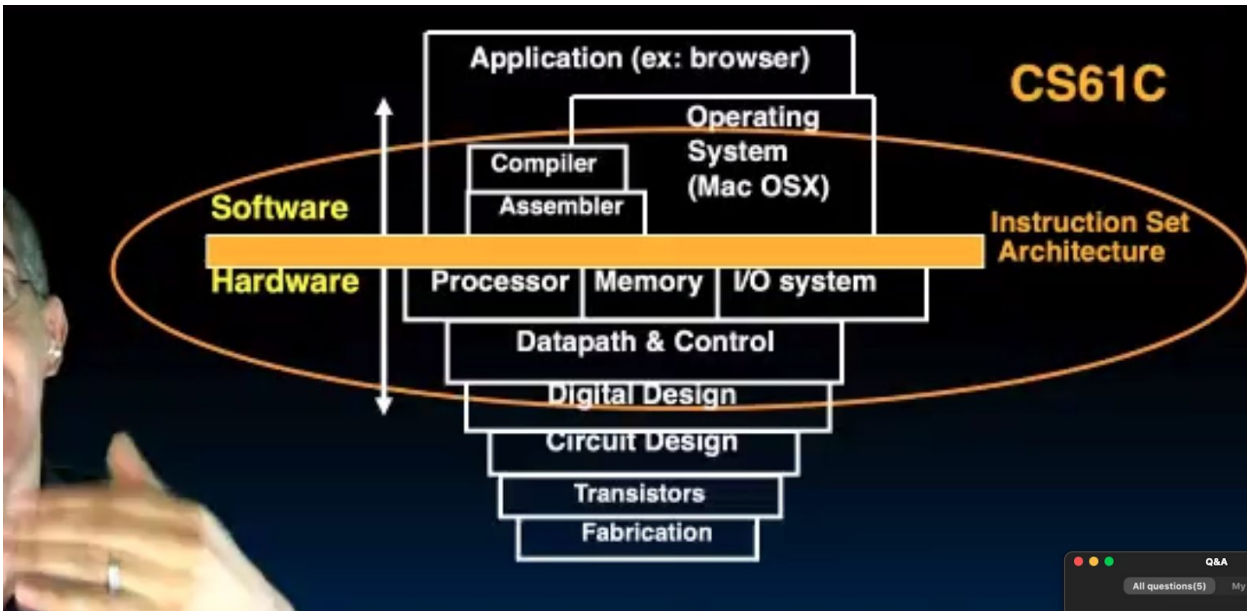
# CS 61C: Machine Structures Lecture Notes

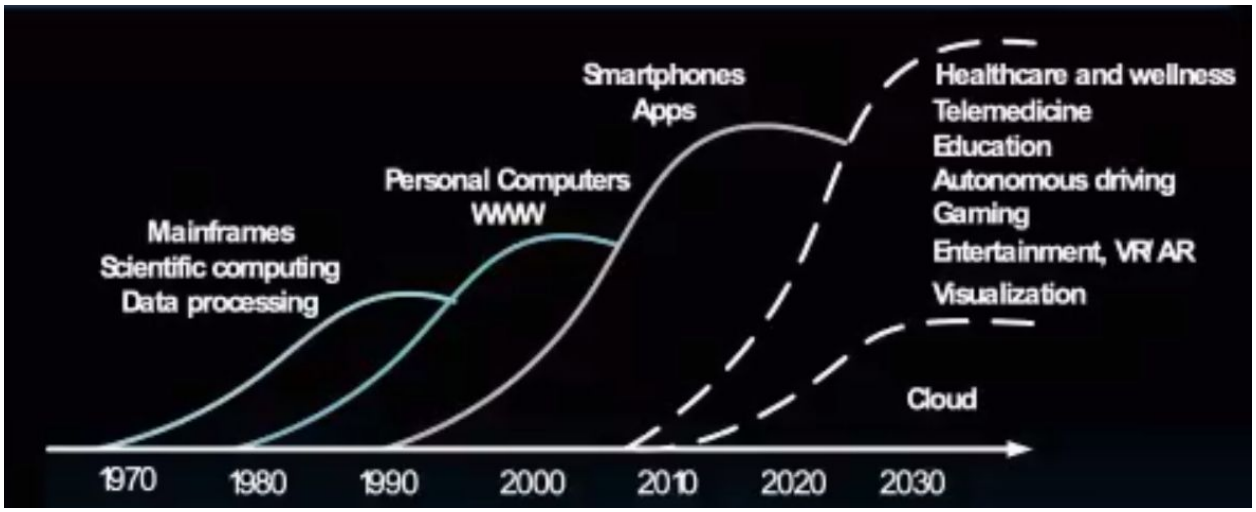
## Week 1: Lecture 1 Great Ideas of Computer Architecture (8/26)

Great Ideas of Computer Architecture

1. Abstraction
    - High Level Languages: C
    - Lower level: RISC




2. Moore's Law
3. Principle of Locality/ Memory Hierarchy
4. Parallelism
    a. Concurrent operations
5. Performance Measurement and Improvement
    a. Exploit locality parallelism
6. Dependability via Redundancy
    a. Redundancy so that a failing piece doesn't make the whole system fail
    b.




## Week 2: Lecture 2 Bits (8/28)

Analog vs Digital Data

- Need to convert between them using samples every t time and quantize the data

Big Idea: Bits can represent anything

- ASCII for all the english language
- Unicode uses 8,16,32 bits
- N bits is at most $2^n$ things

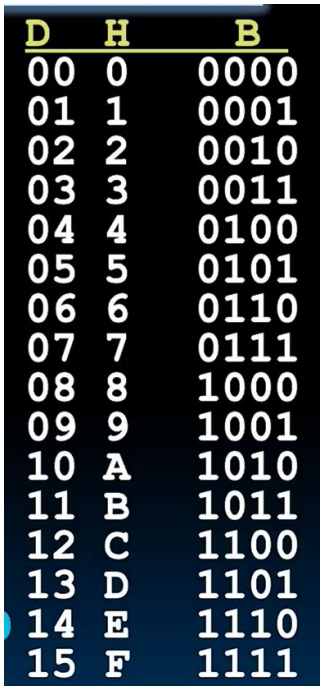
What if too big?

- Binary bit patterns are representative of numbers "numerals" is written number
- If result of add cannot be represented by these we say **overflow**

Convert from Decimal to Binary

- Left to right, is column ≤ number n
- If yes put how many can fit in n and subtract from n
- If not put 0 and keep going

Convert Binary Hexadecimal

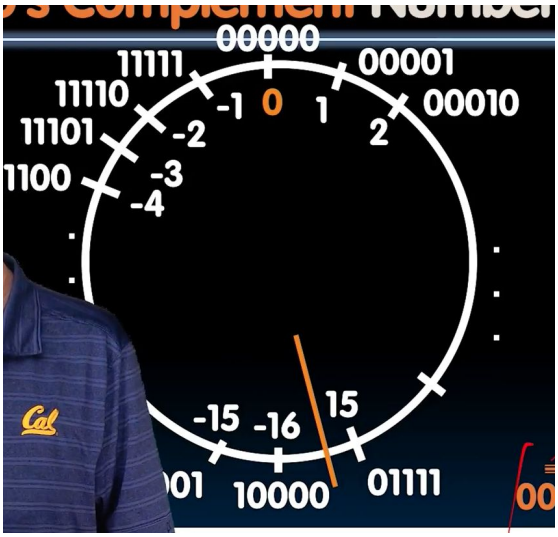
| | |
|---|---|
| - Always left pad with 0s to make full 4 bit numbers<br>- Look up<br>- Hex to binary<br>  - Just look up and drop leading 0s | -  |



-

How to Represent Negative Numbers

- Define leftmost bit to be sign
  - 0 is + , 1 is -
  - One's complement each negative bit is flipped
- Two's complement
  - Flipp the positive then add 1 to the result
    - 1101 > 0010 > 0011
  - Add the bits with a negative as the first one




  - $2^{n-1}$ non-negatives
  - $2^{n-1}$ negatives

    - One zero
- Bias Encoding
    - Shifting a unsigned bit
    - # = unsigned + bias (usually -15)
    - Can continuously count upwards from smallest number 0

## Week 2: Lecture 3 C Intro (8/31)

ENIAC (1946)

- First Electronic General-Purpose Computer
- Multiplies in 2.8ms
- Needed 2-3 days to set up new programs

EDSAC (1949)

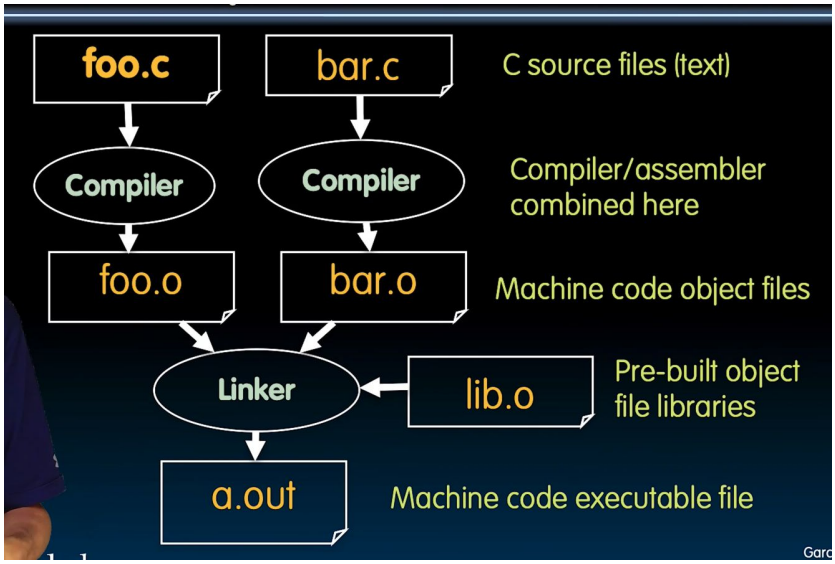
- First General Stored Program Computer
- Instructions on how to manipulate the numbers

Introduction to C (Kernighan and Ritchie)

- Enabled first operating system not written in assembly language UNIX
- C allows us to exploit underlying features of the architecture
- Popular after 40 years
- Rust: "C-but-safe"
- Go: "Concurrency"

Compilation

- Compiles directly into architecture-specific machine code (1s and 0s)
- Compile down to .o file and then link .o into executable




- If one file is changed, only recompiles that file
- Architecture-specific code
- Can compile in parallel

C Pre-Processor (CPP)

- C files pass through CPP before compiler sees code
- Ignores comments, # lines include
- Can declare macros which will replace a function
    - Could cause errors if parameter is function call

C vs. Java

| | C | Java |
|---|---|---|
| Type of Language | Function Oriented | Object Oriented |
| Programming Unit | Function | Class = Abstract Data Type |
| Compilation | `gcc hello.c` creates machine language code | `javac Hello.java` creates Java virtual machine language bytecode |
| Execution | `a.out` loads and executes program | `java Hello` interprets bytecodes |
| hello, world | `#include <stdio.h>`<br>`int main(void)`<br>`{`<br>`  printf("Hi\n");`<br>`  return 0;`<br>`}` | `public class HelloWorld {`<br>`  public static void`<br>`main(String[] args) {`<br>`  System.out.println("Hi");`<br>`} }` |
| Storage | Manual (`malloc`, `free`) | New allocates & initializes,<br>Automatic (garbage collection) frees |

Update to ANSI C is C99

- Inttypes.h
- C11 has multi threading

To get main function to accept arguments, use

- Int main (int argc, char *argv [])

<u>C Syntax</u>

True or False

- False
    - 0 (integer)
    - NULL (pointer)
    - Stdbool.h
- True
    - Everything else

Typed Variables

- Type must be declared

Int

- Prefer to use
    - intN_t and uintN_t
- Int can change depending on computer

Consts and Enums

- Declare consts when value can't change

Typed Functions in C

- Function must return or be void

Structs

- **Structs** are structured groups of variables
    - Typedef struct {
        - Int length_in_seconds;
    - }

Control Statements

- Do statement while ();

## Week 2: Lecture 4 Pointers (9/2)

Variable Declarations

- When you make a new variable without declaration, it holds garbage (contents undefined)

Undefined Behavior

- A lot of undefined behaviors in C
- "Heisenbugs" - bugs that are random/hard to reproduce
- "Bohrbugs" bugs that are repeatable

Address vs. Value

- Address refers to the memory location
- Value is stored in that location

Pointers

- A variable that contains the address of a variable

Pointer Syntax

- Int *p
    - Variable p is address of an int
- P = &y (p is "address" of y)
    - Assign address of y to p
- Z = *p
    - Tells compiler to assign value at address in p to z
    - * called the "deference operator"
- How to change a variable pointed to
    - *p = 5 changes value of x

Points and Parameter Passing

- Java and C pass parameter "by value"
- How to get a function to change a value?
- `Void addOne (int *p) {`
    - `*p = *p +1;`
- `}`
- `Int y = 3;`
- `addOne(&y)`

More Pointer Dangers

- Declaring a pointer just allocates space to hold the pointer, does not allocate something to be pointed to
- Local variables in C are not initialized

Pointers in C ... The Good, Bad and the Ugly

- Pass a pointer instead of large struct or array
- Pointers allow cleaner, more compact code

<u>Using Pointers Effectively</u>

- Can be used to point to any data type
- Can have pointers to functions
    - Int (*fn) (void *, void *)
    - (*fn)(x, y) to call function

Pointers and Structures

- Arrow notation
    - Int h = paddr->x;
        - Where pddr is a pointer
    - Aka int h = (*paddr).x;
- Point p1, p2
    - P1 = p2 will copy over values, not point to same struct

Pointing to Different Size Objects

- Modern machines are "byte-addressable"
- Type declaration tells compiler how many bytes to fetch on each access through pointer
- "Word alignment": they have to be aligned together will use up equal amount

sizeof() operator

- How many bytes in object

<u>Arrays</u>

- int ar[2]
- Allocates block of memory
- An array variables points to the first element
    - Ar[2] is same as *(ar + 2)
- Declare variable for ARRAY_SIZE rather than to declare the size of the array
- An array in C does not know its own length & bounds not checked,
    - Need to pass an array and its length
- Segmentation faults
    - Reading writing to memory without access
- Bus error
    - Wrong alignment like in Work alignment

Pointer Arithmetic

- Pointer + n: Move "that many" bytes over
- Can pass a pointer to a pointer
    - Declare as **h (handle)
    - Move where the *q is using pointers in

## Week 2: Lecture 5 Memory (9/4)

Dynamic Memory Allocation

- sizeof() gives size in bytes
- malloc() allocates memory
    - Ptr = (int *) malloc (sizeof(int))
- After Using malloc and it allocates memory, once you are done you must dynamically free it, cannot use it after free
    - free(ptr);
- Key Errors: runtime does not check)
    - free() ing same piece of memory twice
    - Calling free() on something you didn't get from malloc

Managing the Heap: realloc (p, size)

- Behaves like malloc
- After doing malloc check with NULL == ptrn
- Ip = (int *) realloc(ip, 20*sizeof(int));
- After doing realloc, check NULL

Arrays not implemented as you'd think

- An array name is not a variable
- a is the same as &a

Example Linked List

- strlen(string) + 1 used to allocate space for a string and need +1 to include null terminator (0)

Globals

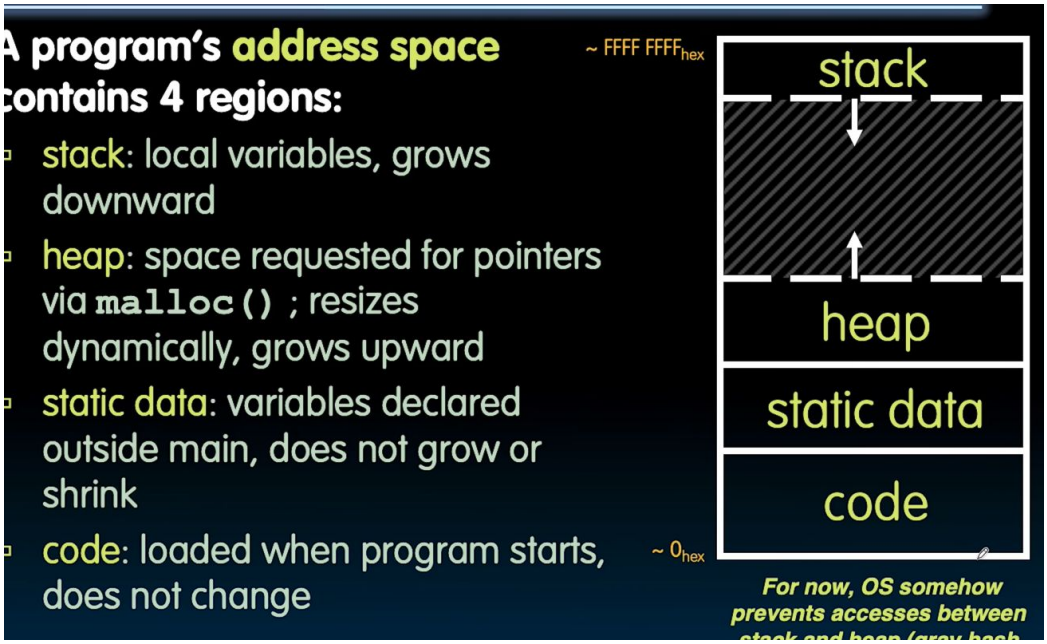
- Structure declaration *does not* allocate memory
- Variable declaration *does* allocate memory
- Global variable: data declared outside of any procedure "global scope"

C Memory Management

- 3 pools of memory
    - Static storage: global variable storage, basically permanent memory
    - Stack: local variable storage, parameters, return
    - Heap (dynamic malloc storage): data lives until deallocated by programmer
- C required knowing where objects are in memory

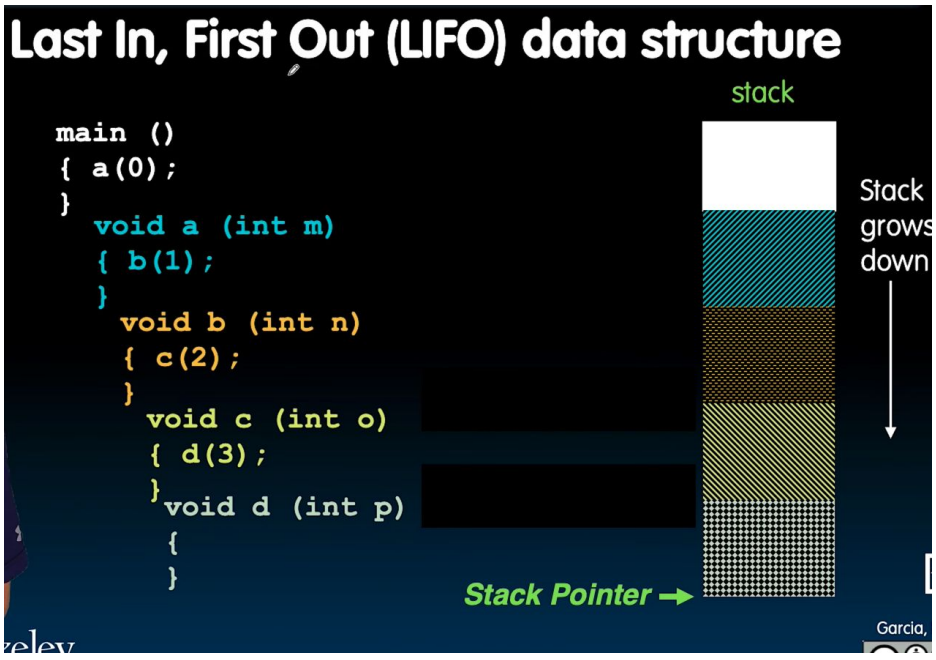
Normal C Memory Management

- Heap space: big call to malloc, photoshop pixels,
- Stack space: recursive calls, long stack,
    - If inside procedure, memory is freed when procedure returns
- Code can be resilient to memory if there isn't enough memory by using malloc will give you NULL, whereas would normally crash




Stack

- Includes
  - Return "instruction" address
  - Parameters
  - Space for other local variables
- When procedure ends, stack frame is tossed off the stack and frees memory for future stack frames
- Stack is contiguous blocks of memory, tells you where the top stack is



The Heap

- Not necessarily contiguous
- In C need to specify number of bytes of memory
- Malloc() allocates raw, uninitialized memory

Memory Management

- Code, static storage are easy, never grow or shrink
- Stack space is created and destroyed
- Managing the heap is tricky, manually allocate memory

Heap Management Requirements

- Minimal memory overhead
- Avoid fragmentation : free memory is in many small chunks

Malloc/Free Implementation

- Free blocks are kept in a circular linked list, pointer field is unused
- malloc(): searches free list for block big enough, mem is requested from OS, if can't satisfy the request it fails
- Free(): checks if the blocks adjacent to free block are also free, merges or adds to free list

Choosing block in malloc()

- Best-fit: smallest block that is big enough for request
- First-fit: choose the first block we see big enough
- Next-fit: remember where we finished searching and resume from there

Common Errors in C

Pointers in C

- Dangling reference: need to malloc before use
- Memory Leaks: free it too late

Writing off the end of the arrays

- Corrupts other parts of the program including internal C data

Returning Pointers into the Stack

- Don't return addresses of local variables in a frame
    - It may be overwritten

Don't use after free

Forgetting realloc can Move Data

- Struct foo *g = f
- If f gets realloc and data gets moved g no longer points to the correct thing (static pointer)

Freeing the Wrong Stuff

Double-Free

Losing the initial pointer

Valgrind to the rescue

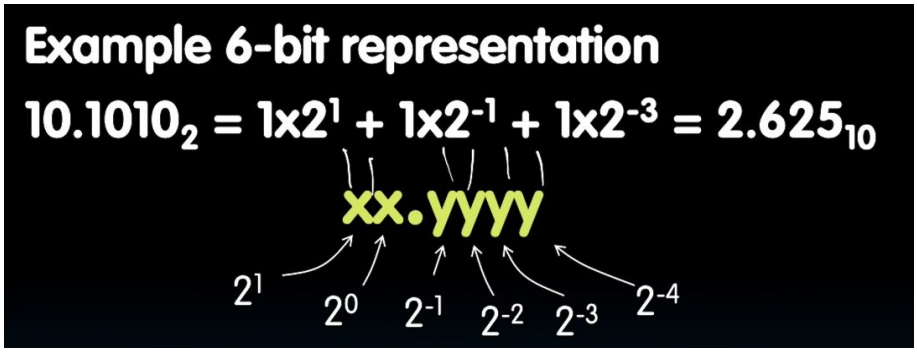
- Checks a lot of errors

## Week 3: Lecture 6 Floating Point: Basics & Fixed Point (9/9)

What can you do with N bits

- 2^N things,
- What about very large numbers or small numbers

Representation of Fractions

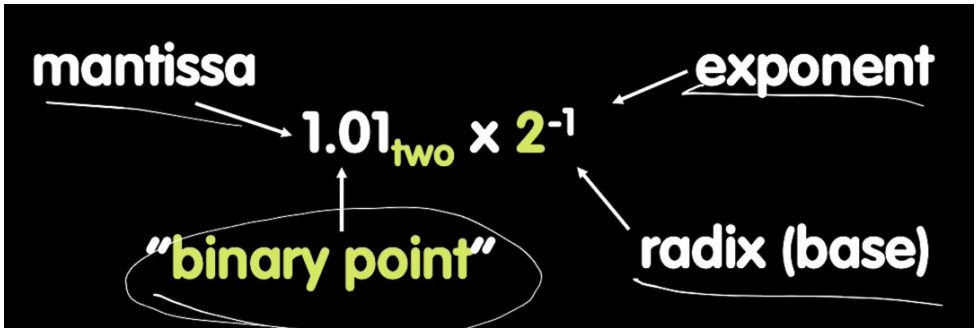
- 


- Representation of Fractions with Fixed pt.

<u>Floating point</u>

- Had fixed binary point but now we can try to "float" the binary point
- One field with the numbers and the other with the exponent

Scientific Notation

- 


- Normalized form where there is no leading 0

Floating point Representation

- Don't store the leading 1 since there is always a 1
- 1.xxxx * 2^{yyyy}
- 

- S sign bit
- Exponent represents y's
- Significand represents x's
- Overflow - exponent larger than represented in 8 bit exponent field

- Overflow: too far toward infinity or negative infinity 2*10^{38}
- Underflow: too close to 0 smaller than 2*10^{-38}

IEEE 754 Floating Point Standard

- 1 + 23 bits single, 1 + 52 bits double
- Uses Biased exponent representation
- (-1)^S * (1 + Significand) * 2^{Exponent - 127}

Special Numbers

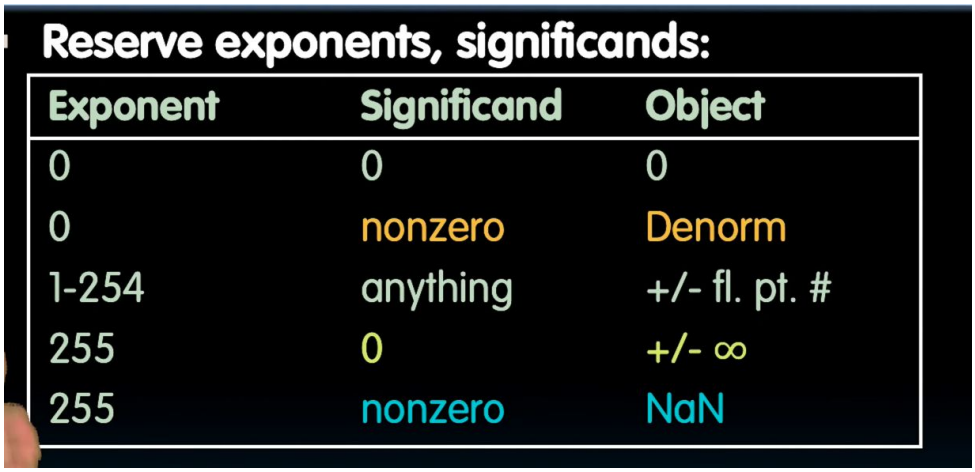
- Representation for +-infinity
- Divide by 0 should produce +- infinity
- +0: 0 00000 000000
- -0: 1 0000 000000

Representation for Not a Number

- Exponent = 255, significand nonzero is Not a Number (NaN)
- NaN helps with debugging

Denorm

- We still haven't used Exponent = 0, significand nonzero
- Smallest a = 2^{-149}
- Second Smallest is b = 2^{-148}




Examples, Discussion

(-1)^{S} + (1 + Significand) * 2^{Exponent - 127}

- We first shift decimal point then convert binary to decimal

Example: Representing ⅓

- ⅓ = ¼ + 1/16 + 1/256
- = 0.0101010101... * 2^0
- 1.0101010101 * 2^{-2}
- Exponent: -2 + 127 = 125 = 011111101

Method 2: Place Values

- Convert from scientific notation
- In binary:
    - 1.1001 = 1* 2^0 + 1*2^{-1} + 1*2^-4

Floating Point Discussion

FP add associative

- Fp add is not associative, approximates real results

Precision

- Count of the number of bits in used to represent a value

Accuracy

- Difference between the actual value of a # and its computer representation

Rounding bits

- Can always round toward + infinity, - infinity, truncate
- Unbiased (default): round like normal but on the borderline, round to nearest even number

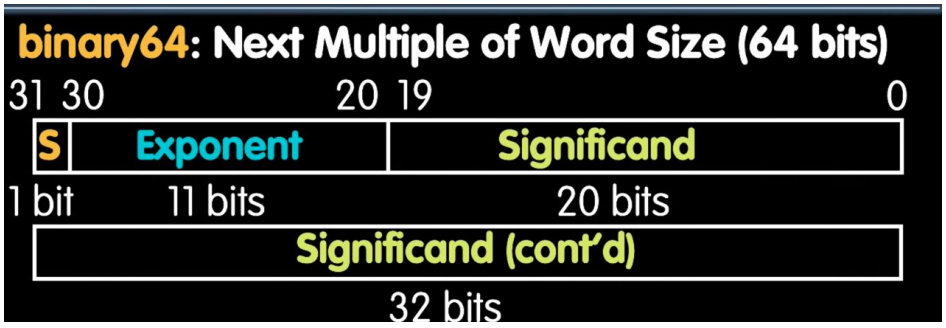
FP Addition

- Denormalize to match exponents
- Add significance
- Keep the same exponent
- Normalize

Int to float to int

- Most large values of integer don't have exact floating point representations

<u>Other Floating Point</u>

- C variable declared as double
- 

- As small as 2*10^{308}
- Quad-precision (128 bits)
- Oct-Precision (256)

Unum

- You can vary the exponent and significand

## <u>Week 3: Lecture 7 RISC-V: Introduction (9/11)</u>

Great Idea #1: Abstraction

- High Level Language Program -> (Complier) -> Assembly Language Program
- -> (Assembler) -> Machine Language Program

Assembly Language

- Job of CPU: execute lots of instructions
- Different CPUs implement different sets of instructions
    - Set of instructions is Instruction Set Architecture (ISA)
        - ARM (cell phones), Intel x86 (i9, i7)

- Understand Computers themselves at a much deeper level

Instruction Set Architectures

- Add more and more instructions to new CPUs
- RISC philosophy (Cocke IBM, Patterson, Hennessy, 1980s)
  - Reduced Instruction Set Computing
  - Keep instruction set small and simple, make it easier to build fast hardware
  - Let software do complicated operations by composing simpler ones

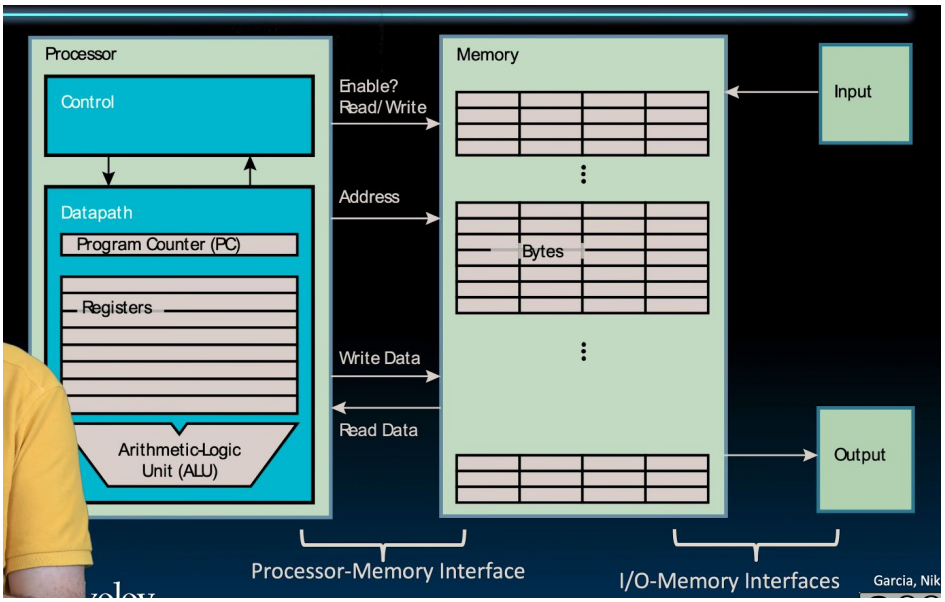
RISC-V Architecture

- Started at Berkeley and matured and more projects based on it

<u>Elements of the Architecture: Registers</u>

Instruction Set

Assembly Variables: Registers

- Assembly cannot use variables
  - Because: Keep Hardware Simply
- Assembly operands are *registers*
  - Registers are hardware objects
  - Operations are performed on data in the registers
- 


Great Idea #3: Principle of Locality / Memory Hierarchy

- CPU as fast as hardware can go
- Predetermined number of registers
- RISC-V has 32 registers

Registers

- 0x to 0x31
- 0x is always 0
- Operations determine how registers are used

<u>Assembly Instructions: RISC-V Add/Sub</u>

- Each statement executes one of a short list of simple commands

RISC-V Addition and Subtraction

- One two three, four
- One = operation by name
- Two = operand getting result (“destination” x1)
- Three = 1st operand for operation (“source1, x2)
- Four = 2nd operand for operations (“source2, x3)
- Example
    - Add x1, x2, x3
        - C: a = b + c
    - Sub x3, x4, x5
        - C: d = e - f
- Break into multiple instructions
    - Add x10, x1, x2  # a_temp = b + c
    - Add x10, x10, x3 # a_temp = a_temp + d
    - Sub x10, x10, x4 # a = a_temp - e

RISC-V Immediates

Immediates

- Immediates are numerical constants
- Special instructions for them: addi (add immediate)
- Addi x3, x4, 10
    - C: F = g + 10

Register Zero

- 0x is hardwired to value 0
- Add x3, x4, x0

## Week 4: Lecture 8 RISC-V: Decisions I (9/14)

Storing Data in Memory

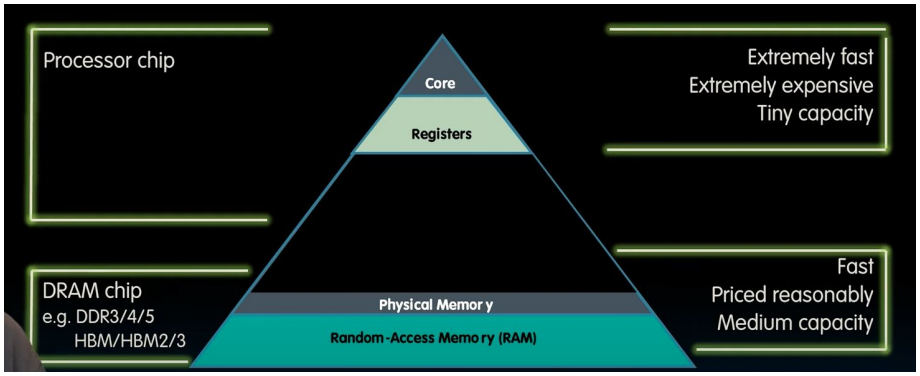
- Data Transfer Processor and memory: Load from and Store to memory

Memory Addresses are in Bytes

- Order in which *bytes* are stored
- Little-endian, least significant byte is smallest word address


- RISC-V is little endian

Data Transfer Instructions




- Great Idea #3 Memory Hierarchy
- Registers: 32 words (128 Bytes)
- Memory (DRAM): Billions of bytes (2 GB to 64 GB)
- Registers are about 50 - 500 times faster

Load From Memory to Register

- Load Word (lw) in RISC-V
  - lw x10, 12(x15) # Reg x10 gets A[3]
    - x15 base register (pointer to A[0]
    - 12 offset in bytes 3rd element * 4 bytes per word
    - Offset must be a constant known at assembly time
- Data flow from right to left

Store from register to Memory

- Store word (sw) in RISC-V
  - sw x10, 40(x15) # A[10] = h + A[3]
    - 40 bytes offset
    - 12 40 must be multiples of 4
- Data flow from left to right

Loading and Storing Bytes

- Load byte (lb) and Store Byte (sb)
  - lb x10, 3 (x11)
  - Doesn't have to be multiple of 4
  - Sign extend: smears over most significant byte to fill in

Substituting addi

- Adding immediate is much faster than loading from memory and adding
- Immediate can only but a certain length since only 32 bits

<u>Decision Making</u>

- Add, sub, addi, lw, lb, lbu, sw, sb

Computer Decision Making

- Branch if equal (beq): If statement counterpart
- Branch if not equal (bne): branch if not equal
- beq reg1, reg2, L1
  - If (value in reg1) == (value in reg2) go to L1

Types of Branches

- Branch - change of control flow
    - Conditional Branch - change control flow depending on outcome of comparison
        - Branch *if* equal (beq) or branch if not equal (bne)
        - Branch if less than (blt) and branch if greater than or equal to (bqe)
        - Unsigned versions (bltu, bgeu)
    - Unconditional Branch - always branch
        - Jump (j)
            - j label

Example if statement
RISC-V: May need to negate branch condition

| C | RISC-V |
|---|---|
| - If (i == j)<br>  - F = g + h; | - Bne x13, x14, Exit<br>- Add x10, x11, x12<br>- Exit: |
| If (i == j)<br>  F = g + h;<br>Else<br>  F = g- h; | bne x13, x14, Else<br>Add x10, x11, x12<br>j Exit<br>Else: sub x10, x11, x12<br>Exit: |

For less than or equal to, flip the operations

Loops in C/Assembly
- While, do ... while... for

| C | RISC-V |
|---|---|
| Int A[20]<br>Int sum = 0;<br>For (int i = 0; i < 20; i++)<br>  Sum += A[i]<br>  F = g + h;<br>Else<br>  F = g- h; | Add x9, x8, x0 # x9 = &A[0]<br>Add x10, x0, x0  # sum<br>Add x<br>Loop:<br>  Bge x11, x13, Done<br>  Lw x12, 0(x9) #x12<br>  Add x10, x10, x12<br>  Addi x9, x9, 4 # INCREMENT POINTER 4<br>  Addi x11, x11, 1<br>  J Loop |

## Week 4: Lecture 9 RISC-V: Decisions II (9/16)

| Logical Operations | C | RISC-V |
|---|---|---|
| Bit AND<br>Bit OR<br>Bit XOR<br>Shift left logical<br>Shift Right logical<br>Shift right arithmetic (carry sign bit) | &<br>\|<br>^<br><<<br>>> | and<br>or<br>xor<br>sll (slli)<br>srl<br>sra (srai) |

RISC-V Logical Instructions

- Register Version
- Immediate version

No NOT

- Use xor with 1111

Machine Program

Program Counter: register internal that holds byte address of next instruction

Helpful RISC-V Assembler Features

- x10 - x17 has symbolic register names for function calls
- Pseudo-instructions
- mv rd, rs = addi rd, rs, 0
- li rd, 13 = addi rd, x0, 13
- nop = addi x0, x0, 0

RISC-V Function Calls

Six Fundamental Steps in Calling a Function

1. Put arguments in a place where function can access them
2. Transfer control to function
3. Acquire storage resources needed for function
4. Perform desired task of function
5. Put return value in place where calling code can access it and restore any registers you used release local storage
6. Return control to point of origin since a function can be called from several points in a program

RISC-V Function Call Conventions

- Registers faster than memory so use them
- a0-a7 L 8 argument registers to pass parameters and two return values (a0 - a1)
- ra: one return address register to return to point of origin (x1)

| C | RISC-V |
|---|---|
| Int A[20]<br>Int sum = 0; | Mv a0, s0<br>Mv a1, s1 |

| | |
|---|---|
| Sum (a, b);<br>Int sum (int x, int y) {<br>Return x+ y | Addi ra, zero, 1016<br>J sum<br>Sum: add a0, a0, a1<br>jr ra |

Single instruction to jump and save return address: jump and link (jal)

- Make common case fast\
- 1008 jal sum # ra=1012, goto sum

Return from a function: jump register instruction (jr)

- Jr ar
- jalr

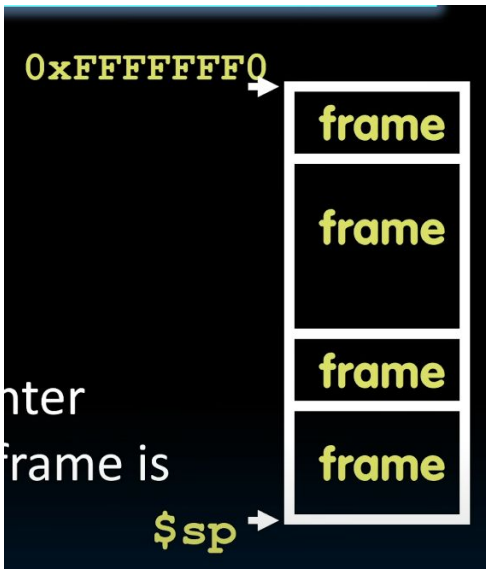
## Week 4: Lecture 10 RISC-V: Procedures (9/17)

6 Basic Steps in Calling a Function

1. Put arguments in registers where function can access them
2. Transfer control to function (jal)
3. Acquire local storage resources needed for function
4. Perform desired task of the function
5. Put return value in a place where calling code can access it release local storage
6. Return control to point of origin, since a function can be called from several points (ret)

Where are old Register values saved to restore them after function call

- Save old values before calling function restore them when return, and delete
- Use stack: LIFO queue
- Stack in memory, so need register to point to it
- sp : stack pointer in RISC-V (x2)




Stack (sp)

- Return instruction address and parameters
- Contiguous blocks of memory and stack pointer tells where bottom of stack frame is
- Stack frame is tossed off the stack after done

Leaf

- Prologue:
  - addi sp, sp, -8: Adjust stack for 2 items
  - sw s1, 4 (sp) #: Save s1 for use afterwards
  - sw s0, 0(sp) #: Save s0 for use afterwards
- F = g + h
- S1 = i + j
- Sub a0, s0, s1 # return value (g + h) - (i + j)
- Epilogue
  - lw s0, 0(sp) # restore register s0
  - Lw s1, 4(sp) # restor register s1
  - addi sp, sp, 8 # reset stack pointer

- jr ra # jump back to calling routine

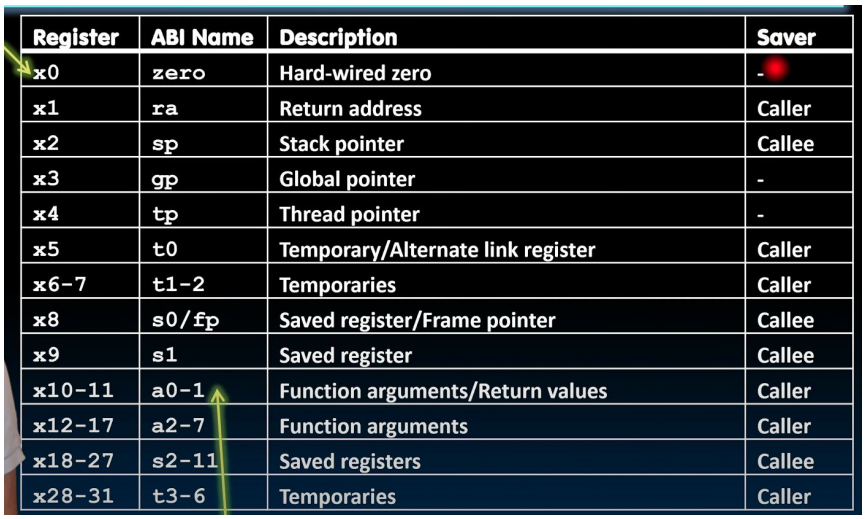
Register Conventions

Nested Procedures

- Caller: the calling function
- Callee: the function being called
- Register Conventions: A set of accepted rules to which registers will be unchanged after call (jal)

Register Conventions

- Preserved across function call
  - Caller can rely on values being unchanged
  - Sp, gp, tp; "saved registers", s0-s11
- Not preserved across function call
  - Caller cannot rely on values being unchanged
  - argumen/return registers a0-a7, ra "temporary registers" t0-t6 "volitile"



| Register | ABI Name | Description | Saver |
|---|---|---|---|
| x0 | zero | Hard-wired zero | - |
| x1 | ra | Return address | Caller |
| x2 | sp | Stack pointer | Callee |
| x3 | gp | Global pointer | - |
| x4 | tp | Thread pointer | - |
| x5 | t0 | Temporary/Alternate link register | Caller |
| x6-7 | t1-2 | Temporaries | Caller |
| x8 | s0/fp | Saved register/Frame pointer | Callee |
| x9 | s1 | Saved register | Callee |
| x10-11 | a0-1 | Function arguments/Return values | Caller |
| x12-17 | a2-7 | Function arguments | Caller |
| x18-27 | s2-11 | Saved registers | Callee |
| x28-31 | t3-6 | Temporaries | Caller |

Memory Allocation

- Automatic variables local to function and discarded
- Static variables exist across exits from and entries to procedures

Using the Stack

- "Push" addi sp, sp, -8
- Save, ret addr (sw ra, 4(sp). Y
- Need to manually push and push variables to and from stack

C: Stack in Memory

- Starts at high memory and grows down.
- Global pointer (gp) points to static
- Heap goes up
- Text starts from 1000 0000 hex text

Conclusion

## Week 5: Lecture 11 RISC-V Instruction Formats: Intro (9/21)

Machine Level Programming: 01s in program

ENIAC (upenn 1946)

- First Electronic General-Purpose Computer
- 2-3 days to set up new program with wires, multiply in 2.8 ms

Big Idea: Stored-Program Computer

- Computer can be reprogrammed in a couple seconds with code stored

EDSAC (Cambridge, 1949)

- Program held numbers in memory as code

Consequence #1: Everything has a memory address

- One register calls it Program counter (PC)

Consequence #2: Binary Compatibility

- Programs distributed in binary form
- New machines want to run old programs as well as programs compiled to new instructions
- "Backward-compatible"

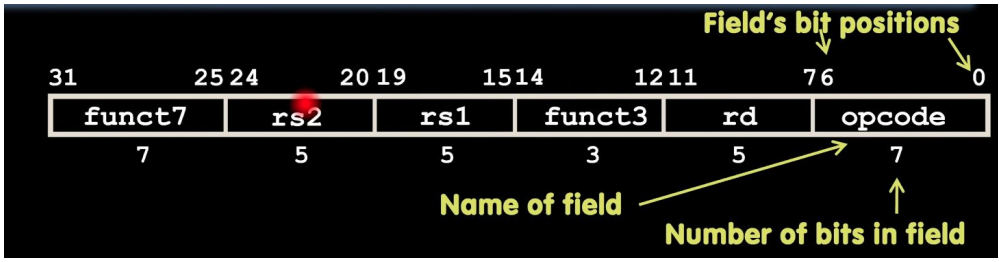
Instructions as Numbers

- Most data we work with is in words (32 bits)
- Divide instruction word into fields
- Each field tells processor 6 basic types of instruction formats
    - R - formate for register register arithmetic
    - I - register -immediate arithmetic loads
    - S - Stores
    - B - Branches
    - U - upper immediate
    - J - Jumps

R-Format Layout

R-Format Instruction Layout

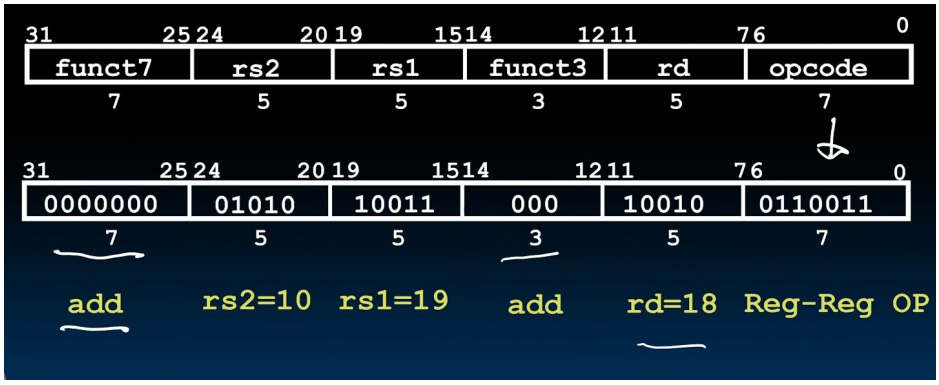
- Divided into 6 fields of varing numbers




- Opcode: specifies what instruction it is: 0110011
- Funct7 + funct3: describe what operations to perform
- Rs1, rs2 (source register)

Example

- Add x18, x18, x19



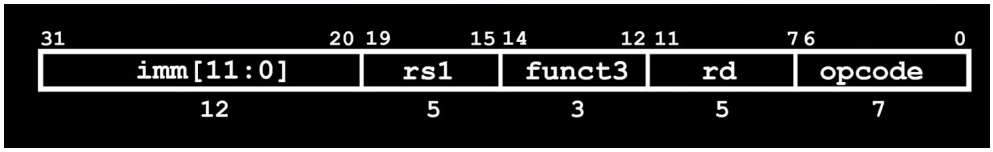


Slt- Set on less than

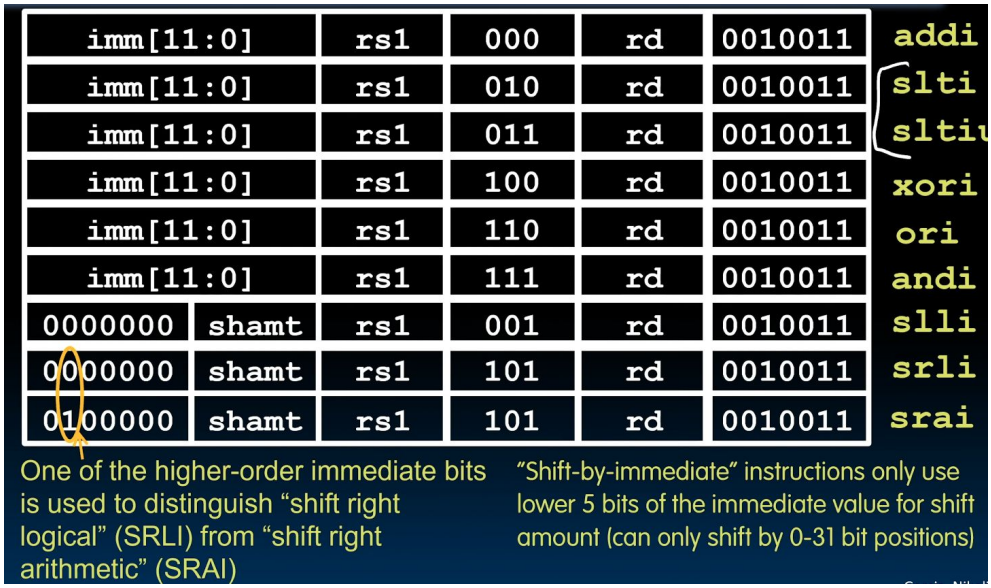
I-Format Layout

Format Instruction layout

- Need to add number larger than 31, drop funct7 field and use immediates to replace so they can have 11 fields
-

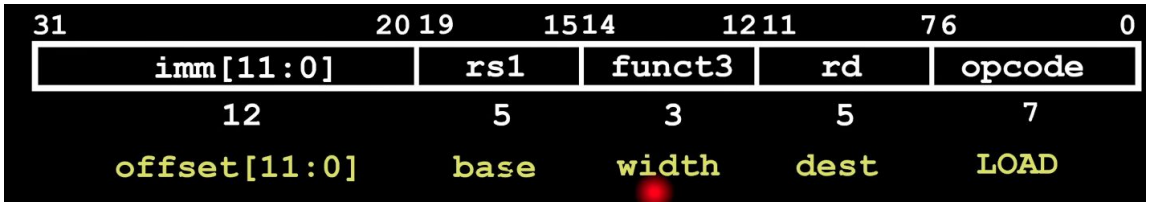



- Immediates values: 2048 to 2048

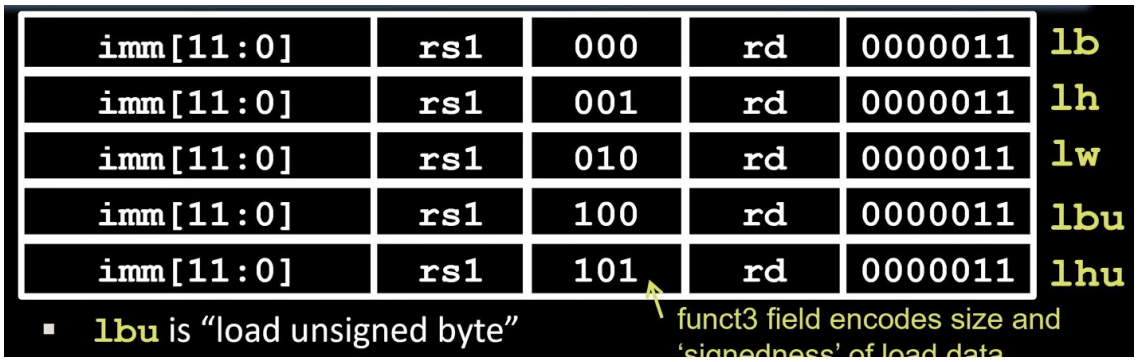



RISC-V Loads

- New opcode but same as I type




- Offset is where the immediate value is

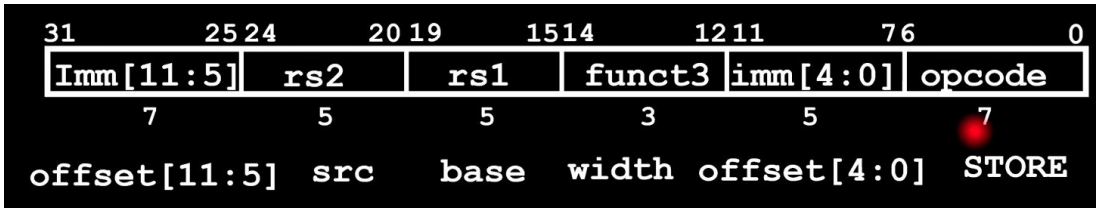



- Load halfword (lh) loads 16 bits and sign extends
- Load unsigned halfword (lhu) zero extends 16 bits to fill destination 32 bit register

S-Format Layout

Format

- Need src, base, and width
- sw src, offset(base)





## Week 5: Lecture 12 RISC-V Instruction Formats II (9/23)

B-Format

Conditional Branches

- How do we store branches, branches used for loops (if else, while, for)
- Instruction are stored in text memory

PC-Relative Addressing

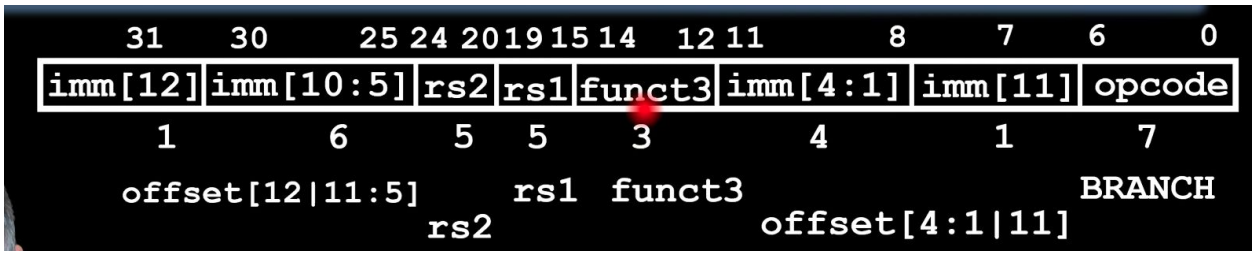
- Mediate value to point to offset from current instruction
- Instructions are always 32-bits (4 bytes) don't want to branch into middle of instruction
- Multiply offset by 4 bytes before adding to PC

Branch Calculation

- If we don't take the branch
  - PC = PC + 4 # next instruction
- If we do take the branch:
  - PC = PC + immediate*2
- Immediate is number of instructions to jump forward(+) or backward(-)

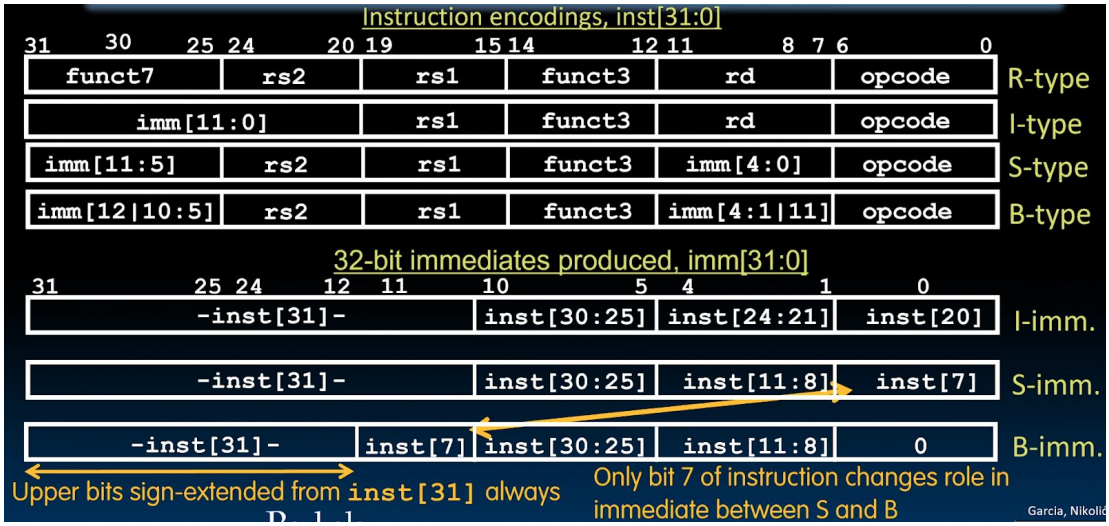
Supports n x 16-bit instructions

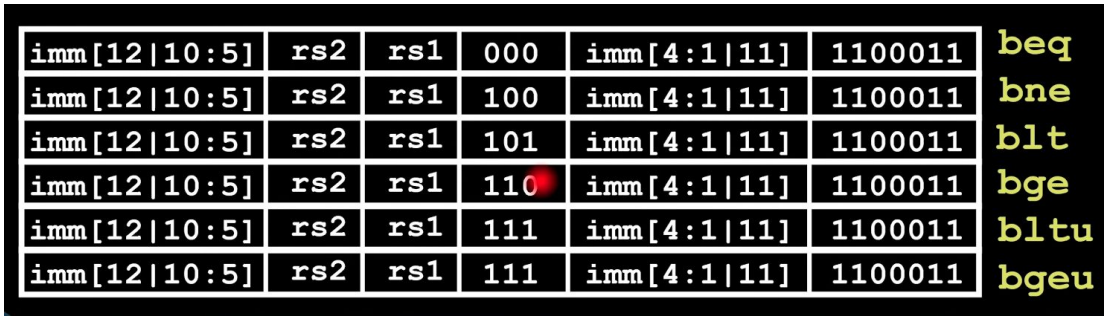
- When we want code to take up least space
- Supports by scaling the branch offset by 2 bytes even when no 16 bit instructions




Branch example

- Branch offset: 4 * 32 bits = 16 bytes



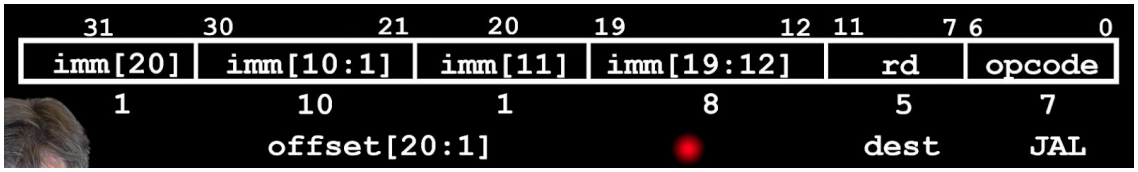

Long Immediates

What if branch is far away or code is moved

- Jump is farther than branch
- U-Format for "Upper Immediate" Instructions
-


- Lui : load upper immediate
  - Can create a full 32 bit value by combining with addi but addi sign extends so need to add extra one
  - Li will handle the case for you
- Auipc: adds upper immediate value to PC place result in destination

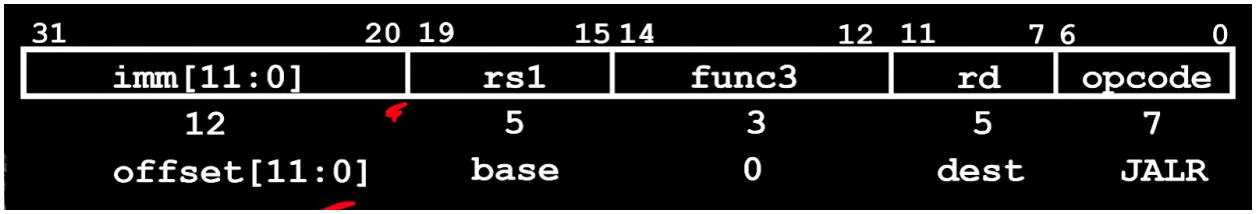
J-Format




- Saves PC+4 in rd
- Supports PC = PC + offset
- Target can be +-2^19 j does not save rd

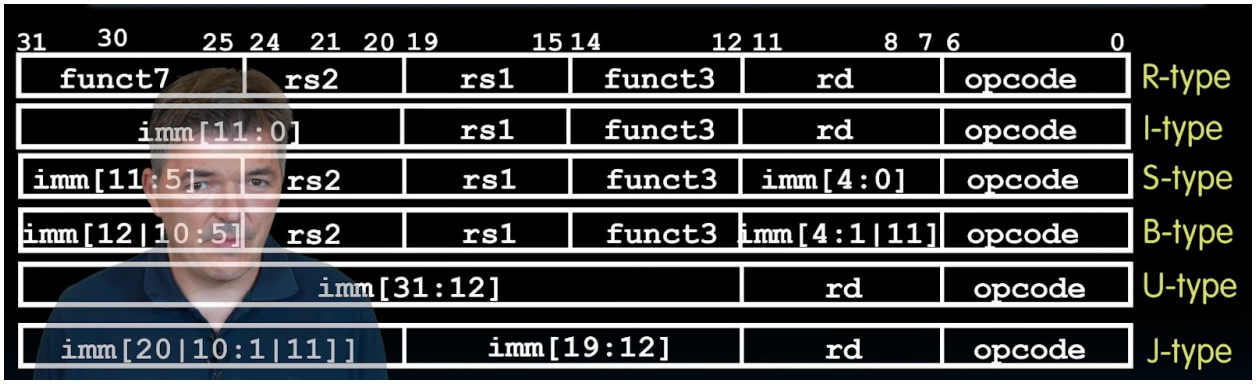
JALR Instruction (I-Format)

- jalr rd, rs, immediate
  - Writes PC + 4 to rd
  - Sets PC = rs + immediate
  - Uses same immediates as arithmetic and loads
    - No multiplication by 2 bytes




Examples

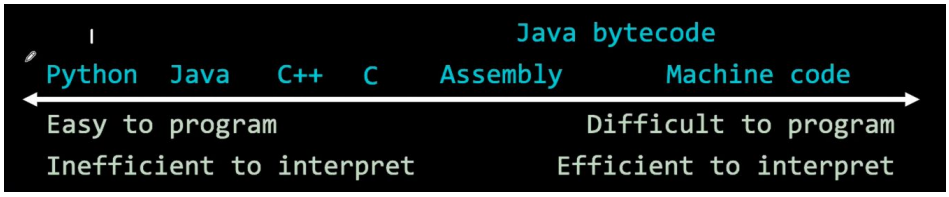
- ret = jr ra = jalr x0, ra, 0
- lui x1, <hi20bits>
  - jalr ra, x1, <lo12bits>
- auipc x1, <hi20bits>
  - jalr x0, x1, <low12bits>




## Week 5: Lecture 13 Compile, Assembly, Linking, Loading (9/25)

Language Execution Continuum

- Interpreter is program that executes other programs
-




- Language Translation
- Interpret a high level language when efficiency is not critical
- Translate to lower-level language to increase performance

Interpretation vs Translation

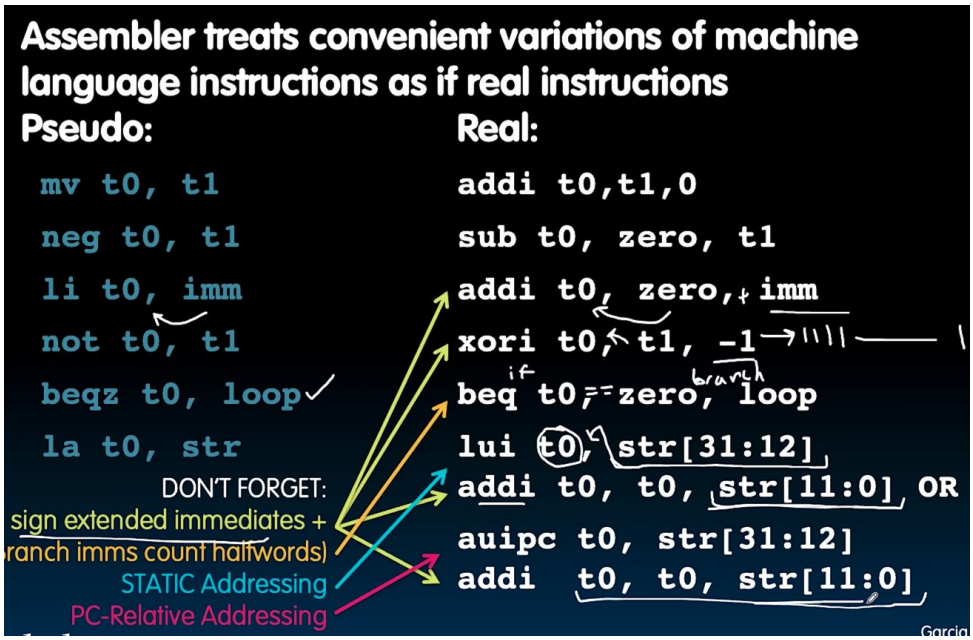
- Interpreter directly executes a language
- Translates language to equivalent program in another language
- Interpreter can be easier to learn/debug
    - Closer to high level so can give better error messages
    - Interpreters slower (10x?)
- Translation: Hides program source from the users
- Pseudo-instructions: assembler understands but not in machine, mv

Assembler

- Input: assembly language code
- Output: Object Code, information tables
- Reads and Uses Directives
- Replace Pseudo-instructions
- Produce Machine Language
- Creates Object File

Assembler Directives

- .text : put in user text segment (machine code)
- .data : put in data segment (source file data)
- .globl sym : declares sym global and can be referenced from other files
- .string str : store str in memory and null terminate
- .word w1.. wn : store in 32 bit segments




- Branching counts by half words

Producing Machine Language

- Set PC and Branching after filling in directives

Forward Reference problem

- Take two passes over the program
- Count the number of instruction half-words between target and jump to determine the offset, position-independent code (PIC)
- Lui and addi are not independent

Symbol Table

- List of items in this file that may be used by other files
- Labels: function calling
- Data: anything in .data

Relocation Table

- Need to know absolute resting place
- Things need to fill in later when you link together
    - Jal, jalr, la, lw, sw.

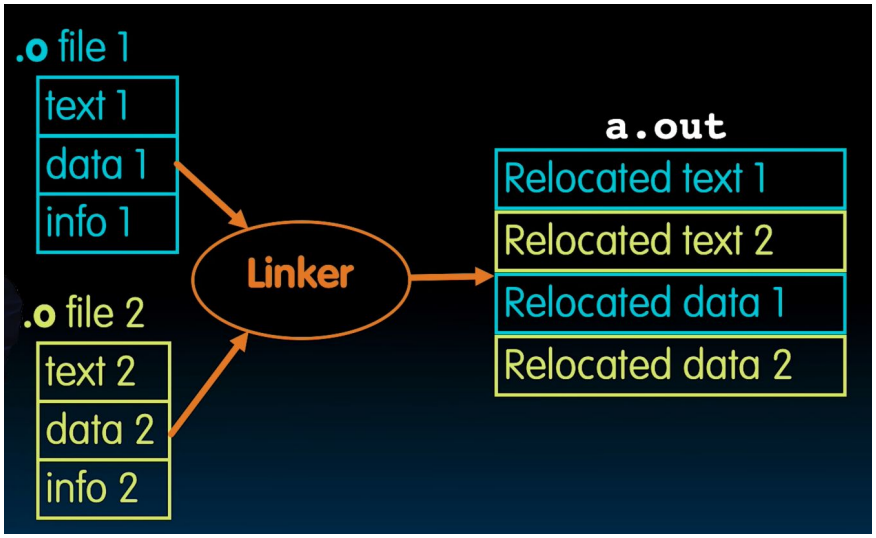
Object file format

- Object file header: size and position of the other pieces of the object file
- Text segment: machine code
- Data segment: binary representation of the static data
- Relocation information: identifies lines of code that need to be fixed up later
- Symbol table: list of labels and static to reference
- Debugging information
- Standard format is ELF

<u>Linker</u>

Linker

- Input: object code files, information tables

- Output: executable code with text and data
- Combines several object (.o) files into linking
- Enable separate compilation of files
- 


1. Take text segment from each .o file
2. Take data segment from each .o

Four types of addresses

- PC-Relative Addressing (beq, bne, jal; addi)
    - Never need to relocate
- Absolution Function Address (jalr/auipc)
    - Always relocate
- External Function Reference (jalr)
    - Always relocate
- Static Data Reference (lui/addi)
    - Always relocate

Absolute Addresses in RISC-V

- J-format : jump/jump and link
- I-, S0 format Loads and stores to variable in static area relative to global pointer

Resolving References

- Assumes first word of first text segment at address 0x10000 for RV32

Linker knows

- Length of each text and data segment
- Ordering of text and data segments

Resolving References

- To resolve references search for reference
- If not found, search library files
- If absolute address is determined, fill in machine code

Static vs Dynamically Linked Libraries

- Statically linked approach
    - Library is part of executable, baked in to the executable can't change to updates

    - Need to recompile
- Dynamically-linked libraries
    - During runtime will link to other library with one line

Dynamically-linked libraries

- Pointer to the library and so update will automatically change
- Time overhead to do link
- Link at the "lowest common denominator" machine code

Loader

Loader Basics

- Input: Executable Code (a.out)
- Output: program is run

What does it do?

- Reads header to determine size of text and data segments
- Creates new address space for program to hold text and data with stack
- Copy instructions + data into new address space
- Copy arguments passed into the program onto the stack
- Initialize machine registers
- Jump to start-up routine that copies program arguments to stack to registers & sets the PC and run, terminate when exits

Example

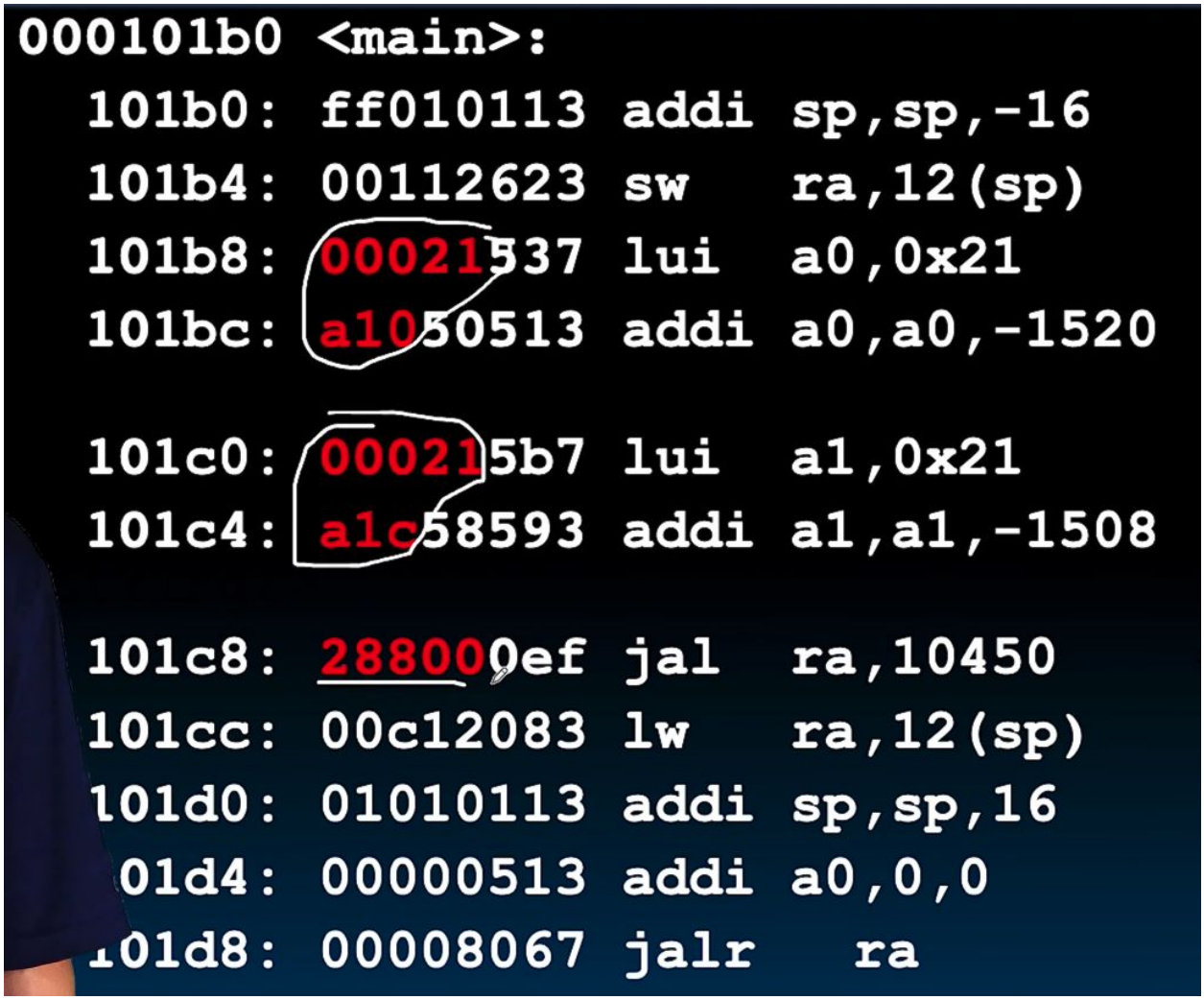
Hello World

Compiled Hello.c: Hello.s

- 
```
.text                       # Directive: enter text section
  .align 2                  # Directive: align code to 2^2 bytes
  .globl main               # Directive: declare global symbol main
main:                       # label for start of main
  addi sp,sp,-16            # allocate stack frame
  sw   ra,12(sp)            # save return address
  lui  a0,%hi(string1)      # compute address of
  addi a0,a0,%lo(string1)   #   string1
  lui  a1,%hi(string2)      # compute address of
  addi a1,a1,%lo(string2)   #   string2
  call printf               # call function printf
  lw   ra,12(sp)            # restore return address
  addi sp,sp,16             # deallocate stack frame
  li   a0,0                 # load return value 0
  ret                       # return
  .section .rodata          # Directive: enter read-only data section
  .balign 4                 # Directive: align data section to 4 bytes
string1:                    # label for first string
  .string "Hello, %s!\n"    # Directive: null-terminated string
string2:                    # label for second string
  .string "world"           # Directive: null-terminated string
```

Assembled Hello.s : linkable Hello.o

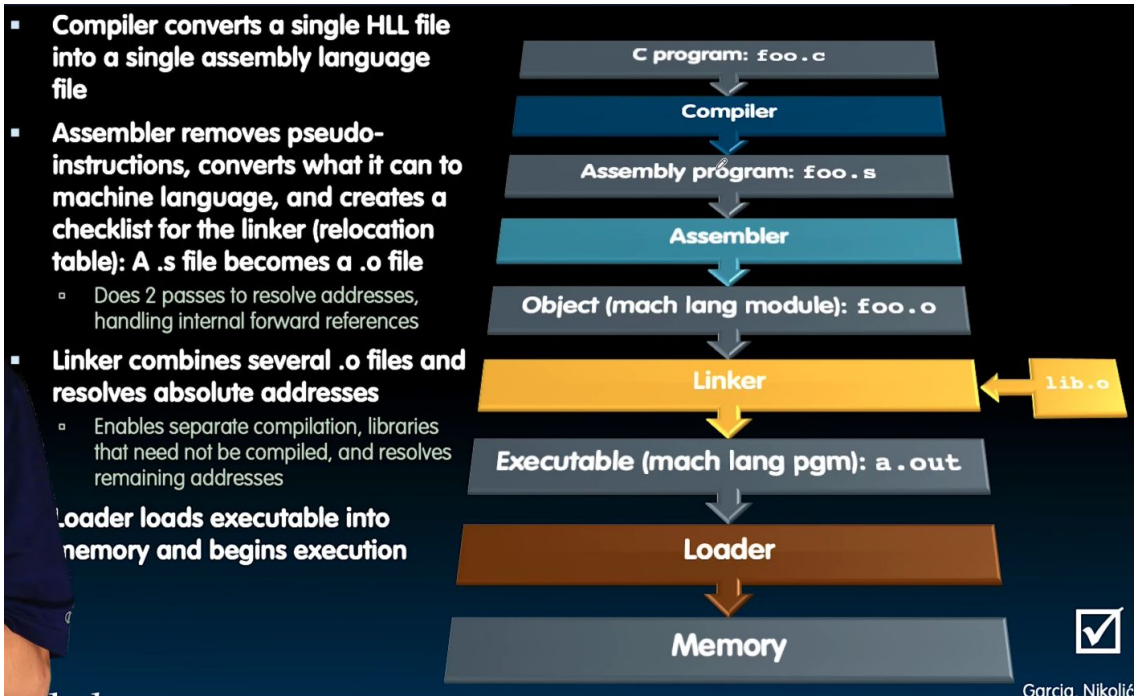
- Only the hex part is stored




- If issue with LUI/ADDI address sign extend causes it to change, we add extra 0x00001 to compensate

In Conclusion

- Converts single HLL file into single assembly language



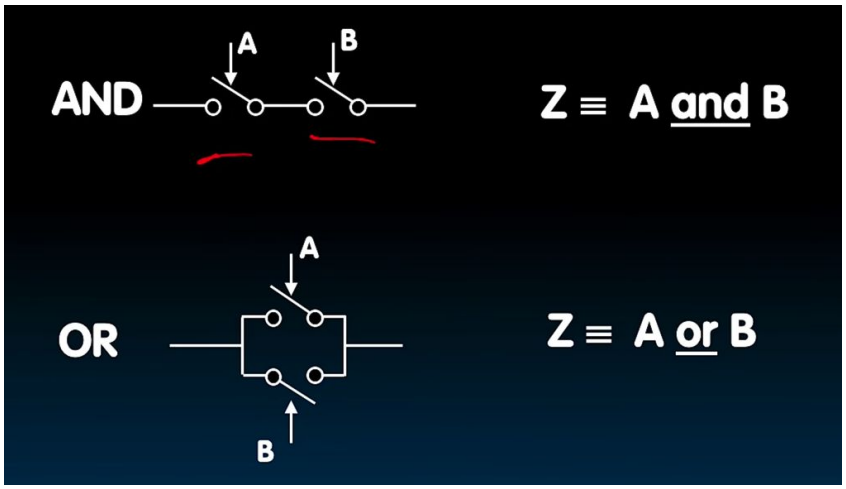

## Week 6: Lecture 14: Synchronous Digital Systems (9/29)

Synchronous Digital System

- Hardware of a processor, RISC-V
- Digital, discrete values

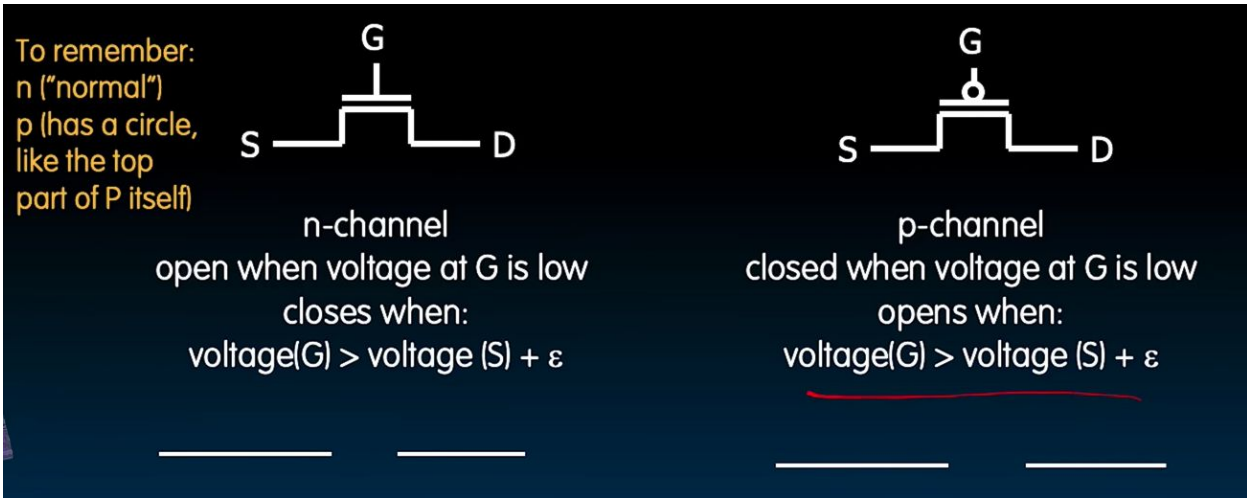
Switches

- Close switch (if A is 1 or asserted) and turn on bulb
- Open switch (if A is 0 or unasserted) and turn off light bulb (Z)
- A = Z




Transistors

- Modern digital systems designed in CMOST

- Three terminals: Drain, Gate, Source
- 


Transistor Circuit Rep vs. Block Diagram

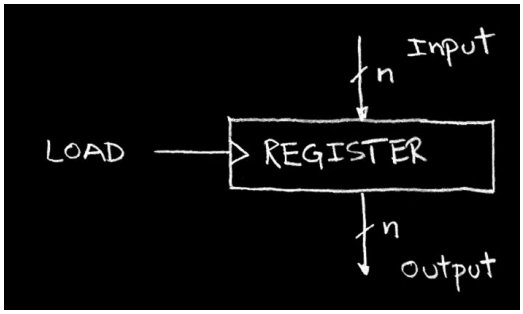
- We can use blocks to represent some form of transistors

<u>Signals and Waveforms</u>

- Digital is 1 or 0
- T is how long before wave rises
- Circuit delay: adder propagation delay
- X_0, x_1, as MSB, LSB

Types of Circuits

- Combinational logic (CL) circuits
  - Output is a function of the inputs only
- State Elements
  - Circuits that store information



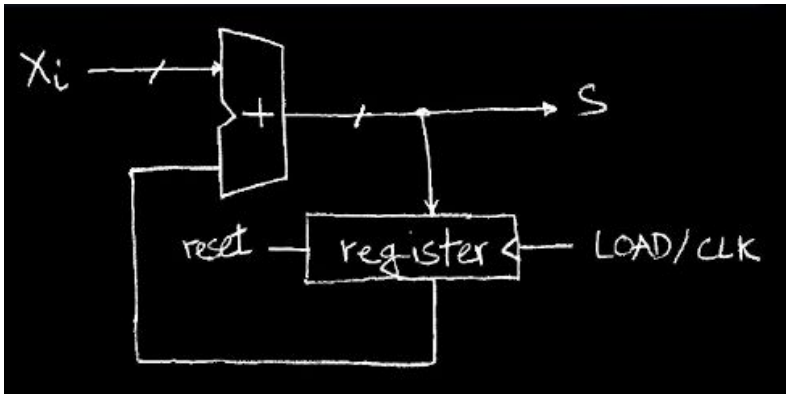

Circuits with STATE

- Register, n Input n Output and Load
- Loader holds the element for set amount of time
- Can clock the data every x time

## <u>Week 6: Lecture 15: State, State Machines (9/30)</u>

<u>Accumulator</u>

- Add up the sum of the array




<u>Register Details Flip-Flips</u>

- N instances of a "Flip-flop"
- Flip flop between 0 and 1

Timing of Flip flo

- Rising edge of clock, d is sampled and transferred to output, or else d is ignored

Samples from d, clock

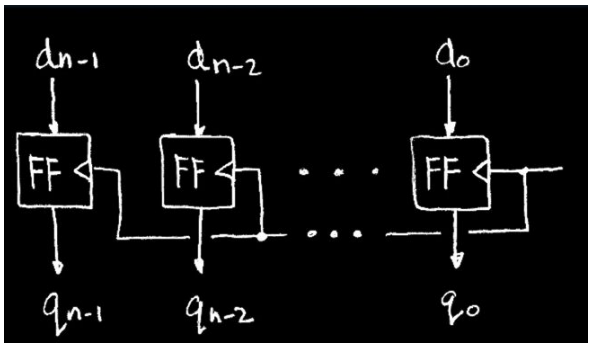
Accumulator Revisited

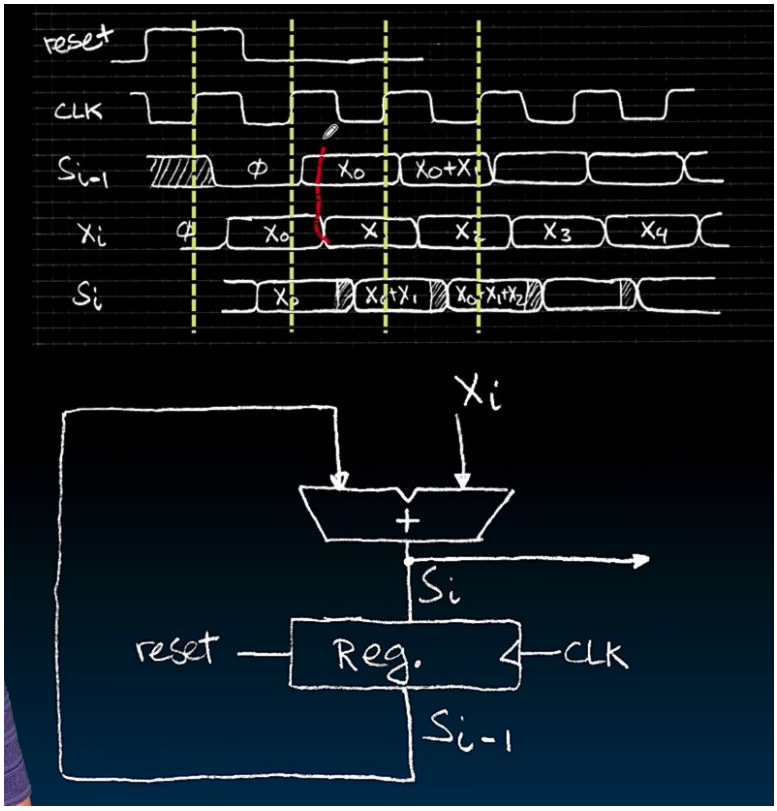
- Tclk-to-q time for clk rise to correct S value
- If you make small Tclk-to-q it's better

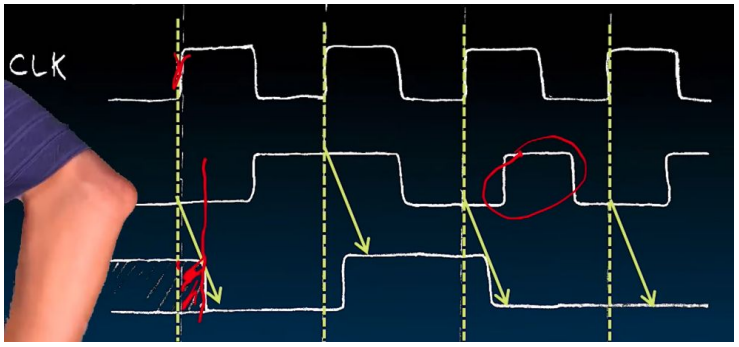
Small time were S_i is temporarily wrong

Set up time can cause the circuit to fail, need to have hold time

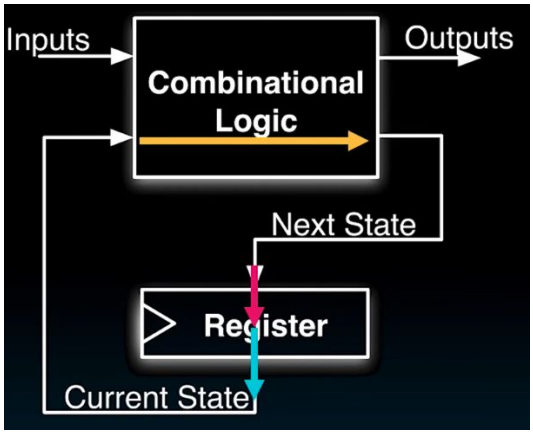
Need set up and hold to be stable




Pipelining for Efficiency



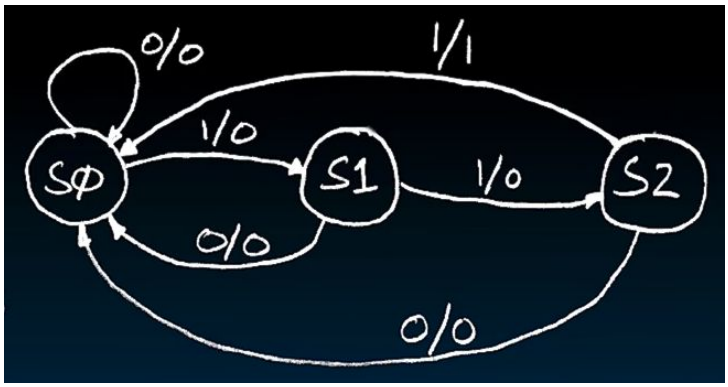
Extra Registers used to help speed up clock rate

Terms

- Clock: steady square wave that synch system
- Setup Time: When input must be stable before the rising edge of the CLK
- Hold Time: input must be stable after the rising edge of the clk
- "CLK-to-Q" Delay - how long it takes the output to change
- Flip-flop: one bit of state that samples every rising edge of the CLK
- Register - several bits of state that samples on rising edge of CLK on LOAD
- Max delay = setup Time + clk to Q Delay + CL Delay

Finite State Machine

- State transition diagram




Finite State Machine Example: 3 ones

- FSM to find 3 consecutive 1's
- State holding how many ones seen and what output

Hardware for FSM

- Combinational logic used to implement
- Register is needed to hold the representation of which state the machine is in

Conclusion

- State elements used to
  - Build memories
  - Control the flow of information between other state elements and combinational logic
  - D-flip-flops used to build registers
  - Clocks tell us when D-flip-flops change
    - Setup and hold times are important
  - We pipeline long-delay CL for faster clock
  - Finite state machines extremely useful

## Week 6: Lecture 16: Combinational Logic (10/2)

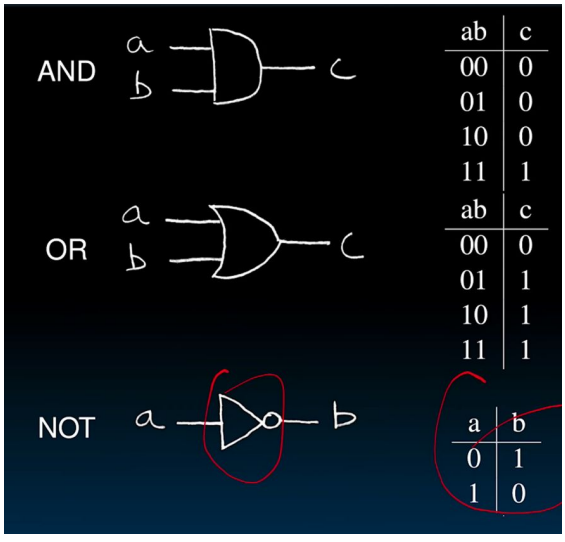
Truth Table

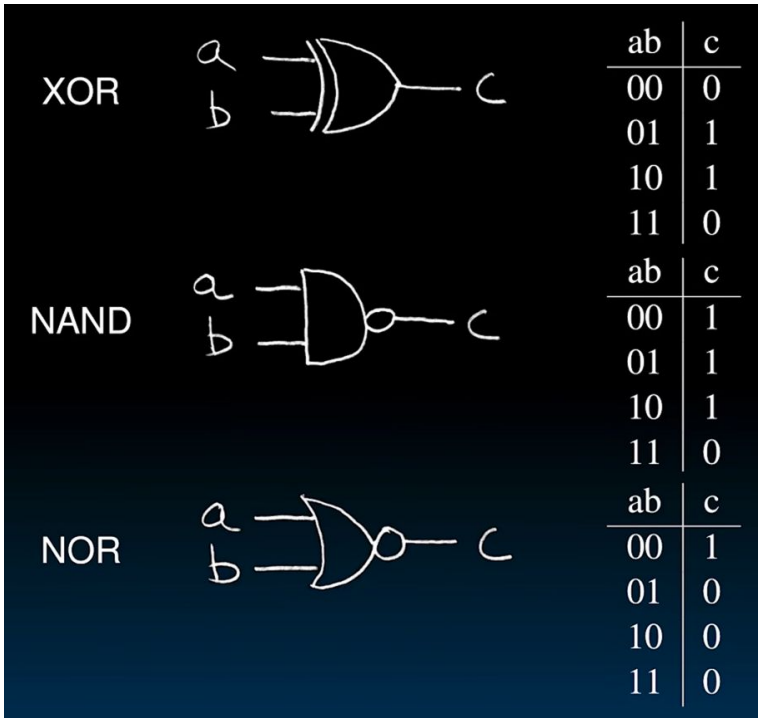
- Start with 0s and end with 1s
- Goes into F function

Examples

- Adder, Majority Circuit,

Logic Gates



- 


| ab | c |
|---|---|
| 00 | 0 |
| 01 | 1 |
| 10 | 1 |
| 11 | 0 |

| ab | c |
|---|---|
| 00 | 1 |
| 01 | 1 |
| 10 | 1 |
| 11 | 0 |

| ab | c |
|---|---|
| 00 | 1 |
| 01 | 0 |
| 10 | 0 |
| 11 | 0 |

  - Xor: OR but 11 is not true
  - NAND: Flip AND
  - NOR: Flip OR

N-inputs

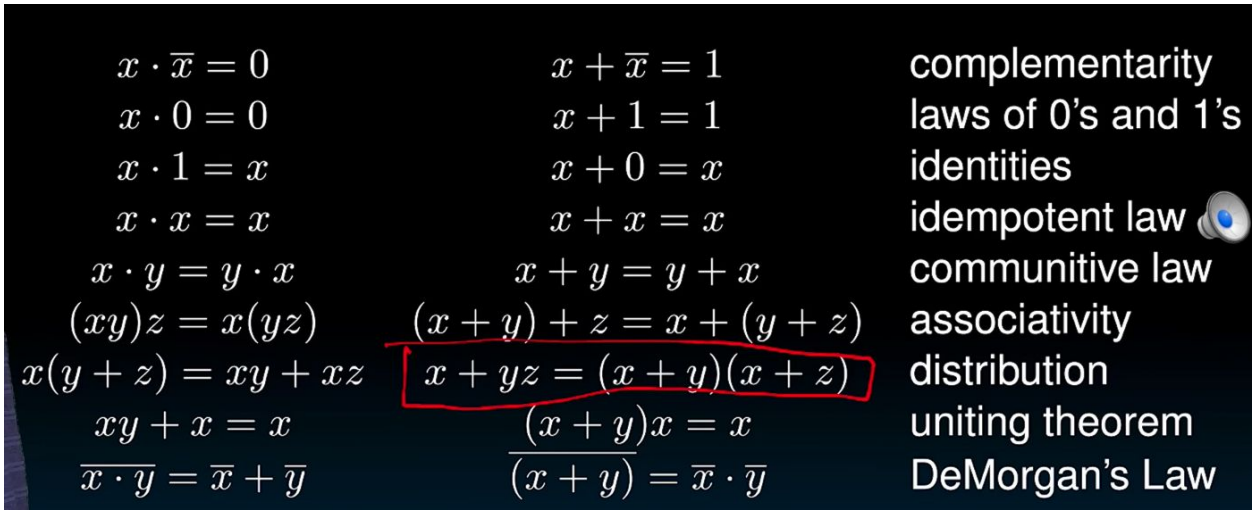
  - For XOR : 1 iff #1 is odd

Boolean Algebra

Boolean Algebra

  - +: OR
  - *: AND
  - Overline: NOT

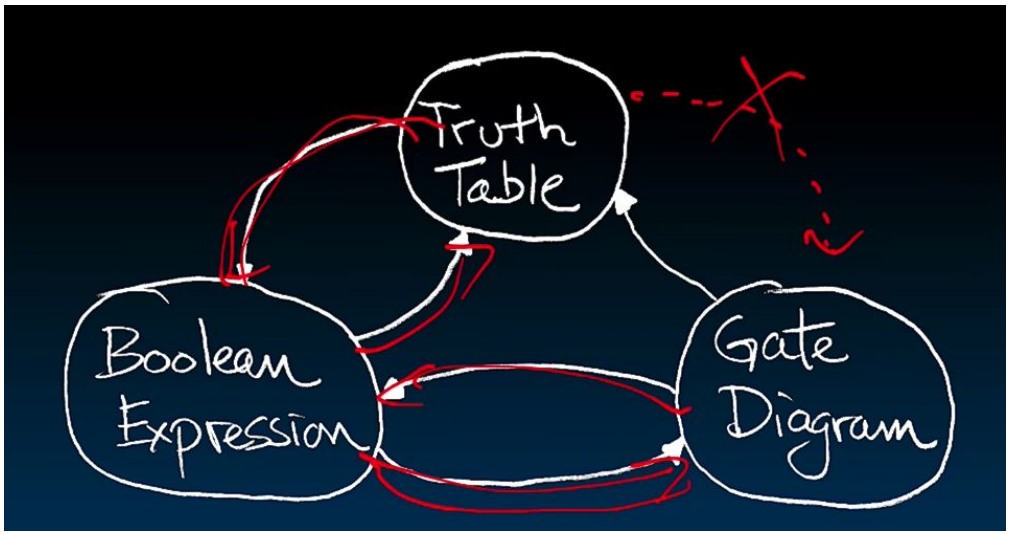
Laws of Boolean Algebra

- 

Canonical Forms

- Sum of products

$$y = \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + a\bar{b}\bar{c} + ab\bar{c}$$




## Week 7: Lecture 17: Combinational Logic Blocks (10/5)

Data Multiplexer

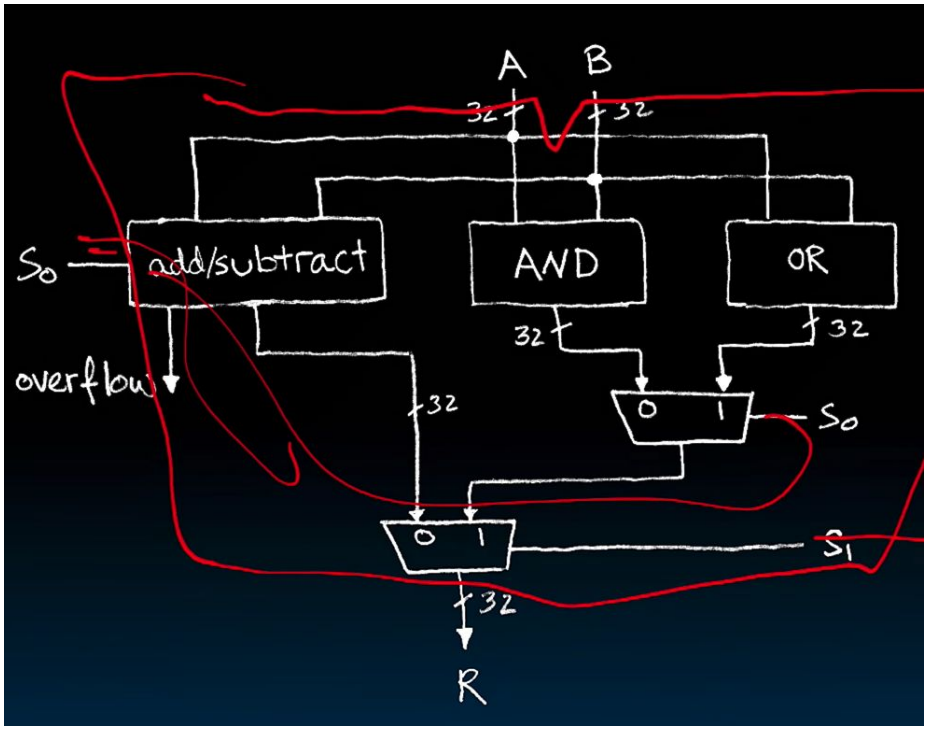
Mux

- Given A and B S decides whether A or B passes to C

4 to 1 mux

- Write what value of index of each is
- made up of 3 2-1 mux build up heirarchy

Arithmetic and Logic Unit




Adder/Subtractor

Truth-table, then determine canonical form

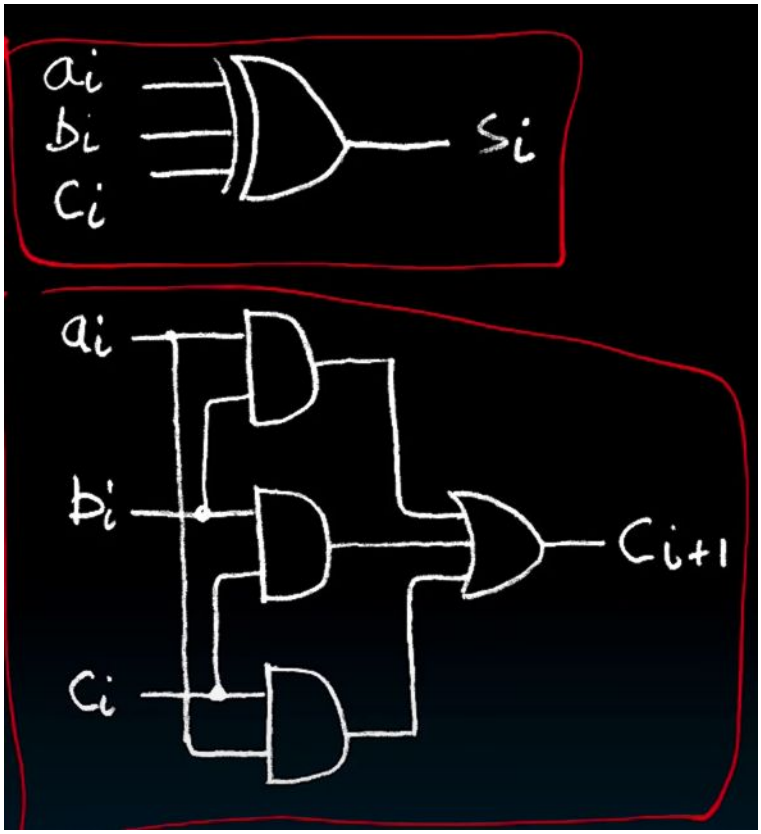
Add up for each bit

$$\begin{array}{cccc|c} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline & s_3 & s_2 & s_1 & s_0 \end{array}$$

| $a_i$ | $b_i$ | $c_i$ | $s_i$ | $c_{i+1}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

$$s_i = \mathrm{XOR}(a_i, b_i, c_i)$$

$$c_{i+1} = \mathrm{MAJ}(a_i, b_i, c_i) = a_i b_i + a_i c_i + b_i c_i$$




Sum of two 2 bit numbers

- No c out or cin > No overflow
- No cin and cout > No overflow
- Cin but not cout > ab both > 0 overflow
- Cout but no cin > a or B are -2 overflow

Subtractor Design

Clever subtractor

- XOR serves as a conditional inverter
- Use unused c has the subtractor, it will add a 1 and connect it to b to flip all the bits of b

Conclusion

- Use muxes to select among input
  - S input bits selects 2^S inputs
  - Each input n bits wide, indep of S
- Can implement muxes hierarchically
- ALU implemented by muxes
- N bit adder subtractor done by using N1 bit adders with XOR gates on input

## Week 7: Lecture 18: Single-Cycle CPU Datapath I (10/7)

RISC-V Processor Design

The CPU

- Processor (CPU) : active part of computer that does all the work (data manipulation and decision making)
- Datapath: portion that has hardware necessary to perform operations required by the processors
    - Want to build one datapath to execute every single instruction
- Control: portion of the processor that tells the datapath which instruction to execute

<u>Building a RISC-V Processor</u>

One-Instruction-Per-Cycle RISC-V Machine

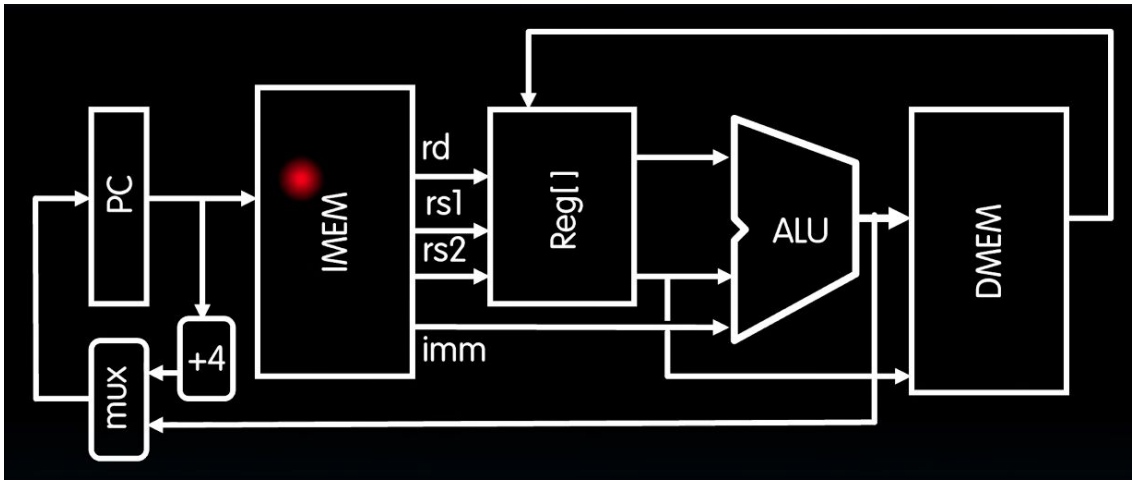
- Pc, imem, reg[], dmem
- State elements are updated with the combinational logic outputs, execution moves to the next clock cycle

Stages of the Datapath: Overview

- Problem: a single, "monolithic" block that executes an instruction too bulky
- Solution: break up process into stages then connect stages to create the whole design

Five stages of the datapath

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute - ALU (EX)
4. Memory Access (MEM)
5. Write Back to Register (WB)




Datapath Elements: State and Sequencing

- Register
- Write Enable
    - Low (or deasserted) (0):
    - Data Out will not change
    - Asserted (1): Data Out will become Data In on positive edge of clock
- Register File (regfile, RF) consists of 32 registers:
    - Two 32 bit output busses: busA and busB
    - Clk (1bit) port W (32 bits) Write enable (1 bit) RW, RA, RB, ( 5 bits)

"Magic Memory"

- One input bus: data in
- One output bus: Data out
- For read: Address selects the word to put on Data Out
- For write set write enable = 1 selects memory work to write

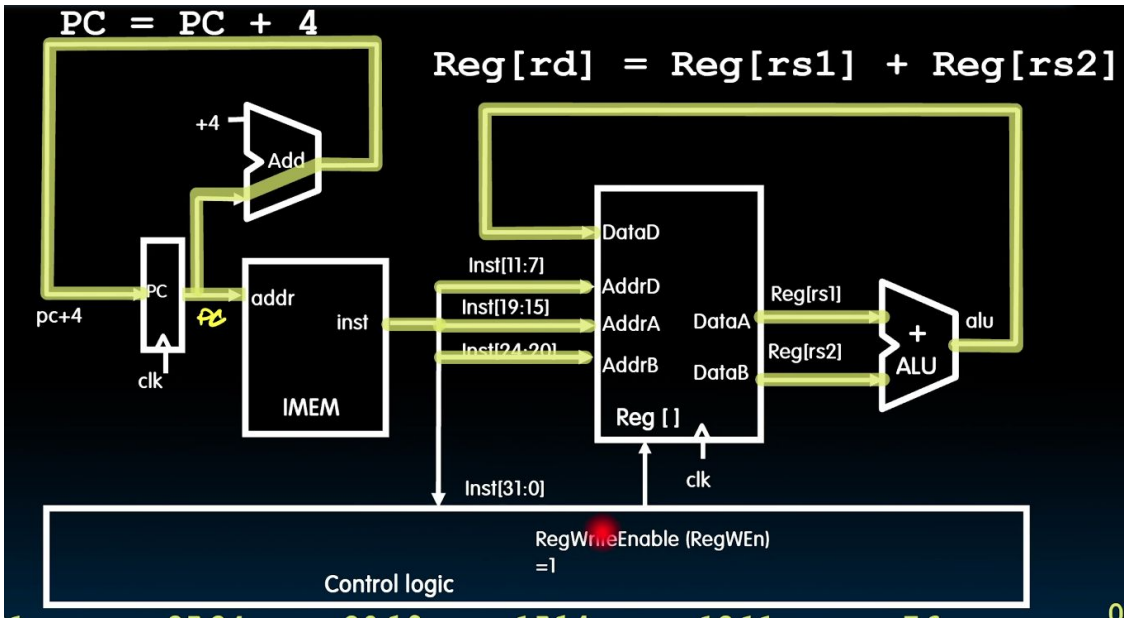
State required by RV321 ISA

- Each instruction during execution reads and updates the state of (1) Registers, (2) Program counter, (3) Memory
- Registers (x0...x31)
    - Register file holds 32 registers first register read specified by rs1 field in instruction
    - Second specified by rs2 field in instruction
- Program Counter (PC)
- Memory (MEM)
    - Holds both instructions & data, in one 32 bit byte addressed memory space

<u>R-Type Add Datapath</u>

Datapath for add

- Updates reg[rd] = sum
- PC = PC + 4

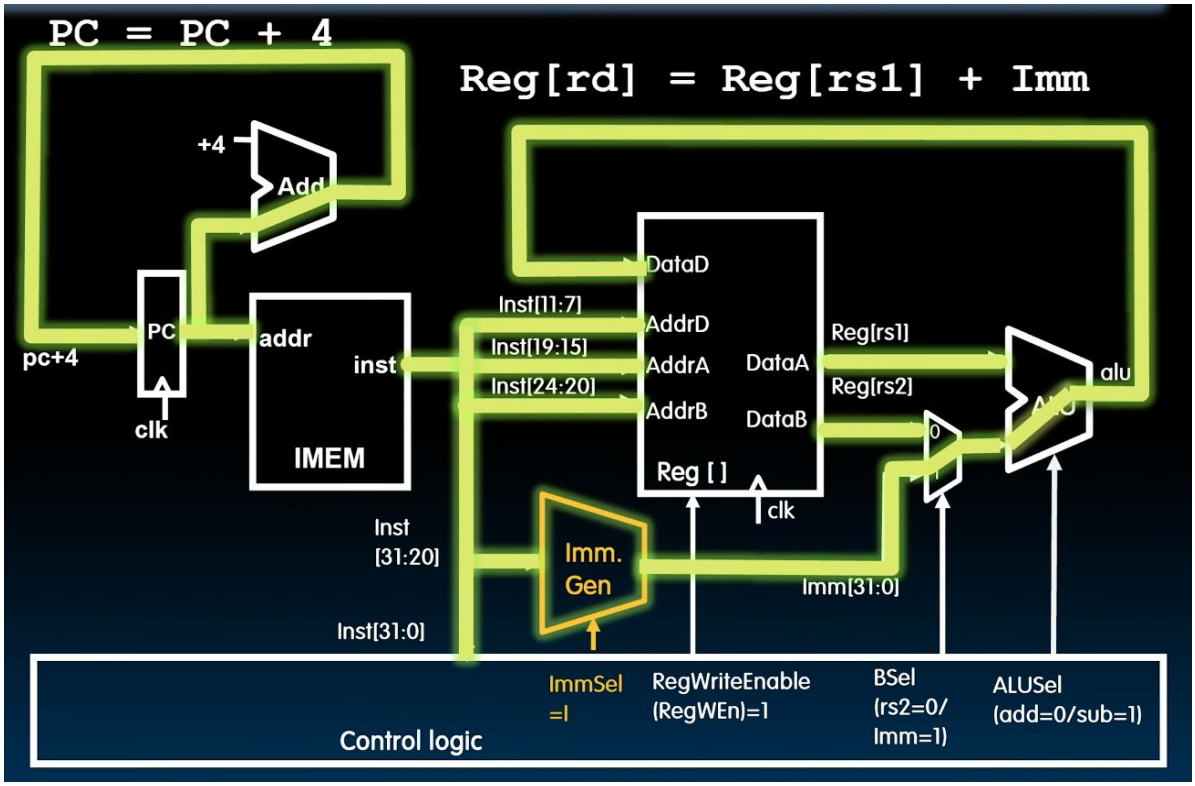



<u>Sub DataPath</u>

Modify add to get sub

- Change ALU first bit to 1 to sub

<u>Datapath with Immediates (I - Format)</u>

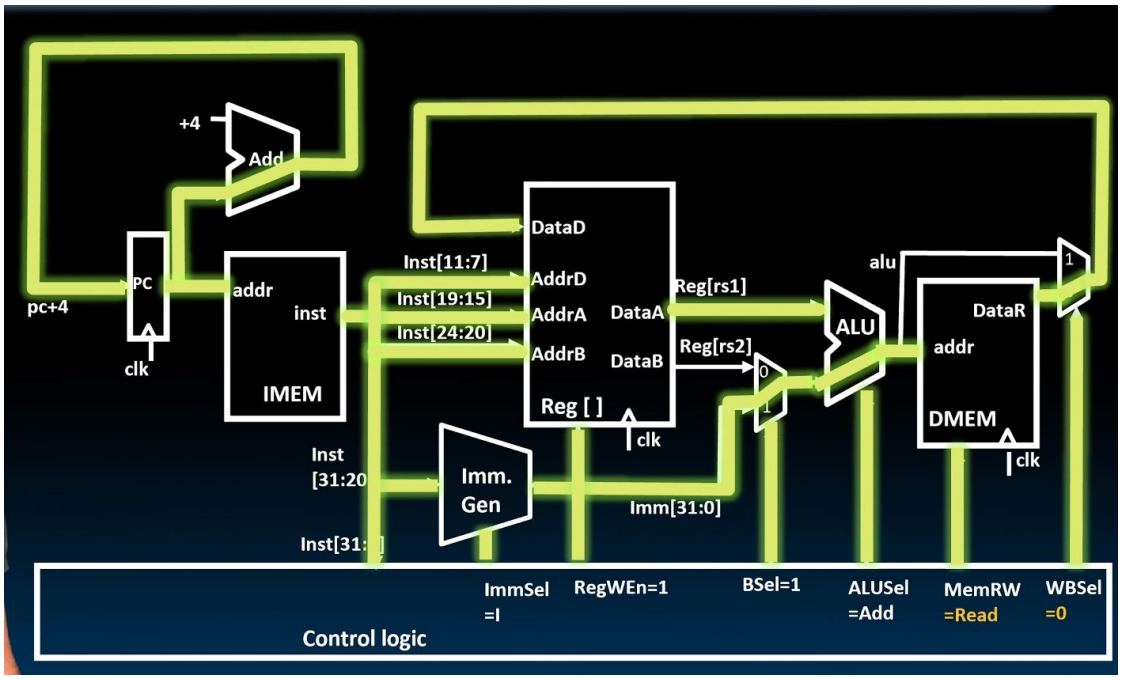



Add multiplexer and have the immediate

## Week 7: Lecture 19: Single-Cycle CPU Datapath II (10/9)

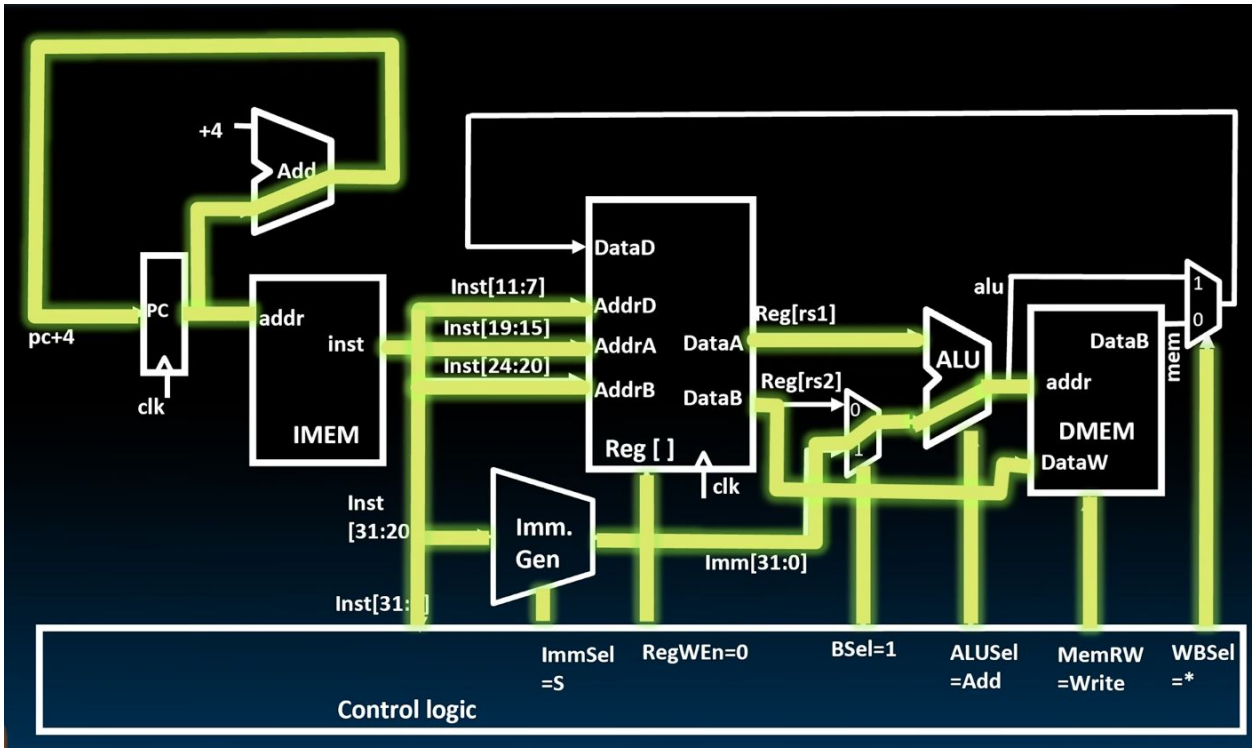
Supporting Loads

- Use multiplexer to select from alu or the memory



Datapath for Stores

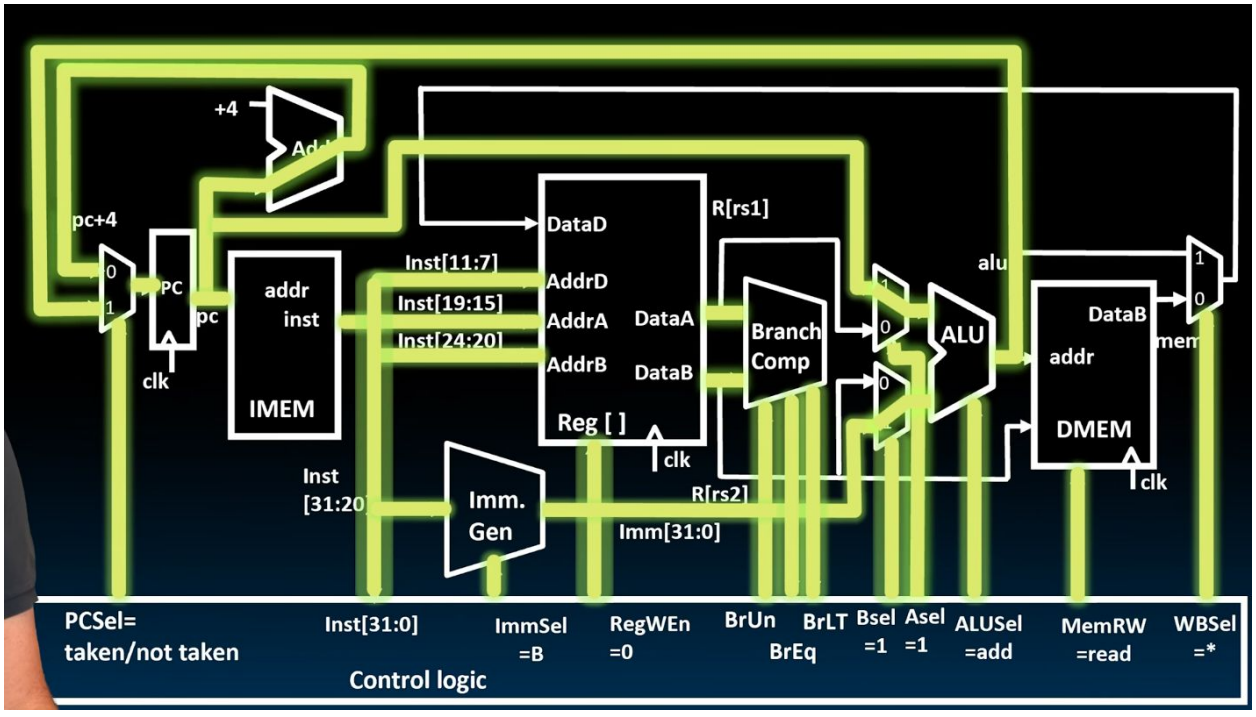
Add from dataW to reg[rs2]




Implementing Branches

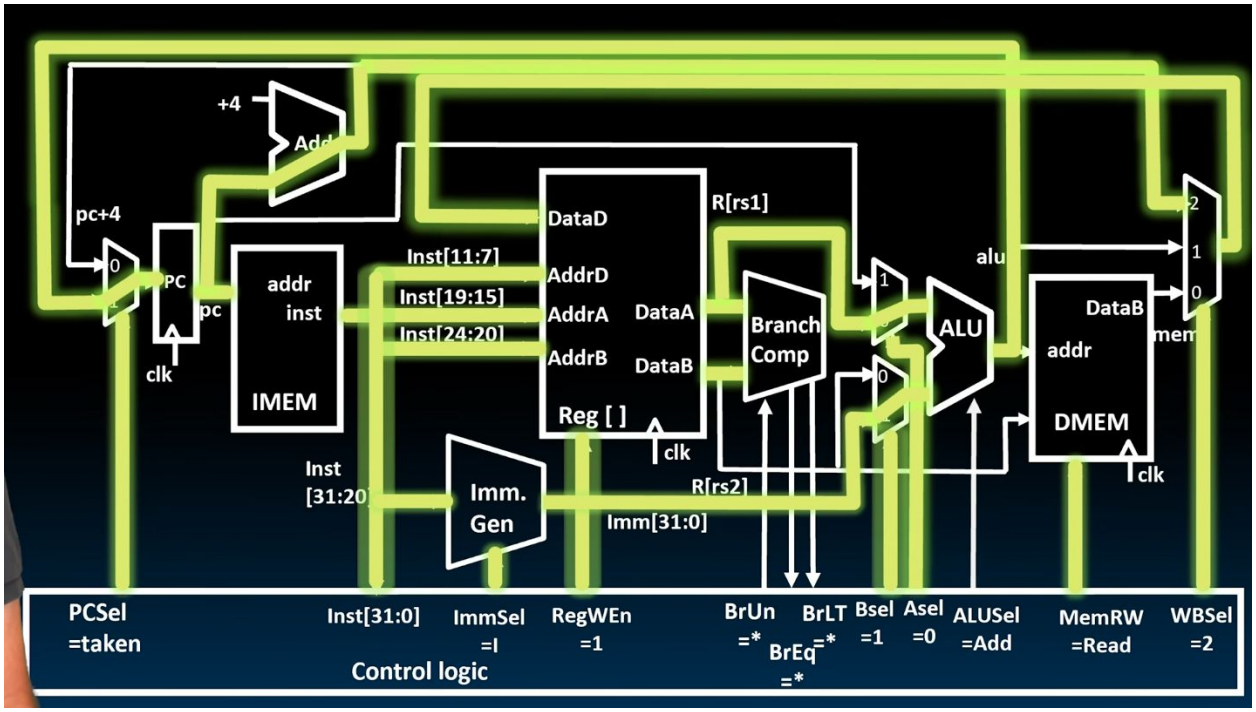
Want to change PC

- Add branch comparison and multiplexer for comparison
- Then we also want to have multiplexer for comparing PC + 4



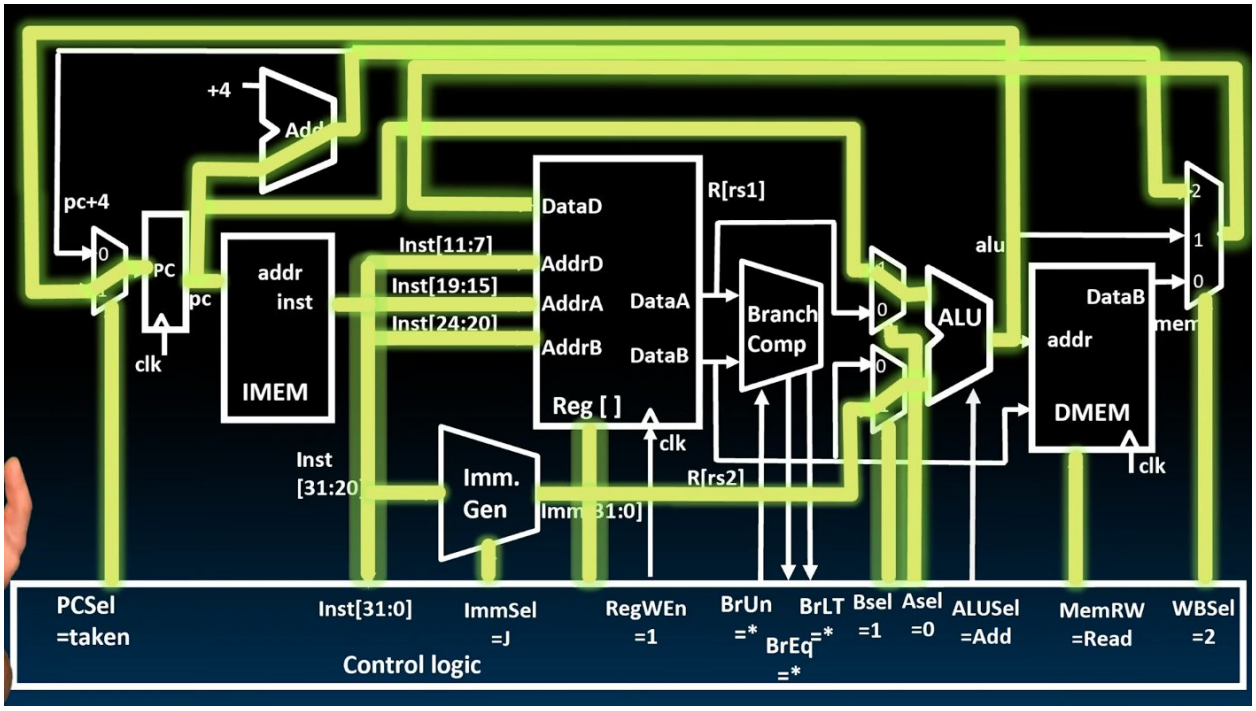
Adding JALR

Add multiplexer for adding to the PC



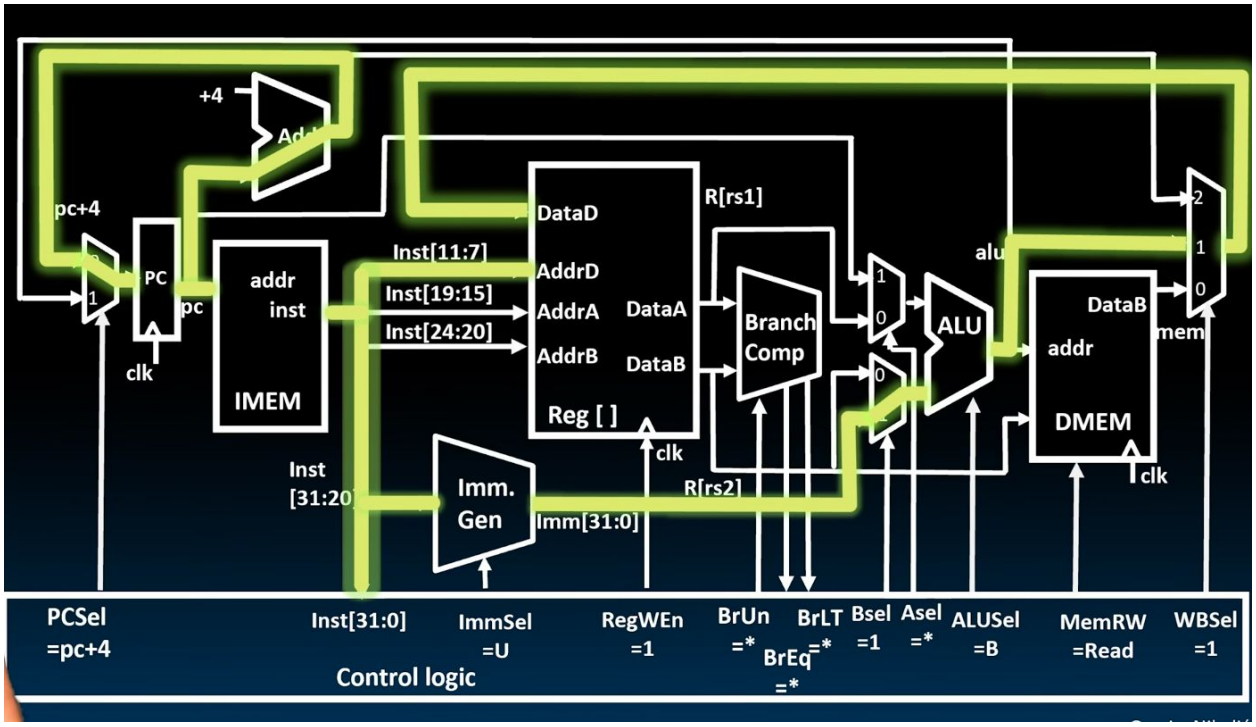

J- Format

Set to always take the J immediate as pc value



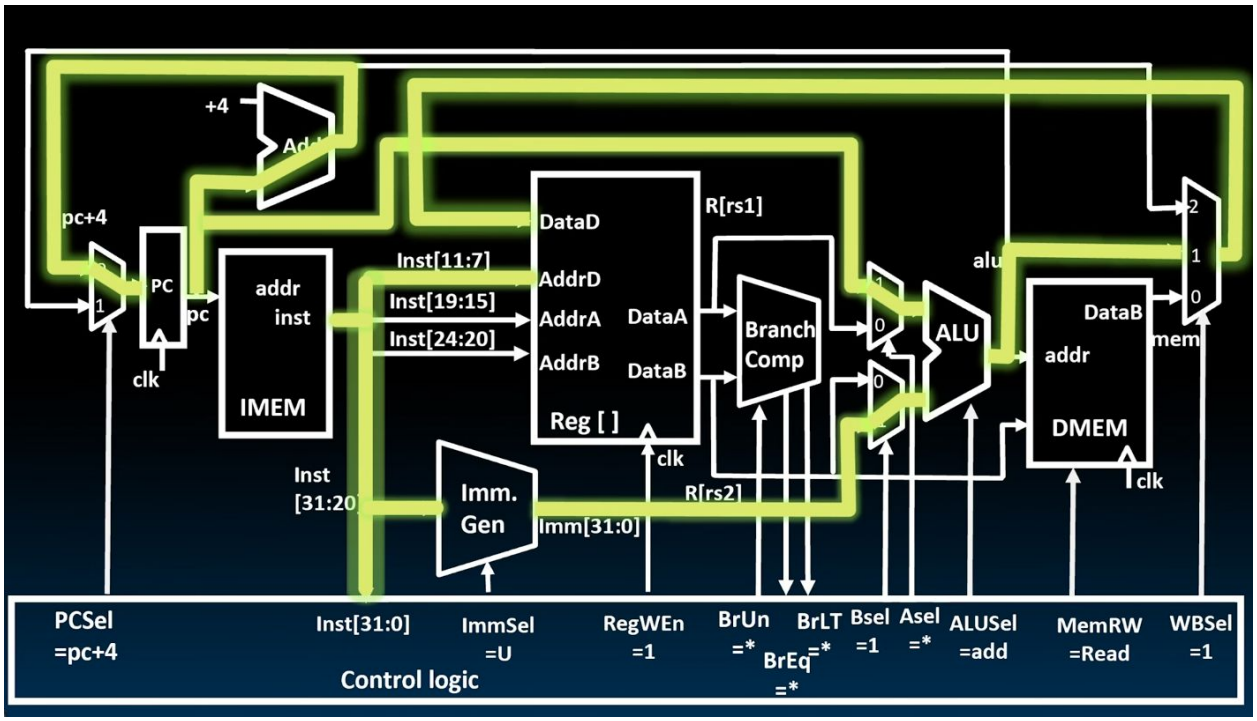

U-format

LUI




Set U to the immediate and point it to the pc

AUIPC




Conclusion

Designed complete datapath

5 phases of execution

- IF, ID, EX, MEM, WB
- Not all instructions are active in all phases

Controller specifies how to execute instructions

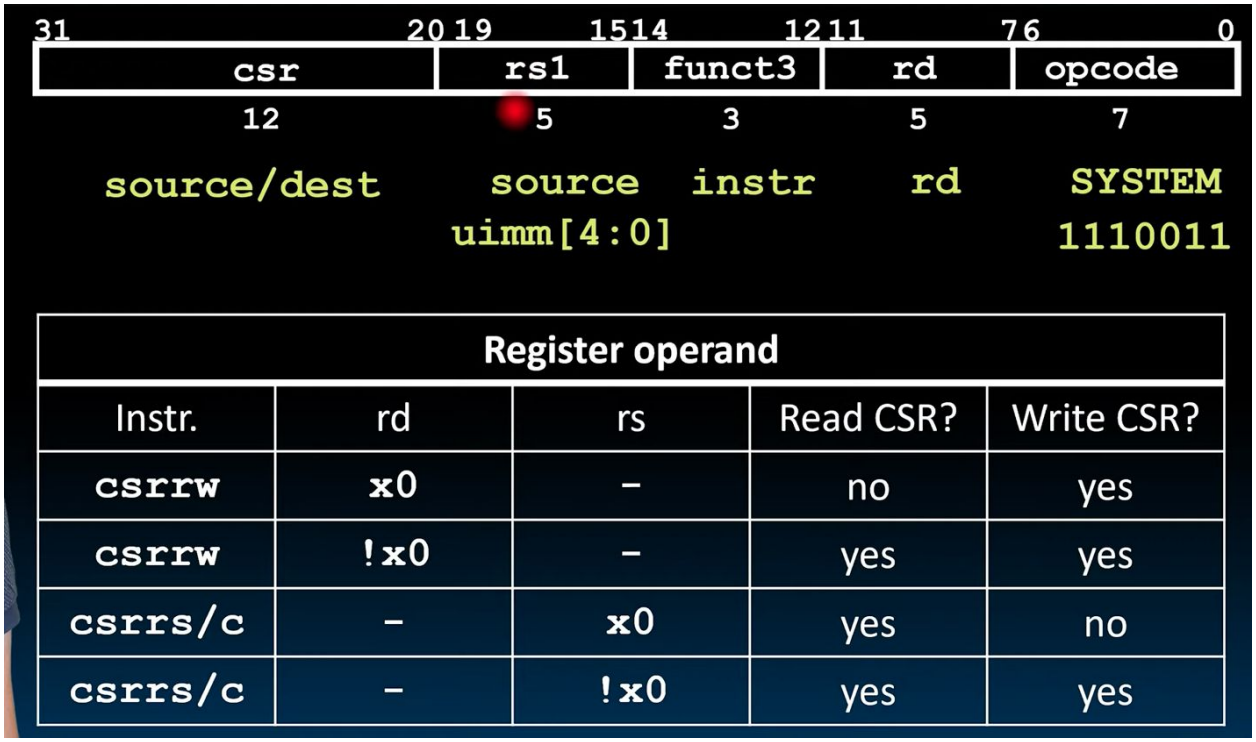
## Week 8: Lecture 20: Single-Cycle CPU Control (10/12)

Control and Status Registers (CSRs)

Separate from register file

- Used for monitoring the status and performance
- Up to 4096 CSRs

Not in base ISA, almost mandatory in every implementation




| Register operand | | | | |
|---|---|---|---|---|
| Instr. | rd | rs | Read CSR? | Write CSR? |
| csrrw | x0 | – | no | yes |
| csrrw | !x0 | – | yes | yes |
| csrrs/c | – | x0 | yes | no |
| csrrs/c | – | !x0 | yes | yes |

Source is 0 extended to

System Instructions

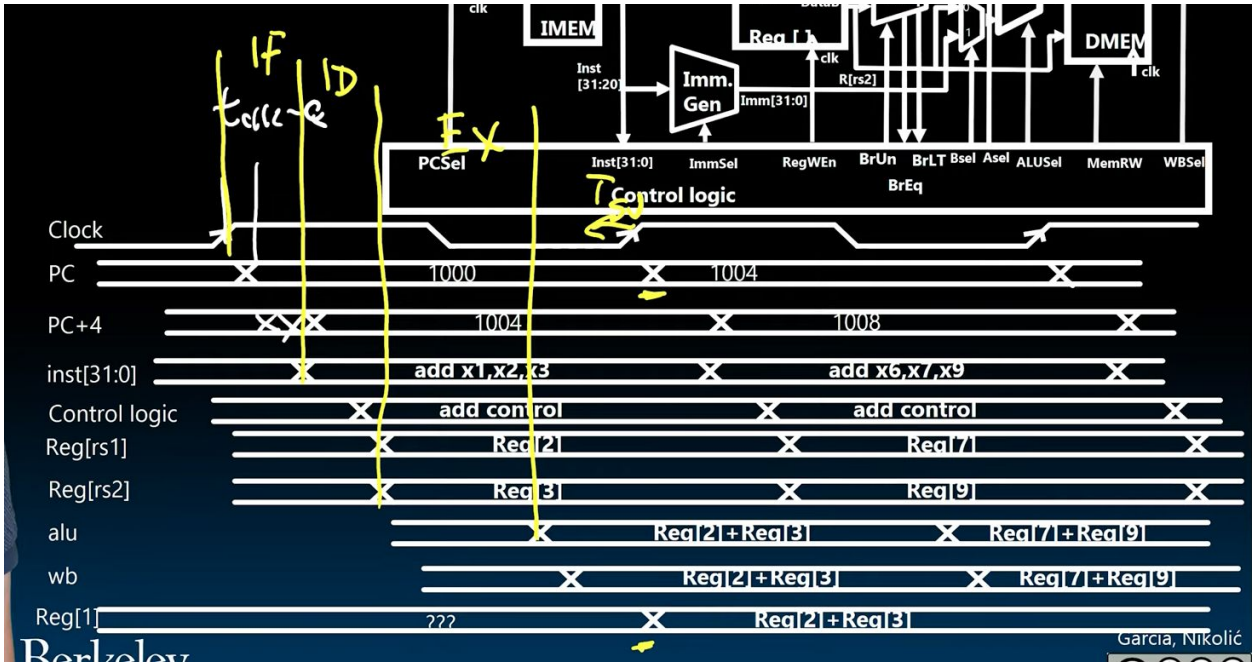
- Ecall - (I-format) requests to supporting execution environment (OS) syscalls
- Ebreak- (I-format) used by debuggers to transfer control to debugging enviro

Datapath Control

sw

- Write value to PC
- Comparable time

Instruction Timing




Critical path = t_clk-1 + t_IMEM + t_Reg + t_mux + t_ALU + t_DMEM + t_mux

| Instr | IF = 200ps | ID = 100ps | ALU = 200ps | MEM=200ps | WB = 100ps | Total |
|---|---|---|---|---|---|---|
| add | X | X | X | | X | 600ps |
| beq | X | X | X | | | 500ps |
| jal | X | X | X | | | 500ps |
| lw | X | X | X | X | X | 800ps |
| sw | X | X | X | X | | 700ps |

- Maximum clock frequency
  - $f_{max}$ = 1/800ps = 1.25 GHz

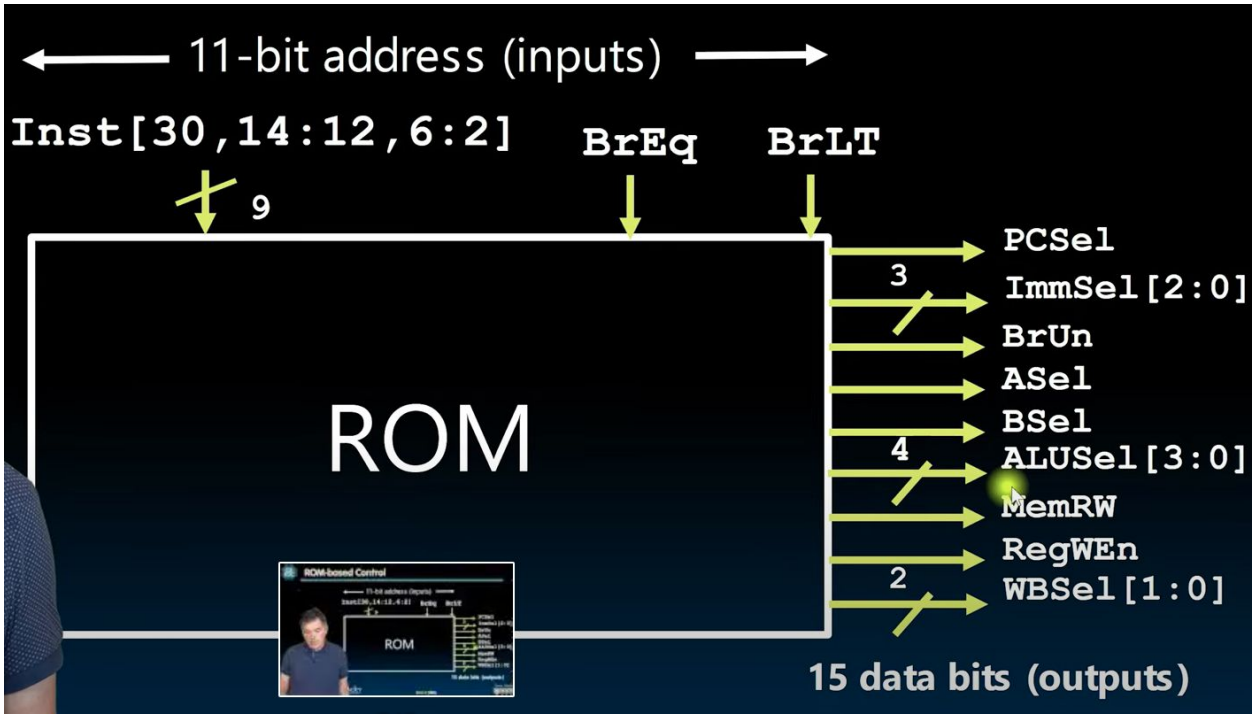
Control Logic Design

Truth Table

| 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 |

ROM

- Read Only Memory
- Can easily reprogrammed




Address Decoder

Conclusion

Built a processor, capable of executing all RISC-V instructions in one cycle each

5 PHases of execution

- IF, ID, EX, MEM, WB

Controller specifies how to execute instructions

- Implemented as ROM or logic
- Read only memory can be used to implement datapath or RV321 Immediate generation
- Lowest 2 bits in opcode equal to 1

## Week 8: Lecture 21: Pipelining I (10/14)

Pipelining

Performance measurement & Improvement

- MInimum cycle time

| IF | ID | EX | MEM | WB | Total |
|---|---|---|---|---|---|
| I-MEM | Reg Read | ALU | D-MEM | Reg W | |
| 200 ps | 100 ps | 200 ps | 200 ps | 100 ps | **800 ps** |

Can we improve performance

- Quicker response time so one job finishes faster
- More jobs per unit time
- Longer batter life
- Bus is better than sports car for delivering more passengers

Computer analogy

- Trip TIme: Program execution time
- Time for 100 passengers: Throughput - # of server requests handled per hours
- Gallons per passenger: Energy per tasks

Processor Performance Iron Law

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

Cycles Per Instruction (CPI)

- Instructions per program
  - Depends on programming language, algorithm
- Average Clock Cycles per Instruction (CPI)
  - ISA, Processor implementation
- Time per Cycle (1/Frequency)
  - Processor microarchitecture
  - Technology

Energy Efficiency

- Maximize performance with least power

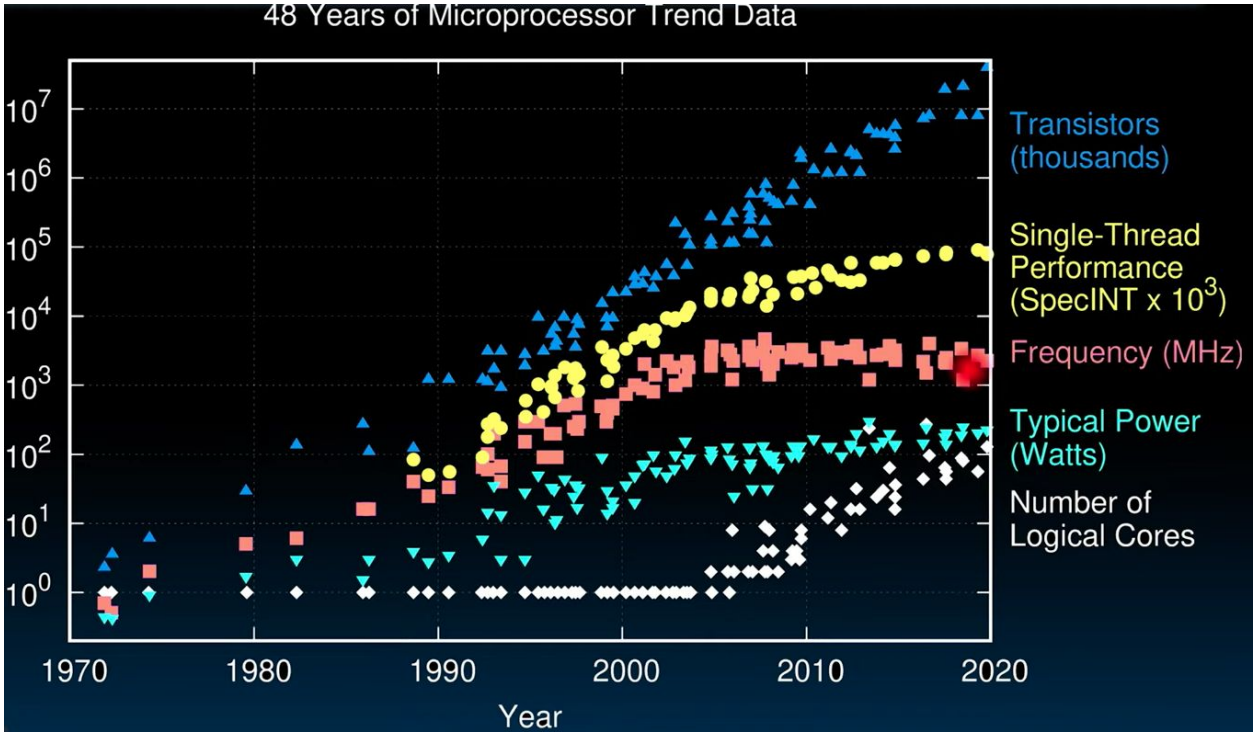
$$\frac{\text{Energy}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Energy}}{\text{Instruction}}$$

Energy = CV^2

- Capacitance depends on technology, processor features # of cores
- Supply Voltage 1V

Next generation processor

- C (Moore's Law) : -15%
- Supply Voltage Vsup: -15%
- Energy consumption : 0 - (1 - 0.85^3) = -39%




End of Scaling,

- can't reduce supply voltage much, further leakage
- Size of transistors capacitance not shrinking as much need to go 3D
  Power becomes growing concern power wall

Energy Iron Law

$$\underset{\textit{(Tasks/Second)}}{\text{Performance}} = \underset{\textit{(Joules/Second)}}{\text{Power}} * \underset{\textit{(Tasks/Joule)}}{\text{Energy Efficiency}}$$

Introduction to Pipelining
Sequentially vs simultaneously
Sequential Laundry

- Pipelining doesn't help latency of single task but it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = number of pipe stages
- Rate is limited by *Slowest* pipeline stage
- Unbalanced lengths of pip stages reduce speedup

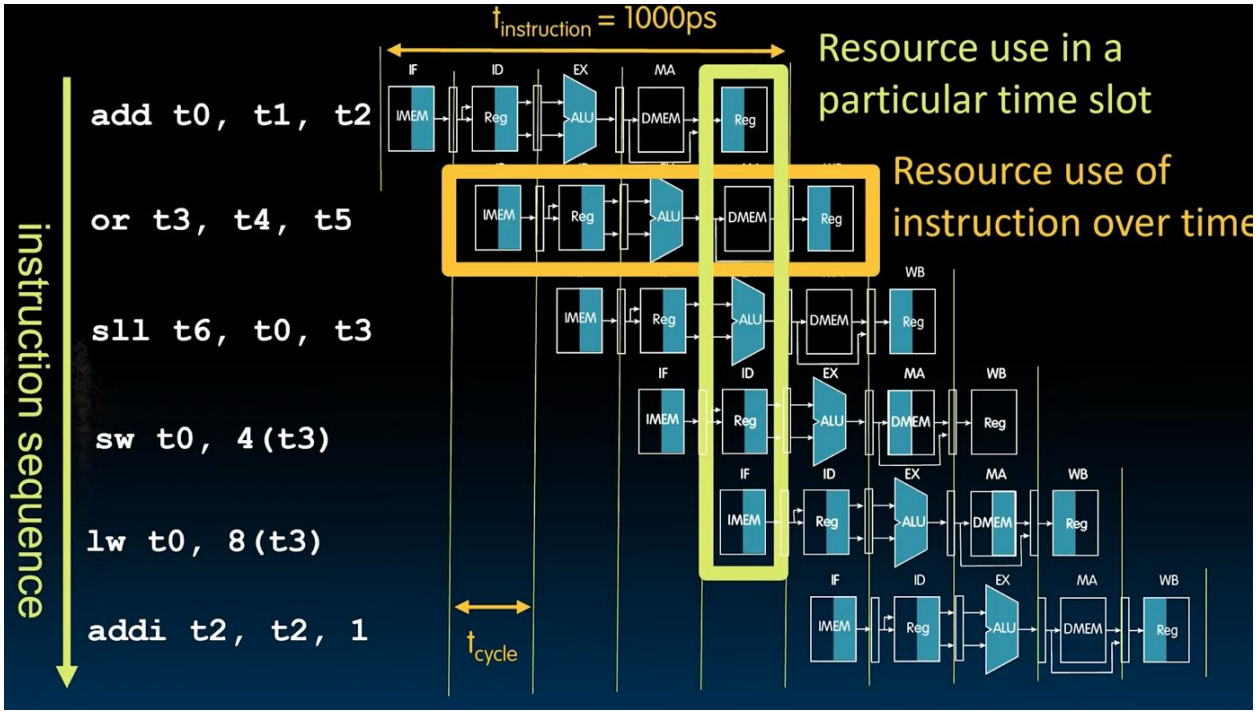
## Week 8: Lecture 22: Pipelining II (10/16)

Piplining RISC-V

| Phase | Pictogram | $t_{step}$ Serial | $t_{cycle}$ Pipelined |
|---|---|---|---|
| Instruction Fetch |   | 200 ps | 200 ps |
| Reg Read |   | 100 ps | 200 ps |
| ALU |   | 200 ps | 200 ps |
| Memory |   | 200 ps | 200 ps |
| Register Write |   | 100 ps | 200 ps |
| $t_{instruction}$ |   | 800 ps | 1000 ps |

Separate stages with registers
Clock cycle for pipelines must match and so latency on the single cycle increases

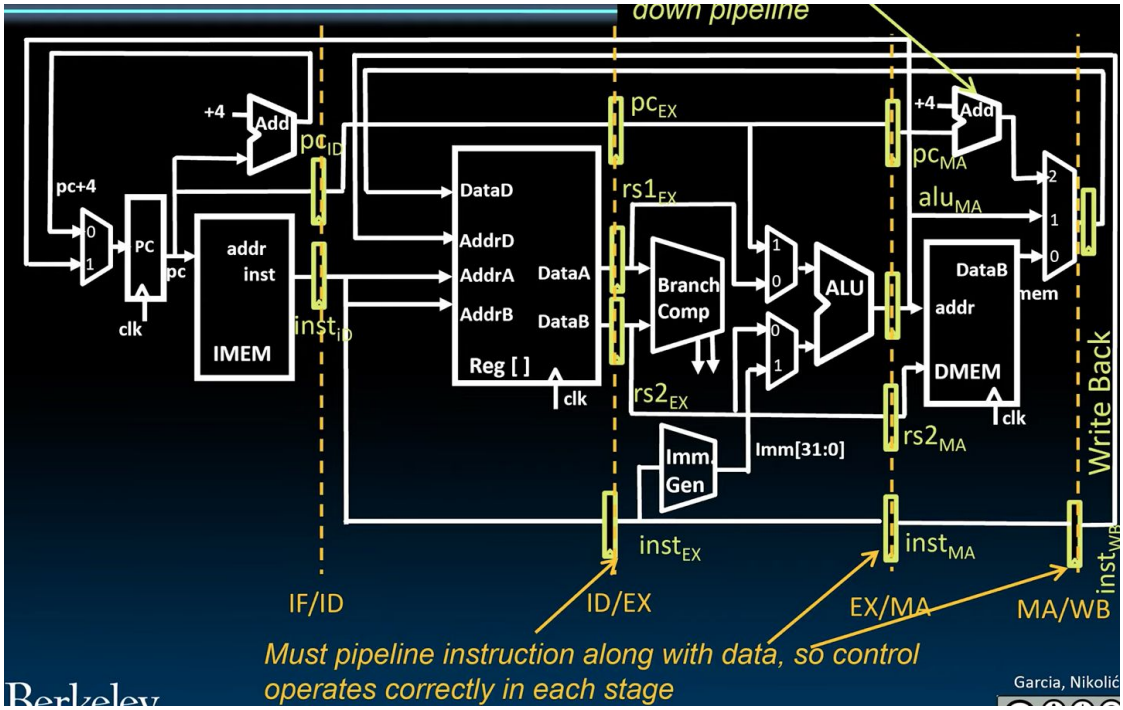
More throughput 4x




Pipeline Hazards

1. Structural hazard
    a. Required resource is busy
    b. Problem: two of more instructions compete for access to single resource
        i. Solution 1: Instructions take it in turns to use resource, some instructions have to stall
        ii. Solution 2: Add more hardware to machine
2. Data Hazard
    a. Data dependency between instructions, need to wait
3. Control Hazard
    a. Flow of execution depends on previous instructions

Pipelining Datapath

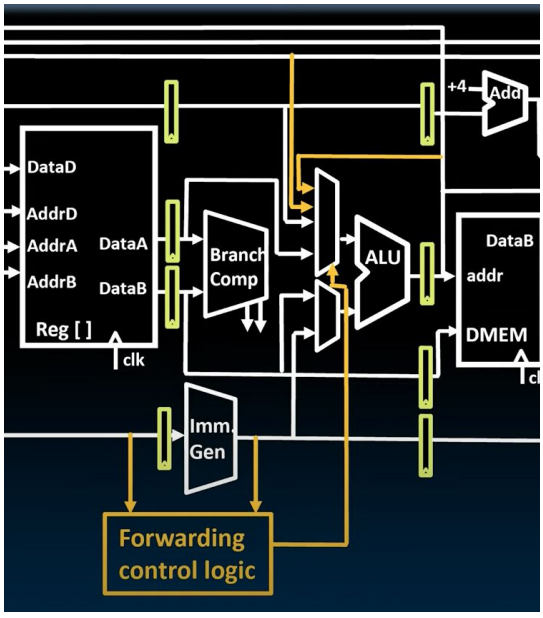



Data Hazards

Bubbles to correctly align: nop

Forwarding (aka Bypassing)

- Use result when it is computed , requires extra connections in the datapath
- Forwarding control logic




Can arrange code or insert nps to avoid hazards and stalls

## Week 9: Lecture 23: Pipelining III (10/19)

Load Data Hazard

Hardware forwarding helps data hazards
Load Data Hazards
- Must do 1 cycle stall, finishes after DMEM
- In order to do nop (Load delay slot)
    - Put unrelated instruction into load delay slot
    - We deactivate all controls for the rest and run again

Control Hazards
Don't know whether branch will be taken or not
Need two cycles to convert to NOP
Reducing Branch Penalties
- Use "branch prediction" to guess which way branch will go earlier in pipeline
- Only Flush pipeline if branch prediction was incorrect
- Predict by seeing if the branch was executed last time

Superscalar
Increasing Processor Performance
1. Clock rate
    a. Limited by technology and power dissipation
2. Pipelining
    a. Deeper Pipline 15 stages
    b. Less work per stage, more potential for hazards
3. Multi-issue "superscalar" processor
    a. Multiple instructions per clock cycle Instructions per Cycle (IPC)
4. Out of order execution
    a. Reorder instructions

Superscalar Processor
- Decode multiple instructions
- Reservation station

Calculating CPI


RISC-V ISA designed for pipelining
- All instructions are 32 bits easy to fetch and decode in one cycle
- Few and regular instruction formats
- Load/store addressing
    - Calculate address in 3rd stage
- Alignment of memory operands

Built a processor, controller specifies how to execute, pipelining improves performance

## Week 9: Lecture 24: Caches I (10/21)

Switched to base 10 but hard drives are base 2

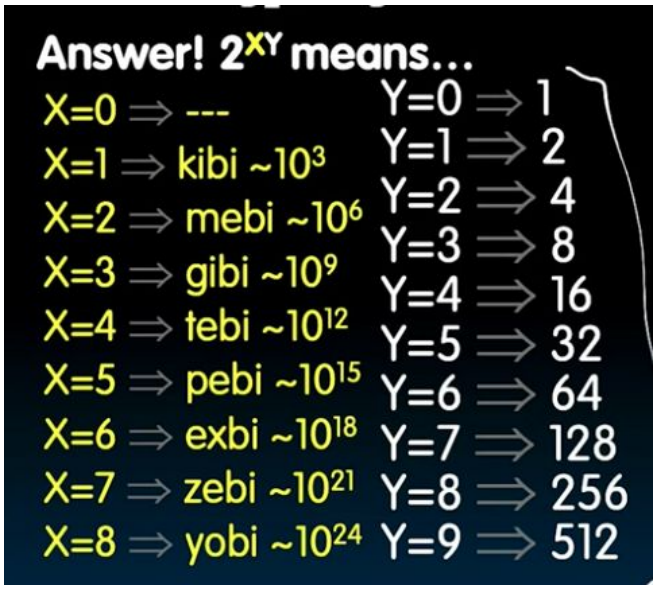
Kibi, mebi

- International Electrotechnical Commission (IEC) set standard
- 

| Name | Abbr | Factor |
|---|---|---|
| kibi | Ki | $2^{10} = 1,024$ |
| mebi | Mi | $2^{20} = 1,048,576$ |
| gibi | Gi | $2^{30} = 1,073,741,824$ |
| tebi | Ti | $2^{40} = 1,099,511,627,776$ |
| pebi | Pi | $2^{50} = 1,125,899,906,842,624$ |
| exbi | Ei | $2^{60} = 1,152,921,504,606,846,976$ |
| zebi | Zi | $2^{70} = 1,180,591,620,717,411,303,424$ |
| yobi | Yi | $2^{80} = 1,208,925,819,614,629,174,706,176$ |

KMGTPEZY

The way to remember #s

- 2^XY means
- 

- 
- 2^64 is  16 exbi

<u>Library Analogy</u>

- Why are Large Memories slow
- Larger the libraries worsen both delays
- We want large yet fast memory
- DRAM gap

<u>Memory Hierarchy</u>

Memory Caching

- Cache is a copy of a subset of main memory
- Implemented with same IC processing as the CPU
- Cache in between CPU core and Physical memory
    - Faster than memory, more expensive than memory, smaller
- Increasing distance from processor in access time

Typical Memory Hierarchy

- Present processor as much memory at cheapest technology with speed offered by fastest technology

<u>Locality, Design, Management</u>

Memory Hierarchy Basis

- Caches work on the principles of temporal and spatial locality
    - Temporal (locality in time) If we use it now, chances are we'll want to use it again soon
    - Spatial (locality in space) chances are we'll use the memory neighbors

What to do about locality

- Keep most recently accessed data items closer to the processor
- Move blocks consisting of contiguous words closer to the processor

Hierarchy Managed

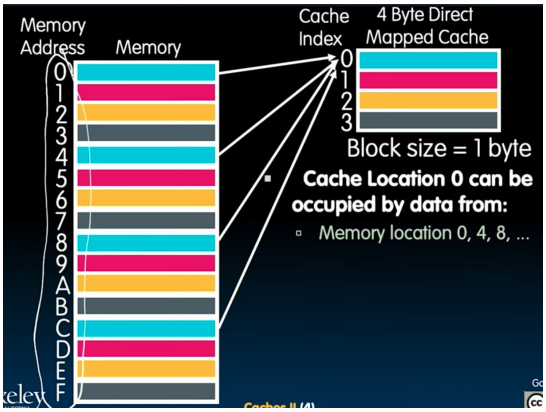
- Registers to memory
- Cache to memory
- Main memory to disks

## Week 9: Lecture 25: Caches II (10/23)

<u>Direct Mapped Caches</u>

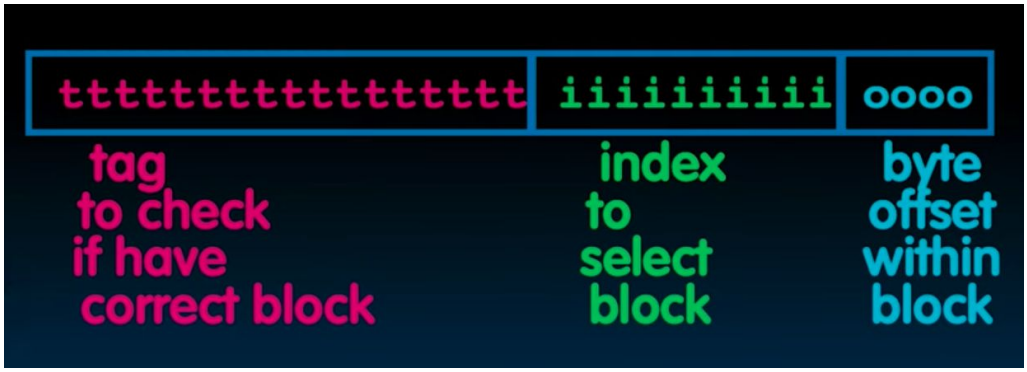
Direct-mapped cache - each memory address is associated with one possible block within a cache

- We only need to look in a single location in the cache for the data if it exists
- Draw memory so that it is the same width as the cache
- Block size = 1 byte




Direct Mapped Cache

- Only need to look at last 2 bits for cache with 4
- Cache should be a word long
- Each cache has a tag
- Some bytes tell you which blue (cache number) +and others tell you if it is blue




Index
- Specifies cache index, which "row" block of the cache

Offset
- Which byte within the block (column)

Tag
- Used to distinguish between memory that map to same location

Tag index offset
- Index and offset determines Area, height is index, offset is width, bytes per block
- Area (cache size in Bytes) = Height (# blocks) * Width (size of one block, B/block)
- 2 ^ (H + W) = 2^H * 2^ W

<u>Direct Mapped Cache Example</u>

8B of data 2 blocks each

Offset (specify correct bye within a block)
- Block contains 2 bytes
- Need 1 bit to specify correct byte

Index (specify correct block in cache)
- # blocks/ cache
- 2^3 / 2^1 = 2^2
- 2 bits to specify number of blocks

Tag (Use remaining bits as tag)
- Tag length = addre length - offset - index
- = 32 - 1 - 2
- 3 bits inside the cache since 8 bytes
- Tag is "cache number"

Memory Access with Cache
1. Processor issues address 1022 to Cache
2. Cache checks to see if has copy of data at address 1022
    a. If finds a match, reads and sends to processor
    b. No match (miss) cache sends address to memory
        i. Reads and puts into cache
3. Loads 99 into register t0

Solving Cache problems
- Starts from top right and moves left due to offset when offset adds one

- Cache is a mirror
- Can stride across blocks, rows, cols,

Cache Terminology

- When reading memory, 3 things can happen
    - Cache hit: cache block is valid and contains proper address
    - Cache miss: nothing in chase in appropriate block, so fetch from memory
    - Cache miss, block replacement: wrong data in cache at appropriate block so discard and fetch
- Cache Temperatures
    - Cold: cache empty
    - Warming: cache filling with values you'll hopefully be accessing again soon
    - Warm: cache is doing its job fair % of hits
    - Hot: cache is doing very well, high % of hits

Cache Terms

- Hit rate: fraction of access that hit ini the cache
- Miss rate: 1 - hit rate
- Miss penalty: time to replace a block from lower level in memory hierarchy to cache
- Hit time: time to access cache memory
- $ = cache

One More Detail: Valid bit

- When start a new program, cache does not have valid information for this program.
- Need an indicator whether this tag entry is valid for this program
- Add a valid bit to cache tag entry starts with 0 not valid

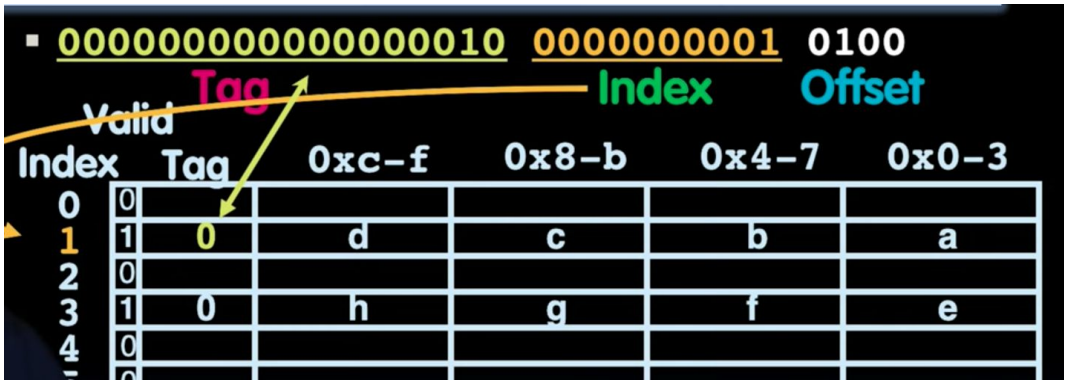
Conclusion

- Direct-mapped cache
- Mechanism for transparent movement of data among levels of a memory hierarchy

## Week 10: Lecture 26: Caches III (10/26)

1. Start with Index
2. Check valid bit
    a. If 0, cache miss
        i. Bring back row neighbors for spatial locality (block replacement)
        ii. Set valid bit to 1
    b. If 1, valid
        i. Check tag
        ii. If tag matches, then check offset

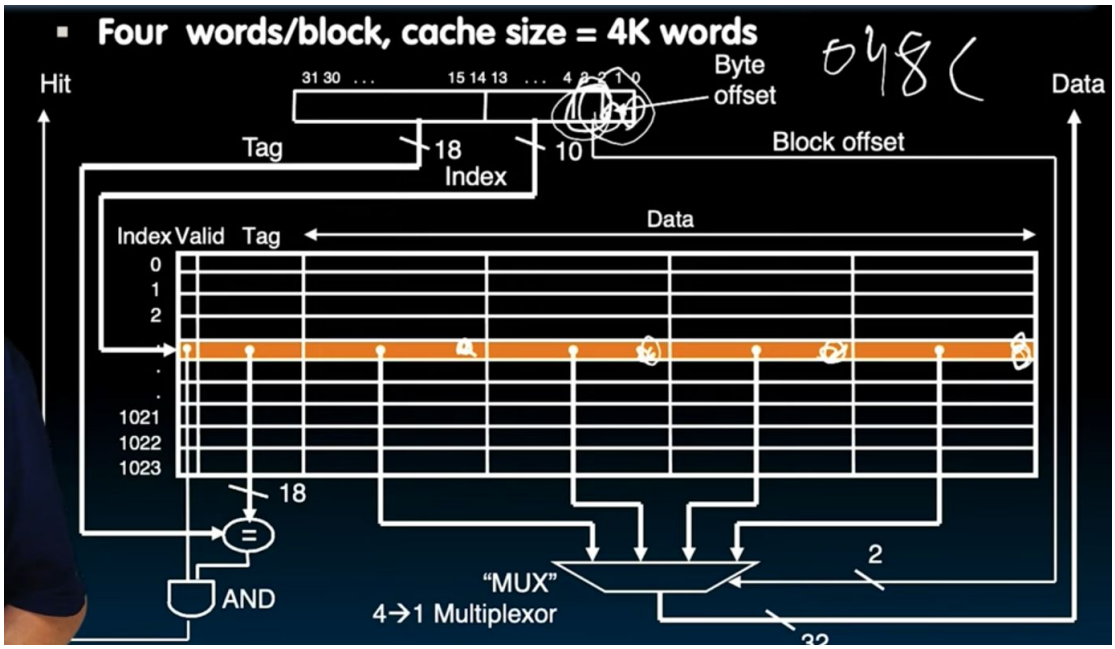
IVTO (index, valid, tag, offset)




Writes, Block Sizes, Misses

Multiword-Block Direct-Mapped Cache

- Byte offset and block offset



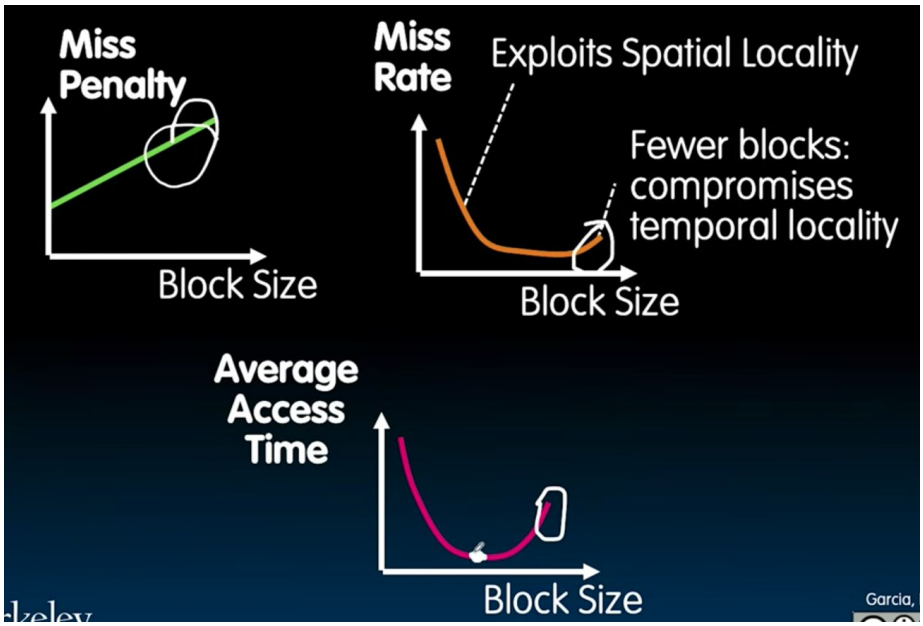

What to do with write hit

- Write through update both cache and memory
- Write back: update word in cache block, allow memory to be "stale", memory and cache are inconsistent
- Add dirty bit to block needs to be updated when block is replaced
- Write allocate: bring block into the cache after a write miss

Block SIze Tradeoff

- Benefits of larger block size
    - Spatial locality can access other nearby words soon
- Drawbacks of larger block size
    - Larger block size means larger miss penalty, takes longer time to load a new block

Block Size tradeoffs




Types of Cache Misses

- Compulsory Misses
  - Ever block has at least one compulsory miss, have to get it
- Conflict Miss
  - Two distinct memory addresses map to the same cache location, keep overwriting

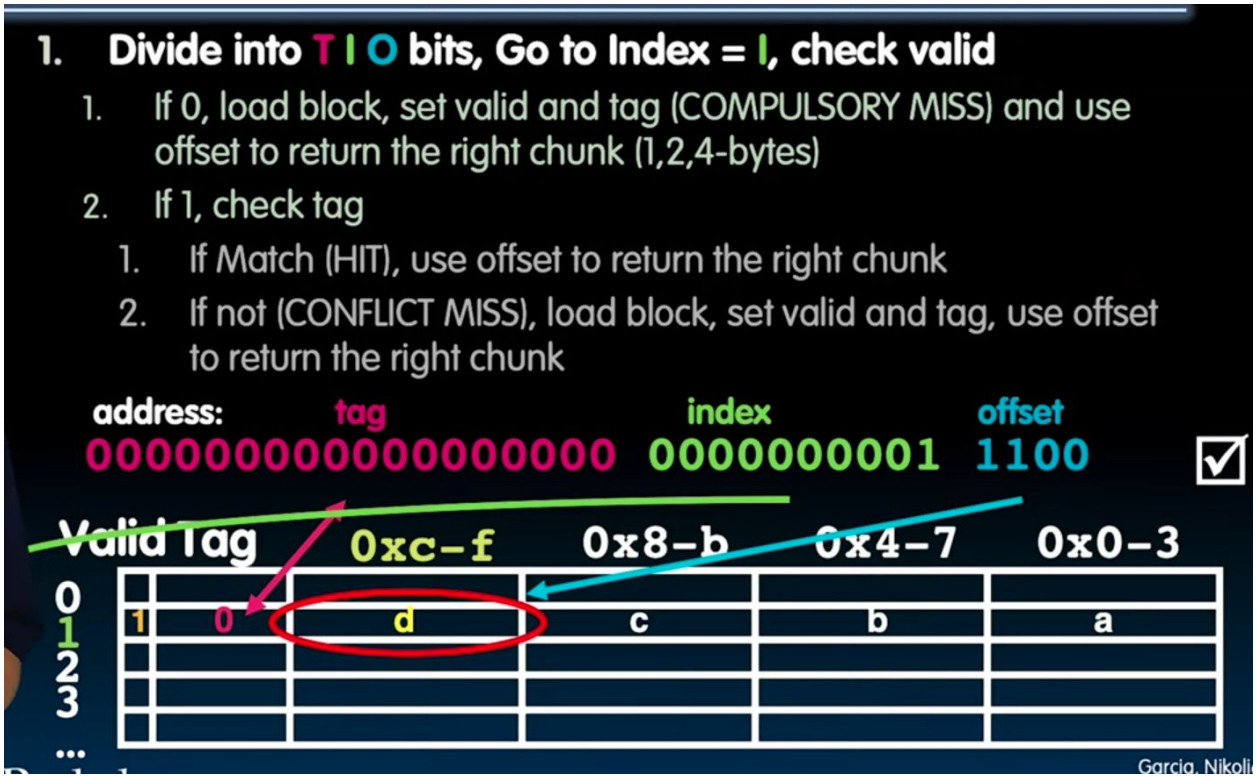
<u>Fully Associative Caches</u>

Index non-existent can go in any row

- Capacity Miss
  - Miss that occurs because cache has limited size
  - Most common for fully associative caches

Categorizing misses

- Consider infinite size fully associative cache any miss would be compulsory miss
- Consider finite sized cache with fully associativity would be *capacity misses*
- Consider finite size cache with finite associativity would be *conflict misses*




## Week 10: Lecture 27: Caches IV (10/28)

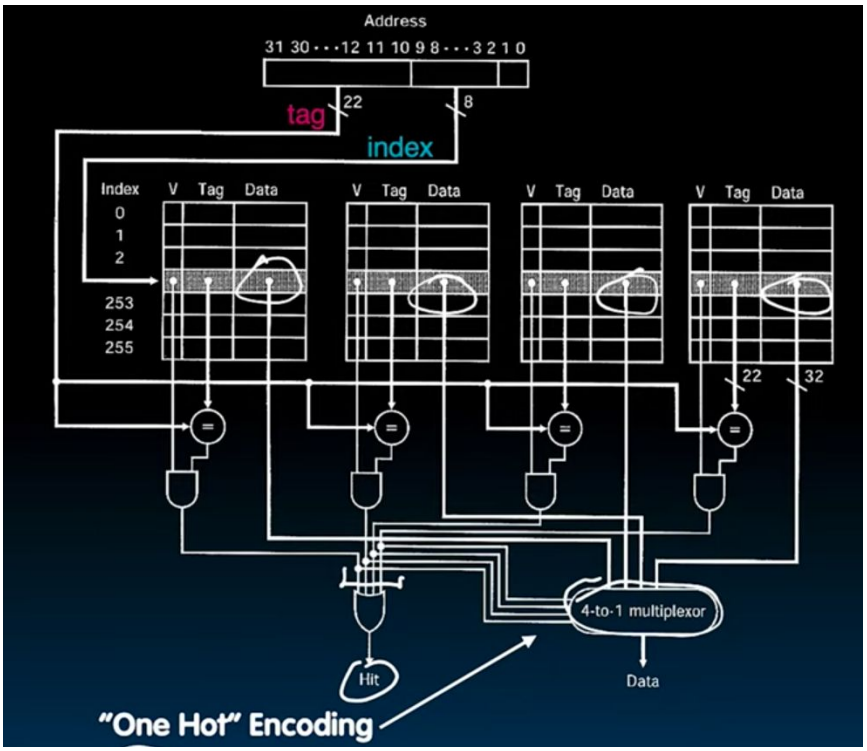
Set-Associative Cache

N-Way Set associative Cache

- Size of cache: # sets * N blocks/set * block size
- 2- way set associative cache : 2 sets, 2 blocks in set
- Basic Idea: direct-mapped w/ respect to sets, each set is fully associative with N blocks in it
- Avoids a lot of conflict misses

Direct-Mapped is 1 way set assoc

Fully Assoc if its M-way set assoc



Block Replacement with Example

Block Replacement Policy

1. Least Recently Used (LRU)
    a. Idea: cache out block which has been accessed least recently
    b. Pro: temporal locality - recent past use implies likely future use
    c. Con: requires more complicated hardware for 4 way and much time to keep track
2. FIFO
    a. Idea: ignores accesses, just tracks initial order
3. Random
    a. If low temporal locality of workload, works ok

Block Replacement Example: LRU

- In 2 block have a lru bit that keeps track of least recently used

Average Memory Access Time (AMAT)

Big Idea

- Minimize: Average Memory Access Time = Hit Time + Miss Penalty * Miss Rate

Improve Miss Penalty

- Another cache between memory and the processor cache: Second level (L2) Cache

Analyzing Multi-level cache hierarchy

- Recursive nature
- Each goes down one level with L2, L3


Typical Scale

- L1
    - Size: tens of KB
    - Hit time: complete in one clock cycle
    - Miss rates: 1 - 5%
- L2
    - Size: hundreds of KB
    - Hit Time: Few clock cycles
    - Miss rates: 10-20%

An Actual CPU

- L1 and L2 inside core, L3 inside CPU for i7

Conclusion

- Big Idea; if something is expensive, but we want to do it repeatedly do it once and cache the result
- Cache design choices:
  - Size of cache: speed v. capacity
  - Block size
  - Write Policy (write through v write back)
  - Associativity choice of N
  - Block replacement policy
  - 2nd level cache
  - 3rd level cache
- Use performance model to pick between choices

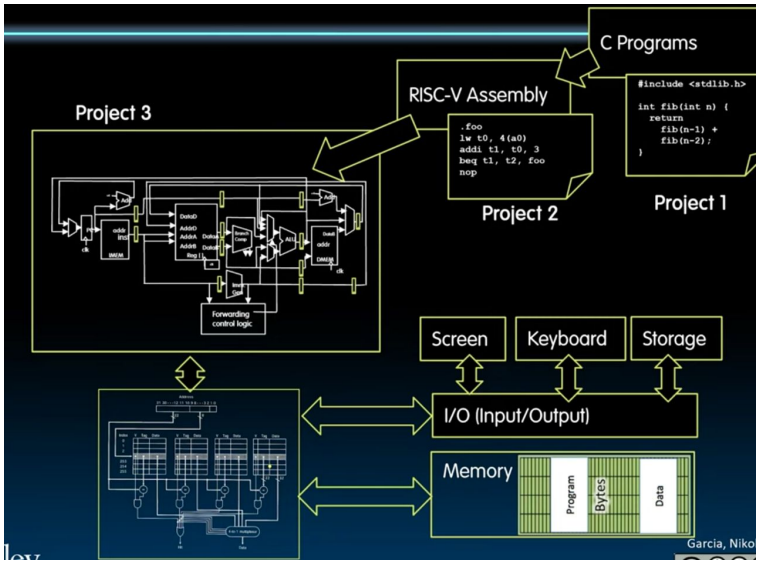
## Week 10: Lecture 28: OS & Virtual Memory (10/30)

Intro

Can have small single computers like raspberry pi

Codebases

- Space Shuttle - 400k
- Boeing 787 - 14 mil
- Facebook - 62 mil
- Car Software - 100 mil
- Mouse DNA basepairs - 120 mil
- Google - 2 billion




Operating system basics

What does the OS do?

- First thing that runs when computer starts
- Finds all devices and control
- Starts services: file system, TTY (keyboard)
- Loads, runs, and manages programs, multiple programs at the same time

Core of the OS

- Provide isolation between running processes
- Provide Interaction with outside world

OS use from Hardware

- Memory translation:
  - mapping from virtual memory to physical address, load, store is to virtual address but converts to physical address
- Protection and privilege
  - Split processor into User and supervisor

What happens at Boot?

1. Basic Input Output System (BIOS)
    a. Find a storage device and load first sector (block of data)
2. Bootloader
    a. Load the OS kernel from disk in a location in memory and jump into it
3. OS Boot:
    a. Initialize services driver
4. Init
    a. Launch an application that waits for input, terminal desktop

<u>Operation System Functions</u>

Launching Applications

- Applications are called "processes" in most OSs
    - Thread: shared memory
    - Process: separate memory
    - Both threads and processors run simultaneously
- Apps started by another process (shell) calling an OS routine (syscall)
- Loads executable file from disk and puts instructions & data into memory
- Set argc argv

Supervisor Mode

- OS enforces resource constraints to applications
- To help protect the OS from the application, CPUs have a supervisor mode
    - A process can only access a subset of instructions
    - Supervisor mode errors can cause catastrophic errors

Syscalls

- Set up function arguments in registers
- Raise software interrupt
- OS will perform operation and return to user mode

Interrupts, Exceptions

- Interrupt:
    - Something external to the running program (key press, disk IO)
    - Can answer whenever convenient, just don't wait too long
- Exception:
    - Something done by the running program
    - Must handle exception precisely on instruction that errored
- ECALL
    - Trigger an exception to the higher privilege
- EBREAK
    - Trigger an exception within the current privilege
- Trap
    - Action to handle interrupt or exception
    - Handles before the next instruction happens

Trap Handling

- Similar to pipeline hazards

1. Complete executions of instructions before exception

2. Flush instructions in pipeline
3. Optionally store exception cause in status register
4. Transfer execution to trap handler
5. Optionally return back to program

Multiprogramming

- Switches between processes very quickly in processor

Protection, Translation, Paging

- Application never overwrites another application's memory
- Solution:
    - Gives each process the illusion of a full memory address space always start at fixed address

## Week 11: Lecture 29: Virtual Memory I (11/2)

Virtual Memory Concepts

Virtual Memory

- Provides program with illusion of very large main memory
- Run programs larger than DRAM
- Each process thinks it has all the memory to itself
- Important for protection

Virtual vs Physical Addresses

- Memory manager maps virtual to physical addresses

Address Spaces

- Virtual Address Space
    - Addresses user program knows about
- Physical Address space
    - Maps to real physical addresses

Physical Memory and Storage

Memory

- DRAM - Volatile (when turned off, forgets data)
    - Successive accesses are faster

Storage - "Disk" (Non-volatile)

- SSD
    - Access: 40-100 us faster
    - $0.05 - 0.5 / GB
    - Made with transistors, like ginormous register file
- HDD
    - Access < 5 - 10ms
    - $0.01 - 0.1/ GB
    - Disk, mechanical, rotates and magnitizes

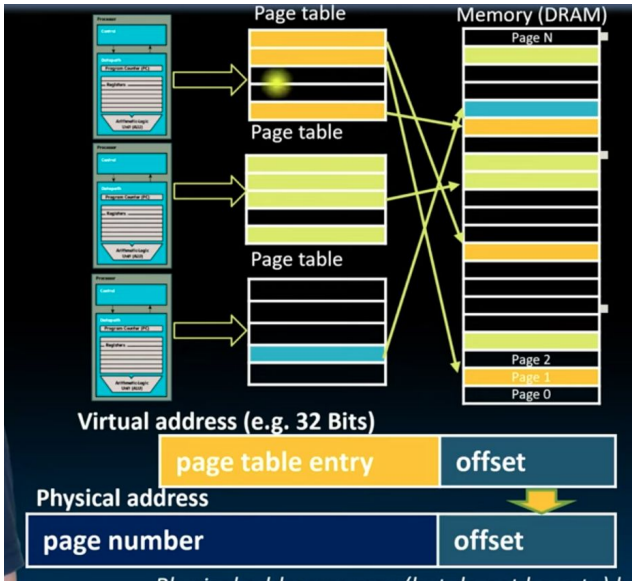
Flash Memory

- 3D array of bit cells

Memory Manager

Virtual Memory

- 100+ processes
- Maps part of memory




Responsibilities of Memory Manager

1) Map Virtual to physical address
2) Protection
    a) Isolate memory between processes
    b) Get dedicated private memory
3) Swap memory to disk

<u>Paged Memory</u>

Memory Manager and Paged Memory

- Physical Memory (DRAM) is broken into pages
- Virtual address has page number and offset

Paged Memory Address Translation

- OS keeps track of which process is active
- Extracts page number from virtual address
- Looks up page address in page table
- Computer physical from virtual
- Different pages in DRAM keeps them from accessing each others memory
- Sharing is possible with bit (write protection)

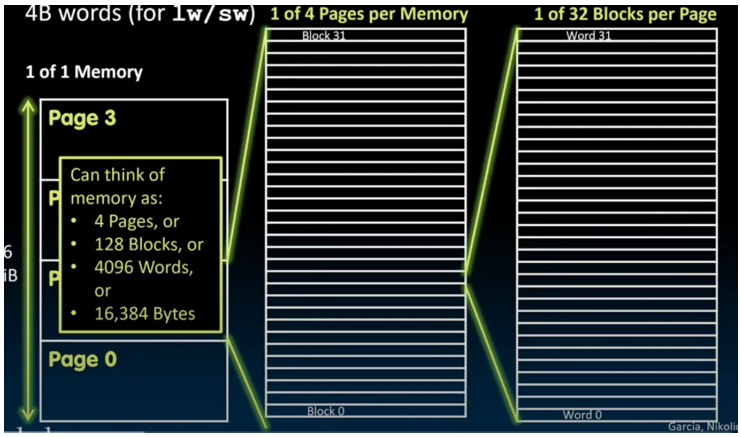
Where Do Page Tables Reside?

- 32 bit virtual address 4-KiB
- Parts of page table will be in cache

<u>Page Faults</u>

Blocks vs Pages

- Caches deal with individual blocks
- In VM, we deal with individual pages




- Memory > Pages > Blocks > Words > Bytes

Memory Access

- Check page table entry
    - Valid?
        - Yes, valid > in DRAM
            - Yes, in DRAM: read write

        - No, on disk: allocate new page in DRAM
            - If out of memory, evict a page from DRAM
            - Store evicted page to disk
            - Read page from disk into memory
            - Read/Write
    - Not Valid
        - Allocate new page in DRAM
        - If out of memory, evict a page and read/write

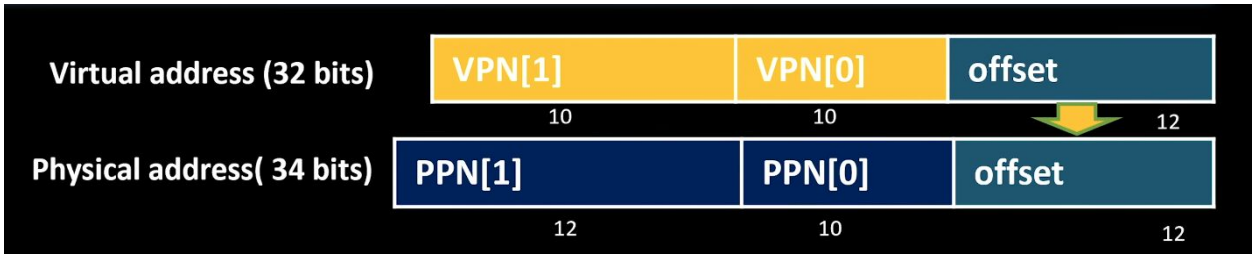
Page Fault
- Treated as exceptions

Write-Through or Write-Back
- DRAM acts like "cache" for disk
- Use write-back because Disk is too slow

## Week 11: Lecture 30: Virtual Memory II (11/4)

Options for Page Tables
- Increase page size
- Hierarchical page tables
- Most programs only use fraction of memory

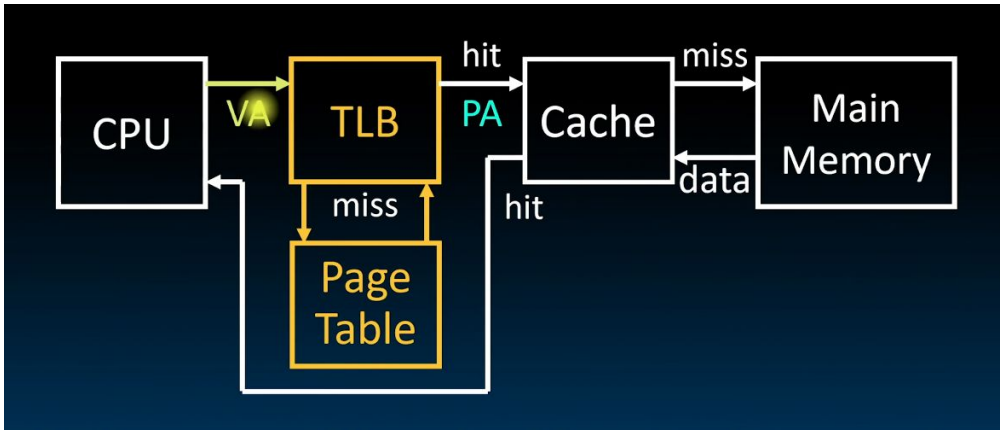
Hierarchical Page Table
- Split into 3 sections, p1, p2, offset
- Have level 1 and level 2 page tables




Translation Lookaside Buffers
- Address translation is very expensive, like a small cache

TLB Designs
- Typically 32-128 entries, usually fully associative
- Random or FIFO replacement policy
- "TLB Reach" : Size of largest virtual address space that can be simultaneously mapped by TLB

Where are the TLBs Located
- Which to check first: cache or TLB
- Ca cache hold requested data if corresponding page is not in physics memory? No
- TLB translates the VA



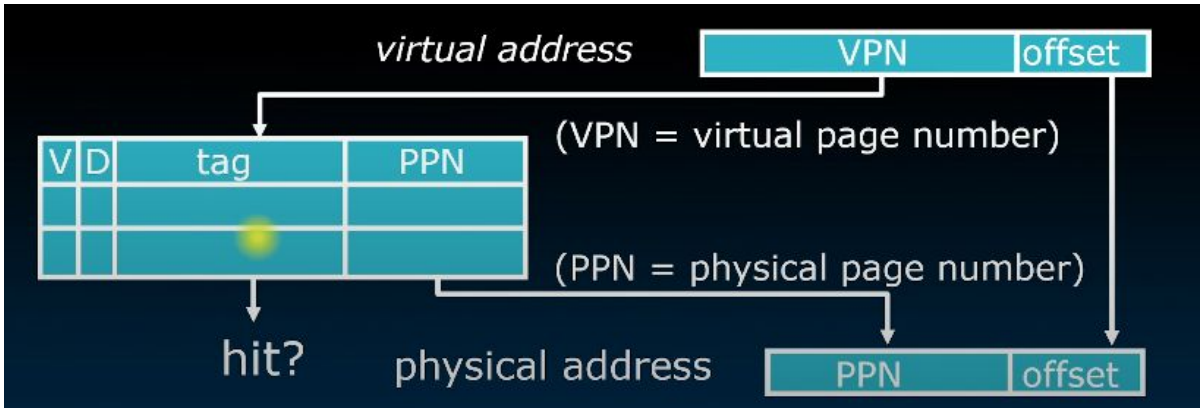

-

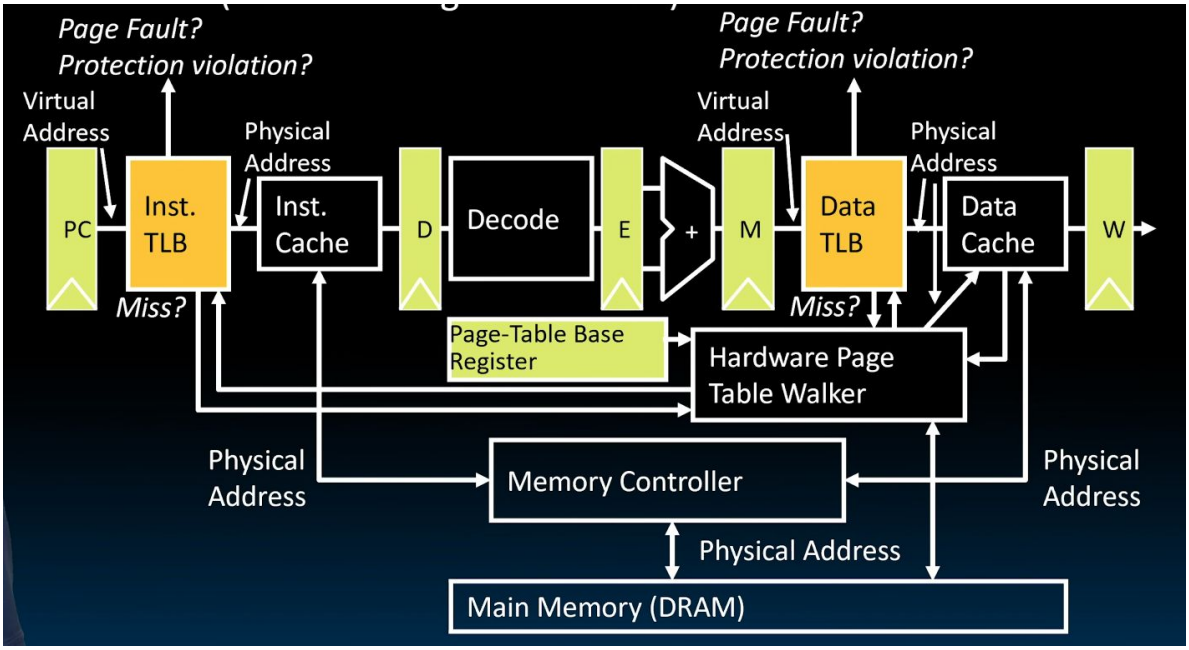
<u>TLBs in Datapath</u>

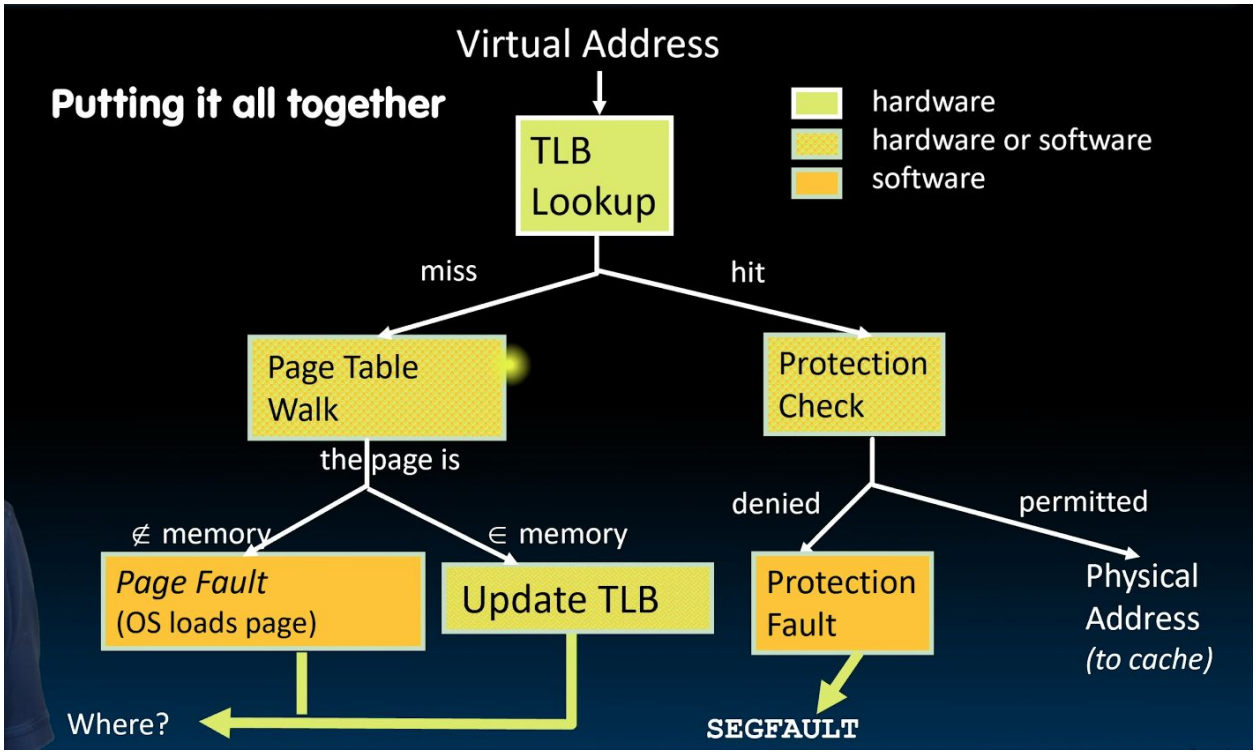
VM-related Events in Pipeline

- Handling a TLB miss needs hardware

Handling a page fault needs a precise trap so software can resume






Modern Virtual Memory Systems

Protection & Privacy

- Several users/processes each with their own private address space

Demand Paging

- Provides the ability to run programs larger than primary, hides configuration

Context Switching

- Def: Changing of internal state of processor (switching between processes)
- Save register values

VM Performance

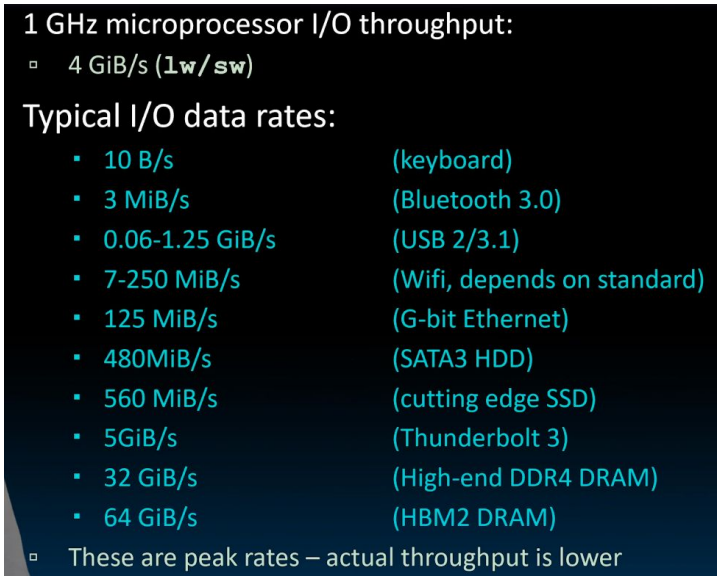
| Cache version | Virtual Memory version |
|---|---|
| Block or Line | Page |
| Miss | Page Fault |
| Block Size: 32-64B | Page Size: 4K-8KiB |
| Placement: Direct Mapped, N-way Set Associative | Fully Associative |
| Replacement: LRU or Random | Least Recently Used (LRU), FIFO, random |
| Write Thru or Back | Write Back |

Level below the main memory

Impact of Paging on AMAT

- L1 cache hit = 1 clock cycles, hit 95% of accesses

- L2 cache hit = 10 clock cycles, hit 60% of L1 misses
- DRAM = 200 clock cycles
- Disk = 20,000,000 clock cycles
- Trashing is slower




## Week 11: Lecture 31: I/O (11/6)

I/O Devices

Processor for I/O

- Input: Read a sequence of bytes
- Output: Write a sequence of bytes

Memory mapped I/O

I/O Polling

Polling: Processor Checks status then acts

- Device registers
  - Control register says OK to read/write
  - Data Register, contains data

Polling

- Keep looping until control is 1
- % processor time to poll hard drive can be 40%

I/O Interrupts

Alternatives to Pollings: Interrupts

- Like a doorbell, occurs when I/O is ready or needs attention
  - Interrupt current program
  - Transfer control to the trap handler in the OS

Programmed I/O

DMA

Direct Memory Access (DMA)

- Allows I/O devices to directly read/write main memory
- Contains registers written by cpu Memory address to place data, # bytes

Incoming Data

1. Receive interrupt from device
2. CPU takes interrupt, initiates transfer
3. Device/DMA engine handle transfer
4. Upon completion interrupt CPU again

Outgoing Data

1. CPU decides to initiate transfer

DMA: Some New problems

- Where in memory hierarchy do you plug in DMA
  - Between l1 Cache and CPU
  - Between last level cache and main memory

Networking

The internet (1962)

- Wanted to connect computers
- 1973 Kahn and Cerf invented TCP
- Www (1989)

Software Protocol to Send and Receive

- SW Send steps
- 1. Application copies data to OS buffer
- OS calculates checksum starts time
- OS sends data to network interface HW and says start
- SW Receive steps
- 3. OF copies from HW to OS buffer

## Week 12: Lecture 32: Flynn Taxonomy, SIMD (11/6)

Parallelism

- Important to make processes faster without taking more power

Matrix Multiplication

- Matrix multiplication in Python

```
def dgemm(N, a, b, c):
    for i in range(N):
        for j in range(N):
            c[i+j*N] = 0
            for k in range(N):
                c[i+j*N] += a[i+k*N] * b[k+j*N]
```

| N | Python [MFLOPs] |
|---|---|
| 32 | 5.4 |
| 160 | 5.5 |
| 480 | 5.4 |
| 960 | 5.3 |

- 1 MFLOP = 1 Million floating-point operations per second (fadd, fmul)
- dgemm(N ...) takes $2*N^3$ FLOPs

- C is much faster

Flynn's Taxonomy

| | | Data Streams: Single | Data Streams: Multiple |
|---|---|---|---|
| Instruction Streams | Single | SISD: Intel Pentium 4 | SIMD: SSE instructions of x86 |
| | Multiple | MISD: No examples today | MIMD: Intel Xeon e5345 (Clovertown) |

Data streams and Instruction Streams

SIMD and MIMD most often

Single Instruction / single data stream

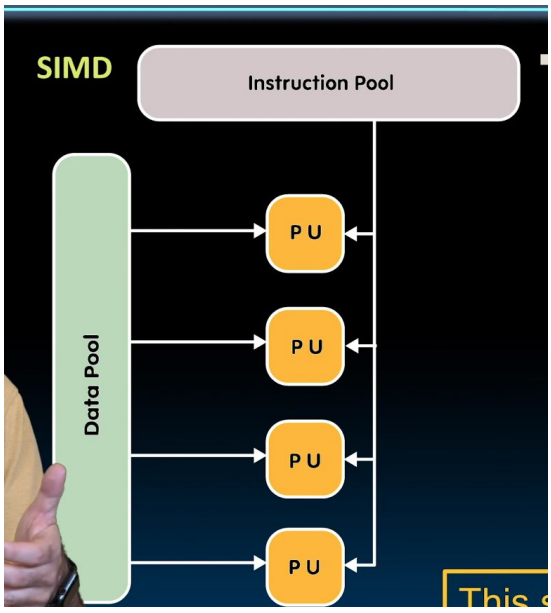
- Traditional

Single instruction / Multiple Data Stream (SIMD)

- Single stream to data for parallelized

Multiple INstruction / Multiple Data Stream (MIMD)

- Typically multiple processors of SIMD




SIMD Architecture

SIMD Architectures

- Data Level Parallelism
  - Operation on multiple data streams
- Multiplications independent,
- Pipelining and concurrency in memory access

Intel x86 SIMD Evolution

- 512b SIMD
  - New instructions every few years, new and wider register, more parallelism
- 128 bit packed

SIMD Registers in AVX512

- Registers are getting bigger and instructions bigger

SIMD Array Processing

Example SIMD Array Processing

- We pull 4 elements in the for loops and do operations
- Move all bits to be memory aligned, packed, single precision

Intel SSE Intrinsics

- Instrisics Can put C functions in assembly language
- Otherwise compiler can handle

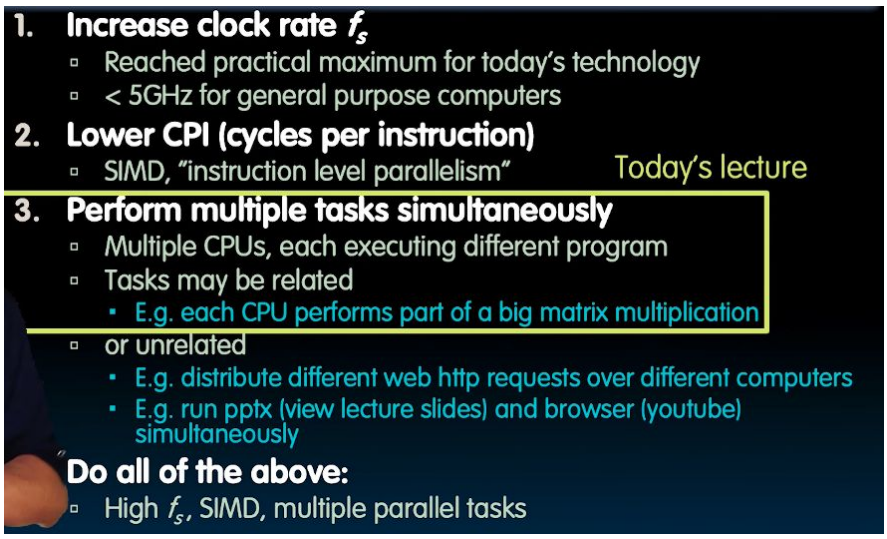
Matrix Multiply Example

Example: 2x2 Matrix Multiply

- _mm_load_pd(x1, _mm_mul_pd(a, b1));
-

## Week 12: Lecture 33: Thread-Level Parallelism I (11/13)

Parallel Computer Architectures




What happens with two cores?

- Can run separate programs at the same time
- Separate pc register alu

Multiprocessor Execution Model

- Separate resources, Highest level caches
- Shared DRAM (3rd level cache)
- Multiprocessor Microprocessor - two or more processors

Multicore

Transition to Multicore

- Switched to multicore and was able to improve app performance

Multiprocessor Execution Model

- Shared memory
    - Core has access to the entire memory in the processor
- Drawbacks
    - Slow memory shared by many customers
    - May become bottleneck
- Job-level parallelism
    - Processors work on unrelated problems
- Partition work of single task between several cores
    - Each part performs matrix operations

Parallel processing

- Only path to increase performance and lower energy consumption

Threads

Threads

- A thread of execution, single stream of instructions
    - Can split, fork itself into separate threads which can execute simultaneously
- A single core can execute many threads by Time Sharing
- Sequential flow of instructions that performs some task
- Each thread has
    - Dedicated PC, separate registers, accesses the shared memory
- Each physical core provides one or more
    - Hardware thread that actively execute instructions
    - Each executes one "hardware thread"

Operating System Threads

Give illusion of many "simultaneous" active threads

1. Multiplex software threads onto hardware threads
    a. Switch out blocked threads
    b. Timer
2. Remove a software thread from hardware thread by
    a. Interrupt its execution
    b. Saving its registers and PC to memory
3. Start Executing a different software thread by
    a. Loading its previously saved registers into a hardware threads registers
    b. Jumping to its saved PC

Multithreading

Hardware Assisted Software Multithreading

- Two copies of PC and Registers inside processor hardware
- Looks identical to two processors to software
- Hyper-Threading
    - Both Threads can be active simultaneously on one core

Hyper-Threading
- Simultaneous Multithreading: Logical CPUs > Physical CPUs
- Run Multiple threads

Multithreading
- Logical threads
  - 1% more hardware
  - 10% better performance
- Multicore
  - => Duplicate processors
  - 50% more hardware
  - 2x better performance?
- Modern machines do both 8 logical cpu, hardware threads

Review
- THread Level Parallelism
  - Multicore
    - Physical CPU: One thread at a time per CPU
    - Logical CPU: Fine grain thread switching
    - Hyper-Threading

## Week 13: Lecture 34: Thread-Level Parallelism II (11/16)

Parallel Programming Languages

Go is very good for parallel programming

Why so many languages
- SIMD features are continually added to compilers
- Specialized languages for different tasks
- OpenMP

OpenMP

Parallel Loops
- Parallel Execution: 4 loops 0-24, 25-49, 50-74, 75-99

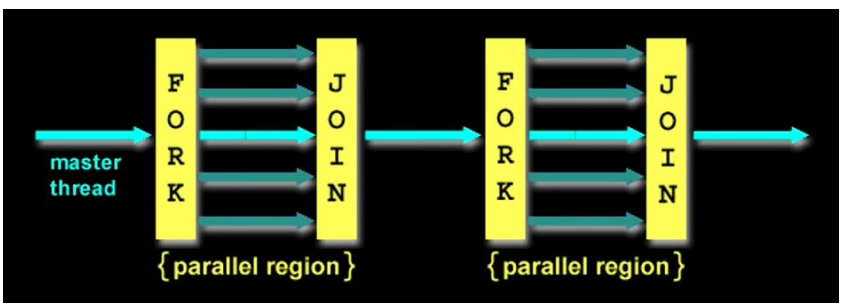
```
#include <omp.h>
#pragma omp parallel for
for (int i =0....) {}

$ gcc-5 -fopenmp for.c; ./a.out
```

Make code that is resilient to how threads are loaded and finished

OpenMP
- C extension: no new language to learn
- Multi-threaded,m shared-memory parallelism
- #pragma

OpenMP Programming Model
- Fork - join model



```c
/* clang -Xpreprocessor -fopenmp -lomp -o for for.c */

#include <stdio.h>
#include <omp.h>
int main()
{
    omp_set_num_threads(4);
    int a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int N = sizeof(a)/sizeof(int);

    #pragma omp parallel for
    for (int i=0; i<N; i++) {
        printf("thread %d, i = %2d\n",
            omp_get_thread_num(), i);
        a[i] = a[i] + 10 * omp_get_thread_num();
    }

    for (int i=0; i<N; i++) printf("%02d ", a[i]);
    printf("\n");
}
```

```
$ gcc-5 -fopenmp for.c;./a.out

% gcc -Xpreprocessor -fopenmp -
lomp -o for for.c; ./for
thread 0, i =  0
thread 1, i =  3
thread 2, i =  6
thread 3, i =  8
thread 0, i =  1
thread 1, i =  4
thread 2, i =  7
thread 3, i =  9
thread 0, i =  2
thread 1, i =  5
00 01 02 13 14 15 26 27 38 39
```

What Kind of Threads

- OpenMP threads are operating system threads
- Multiplex requested OpenMP threads
- Be careful when doing timing results

Computing Pi

Example 2: Computing pi

- Integrate quarter of circle
- Iterate on integer value
- Shared value is an issue if always add up to sum

```c
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
            printf("i =%3d,  id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
```

```
i =  1,  id =  1
i =  0,  id =  0
i =  2,  id =  2
i =  3,  id =  3
i =  5,  id =  1
i =  4,  id =  0
i =  6,  id =  2
i =  7,  id =  3
i =  9,  id =  1
i =  8,  id =  0
pi = 3.1424259850
```

Race condition

- Result is non deterministic : Garbage values

Synchronization

Problem:

- Limit access to shared resource to 1 actor at a time

Solution

- Take turns, only one person gets microphone and talks at a time

Locks

- Computers use locks to control access to shared resources
- Implemented with a variable
- Two threads could be intertwined and overwrite

## Week 13: Lecture 35: Thread-Level Parallelism III (11/18)

Hardware Synchronization

OpenMP Building Block: for loop

- Breaks for loop into chunks and allocate each to a separate thread

Data Races and Synchronization

- Two memory accesses form a data race if from different threads access same location

Hardware Synchronization

- Solution: Atomic read/write
- Read & write in single instruction, not other access
- Atomic swap of register and memory

RISC-V Atomic Memory Operations (AMOs)

- amoadd.w rd, rs2, (rs1)
- Add, and update new value in one instruction

RISC-V Critical Section

- If fall through it, I was the only one who can change it

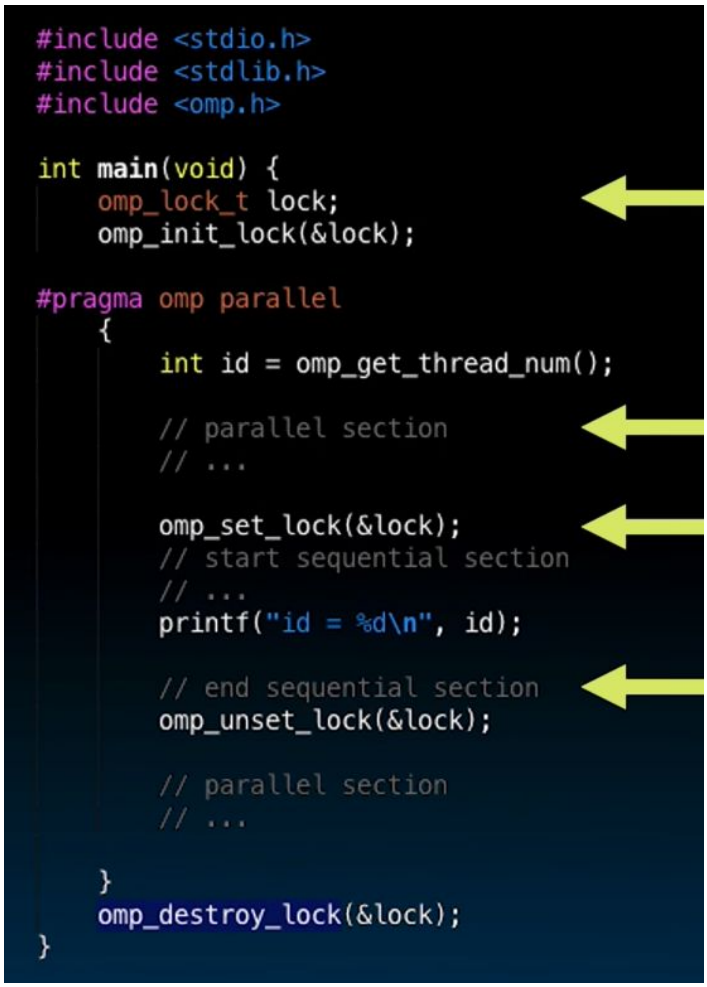
OpenMP

- Declare lock

When you above the lowest level, there are typically libraries

OpenMP Critical Section

- Only one thread at a time can enter a critical region
- Threads wait their turn
- #pragma omp critical



```
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
#pragma omp critical
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

Deadlock

- A system state in which no progress is possible
- Each waiting for each other

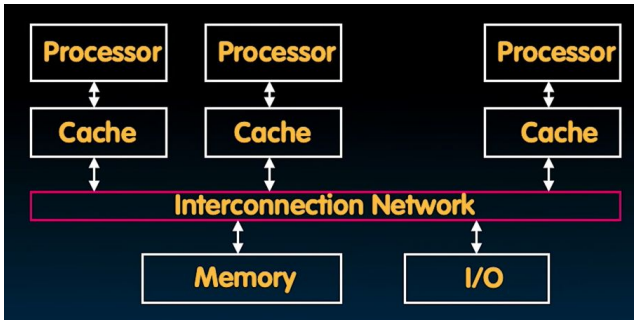
OpenMP Timing

- Double omp_get_wtime(void);
- Time is measured per thread, can compare to one from one point

<u>Shared Memory and Caches</u>

Multicore Multiprocessor

- SMP: (Shared memory)
- Single address space shared by all processors/cores
- Communicate through shared variables in memory with locks
- All multicore computers today are SMP




Multiprocessor Caches

- Use Caches to reduce bandwidth demands on main memory

<u>Cache Coherency</u>

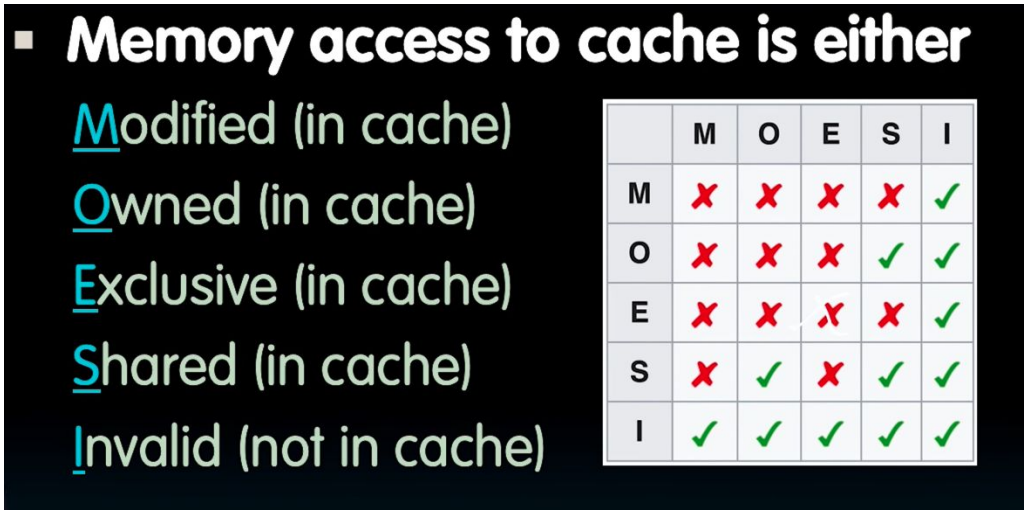
Keeping Multiple Caches Coherent

- Keep Cache values coherent
- Idea: When any processor has cache miss or writes, notify other processors via interconnection network
- Other caches snoop the common interconnect

Each cache tracks state of each block in cache

1. Shared: up-to-date data, other caches may have a copy
2. Modified: up-to-date data, changed, no other cache has a copy, OK to write, memory out of date
3. Exclusive: up to date data, no other cache has a copy, OK to write, memory up to date

    a. Avoids writing to memory if block replaced
4. Owner: up-to-date data, other caches may have a copy (shared state)
    a. Has only right to make changes, pass from owner to other
    b. Owned cache lines must respond to a snoop request with data




Shared Memory and Caches
- Invalidate other caches and update memory

Cache Coherency Tracked by Block
- Black ping pongs between two caches even though processors are different

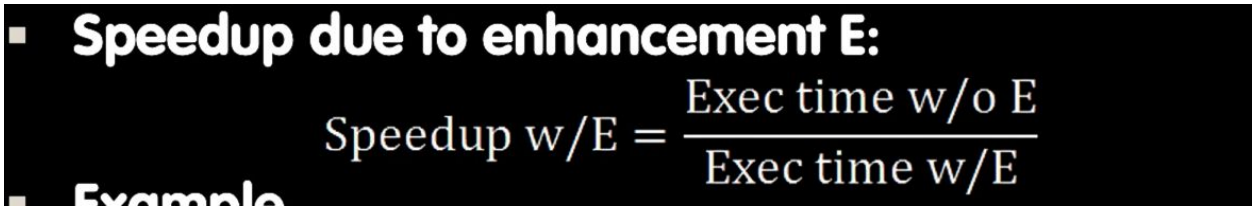
Misses
- Compulsory: Solution increase block size
- Capacity: increase cache size
- Conflict: increase cache size, increase associativity, better replacement policy
- Coherence Misses:
    - Misses cause by coherence traffic with other processors
    - Data moving between processors working together

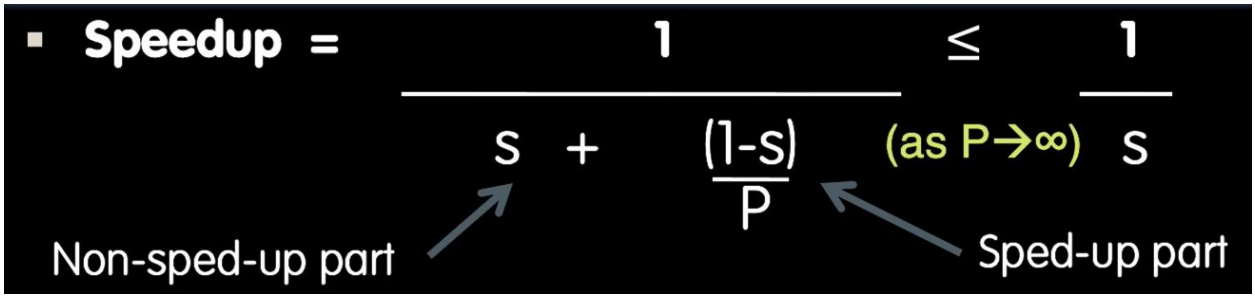
Conclusion
- OpenMP as a simple parallel extension in C
- TLP: Cache coherency implements shared memory even with multiple copies in multiple caches

## Week 13: Lecture 36: MapReduce, Spark (11/20)

Amdahl's Law

Amdahl's (Heartbreaking) Law

- 

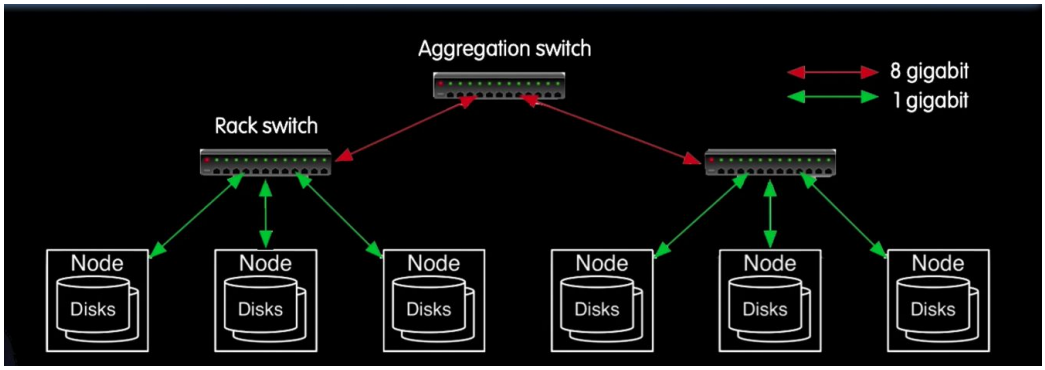
- 

- Where P is the factor that is it sped up
- Ex) Execution time of ⅝ program accelerated by factor of 16,
  - 1/(0.2 + 0.8/16) = 4, 16 times factor but only 4 times improvement
- Amount of speedup is limited by the serial portion

Request-Level Parallelism (RLP)
- Hundreds of thousands of requests/sec
- Computation easily partitioned within a request and across different requests

Data-Level Parallelism (DLP)
- Lost of data in memory, lots of data on many disks
- DLP actress many servers and disks using MapReduce

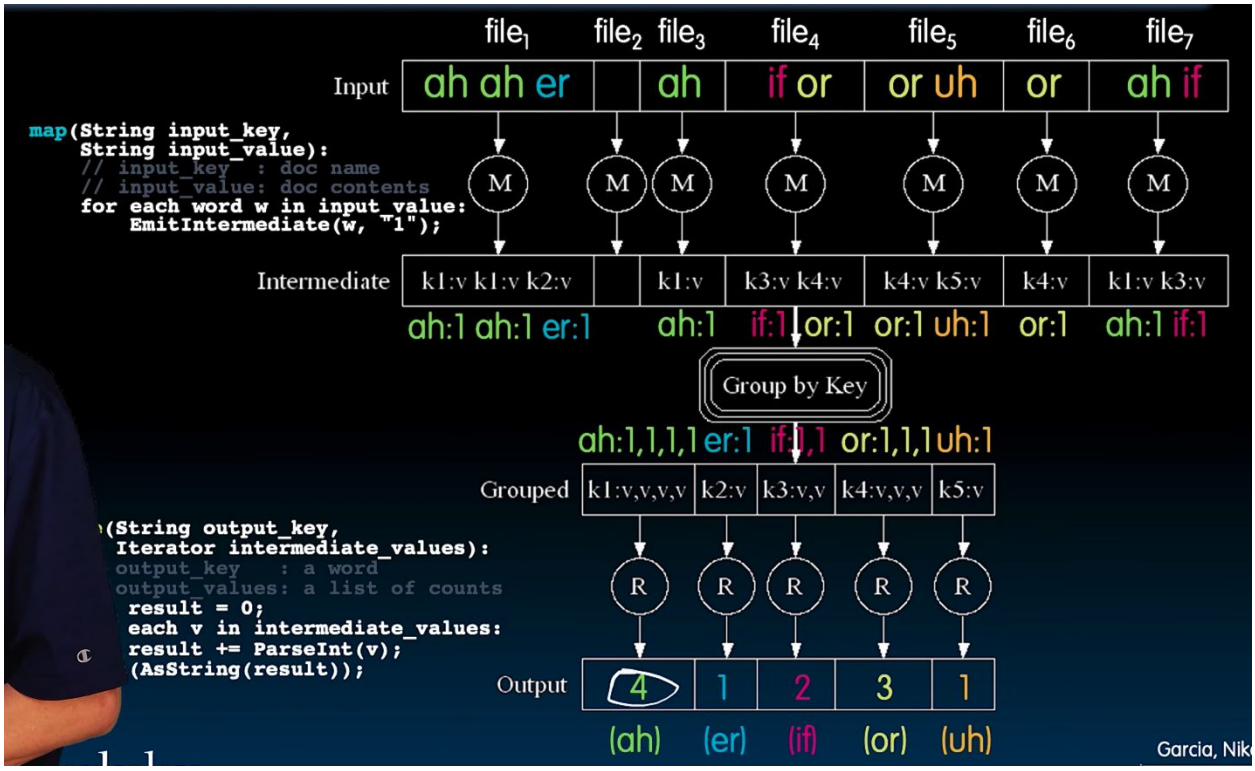
<u>MapReduce</u>
- Simple data-parallel programming model for scalability and fault-tolerance
- Scalability to large data volumes

Typical Hadoop Cluster
- 

- 40 nodes/rack, 1000-4000 nodes in cluster
- More expensive switches at higher end

MapReduce Programming Model
- Input & Output: each a set of key/value pairs
- Programmer specifies two functions
- map(in_key, in_value) -> list(interm_key, interm_value)
  - Processes input key/value pair
  - Slices data into shards distributed to workers
  - Produces set of intermediate pairs
- reduce(iterm_key, list(interm_value) -> list (out_value)
  - Combines all intermediate values for a particular key
  - Produces a set of merged output values

- Word count

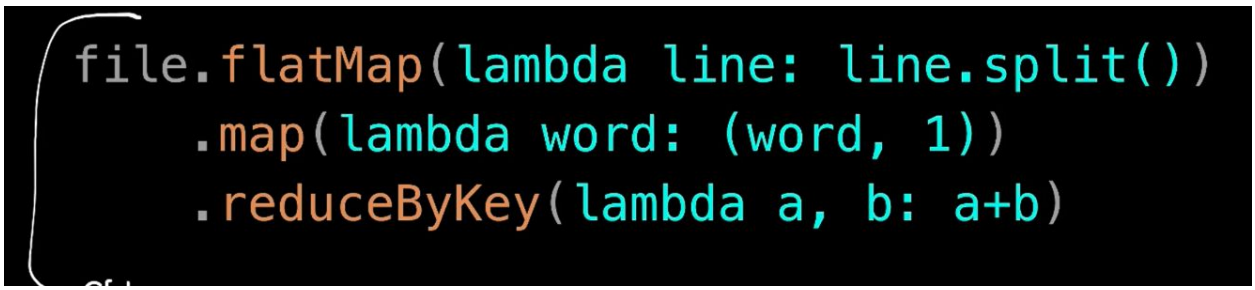



- Master assigns map and reduce tasks to worker servers
- To tolerate faults, reassign task if a worker server "dies"

<u>Spark</u>

- Apache Spark is a fast and general engine for large scale data processing
- Run programs up to 100x faster than Hadoop
- Advanced DAG execution engine that support cyclic data flow and in memory computing

Python API example

- 


Conclusion

- 4th big idea is parallelism
- Amdahl's Law constrains performance wins

## <u>Week 14: Lecture 37: Data Center, Cloud Computing (11/23)</u>

<u>Eras of computing</u>

PostPC Era: Personal Mobile Devices (PMD)

- Cloud Computing serving data

Warehouse Scale Computing

Economy of Scale: 5-7x cheaper by using commodity PCs 10k

Warehouse Scale Computers

- Massive scale datacenters
- Emphasize cost-efficiency
- Data-Level Parallelism
- Relatively small number of these make design cost expensive
- Operational Costs count

Equipment Inside a WSC

- Server 1x 19x8 computer 8 cores
- 7 ft rack of 40-80 servers with switch in middle
- Array (cluster) 16-32 server racks, larger cluster switch to manage array

Defining Performance

- Response time or latency
    - Time between start and completion
- Throughput or Bandwidth
    - Total amount of work in a given time
- Can write to others DRAM or Disk

Power Usage Effectiveness

Coping with Workload Variation

- Peak hours
- Cope with failures gracefully, scale up and down due to varying demand

Power vs Server Utilization

- % peak load = %peak energy

Power Usage Effectiveness

- Power Usage Effectiveness (PUE)
    - Total building power / IT equipment power

High PUE

- 33% + 9% to cool computers
- 30% for servers + networking

Localize to smaller container

## Week 15: Lecture 38: Dependability, Parity, ECC, RAID (11/30)

Intro

May fail transiently or permanently

Dependability via Redundancy

- Redundancy so that a failing piece doesn't make the whole system fail
- Datacenters, disks, memory bits all have Error Correcting COdes

Dependability Metrics

Dependability

- Fault: failure of a component

Time vs Space

- Spatial Redundancy : replicated data or check information or hardware to handle hard and soft failures
- Temporal redundancy: redundancy in time to handle soft failures

Dependability Measures

- Reliability: Mean Time to Failure (MTTF)
- Service interruption: Mean Time to Repair (MTTR)
  Mean tiem between failures (MTBF)
    - MTBF = MTTF + MTTR
- Availability = MTTF /(MTTF + MTTR)
- Improving Availability
    - Increase MTTF
    - Decrease MTTR

Availability Measures

- Rarely down number of 0s of availability per year

Reliability Measures

- Annualized Failure Rate (AFR): average number of failures per year

Failures In Time (FIT) Rate

- Rate of a device number of failure that can be expected in a one billion device hours of operations
- Automotive safety integrity level for components in vehicles

Dependability Design Principle

- No single points of failure
- Dependability corollary of Amdahl's Law

<u>Error Detection</u>

Error Detection/Correction Codes

- Memory systems generate errors
    - Soft eros when cells stuck by alpha particles
    - "Hard" errors can occur when chips permanently fail
- Memories protected against soft errors with EDC/ECC

Block Code Principles

- Hamming distance = difference in # of bits

Parity: Simple Error-Detection Coding

- Each data value is tagged with an extra bit for even parity
- When it is read, checked by finding parity bit

Error Detection and Correction

- Choose bits that has shortest hamming distance

<u>ECC Example</u>

Hamming ECC

- Leave bits for parity bits
- When decoding can check each parity bit and find the bit that is wrong
- Solve with Cyclic Redundancey check
-

Redundancy with RAID

RAID: Redundant Arrays of (inexpeinsive) disks

- Data stored across multiple disks
- Files "striped across multiple disks
- Redundancy yields high data availability
    - Availability: Service still provided to user, even if some components failed

RAID 1: DIsk Mirroring/Shadowing

- Each disk fully duplicated onto its mirror
- Most expensive solution: 100% capacity overhead

RAID 2: Parity DIsk

- Save logical record striped physical records, can use another disk to store the parity bit

RAID 3: High IO Rate Interleaved Parity

## Week 15: Lecture 39: GPU Architecture (12/2)

CPU: minimize latency of limited threads using complex

- better performance by minimizing latency

GPU: Maximize throughput by scheduling many parallel threads

- Many cores, thread level parallelism, many registers, many execution units

CPU Maximize throughput, hide latency

- Use SIMD to thread level parallelism

Simplified Graphics Piplining

- Vertices (Vertex Processing) -> Primitives (Rasterize) -> Fragments (Fragment Processing) -> Pixels (Frame buffer)
- 3D World Graphics API of Metal, Vertex Processing from 3D to 2D + depth, clipping and assempby, then rasterize by scan conversion and fragment processing

Wrapping U

- GPU are all about maximizing throughput via parallelism
- Pixels is great candidates for parallel processing
- Easily expanded to general purpose compute

## Week 15: Lecture 40: Conclusion (12/4)


1. Important to make sure you don't have to know all the details
2. Still improving but it is declining
3. L1 cache speed with the size of disk
4. New parallelism to improve more
5. New things improve performance in other ways
6. How to do important things make sure things don't fail

Need a metric to measure