



1/ max: LUDL 1 ; op1 2  
 SUBL 2 ; op2 - op1  
 JNEG op1big:  
 LUDL 1  
 RETN ; op2  
 op1big: LUDL 2  
 RETN ; op1

SP → RETN  
 SP+1 op2  
 SP+2 op1

2/main: LODD index  
 Ex: DESR 2 ; sp := sp - 2  
 LOCO 0 ; ac := 0  
 STOL 0 ; m[sp+0] := ac (sum := 0) [sum]  
 STUL 1 ; m[sp+1] := ac (i := 0)  
 5: SUBL 2 ; ac := ac - m[sp+1] (i - N)  
 JZER 15 ; jump if (i == N)  
 JPUS 15 ; jump if (i > N)  
 LODL 0 ; ac := m[sp+0] (sum)  
 ADDL 1 ; ac := ac + m[sp+1] (sum+i)  
 STOL 0 ; m[sp+0] := ac (sum = sum+i)  
 LOCO 1 ; ac := 1  
 ADDL 1 ; ac := ac + m[sp+1] (1+i)  
 STUL 1 ; m[sp+1] := ac (i+1)  
 JUMP 5;  
 15: LODL 0 ; ac := m[sp+0] (sum)

2 places

INSI 2 ; sp := sp + 2, move stack pointer down  
 STOL 1 ; m[sp+1] := sum (return sum)  
 RETN ; return to caller

3/ STOD result: ; store ac to result location count

4/ adder: LODL 1 ; get 1<sup>st</sup> arg from stack into ac (data/r)  
 STOD myent: ; store count at location myent:  
 LODL 2 ; get 2<sup>nd</sup> arg from stack into ac (data addr)  
 IISHI ; push indirect 1<sup>st</sup> datum to stack  
 ADDD myed: ; add 1 (value at myf1:) to addr in AC  
 STOD myptr: ; store new addr to location myptr

5/ MySub: LODL 2 ; ac := topNum  
 SUBL 3 ; ac := ac - bottomNum

↓ low add  
 ↓ place for local variable  
 ↓ return address  
 ↓ result  
 ↓ topNum  
 ↓ bottomNum

JNEG neg:  
 LODL 2 ; ac := result  
 PUTI ; m[ac] = m[sp]

↓ high address  
 ↓ sp is at local variable (value of subtraction)  
 ↓ gain going subtraction into result

LOCO 0 ; ac := 0  
 RETN  
 neg:  
 LODL 2  
 PUTI  
 LODD cn1: ; ac := m(x) x: address of cn1  
 RETN

cn1: - 1 and value = - 1

6/ Recursion  
 LOOP: LODD lpcnt  
 JZER RTN  
 SUBD c1  
 STOD faddr  
 PIISI  
 LODD adddr.  
 TSII  
 ADDD c1  
 STOD adddr  
 CALL FIB  
 TSIZ 1

P1: LODD adddr.  
 TSII  
 ADDD c1  
 STOD adddr  
 CALL FIB  
 TSIZ 1

P2: PUSH  
 LODD faddr  
 PIISI  
 ADDD c1  
 STOD faddr: ; inc, store pointer for next f[n]

JUNE LNE: ; goto top for next pass

FIB: LODL 1 ; fib func loads arg from stack  
 JZER FIBZER: ; if fib(0) go to FIBZER  
 STOD c1: ; dec arg value in AC (arg-1)

JZER FIBONE: ; if fib(1) go to FIBONE  
 STOD LpCnt: ; number of iterations in LpCnt  
 LODD c0: ; load a 0 into AC  
 STOD fm2: ; store 0 in fib(n-2)  
 LODD c1: ; load a 1 into AC  
 STOD fm1: ; store 1 in fib(n-1)

ITER: LODD LpCnt: ; LpCnt arg-1 iterations needed  
 JZER RTN: ; when LpCnt == 0 go to RTN  
 SUBD c1: ; dec arg value in AC (LpCnt - 1)  
 STOD LpCnt: ; store LpCnt for next iteration  
 LODD fm2: ; arg must be ≥ 2, fm2 initially fib(0)  
 ADDD fm1: ; fm1 initially fib(1), AC = fm2 + fm1  
 STOD tmp: ; store AC to tmp:  
 LODD fm1: ; now load AC with fib(n-2)  
 STOD fm2: ; replace old fib(n-2) with AC  
 LODD tmp: ; load AC with tmp: becomes fib(n-1)  
 STOD fm1: ; store AC as next fib(n-1) to fm1  
 JUMP ITER;

RTN: LODD temp: ; load AC with temp: final result  
 RETN

Recursion PIISI ; pushing (arg-1)  
 CALL FIB: ; fib(arg-1)  
 PIISI  
 LODL 1 ; load value (arg-1) to AC  
 SUBD c1: ; arg-1 is decremented  
 PIISI ; push (arg-2)  
 CALL FIB: ; call fib(arg-2)  
 INSE 1 ; remove (arg-2) from stack  
 ADDL 0 ; ac = f1 + f2  
 INSI 2 ; remove f1 & (arg-1) from stack  
 RETN

FIBZER: LODD c0:  
 RETN ; ac = 0 for fib(0)

FIBONE: LODD c1:  
 RETN ; ac = 1 for fib(1)

, num of fibs to do in TasCnt  
 , no more passes, go to done.  
 ; - passes remaining  
 ; load a pointer to fib arg  
 ; push arg for fib on stack  
 ; inc, store pointer for next d[n]  
 ; call fib(arg+stack)  
 ; clear stack on fib return  
 ; put return AC (fib(n)) on stack  
 ; load a pointer to result f[n]  
 ; pop result off stack into f[n]  
 ; inc, store pointer for next f[n]

JUNE LNE: ; goto top for next pass

FIB: LODL 1 ; fib func loads arg from stack  
 JZER FIBZER: ; if fib(0) go to FIBZER  
 STOD c1: ; dec arg value in AC (arg-1)

JZER FIBONE: ; if fib(1) go to FIBONE  
 STOD LpCnt: ; number of iterations in LpCnt  
 LODD c0: ; load a 0 into AC  
 STOD fm2: ; store 0 in fib(n-2)  
 LODD c1: ; load a 1 into AC  
 STOD fm1: ; store 1 in fib(n-1)

ITER: LODD LpCnt: ; LpCnt arg-1 iterations needed  
 JZER RTN: ; when LpCnt == 0 go to RTN  
 SUBD c1: ; dec arg value in AC (LpCnt - 1)  
 STOD LpCnt: ; store LpCnt for next iteration  
 LODD fm2: ; arg must be ≥ 2, fm2 initially fib(0)  
 ADDD fm1: ; fm1 initially fib(1), AC = fm2 + fm1  
 STOD tmp: ; store AC to tmp:  
 LODD fm1: ; now load AC with fib(n-2)  
 STOD fm2: ; replace old fib(n-2) with AC  
 LODD tmp: ; load AC with tmp: becomes fib(n-1)  
 STOD fm1: ; store AC as next fib(n-1) to fm1  
 JUMP ITER;

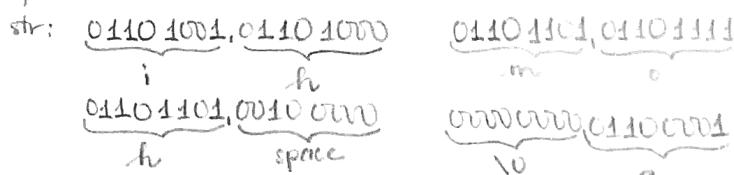
RTN: LODD temp: ; load AC with temp: final result  
 RETN

Recursion PIISI ; pushing (arg-1)  
 CALL FIB: ; fib(arg-1)  
 PIISI  
 LODL 1 ; load value (arg-1) to AC  
 SUBD c1: ; arg-1 is decremented  
 PIISI ; push (arg-2)  
 CALL FIB: ; call fib(arg-2)  
 INSE 1 ; remove (arg-2) from stack  
 ADDL 0 ; ac = f1 + f2  
 INSI 2 ; remove f1 & (arg-1) from stack  
 RETN

FIBZER: LODD c0:  
 RETN ; ac = 0 for fib(0)

FIBONE: LODD c1:  
 RETN ; ac = 1 for fib(1)

# Mic 1 - IO:



2 characters for every 16 bits word

\* STOD 4093<sup>RCVR</sup> Turn "ON" the RCVR and XMTR  
STOD 4095<sup>XMT</sup>

- Receiver will accept characters from a terminal device only if RCVR is on by setting the O bit

+ RCVR on: set B & clear D

+ Character arrived: set D & clear B

+ I is set: interrupt will be posted each time the done bit set

+ Character removes: clear the interrupt & done bit, will set busy

+ O is cleared: device is turned off, clear all other bits

- Transmitter: XMTR

+ O is set: XMTR will set done & force an interrupt if I is set

+ Any character moved to XMTR → cause D to clear & clear interrupt

+ Character displays → D is set, B cleared & interrupt if I set.

+ Move another character to XMTR → clear D & any interrupt.

\* Character "8" = 56 (in ASCII) = 48 + 8

Ex. 79 "7" = 48 + 7 = 55

$$\begin{array}{r} \text{#L} \\ \hline \text{"4"} \\ \text{"9"} \\ \text{"2"} \\ \text{"8"} \end{array} \quad \begin{array}{r} \text{"7"} \rightarrow -48 \rightarrow 7 \text{ (number)} \times 10 = 70 \\ \text{"9"} \rightarrow -48 \rightarrow 9 \\ \hline \end{array} \quad \begin{array}{r} + 9 \\ 79 \times 10 = 790 \end{array}$$

$$\begin{array}{r} \text{"2"} \rightarrow -48 \rightarrow 2 \\ \hline \end{array} \quad \begin{array}{r} + 2 \\ 792 \end{array}$$

Ex.  $8279 + 15 = 8294 / 10 = \text{remainder } 4 + 48 = "4"$

stash. LIFO

\* 4093 OJDB: O is on  $\rightarrow 1000 = 8 \rightarrow$  on: 8

$\downarrow$  Busy bit is off  $\rightarrow$  Done & not busy  $\rightarrow$  do sth more  
 $\downarrow$  Done bit is on

$1001 = 9 \rightarrow$  Not done & busy  $\rightarrow$  need to wait, pause more

\* ISHI ; add 2 characters at once

the stash: "T+I"

"T" "I"  
subd C255  $\rightarrow$  to check if it terminates the string yet  
Jmp crnl:  $\rightarrow$  create new line

call sb:  $\rightarrow$  swap 2 characters  $\rightarrow$  "T" "I"

(sb: loco 8)

pop  $\rightarrow$  pop "T+I"

ladd pstr1  $\rightarrow$  now point to "I"

top: ladd ent:  $\rightarrow$  ent; 30  $\rightarrow$  max count

\* RAM: is packaged as a chip. Basic storage: cell (1 bit per cell). Multiple RAM chips form a memory

SRAM (Static): Each cell stores a bit with a four or six transistor circuit. Retain values indefinitely, as long as it is kept powered. Relatively insensitive to electrical noise (EMI) radiation. Faster & more expensive than DRAM.

DRAM (Dynamic): Each cell stores bit with a capacitor. One transistor is used for access. Value must be refreshed every 10-100ms. More sensitive to disturbances (EMI, radiation) than SRAM. Slower & cheaper than SRAM

Trans Access Needs Needs Cost Application  
per bit time refresh? ECC? 100x Cache memory

SRAM 4 or 6 1x No Maybe 100x Cache memory

DRAM 1 10x Yes Yes 1x Main memory

(ECC, ECC = error detecting codes, error correcting codes)  
 $\Rightarrow$  SRAM faster than DRAM: All transistors, no capacitors to refresh

$\Rightarrow$  DRAM less expensive: One transistor & a capacitor cost much less than 6 or 7 transistors

$\Rightarrow$  Temporal & spatial locality (loops, function calls)

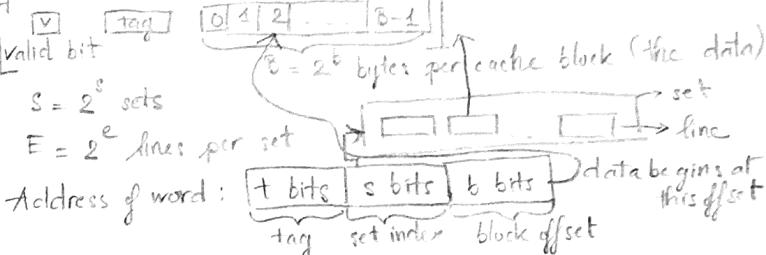
\* Conventional DRAM Organization:

- dw: total bits organized as 16 supercells of size w bits

Ex. 16x8 DRAM chip (16 supercells, 16x8)

- RAS ≠ CAS (a cycle)

\* Cache size:  $C = S \times E \times B$  data bytes



Ex. 64 bytes =  $2^6 \Rightarrow$  block offset: 6 bits

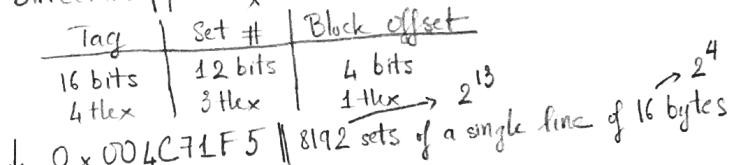
# of sets =  $2^{14} \Rightarrow$  S bits = 14 bits

Ex. Block size =  $2^4 = 16$

$\Rightarrow$  # of sets =  $2^{12} = 4K$

Address: 7#43 valid | tag | 0 | 0007 | Set A43

Direct Mapped:



000000 0 0100 110 | 0 0111 0001 1111 | 0101  
13 bit lines = 1823 Byte offset  
Tag = 0x26 (set/line number)

If 8192 cache lines were broken up into a 4 way set associative cache (4 lines per set, 2048 sets)  $2^{11} : 2^{13} \text{ sets} = 2^{11} \text{ sets} \times 2^2 \text{ lines per set}$

0000000 1 1000 1 | 111 0001 1111 | 0101  
11 bit lines = 1823 Byte offset  
Tag = 0x98

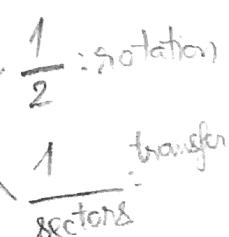
4 lines searched for this tag in set 1823

Direct Mapped: 1 line for every set

Hit: set  $\rightarrow$  Tag

Miss: Tag #

$$\frac{60 \text{ sec}}{\text{RPM}} \times 1000 \text{ ms/sec} \times 8$$



Taccess = seek + rotation + transfer

OJDB

0 load start:  $\rightarrow S : 1000$ , turn on 0 bit  
 1 std 4093  
 2 std 4095  
 3 bps: load 4095  
 4 end mask:  
 5 jzer print:  
 6 jump top:  
 7 print: load char:  $\rightarrow "Z"$   
 8 std 4094  
 9 subl c2:  $\rightarrow AC = "X" \quad ("Z" - 2 = "X")$   
 10 std char:  $\rightarrow char = "X"$   
 11 subl charA:  $\rightarrow AC = "X" - "A" > 0$   
 12 jpos cord: ch2:  
 13 jpos cord: ch2:  
 14 std char:  
 15 end: load 4093  
 16 subl mask: q: busy  
 17 jzer done: check to see if it receives anything  
 18 jump top:  
 19 done: halt  
 20 start: 8  
 21 mask: 10  
 22 data: "A"  
 23 ch2: "Z"  
 24 char: "Z"  
 25 c2: 2  
 $\rightarrow$  done, halt command  
 when RCVR is done  $\rightarrow$  when some key board input is available  
 (if RCVR is busy  $\rightarrow$  not done yet)  
 $\rightarrow$  the character is at index 2 from ["Z"]  $\rightarrow ["X"]$

xbsywt: load 4095	xbsywt: load 4095
subl xdmask:	subl xdmask:
jzer xrly:	=>
jump xbsywt	xrly: retr
xdmask: 10	xbsywt: xdmask: 10

sb: load 1  $\rightarrow$  memory address  
 pshl  
 loco 8  $\rightarrow$  byte swap on the content of the location  
 loop: jzer finish:  
 subl c1:  $\rightarrow (8-1)(\text{fill}1)$ ; add the missing code segments to  
 std lpcnt:  $\rightarrow$  lpcnt = each location  
 lodd 0  $\rightarrow$  top of the stack, now is "CS"  
 jneg add1:  $\rightarrow$  to know if we should copy 0 or 1  $\rightarrow$  negative  
 addl 0  $\rightarrow$  add the top stack (CS);  
 stdl 0  $\rightarrow$  means add the value  
 lodd lpcnt:  $\rightarrow$  shift to the left by 1  
 jump loop:  
 ldd1: addl 0  
 adddl c1.  
 stdl 0  
 lodd lpcnt:  
 jump loop:  
 finish: load 2  
 popl  
 retr  
 c1: 1  
 lpcnt: 0  
 $\rightarrow$  procedure ends here  
 $\rightarrow$  data location for the procedure

AC = ~~0001 0011~~ | 0100 0011 addl 0: shift left  
 8 times

BSWARD

```

0 mar := ir ; rd;
01 c := smask' rd; (8bits mask)
02 a := mbr; (16bits)
03 c := rshift(c); if z goto 107;
04 a := lshift(a); if n goto 106;
05 goto 103;
06 a := a+1 ; goto 103;
07 mbr := a ; wr;
08 goto 0 ; wr;
    
```

Rshift:  
 103: a := lshift(1);  
 104: a := lshift(at1);  
 105: a := lshift(at1);  
 106: a := a+1;  
 107: b := band(ir, a);  
 108: b := bt(-1); if n then goto 0;  
 109: ac := rshift(ac), goto 108  
 110: goto 0;

```

    82: a := lshift(1); 93: d := 0
    83: a := lshift(at1); 94: alu := a; if n then goto 107
    84: a := lshift(at1); 95: c := c+1; goto 107
    85: a := lshift(at1); 96: alu := c; if n then 105
    86: a := lshift(at1); 97: b := bt(d-1);
    87: a := a+1; 98: d := d+a; — 96;
    88: b := band(ir, a); 99: alu := c; — 97;
    (build 6bit mask & put multiplier in b)
    100: ac := d-1; goto 0;
    101: mar := sp; ac := 0;
    102: mbr := d; wr;
    91: rd;
    92: c := (-1);
    
```

6/ 001100101.00111 IEEE 754 shift 6, +6  
 IBM shift 8, +2

IEEE: 127+6 = 133  
 $\rightarrow 1000 0101 \mid 100 1010 0111 0000 \dots$   
 IBM: 64+2 = 66  
 $\rightarrow 100 0010 \mid 010 0101 0011 1010 \dots$

Result: 01110001011101010000000000000000

1/ Convert base 10 into base 2, base 8, base 16  
 - Base 2: 119.75125  
 $\downarrow /2R \rightarrow 2 \downarrow \Rightarrow$   
 - Base 8:  $2^3 \Rightarrow$  combine 3 numbers  
 $\begin{array}{c} 01 \\ 110 \\ 111 \end{array}$   $\Rightarrow$  167  
 - Base 16:  $2^4 \Rightarrow$  combine 4 numbers  
 $\begin{array}{c} 03 \\ 111 \\ 0011 \\ 1100 \end{array}$   $\Rightarrow$  C8  
 - Negative number ... 328 328  $\rightarrow$  converted to 10110100  
 $\begin{array}{c} 1 \\ 1101 \\ 1110 \\ 0101 \\ 0111 \end{array}$   
 - 328  $\rightarrow$  1111111101111100

2/ Convert 2's Complement into base 10

+ Negative:  
 - Notice the sign if  $\neq 0$   
 - Change (flip) the signs to get the magnitude of the number  
 $\begin{array}{c} 1 \\ 0101 \\ 1011 \end{array}$

- Two's Complement

$\begin{array}{c} 1 \\ 0101 \\ 0011 \end{array}$

- Convert the magnitude to decimal:

$\begin{array}{c} 00100101 \\ 2 \\ 2 \end{array} = 25_{16} = 2 \times 16 + 5 = 37$

$\rightarrow$  Negative: -37

+ Positive:  $01101001_2 = 69_{16} = 6 \times 16 + 9 = 105_{10}$

3/ Convert  $\star$  into floating number (base 10)

+ IBM format:

1	7	24 bits
Sign	Exponent (base 16 excess 64)	Mantissa (0 ≤ mant < 1)
0	100 0011	0101 1100 0000 0000 0000
	= 67 - 64	00 00 00
	= 3 $\Rightarrow$ 16	
	$= 2^{12} \cdot (2^{-2} + 2^{-4} + 2^{-5} + 2^{-6})$	
	$= 2^{10} + 2^8 + 2^7 + 2^6 = 1672.0$	

+ IEEE 754 single precision:

1	8	23 bits
	(base 2 excess 127)	exponent hidden bit = 1, = 0 (if all 0's)
0	1000 0110	101 1100 0000 0000 0000
	= 134 - 127	00 00
	= 7 $\Rightarrow 2^7$	
	$= 2^7 \cdot (1 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5})$	
	$= 2^7 + 2^6 + 2^4 + 2^3 + 2^2 = 220.0$	

4/ Convert floating number into IEEE 754 single precision

$\begin{array}{c} 812.5_{10} \\ \times 2 \\ \hline 1.6250 \end{array} \Rightarrow$   
 $\begin{array}{c} 1101 \\ \downarrow \\ 1. \\ \Rightarrow 1.F \end{array}$   
 to the left: +  
 to the right: -  
 $\Rightarrow$  Move 1  $\Rightarrow 127 - 1 = 126$

Exponent: 0 111 1110  
 $\Rightarrow 0 \mid 0 111 1110 \mid 101 000 0000 0000$

5/ Add 2 IEEE 754 single precision

- Float 1: 01110010110111000000000000000000  $\Rightarrow 13.75$

- Float 2: 01011110110000000000000000000000  $\Rightarrow 0.8125$

Exponent: ①  $2^7 + 2^4 = 128 + 2 = 130 - 127 = 3$  } shift 4 bits of  
 ②  $126 - 127 = -1$  } mantissa ②

Mantissa: ① 101 1100 0000 0000 0000  
 ② 100 1101 0000 0000 0000 } Add  $\rightarrow 110 1001 0000 0000 0000$



```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 #define HEADERS 1
7 #define NO_HEADERS 0
8
9 typedef struct nament {
10     char name[26];
11     int addr;
12     struct nament *next;
13 } SYMTABENTRY;
14
15 void add_symbol( char *symbol, int line_number );
16 void generate_code( int );
17 void print_first_pass( int );
18 void append_table( void );
19 void dump_table( void );
20 int get_sym_val( char * );
21
22 SYMTABENTRY *symtab = NULL;
23 FILE *p1, *p2;
24 char cstr_12[13];
25
26 int main( int argc, char* argv[]){
27     int i, start, pc_offset = 0, pc = 0;
28     int linum = 0, object_file = 0, dump_tab = 0;
29     int line_number, new_pc;
30     char instruction[18];
31     char symbol[26];
32     if(argc > 1 && (strcmp(argv[1], "-s") == 0)){
33         dump_tab = linum = 1;
34     } else if( argc > 1 && (strcmp(argv[1], "-o") == 0 ) ){
35         object_file = 1;
36     }
37     if((dump_tab == 1) | (object_file == 1)){
38         start = 2;
39     } else {
40         start = 1;
41     }
42     if(argc > 1 && (strcmp(argv[1], "-c") == 0)){
43         rewind(p1);
44         while(fgets(inbuf, 80, p1) != NULL){
45             printf(" %s", inbuf);
46         }
47     }
48     void add_symbol( char *symbol, int line_number ){
49         SYMTABENTRY *_sym;
50         _sym = (SYMTABENTRY *) malloc (sizeof(SYMTABENTRY));
51         if(symtab != NULL){
52             _sym -> next = symtab;
53             symtab = _sym;
54         } else {
55             symtab = _sym;
56             _sym -> next = NULL;
57         }
58         strcpy(_sym -> name, symbol);
59         _sym -> addr = line_number;
60         return;
61     }
62     void append_table(){
63         while(symtab != NULL){
64             printf("%s \t\t%d\n", symtab -> name, symtab -> addr);
65             symtab = symtab -> next;
66         }
67         return;
68     }
69     int get_sym_val(char *symbol){
70         SYMTABENTRY *rec = symtab;
71         int cmp;
72         while(rec != NULL){
73             cmp = strcmp(rec -> name, symbol);
74             if(cmp != 0){
75                 rec = rec -> next;
76             } else {
77                 return rec -> addr;
78             }
79         }
80         return -1;
81     }
82     p1 = fopen("/tmp/daedalus", "w+");
83     unlink("/tmp/daedalus");
84     for(i = start; i < argc; i++){
85         if((p2 = fopen(argv[i], "r")) == NULL){
86             printf("Cannot open file named: %s", argv[i]);
87             exit(2);
88         }
89         while(fscanf(p2, "%d %s", &pc, instruction) != EOF){
90             if(pc == 4096) break;
91             new_pc = pc + pc_offset;
92             symbol[0] = '\0';
93             if(instruction[0] == 'U'){
94                 fscanf(p2, "%s", symbol);
95             }
96             fprintf(p1, " %d %s %s \n", new_pc, instruction, symbol);
97         }
98         while(fscanf(p2, "%s %d", symbol, &line_number) != EOF){
99             add_symbol(symbol, line_number + pc_offset);
100         }
101         pc_offset = new_pc + 1;
102         fclose(p2);
103     }
104     if(object_file){
105         print_first_pass(NO_HEADERS);
106         printf("4096 x\n");
107         append_table();
108         return 0;
109     }
110     if(linum){
111         print_first_pass(HEADERS);
112     }
113     generate_code(linum);
114     return 0;
115 }
116 void generate_code(int linum){
117     char instruction [18];
118     char symbol[26];
119     int i, j, pc, difference, mask, _symbol;
120     int old_pc = 0;
121     rewind(p1);
122     while(fgets(inbuf, 80, p1) != NULL){
123         printf(" %s", inbuf);
124     }
125     while(fscanf(p1, "%d %s", &pc, instruction) != EOF){
126         if((difference = pc - old_pc) > 1 ){
127             for(j = 1; j < difference; j++){
128                 printf("1111111111111111 \n");
129             }
130             old_pc = pc;
131             if(instruction[0] != 'U'){
132                 printf("%s\n", instruction);
133             } else {
134                 fscanf(p1, "%s", symbol);
135                 _symbol = get_sym_val(symbol);
136                 if(_symbol == -1){
137                     exit(3);
138                 }
139             }
140             for(i = 0; i<12; i++){
141                 cstr_12[i] = '0';
142             }
143             cstr_12[12] = '\0';
144             mask = 2048;
145             for(i = 0; i < 12; i++){
146                 if(_symbol & mask){
147                     cstr_12[i] = '1';
148                 }
149                 mask >>= 1;
150             }
151             for(i = 0; i < 12; i++){
152                 instruction[i + 5] = cstr_12[i];
153             }
154             printf("%s\n", &instruction[1]);
155         }
156     }
157     return;
158 }
159 void print_first_pass(int headers){
160     char inbuf[81];
161     if(headers != HEADERS){
162         rewind(p1);
163         while(fgets(inbuf, 80, p1) != NULL){
164             printf(" %s", inbuf);
165         }
166     }

```

0x100 | 0x101 | 0x102 | 0x103  
01 25 45 67

Big Endian. Least significant byte has highest address  
Little \_\_\_\_\_, lowest

~~164~~

170

COMP.3050.201

Test 2 ( 200 points)

4/9/18

**CLOSED BOOK/ LAPTOP. Calculators and two pages of notes allowed. No phones.**

NAME: Dangnhi Ngo

Give specific numbers where appropriate, not a general verbal description.  
Powers of 2 are on page 3. Exam is printed double-sided.

1. The following assembly program (which has line reference numbers attached to help you answer the questions below) will begin executing at location 0:

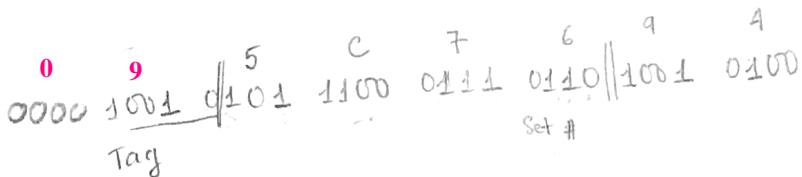
0	lodd start:					
1	stod 4093					
2	stod 4095					
3 top:	lodd 4095					
4	subd mask:					
5	jneg top		"	"	"	"
6 print:	lodd char: "A"	E	B	F	B	F
7	stod 4094		"	"	J	
8	addd c4: AC = "E"	I				
9	stod char: char = "E"	I	"	"	J	
10	subd chlim: AC = "E" - "G" < 0 > 0	<0	>0			
11	jneg cont:					
12	lodd chB:		"B"			"B"
13	stod char:		char = "B"			
14 cont:	lodd 4093					
15	subd mask:					
16	jneg top:					
17 done:	halt					
18 start:	8					
19 mask:	10					
20 chlim:	"G"					
21 chB:	"B"					
22 char:	"A"					
23 prompt:	"Please type in a positive integer"					
24 c4:	4					

- A. (15 pts) Line number 3 loads a value out of the transmitter status register. What value will it read when the UART is NOT ready to transmit?

q

- B. (40 pts) What are the first ten characters written to the display?

AEBFBFBFBF



2. (20 pts) Assume a direct-mapped cache with a 256-byte block size and 32K sets. (If needed,  $256 = 2^8$ ) What is the set number and what is the tag for address 0x095C7694? Show the answer in hex

Tag  $0 \times 12$

Set #  $0 \times 5C76$

$$2^4 \text{ offset} = 4 \quad 2^{12} \text{ set\#} = 12$$

3. (20 points) Now assume a direct-mapped cache with a 16-byte block size and 4K sets. Specify for each memory reference below whether it is a hit or miss. Assume that initially the cache is empty, but then the following memory references occur one after each other. (The "important" ones are worth more points.)

Set	79F	Tag	E8	Address	Result	Cache State	Address	Result	Cache State
			E879F0		Miss ✓		1110 1000    0111		1001 1111    0000
79F	E8	E8	E879F4		Hit ✓	E8			\$100
79F	A4	A4	A479F8		Miss ✓	A4			
79F	E8	E8	E879F8		Hit Miss (conflict) E8			X2	
79F	A4	A4	A479FC		Hit Miss (conflict) A4				
									Tag   Set #   Block offset
									16 bits 12 bits 4 bits
									4 hex 3 hex 1 hex

4. (25 pts) To **begin** accessing information on a magnetic disk requires a period of time which is usually expressed in two additive components of disk access time known as average seek time and average rotational latency.

If the average seek time for a magnetic disk which spins at 10,000 RPM is 7 ms and for each track the disk averages 1000 sectors of 512 bytes per sector, what is the expected average time required to actually transfer a 512 byte sector from disk to memory?

$$T_{avg\ seek} = 7\ ms$$

$$T_{avg\ rotation} = \frac{1}{2} \times \frac{60\ sec}{10,000\ RPM} \times 1000\ ms/sec = 3\ (ms)$$

$$T_{avg\ transfer} = \frac{60\ sec}{10,000\ RPM} \times \frac{1}{1000\ sectors/track} \times 1000\ ms/sec = 0.006\ (ms)$$

$$T_{access} = 7 + 3 + 0.006$$

$$= 10.006\ (ms)$$

Powers of 2:

2 to the 0 : 1,  
2 to the 1 : 2  
2 to the 2 : 4  
2 to the 3 : 8  
2 to the 4 : 16  
2 to the 5 : 32  
2 to the 6 : 64  
2 to the 7 : 128  
2 to the 8 : 256  
2 to the 9 : 512  
2 to the 10 : 1024  
2 to the 11 : 2048  
2 to the 12 : 4096  
2 to the 13 : 8192  
2 to the 14 : 16384  
2 to the 15 : 32768  
2 to the 16 : 65536

5. (30 pts) Refer to the Mic1 microcode and specify ALL the lines of microinstructions after 0, 1, and 2 that are executed to do a STOL. Don't list lines 0, 1, and 2.

28, 29, 30, 33, 34, 10

- 4

6. (10 pts) In line 3 of the microcode, what is affecting the N flag?

- a) lshift      *only the result of ALU*  
 b) ir + ir  
 c) both

- 10

7. (40 pts) Suppose a new MACRO instruction is to be added to the set in the book (like you did in Assign 4). This new instruction is called **QUINTUPLE** and will take a value on the top of the stack and will multiply it by 5 (by adding it five times to itself, NOT actually multiplying) and put the new value back on the top of the stack. You should not modify the stack in any other way or modify AC. **Don't give up! This isn't that difficult.** Just get the value off the top of the stack, do the addition five times (you don't need to loop), and put it back. Use existing microcode for guidance for how to write the new microcode. You do **NOT** have to worry about **DECODING its OPCODE**, but will assume that your code begins at line 100 with the first MAL statement needed after the instruction has been successfully decoded by earlier parts of the microprogram and has jumped to your first MAL statement. You must make sure that when you finish putting the new number on the top of the stack, your microprogram segment will continue normal machine execution.

```

100: mar := sp; rd;
101: rd;
102: a := mbr;
103: c := mbr;
104: a := a + c;
105: a := a + c;
106: a := a + c;
107: a := a + c;
108: a := a + c;
109: a := sp;
110: mbr := a; wr;
111: goto 0; wr

```

-4

> 10

```

100: mar := sp; rd; {read value from memory}
101: rd;
102: a := mbr;
103: b := a;
104: b := a + b;
105: b := b + b;
106: b := a + b;
107: mbr := b; wr;
108: wr; goto 0;

```

- hidden bit
- shift mantissa of smaller value for common exponent
- Add & adjust mantissa

$$\begin{array}{r}
 01000 \quad 00000000000000000000000000000000 \\
 00111 \quad 00000000000000000000000000000000 \\
 \hline
 \text{→ shift right 5 bits}
 \end{array}$$

04.5

- hidden bit: 1000 1010 0000 0000 0000
- Shift 1010 0000 0000 0000 0000
- Sum 1000 1010 0000 0000 0000

$$\begin{array}{r}
 1000 1010 0000 0000 0000 \\
 1010 0000 0000 0000 0000 \\
 \hline
 0000 0101 0000 0000 0000
 \end{array}$$

100

Result: 000 1111 0000 0000 0000

$\Rightarrow$  Result: common exponent + mantissa  
correct sign (larger)

$$\begin{array}{r}
 01000 \quad 01000000000000000000000000000000 \\
 \hline
 \underline{01000} \quad \underline{01000}00000000000000000000000000
 \end{array}$$

- Calculate number:  $2^{\pm}$

- Find the hidden bit

$$2^2 + 2^7 = 128 + 128 - 128 = 2^7$$

128

- Hidden bit: implied bit left of the radix

1: almost all circumstances

0: denormal or Subnormal (if the exponent is all 0's)

173.7

$$173 = 1.35 \times 2^7$$

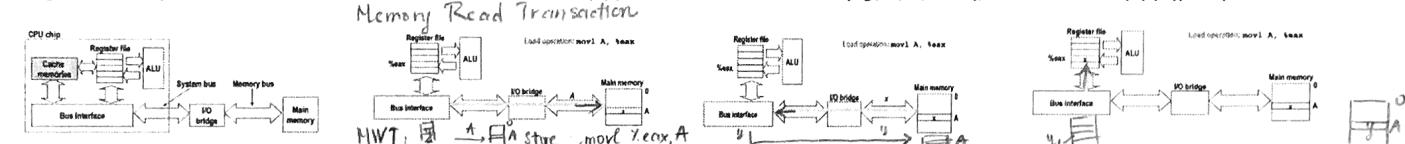
$$\rightarrow \text{Exponent: } 127 + 7 = 134 \rightarrow 1000 \ 0110$$

**1.DRAM cores with better interface logic and faster I/O :** ■ Synchronous DRAM (SDRAM) Uses a conventional clock signal instead of asynchronous control ■ Allows reuse of the row addresses (e.g., RAS, CAS, CAS, CAS). Double data-rate synchronous DRAM (DDR SDRAM). Double edge clocking sends two bits per cycle per pin. Different types distinguished by size of small prefetch buffer: DDR (2 bits), DDR2 (4 bits), DDR3 (8 bits), DDR4 (16 bits) ■ By 2010, standard for most server and desktop systems ■ Intel Core cpus (i3, i5, i7) support only DDR3, DDR4 SDRAM. **2.Nonvolatile Memories:** DRAM and SRAM are volatile memories ■ Lose information if powered off. Nonvolatile memories retain value even if powered off: ■ Read-only memory (ROM): programmed during production ■ Programmable ROM (PROM): can be programmed once ■ Eraseable PROM (EPROM): can be bulk erased (UV, X-Ray) ■ Electrically erasable PROM (EEPROM): electronic erase capability ■ Flash memory: EEPROMs with partial (block) erase capability : Wears out, erase life short depending on flash technology. – NAND SLC, MLC, TLC, QLC, etc. progressively shorter life **Uses for Nonvolatile Memories** ■ Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,...) ■ Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops,...) ■ Disk caches

**3.Traditional Bus Structure Connecting CPU and Memory:** A bus is a collection of parallel wires that carry address, data, and control signals. Buses are typically shared by multiple devices. **Memory Read Transaction:** (1)CPU places address A on the memory bus. (2)Main memory reads A from the memory bus, retrieves word x, and places it on the bus. (3)CPU read word x from the bus and copies it into register %eax. **Memory Write Transaction:** (1) CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive. (2) CPU places data word y on the bus. (3) Main memory reads data word y from the bus and stores it at address A. **4.Inside a Disk drive:** Actuator, Arm, Spindle, Platters, SCSI connector, Electronics (including a processor memory). **5.Disk Geometry:** Disks consist of platters, each with two surfaces. Each surface consists of concentric rings called tracks. Each track consists of sectors separated by gaps. **6.Disk capacity:** (1) Capacity: maximum number of bits that can be stored ■ Vendors express capacity in units of gigabytes (GB), where 1 GB =  $10^9$  Bytes (Lawsuit pending! Claims deceptive advertising). (2) Capacity is determined by these technology factors: ■ Recording density (bits/in): number of bits that can be squeezed into a 1 inch segment of a track. ■ Track density (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment. ■ Areal density (bits/in<sup>2</sup>): product of recording and track density. (3) Modern disks partition tracks into disjoint subsets called recording zones ■ Each track in a zone has the same number of sectors, determined by the circumference of innermost track. ■ Each zone has a different number of sectors/track. **COMPUTING DISK CAPACITY:** Capacity (GB) = (# bytes/sector) x (avg. # sectors/track) x (# tracks/surface) x (# surfaces/platter) x (# platters/disk). **7.Disk Access Time:** (1) Average time to access some target sector approximated by: ■ Taccess = Tavg seek + Tavg rotation + Tavg transfer. (2) Seek time (Tavg seek): ■ Time to position heads over cylinder containing target sector. ■ Typical Tavg seek is 3–9 ms (3) Rotational latency (Tavg rotation): ■ Time waiting for first bit of target sector to pass under r/w head. ■ Tavg rotation =  $1/2 \times 1/\text{RPMs} \times 60 \text{ sec}/1 \text{ min}$  ■ Typical Tavg rotation = 7200 RPMs. (4) Transfer time (Tavg transfer): ■ Time to read the bits in the target sector. ■ Tavg transfer =  $1/\text{RPM} \times 1/(\text{avg. # sectors/track}) \times 60 \text{ secs}/1 \text{ min}$ . **Disk Access Time Example** ' Given: ■ Rotational rate = 7,200 RPM ■ Average seek time = 9 ms. ■ Avg # sectors/track = 400. ' Derived: ■ Tavg rotation =  $1/2 \times (60 \text{ secs}/7200 \text{ RPM}) \times 1000 \text{ ms/sec} = 4 \text{ ms}$ . ■ Tavg transfer =  $60/7200 \text{ RPM} \times 1/400 \text{ sectors/track} \times 1000 \text{ ms/sec} = 0.02 \text{ ms}$  ■ Taccess = 9 ms + 4 ms + 0.02 ms. ' Important points: ■ Access time dominated by seek time and rotational latency. ■ First bit in a sector is the most expensive, the rest are free. ■ SRAM access time is about 4 ns/doubleword, DRAM about 60 ns ■ Disk is about 40,000 times slower than SRAM, ■ 2,500 times slower than DRAM. **8.Logical Disk Blocks:** (1) Modern disks present a simpler abstract view of the complex sector geometry: ■ The set of available sectors is modeled as a sequence of b-sized logical blocks (0, 1, 2, ...) (2) Mapping between logical blocks and actual (physical) sectors: ■ Maintained by hardware/firmware device called disk controller. ■ Converts requests for logical blocks into (surface,track,sector) triples. (3) Allows controller to set aside spare cylinders for each zone: ■ Accounts for the difference in "formatted capacity" and "maximum capacity". **9.Solid State Disk (SSDs):** Pages: 512KB to 4KB, Blocks: 32 to 128 pages. Data read/written in units of pages. Page can be written only after its block has been erased. A block wears out after 100,000 repeated writes. Why are random writes so slow? ■ Erasing a block is slow (around 1 ms) ■ Write to a page triggers a copy of all useful pages in the block: Find an used block (new block) and erase it. Write the page into the new block. Copy other pages from old block to the new block. **SSD Tradeoffs vs Rotating Disks** (1) Advantages ■ No moving parts → faster, less power, more rugged (2) Disadvantages: ■ Have the potential to wear out: Mitigated by "wear leveling logic" in flash translation layer ■ E.g. Intel X25 guarantees 1 petabyte (10<sup>15</sup> bytes) of random writes before they wear out ■ In 2010, about 100 times more expensive per byte (October 2013 about 75 times more expensive \$69.99 vs \$5,288.36 MLC -1TB) ■ Today in late 2015 about 7timesmore. (3) Applications ■ MP3 players, smart phones, laptops, desktops ■ Beginning to take over in servers and storage systems. **The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as locality.** **10.Locality:** (1) Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently (2) Temporal locality: ■ Recently referenced items are likely to be referenced again in the near future (3) Spatial locality: ■ Items with nearby addresses tend to be referenced close together in time. **Locality Example:** (1) Data references ■ Reference array elements in succession (stride=1 reference pattern) → Spatial locality. ■ Reference variable sum each iteration → Temporal locality (2) Instruction references ■ Reference instructions in sequence → Spatial locality ■ Cycle through loop repeatedly → Temporal locality. **11.Memory Hierarchies:** (1) Some fundamental and enduring properties of hardware and software: ■ Fast storage technologies cost more per byte, have less capacity, and require more power (heat). ■ The gap between CPU and main memory speed is widening. ■ Well-written programs tend to exhibit good locality. (2) These fundamental properties complement each other beautifully. (3) They suggest an approach for organizing memory and storage systems known as a memory hierarchy.

**12.CACHES:** (1) Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device. (2) Fundamental idea of a memory hierarchy: ■ For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1. (3) Why do memory hierarchies work? ■ Because of locality, programs tend to access the data at level k more often than they access the data at level k+1. ■ Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit. (4) Big Idea: The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top. **13.General Caching Concepts:**

**Types of Cache Misses:** (1) Cold miss (compulsory) ■ Cold misses occur because the cache is empty. (2) Conflict miss ■ Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k. E.g. Block i at level k+1 must be placed in block (i mod 4) at level k. ■ Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block. E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time. (3) Capacity miss ■ Occurs when the set of active cache blocks (working set) is larger than the cache. **14.Cache Memories:** (1) Cache memories are small, fast SRAM-based memories managed automatically in hardware. ■ Hold frequently accessed blocks of main memory. (2) CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory. (3) Typical system structure:



**15.What about writes?** (1) Multiple copies of data exist: ■ L1, L2, Main Memory, Disk (2) What to do on a write-hit? ■ Write-through (write immediately to memory) ■ Write-back (defer write to memory until replacement of line): Need a dirty bit (line different from memory or not) (3) What to do on a write-miss? ■ Write-allocate (load into cache, update line in cache). Good if more writes to the location follow ■ No-write-allocate (writes immediately to memory) (4) Typical: Write-through + No-write-allocate ■ Write-back + Write-allocate. **16.Cache Performance Metrics** (1) Miss Rate: ■ Fraction of memory references not found in cache (misses / accesses) = 1 – hit rate ■ Typical numbers (in percentages): 3-10% for L1. Can be quite small (e.g., < 1%) for L2, depending on size, etc. (2) Hit Time: ■ Time to deliver a line in the cache to the processor. Includes time to determine whether the line is in the cache ■ Typical numbers: 1-2 clock cycle for L1, 5-20 clock cycles for L2. (3) Miss Penalty: ■ Additional time required because of a miss. Typically 50-200 cycles for main memory (Trend: increasing!). **Example:** Consider: cache hit time of 1 cycle miss penalty of 100 cycles.. **Average access time:** 97% hits: 1 cycle + 0.03 \* 100 cycles = 4 cycles 99% hits: 1 cycle + 0.01 \* 100 cycles = 2 cycles → This is why "miss rate" is used instead of "hit rate". **17.Writing Cache Friendly Code** (1) Make the common case go fast ■ Focus on the inner loops of the core functions (2) Minimize the misses in the inner loops: ■ Repeated references to variables are good (temporal locality) ■ Stride-1 reference patterns are good (spatial locality). **Key Idea: Our qualitative notion of locality is quantified through our understanding of cache memories.** **18.Concluding Observations:** (1) Programmer can optimize for cache performance ■ How data structures are organized ■ How data are accessed: Nested loop structure. Blocking is a general technique (2) All systems favor "cache friendly code": ■ Getting absolute optimum performance is very platform specific: Cache sizes, line sizes, associativities, etc. ■ Can get most of the advantage with generic code: Keep working set reasonably small (temporal locality). Use small strides (spatial locality).

## Threads

the executable (schedulable) elements in a Linux system

Each thread in the system is uniquely contained by some PID

Each user thread is contained by some user PID

Each kernel thread is contained by PID 0

When a new process is created, it is populated by exactly one executable thread, known as the *Initial Thread* (IT) of the new process

The IT of a process can create new threads only within its own process

While the IT must create the second thread in a process, any subsequent threads can then create new threads, but only within their own process

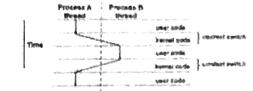
## Context Switching

Processes are managed by a shared chunk of OS code called *the kernel*

Control flow passes from one thread in a process to another via context switches

Control flow passes from one thread in a process to a different process via a context switch

Time



## fork: Creating new processes

### int fork(void)

- \* creates a new process (child process) that is identical to the calling process (parent process)
- \* returns 0 to the child process
- \* returns child's pid to the parent process

```
if (fork() == 0) {
    /*Hello from child\n*/
} else {
    printf("Hello from parent\n");
}
```

Fork is interesting (and often confusing) because it is called once but returns twice

### wait: Synchronizing with children

#### int wait(int \*status)

- \* waits a process
- \* Normally returns 0 if status is 0
- \* waitpid() registers function to be executed upon exit

#### zombies

##### Idea

- \* When process becomes still consumes system resources
- \* Various tasks maintained by OS

Called a "zombie"

Living, but alive but dead

Reaper

Performed by parent or terminated child

Parent is given exit status information

Kernel destroys process

What If Parent Doesn't Reap?

\* Any parent can reap its child, then child will be reaped by next process

\* Only need to reap for long-running processes

\* E.g., sockets and servers

##### waitpid

##### exit

##### destroying process

##### zombie

##### reaper

##### what if parent dies?

##### parent dies

