1/ Convert base 10 into base 2, base 8, base 16

- Base 2: 119.78125 → ÷2 ↑ → ×2 ↓ → 01110111.11001

- Base 8: 2^3 => combine 3 numbers

01 110 111 . 110 010 00

1 6 7 . 6 2

- Base 16: 2^4 => combine 4 numbers

0111 0111 . 1100 1000

7 7 . C 8

Negative number: -328

328 → 0000 | 0001 | 0100 | 1000

¬ 1111 | 1110 | 1011 | 0111

+ 1

-328 → 1111 | 1110 | 1010 | 1000

2/ Convert 2's Complement into base 10

-1 :	1111	+7 :	0111
-2 :	1110	+6 :	0110
-3 :	1101	+5 :	0101
-4 :	0100	+4 :	0100
-5 :	0011	+3 :	0011
-6 :	0010	+2 :	0010
-7 :	0001	+1 :	0001
-8 :	1000	+0 :	0000

* Negative:

- Notice the sign : 1 or 0 — 11011011

- Change (flip) the sign to get the magnitude of the number

11011011

¬ 00100100

+ 1

00100101

- Convert the magnitude to decimal:

$00100101_2 = 25_{16} = 2 \times 16 + 5 = 37$

2 5

=> Negative : -37

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

3/ Convert * into floating number (base 10)

* IBM format:

1 | 7 | 24 bits

Sign | Exponent (base 16 excess 64) | Mantissa (No hidden bit, $0 \le$ mant < 1)

0 | 100 0011 | 0101 1100 0000 0000 0000 0000

= 67 - 64

= 3 => 16^3

$= 2^{12} \times (2^{-2} + 2^{-4} + 2^{-5} + 2^{-6})$

$= 2^{10} + 2^8 + 2^7 + 2^6 = 1472.0$

* IEEE 754 single precision

1 | 8 | 23 bits

Sign | Exponent (base 2 excess 127) | Mantissa — Hidden bit = 1, = 0 (if exponent is all 0s)

0 | 1000 0110 | 101 1100 0000 0000 0000 0000

= 134 - 127

= 7 => 2^7

$= 2^7 \times (\boxed{1} + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5})$

$= 2^7 + 2^6 + 2^4 + 2^3 + 2^2 = 220.0$

Ex: Exponent: 0000 0000 : 2^{-126}

1 1111 1111 . 0110 = ... 0000 = NAN

1 1111 1111 . 0000 = ... 0000 => $-\infty$

4/ Convert floating number into IEEE 754 single precision

.8125 (10) × 2 = 1.6250 ↓ → .1101 → 1.F

Move the decimal point: to the left: +, to the right: −

Move 1 => 127 - 1 = 126

Exponent : 0111 1110

=> 0 | 0111 1110 | 101 0000 0000 0000 0000

5/ Add 2 IEEE 754 single precision

- Float 1 : 0 | 1000 0011 | 101 1100 0000 0000 0000 0000 => 13.75

- Float 2 : 0 | 0111 1110 | 101 0000 0000 0000 0000 0000 => 0.8125

Exponent: Float 1 $2^3 + 2^1 = 128 + 2 = 130 - 127 = 3$ shift 4 bits

Float 2: 126 - 127 = -1 shift mantissa 2

Mantissa: Float 1: 1011 1001 0000 0000 0000 0000

Float 2: 0001 1010 0000 0000 0000 0000

1101 0011 0000 0000 0000 0000

=> Result: 0 | 1000 0010 | 1101 0011 0000 0000 0000 0000

6/ 01100101.0011 IEEE 754 shift 6 places, +6

IBM shift 8 places, +2

IEEE754: 127+6 = 133 — 1000 0101 | 1100 1010 0111 0000...

IBM: 64+2 = 66 — 100 0010 | 0110 0101 0011 0000.

7/ Most current machines use 32 bits words. Limit addresses to 4G

High-end systems use 64 bits words. Address space = 1.8×10^{19} bytes

8/ Byte Ordering: Conventions

- Big Endian: Sun Spare (bi), older IBM, Mac (bi), Internet, JTEG

Least significant byte has highest (numerically largest) address

- Little Endian: x86, x86-64, ARM (bi), PCI and USB buses, BMP

Least significant byte has lowest (numerically smallest) address

	0x100	0x101	0x102	0x103
Big E.	01	23	45	67
Little E.	67	45	23	01

Ex: $15213_{10} = 3B6D_{16}$ — IA32, x86-64 Sun Spare

int: 4 bits (6D, 3B, 00, 00) → (00, 00, 3B, 6D)

(long int: 8 bits, 8 squares)

- String: char S[6] = "18243". Character "0" has code 0x30

Digit i has code 0x30 + i. Final character = 0

Byte ordering is not an issue

Smallest address	x86, x86-64	Sun spare
↓	31	31
	38	38
	32	32
	34	34
	33	33
	00	00

- Pointer:

	Sun spare	IA32	x86-64
MSB	EF	D4 LSB	0C LS
	FF	F8	89
	FB	FF	EC
	2C	BF	:
	0xEFFFFB2C	0xBFFFF8D4	0x00007F

- Unsigned values: UMin = 0, UMax = $2^w - 1$

Two's Complement TMin = -2^{w-1}, TMax = $2^{w-1} - 1$

- Sign Extension: Make k copies of "sign bit"

- IEEE754

Least positive/N Normalized: 0/1 0000 0001 0000...0000 = $\pm 2^{-126} = \pm 10^{-38}$

Most : 0/1 1111 1110 111... 111 = $\pm 2^{128} = \pm 10^{38}$

Least DeN : 0/1 0000 0000 000...001 = $\pm 2^{-149} = \pm 10^{-4}$

zero : 0 0000 0000 000...000 = 0

Pos/Neg Infinity : 0/1 1111 1111 000...000 = $\pm\infty$

NAN SP+3 0/1 1111 1111 ≠0 = NAN

SP+2 NULL

SP+1 SEG

SP

50 + 52 + 50

1/
```
max: LODL  1   ; OI 2
     SUBL  2   ; OI 2 - OI 1
     JNEG  op1big:
     LODL  1
     RETN      ; OI 2
op1big: LODL 2
     RETN      ; OI 1
```

SP → RETN, SP+1 op2, SP+2 op1

```
op1:
op2:
opres: 0
     .LOC 50
main: LODD op1:
     PUSH
     LODD op2:
     PUSH
     CALL max:
     INSP 2
     STOD opres:
     HALT
```

2/
```
main: LODD index
```

Ex,
```
     DESI  2   ; sp := sp - 2
     LOCO  0   ; ac := 0
     STOL  0   ; m[sp+0] := ac  (sum := 0)
     STOL  1   ; m[sp+1] := ac  (i := 0)
5:   SUBL  2   ; ac := ac - m[sp+3]  (i - N)
     JZER  15  ; jump if (i == N)
     JPOS  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)
     STOL  1   ; m[sp+1] := ac  (i=1+i)
     JUMP  5   ;
15:  LODL  0   ; ac := m[sp+0]  (sum)
     INSP  2   ; sp := sp + 2  move stack pointer down   (2 places)
     STOL  1   ; m[sp+1] := sum  (return sum)
     RETN      ; return to caller
```

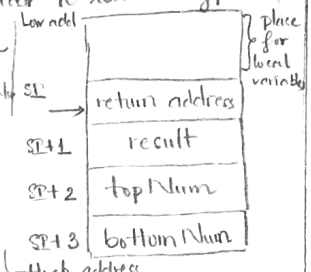
sum

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

4/
```
adder: LODL  1   ; get 1st arg from stack into ac (data v)
     STOD myent:  ; store count at location myent:
     LODL  2   ; get 2nd arg from stack into ac (data addr)
     PSHI      ; push indirect 1st datum to stack
     ADDD myc1:; add 1 (value at myc1:) to addr in AC
     STOD myptr:; store new addr to location myptr:
```

5/
```
MySub: LODL  2   ; ac := topNum
     SUBL  3   ; ac := ac - bottomNum
     PUSH
     JNEG  neg:
     LODL  2   ; ac := result
     PUPI      ; m[ac] = m[sp]
     LOCO  0   ; ac := 0
     RETN
neg: LODL  2
     PUPI
     LODD cn1: ; ac := m(x)
     RETN
cn1: -1
```

x : address of cn1 and value = -1



sp is at local variable (value of subtraction) gain this subtraction is to result

6/ Recursion
```
LOOP: LODD TasCnt    ; num of fibs to do in TasCnt
     JZER DONE:    ; no more passes, go to done
     SUBD c1
     STOD TasCnt   ; -- passes remaining
P1:  LODD daddr:   ; load a pointer to fib arg
     PSHI          ; push arg for fib on stack
     ADDD c1:
     STOD daddr:   ; inc, store pointer for next d[n]
     CALL FIB:     ; call fib(arg on stack)
     INSP 1        ; clear stack on fib return
P2:  PUSH          ; put return AC (fib(n)) on stack
     LODD faddr:   ; load a pointer to result f[n]
     POPI          ; pop result off stack into f[n]
     ADDD c1:
     STOD faddr:   ; inc, store pointer for next f[n]
     JUMP LOOP:    ; go to top for next pass
FIB: LODL 1        ; fib func loads arg from stack
     JZER FIBZER:  ; if fib(0) go to FIBZER
     SUBD c1:      ; dec arg value in AC (arg -1)
     JZER FIBONE:  ; if fib(1) go to FIBONE
```

Iterative
```
     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 goto 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-1)
     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 tmp:     ; load AC with tmp: final result
     RETN
```

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

str: 0110 1001, 0110 1000 | 0110 1101, 0110 1111
i h m o
0110 1101, 0010 0000 | 0000 0000, 0110 0001
h space \0 a

2 characters for every 16 bits word

* STOD 4093 RCVR
STOD 4095 XMTR } => Turn "ON" the RCVR and XMTR

- Receiver will accept characters from a terminal device only if RCVR is on by setting the 0 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
+ 0 is cleared: device is turned off, clear all other bits
- Transmitter: XMTR
+ 0 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

"7" → -48 → 7 (number) × 10 = 70

"9" → -48 → 9 (+9) → 79 ×10 = 790

"2" → -48 → 2 (+2) → 792

"4"
"9"
"2"
"5"

Ex = 8279 + 15 = 8294/10 = remainder 4 + 48 = "4"

stash: LIFO

* 4093 OIDB: 0 is on → 1000 = 8 => on: 8

1010 = 10 → Busy bit is off => Done & not busy → do sth more
↓ Done bit is on

× 1001 = 9 → Not done & busy → need to wait, pause more

* ISHI ; add 2 characters at once
the stash: "T+H"
"+H" "T"
subd c255 => to check if it terminates the string yet
jneq crnl: => create new line
call sb: ⇒ swap 2 characters → "T" "+H"
(sb: loco 8)
⋮
pop → pop "T+H"
⋮
lodd pstr1 → now point to "I"
top: lodd cnt: ⇒ cnt: 30 → 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, radiate) than SRAM. Slower & cheaper than SRAM

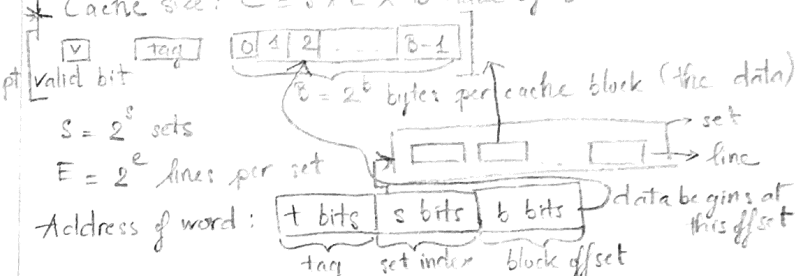
	Trans per bit	Access time	Needs refresh?	Needs EDC?	Cost	Application
SRAM	4 or 6	1×	No	Maybe	100×	Cache memories
DRAM	1	10×	Yes	Yes	1×	Main memory, frame buffers

(EDC, ECC = error detecting codes, error correcting codes)

⇒ SRAM faster than DRAM: All transistors, no capacitors to refresh
⇒ DRAM less expensive: One transistor & a capacitor cost much less than 6 or 7 transistors
⇒ Temporal & spatial locality (loops, functions, calls)

* Conventional DRAM organization:
- d×w total bits organized as d supercells of size w bits
Ex. 16 × 8 DRAM chip: (16 supercells, 4×4)
- RAS # CAS (same pin)

* Cache size: C = S × E × B data bytes



Ex, 64 bytes = 2^6 ⇒ block offset = 6 bits
# of sets = 2^{14} ⇒ s bits = 14 bits

Ex. Block size = 2^4 = 16
⇒ # of sets = 2^{12} = 4K
Address: 7A43

valid	tag	
0	0007	Set #43

Direct Mapped: ×

Tag	Set #	Block offset
16 bits	12 bits	4 bits
4 hex	3 hex	1 hex → 2^4

* 0x004C71F5 | 8192 sets of a single line of 16 bytes → 2^{13}

0000 0000 0 010 0 110 | 0 0111 0001 1111 | 0101
Tag = 0x26 | 13 bit lines = 1823 (071F set/line number) | Byte offset

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

0000 0000 0 100 1 | 000 0 | 111 0001 1111 | 0101
Tag = 0x98 | 11 bit lines = 1823 | Byte offset

4 lines searched for this tag in set 1823
Direct Mapped: 1 line for every set

Hit: set → Tag
Miss: Tag ≠

Tavg: $\frac{60\text{sec}}{\text{RPM}} \times 1000\text{ms/sec} \times \left(\frac{1}{2}\text{ : rotation}\right) + \left(\frac{1}{\text{sectors}}\text{ : transfer}\right)$

Taccess = seek + rotation + transfer

```
0  lodcl  start:  --> S : 1000 _ turn on  0 bit
1  stocl  4093
2  stocl  4095
3tpr: lodd  4095
4     subd  mask:   } wait for XMTR ready
5     jzer  print:  }
6     jump  tpr:    }
7print: lodd  char:  --> "Z"
8     stod  4094
9     subd  c2:    --> AC = "X"  ("Z" - 2 = "X")
10    stod  char:  --> char = "X"
11    subd  ch1:   --> AC = "X" - "A" > 0
12    jpos  cont:
13    lodd  ch2:   --> < 0, char = "Z" again
14    stod  char:
15 cont: lodd  4093  } 10: done
16    subd  mask:   } 9: busy
17    jzer  done:   } check to see if it receives anything
18    jump  tpr:    }
19 done: halt      --> done _, halt command
20 start: 8              when RCVR is done --> when some keyboard input is
21 mask: 10              available
22 ch1: "A"        (if RCVR is busy --> not done yet)
23 ch2: "Z"
24 char: "Z"
25 c2: 2           --> the character is at index 2 from ["Z"] => ["X"]
```

Output: ZXVT-BZXVT-B....

"X": AC = "V" ("X" - 2 = "V"); char = "V"; AC = "V" - "A" > 0

"V": "T"; "T": "R"; "R": "P"

"B": AC = @ (B - 2); char = @; AC = "@" - "A" < 0

"@": AC = "B"; char = "B"; AC = "B" - "A"

```
xbsywt: lodd  4095          xbsywt: lodd  4095   } check if it's
        subd  xdmask:               subd  xdmask: } safe to place
        jzer  xrdy:    =>           jneg  xbsywt: } character in
        jump  xbsywt                xrdy : retn   } the XMTR.
xrdy:   retn                        xdmask: 10    }
xdmask: 10
```

```
sb:   lodl  1   --> memory address          [sb1] : BYTE SWAP
      pshi
      loco  8   --> byte swap on the content of the location
loop: jzer  finish:
      subd  c1:  --> (8-1) [addl 1]: add the missing code segments to
      stod  lpcnt: --> lpcnt = 7 each location
      lodl  0   --> top of the stack, now is "CS"
      jneg  add1: --> to know if we should copy 0 or 1 --> negative
      addl  0   --> add the top stack (CS),
      stol  0   --> means add the value in the top of stack shift to the left by 1
      lodd  lpcnt:
      jump  loop:
add1: addl  0
      addd  c1.
      stol  0
      lodd  lpcnt:
      jump  loop:
finish: lodl  2
      popi
      retn        ; procedure ends here
c1:   1           ; data location for the procedure
lpcnt: 0
```

lodl 1 → Memory loc (top + 1); retn addr; "CS" (top)

"CS" _, store in 16 bits word

0101 0011 | 0100 0011

loco 8 ⇐ "S" = 83 | "C" = 67 (ASCII)

=> swap => CS

AC = XXXX 0101 0011 | 0100 0011 01 ← addl 0 : shift left

8 times

BSWAPD

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

MULT

```
82: a:= lshift(1);        93: d:= 0
83: a:= lshift(a+1);      94: alu:= a; if n then goto 97;
84: a:= lshift(a+1);      95: c:= c+1; goto 94;
85: a:= lshift(a+1);      96: alu:= c; if z then 100
86: a:= lshift(a+1);      97: b:= b1 (=1);
87: a:= a+1;                  if n then goto 105;
88: b:= band(ir,a);       98: d:= d+a; ---- 96;
(build 6 bit mask + put   99: alu:= c; ---- 99;
multiplier in b)          100: ac:= (-1); goto 0;
89: mar:= sp ;rd;         101: mar:= sp ; ac:= 0;
90: rd;                   102: mbr:= d; wr;
91: a:= mbr;                   goto 101;
92: c:= (-1);
```

RShift:

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

1/ Convert base 10 into base 2, base 8, and 16

- Base 2: 119.7125 ↻/2 ↻ ×2 ↓ =>
- Base 8: 2^3 => combine 3 numbers 01 | 110 | 111 . 110 | 000 (1 | 6 | 7 . 6 | 2)
- Base 16: 2^4 => combine 4 numbers 0111 | 0111 . 1100 | 1000 (7 | 7 . C | 8)
- Negative number -- 328: 328 → 0000 | 0001 | 0100 | 1000; 1's: 1111 | 1110 | 1011 | 0111; +1: 1111 | 1110 | 1011 | 1000 → -328

2/ Convert 2's Complement into base 10

* Negative.
- Notice the sign 1 or 0
- Change (flip) the signs to get the magnitude of the number

1101 1011 → 0010 0100 → +1 → 0010 0101

- Convert the magnitude to decimal:

0010 0101 $_2$ = 25_{16} = 2×16 + 5 = 37

(2, 5)

→ Negative: -37

* Positive: $0110\,1001_2 = 69_{16} = 6 \times 16 + 9 = 105_{10}$

3/ Convert ★ into floating number (base 10)

* IBM format:

24 bits / No hidden bit

1	7	Mantissa (0 ≤ mant < 1)
Sign	Exponent (base 16 excess 64)	
0	100 0011	0101 1100 0000 0000 0000 0000
	= 67 - 64	①②⑤④
	= 3 => 16^3	
	= $2^{12} \times (2^{-2} + 2^{-4} + 2^{-5} + 2^{-6})$	
	= $2^{10} + 2^8 + 2^7 + 2^6$ = 1472.0	

* IEEE 754 single precision.

1	8	23 bits
	(base 2 excess 127)	Hidden bit = 1, = 0 (if exponent is all 0's)
0	1000 0110	101 1100 0000 0000 0000 0000
	= 134 - 127	①②③
	= 7 => 2^7	
	= $2^7 \times (1 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5})$	
	= $2^7 + 2^6 + 2^4 + 2^3 + 2^2$ = 220.0	

4/ Convert floating number into IEEE 754 single precision

$.8125_{10}$ × 2 ↓ = 1.6250

.1101 ↻ => 1.F

Move the decimal point to the left: +; to the right: -

=> Move 1 => 127 - 1 = 126

Exponent: 0111 1110

=> 0 | 0111 1110 | 101 0000 0000 0000 0000

5/ Add 2 IEEE 754 single precision

- Float 1: 0 | 1000 0010 | 101 1100 0000 0000 0000 0000 => 13.75
- Float 2: 0 | 0111 1110 | 101 0000 0000 0000 0000 => 0.8125

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

Mantissa: ① 1101 101 1100 0000 0000 0000; ② +1101 0000 0000 0000 0000 } Add → 1110 1001 0000

6/ 0111 0010 1 . 0001 1100 0000 0000

IEEE754 shift 6, +6

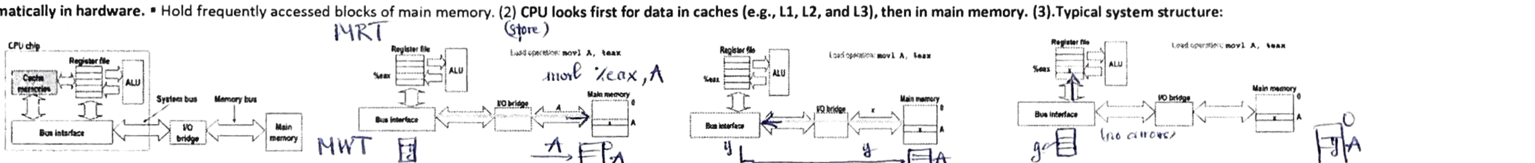
IBM shift 8, +2

IEEE: 127 + 6 = 133 => 1000 0101 | 1001 1100 0000 0000

IBM: 64 + 2 = 66 => 1000 0010 | 0001 1001 1100 0000

Result: 0 | 1000 0010 | 1001 1100 0000 0000

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 eraseable 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), where1 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/in2): 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 x 1/RPMs x 60 sec/1 min ▪ Typical Tavg rotation = 7200 RPMs. (4) **Transfer time (Tavg transfer):** ▪ Time to read the bits in the target sector. ▪ Tavg transfer = 1/RPM x 1/(avg # sectors/track) x 60 secs/1 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 x (60 secs/7200 RPM) x 1000 ms/sec = 4 ms. ▪ Tavg transfer = 60/7200 RPM x 1/400 sects/track x 1000 ms/sec = 0.02 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 then 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 (10^15 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 about7timesmore. (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 CacheMisses:**(1)Cold (compulsory) miss ▪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).

1000 (0IDB)

```
start:  lodd on:
        stod 4095
        call xbsywt:
        loco str1:              ; print out the first string
        call nextw:
        call bgndig:
        lodd binnum:            ; put first number into ac
        stod total:             ; store the first number into total
        loco str1:              ; print out the first string to get the second number
        call nextw:
        call bgndig:
        lodd binnum:            ; put second number into ac
        addd total:             ; add two numbers
        stod total:
        stod addRst:            ; Store answer in addRst
        jneg over:              ; If negative we got an overflow
        call xbsywt:
        loco str2:              ; if not overflow, print out the second string
        call nextw:
        call xbsywt:
        call result:            ; call the result
        call xbsywt:
        lodd zero:              ; put zero into AC
        halt
over:   loco str3:              ; print out the third string for overflow result
        call nextw:
        lodd num:               ; put num(-1) into AC
        halt
```

write str. asm

```
nextw:  pshi
        addd c1:
        stod pstr1:
        pop
        jzer crnl:
        stod 4094
        push
        subd c255:
        jneg crnl:
        call sb:
        insp 1
        push
        call xbsywt:
        pop
        stod 4094
        call xbsywt:
        lodd pstr1:
        jump nextw:
crnl:   lodd cr:
        stod 4094
        call xbsywt:
        lodd nl:
        stod 4094
        call xbsywt:
        retn
```

readint. asm

```
bgndig: lodd on:
        stod 4093
        call rbsywt:
        lodd 4092
        subd numoff:
        push
nxtdig: call rbsywt:
        lodd 4092
        stod nxtchr:
        subd nl:
        jzer endnum:
        mult 10
        lodd nxtchr:
        subd numoff:
        addl 0
        stol 0
        jump nxtdig:
```

```
                        ; micl program to print string
                        ; and scan in a 1-5 digit number
                        ; using CSR memory locations

                        ; substract the numoff(48)
```

```
sum:    lodd addRst:            ; put addRst into ac
        jzer output:            ; call output if zero
        lodd mask:
        push
        lodd addRst:            ; put addRst into ac
        push
        div                     ; divide addRst by mask(10)
        pop
        stod addRst:            ; store into addRst
        pop                     ; pop the result
        insp 2                  ; clear up stack
        addd numoff:            ; add numoff(48)
        push
        jump sum:
output: pop
        jneg done:              ; jump to done if negative
        stod 4094
        call xbsywt:
        jump output:
done:   retn                    ; Return
numoff: 48
nxtchr: 0
binnum: 0
lpcnt:  0
mask:   10
on:     8
nl:     10
cr:     13
c1:     1
c255:   255
pstr1:  0
num:    -1
total:  0
addRst: 0
zero:   0
str1:   "Please enter an integer between 1 and 32767."
str2:   "The sum of these integers is: "
str3:   "Overflow no sum possible"
```

writeint. asm

```
endnum: pop
        stod binnum:
        lodd 4092
        loco 0
        retn
xbsywt: lodd 4095
        subd mask:
        jneg xbsywt:
        retn
rbsywt: lodd 4093
        subd mask:
        jneg rbsywt:
        retn
```

addints. asm

```
sb:     loco 8
loop1:  jzer finish:
        subd c1:
        stod lpcnt:
        lodl 1
        jneg add1:
        addl 1
        stol 1
        lodd lpcnt:
        jump loop1:
add1:   addl 1
        addd c1:
        stol 1
        lodd lpcnt:
        jump loop1:
finish: lodl 1
        retn
result: lodd on:
        stod 4095
        lodd num:               ; put num(-1) into ac
        push
        lodd mask:              ; put mask(10) into ac
        push
```

main. asm:
start:
over:
finish:
result:
sum:
output

```
start:  lodd on:
        stod 4095
        call xbsywt:
        loco str1:
nextw:  pshi
        addd c1:
        stod pstr1:
        pop
        jzer crnl:

        push
        subd c255:
        jneg crnl:
        call sb:
        insp 1
        push
        call xbsywt:
        pop
        stod 4094
        call xbsywt:

crnl:   lodd cr:
        stod 4094
        call xbsywt:
        lodd nl:
        stod 4094
        call xbsywt:
        lodd on:
        stod 4093
top:    lodd cnt:
        subd c1:
        stod cnt:
        jzer over:

        lodd 4092
        stod 4094
        call xbsywt:
        jump top:
over:   lodd cr:
        stod 4094
        call xbsywt:
        lodd nl:
        stod 4094
        halt
xbsywt: lodd 4095

        jneg xbsywt:
        retn
rbsywt: lodd 4093
        subd mask:
        jneg rbsywt:
        retn
```

Data location:

lpcnt: 0	cnt: 30
mask: 10	c1: 1
on: 8	c255: 25[illegible]
nl: 10	pstr1: 0

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

#define HEADERS 1
#define NO_HEADERS 0

typedef struct nament {
  char name[26];
  int addr;
  struct nament *next;
} SYMTABENTRY;

void add_symbol( char *symbol, int line_number );
void generate_code( int );
void print_first_pass( int );
void append_table( void );
void dump_table( void );
int get_sym_val( char *);

SYMTABENTRY *symtab = NULL;
FILE *p1, *p2;
char cstr_12[13];

int main( int argc, char* argv[]){
  int i, start, pc_offset = 0, pc = 0;
  int linum = 0, object_file = 0, dump_tab = 0;
  int line_number, new_pc;
  char instruction[18];
  char symbol[26];
  if(argc > 1 && (strcmp(argv[1], "-s") == 0)){
    dump_tab = linum = 1;
  } else if( argc > 1 && (strcmp(argv[1], "-o") == 0 )){
    object_file = 1;
  }
  if((dump_tab == 1) | (object_file == 1)){
    start = 2;
  } else {
    start = 1;
  }
   } else {
    rewind(p1);
    while(fgets(inbuf, 80, p1) != NULL){
      printf(" %s", inbuf);
    }
  }
}
void add_symbol( char *symbol, int line_number ){
  SYMTABENTRY *_sym;
  _sym = (SYMTABENTRY *) malloc (sizeof (SYMTABENTRY));
  if(symtab != NULL){
    _sym -> next = symtab;
    symtab = _sym;
  } else {
    symtab = _sym;
    _sym -> next = NULL;
  }
  strcpy(_sym -> name, symbol);
  _sym -> addr = line_number;
  return;
}
void append_table(){
  while(symtab != NULL){
    printf("%s \t\t\t%d\n", symtab -> name, symtab -> addr);
    symtab = symtab -> next;
  }
  return;
}
int get_sym_val(char *symbol){
  SYMTABENTRY *rec = symtab;
  int cmp;
  while(rec != NULL){
    cmp = strcmp(rec -> name, symbol);
    if(cmp != 0){
      rec = rec -> next;
    } else {
      return rec -> addr;
    }
  }
  return -1;
}
```

```c
  p1 = fopen("/tmp/daedalus", "w+");
  unlink("/tmp/daedalus");
  for(i = start; i < argc; i++){
    if((p2 = fopen(argv[i], "r")) == NULL){
      printf("Cannot open file named: %s", argv[i]);
      exit(2);
    }
    while(fscanf(p2, "%d %s", &pc, instruction) != EOF){
      if(pc == 4096) break;
      new_pc = pc + pc_offset;
      symbol[0] = '\0';
      if(instruction[0] == 'U'){
    fscanf(p2, "%s", symbol);
      }

      fprintf(p1, " %d %s %s \n", new_pc, instruction, symbol);
    }
    while(fscanf(p2, "%s %d", symbol, &line_number) != EOF){
      add_symbol(symbol, line_number + pc_offset);
    }
    pc_offset = new_pc + 1;
    fclose(p2);
  }
  if(object_file){
    print_first_pass(NO_HEADERS);
    printf("4096 x\n");
    append_table();
    return 0;
  }
  if(linum){
    print_first_pass(HEADERS);
  }
  generate_code(linum);
  return 0;

}
void generate_code(int linum){
  char instruction [18];
  char symbol[26];
  int i, j, pc, difference, mask, _symbol;
  int old_pc = 0;
  rewind(p1);
  while(fscanf(p1, "%d %s", &pc, instruction) != EOF){
    if((difference = pc - old_pc) > 1 ){
      for(j = 1; j < difference; j++){
    printf("1111111111111111 \n");
      }
    }
    old_pc = pc;
    if(instruction[0] != 'U'){
      printf("%s\n", instruction);
    } else {
      fscanf(p1, "%s", symbol);
      _symbol = get_sym_val(symbol);
      if(_symbol == -1){
        exit(3);
      }
      for(i = 0; i<12; i++){
        cstr_12[i] = '0';
      }
      cstr_12[12] = '\0';
      mask = 2048;
      for(i = 0; i < 12; i++){
        if(_symbol & mask){
          cstr_12[i] = '1';
        }
        mask >>= 1;
      }
      for(i = 0; i < 12; i++){
        instruction[i + 5] = cstr_12[i];
      }
      printf("%s\n", &instruction[1]);
    }
  }
  return;
}

void print_first_pass(int headers){
  char inbuf[81];
  if(headers != HEADERS){
    rewind(p1);
    while(fgets(inbuf, 80, p1) != NULL){
      printf(" %s", inbuf);
    }
```

0x100 | 0x101 | 0x102 | 0x103
01 | 23 | 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: Danghi 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:
7          stod 4094
8          addd c4:
9          stod char:
10         subd chlim:
11         jneg cont:
12         lodd chB:
13         stod char:
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" E "B" "F" "B" "F"
"F" "J"
AC = "E" I "F" "J"
char = "E" I
AC = "E" - "G" <0 >0 <0 >0
"B" "B"
char = "B"

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?

0

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

AEBFBFBFBF

0 9 5 C 7 6 9 4

0000 1001 0101 1100 0111 0110 1001 0100

Set #

Tag = 12

2^8 2^{15}

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

Tag 0x12

Set # 0x5C76

2^4 offset = 4 2^{12} 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.)

Tag | Set the bits of the index | Block offset

E 8 7 9 F 0

Set 79F Tag E8 E879F0 Miss ✓ 1110 1000 | 0111 1001 1111 | 0000

79F E8 E879F4 Hit ✓ E8 0100

79F A4 A479F8 Miss ✓ A4

79F E8 E879F8 ~~Hit~~ Miss (conflict) E8

79F 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 ?

$T_{avg\ seek} = 7ms$

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

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

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

$= 10.006\ (ms)$

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
b) ir + ir → only the result of ALU
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 mantisse of smaller value for common exponent
- Add & adjust mantissa

0 1000 0100 | 0001 1010 0000 0000 0000 0000 → 24.5
0 0111 1111 | 0100 0000 0000 0000 0000 0000 → 1.25

→ shift right 5 bits

- Hidden bit: 1000 1010 0000 0000 0000 0000
  1 0100 0000 0000 0000 0000 0000
- Shift: 1000 1010 0000 0000 0000 0000
  0000 0101 0000 0000 0000 0000
- Sum: 1000 1111 0000 0000 0000 0000
  Rounc 0001 1111

⇒ Result: common exponent + mantissa
correct sign (larger)
0 1000 0100 0001 1111 0000 0000 0000 0000

- Calculate number: $2^{\pm}$
- Find the hidden bit

$2^2 + 2^7 = 128 + 4 + 132 - 127 = 2^{\square}$

127

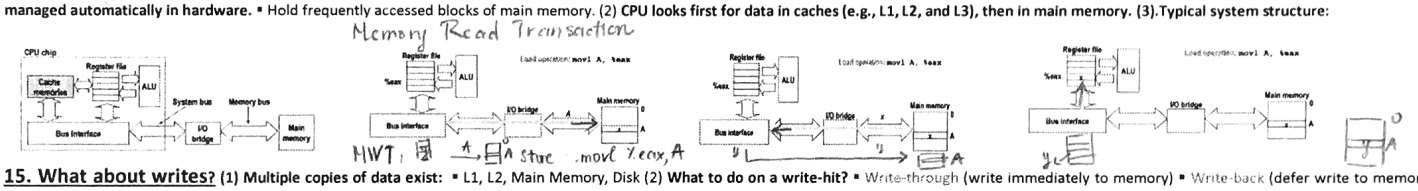
Hidden bit: implied bit left of the radix
1: almost all circumstances
0: denormal or Subnormal (if the exponent is all 0's)

173.7

$173 = 1.35 \times 2^7$

→ 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 eraseable 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), where1 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/in2): 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 x 1/RPMs x 60 sec/1 min ▪ Typical Tavg rotation = 7200 RPMs. (4) **Transfer time (Tavg transfer):** ▪ Time to read the bits in the target sector. ▪ Tavg transfer = 1/RPM x 1/(avg # sectors/track) x 60 secs/1 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 x (60 secs/7200 RPM) x 1000 ms/sec = 4 ms. ▪ Tavg transfer = 60/7200 RPM x 1/400 sects/track x 1000 ms/sec = 0.02 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 then 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 (10^15 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 about7timesmore. (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 CacheMisses:**(1)Cold (compulsory) miss ▪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:**



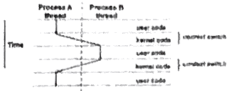
15. What about writes? (1) **Multiple copies of data exist:** ▪ L1, L2, Main Memory, Disk (2) **What to do on a write-hit?** ▪ Write-through (write immediately to memory) ▪ Write-back (defer write to memory until replacement of line): Need a dirty bit (line different from memory or not) (3) **What to 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 process
  - Each user thread is contained by some user PID
  - Each kernel thread is contained in 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*
  - Important: the kernel is not a separate process, but rather runs as part of some thread in some user process
- Control flow passes from one thread in a process to another thread in the same or a different process via a *context switch*



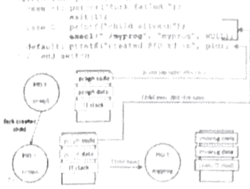
fork: Creating new processes

`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) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

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



Fork Example #1

Key Points
- Parent and child both run same code
- Child inherits a copy-on-write (COW) version of parent
- Relative ordering of parent/child print statements undefined

Fork Example #2

Key Points
- Both parent and child can continue forking

```
void fork2()
{
   printf("L0\n");
   fork();
   printf("L1\n");
   fork();
   printf("Bye\n");
}
```

exit: Destroying Process

`void exit(int status)`
- exits a process
  - Normally return with status 0
- `atexit()` registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Zombies

Idea
- When process terminates, still consumes system resources
  - Various tables maintained by OS
- Called a "zombie"
  - Living corpse, half alive and half dead

Reaping
- Performed by parent on terminated child
- Parent is given exit status information
- Kernel discards process

What if Parent Doesn't Reap?
- If any parent terminates without reaping a child, then child will be reaped by `init` process
- Only need explicit reaping for long-running processes
  - E.g., shells and servers

wait: Synchronizing with children

`int wait(int *child_status)`
- suspends current process until one of its children terminates
- return value is the pid of the child process that terminated
- if `child_status != NULL`, then the object it points to will be set to a status indicating why the child process terminated

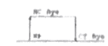
Declare a typedef for the exit status information returned from the `wait()` call (`pid = wait(int *status)`)

```
typedef union{
        int exit_status;
        struct{
                unsigned  sig_num :7;
                unsigned  core_dmp:1;
                unsigned  exit_num:8;
        }parts;
}LE_Wait_Status;
```

wait: Synchronizing with children

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



exec: Running new programs

- loads and runs executable at path with args arg0, arg1, ...
- returns -1 if error, otherwise doesn't return!

Summarizing

Exceptions
- Events that require nonstandard control flow
- Generated externally (interrupts) or internally (traps and faults)

Processes
- At any given time, system has multiple active processes
- Each process must have at least one execution thread
- Only one thread can execute on a processor (core) at a time, but the address space used on a core is that of the process whose thread is currently running there
- All threads of a given process share a common address space
- Each running thread appears to have total control of its core and its process's private address space
- The address space of a process can be in simultaneous use on multiple cores if the process has multiple running threads deployed across these multiple cores

Summarizing (cont.)

Spawning Processes
- Call to `fork()`
  - One call, two returns: one to parent, one to child in new process

Terminating Processes
- Call `exit(int exit_code)`
  - One call, no return
  - If called by any thread of a process, then all threads in the process will terminate, as will the process itself

Reaping Processes
- Call `wait (int * exit_status);`

Replacing Program Executed by Process
- Call `execl(char* path, char* argv0, … (char *)NULL);`
  - Actually can use any of 6 exec variants (execl, execlp, execv, etc.)
  - One call, new program starts at main() (no return to caller)

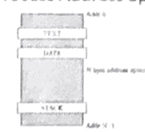
Processes

- In most contemporary Operating Systems such as Windows and Linux/UNIX, the unit of management is called a **process**
- A process is a resource container
  - Depending on the specific operating system, a process will have a set of defining attributes
  - At any given moment, the collection of processes in a system completely defines the system
    - All computations must be done in the context of a process

Processes (cont'd)

- While processes on various systems share much more in common than in difference, we will focus on the process model used in Linux
- A Linux process is characterized by many attributes, but foremost among these are:
  - An executable program
  - One or more threads that can run the program
  - An address space to contain all process memory objects (i.e. text, data, stack, etc.)

Process Address Space



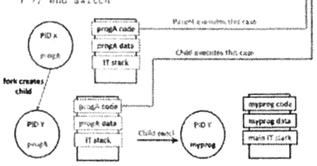
- Each memory object is a contiguous range of bytes within the address space
- The size of the address space is limited by the CPU architecture and the operating system version
- In a 32 bit Linux system on an x86 processor, the user default space is 3 GB (it's 128 TB in a 64 bit x86 system)

Threads

The executable (schedulable) elements in a Linux system
Each thread in the system is uniquely contained by some process
- Each user thread is contained by some user PID
- Each kernel thread is contained in 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

```
switch (int pid = fork()){
  case -1: perror("fork failed ");
           exit(1);
  case 0:  printf("child alive\n");
           execl("./myprog", "myprog", NULL);
  default: printf("created PID %d \n", pid);
} // end switch
```

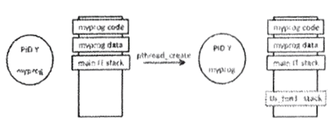


The new child program myprog executes from the first statement in its **main()** function.

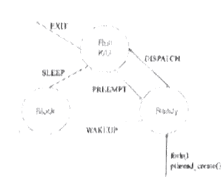
If the new program executes the following statement:

pthread_create(&tid_id, NULL, th_fun1, NULL);

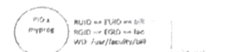
a **new stack** will be mapped into the address space



Thread States and Transitions



Thread Access Example



- A *system call* made by a thread in PID x is:
- The system call succeeds and returns a valid channel to read and write
- A second call made by a thread in PID x is:
- This call fails, since the calling process is the owner, and owner permissions don't allow WRITE, even though group and other do

---

```
int main(int argc, char* argv[]){
        int arg;
        char nstr[10];

        printf("LEVEL: %s\n", argv[1]);
        arg = atoi(argv[1]);                // convert arg string to int
        --arg;                              // decrement arg value
        sprintf(nstr, "%d", arg);           // convert int to arg string

        if(arg){
            switch(fork()){
                case -1:
                        exit(0);
                case 0:
                        execl("./proc_run", "proc_run", nstr, NULL);
            }
        }
        wait(NULL);
        printf("LEVEL: %s  PROC IS DONE\n", argv[1]);
}
```

A. Write all the output that will be generated when this program is run with the shell command shown above.

LEVEL 5
LEVEL 4
LEVEL 3
LEVEL 2
LEVEL 1
LEVEL 1 PROC IS DONE
LEVEL 2 PROC IS DONE
LEVEL 3 PROC IS DONE
LEVEL 4 PROC IS DONE
LEVEL 5 PROC IS DONE

B. Although we expect the `fork()` calls made above to succeed, in general, what could lead to a `fork()` call failing ?

fork() can fail when the system is out of resources (mem, swap) or a process limit is hit (too many user processes)

A. Exceptions are broadly **categorized** as either **synchronous or asynchronous**. **Explain the difference** between the two types of exceptions, and **provide an actual example of each type.**

Sync – divide by zero – caused by instruction execution
Async – disk controller interrupt – external event

B. An **running thread** tries to execute an instruction that **dereferences a NULL pointer**, which results in that thread running an **exception handler** in the kernel.
1) What **specific kind** of an exception is this event ?
This is a FAULT
2) When the **exception handler completes** and returns back to the code of the running thread, **which instruction** will the running thread **start to execute ?**
Attempt to re-execute the offending instruction

C. Consider a **CPU** that is currently executing an **IDLE** thread in **user space** at a time when a local disk controller has just completed transferring disk blocks from a disk into memory, and has sent an **interrupt** to that CPU. Between the executions of each instruction of the IDLE thread, the CPU checks for interrupts, and when it finds this one it **forces the IDLE thread** into the kernel to run the exception handler for this event. Can the IDLE thread **lose the CPU now** while in the kernel (i.e. can a context switch happen here), or **must the IDLE thread return to user mode** after completing the exception code ? **Explain your answer.**

IDLE thread may be pre-empted, and not get back to user run state for a while, but cannot block

The following shows the original sources for a simple **masm** program to be built from **two separate source files**. It also shows the object files produced when each is assembled with **masm** using the –o flag.

```
                    A SIMPLE MAIN PROGRAM
bash-2.05$ cat main1.asm
main:   lodd arg1:
        push
        lodd arg2:
        push
        call myadd:
        stod rslt:
        halt
        .LOC 10
arg1:   25
arg2:   75
rslt:   0
```

7, 11, 12, 13

```
                    A SIMPLE EXTERNAL FUNCTION
bash-2.05$ cat myadd.asm
myadd:  lodl 1
        addl 2
        addd bias:
        retn
bias:   100
```

```
          ASSEMBLE WITH -o OPTION
bash-2.05$ ./masm_mrd -o < main1.asm > main1.obj
bash-2.05$ cat main1.obj
   0    0000000000000000    arg1:
   1    1111010000000000
   2    0000000000000000    arg2:
   3    1111010000000000
   4    0111000000000000    myadd:
   5    0001000000000000    rslt:
   6    1111111111000000
  10    0000000000011001
  11    0000000001001011
  12    0000000000000000
 4096 x
  rslt:                          12
  arg2:                          11
  arg1:                          10
  main:                           0

          ALSO ASSEMBLED WITH -o OPTION
bash-2.05$ ./masm_mrd -o < myadd.asm > myadd.obj
bash-2.05$ cat myadd.obj
   0    1000000000000001
   1    1010000000000010
   2    0001000000000000    bias:
   3    1111100000000000
   4    0000000001100100
 4096 x
  bias:                           4
  myadd:                          0
```

13

If you built a **linker program** (just as you did in assignment #6), and linked these two separate object files into an **executable binary output file** so the first executable instruction from the file **main1.asm** was placed **at location 0** in the executable, that output file would have **18 lines of 16 bit entries**. The first **4** of these entries are provided below, **you must fill in the last 14.**

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1
3	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0
4																
5																
6																

Executable Content

```
 0 0000000000001010
 1 1111010000000000
 2 0000000000001011
 3 1111010000000000
 4 1110000000001101
 5 0001000000001100
 6 1111111111000000
 7 1111111111111111
 8 1111111111111111
 9 1111111111111111
10 0000000000011001
11 0000000001001011
12 0000000000000000
13 1000000000000001
14 1010000000000010
15 0010000000010001
16 1111100000000000
17 0000000001100100
```

Corresponding code

```
main:   lodd arg1:      10
        push
        lodd arg2:      11
        push
        call myadd:     12 → 13
        stod rslt:      12
        halt
        .LOC 10

arg1:   25
arg2:   75
rslt:   0
myadd:  lodl 1
        addl 2
        addd bias:      17 = 4 + 13
        retn
bias:   100
```

x : same
o : change
start at 13, so add 13 to myadd
myadd : address of 0 in the sample table

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/resource.h>
#include <string.h>
int main(void) {
    /* some local variables */
    pid_t pid, ppid;
    int ruid,rgid, euid, egid;
    int priority;
    char msg_buf[100];
    int msg_pipe[2];
    /* use the pipe() system call to create the pipe */
    if(pipe(msg_pipe) == -1){
        perror("failed in parent pipe creation:");
        exit(3);
    }
    /* use various system calls to collect and print process details */
    printf("\nThis is the Parent process report:\n");
    pid  = getpid();
    ppid = getppid();
    ruid = getuid();
    euid = geteuid();
    rgid = getgid();
    egid = getegid();
    priority = getpriority(PRIO_PROCESS, 0);
    printf("\nPARENT PROC: Process ID is:\t\t%d\n\
PARENT PROC: Process parent ID is:\t%d\n\
PARENT PROC: Real UID is:\t\t%d\n\
PARENT PROC: Real GID is:\t\t%d\n\
PARENT PROC: Effective UID is:\t\t%d\n\
PARENT PROC: Effective GID is:\t\t%d\n\
PARENT PROC: Process priority is:\t%d\n",
    pid, ppid, ruid, rgid, euid, egid, priority);
    printf("\nPARENT PROC: will now create child, write pipe \n\
and do a normal termination\n");
    /* use the sprintf() call to build a message to write into the pipe

    /* now use the fork() call to create the child */
    switch (pid = fork()) {
        /* if the call failes */
        case -1:
            printf("\nfailed to create a child process.\n");
            break;
        /* this is the parent's case
         * parent must write message to pipe and
         * do a normal exit */
        default:
            printf("\nPARENT PROC: created child with %d PID.\n", pid);
            write(msg_pipe[1], msg_buf, strlen(msg_buf));
            break;
        /* This is the child's case
         * child must create and print report
         * child must read pipe message and print
         * a modified version of it to output
         * child must do a normal exit */
        case 0:
            printf("\nThis is the Child process report:\n");
            pid  = getpid();
            ppid = getppid();
            ruid = getuid();
            euid = geteuid();
            rgid = getgid();
            egid = getegid();
            priority = getpriority(PRIO_PROCESS, 0);
            printf("\nCHILD PROC: Process ID is:\t\t%d\n\
CHILD PROC: Process parent ID is:\t%d\n\
CHILD PROC: Real UID is:\t\t%d\n\
CHILD PROC: Real GID is:\t\t%d\n\
CHILD PROC: Effective UID is:\t\t%d\n\
CHILD PROC: Effective GID is:\t\t%d\n\
CHILD PROC: Process priority is:\t%d\n",
        pid, ppid, ruid, rgid, euid, egid, priority);
            printf("\nCHILD PROC: about to read pipe and report parent message:\n");
            read(msg_pipe[0], msg_buf, 100);
            printf("CHILD PROC: parent's msg is:\t%s\n", msg_buf);
            printf("CHILD PROC: Process parent ID now is: %d\n", getppid());
            printf("CHILD PROC: ### Goodbye ###\n");
            exit(0);
```

Register designations are as follows:
pc=0 (prog counter) ac=1 (accumulator) sp=2 (stack ptr) ir=3 (instr reg)
tir=4 (tmp inst reg) zr=5 (fixed zero) po=6 (plus 1) no=7 (minus 1)
amsk=8 (addr msk) smsk=9 (stack msk) a=10(a scratch) b=11(b scratch)
c=12(c scratch) d=13(d scratch) e=14(e scratch) f=15(f scratch)

A. pc := pc + 255; mar := pc; rd; → mar: can't be put from mbr, only from Blatch
B. ac := inv(mbr); mar := inv(mbr); wr; ← can't invert mbr
C. tir := lshift(band(tir, mbr)); if n then goto 150;
D. mbr := lshift(band(ir, amsk)); ir := lshift(band(ir, amsk)); goto 0; wr;
E. b := ir - ac; mar := ac; if z then goto 158; ← can't do subtraction

enable; writing to the C; pc; pc (program counter); smask

MAL	VALID ?	AMUX	COND	ALU	SH	MBR	MAR	RD	WR	ENC	C	B	A	ADDR
A	yes	0	0	0	0	0	1	1	0	1	0	0	9	
B	no					1 (not writing to MAR)								
C	yes	1	1	1	2	0	0	0	0	1	4	4		150
D	yes	0	3	1	2	1	0	0	1	1	3	8	3	0
E	no										ir			

AMUX	COND	ALU	SH	MBR, MAR, RD, WR, ENC
0 = A latch	0 = no jmp	0 = A + B	0 = no shift	0 = no
1 = MBR	1 = jmp if n=1	1 = A and B	1 = shift rt	1 = yes
	2 = jmp if z=1	2 = A	2 = shift lt	
	3	3		

A Simplistic Program Translation Scheme

Problems:
- Efficiency: small change requires complete recompilation
- Modularity: hard to share common functions (e.g. printf)

Solution:
- Static linker (or linker)

A Better Scheme Using a Linker

Translating the Example Program

Compiler driver coordinates all steps in the translation and linking process.
- Typically included with each compilation system (e.g., gcc)
- Invokes preprocessor (cpp), compiler (cc1), assembler (as), and linker (ld).
- Passes command line arguments to appropriate phases

Example: create executable p from m.c and a.c:

```
bass> gcc -O2 -v -o p m.c a.c
cpp [args] m.c /tmp/cca07630.i
cc1 /tmp/cca07630.i m.c -O2 [args] -o /tmp/cca07630.s
as [args] -o /tmp/cca076301.o /tmp/cca07630.s
<similar process for a.c>
ld -o p [system obj files] /tmp/cca076301.o /tmp/cca076302.o
bass>
```

What Does a Linker Do?

Merges object files
- Merges multiple relocatable (.o) object files into a single executable object file that can be loaded and executed by the loader.

Resolves external references
- As part of the merging process, resolves external references.
  - External reference: reference to a symbol defined in another object file.

Relocates symbols
- Relocates symbols from their relative locations in the .o files to new absolute positions in the executable.
- Updates all references to these symbols to reflect their new positions.
  - References can be in either code or data
    - code: a(); /* reference to symbol a */
    - data: int *xp=&x; /* reference to symbol x */

Why Linkers?

Modularity
- Program can be written as a collection of smaller source files, rather than one monolithic mass.
- Can build libraries of common functions (more on this later)
  - e.g., Math library, standard C library

Efficiency
- Time:
  - Change one source file, compile, and then relink.
  - No need to recompile other source files.
- Space:
  - Libraries of common functions can be aggregated into a single file...
  - Yet executable files and running memory images contain only code for the functions they actually use.

Executable and Linkable Format (ELF)

Standard binary format for object files

Derives from AT&T System V Unix
- Later adopted by BSD Unix variants and Linux

One unified format for
- Relocatable object files (.o)
- Executable program files (a.out)
- Shared object library files (.so)
- Core dump files

Generic name: ELF binaries

ELF Object File Format

Elf header
- Magic number, type (.o, exec, .so, core), machine, byte ordering, etc.

Program header table
- Page size, virtual addresses memory segments (sections), segment sizes.

.text section
- Code

.data section
- Initialized (static) data

.bss section
- Uninitialized (static) data
- "Block Started by Symbol"
- "Better Save Space"
- Has section header but occupies no space

ELF Object File Format (cont)

.symtab section
- Symbol table
- Procedure and static variable names
- Section names and locations

.rel.text section
- Relocation info for .text section
- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying.

.rel.data section
- Relocation info for .data section
- Addresses of pointer data that will need to be modified in the merged executable

.debug section
- Info for symbolic debugging (gcc -g)

Example C Program

```
m.c
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

```
a.c
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

Merging Relocatable Object Files into an Executable Object File

Relocating Symbols and Resolving External References

- Symbols are lexical entities that name functions and variables.
- Each symbol has a value (typically a memory address).
- Code consists of symbol definitions and references.
- References can be either local or external.

Strong and Weak Symbols

Program symbols are either strong or weak
- strong: procedures and initialized globals
- weak: uninitialized globals

```
p1.c                    p2.c
strong → int foo=5;     int foo;  ← weak
strong → p1() {         p2() {    ← strong
         }              }
```

Linker's Symbol Rules

Rule 1. A strong symbol can only appear once.

Rule 2. A weak symbol can be overridden by a strong symbol of the same name.
- references to the weak symbol resolve to the strong symbol.

Rule 3. If there are multiple weak symbols, the linker can pick an arbitrary one.

Linker Puzzles

int x; p1() {}	p1() {}	Link time error: two strong symbols (p1)
int x; p1() {}	int x; p2() {}	References to x will refer to the same uninitialized int. Is this what you really want?
int x; int y; p1() {}	double x; p2() {}	Writes to x in p2 might overwrite y! Evil!
int x=7; int y=5; p1() {}	double x; p2() {}	Writes to x in p2 will overwrite y! Nasty!
int x=7; p1() {}	int x; p2() {}	References to x will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Packaging Commonly Used Functions

How to package functions commonly used by programmers?
- Math, I/O, memory management, string manipulation, etc.

Awkward, given the linker framework so far:
- Option 1: Put all functions in a single source file
  - Programmers link big object file into their programs
  - Space and time inefficient
- Option 2: Put each function in a separate source file
  - Programmers explicitly link appropriate binaries into their programs
  - More efficient, but burdensome on the programmer

Solution: static libraries (.a archive files)
- Concatenate related relocatable object files into a single file with an index (called an archive).
- Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
- If an archive member file resolves reference, link into executable.

Static Libraries (archives)

Further improves modularity and efficiency by packaging commonly used functions (e.g., C standard library (libc), math library (libm))

Linker selectively includes only the .o files in the archive that are actually needed by the program.

Creating Static Libraries

Archiver allows incremental updates:
- Recompile function that changes and replace .o file in archive.

Commonly Used Libraries

libc.a (the C standard library)
- 2.4 MB archive of 1395 object files on mercury.cs.uml.edu
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

libm.a (the C math library)
- 450 KB archive of 401 object files
- floating point math (sin, cos, tan, log, exp, sqrt, ...)

Using Static Libraries

Linker's algorithm for resolving external references:
- Scan .o files and .a files in the command line order.
- During the scan, keep a list of the current unresolved references.
- As each new .o or .a file obj is encountered, try to resolve each unresolved reference in the list against the symbols in obj.
- If any entries in the unresolved list at end of scan, then error.

Problem:
- Command line order matters!
- Moral: put libraries at the end of the command line.

```
bass> gcc -L. libtest.o -lmine
bass> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

Loading Executable Binaries

Variably sized byte regions overlaid on a fixed page size object

Shared Libraries

Static libraries have the following disadvantages:
- Potential for duplicating lots of common code in the executable files on a filesystem.
  - e.g., every C program needs the standard C library
- Potential for duplicating lots of code in the virtual memory space of many processes.
- Minor bug fixes of system libraries require each application to explicitly relink

Solution:
- Shared libraries (dynamic link libraries, DLLs) whose members are dynamically loaded into memory and linked into an application at run-time.
  - Dynamic linking can occur when executable is first loaded and run.
    - Common case for Linux, handled automatically by ld-linux.so.
  - Dynamic linking can also occur after program has begun.
    - In Linux, this is done explicitly by user with dlopen()
    - Basis for High-Performance Web Servers.
  - Shared library routines can be shared by multiple processes.

Dynamically Linked Shared Libraries

The Complete Picture

A Simplistic Program Translation Scheme

m.c
ASCII source file
Translator
Binary executable object file (memory image on disk)
Problems:
- Efficiency: small change requires complete recompilation
- Modularity: hard to share common functions (e.g. printf)
Solution:
- Static linker (or linker)

Altering the Control Flow

Up to Now: two mechanisms for changing control flow:
- Jumps and branches
- Call and return using the stack discipline.
- Both react to changes in program state.

Insufficient for a useful system
- Difficult for the CPU to react to changes in system state.
  - data arrives from a disk or a network adapter.
  - Instruction divides by zero
  - User hits ctl-c at the keyboard
  - System timer expires

System needs mechanisms for "exceptional control flow"

Exceptional Control Flow

- Mechanisms for exceptional control flow exists at all levels of a computer system.

Low level Mechanism
- exceptions
  - change in control flow in response to a system event (i.e., change in system state)
- Combination of hardware and OS software

Higher Level Mechanisms
- Process context switch
- Signals
- Nonlocal jumps (setjmp/longjmp)
- Implemented by either:
  - OS software (context switch and signals).
  - C language runtime library: nonlocal jumps.

Synchronous Exceptions

Caused by events that occur as a result of executing an instruction:
- Traps
  - Intentional
  - Examples: system calls, breakpoint traps, special instructions
  - Returns control to "next" instruction
- Faults
  - Unintentional but possibly recoverable
  - Examples: page faults (recoverable), protection faults (unrecoverable).
  - Either re-executes faulting ("current") instruction or aborts.
- Aborts
  - unintentional and unrecoverable
  - Examples: parity error, machine check
  - Aborts current program

Trap Example

Opening a File
- User calls open(filename, options)

```
0804d070 <__libc_open>:
 . . .
 804d082:   cd 80       int   $0x80
 804d084:   5b          pop   %ebx
 . . .
```

- Function open executes system call instruction int
- OS must find or create file, get it ready for reading or writing
- Returns integer file descriptor

System context for exceptions

Exceptions

An exception is a transfer of control to the OS in response to some event (i.e., change in processor state)

Interrupt Vectors

- Each type of event has a unique exception number k
- Index into jump table (a.k.a., interrupt vector)
- Jump table entry k points to a function (exception handler).
- Handler k is called each time exception k occurs.

Asynchronous Exceptions (Interrupts)

Caused by events external to the processor
- Indicated by setting the processor's interrupt pin
- handler returns to "next" instruction.

Examples:
- I/O interrupts
  - hitting ctl-c at the keyboard
  - arrival of a packet from a network
  - arrival of a data sector from a disk
- Hard reset interrupt
  - hitting the reset button
- Soft reset interrupt
  - hitting ctl-alt-delete on a PC

Fault Example #1

Memory Reference
- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7:   c7 05 10 9d 04 08 0d    movl   $0xd,0x8049d10
```

- Page handler must load page into physical memory
- Returns to faulting instruction
- Successful on second try

Fault Example #2

Memory Reference
- User writes to memory location
- Address is not valid

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7:   c7 05 60 e3 04 08 0d    movl   $0xd,0x804e360
```

- Page handler detects invalid address
- Sends SIGSEGV signal to user process
- User process exits with "segmentation fault"

Processes

Def: A process is an instance of a running program.
- One of the most profound ideas in computer science.
- Not the same as "program" or "processor"

A process provides each program with two key abstractions:
- Logical control flow
  - Each program seems to have exclusive use of the CPU.
- Private address space
  - Each program seems to have exclusive use of main memory.

How are these illusions maintained?
- Process executions are interleaved (multitasking)
- Address spaces are managed by a virtual memory system

Thread States and Transitions