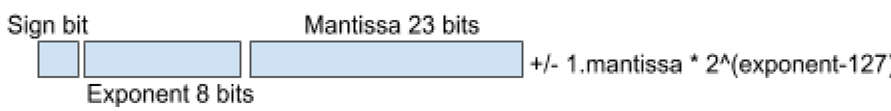
CSE30 Final Review

This review doc summarizes essential concepts covered in lectures. For detailed explanation, please also kindly refer to in class examples. Created by Yilin.

Number representation:

1. Unsigned binary number - decimal $101_2 = 3_{10}$
    a. Signed
        i. 1st bit to represent +/-, rest bits represent **magnitude**.
        ii. $101_2 = -1$;
2. 2's complement
    a. Inverting all bits and add 1(negative numbers)
        i. -2_{10} = 0010 in magnitude
        ii. Flip bits => 1101
        iii. Add 1 => 1110
        iv. 1001_2 flip bits => 0110_2
        v. Add 1 => 0111_2
        vi. $0111_2 = 7_{10}$, signed is negative => -7_{10}
3. Hex binary
4. Hex decimal
5. Float
    a. How to represent numbers like ½, 0.001, $6.023*10^{23}$
    b. IEEE standard floating point



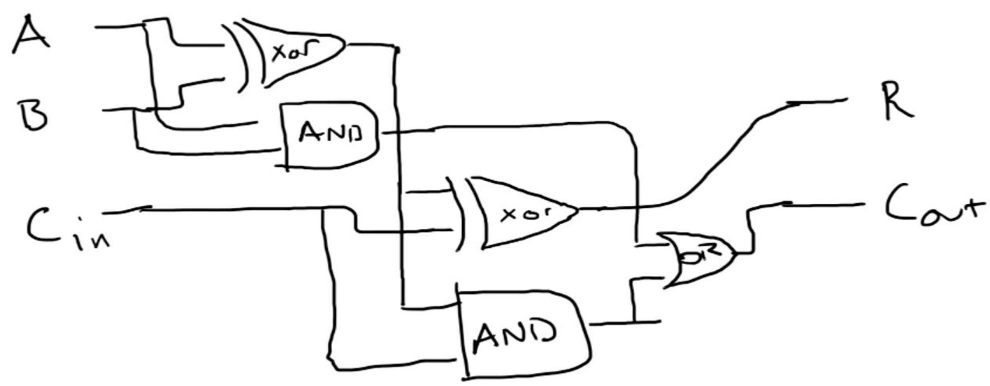
    c. Always have a 1 in the beginning of the mantissa(implicit existence)
    d. Exponent = exponent bits - 127(bias)
    e. zero: exponent all 0s, mantissa all 0s.

Bitwise operation

1. and/ or/ xor
    a. And: both inputs are 1, output 1, otherwise output 0
    b. Or: both inputs are 0, output 0; otherwise output 1
    c. Xor: exactly one of the input is 1, output 1; otherwise output 0
2. Masking - use add to clear bits

Combinatorial logic

1. Show a circuit, and fill the truth table
   a. AND operator: returns 1 when both of the inputs are one
   b. OR operator: returns 1 when either one of the inputs is one
   c. XOR operator: returns 1 only when one of the inputs is one
2. Full adder:



3. full adder

ASCII & C strings

1. Null-terminated strings
   a. “Hello” = ‘H’ ‘e’ ‘l’ ‘l’ ‘o’ ‘\0’
   b. Length = 5 (strlen()的结果）
   c. Char str[4] = “four”; no null terminator(garbage data)
   d. Char str[5] = “four”; null terminator
   e. Char str[3] = “four”; **warning!**
2. Operations on strings
   a. Strdup - copy
   b. Strlen - return the length of a string. # of characters before encounter ‘\0’.
   c. Changing characters

Undefined Behavior:

- char* my_str = “asdf”;

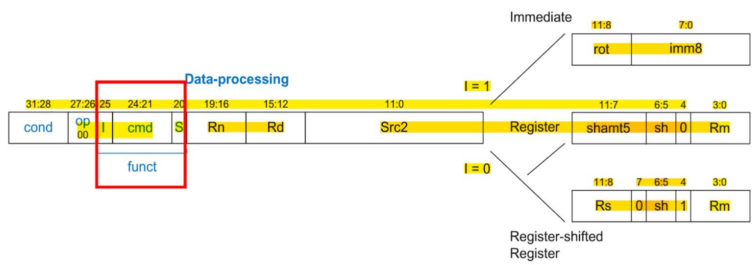
- Do not know where is it stored due to undefined behavior
- To solve this, we use strdup, which copies “asdf” and stores it in **heap**

- Examples
    - Memory Leaks
    - Incorrect execution (crashes, incorrect results)
    - Accidentally correct execution
        - I.e. Garbage values just so happen to line up with desired values
    - Reading/writing memory out of bounds
        - I.e. allocating room for 3 words (12 bytes) on the stack, then trying to access the 13th byte
- arr[n] means: *(arr + n)
- int size = 10;
  int * some_num = malloc(sizeof(int) * size);
  Some_num[12] = 1;
- **Cause undefined behavior: could have changed the value stored after some_num.**

Machine code

1. Encoding of register

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\bar{Z}$
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	$\bar{C}$
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	$\bar{N}$
0110	VS	Overflow / overflow set	V



Basic Assembly functions

1. Mov
2. Add
3. Sub
4. Ldr
5. lsl - logic shift left:
    a. 0x80000000 -> 0x0000 0000
    b. N = 0, Z = 1, C = 1, V = 0
6. …

CPSR

1. 0xFFFFFFFF (-1) + 0x00000001 (1) = 0x00000000(-0)
    a. Does not set N bit because result is not negative
    b. Does set Z bit because result is **zero**
    c. Does set C bit because there is a carry
    d. Does not set V bit because it’s only set when two + -> -
        i. eg: 0100 0000 + 0100 0000 -> 1000 0000

2. 0xFFFFFFFF (-1) + 0x00000002 (2) = 0x00000001 (1)
    a. Carry bit is set
3. **Running commands based on CPSR**

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\overline{Z}$
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	$\overline{C}$
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	$\overline{N}$
0110	VS	Overflow / overflow set	V

Labels and Loops

1. Program counter
    a. Where we are in a program (which **instruction**)
    b. Increase by 4 because every instruction is 4 bytes
    c. If ask for an instruction, pc is always 8 bytes ahead
2. B label
    a. Branch
    b.
3. Loops
4. If statement

Memory instructions

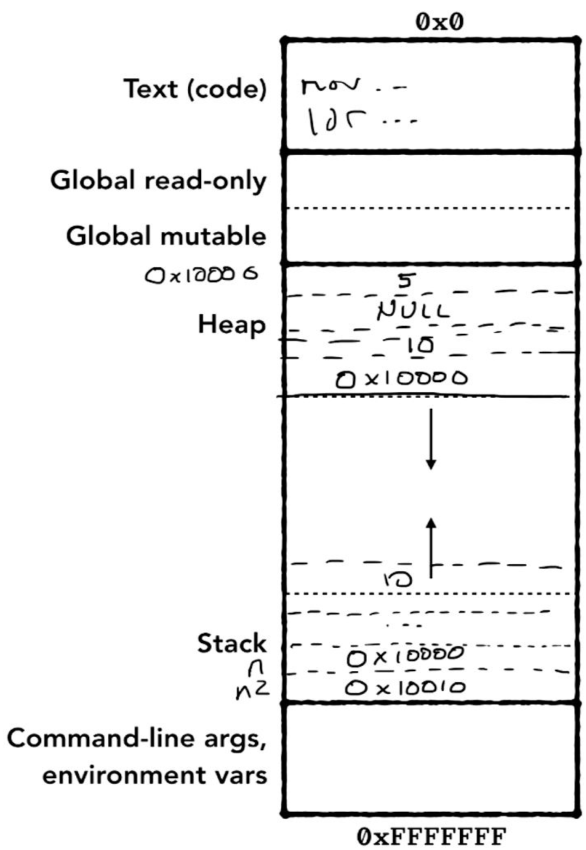
1. Loading and storing bytes
2. Loading and storing words
3. Pushing and popping

Calling

1. **bl** command
    a. r0 - r3 are often overwritten in the process of calling a function. Hence, the caller must save those registers if it depends on any of its own arguments (original values of r0 - r3) after a function returns.

Preserved	Nonpreserved
Saved registers: R4–R11	Temporary register: R12
Stack pointer: SP (R13)	Argument registers: R0–R3
Return address: LR (R14)	Current Program Status Register
Stack above the stack pointer	Stack below the stack pointer

2. Stack pointer's role
    a. Close to where the command-line arguments are stored when the program starts
3. Callee-save and caller-save
    a. Caller must preserve these registers: r0, r1, r2, r3 and r12 (caller save rule)
        i. Stack below the sp
    b. Callee must preserve these registers: r4 - r11, sp, lr (callee save rule)
        i. Stack above the sp
4. Passing in arguments in registers
5. Restoring before returning
    a. Return from a function: MOV pc, lr
    b. PUSH {lr} at the start of a non-leaf function + POP {pc} at the end
        - This is done to maintain the stack frames of functions
        - Link register is stored on the stack to return to the caller after the callee terminates
    - Recursive function calls




```
struct Node* {
  int value;
  struct Node* next;
};

typedef struct Node Node;

Node* make_node(int v, Node* n) {
  Node* new_node = malloc(sizeof(Node));
  new_node->value = v;
  new_node->next = n;
}

int main() {
  Node* n = make_node(5, NULL);
  Node* n2 = make_node(10, n);
}
```

Heap memory

1. Using malloc for heap-allocated arrays (has todo with sp)
    a. Argument: number of **bytes**
    b. Returns an **address**
2. Using malloc for struct data
3. Appropriate use of free()
4. **global variable is stored in global mutable block**

Stack Memory

1. Where stack memory should go for struct and array

2. Correct copying and referencing
    a. Stack copying: when the pass in arguments are not pointers, copy to stack
        i. f(Pointer p) {p.x = 22;} - calling f **does not** change the value of p in main
        ii. f(Pointer* p) {p.x = 22;} - calling f **does** change the value of p in main
        iii. **Reminder: pass in an array is same as pass in pointer**
    b. In java, it's more like the code with pointer because java heap allocates all its objects
3. How & gives the address

The type of x is...	The compiler generates code to...	Example
primitive (int, char)	Pass (copy) directly to callee	`int x = 10;` `f(x); // 10 passed in r0`

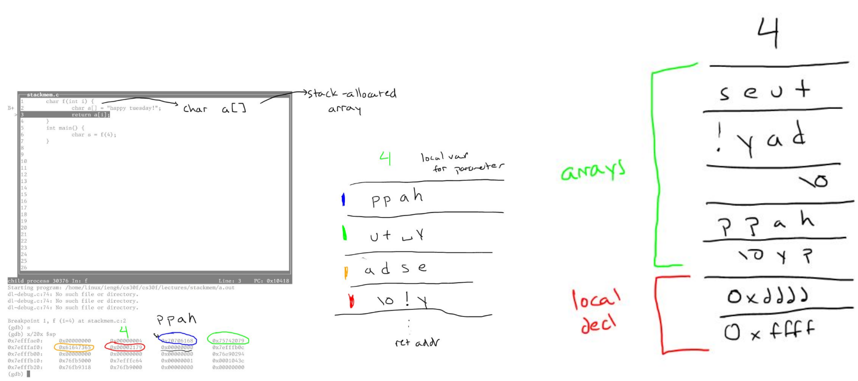
pointer	Pass (copy) directly to callee; copies an address	`int* x = malloc(sizeof(int));`
		 `f(x); // address returned from`
		 `// malloc passed in r0`

array	Pass ***address*** of array directly to callee	`char cs[] = "abcd";`
		 `f(cs); // address for start`
		 `// of cs passed in r0`

struct	***Copy*** struct contents to callee	`struct Point p = {1, 4};`
		 `f(p); // 1, 4 copied to stack`
		 `// frame for f to use`

Stack Layout

1. Grows in decreasing address order
    a. char s1[] = “Happy”;
       char s2[] = “Tuesday”;
       “Happy” is stored at higher address and “Tuesday” is stored at lower address
    b. char s1[] = “Happy”;
       int fs = 0xffff;
       char s2[] = “Tuesday”;
       int ds = 0xdddd;
       Because gcc puts local first, and arrays afterwards



gcc结果

Malloc and free implementation

1. char s[4] = “hello!”; sizeof(s) -> 4 instead of 7
2. char* s = strdup(“hello!”); sizeof(s) -> 4 instead of 7 because only one word

3. char* s = strdup(“hello!”); sizeof(*s) -> 1 because *s returns a char （type）
4. char s[10] = “hello!” ; sizeof(s) -> 10
5. Struct A{ char s[10]; char* name;}; sizeof(A) -> 16 because multiple of 4（padding) (largest field)
6. Struct A{ char s[10]; }; sizeof(A) -> 10 because treated as char array
7. Struct A{ char s1[5];char s2[9]; }; sizeof(A) -> 14 because rounded to the biggest size
8. Struct A{ short s1[5];char s2[9]; }; sizeof(A) -> 20 because rounded to the biggest size
9. Struct A{ char s1; char s2; int i; }; sizeof(A) -> 8 because char can be fitted in
10. Struct A{ char s1; int i; char s2; }; sizeof(A) -> 12 because char is separated
11. If it comes to the case where one struct has a struct field, only consider the largest field in those struct.
    a. Struct A {char c1; int i;}; struct B {char c2; struct A;}; sizeof(B) -> 12; c2 takes 4 bytes of space because the largest field is int in these two structs.
    b. Struct A {char c1; int i;}; struct B {struct A; char c2;}; sizeof(B) -> 12, same thing
12. Signature:
    a. void* malloc(size_t size);
       void* free(void* ptr);
    b. PA6

Caching

1. The index in a direct-mapped cache, how to evict/ replace data
2. CPU:
    a. Registers
3. Random-access memory
    a. Connected with CPU in some physical distance
    b. Accessing the data in RAM is 100 times longer than accessing that in registers
        i. Instructions such as: LDR, STR
4. **Cache**: for recently used memory
    a. Not very large (in class 8 entries)
    b. Fetch address:
        i. 0x0001 0004
           Last 2 bytes: 0 4
           0000 0100
           Last 5 bits - 1st 3: the index in cache; last 2: byte offset

        ii. Byte offset:
           Which part of the word u want
    c. Representation:
        i. 1st bit to indicate use
        ii. 27 bits for tag
        iii. 32 bits for data
    d. How to get data out?
        i. Go to index first
        ii. Compare two tags, match**?** Return data **:** update the cache

e. **Str instruction:**
    i. **Write-through:** Write to cache, and to memory(update cache & memory)
    ii. **Write-back: Dirty bit: (set dirty bit 1st)**
        1. **When in the future, encountering an address with ≠ tag, write into memory. (waits to save to memory)**

Virtual Memory

1. Sharing the same physical memory
    a. In process, same size of memory used in physical memory, but not necessary exactly the same address.
    b. 1st 5 bytes: **page (different size) PAGE TABLE**
        i. Page could be different in physical memory
        ii. Page table ≈ 4MB
        iii. Each entry gets an index
        iv. 1 bit indicates use or not
            1. If there is mapping, set to 1
        v. If page table is full, **data overlapping**
            1. Solution: **mmap (asking OS for more memory)**
        vi. **Access address that is not mapped in the page table, segmentation fault. (OS has not blessed it….)**
        vii. **If we use more and more memory, OS automatically allocate pages until segment fault.**
        viii. What if the physical memory is **full**
            1. MMU(memory management unit in CPU)
               Only knows one page table at one time
               Switch page tables so that convert to the right page table to run the process.
            2. OS -
                a. telling MMU which process is running so which page table to use
                b. Track used space and swap if needed
        ix. **SWAP???(virtual memory → hard disk, if physical space not enough)**
    c. 3 left bytes: **offset**