CS 61C Quest Notes

Jeffrey Shen 30345555 77

Bits

- N bits is at most 2^n things
- Numerals: Binary (2), Decimal (10), Hexadecimal (16)
- ASCII: for all characters in English Language, 7 bits

Bit Conversions

Binary → Decimal, Hex → Decimal

- Add up powers of 2/16

$0b101$
$1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5$

Binary ↔ Hex

1. Pad left 0s to make multiple of 4
2. Read off groups of 4, using table
3. Drop any leading 0s

Decimal → Binary, Decimal → Hex

1. From left to right, find largest power of 2/16
2. If it fits, subtract and repeat w/ next digit

Decimal	Hex (0x...)	Binary (0b...)
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

Overflow: number is too large to be represented, positive or negative

Bit Operations

&	AND	x & y	extract bits, check neither are 1, turn off bits
|	OR	x | y	combine w/ mask, check either are 1, turn on bits
^	XOR (not equal)	x ^ y	flip bits w/ mask
~	Complement (flip)	~x	flip bits
<<	shift left	x << n	multiply by 2
>>	shift right	x >> n	divide by 2

1<<n is n 0s to right of 1

Number Representations

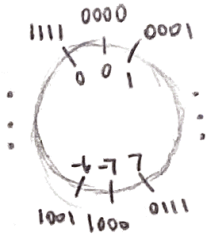
N bits

① Sign and Magnitude $[-(2^{N-1}-1), 2^{N-1}-1]$

Negation: Leftmost bit is sign bit, 0: positive 1: negative

- 2 zeros

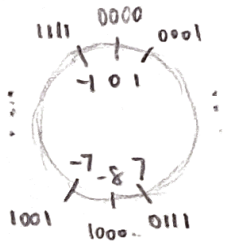
② One's Complement $[-(2^{N-1}-1), 2^{N-1}-1]$



Negation: Flip the bits

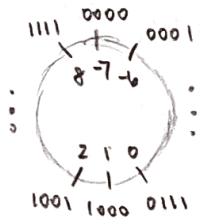
- 2 zeros, leftmost bit tells sign

③ Two's Complement $[-2^{N-1}, 2^{N-1}-1]$



Negation: Flip bits and add one

④ Bias Notation $[b, 2^N - 1 + b]$



b = -7

- Shift on unsigned notation

x + b = n

x: unsigned, b: bias, n: number represented

⑤ Unsigned $[0, 2^N - 1]$

- No negative numbers, max at $2^N - 1$

C

- Allows us to exploit underlying architecture, created in 1970s, C99
- Files first pass through C Pre-Processor where macros replace functions

Variable types

Ex) char: 8 bits (1 byte)
int: 32 bits (4 byte)
int *: 32 bits depending on machine [index into memory array]

False values: '\0', 0, NULL

Pointer Arithmetic: increments in sizeof(pointer_type)

Structs: structured groups of variables

sizeof: returns number of bytes

Pointers (type *var)

- Stores an address
- dereference operator (*): gets value at address
- C passes parameter by value, pass pointer
- If a variable is not initialized, it holds garbage
- Arrow Notation: var → x same as (*var).x

&var	= 0xA
var	= 0xB
*var	= 3

Memory Allocation

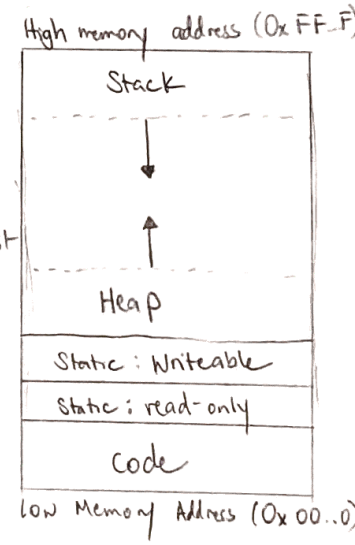
malloc(byte-size): returns void * pointer, initalizes with default garbage

free(ptr): must free any malloc'ed pont, once

realloc(ptr, size): reallocate memory

- After allocating memory, check if pointer is NULL

Memory Management



1. Stack: function local variables, strings allocated as arrays (LIFO)
2. Heap: dynamically allocated memory (malloc, calloc, realloc)
   Not necessarily contigous, fragmentation is an issue, circular Linked List
3. Static: global variables, statically allocated strings, basically permanent memory
4. Code: machine instructions

NOTE: - memory of pointers and what they point to may be in different memory

- char* s1 = "CS61C" s1[0] is in static, read-only
- char s2[] = "CS61C" s2[0] is in stack

Tips

- strlen(char*): # chars (bytes) in string not including terminator
- Big Endian: lowest address on left
- Little Endian: lowest address on right
- "\n" after print
- arr is pointer to first element in array

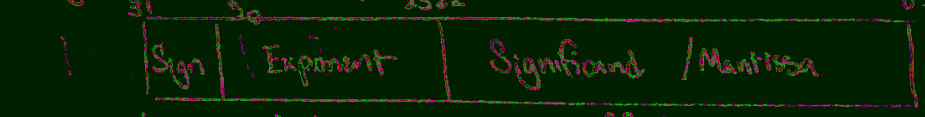
C Problems

- Draw out diagram w/ addresses
- Check if malloc or parameters are NULL

CS61C Midterm Notes — Jeffrey Shen

Floating Point Representations

Single Precision Representation (32 bit)



Exponent	Significand	Object
0	0	0
0	nonzero	Denorm
1-254	any	+/- floating pt #
255	0	+/- ∞
255	nonzero	NaN

$$Value = (-1)^{sign} \times 2^{exp + bias} \times (1 + Significand)$$

Denorm: $Value = (-1)^{sign} \times 2^{bias+1} \times (0.\ Significand)$

- Allows smallest value $\approx 2^{-149}$

Assembly

Instructions: [operation] [destination], [source1], [source2]

- RISC-V is little Endian - lowest byte on right w/ lowest address
- Program Counter: Internal Register holding byte address of next instruction
- Pseudo-instructions: shorthand syntax for common assembly idioms (mv, li, nop)

Function Calls

Steps

1. Put arguments in registers for function (a0-a7)
2. Transfer control to function (jal)
3. Acquire local storage resources (prologue)
4. Perform desired task of function
5. Put return value into register and release local storage
6. Return control to point of origin (ret)

Ex

Prologue:
```
addi sp, sp, -8
sw   s1, 4(sp)
sw   s0, 0(sp)
```

Epilogue:
```
lw   s0, 0(sp)
lw   s1, 4(sp)
addi sp, sp, 8
jr   ra
```

Calling Convention

In function (callee), at prologue, epilogue:

Save:
- sp
- S registers (s0-s11)

Before calling function (caller), save:
- ra
- t registers (t0-t6)
- a registers (a0-a7)

Instruction Formats

R: register register arithmetic

I: immediate arithmetic, loads

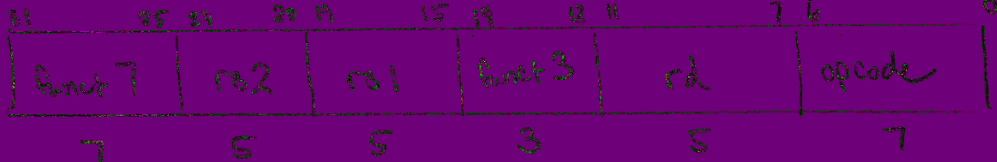
S: Stores

B: Branches

U: Upper Immediate

J: Jumps

R-Format (Register)



Operations: add, sub, sll, slt, sltu, xor, srl, sra, or, and

I-Format (Immediate)

Immediate values (-2048, 2047)

Operations:
- Immediates: addi, slti, sltu, xori, ori, andi, slli, srli, srai
- Loads: lb, lh, lw, lbu, lhu
- Jumps: jalr

S-Format (Stores)

SW src, offset(base)

Operations: sb, sh, sw

B-Format (Branches)

Implicit imm[0] = 0

Operations: beq, bne, blt, bge, bltu, bgeu

U-Format (Upper Immediate)

li will handle case of 32bit values

Operations: lui, auipc

J-Format (Jumps)

jal: rd = PC+4
PC = PC + offset

jalr: rd = PC+4
PC = rs1 + imm

Implicit 0

Operations: jal

Compiler, Assembler, Linker, Loader (CALL)

- Interpret a high level language when efficiency not critical to exec other programs
- Translate to lower level language to increase performance, hide source

Compiler

High level language → Assembly Language (C → RISC-V)

- Translates language

Assembler

Assembly language → object files (RISC-V → file.o)

- Replace pseudo-instructions
- Two passes to determine offset between jumps
- Reads and uses directives (.text, .data, .string, .globl)
- Produces relocation tables (to fill in later when you link) in other files) and symbol tables (list of items in this file that may be used/referenced)

Linker

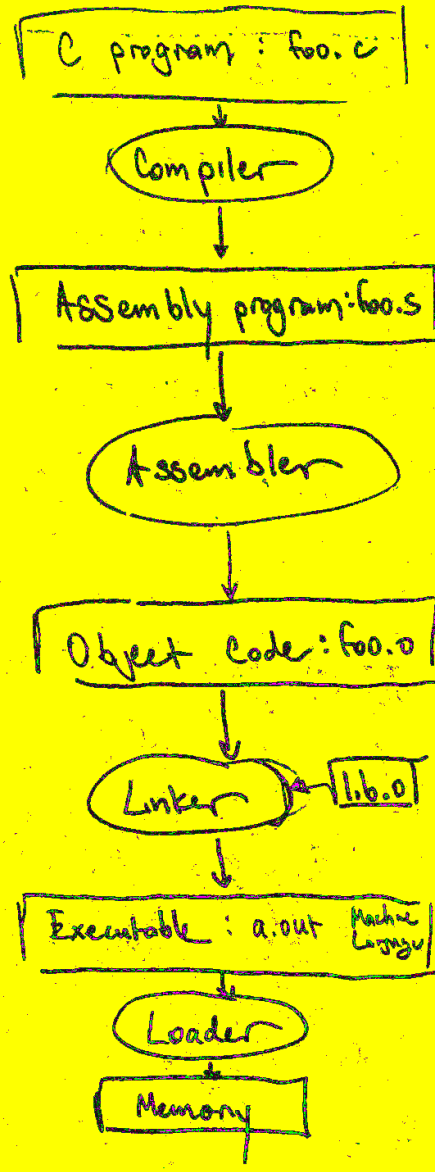
object files → executable code (file.o → a.out)

- Fulfills missing labels in relocation symbol tables
- Combines object files into binary executable

Loader

executable file → program run

- Reads header and creates memory space, set up for execution



Combinational Logic

NOT
flips input

AND
1: both inputs 1
0: otherwise

OR
1: at least one input 1
0: otherwise

XOR
1: inputs different
0: otherwise

MUX
pick among inputs
S input selects one of 2^S

Boolean Algebra

+ : OR
* : AND
¯ : NOT

$x + yz = (x+y)(x+z)$

$xy + x\bar{y} = x$

$\overline{x+y} = \bar{x}+\bar{y}$

$(x+y)x = x$

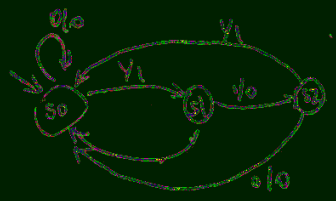
$\overline{(x+y)} = \bar{x}\cdot\bar{y}$

distributive
uniting theorem
DeMorgan's Law

Canonical Forms : sum of products

- $y = \bar{a}\bar{b}\bar{c} + \bar{a}b\bar{c} + a\bar{b}\bar{c} + abc$

Can find simplified using truth table



Finite State Machines : number of states, transitions

- State denoted as start state, one arrow for input
- X/Y where input X, then output Y following arrow

Sequential Digital Systems

Registers : Controlled by clock, storage object

Value updated at "rising edge of the clock" otherwise constant

- Setup time: time before rising edge where input must be stable
- Hold time: time after rising edge where input must be stable
- Clock-to-Q Delay: time it takes for register's input to become its output after rising edge

Critical Path : longest delay between state elements

$$t_{cc} \geq t_{clk\text{-}to\text{-}q} + t_{logic} + t_{setup}$$

t_{logic} = longest logical path

Hold time requirement: $t_{clk\text{-}to\text{-}q} + t_{logic,shortest} > t_{hold}$

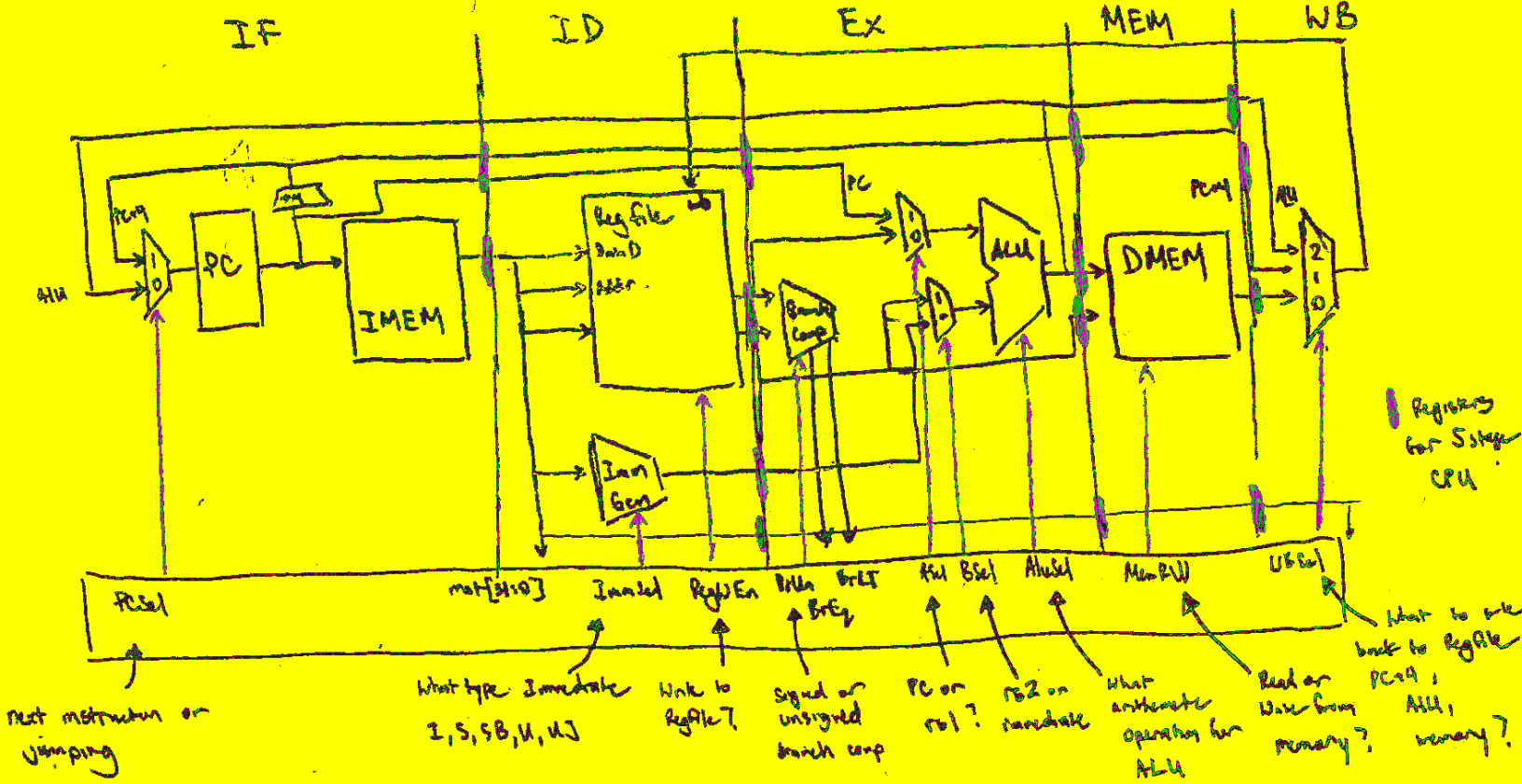
Processor

Processor (CPU): implemented directly in hardware (ISA)

- Datapath : contains hardware to perform operations
- Control : part of processor telling datapath what needs to be done

5 Stages of Data path

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



Critical Path for Single cycle = $t_{clock-q} + t_{IMEM} + t_{RegRead} + t_{muxes} + t_{ALU} + t_{DMEM} + t_{muxes}$

Clock frequency = $\frac{1}{\text{critical path}}$

Pipelining: Add registers between stages to speed up

Latency: time for 1 instruction to finish

Throughput: # instructions processed per unit of time

-Pipelining increases throughput but also latency

Pipelining Hazards

1. Structural Hazards
   - more than one instruction needs to use resource
   caused by: register file ID, WB, Memory IMEM, DMEM
   solved by: hardware
2. Data Hazards
   - data dependences between instructions
   caused by: instruction reads register before prev finished writing
   solved by: 1. Forwarding: result of EX, MEM sent to EX for next
   2. Stalls (lw) nop to stall

3. Control Hazards

- jump and branch and some of rest RE

Caused by: jump and branch instruction

Solved by: 1. Branch prediction: predict when to go from pred

kill: instruction if not taken

2. Stall

Double pumping: allows writing and reading from reg file in one stage