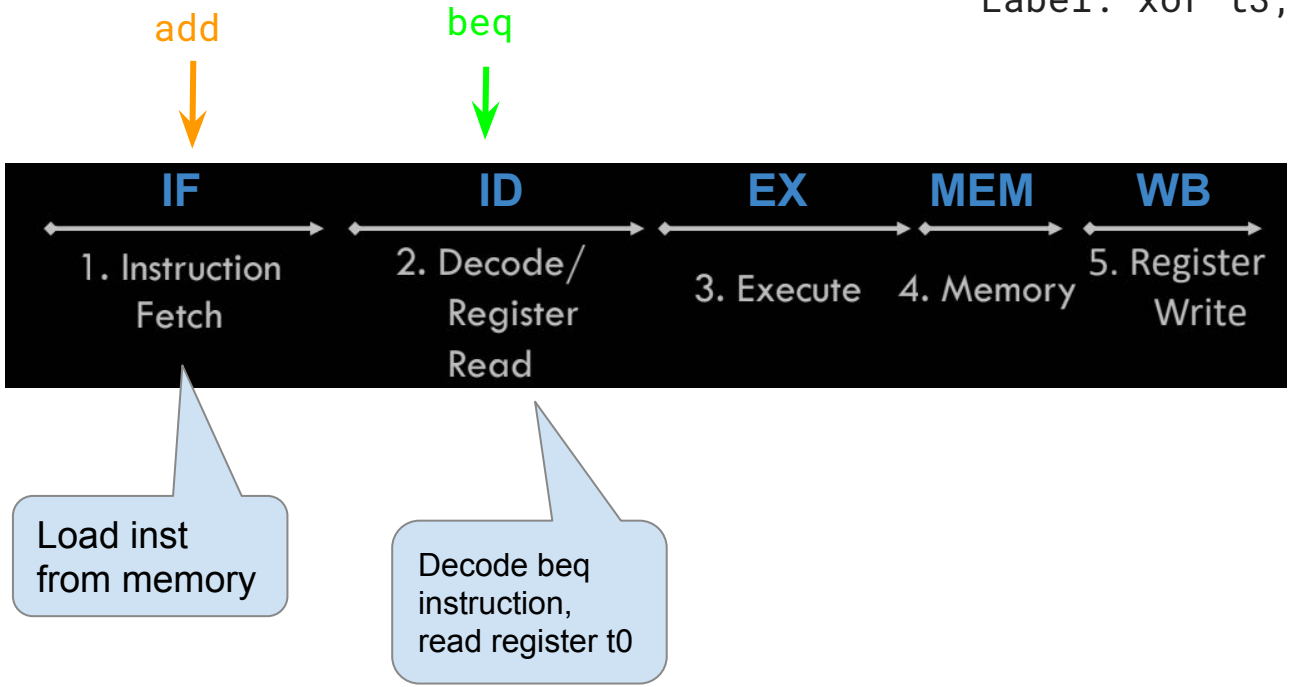
Control Hazard

```
beq t0, t0, Label
add s0, s1, s2
sub a0, a1, a2
...
Label: xor t3, t4, t5
```



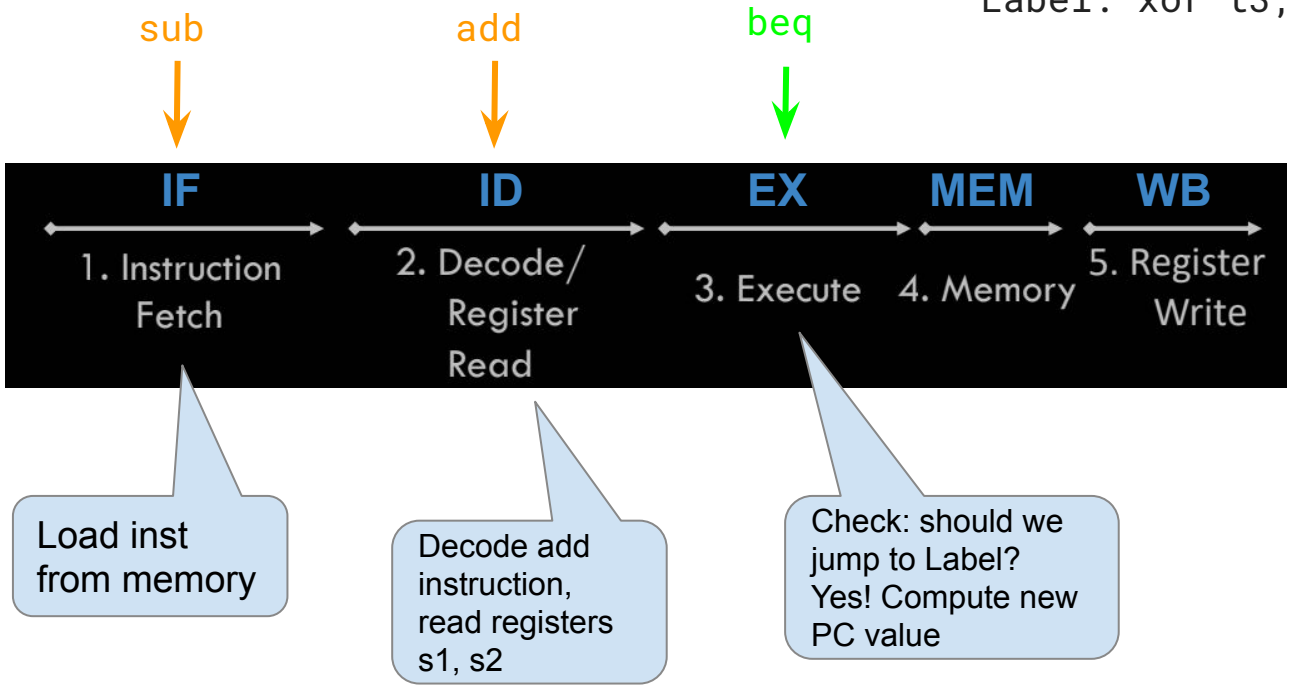
Control Hazard

```
beq t0, t0, Label
add s0, s1, s2
sub a0, a1, a2
...
Label: xor t3, t4, t5
```



Control Hazard

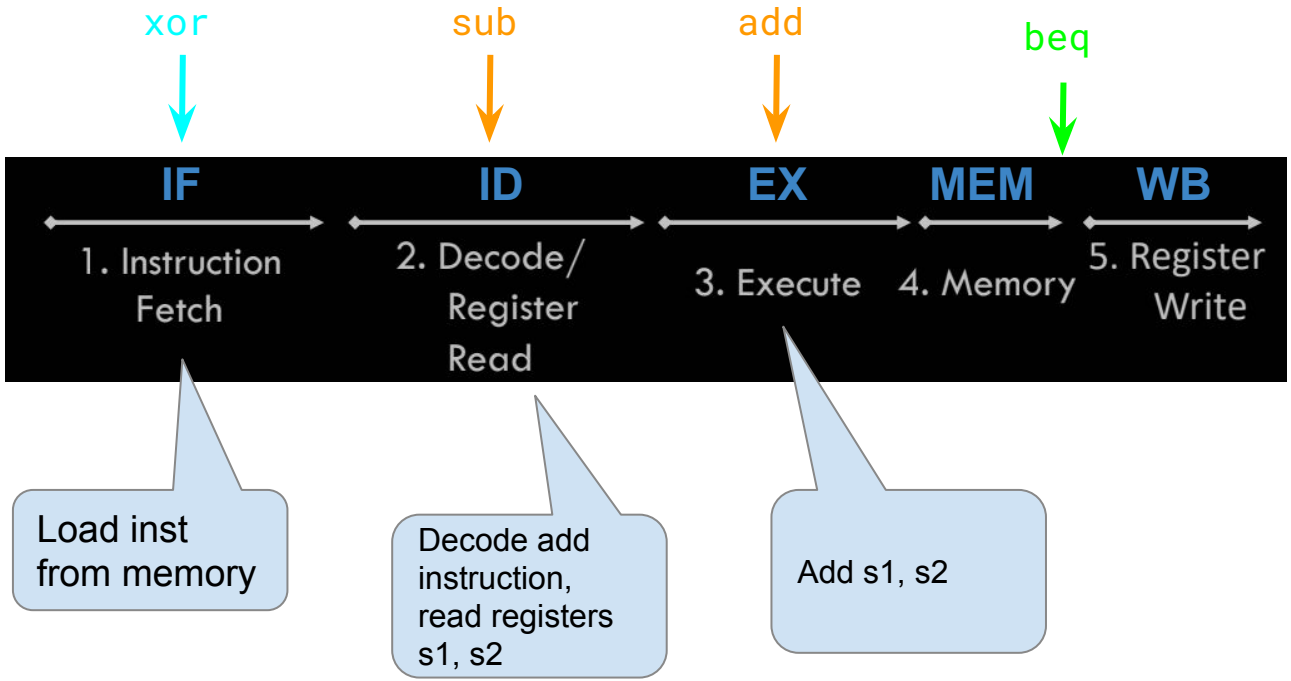
```
beq t0, t0, Label
add s0, s1, s2
sub a0, a1, a2
...
Label: xor t3, t4, t5
```



Control Hazard

Wait a minute - we're branching away, so we don't want to execute add and sub. But if we let the pipeline continue, they'll be executed before xor!

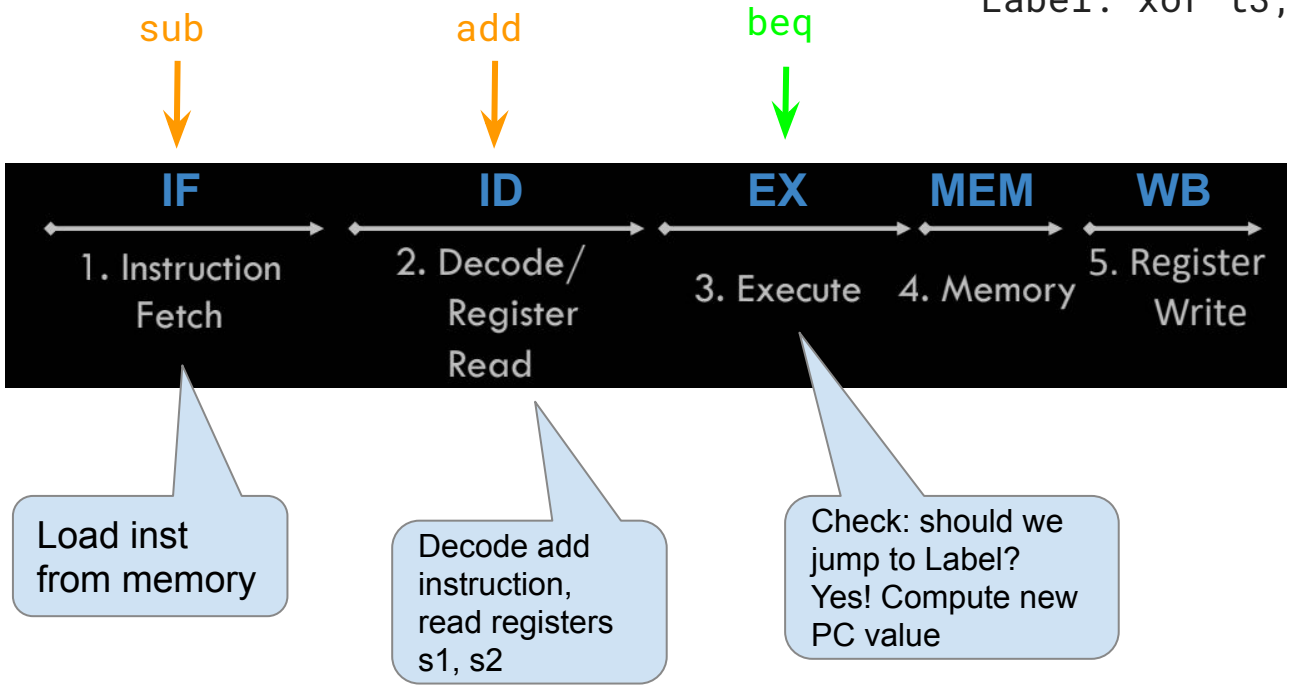
```
beq t0, t0, Label
add s0, s1, s2
sub a0, a1, a2
...
Label: xor t3, t4, t5
```



Control Hazard

Solution: replace add and sub with nops

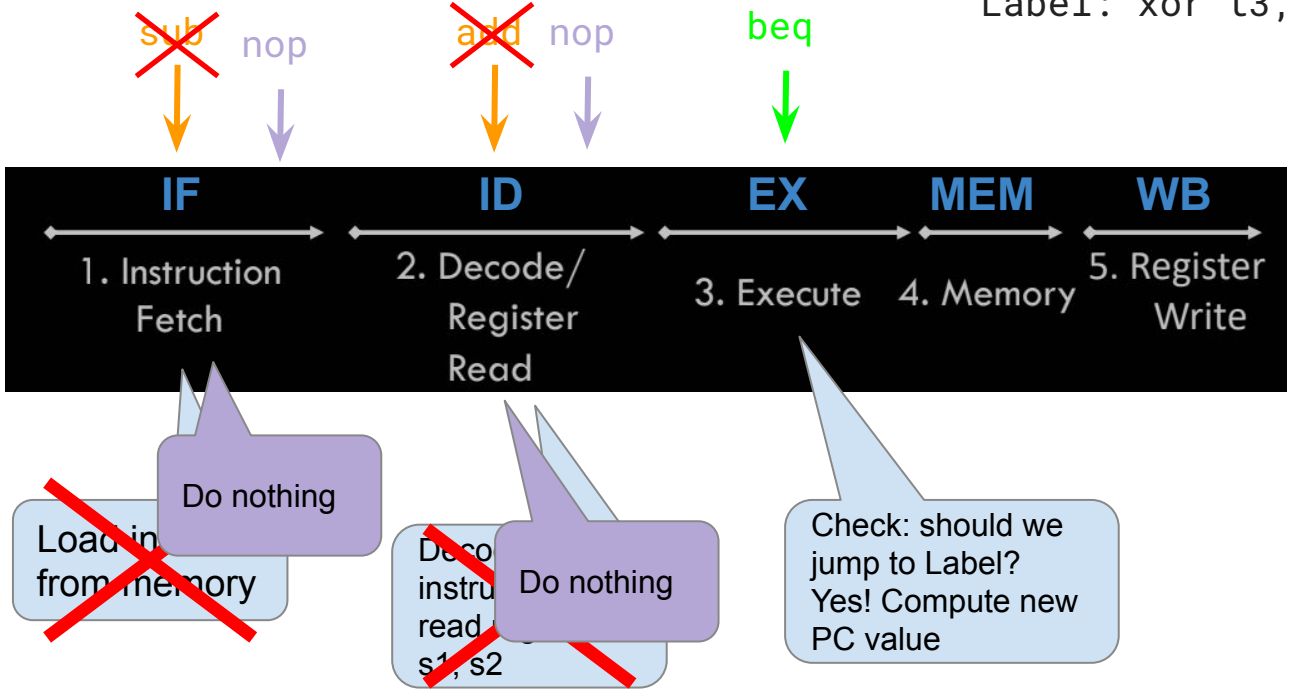
```
beq t0, t0, Label
add s0, s1, s2
sub a0, a1, a2
...
Label: xor t3, t4, t5
```



Control Hazard

Solution: replace add and sub with nops

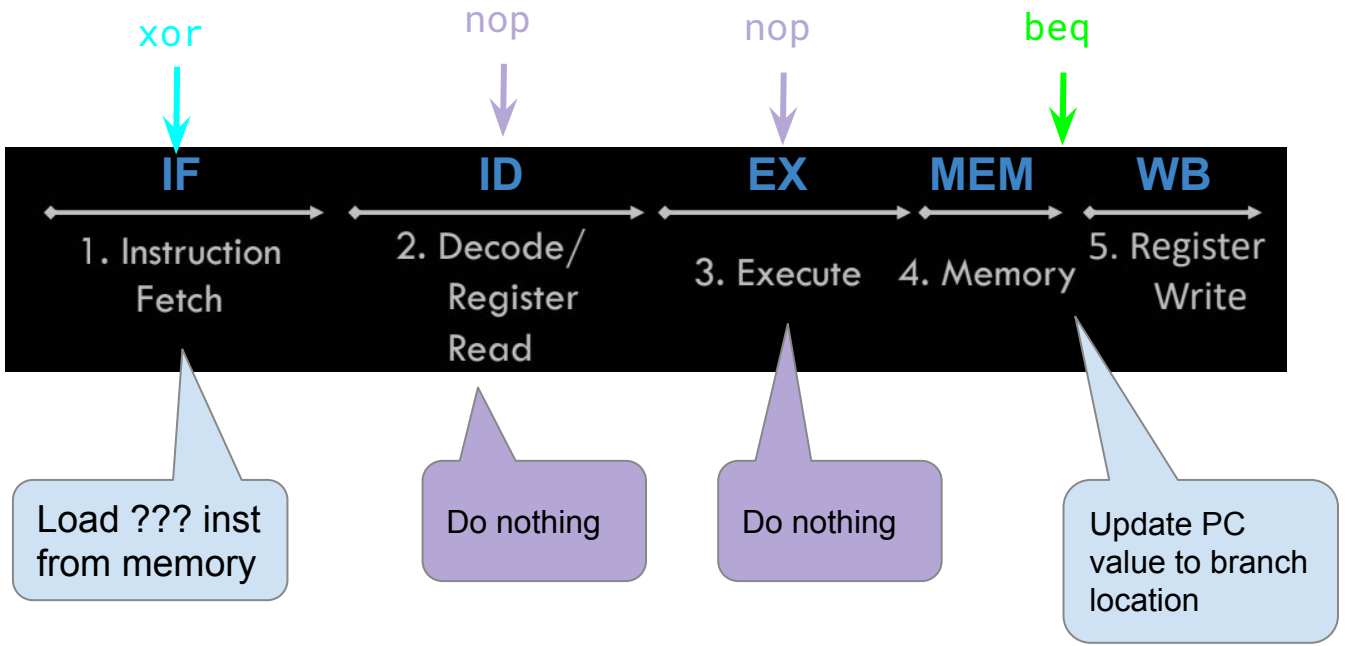
```
beq t0, t0, Label
add s0, s1, s2
sub a0, a1, a2
...
Label: xor t3, t4, t5
```



Control Hazard

Solution: replace add and sub with nops

```
beq t0, t0, Label
add s0, s1, s2
sub a0, a1, a2
...
Label: xor t3, t4, t5
```



Parallelism

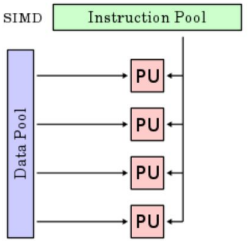
Flynn’s Taxonomy

- Single instruction, single data stream (SISD)
  - A basic sequential computer
- Single instruction, multiple data streams (SIMD)
  - Each instruction acts on multiple data values at once
- Multiple instructions, multiple data streams (MIMD)
  - Independent, parallel processors (multicore machines)
- Multiple instructions, single data stream (MISD)
  - Completes the taxonomy; isn’t used outside special cases

Intel SSE (SIMD)

SIMD: Intel SSE

- Intel SSE is an implementation of SIMD, defining its own 128-bit registers
  - These registers can be split into 8 shorts, 4 ints, 4 (single-precision) floats, 2 doubles, etc.
  - You can tell which type it is from the instruction name!
    - _mm_add_epi**32** vs. _mm_add_epi**64**



Remember remainder (tail) cases!

OMP (MIMD)

Fork-Join Model of Computation

- Directives in C:
  - #pragma omp parallel
  - #pragma omp parallel for
  - #pragma omp critical
- Each thread is independently executed:
  - But variables are *shared* by default except loop indices!
  - Private? private (var)

OpenMP

```
#pragma omp parallel for
for (int x = 0; x < len; x++) {
    *A = x;
    A++;
}
```

```
#pragma omp parallel
{
    for (int x = 0; x < len; x++) {
          *(A+x) = x;
    }
}
```

```
#pragma omp parallel
{
    for(int x = omp_get_thread_num(); x < len; x += omp_get_num_threads()) {
         A[x] = x;
    }
}
```

Spring 2013, #4:

- Are any of these implementations correct?
- Which implementation is fastest?
- How does each compare to a serial implementation?

OpenMP

```
#pragma omp parallel for
for (int x = 0; x < len; x++) {
    *A = x; // data race!
    A++;
}

#pragma omp parallel
{
    for (int x = 0; x < len; x++) {
        *(A+x) = x;
    }
}

#pragma omp parallel
{
    for(int x = omp_get_thread_num(); x < len; x += omp_get_num_threads()) {
        A[x] = x;
    }
}
```

Spring 2013, #4:

- Are any of these implementations correct?
- Which implementation is fastest?
- How does each compare to a serial implementation?

OpenMP

```
#pragma omp parallel for
for (int x = 0; x < len; x++) {
    *A = x; // data race!
    A++;
}
```

```
#pragma omp parallel // repeated work
{
    for (int x = 0; x < len; x++) {
        *(A+x) = x;
    }
}
```

```
#pragma omp parallel
{
    for(int x = omp_get_thread_num(); x < len; x += omp_get_num_threads()) {
        A[x] = x;
    }
}
```

Spring 2013, #4:

- Are any of these implementations correct?
- Which implementation is fastest?
- How does each compare to a serial implementation?

OpenMP

```
#pragma omp parallel for
for (int x = 0; x < len; x++) {
    *A = x; // data race!
    A++;
}

#pragma omp parallel // repeated work
{
    for (int x = 0; x < len; x++) {
        *(A+x) = x;
    }
}

#pragma omp parallel
{
    for(int x = omp_get_thread_num(); x < len; x += omp_get_num_threads()) {
        A[x] = x; // It works, but is it fast?
    }
}
```

Spring 2013, #4:

- Are any of these implementations correct?
- Which implementation is fastest?
- How does each compare to a serial implementation?

OpenMP

Spring 2014 Final #4

```
void outer_product(float* dst, float *x, float *y, size_t n) {
   for (size_t i = 0; i < n; i += 1)
       for (size_t j = 0; j < n; j += 1)
           dst[i*n + j] = x[i] * y[j];
}
```

Consider the unparallelized outer product code, above.

If x and y are two column vectors, $O := xy^T$, such that $O_{ij} = x_i y_j$

OpenMP

Spring 2014 Final #4a

```
void outer_product(float* dst, float *x, float *y, size_t n) {

  ____________________________________
    for (size_t i = 0; i < n; i += 1)

        ____________________________________
        for (size_t j = 0; j < n; j += 1)

          ____________________________________
            dst[i*n + j] = x[i] * y[j];
}
```

Insert openMP directives in the blanks to best parallelize the code!

You can use as little/many of the blanks as you want

OpenMP

Spring 2014 Final #4a

```
void outer_product(float* dst, float *x, float *y, size_t n) {
 #pragma omp parallel for
    for (size_t i = 0; i < n; i += 1)
        // if you inserted here, it wouldn't help much...
        for (size_t j = 0; j < n; j += 1)
         // no need for a critical section
            dst[i*n + j] = x[i] * y[j];
}
```

- Remember that **for** means that openMP automatically breaks up the threads
- We don't need to worry about any critical sections because each section is independent! Each entry in memory is only edited by one thread.

Other Issues

- Cache coherence becomes hard
- False Sharing: Cache invalidations when accessing disjoint variables
    - Result of block-based caching

Data Race, Atomics (Implementation of #pragma omp critical)

- Consider two memory access to same location
  - Synchronize access to bring determinism
- Atomicity - Read, Modify, Write in one inst
  - Commonly done with locks
- RISC-V
  - **Load Reserved** and **Store Conditional** (lr, sc)
    sc returns 1 if address unchanged (fail), 0 otherwise (success)
  - Alternative: Atomic Swap
    Amoswap: store and load previous value

Do it poorly? Deadlock

Data race - example

Consider two threads, each executing:
(Note: xdata and xdatap are register names)

```
lw xdata (xdatap)
addi xdata xdata 1
sw xdata (xdatap)
```

Assume all memory accesses happen in local program order, but make no assumptions about the global ordering.
What are the possible final values of M[xdatap]?

Mutual Exclusion using amoswap

```
_____________________
Bne _________________          # Acquire lock

Lw xdata, (xdatap) # Critical section begin
Addi xdata xdata 1
Sw xdata, (xdatap)   # Critical section end

_____________________          # Release lock
```

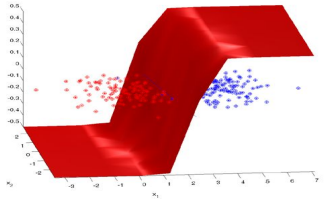
Mutual Exclusion using amoswap

```
Acq_spin: Amoswap.w.aq xlock, xone, (xlockp)
Bne xzero, xlock, acq_spin   # Acquire lock

Lw xdata, (xdatap) # Critical section begin
Addi xdata xdata 1
Sw xdata, (xdatap)   # Critical section end

Sw xzero, (xlockp)              # Release lock
```

MapReduce

Let's use map-reduce to make a logistic regression (machine learning) classifier!

To make a logistic regression classifier,
we must determine the value of a vector: w



w will specify a separating plane, separating our two classes (perhaps spam v. non-spam, etc.). The details of how we proceed using w are not too important -- for this question we just wish to compute it.

To compute w, it is in iterative process: first you compute a weighted term for each point, and add them together to get a gradient. Then, you subtract this gradient from w, and repeat!

MapReduce

Your job: compute w using Spark!

The algorithm: for ITERATIONS, compute the weighted term for each point, then sum them all together. Subtract this from w and repeat.

Here's some starter code, you can assume that you have a pre-existing w as well as a function to compute the weighted term for a specific point:

```
# current separating plane
for i in xrange(ITERATIONS):
 gradient = 0
 for p in points:
  gradient += computeWeightedTerm(p)
 w -= gradient

print "Final separating plane: %s" % w
```

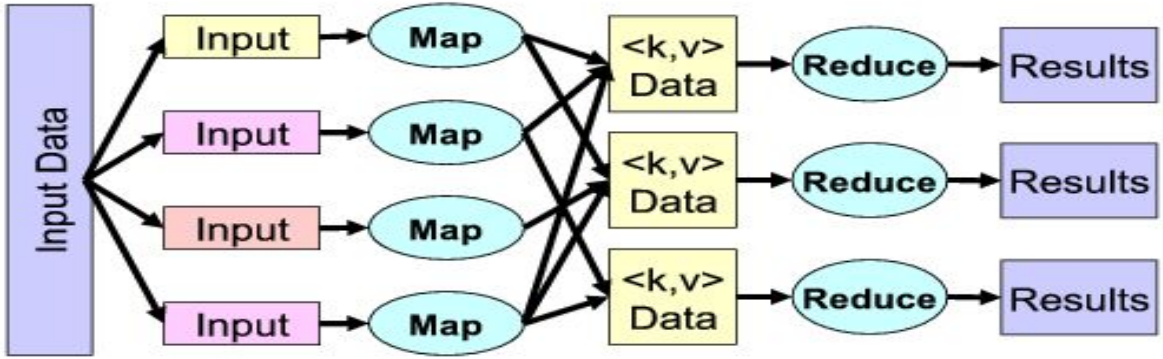
MapReduce

```
# current separating plane
for i in xrange(ITERATIONS):
    gradient = points.map(computeWeightedTerm)
                     .reduce(lambda a, b: a + b)
    w -= gradient

print "Final separating plane: %s" % w
```

Look how short the code is! There are other nice examples on: spark.apache.org

MapReduce (SIMD/SPMD)

MapReduce:



- Map: Initial Load Balancing and Partial Computation
- Shuffle: Move data around
- Reduce: Aggregate results by key
- Directed by a master node

MapReduce

```
# Word count in Spark -- usually ~100 lines in Hadoop!

file = spark.textFile("hdfs://...")
counts = file.map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
```

- Spark abstracts away the map() and reduce() functions into higher level functions operating on **Resilient Distributed Datasets** (RDDs)

Break!

Please fill out this feedback form:

hkn.mu/feedback

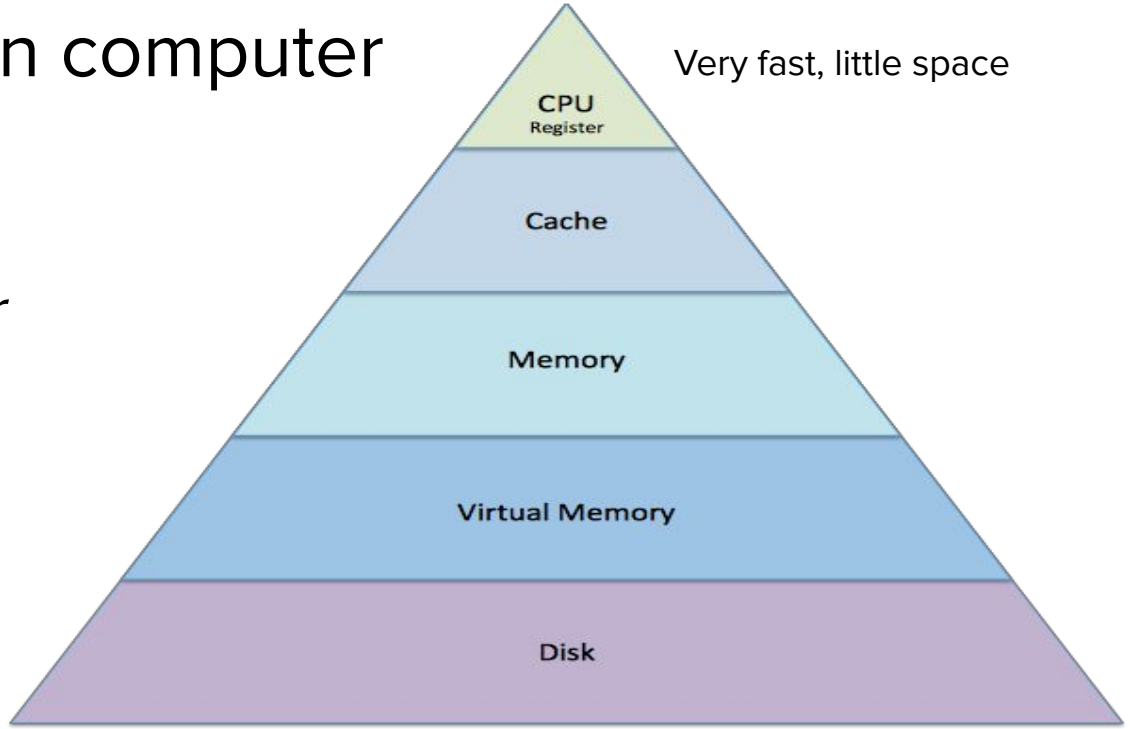
Caches

why couldn’t the CS student afford a new laptop? they didn't have enough cache

Memory Hierarchy

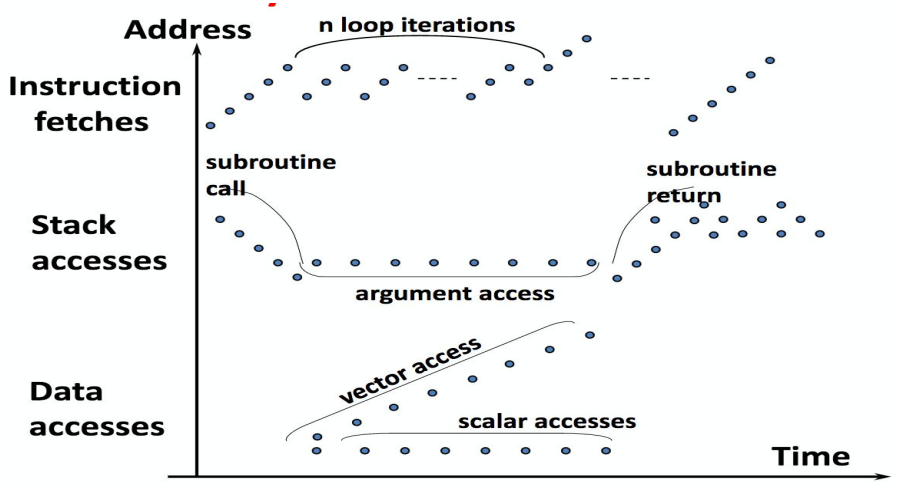
Nothing is ever free in computer engineering!

You can favor either size or speed, but never both (due to cost).



Locality

Temporal (Reuse) and Spatial (Nearby)

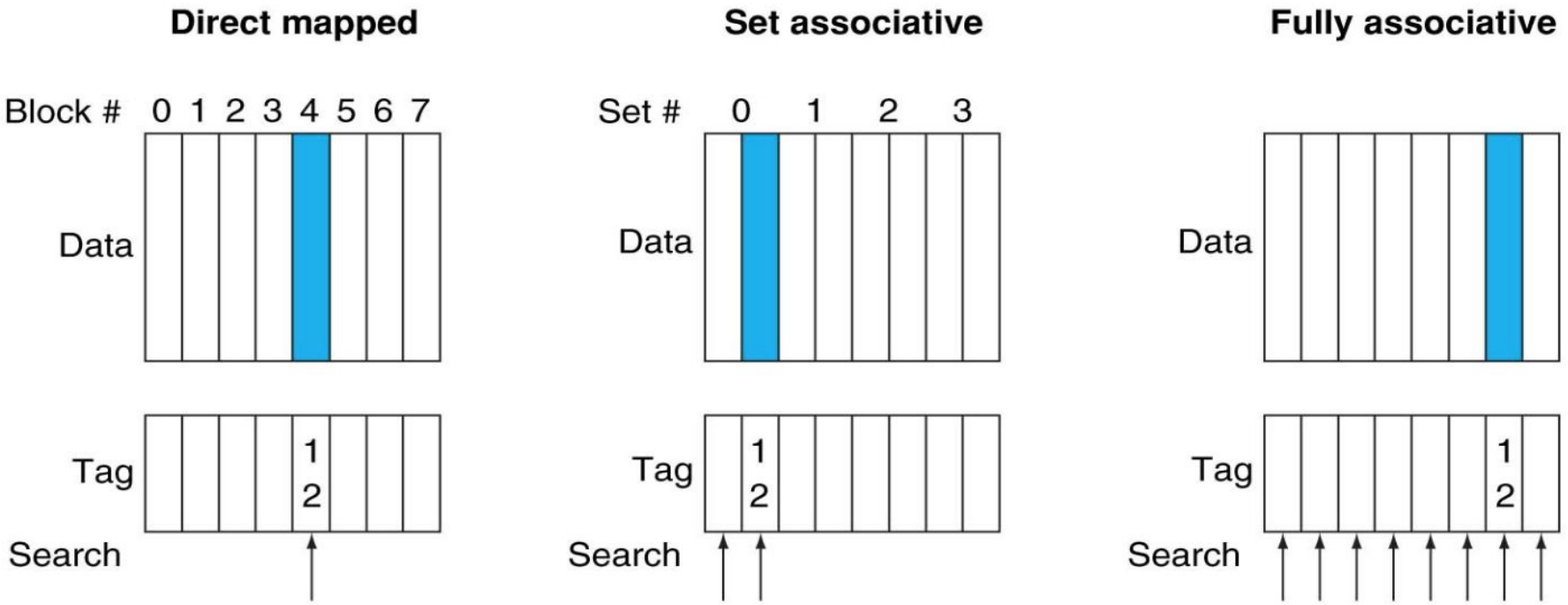


Applies to instruction and data memory alike

Caches

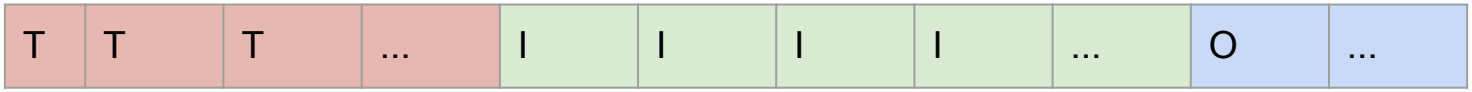
- Often, your programs exhibit temporal and spatial locality
- Make a cache in between CPU and DRAM
- Pull one piece of data in, also pull in a bunch of stuff physically near it in memory
- In the likely event of CPU asking for data at a location close to the first thing, the cache can produce it right away
- Cache INPUT: a memory address
- Cache OUTPUT: the data stored at that memory address

Caches


Direct mapped
Block # 0 1 2 3 4 5 6 7
Data
Tag
1
2
Search
Set associative
Set # 0 1 2 3
Data
Tag
1
2
Search
Fully associative
Data
Tag
1
2
Search

Caches

For a location in memory (usually 32 bits):



Tag bits: Address Identification in Cache

- Size is how much bits left after Index, Offset used
- # of bits in an address - (# of index bits + # of offset bits)

Index bits: Which set in cache

- $\log_2$(cache size/block size/associativity) ⇔ $\log_2$(number of blocks/associativity)

Offset bits: Which byte within the block

- $\log_2$(block size or row size)

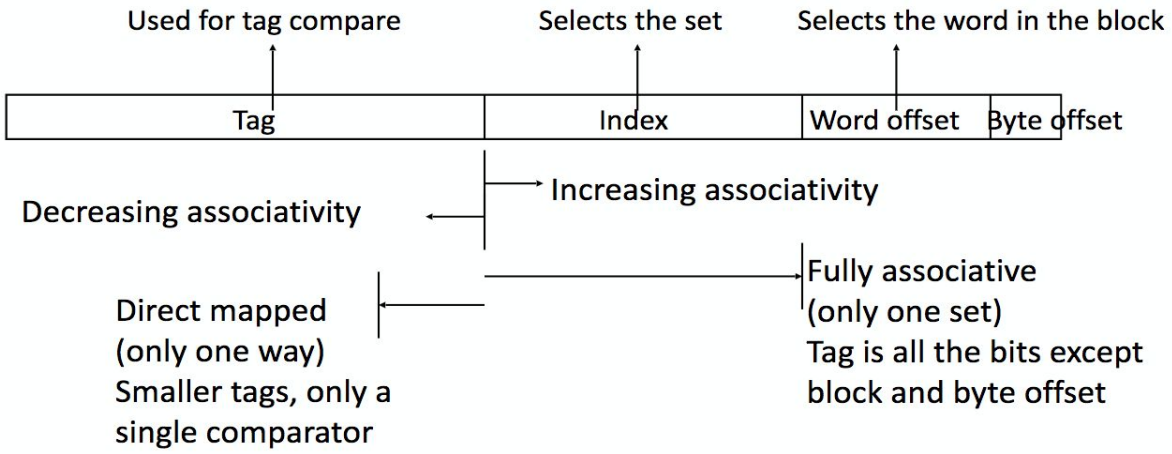
Caches

- # bits in a cache line = tag bits + data + dirty bit + valid bit
  - Generally! Not always true
  - Can have other metadata besides dirty / valid bits
  - i.e. cache design influences how many bits are in a cache line
- Dirty bit: Used in write-back caches (will get to later)
  - 1 if cache line has changed since it was pulled in, but that change is not yet reflected in main memory
- Valid bit: 1 if valid data exists, 0 otherwise/on-start
- Line count depends on associativity

Caches

Capacity = Associativity * Set Count * Block Size

- Direct Map? 1-way set associative
- Fully Associative? No Index bits



Direct Mapped Caches

```
AddVectors(uint8_t* A, uint8_t* B, uint8_t* C, int n) {
    for (int i = 0; i < n; i++)
        C[i] = A[i] + B[i];
}
```

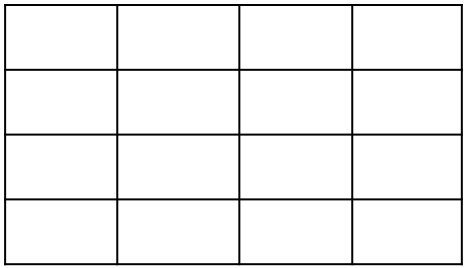
sizeof(uint8_t) = 1 byte
32 bits memory address
4 KB Cache
10 Offset bits
n is power of 2 and is much greater than cache size
block aligned

If the cache is direct mapped, what is the lowest and highest hit rate?

Direct Mapped Caches

10b Offset ➜ Block size of 2^10 or 1024 bytes
# of blocks = cache size / block size = 4KB/1KB = 4
4 Blocks ➜ 2b Index
32b - 10b Offset - 2b Index = 20b Tag

TTTTTTTTTTTTTTTTTTTTIIOOOOOOOOOO



Lowest Hit Rate: 0% when A, B, C have same index but different tags

- i.e. All three accesses don't use cache information

Highest Hit Rate: 2/3 when A, B, C are the same exact array

- i.e. All three accesses use the exact same information, so only first access is a miss (compulsory)

N-Way Set Associative Caches

- **N** is typically a power of 2
  - If you know the power of 2, then you can figure out the **n (bit amount)** where $2^{(n)}$ = **N**
- Form sets of **N** blocks
  - All blocks in the same set correspond to one index
  - [number of blocks / **N**] sets
- TIO bits change
  - New # tag bits = Old # tag bits + **n**
  - New # index bits = Old # index bits - **n**
  - Offset stays the same

Average Memory Access Time (AMAT)

- AMAT = Time for a hit + Miss rate * Miss Penalty
  - Miss penalty is the additional time for a cache miss
- In a multi-level Cache this becomes a recursive formula
- For L1, L2, and DRAM
  - AMAT = (L1 hit time) + (L1 miss rate) * (L1 miss penalty)
  - L1 miss penalty = (L2 hit time) + (L2 miss rate) * (L2 miss penalty)
  - L2 miss penalty = DRAM access time

Cache Replacement Policies

- Only relevant for associative caches
- Random Replacement
- Least Recently Used (LRU)
    - One way can be to keep explicit last-use scheme

Caches

Miss Rates

Local at level n cache?

# of misses at level n / # of misses at level n-1 (= # of accesses to level n)

Globals?

# of misses at level n / total # of accesses

Caches

How do hit time, miss rate, penalty change if:

- More Associativity
- More Entries (Set * Assoc)
- Larger Blocks

Caches

How do hit time, miss rate, penalty change if:

- More Associativity
  - Hit Time log increase, Miss Rate decrease, Penalty same
- More Entries (Set * Assoc)
  - Hit Time increase, Miss Rate decrease, Penalty same
- Larger Blocks
  - Hit Time same, Miss Rate decreases then increases, Penalty increase

Write Policy

On cache hits, one can:

- **Write-through**
  - Update both cache and main memory synchronously
  - Mitigate main memory write through buffer
  - Simple, predictable timing, reliable
- **Write-back**
  - Update cache, but don’t affect main memory yet
  - Write to main memory when cache block is evicted
    - Dirty bit
  - Reduce write traffic

Write Policy

On cache miss, one can:

- **No Write Allocate**
  - Write only to main memory
  - i.e. If block not in cache, don’t load it in
- **Write Allocate** (fetch-on-write)
  - If block not in cache, load it in first

Common combinations

- Write through, no write allocate
- Write back with write allocate

3* Cs of Caches

Compulsory

- Accessing the data **for the first time**

Conflict

- Wouldn’t have happened with a fully associative cache

Capacity

- Wouldn’t have happened with an infinitely **large** cache
- As time goes on, all misses become capacity misses

(Coherence only matters for multiprocessing)

3 Cs

Cache that holds 8 blocks, direct mapped, each block is 1 word (4 bytes)

```
#define LARGE 2*(cache size in words)  // so LARGE is 16
int a[LARGE];
int sum = 0;
for (int z = 0; z < 2; z++) // LOOPS 1
    for (int x = 0; x < LARGE; x++)
        sum += a[x];

for (int z = 0; z < 2; z++) // LOOPS 2
    for (int x = 0; x < LARGE; x += 8)
        sum += a[x];
```

What cache misses happen if z = 0? z = 1?

3 Cs

Cache that holds 8 blocks, direct mapped, each block is 1 word (4 bytes)

```
#define LARGE 2*(cache size in words)  // so LARGE is 16
int a[LARGE];
int sum = 0;
for (int z = 0; z < 2; z++) // LOOP 1
    for (int x = 0; x < LARGE; x++)
        sum += a[x];

for (int z = 0; z < 2; z++) // LOOP 2
    for (int x = 0; x < LARGE; x += 8)
        sum += a[x];
```

For LOOP 1, when z = 0, all accesses are **compulsory** misses
For LOOP 1, when z = 1, all accesses are **capacity** misses

For LOOP 2, when z = 0, all accesses are **capacity** misses
For LOOP 2, when z = 1, all accesses are **conflict** misses

Cache Coherency (MOESI)

Cache Coherency

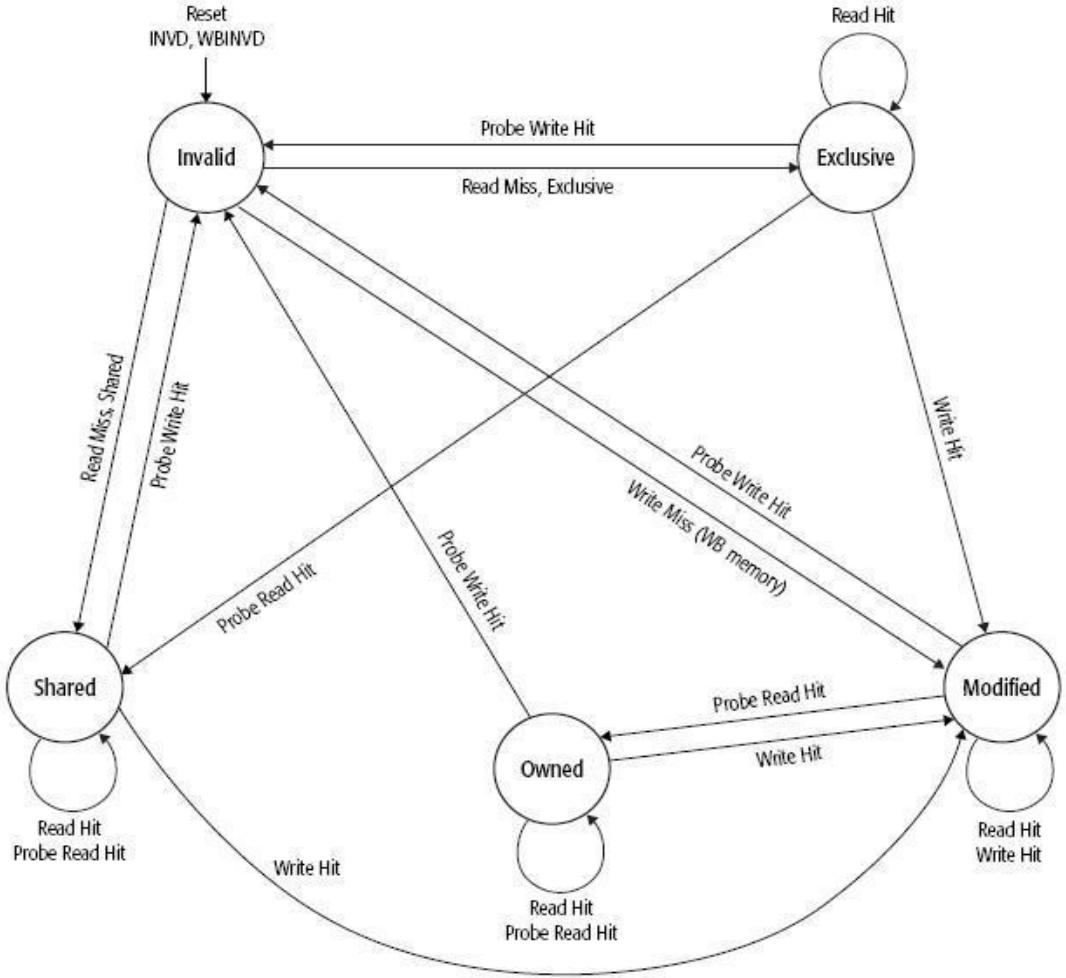
- Multiprocessor caches, each processor has its own local cache
- Issue with this, need a way for multiple processors to keep memory **coherent** (consistent)
- Have caches communicate over an interconnection network with a cache coherency protocol
    - Each processor’s cache keeps track of the coherency status of each **block**

MOESI Protocol

- **Modified** - Up-to-date data, changed (dirty), memory out-of-date, no other cache has copy
  - Only cache that supplies data on read instead of going to memory
- **Owned** - Up-to-date data, other caches may have a copy (they must be in shared state)
- **Exclusive** - Up-to-date data, no other cache has a copy, memory up-to-date
  - Supplies data on read instead of going to memory
  - Avoids writing to memory if block replaced
- **Shared** - Up-to-date data, other caches may have a copy
- **Invalid** - Data is not up-to-date, must fetch from memory to access

*See Fall 2016 Final for Practice Problem

MOESI


Reset
INVD, WBINVD
Read Hit
Invalid
Probe Write Hit
Exclusive
Read Miss, Exclusive
Read Miss, Shared
Probe Write Hit
Probe Write Hit
Write Miss (WB memory)
Write Hit
Probe Read Hit
Probe Write Hit
Shared
Modified
Probe Read Hit
Owned
Write Hit
Read Hit
Probe Read Hit
Write Hit
Read Hit
Write Hit
Read Hit
Probe Read Hit

Performance (of computers)

Performance

Latency (Time/Task) and Bandwidth (Tasks/Time)

$$\frac{CPU\ Time}{Program} = \frac{Clock\ Cycles}{Program} * \frac{Time}{Clock\ Cycle}$$

$$= \frac{Instructions}{Program} * \frac{Avg\ Clock\ Cycle}{Instruction} * \frac{Time}{Clock\ Cycle}$$

Out of Algorithms, Compilers/PL, and ISAs, only ISAs affect the Clock Rate

Feedback

We would like your feedback on this review session, so that we can improve for future review sessions.

If you would like to provide suggestions, complaints or comments, please go to:

hkn.mu/feedback

(we’re not done yet)

Operating Systems

Terminology

- A “**program**” refers to a set of instructions to run
- A “**process**” is a separate instance that executes a program, each process is isolated from the others
  - Every process has one or more **threads**
  - Each thread can run independently and share certain resources (e.g., heap, static, etc.)
- A datapath as seen so far represents a single CPU **core**, or **processor**
  - Each core can execute one thread at a time
- CPUs generally have multiple cores, allowing many threads to run simultaneously

What does an OS do?

Operating systems (OS) manage hardware resources - think memory, CPU, networking, connected devices

- Acts as the “middleman” that bridges software applications to hardware
  - No “standard” for many things - that’s why we have drivers!
- Linux, Windows, macOS, but also Android and iOS
  - Provides abstractions like filesystems and processes

How exactly does the OS manage other processes? Want to run lots of different programs all on the same computer, at the same time

- Solution: set up one process whose main purpose is to coordinate other processes and run it on startup (known as the Operating System)
- Loads startup program known as the **BIOS** which checks if the computer can run
  - BIOS loads the OS from memory and then transfers control to the OS
- From then on, the OS is responsible for running other programs

What does an OS Do?

- **Illusionist:**
  - Provides easy to use abstractions of resources
  - Ex: Gives the illusion of its own “empty” address space with near-unlimited space
- **Conductor:**
  - Receives commands and assigns resources to tasks
- **Referee:**
  - Manage protection and isolation of all resources
  - Terminate any processes that crash or exceed what its allowed to do without crashing the entire system
  - Ensures all programs get “fair” access to resources

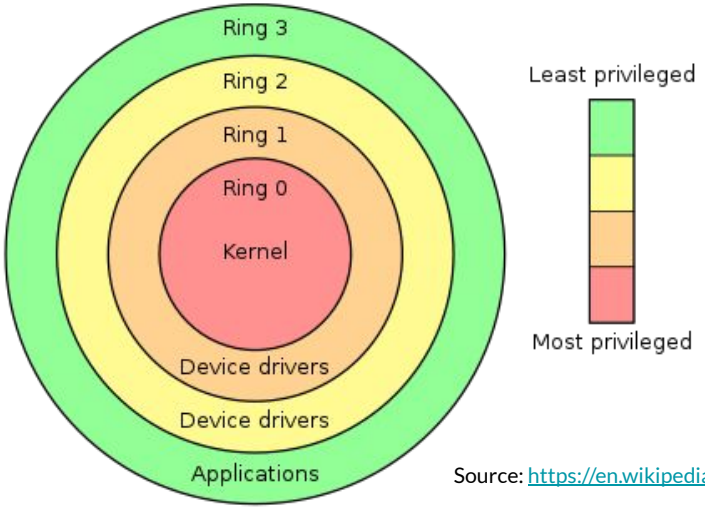
Process Memory Usage

- Always assumed programs had its own exclusive access to memory
  - Don’t want to have to step around other processes’s used memory, and want to consistently have access to the same resources
  - RAM is limited, nowadays typically 16 or 32 GiB for a midrange PC
    - We don’t want to assign massive chunks of memory for each process
    - Use translation to give *used* “pages” of data an actual spot on RAM (e.g., stack, heap)
  - OS will handle this! More in the next section (virtual memory/VM)
- Referee will stop processes from trying to access other processes’s memory, as well as control access to I/O!

Dual Mode Operation

Hardware provides two modes.

- **Kernel/supervisor/privileged mode** has the most privileges (kernel + OS).
- **User mode** prohibits certain operations.
- Restricted user mode is important to make sure user process cannot maliciously corrupt the system.



Source: https://en.wikipedia.org/wiki/Protection_ring

System Calls

- What if a process wants to access I/O?
  - It can use a “system call”, or “syscall”
- User program can transfer control back to OS (kernel mode) for syscall, requesting a certain operation (such as `read` or `printf`)
  - OS will check if the syscall is allowed, then execute it if so

Context Switches

- As part of OS's job as conductor, need to give different processes "fair" use of the processor
  - Need to switch threads if the current thread is taking a very long time
  - We want threads to be "paused," and able to fully resume operation after being switched back
- Problem: every processor only has one set of registers!
- Requires a "context switch":
  - OS takes control of old thread
  - Saves register values for old thread in memory
  - Loads register values for new/target thread
  - OS transfers control to new thread
- Takes a long time, so do this when waiting on something else
  - e.g., writing to a file

Virtual Memory

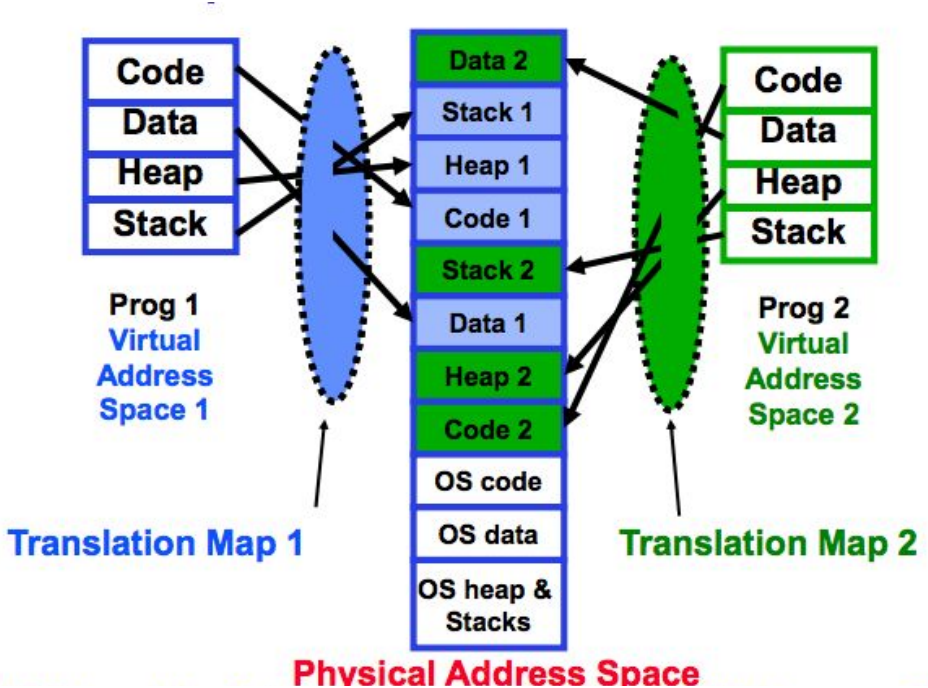
Physical Memory

- Actual memory on computer
- Primary memory that is only accessible by **kernel** (& firmware)
- Denoted with **physical address space**
- *Primary storage* (main/internal memory)
  - Fast, but expensive
  - Generally **volatile** (loses data on power loss)
  - Directly accessible by CPU
  - Examples: registers, cache, DRAM
- *Secondary storage* (auxiliary/external mem.)
  - Cheap, but slow
  - Generally **non-volatile** (retains data on power loss)
  - Not directly accessible by CPU
  - Examples: hard drive (disk), SSD, flash drive, CD/DVD/BD

Virtual Memory (“VM”) - Concepts

- Primary memory visible to **user programs**
- Denoted with **virtual address space**
- Abstraction barrier that gives user programs the illusion of infinite memory
    - Hides physical memory from general programs (protection)
    - Hardware-accelerated (on most modern systems)
    - Cannot be disabled (on most modern systems)
    - Has nothing to do with your hard drive or SSD
    - Enables memory hierarchy
- **Memory Management Unit (MMU)** - is a hardware accelerator that handles all memory references, mainly converting virtual address space to physical address space

Virtual Memory - Brief Overview



Virtual Memory - Implementation

- Goal: separate programs' memory spaces

- Two common (but *orthogonal*) approaches:
  - Segmentation: split mem. into segment base + offset
    - Less popular nowadays
    - Issue of external fragmentation

  - **Paging**: split mem. into conveniently-sized blocks (called pages)
    - In 61C we focus on paging

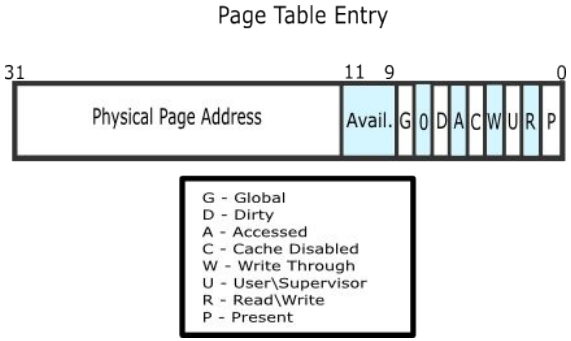
Paging

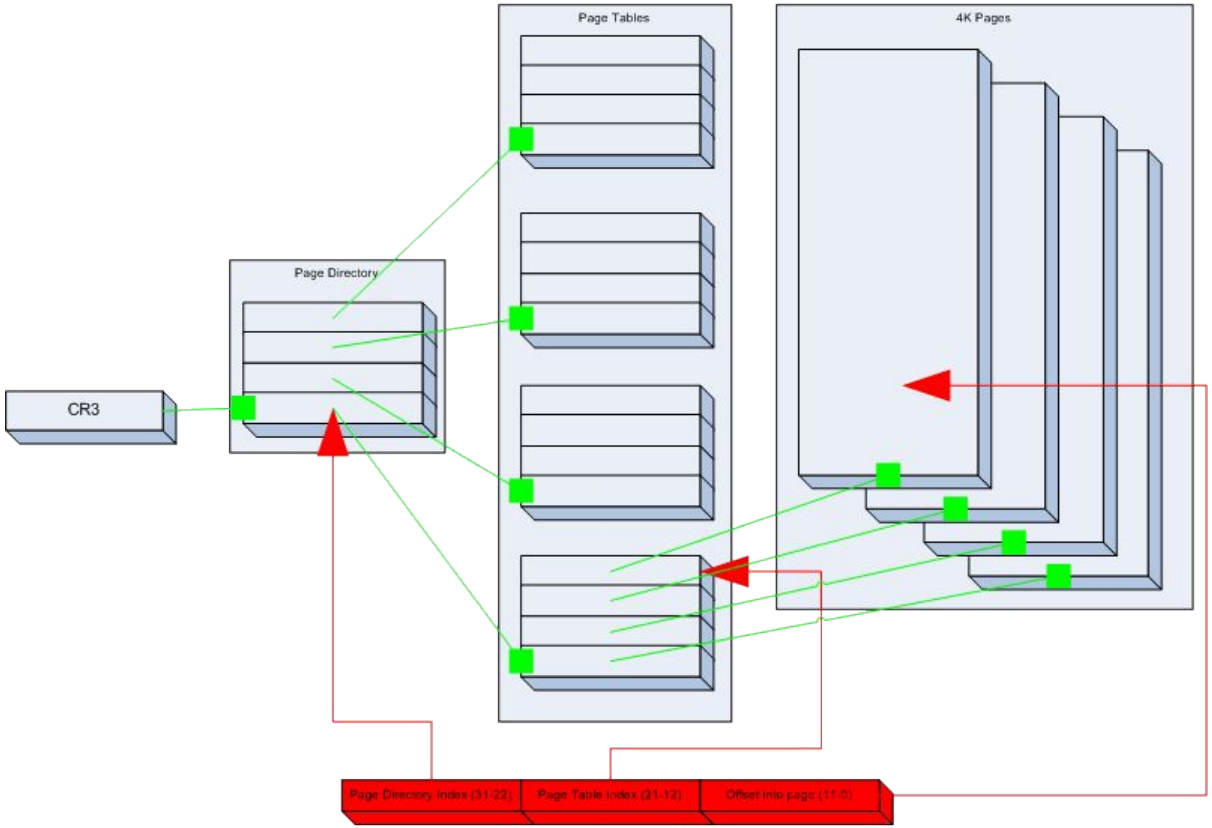
- Divide memory space into blocks aka “**pages**” (e.g. 4KiB)
- Treat entire page as a single unit of memory (attributes *uniform* throughout each page)
    - Downside: internal fragmentation
- Goal: find an efficient & practical way to represent attributes and permissions
    - Then tell the CPU to use these “page tables” in memory for address translation
- **Translation Lookaside Buffer (TLB)**
    - Reading page tables from DRAM is slow and page table lookups happen often!
    - Dedicated cache for **page table entries (PTEs)**
    - Usually fully-associative; therefore usually small

Paging

- Problem: Page tables are wasteful. Why?
  - 4GiB of RAM ÷ 4KiB pages ≃ 1M pages
  - *Even 4 bytes of meta-information per page* would use 4MiB of memory *per process*
  - 256 processes use 1GiB of RAM *just* for page tables!
- Better idea? Hierarchy (add indirection)
  - Sub-divide each “large page” into “smaller pages” *when necessary*
  - Separate page tables for each level
  - Space: massive improvement
  - Time: small penalty

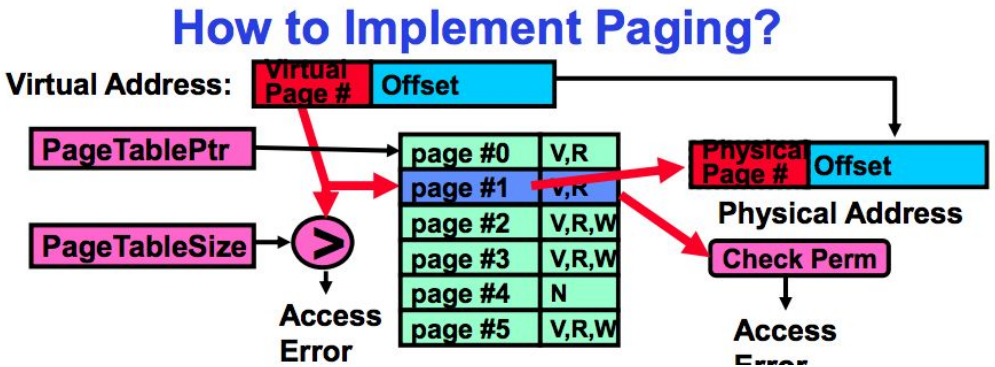
Paging in x86





wiki.osdev.org/Paging

Diagram



Virtual address (VA): What your program uses

Virtual Page Number	Page Offset

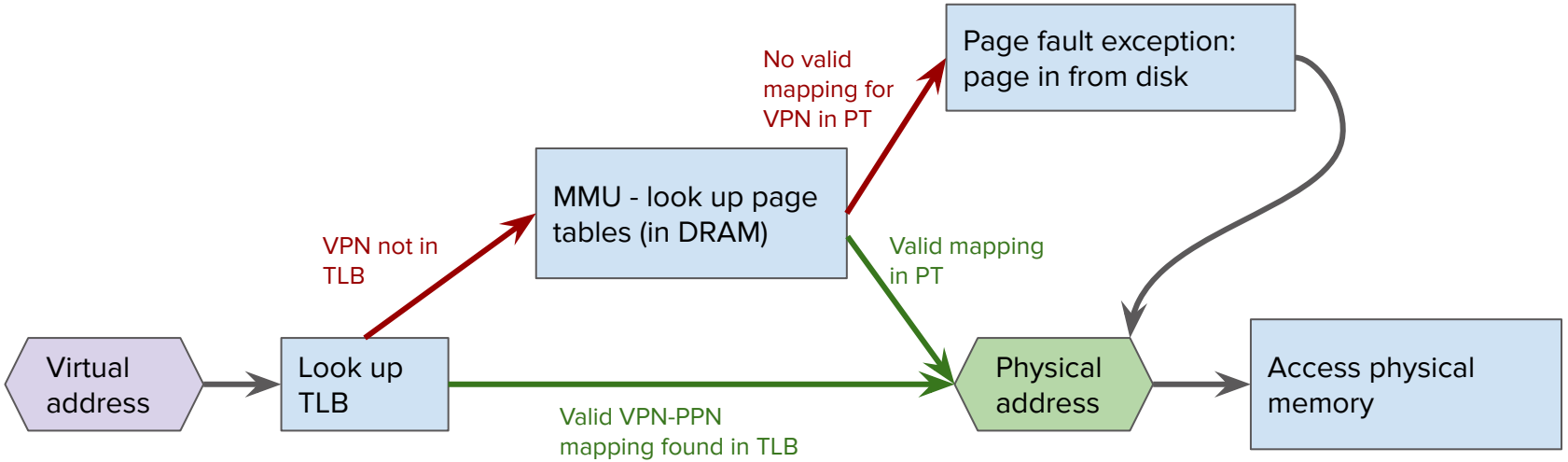
Physical address (PA): What actually determines where in memory to go

Physical Page Number	Page Offset

VPN bits = $\log_2$(VA size / page size)
PPN bits = $\log_2$(PA size / page size)
Page offset bits = $\log_2$(page size)

Bits per row of PT: PPN bits + valid + dirty + R + W

Virtual Address Translation



Question

What relationship must hold true between the size of physical address space, P, and the size of virtual address space, V?

1. None
2. P > V
3. P = V
4. P < V

Answer

What relationship must hold true between the size of physical address space, P, and the size of virtual address space, V?

1. **None**
2. P > V
3. P = V
4. P < V

Question

Spring 2013 #F2:

For the following questions, assume the following:

- 32-bit virtual addresses
- 1 MiB pages
- 512 MiB of physical memory with LRU page replacement policy
- Fully associative TLB with 32 entries and an LRU replacement policy
- Single-level page table

Question

Spring 2013 #F2:

For the following questions, assume the following:

- 32-bit virtual addresses
- 1 MiB pages
- 512 MiB of physical memory with LRU page replacement policy
- Fully associative TLB with 32 entries and an LRU replacement policy
- Single-level page table

➜ ***Before*** **we start, let's determine the bit-breakdown for virtual and physical addresses.**

Question: VPN

Spring 2013 #F2:

For the following questions, assume the following:

- 32-bit virtual addresses
- 1 MiB pages
- 512 MiB of physical memory with LRU page replacement policy
- Fully associative TLB with 32 entries and an LRU replacement policy
- Single level page table

Virtual Addresses:

- offset bits = $\log_2$(size of page in bytes) = $\log_2(2^{20})$ = 20 bits (just like cache offset!)
- VPN bits = $\log_2$(size of virtual memory / size of page in bytes)
  = $\log_2(2^{32}/2^{20})$ = 12 bits (just like cache index!)

Question: PPN

Spring 2013 #F2:

For the following questions, assume the following:

- 32-bit virtual addresses
- 1 MiB pages
- 512 MiB of physical memory with LRU page replacement policy
- Fully associative TLB with 32 entries and an LRU replacement policy
- Single level page table

Virtual Addresses:

- Offset bits = 20 bits, VPN bits = 12 bits

Physical Addresses:

- Offset bits = always the same as virtual pages! → 20 bits (just like cache offset!)
- PPN bits = $\log_2$(size of physical memory / size of page in bytes)
  = $\log_2(2^{29}/2^{20})$ = 9 bits (just like cache index!)

Question: VPN and PPN

Spring 2013 #F2:

For the following questions, assume the following:

- 32-bit virtual addresses
- 1 MiB pages
- 512 MiB of physical memory with LRU page replacement policy
- Fully associative TLB with 32 entries and an LRU replacement policy

trick to note: VPN + Offset bits = virtual address bits, same for physical memory...

Virtual Addresses:

- Offset bits = 20 bits, VPN bits = 12 bits

Physical Addresses:

- Offset bits = 20 bits, PPN bits = 9 bits

Question: Part a)

Spring 2013 #F2:

For the following questions, assume the following:

- 32-bit virtual addresses
- 1 MiB pages
- 512 MiB of physical memory with LRU page replacement policy
- Fully associative TLB with 32 entries and an LRU replacement policy
- Single level page table

alright, now let's dig into it:

a) How many entries does a page table contain?

Answer: Part a)

Spring 2013 #F2:

For the following questions, assume the following:

- 32-bit virtual addresses
- 1 MiB pages
- 512 MiB of physical memory with LRU page replacement policy
- Fully associative TLB with 32 entries and an LRU replacement policy
- Single level page table

a) How many entries does a page table contain?

- The number of VPN bits tell you how many virtual page translations we can store in the page table at any given time
  - Just like the index bits of a cache
  - The VPN gives you the index position into your page table
  - Need a mapping from every VPN to its PPN (or “invalid”)

Therefore, 12 VPN bits → **2^{12} (virtual) page entries!**

(remember, page table is from VPN to PPN, so we look at VPN bits)

Question: Part b)

Spring 2013 #F2:

For the following questions, assume the following:

- 32-bit virtual addresses
- 1 MiB pages
- 512 MiB of physical memory with LRU page replacement policy
- Fully associative TLB with 32 entries and an LRU replacement policy
- Single level page table

b) How wide must the page table base register be?

Answer: Part b)

Spring 2013 #F2:

For the following questions, assume the following:

- 32-bit virtual addresses
- 1 MiB pages
- 512 MiB of physical memory with LRU page replacement policy
- Fully associative TLB with 32 entries and an LRU replacement policy
- Single level page table

b) How wide must the page table base register be?

- **First, remember what the page table base register is:** the page table must live in physical memory somewhere, so the page table base register tells you where to find it!
  - Since the page table lives in *physical memory*, we must have a *physical address* to find it. Therefore, our page table base register must hold a physical address, which is (at least) offset (20) + PPN (9) = **29 bits wide.**

Question: Part c)

Spring 2013 #F2:

```
int histogram[MAX_SCORE];
void update_hist(int *scores, int num_scores) {
     for (int i = 0; i < num_scores; i++)
          histogram[scores[i]] += 1;
}
```

Assume that only the code and the two arrays take up memory, ALL of code fits in 1 page, the arrays are page-aligned (start on page boundary), the page table is already in DRAM, and this is the only process running

c) If update_hist were called with num_scores = 10, how many page faults can occur in the worst-case scenario?

Some Tips:

- Remember the sequential memory layout of arrays
- Don't forget to include any code page faults
- This is the *worst* case. Be evil!

Answer: Part c)

Spring 2013 #F2:

```
int histogram[MAX_SCORE];
void update_hist(int *scores, int num_scores) {
     for (int i = 0; i < num_scores; i++)
          histogram[scores[i]] += 1;
}
```

Assume that only the code and the two arrays take up memory, ALL of code fits in 1 page, the arrays are page-aligned (start on page boundary), the page table is already in DRAM, and this is the only process running

c) If update_hist were called with num_scores = 10, how many page faults can occur in the worst-case scenario?

Page faults = 12

- Code -- **1:** In this case we're okay because we have ALL code in the page table/TLB, including the code that called this function
- Scores -- **1:** It's an array of ints, so with 10 ints that's only 40 B which << 1 page
- Histogram -- **10:** In the spirit of being evil for the worst case, what if each entry of scores caused histogram to skip a page? We'd page fault all 10 accesses :(

Question: Part d)

Spring 2013 #F2:

```
int histogram[MAX_SCORE];
void update_hist(int *scores, int num_scores) {
     for (int i = 0; i < num_scores; i++)
          histogram[scores[i]] += 1;
}
```

Assume that only the code and the two arrays take up memory, ALL of code fits in 1 page, the arrays are page-aligned (start on page boundary), and this is the only process running

d) In the best-case scenario, how many iterations of the loop can occur before a TLB miss? You can leave your answer as a product of two numbers.

Some Tips:

- Think about the size of your TLB, and what memory accesses we have! (code, score, histogram)

Answer: Part d)

Spring 2013 #F2:

```
int histogram[MAX_SCORE];
void update_hist(int *scores, int num_scores) {
     for (int i = 0; i < num_scores; i++)
          histogram[scores[i]] += 1;
}
```

d) In the best-case scenario, how many iterations of the loop can occur before a TLB miss? You can leave your answer as a product of two numbers.

Iterations = $30*2^{18}$

- Firstly, your TLB has 32 entries, one per page.
  - You need at least one for code, scores, and histogram.
- Now, continuing being nice, what if score[i] was always the same number?
  - Then we'd only need one page for histogram (and only one page for code)
- This leaves 30 pages to iterate through the score array!
  - Each page can hold (2^{20} / 2^{2}) = 2^{18} ints, so we can iterate for **$30*2^{18}$ i values**

I/O, Interrupts, and DMA

Interrupts/Exceptions

- **Interrupts and Exceptions** - “An unscheduled event that disrupts program execution”
  - Hardware (async)
    - Ex: I/O service, timer expiration, hardware failure, etc.
  - Software (sync)
    - Ex: div-by-zero, overflow, invalid mem acc., page fault
- Interrupts = Exceptions, except:
  - Exceptions come from within or outside processor
  - Interrupts are only from outside processor (external events)
- **Trap** - Services interrupt/exception
  - Interrupts generally are expensive
- Interrupts enable **context switching**
  - Run multiple programs
- Choose interrupt when event is asynchronous, urgent, infrequent

Polling

- **Polling** - When the CPU repeatedly checks the status of the device that it's communicate with or controlling
- The CPU polls every N seconds, called the **polling interval**
- But every poll can take many clock cycles
  - Can be wasteful
  - Not as efficient as interrupt driven I/O since it wastes CPU cycles
- Choose polling when event is synchronous, not urgent, frequent

PIO vs DMA

- **Programmed I/O (PIO)** requires that the CPU actively transfer data
  - Memory mapped I/O is a form of PIO
  - CPU software uses instructions to access I/O address space to perform data transfer
- Instead, we introduce a hardware element called a **Direct Memory Access (DMA)** engine or controller
  - CPU gives DMA control of main memory and DMA handles all transfers
  - CPU initiates a transfer, then DMA handles transfer in background while CPU does other work

Memory Mapped IO

- RISCV uses **Memory Mapped I/O**
  - Use same address space to address both memory and I/O devices
  - We treat our input/output devices as special parts of memory
  - Deal with them using plain old **lw and sw**
  - Addresses are mapped to registers on the device
  - Usually broken up into status bits:
    - Ex: ready bit

Question

Consider a small CPU whose only job is to process data produced by a sensor. The sensor produces data at 10 Hz and the CPU runs at 1 GHz.

1) Assume we are using PIO. Should we use polling or interrupts?

Answer

Consider a small CPU whose only job is to process data produced by a sensor. The sensor produces data at 10 Hz and the CPU runs at 1 GHz.

1) Assume we are using PIO. Should we use polling or interrupts?

We should use interrupts since the frequency of incoming data is infrequent relative to the CPU frequency.

Question

Consider a small CPU whose only job is to process data produced by a sensor. The sensor produces data at 10 Hz and the CPU runs at 1 GHz.

2) Suppose it takes 500,000,000 cycles to read the data from the sensor. Discuss the tradeoffs of using PIO vs DMA.

Answer

Consider a small CPU whose only job is to process data produced by a sensor. The sensor produces data at 10 Hz and the CPU runs at 1 GHz.

2) Suppose it takes 500,000,000 cycles to read the data from the sensor. Discuss the tradeoffs of using PIO vs DMA.

- CPU @ 1 GHz, this means we'd spend 50% of the time copying with PIO
- What kind of computation are we doing?
  - Streaming: Probably ok to use PIO
  - Random access: Probably want DMA
- Energy (assume processing takes little time):
  - DMA: let power-hungry CPU idle
  - PIO: CPU has to be active most of the time

Backup

Disk access time

- 15000 Cylinders, 1 ms to cross 1000 Cylinders
- 7200 RPM
- Want to copy 1 MB, transfer rate of 200 MB/s
- 1 ms controller processing time

What’s the access time?

Disk access time

- 15000 Cylinders, 1 ms to cross 1000 Cylinders
- 7200 RPM
- Want to copy 1 MB, transfer rate of 200 MB/s
- 1 ms controller processing time

What's the access time?

- Seek = # cylinders/3 * time = 15000/3 * 1ms/1000 cylinders = 5ms
- Rotation = 1/((7200 rpm)*(1/60 min/s)) * ½ = 4.2 ms
- Transfer = Size / transfer rate = 1 MB / (200 MB/s) = 5 ms
- Controller = 1 ms
- Total = 5 + 4.2 + 5 + 1 = **15.2 ms**

Average Page Access Time (APAT)

- 15000 Cylinders, 1 ms to cross 1000 Cylinders
- 15000 RPM
- Transfer rate of 500 MB/s
- 1 ms controller processing time
- Assume standard page size = 4KB

Measurement	Value
P_T=prob of TLB miss	0.1
P_F=prob of a page fault when a TLB miss occurs	0.0002
P_D=prob page is dirty when replaced	0.5
T_T= time to access TLB	0 µs
T_M = time to access memory	1 µs
T_D = time to transfer a page to/from disk	10 ms = 10000 µs

APAT?

What's the access time?

Seek = # cylinders/3 * time = 15000/3 * 1ms/1000 cylinders = 5ms

Rotation = 1/((15000 rpm)*(1/60 min/s)) * ½ = 2 ms

Transfer = Size / transfer rate = (4 KB * 1/1000 MB/KB) / (500 MB/s) = .008 ms

Controller = 1 ms

Total disk = 5 + 2 + .008 + 1 = 8.008 ms = 8008 µs

$APAT = T_M + T_T + P_T * (T_M + P_F * (T_D * (1 + P_D)))$

$= 1\mu s + 0 + 0.1 * (1\mu s + 0.0002 * (8008\mu s * (1 + 0.5)))$ = **1.34 µs**

Data race - example

Consider two threads, each executing:

```
Lw xdata (xdatap)
Addi xdata xdata 1
Sw xdata (xdatap)
```

Assume all memory accesses happen in local program order, but make no assumptions about the global ordering.
What are the possible final values of M[xdatap]? Original +1, or +2

ECC and RAID:
Redundancy and Failure Recovery

Error Detection and Correction

- Memory Errors:
  - DRAMs use very little charge/bit
  - “Soft” Errors due to environmental factors
    - i.e. radiation
  - “Hard” Errors due to physical failures
    - i.e. Early SSDs had a very limited lifespan
    - Without correction, as density ⬆, likelihood of hardware failure ⬆

EDC: Parity

- Even Parity: number of set bits (1s) is even
- Add a parity bit, assigned a value such that the entire word has even parity
  - Equivalent: parity bit is 1 if the word starts out with odd parity
- Ex: Data we want to transmit: 1011
  - Encoded version: 1011**1**
- Allows us to detect (but not correct) any odd number of errors

ECC: Hamming Code

- Use more parity bits!
- Mechanical procedure for encoding:
  1. Number the bits (1 … n), left to right
  2. All bits numbered with a power of two are parity
  3. All other bit positions are data
  4. Which bits does each parity bit check?
     a. All bits with the corresponding bit set in the binary representation of their index
  5. Set parity bits to create even parity for each parity group

Hamming Code

From prev slide:

Which bits does each parity bit check?

a. All bits with the corresponding bit set in the binary representation of their index

Parity bit P1 = 2^0
Bit positions 1, 3, 5… => binary 001, 011, 101… => all have a 1 in their 2^0 position, so P1 checks them.

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	X		X		X		X		X		X		X		X		X		X		
	p2		X	X			X	X			X	X			X	X			X	X		…
	p4				X	X	X	X					X	X	X	X					X	
	p8								X	X	X	X	X	X	X	X						
	p16																X	X	X	X	X	

Hamming Code

Encode 1011

Hamming Code

Encode 1011

Write out locations:

_	_	_	_	_	_	_	_
1	2	3	4	5	6	7	8
p	p	d	p	d	d	d	p

Hamming Code: Step 1

Encode 1011

Fill data:

_	_	1	_	0	1	1	X
1	2	3	4	5	6	7	8
p	p	d	p	d	d	d	p

Hamming Code: Step 2

Encode 1011

Fill data:

```
_ _ 1 _ 0 1 1
1 2 3 4 5 6 7
p p d p d d d
bit 1 covers: 1, 3, 5, 7:
_ 1 0 1, so the _ = 0
bit 2 covers: 2, 3, 6, 7:
_ 1 1 1, so the _ = 1
```

Hamming Code: Step 3

Encode 1011

Fill data:

```
0 1 1 _ 0 1 1
1 2 3 4 5 6 7
p p d p d d d
```

bit 4 covers 4, 5, 6, 7:

_ 0 1 1, so the _ = 0

Hamming Code

Encode 1011

Fill data:

0	**1**	**1**	**0**	**0**	**1**	**1**
1	2	3	4	5	6	7
p	p	d	p	d	d	d

Done!

Fixing Errors

- Suppose there's a single error in transmission of the encoded version of our previous transmission:
  - Receiver gets 0110010
  - How do we detect errors?
    - Write out groups of parity + data (just like encoding)
    - If incorrect, mark as incorrect
    - Add up all the incorrect "parity spot-numbers" = Bit to flip to recover from error

Detecting and Fixing Errors

Let’s say we receive: 0110010

P1 (1, 3, 5, 7): 0, 1, 0, 0 -> odd

P2 (2, 3, 6, 7): 1, 1, 1, 0, -> odd

P4 (4, 5, 6, 7): 0, 0, 1, 0 -> odd

1 + 2 + 4 = 7 is position of error, so flip it!

RAID

- Redundant Arrays of (Inexpensive) Disks (RAID)
- **RAID 0**: Striping
  - Address designates which disk (i.e. 4 disk, mod 4)
  - Better bandwidth, unchanged latency, less reliable
- **RAID 1**: Mirroring/Shadowing
  - Constantly duplicate to mirror
  - Reliable, but very redundant
- **RAID 2:** Hamming ECC
  - Bit-level striping, one disk per parity group
- **RAID 3**: Parity
  - Basically RAID 0 with a parity disk (non-block level striping)
  - Need to read all disks to detect errors

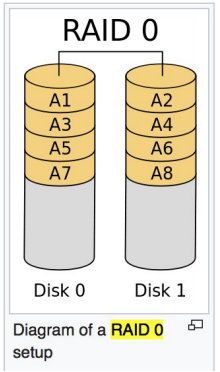


Diagram of a RAID 0 setup

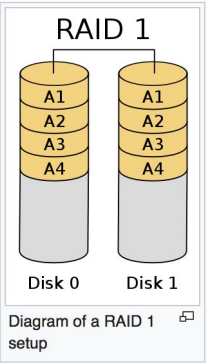
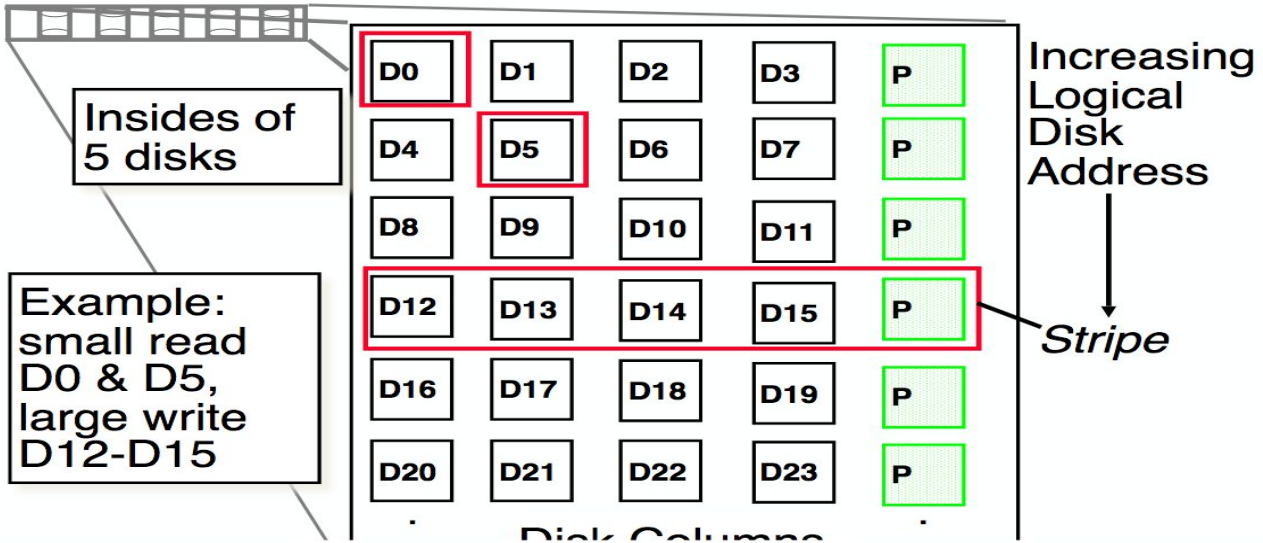


Diagram of a RAID 1 setup

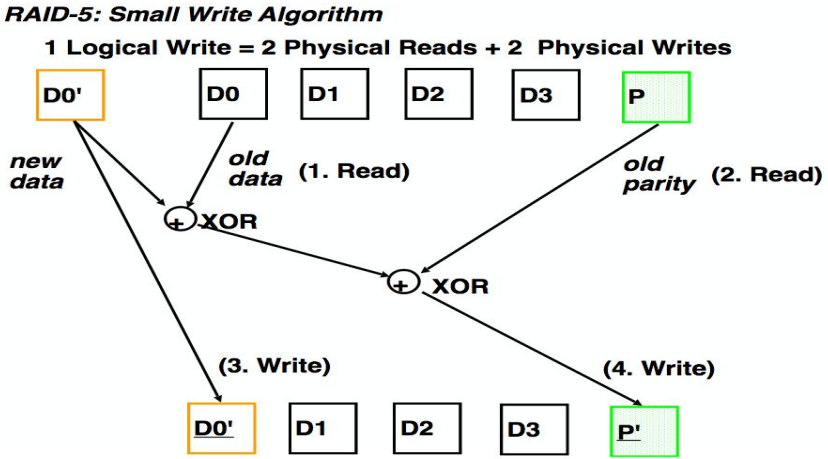
RAID

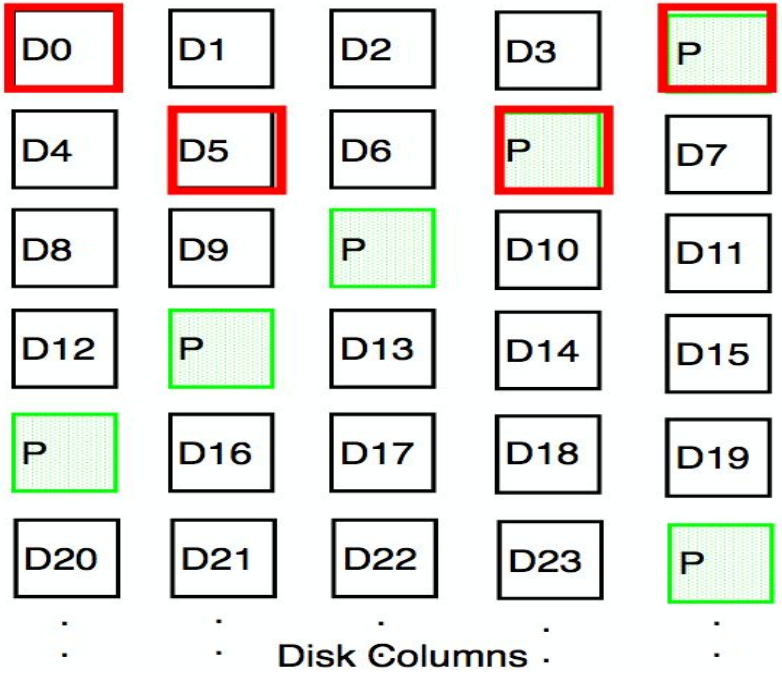
- RAID 4: Block-Based Striping with Parity



RAID

RAID 5: Block-Based Striping with Interleaved Parity





Thank you!

1. Feedback! hkn.mu/feedback
2. Email tutoring@hkn.eecs.berkeley.edu
    a. Or post on the thread on the ed
3. **Stay safe, we love you :)**
4. Good luck on your finals!! :D