CS 61C Final Review

Logistics

Scope

- all lectures up to CALL (Mon 9/27)
- disc 1-5
- labs 1-4
- hw 1-4
- 1 and 2a

Topical Review with Questions

- Number Representation (Avi)
- C (Ryan)
    - Basics
    - Pointers
    - Arrays
- Memory Management (Ryan)
- Floating Point (Ryan)
- RISC-V Basics (Avi)
- RISC-V Coding (Avi)
- Instruction Formats (Avi)
- CALL (Avi)

Number Representation

- common questions are just converting between bases
- Fall 2018 Q1b
    - binary to decimal: 0b01101101 –> 109

Example

- 26_{10} to binary
    - 16+8+2 so 0b11010
- 26_{10} to hex
    - pad binary with leading 0s to fill nibble
    - 0b0001+1010 = 0x1A

Quick Tricks

- how to represent power of two in binary
    - only 1 bit is 1, the rest are 0
- representing one less than a power of 2
    - all ones up to the next power of 2
- Two's Complement
    - like normal except you start with all 1s and use 0s to negate

Two's Complement: Negative Nums

- treat most sig bit (leftmost_ as adding a negative power of two

Example: interpreting this number

	1	0	1	0	1	1	0	1	
8-bit unsigned:	2^7		2^5		2^3	2^2			$2^0 = 128 + 32 + 8 + 4 + 1 = 173$
8-bit 2's comp:	-2^7		2^5		2^3	2^2			$2^0 = -128 + 32 + 8 + 4 + 1 = -83$

Converting Binary to Hex

0b10100

- hex is implicitly unsigned
- so converting a negative number to hexadecimal is hard
- -64, first convert to binary, then convert binary to hex
- just (mod 16) and take the remainders!
    - fill in from least significant bit to most
    -

Bias Notation

- fixes the problem of converting negative nums to hex
- in bias notation we shift the unsigned interpretation of the number
- 8-bit with a bias of -253
    - smallest representatable num = 0b0000_0000 is 0-252 = -252
    - largest num = 0b1111_1111 = 255 - 252 = 3
- we don't have to put the bias in the middle!
- bias of 0 is just a regular unsigned bit

Fall 2019 Q1

$(37)_{10} = (25)_{16}$

Step by step solution

Step 1: Divide $(37)_{10}$ successively by 16 until the quotient is 0:

37/16 = 2, remainder is 5

2/16 = 0, remainder is 2

Step 2: Read from the bottom (MSB) to top (LSB) as 25. So, 25 is the hexadecimal equivalent of decimal number 37 (Answer).

Decimal:	0	1	2	3	4	5	6	7
Hexa	0	1	2	3	4	5	6	7
Decimal:	8	9	10	11	12	13	14	15
Hexa	8	9	A	B	C	D	E	F

Negate the following nibble binary/hex numbers, or write N/A if not possible. Remember to write your answer in the appropriate base.

Representation	Binary/Hex	Negation
Unsigned	0b0101	N/A (no negatives in unsigned)
Bias = -7	0b0100	0b1010
Bias = -7	0xF	N/A (out of range)
Two's Complement	0b1100	0b0100
Two's Complement	0xA	0x6

- 0xF = 15 so out of range!
- 0b0100 = 8, bias of -7 = 0b1010

CHEAT SHEET STUFF

Cheat Sheet Stuff

- N bits represents 2^N "things"
  - N-bit unsigned: $[0, 2^N - 1]$
    - Smallest: all 0s, base 10 value 0
    - Largest: all 1s, base 10 value $2^N - 1$
  - N-bit two's complement: $[-2^{N-1}, 2^{N-1} - 1]$
    - Smallest: 1 in sign bit followed by all 0s, base 10 value -2^{N-1}
    - Largest: 0 in sign bit followed by all 1s, base 10 value $2^{N-1} - 1$
    - base 10 value of -1 is represented by all 1s
    - To negate: flip all bits and add 1
    - ^ think about why for all of these!
  - N-bit bias notation with bias *b*: $[b, 2^N - 1 + b]$
    - Smallest: all 0s, base 10 value *b*
    - Largest: all 1s, base 10 value $2^N - 1 + b$
- Glance over definitions of "sign and magnitude" from lecture slides (less important than the three above notations, but will come up occasionally)

Bitwise Operations

- & = AND = turns bits ON
- | = OR = turns bits OFF
- ^ = XOR = flips bits

C Types

- C is weakly typed (casting is hard)

Types

- char (8 bits, 1 byte)
- int (32 bits, 4 bytes)
  - int32_1, uint8_t, etc. have explicit sizes given by `<stdint.h>`
- int* (always either 32 bit of 64 bit based on operating system)

False Values

- ''\0' = null terminator of char
- 0
- NULL pointer

Pointer Arithmetic

If you do arithmetic with pointers, remember that you're incrementing in sizeof(pointer_type), not bytes!

Example: assume `sizeof(int) == 4`

```
int x = 5;
int *y = &x;              // y = 0x4000
y = y + 2;                // y = 0x4008
char* z = "bears!"        // z = 0x2000
z = z + 2;                // z = 0x2002
```

Bit Manipulation

- review lab 2 exercise 3!!

Example (Fa19 Q3)

```
1  void ConvertTo2sArray(int32_t *A) {
2    while (*A) {            // checks if A != 0x00000000 (terminator)
3      if (*A < 0) {         // checks if value in A is negative
4        ConvertTo2s(A);     // calls other function on address
5      }
6      A = A + 1;            // increments A's address by 1
7  }
8  }
9
10 void ConvertTo2s(int32_t *B) {
11   *B = (*B & 0x7FFF_FFFF) + 1;    // converts to 2s complement
12 }
13
```

Structs

- kinda like classes
- also just some sequence of bits

sizeof

- returns num of bits being occupied
- sizeof an array and sizeof a pointer are diff based on the size of the array and the platform you are on

```
char x[5] = {0, 0, 0, 0, 0};
char *y = calloc(5, sizeof(char));
//size of
--> sizeof(x) == 5 * sizeof(char) == 5
--> sizeof(y) == sizeof(char *) == 4
// (or 8 depending on your platform)
```

Memory

Endianness

- bucket of buckets!
- if we have data 0x01020304:

Address	0x100	0x101	0x102	0x103
Little Endian	0x04	0x03	0x02	0x01
Big Endian	0x01	0x02	0x03	0x04

Types

- **Stack =** function local variables, strings allocated as arrays
- **Heap =** dynamically allocated memory (with malloc, calloc, realloc)
- **Static =** global variables, statically allocated strings
- **Code =** machine instructions

Example

```
int a = 5;
int main(){
  int b = 0;
  char* s1 = “cs61c”;
  char s2[] = “cs61c”;
  char *c = malloc(sizeof(char) * 100);
  return 0;
}
```

- s1 = stack
- s2 = stack
- s1[0] = static
- s2[0] = stack
- c[0] = heap
- a = static

Practice

Floating Point

Practice Problems

- What's the largest representable number?
- What's the smallest representable number?
- What's the smallest unrepresentable number?
- What's the largest unrepresentable numbers?
- What's the smallest distance between two numbers? What about denorms?
- How do you represent a particular number in floating point?
    - It basically reduces to these last two (see discussion 3 for practice with other types of problems)

Conversion

Sign	Exponent	Mantissa/Significand/Fraction
1	8	23

- **Normalized** = $(-1)^{Sign} * 2^{Exp+Bias} * 1.significand_2$
- **Denormalized** = $(-1)^{Sign} * 2^{Exp+Bias+1} * 0.significand_2$

Exponent	Significand	Meaning
0	0	0
0	any	denorm
255	0	+ or $-\infty$
255	nonzero	NaN
1-254	any	norm

Example Fa18 Q1

- **minifloat** = 8-bits with 1sign, 5exp, 2mantissa, bias -15
    - SEEEEEMM

- how many NaNs do we have?
  - we have **6** because we need max exponent and nonzero mantissa bits
    - $2 * (2^2 - 1) = 6$
- **bit rep in hex of next minifloat bigger than 0x3F**
  - 0x3F = 0b0011_0111
  - which is SEEEEEEMM
  - increasing mantissa would cause overflow, so we need to increment the exponent once and wrap mantissa back to 0!
  - we get 0b0100_0000 = $0x40$ (or just $0x3F + 1$)
- **bit rep of encoding -2.5**
  - convert to binary: $-2.5 = -10.1_2$
  - sign = 1
  - exp + bias = exp - 15 = 1, so exp = 16 = 10000
  - .5 = 10
  1. represent num in binary
     1. $-2.5 = -10.1_2$
  2. normalize so that one number is before the binary point
     1. $-10.1 = -1.01 * 21 \rightarrow$ MM = 01
     2. Ex: 1101 -> $1.101 * 2^3$
  3. calculate exp bits
     1. 1 = EEEEE - 15 = 10000 (we need 16)
  4. sign bit
     1. 1 = negative
     2. 0 = positive
  5. Put it together!
     1. $1100_0001 = 0xC1$
- **What does this return?**

```
// NOTE: always round down to 0
minifloat should_be_a_billion() {
  minifloat sum = 0.0;
  for (unsigned int i = 0; i < 1000000000; i++) {
    sum = sum + 1.0;
  }
  return sum;
}
```

- if you have a really big number, the FP can't represent too small or too big of a number
  - you need to find the largest number that the minifloat can represent
- once the exp = 3, you would need to increase the sum by 2.0 to actually change the number
- When does incrementing mantissa increment by 1.0?
  - at $1.11 * 2^2 = 111 = 7$, so we would need the exponent to be 2

    - if the exponent gets larger, the represented number is $1.00 * 2^3 = 1000 = 8$
- **shorthand way** = once the exponent is greater than the number of mantissa bits, the number will start increasing by more than just 1.0 every time.
- so 8.0 is returned!

RISC-V

C to RISC-V

- C contructs are translated to assembly
- like while loops!

```
int s0 = 0;
while (s0 < 4) {
  do_work();
  s0 += 1;
}
```

```
  li s0, 0
while:
  li t0, 4
  bge s0, t0, done
  jal do_work
  addi s0, s0, 1
  j while
done:
```

- sw saves registers to memory
- lw loads memory to registers

Jumping

	labels	**registers**
no linking	j	jr
linking	jal	jar

Calling Convention

- make sure that when you are writing a function, you HAVE TO save your save registers and then restore them!
- temp variables can CHANGE if you jal to something else :')
- you need to save ur return address (ra) to the stack so you know how to get back when you jump to a different function

Example Su19 Q5

- implement strlcpy in RISC-V

    - `char* strncpy(char* destination, char* source, unsigned int n);`
    - takes in two char* args and copies up to the first n characters
    - if it reaches a null terminator, it copies that value into the destination and then stops
    - returns a pointer to the destination string

```
# iterative python-ish pseudocode:
i = 0
while (i != n):
  copy_one_char_from_source_to_dest
  if source_is_null_terminator:
    return dest
  increment source pointer
return dest
```

```
# iterative RISC-V solution
# a0 = dest, a1 = source, a2 = n
strncpy:
  li t0, 0    # initialize counter
  mv t2, a0   # store a0 in t2 as a temp
loop:
  beq t0, a2, end # check end of while loop
  lb t1, 0(a1)  # read char from source
  sb t1, 0(a0)  # copy char to dest
  beq t1, x0, end # check null terminator
  addi t0, t0, 1  # increment counter
  addi a1, a1, 1  # increment source
  addi a0, a0, 1  # increment dest
  j loop    # go back to top of loop
end:
  mv a0, t2   # t2 contains the original dest
  ret
```

- since we are loading and saving ints, we just need to do lb and sb since ints are just 1 byte
    - lw and sw depend on the system
- there's also a way to do it recursively!

```
# recursive solution
# a0 = dest, a1 = source, a2 = n
strncpy:
  # missing! we need to save the return address and s0
  addi sp sp -8
  sw ra 0(sp)
  sw s0 4(sp)

  beq a2, x0, end # n == 0
  lb s0, 0(a1)  # read *source
  sb s0, 0(a0)  # copy to *destination
```

```
  beq s0, x0, end # check if *source == 0
  # set up recursive call
  addi a0, a0, 1
  addi a1, a1, 1
  addi a2, a2, -1
  jal strncpy
  addi a0, a0, -1     # decrements the address back to the destination
end:
  # missing!
  lw ra 0(sp)         # restores ra and s0
  lw s0 4(sp)
  addi sp sp 8

  ret     # return (a0 is already dest)
# Q: What's missing?
# A: Calling convention (need to save s0 and ra)!
```

- CC contract
  - says that ra = contains the address to jump back to
  - and that sp is the same before AND after a function call
  - as long as these two rules are followed, everything works!

Instruction Formats

- SB = (B)ranches = conditional jumps
- U = (U)pper Immediates = long immediates
- UG = (J)umps = unconditional jumps
- R = (R)egister
- I = (I)mmediate + loads from memory
- S = (S)tores to memory

Practice Problem

`lw t5, 17(t6)`

- lw = 0b0000011 (I-type)
- funct3 = 010
- t5 = x30 = 0b11110
- t6 = x31 = 0b11111
  - 17 (in 12 bits) = 0b0000_0001_0001
- For I types
  - **imm[11:0] rs1 funct3 rd opcode**
  - 0000_0001_0001 11111 010 11110 0000011
  - 0b0000_0001_0001_1111_1010_1111_0000_0011

    - or 0x011FAF03
- DON'T mess up converting to hex!!!!

CALL

C = Compile

- C -> risc or other assembly languages
- produced assembly is similar to what you might write by hand

A = Assemble

- RISC-V –> object files
- expands psuedo instructions
- requires 2 passes to resolve labels within files
- produces machine code with relocation tables (identifies labels defined in *other* files) and symbol tables (identifies labels/addresses that *this* file defines)

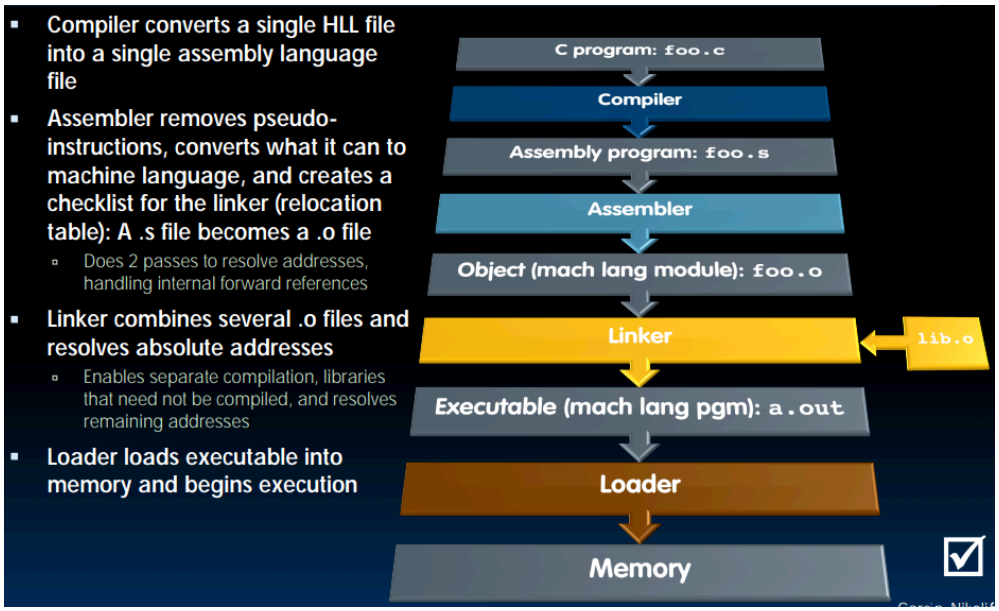
L = Link

- combines object files together into binary executable
- fulfills missing labels/addresses required by examining relocation and symbol tables

L = Load

- part of operating system
- loads machine code into memory for execution

Summary of CALL

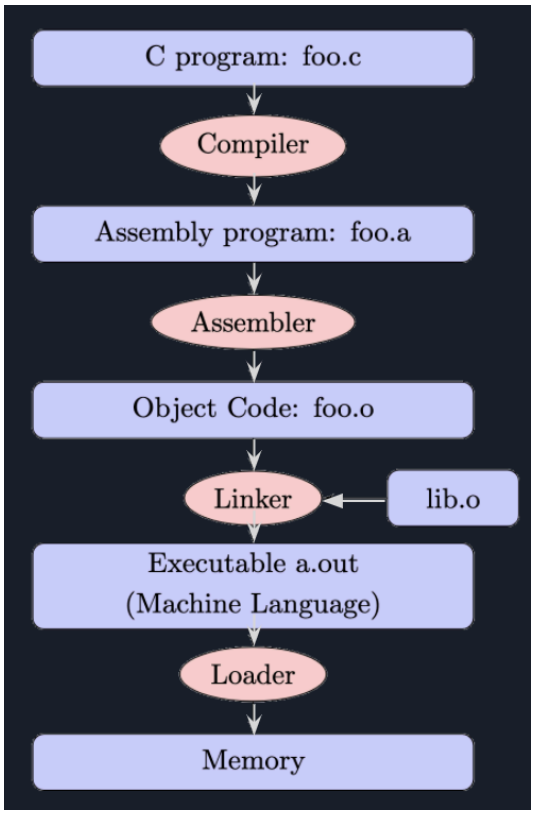


Lecture 13: CALL Notes

Interpretation vs Translation

- **Interpreter** = runs and executes program directly
    - python, java, c++ = easy to program, inefficient to interpret
    - assembly, machine code, bytes = difficult to program, efficient!
    - easier to write interpreter
        - closer to high-level, so better error and debugging messages
    - interpreters are slower and smaller
    - provides instruction set independence (runs on any machine)
    - usually open source
- **Translator** = converts a program from source language to equivalent program in another language and then runs it
    - helps to "hide" the program/intellectual property from its users
    - translates and compiles to a different machine!

Full Picture



- see the big picture by doing `gcc -O2 -S -c foo.c`, the `-S` makes it verbose

Compiler

- input = high-level language code (like foo.c)
- output = assembly language code (like foo.s for RISC-V)
- output may contain pseudo-instructions
    - instructions that assembler understands but not machine (like "copy value from t2 to t1")
    - `mv t1 t2` –>`addi t1 t2 0`
    - assembler added things like `mv` so we don't have to use simple commands like `addi`

Assembler

- input = assembly code (like foo.s for RISC-V)
- output = object code, info tables (true assembly only)
    - e.g., foo.o for RISC-V
- reads and uses directives
- replaces pseudo-instructions with machine language and then creates object file

Assember Directives

gives directions to assembler, but doesn't produce machine instructions

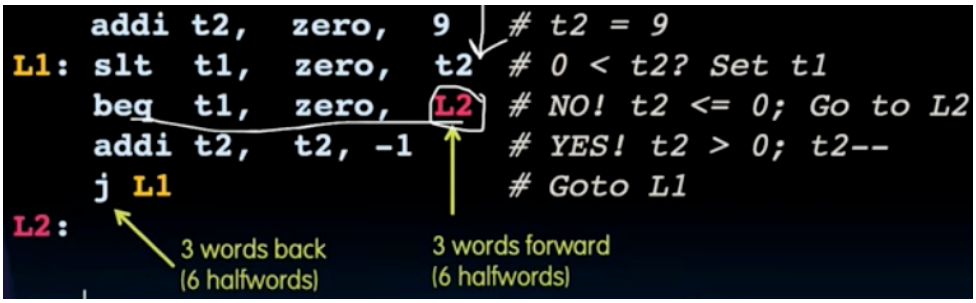
- `.text` = subsequent items put in user text segment (machine code)
- `.data` = subseqent items put in user data segment (source file data in binary)
- `.global sym` = declares sym global and can be references from other files
- `.string str` = stores string str in memory and null-terminates it
- `.word w1...wn` = store the n 32-bit quantities in successive memory words

Pseudo-instruction Replacement

- `neg t0 t1` is the same as `sub t0 zero t1`
- `la t0 str`
    - `lui t0 str[31:12]` then `addi t0 t0 str[11:0]` (STATIC addressing)
    - `auipc t0 str[31:12` then `addi t0 t0 str[11:0]` (PC-relative addressing)
- don't forget!!!!
    - sign extended immediates
        - machine code can only do operations on 32 bits, so if you have 12 bits, you have to sign-extend it to 32 bits
    - Branch imms count halfwords
        - each line is a "word", so if we wanted to travel back 3 lines, we would go back 6 halfwords

Producing Machine Code

- simple case
    - arithmetic, logical, shifts, etc.
- branches and jumps are PC-relative (e.g., beq/bne, jal)
- once pseudo-instructions are replaced by real ones, we know by how many instructions to branch/jump over
- "Forward Reference" problem
    - branch instructions sometimes branch to labels that are forward in the problem (like a loop)
    - SOLVED: take two passes over the program
        - first = remember position of labels
        - second = use label positions to generate the code
    - Example of halfwords and remembering label positions:



- for PC-relative jumps (jal) and branches (bne, ben, etc)
    - `j offset` (pseudo-instructions that expands to `jal zero, offset`)
    - count the number of instruction "half words" between the target and jump to determine the offset (position-independent code aka PIC)
- references to static data
    - `la` gets broken into `lui` and `addi`
    - requires the full 32-bit address of data

Tables

- **Symbol table** = list of "items" in this file that may be used by other files
    - labels = function calling
    - data = anything in the .data section, variables that may be accessed across files
- **Relocation table** = list of "items" whose address this file needs
    - basically saying here's something to fill in the blank, what do we need to remember/store?
    - any absolute label jumped to (using `jal`, `jalr`)
        - internal
        - external (like lib files)
        - EX: the `la` instruction (e.g., for `jalr` base register)
    - any piece of data in static section

- EX: the `la` instruction (e.g., for `lw`/`sw` base register)

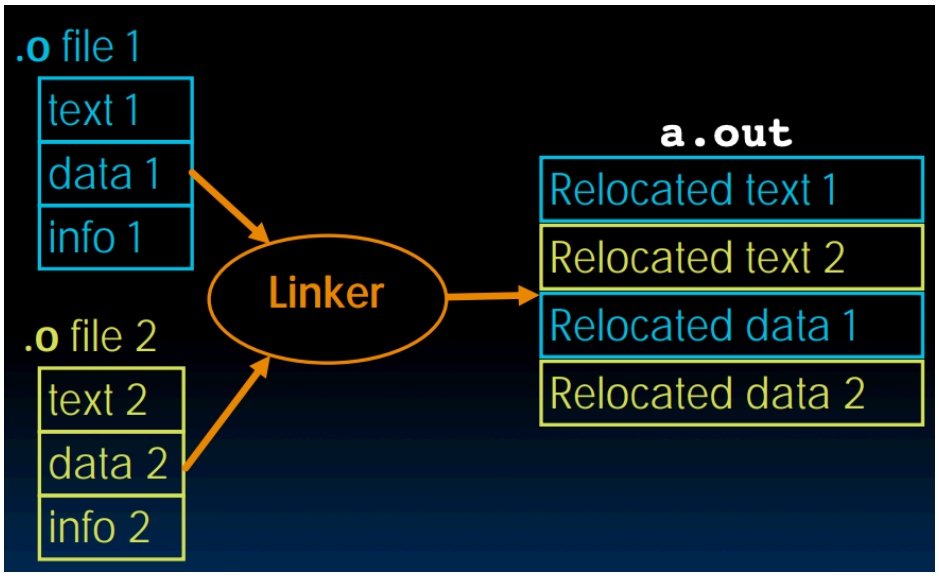
Object File Format

- **object file header** = size and position of the other pieces of the object file
- **text segment** = the machine code
- **data segment** = binary representation of the static data in the source file
- **relocation information** = identifies lines of code that need to be fixed up later
- **symbol table** = list of this file's labels and static data that can be referenced
- **debugging information**

Linker

General Info

- input = object code files, info tables (like foo.o, libc.o)
- output = executable code (a.out for RISC)
- combines several object (.o) files into a sngle executable aka "linking" them together
- compiles files, BUT doesn't have to recompile if you change one file!



Steps

1. take *text* segment from each .o file and put them together
2. take *data* segment from each .o file, put them together, and concatenate onto the end of text segments
3. resolve references (i.e., handle each entry from reloc table, fill in all absolute addresses)

4 Types of Addresses

- PC-Relative Addressing
  - ben, bne, jal, auipc/addi
  - never need to relocate (cuz PIC aka Position-Independent Code)
- Absolute Function Address

    - auipc/jalr
    - always relocate
- External Function Reference
    - auipc/jalr
    - always relocate
- Static Data Reference
    - lui/addi
    - always relocate

Relocation Editing

- J-format = jump/jump and link
    - the upper bits need to be modified
- I-, S- = loads and stores to variables in static area
    - all relative to global pointer so all the upper bits need to be modified based on where the static area begins
- conditional branches don't need to be modified since they are PC-relative!

Resolving References

- the linker assumes the first word of the first text segment is at 0x10000
- it knows the length and ordering of each text and data segment
- it calculates the absolute address of each label to be jumped to and each data being referenced
- to resolve!
    - search for reference (data or label) in all user *symbol tables*
    - if not, search the library files
    - once absolute address is determined, fill in the machine code appropriately
- *conflicting symbol type* = occurs when the machine finds two of the same symbol
- output = executable file containing text, data, and header

Static vs Dynamically Linked Libraries

- what we just described is statically linked
    - library is part of the executable ("baked in")
- dynamically-linked libraries (DLL)
    - executable doesn't have to recompile every time
    - puts the responsibility on the loader to get the library when it runs
    - space/time issues
        - storing a program requires less disk space
        - sending a program requires less time
        - 2 programs requires less memory since they share a library

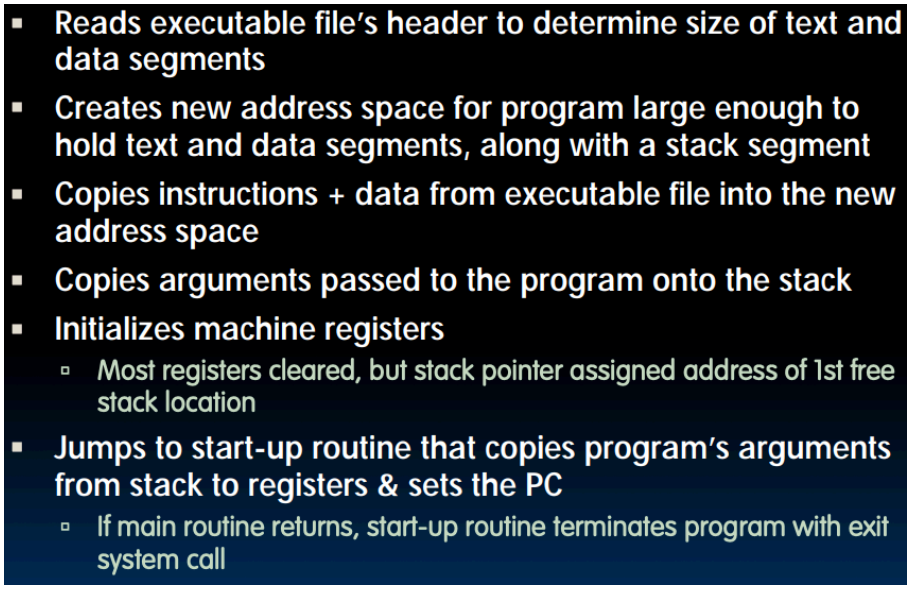
    - upgrades
        - replacing one library file upgrades every program that uses the library
        - but now the executable isn't enough! :')
    - prevailing approach to dynamic linking uses macine code as the "lowest common denominator" aka linking code only at the machine level

Loader

General

- Input = executable code
- output = running programs
- executable files are stored on disk
- loader's job is essentially to load the file onto memory and start running it (aka loader = OS)

How it works



SDS/Boolean Algebra/FSM

Hold Time

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

- look for the shortest combinatorial path
- the longest that the path can be stable