Binary Representation

Powers of 2: 1 bit is 1, rest are 0.

Ex: $32 = 2^5 = 10000_2$

Power of 2 minus 1. All 1s.

Ex: $31 = 2^5 - 1 = 1111_2$ $(0, 2^n - 1)$

Negative Numbers

Sign Magnitude: First bit is sign: $(-1)^{bit}$
Rest are magnitude
Range: $(-(2^{n-1}-1), 2^{n-1}-1)$

Ex: $11111_2 = -15$

Two's Complement: Nonnegative = Normal binary
Negative:
Range: $(-2^{n-1}, 2^{n-1}-1)$

$-x \rightarrow$ write x in n bit binary, invert bits, add 1.

Ex: -15 in 8 bit $\rightarrow$ $15 = 00001111_2$
invert = 11110000_2, add 1: 11110001_2

Bias: Calculate unsigned binary, add bias

Ex: 011_2, bias = 3 represents 6 b/c
$011_2 = 3 + \text{bias} = 3+3 = 6$ Range: $(B, 2^n - 1 + B)$

Hexadecimal

Dec	Hex	Bin
00	0	0000
01	1	0001
02	2	0010
03	3	0011
04	4	0100
05	5	0101
06	6	0110
07	7	0111
08	8	1000
09	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Unsigned:
Smallest $\rightarrow$ All 0s
Largest $\rightarrow$ All 1s

2's Complement:
Smallest $\rightarrow$ 1, followed by 0's
Largest $\rightarrow$ 0, followed by 1's

Bias
Smallest: All 0's
Largest: All 1's

Sign magnitude and IEEE-754 standard double precision float have multiple representations of 0.

Floating Point

Range = $[-2^{m+1}, 2^{m+1}]$
m = # of mantissa bits

Single Precision: 32 bits, bias = -127
1 sign bit, 8 exponent bits, 23 mantissa
Double Precision: 64 bits, bias = -1023 bits
1 sign bit, 11 exponent bits, 52 mantissa bits

For normalized floats:

$$\text{Value} = (-1)^{\text{Sign}} * 2^{\text{Exp} + \text{bias}} * 1.\text{significand}_2$$

For denormalized floats:

$$\text{Value} = (-1)^{\text{sign}} * 2^{\text{Exp} + \text{bias} + 1} * 0.\text{significand}_2$$

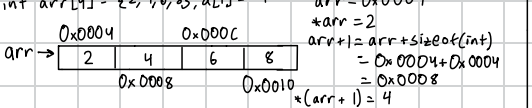
Exponent	Significand	Meaning
00...00	Anything	Denorm
00...01 $\rightarrow$ 11...10	Anything	Normal
11...11	0	$\pm \infty$
11...11	Nonzero	NaN

Variables and Pointers in C

int x; $\rightarrow$ variable x holds 4 bytes, represents an int.
int* z; $\rightarrow$ variable z holds 4 bytes, represents a pointer or memory location or address
int a = *z $\rightarrow$ z holds a location, xz gets whats in the location and stores it in variable a.
int *y = &z $\rightarrow$ y stores the location/memory address of z.

Arrays and Pointer Math

Arrays are pointers to the first element.
int arr[4] = {2, 4, 6, 8}, a[1] == 4



Endianness:

x[0] = 0x DEADBEEF
x[1] = 0x C561C156

Little Endian: a = EF, a+1 = BE, a+2 = AD, a+3 = DE
a+4 = 56, ...

Big Endian: a = DE, a+1 = AD, a+2 = BE, a+3 = EF
a+4 = C5

Strings

- Pointer to 1 or more chars
- The end of a string is the null terminator "\0".
- When malloc, allocate 1 extra byte for b/c sizeof("\0") = 1 byte.
- sizeof(char) = 1
- strlen(String) returns length of a string not including "\0"
- strcpy(destination, src) copies the string that src points to into destination including the null terminator.
- strcmp(s1, s2) compares two strings and returns 0 if identical, neg value if s1<s2, pos value if s2>s1.

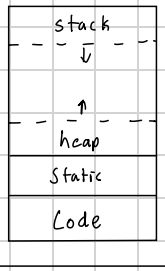
Memory Structure

Stack: -Dynamic memory
- local variables inside functions
- grows downward
- new stack frame when a function is called, removed when function ends
- stack pointer points to top of stack
- last in, first out
- Function calls

Heap: -Dynamic memory
- malloc, realloc, free calls
- grows upward

Static or Data: -global variables
- string literals
- static local variables

Text or Code: -Machine instructions
- RISC V
- Functions themselves



Structs

- Similar to classes

```
struct ex {
    int a;
    char b;
};

struct ex myStruct;
myStruct.a = 21;
myStruct.b = 'z';
struct ex *myPtr = &myStruct
```

myPtr → myStruct: struct ex, a = 21, b = 'z'

C is pass by value, so if you want to change an aspect of a struct instance, pass a pointer to the struct instance.

```
void funcA(Struct *ptr) {
  (*ptr).a = 4  or  ptr->a = 4
  return;
}
```

so if you call funcA(myPtr), then myStruct.a = 4.

Memory Allocation

- malloc(size) allocates a block of size bytes and returns the start of the block(address)
- calloc(count, size) allocates a block of count * size bytes and sets every value to 0. Returns the start of the block(address).
- realloc(*ptr, size) resizes the block at ptr to size bytes and returns the start of the resized block (address).
- free(*ptr) deallocates a block of memory which starts at ptr that was previously allocated by malloc, calloc, or rcalloc.

C Generics

- int* pointers only point to ints
- char* pointers only point to chars/strings
- void* pointers can point to any data type.
  - cannot dereference b/c you don't know how many bytes to read.
  - Instead: memcpy(void* dest, void* src, num_bytes) and dest cannot overlap with src
    OR
    memmove(void* dest, void* src, num_bytes) and dest can overlap with src

RISC V

- Assembly language
- Programs are translated to assembly via the compiler
- 32 registers and program counter
- x0: always stores 0
- t-registers: temp registers
- a-registers: arg registers
- s-registers: save registers
- sp: stores stack pointer
- ra: stores caller's return address

RISC V Functions

Caller = parent calling another function
Callee = function being called

1. Caller puts parameters in registers (a0-a7) for callee to access
2. Transfer control to callee using jump (and link) instruction
3. In callee, allocate space needed for variables in the function
   - Make room for variables on stack/heap
4. Perform desired task of the callee
5. Put result value in a0 register where caller can access it.
6. Return control to caller (line after function call)
   - ret
   - OR jr ra
   - OR jalr x0 ra 0

Calling Convention

- Caller saved registers (ra, t0-t6, a0-a7)
  - From caller's perspective, contents are not preserved after invoking a function.
  - Callee can do whatever it wants to these registers.
  - ra needs to be restored if changed
- Callee saved registers (sp, s0-s11)
  - From caller's perspective, contents are preserved after invoking a function.
  - Callee must restore registers to their original values before returning to the caller function.

Instruction Format

- opcode: specifies instruction type
- funct 3/7: some instructions have it to fully specify operation
- rs1, rs2: source/input registers to an instruction
- imm: constant value used in some instructions

Convert Binary/Hex to Instruction

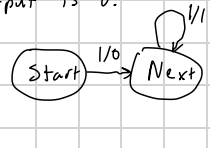
1. Convert to binary
2. Check opcode to identify instruction type.
3. Split bits according to instruction type.
4. Get instruction/rs1/rs2/rd/imm based on corresponding bits

Convert Instruction to Binary/Hex

1. Check instruction opcode, func3/7 (if any), and identify instruction type.
2. Convert registers to their corresponding 5 bits.
3. Split immediate up to corresponding bits in the instruction type.
4. Check GIC Ref Card and plug bits into appropriate sections.

FSM (Finite State Machine)

- Simplified version of computer
- Takes in sequence of characters and outputs a sequence of characters.
- Designated start state.
- Transitions are labeled X/Y, where X is input, and Y is the output.
- So if input is 1 and transition is 1/0, output is 0.



Single Cycle Datapath

Components

State elements:
- PC Register
- RegFile
- IMEM
- DMEM

Combinational Logic:
- ALU
- Branch Comp
- Immgen
- MUXes

Datapath Stages

1. Instruction Fetch (IF):
   - Send address (PC) to IMEM, and read IMEM at that address
2. Instruction Decode (ID):
   - Generate control signals from the instruction bits, generate the immediate, and read registers from the RegFile
3. Execute (EX)
   - Perform ALU operations, and do branch comparison.
4. Memory (MEM)
   - Read from or write to the data memory (DMEM).
5. Writeback (WB)
   - Write back either PC+4, the result of the ALU operation, or data from memory to the RegFile

Signal	Values	Meanings
PCSel	0,1	0: PC = PC + 4; 1: PC = ALU Result
RegWEn	0,1	0: Disable register writing; 1: Enable register writing
ImmSel	0-7	0-4: I,B,S,J,U Type instruction; 5-7: Unused
BrEq	0,1	0: Inputs are equal; 1: Inputs are not equal
BrLt	0,1	0: rs1 >= rs2; 1: otherwise
ALUSel	0-15	Different arithmetic operations (you don't have to memorize)
MemRW	0,1	0: Non s instructions 1: sw, sh, sb
WBSel	0-3	0: DMEM read; 1: ALU output, 2: PC+4, 3: Unused

CALL

Compiler: Responsible for register allocation. Performs syntax analysis, code generation, and optimization. Takes in C and outputs assembly.

Assembler: Converts assembly to machine code. Takes in assembly and outputs object file (.o) w/ text and data segments. Generates relocation table.

Linker: Converts object files into executables. Takes in object files and outputs executable machine code (.out)

Loader: Executes the program and loads it to memory. Takes in executable machine code, loads it to memory and then runs it.

For Floating Point, the step (gap between 2 consecutive representable numbers) is 2^{k-m} where k is the largest power of 2 below the number and m is the number of mantissa bits.

For 1 sign bit, 7 exp bits, 8 mantissa bits, the largest pos finite non int is 255.5 b/c at $256=2^8$, step = 1. At 2^7, step = $2^{-1}=0.5$ so 256-0.5 = 255.5

```
foo:
  lui a0, 0xFFEE0
  addi t0, a0, 4
  jal ra bar
  lw s0, 0(t0)
  jr ra
```

t0 breaks calling convention b/c bar might have changed t0 before its used.

ra breaks calling convention b/c ra is changed and so the return address has changed.

s0 breaks calling convention because it is written to before saving its contents.

GDB

Command	Abbreviation	Description
start	start	begin running program and stop at line 1
step	s	execute current line (steps into functions)
next	n	execute current line (doesn't step into functions)
finish	fin	executes the remainder of current function and returns to calling function
print arg	p arg	prints value of arg
quit	q	exits gdb.
break arg	b arg	sets breakpoint at specified location
cond break	b arg if cond	sets breakpoint at arg if cond is met.
run	r	restarts execution of program, stops at bp or termination.
continue	c	continues execution of a program that was paused, stops at bp or termination.
backtrace	bt	print one line per frame for frames in stack.
print cond	p cond	print 1 if cond is true, else 0.

```
target: addi t1 x0 -10
beq t1 t2 target
```

For this the address of target is 4 bytes before the beq instruction so convert -4 to binary in 2's complement and use that as the immediate.

Memory (32 bit system)

Pointers → 4 bytes size_t → 4 bytes

uintx_t → $\frac{x}{8}$ bytes

char → 1 byte

Use calloc for empty strings, ints at 0, arrays at 0, etc.

Bitwise Operations

- Right shift to isolate digits
- Left shift to move digit to correct place
- OR completed components together
- XOR with 1s for flipping bits
- AND with 1s at relevant bits, 0s at others
  - abcd1111 → extract cd → shift right 4 and then AND with 000011.
- Left shifting all bits by n of IEEE-754 representation denorm number while preserving sign bit doubles number.

Common RISC V Instructions

mv rd rs1 → copy val at rs1 to rd

li rd imm → rd = val

beq rs1 rs2 label → if rs1==rs2 go to label

lw rd offset(rs1) → load val at src + offset (count bytes) to rd

jal label → jump to label, automatically has access to arg regs

add rd rs1 rs2 → add vals at rs1, rs2 and store in rd

addi rd rs1 imm → add val at rs1 and imm, store in rd

jr ra → return to return address

j label → jump to label, use if don't want to come back

jal label → jump to label, change ra to next line, use if you want to come back.

slti/sltiu rd rs1 imm → if val at rs1 < imm, rd val = 1 else 0

slt/sltu rd rs1 rs2 → if rs1[val] < rs2[val]: rd[val] = 1 else rd[val] = 0

sub rd rs1 rs2 → rd[val] = rs1[val] - rs2[val]

jalr → use if function address stored in register (indirect label)

bne, blt, bltu, bge, bgeu, auipc, lui

signed makes very large numbers be neg, could cause errors.

Hex digit is 4 bits, so if shifting n, move $\frac{n}{4}$ places

Shifts

Logical shifts pad with 0s

Arithmetic right → pad with sign bit over and over

Endianness

For an array in a little endian system, rewrite each component right to left then the whole thing right to left

0x0080, 0xA000, 0x0036, 0x0000

→ 0x80, 00, 00, A0,

→ 0x36, 00, 00, 00

arr1 = [0x00000036, 0xA0000080]

for a 32 bit system

For an array in a big endian system, rewrite each component left to right, then the whole thing left to right.

0x0080, 0xA000, 0x0036, 0x0000

→ 0x 0080A000

→ 0x 00360000

arr1 = [0x0080A000, 0x00360000]

for a 32 bit system

$2^0+2^1+2^2+\ldots+2^k = 2^{k+1}-1$

$2^{-1}+2^{-2}+2^{-3}+\ldots+2^{-k} = 1-2^{-k}$

Boolean Algebra

If a term cannot be true, it is 0.

Ex: $A\bar{A} = 0$

Logic Gates

NOT

AND

OR

NAND — AND → NOT

NOR — OR → NOT

XOR — 1 if inputs are same else 0 $(A\bar{B}+\bar{A}B)$

XNOR — XOR → NOT

SDS

$t_{clk\text{-}to\text{-}q} + t_{shortest\text{-}path} \geq t_{hold}$

$t_{clk\text{-}to\text{-}q} + t_{longest\text{-}path} + t_{setup} \leq t_{clk\text{-}period}$

$f = \frac{1}{T}$

$t_{shortest\text{-}path}$ and $t_{longest\text{-}path}$ is from register to register, you can't go through a register to another register.

If probing at the output of a register, mark all rising clock edges. At each rising clock edge, if input is stable, value of input is value of output but account for clk to q time. Stable means constant input through setup and hold times. With logic gates, draw both of its inputs, perform the logic with the waveforms and plot the resulting waveform.

Datapath

If not writing to memory, must read memory.

If writing to rd, RegWEn = write enabled

ImmGen accts for all known formats.

Read input for every rs. New wdata for every register being written to.

DMEM is used for loads and stores

Branch Comp used for instructions w/ branches

BrUn = 0 → signed

BrUn = 1 → unsigned

Pipelined Datapath

IF: $t_{clk\text{-}to\text{-}q} + t_{IMEM\ read} + t_{Reg\ setup}$

ID: $t_{clk\text{-}to\text{-}q} + t_{RF\ read} + t_{Reg\ setup}$

EX: $t_{clk\text{-}to\text{-}q} + t_{mux} + t_{ALU} + t_{Reg\ setup}$

MEM: $t_{clk\text{-}to\text{-}q} + t_{DMEM\ read} + t_{Reg\ setup}$

WB: $t_{clk\text{-}to\text{-}q} + t_{mux} + t_{RF\ setup}$

$t_{clk} \geq \max(IF, ID, EX, MEM, WB)$

$\text{Clock frequency} = \frac{1}{t_{clk}}$

$\text{Throughput} = \frac{\text{Clock Frequency}}{\text{Cycles per Instruction}}$

$\text{Speedup} = \frac{\text{Single cycle } t_{clk}}{\text{Pipelined } t_{clk}}$

Hazards

- Structural hazards → two pipeline stages need the same hardware resource.
- Data hazard → An instruction depends on the result of a previous instruction.
- Control hazard → Pipeline doesn't know instruction to fetch next because the outcome of a branch or jump hasn't been resolved.

Fixing Hazards

- Scheduling = reordering instruction
- Stalling = dependent instruction waits until resource is ready.
- Forwarding = don't wait for result to be written to register to use it, read directly from wire in the datapath.

Caches - Physical piece of hardware

Locality → Principles with which we update or save data.

1. Temporal: Keep recently accessed items in the cache
2. Spatial: Keep continuous blocks of memory in our cache, ie the blocks around what was recently accessed.

Small ←——→ Large

CPU → L1 → L2 → L3 → Memory

Fast ←——→ Slow

Direct Mapped Caches

Index: $\log_2(\#\text{ of blocks})$ bytes

Offset: $\log_2(\text{size of block})$ bytes

Tag: # bits in memory address − # index bits − # offset bits

$\#\text{ address bits} = \log_2(\text{memory space bytes})$

Size of cache = # of blocks * block size

Address = | Tag | Index | Offset |

Cache Misses

Compulsory miss - Haven't tried accessing this block from memory before, so it def is not in the cache.

Conflict miss - Data was evicted from cache previously, not a capacity or compulsory miss

Capacity miss - Cache is full.

Check compulsory → conflict → capacity

ie first time? → cache is full?

Associativity

- Direct mapped cache - each mem address is associated with 1 cache block
- Fully associative cache - index doesn't exist, any block can go anywhere
  - No conflict misses, compare all tags
- N way set associative cache - index points to a set, which contains multiple blocks.
  - avoids conflict misses, compare all tags in a set.
- Direct mapped = 1 way set associative
- fully associative = M way set associative where M = # of blocks.

Eviction

LRU = Least Recently Used

FIFO = First in first out

Random = Random

AMAT (1 cache) = HT + MP(MR)

AMAT (2 caches) = HTL1 + MRL1 * (HTL2 + MRL2(MP))

Memory Quantities

$K_i = 2^{10}$ B = bytes

$M_i = 2^{20}$ b = bits

$G_i = 2^{30}$ 1 byte = 8 bits

$T_i = 2^{40}$

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instruction}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycles}}$$

Intel Intrinsics

_mm_<intrin_op>_<suffix>

<intrin_op> = operation (add, sub, etc.)

<suffix> indicates data type. First 2 letters indicate packed (p), extended packed (ep) or scalar (s). Rest indicate type, e.g. single precision FP (s), double precision FP (d), signed 128 bit int (i128, i#), unsigned 64 bit int (u64, u#), etc...

__m<suffix> = return type

Processes vs. Threads

- Process is an execution of a program
  - Each has its own address space and resources
- Thread is a segment of a process
  - Different threads of the same process share components (e.g. heap, global variables)
  - Each thread has its own registers, PC, and stack.

```
#pragma omp parallel
{
  // something
}
//something goes to every thread
```

```
#pragma omp parallel for
for(i=0; i<=n; i++) {
  //something
}
```

Iterations of the for loop are split amongst threads

Public vs. Private Variables

Public: declared outside #pragma omp parallel

- All threads have RW access to the variable

Private: declared inside #pragma omp parallel

- Each thread has its own copy of this variable and has RW access to it.

Data Race

- A shared variable is accessed by multiple threads
- At least one access is a write
- A lock forces a thread to execute before moving on to the next (executes critical section, uses #pragma omp critical

WBSel chooses what to write to rd but RegWEn actually decides whether or not to write it.

Operating Systems

- Loads processes
- Manages virtual memory
- Manages a programs I/O
- Regulates memory use of processes
- Manages multiple processes and assigning them to cores

Pipelining Example

lw +1 0(s0) → writes +1 in WB stage

sw +1 4(s0) → reads +1 in ID stage

	N	N+1	N+2	N+3	N+4	N+5
lw	IF	ID	EX	MEM	WB	
sw		IF	ID	EX	MEM	WB

N+4 & N+2 so 2 stalls needed.

To create a multi stage pipeline from a single cycle, split up the longest path so you obtain the minimum for the largest stage time. Basically you are creating the smallest critical path

$\text{Offset bits} = \log_2(\text{page size in bytes})$

$\text{VPN bits} = \log_2\left(\frac{\text{Virtual mem size}}{\text{page size}} \text{ in bytes}\right)$

$\text{PPN bits} = \log_2\left(\frac{\text{Physical mem size}}{\text{page size}} \text{ in bytes}\right)$

$\#\text{ entries in page table} = \frac{\text{size of virtual mem}}{\text{page size}}$

$$\#\text{ physical pages to store page table} = \frac{\#\text{ entries in page table} * \text{size of entry}}{\text{page capacity}}$$

Virtual Memory

1. Label rows of page table starting from 0
2. Write first bit for each row
3. For each given virtual address, split into VPN and offset bits based on those given lengths
4. Look for VPN in TLB table. If there, check if valid is 1. If it is, use that PPN. Else, move on to Page Table.
5. Check the row corresponding to the VPN (look in row 5 for 0x0005). If the first bit is 1, use the PPN. Make sure to refer to given format to find the correct PPN. If the first bit is 0, use the PPN of the next available free page.
6. For each physical address write the PPN followed by offset. For example, if PPN is 0x42D and offset is 0x29, the physical address is 0x42D29.