Notecard / CS271 / Lyell Read / Winter 2019

Language Levels, Process of Assembly
- High (Java, C++...)
- Low (ASM)
- Machine Code (Binary)

Compilation and Such
- High Level
  - High <--(de)compile--> Low <--(dis)assemble--> Machine Code
- Lower Level
  - Assembler: Source Code (ASM, Text) --> [Object File + Listing File]
  - Linker: Object File --<with link library>--> Executable File
  - OS Loader: Executable File --> Output (printout)
- Linking
  - <prog> can either link to Irvine32.lib (which links to kernel32.lib) or directly to kernel32.lib.
  - Kernel32.lib executes kernel32.dll.

Bits and Bytes
- A bit is a Binary digit. Can be either 1 or 0.
- A byte is a group of 8 bits.

Speeds
- Listed in [ ]bps (Where [ ] = M, K,...).
- Ex: Mbps means Megabits per second; 8Mbps = 8,000,000 bit/sec

Sizes
- Sizes use [ ]B or [ ]iB (ie. GB == GiB)
- KB = 2^{10}; MB= 2^{20} ... YB = 2^{80}.
- Ex: A 200GiB HDD has 200 * 230 bytes = 1.717 trillion bits.
- "B" = bytes; "b" = Bits

IA-32 Architecture
- 32-bit architecture, means 232 addressable locations.
- Min memory size around 2 byte?
- Is Byte Addressable: each byte of memory is individually addressable.
- Is a CISC: uses microprograms to implement machine instructions.
- Is Little Endian (see section on endianness)

Four CPU parts:
- Clock, Registers, Control Unit, Arithmetic Logic Unit.

Processor Modes
- System Management: Full control. No Memory protection. Used for hardware level things like security and power management.
- Real Address: More locked down than System Management. Programs have full access to memory and devices. About 1MB range accessible.
- Protected: This is the "native state" - all features and instructions are available to the programs. Programs limited to memory in their segment, 4 GB linear space. (Using "extended physical addressing), you can reach up to 64GB).
- Protected -> Virtual Mode (AKA Virtual-8086): Within Protected. Simulates Real Address Mode. Helpful to run some programs (MS-DOS) that need real address mode.
- 64-bit -> Compatibility Mode: Works with 32-bit exe's, typically without recompilation.
- 64-bit Mode: uses the 64 bit address space.

Processing Units:
- Floating Point Unit (FPU)
- Integer Unit (what we will work with)

Registers:
- 64-bit:

Register Size	Reg's of that size. Credit: Irvine Text.
8-bit	AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L, R9L, R10L, R11L, R12L, R13L, R14L, R15L
16-bit	AX, BX, CX, DX, DI, SI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W
32-bit	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D
64-bit	RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15

- EAX (accumulator), EBX (base), ECX (counter), EDX (32-bit)
- EBP, ESP, ESI, EDI (32-bit). These can only be "broken down" to 16-bit sub-registers (i.e. EBP --> BP).
- EFLAGS (flags: set:1 unset:0), EIP (EIP = instruction pointer; cannot modify EIP directly, 32-bit)
- CS, SS, DS, ES, FS, GS (16-bit)
- Sub-Registers (Only for EAX, EBX, ECX, EDX)

		AH (8)	AL (8)
		AX (16)	
EAX (32)			

Flags
- CF: Carry Flag: unsigned operation overflow.
- OF: Overflow Flag: signed operation overflow.
- SF: Sign Flag: sign change as a result of operation
- ZF: Zero Flag: set when an operation produces 0.
- AC: Aux- Carry: set when operation causes carry out of bit 3 to bit 4 in an 8-bit reg.
- PF: Parity Flag: set if LSB contains even # of 1 bits.

Execution Cycle
- CPU does Instruction Fetch; Inc EIP
- CPU does Decode of Instructions
- If operands: CPU does fetch operands
- CPU executes the instruction. Updates some flags.
- If output operand: CPU does store result.
- Simplified as Fetch -> Decode -> Execute.

Read from Memory Process
- Place the Address on address bus
- Change the RD pin value (Assert to the CPU)
- Wait one clock cycle for a response
- Put data from data bus into destination
- Can be sped up with Level-* (1,2,3...) cache (Abbrev'd L* cache)

Operator Precedence
- ()
- +,- (UNARY)
- *,/,MOD
- +,-

Valid Real Literals
- Must have int and decimal.
- 2.
- +3.0
- -44.2E+05
- 26.E5

Rules for Identifiers
- Identifier indicates a Const, Var, Proc, Lbl
- 1..247 Characters
- Not CAse sEnsiTIve
- First character must be in {A..Z, a..z, _, @, ?, $}
- Must not be a reserved word.

Data Types

Type	Details; Credit: Irvine Text
BYTE	8-bit unsigned int
SBYTE	8-bit signed int
WORD	16-bit unsigned int
SWORD	16-bit signed int
DWORD	32-bit unsigned int
SDWORD	32-bit signed int
FWORD	48-bit int
QWORD	64-bit int
TBYTE	80-bit int
REAL4	32-bit IEEE short real
REAL8	64-bit IEEE long real
REAL10	80-bit IEEE extended real

DUP operator
- <type> x DUP (y) ; y concatenated x times...
- For arrays, offset will be data size from the top (0th element)

Calculating Variable Offsets (in declaration)
- The following data segment starts at memory address 0x3700 (hexadecimal)
- .data
- printString BYTE "Do not add decimal to hex",0
- someBytes WORD 29 DUP(0)
- moreBytes BYTE 10, 20, 30, 40, 50, 60, 70, 80, 90
- questionAddr DWORD ?
- ignoreMe WORD ?
- What is the hexadecimal address of questionAddr?
- 0x375D

Endianness
- Little Endian representation of 0x12 34 56 78: (note each pair is a byte): 0x0000: 78; 0x0001: 56; 0x0002: 34; 0x0003: 12
- Big Endian representation of 0x12 34 56 78: 0x0000: 12; 0x0001: 34; 0x0002: 56; 0x0003: 78

Equal Sign Operator
- NAME = <value>
- Mov eax, NAME is better than mov eax, <value>
- Can be redefined in-line.

List Size Cheat
- List_Name BYTE 4,2,4,5,11,2
- List_Size = ($- List_Name)

EQU and TEXTEQU
- <name> EQU <(expression | symbol | <text>)>
- <name> TEXTEQU <(%(constant_expression (i.e. math)) | <text> | textmacro)>
- Can't be redefined at any time

Opcodes / Operands
- Can take 0,1,2,3 operands (X, dest, src-1, src-2)
- There are three types of operands: Immediate (literal); Register (uses reg); Memory (ref to mem loc).

Direct Offsets
- To get the first item, call <name>, then for the first, use [<name> + <size> * <element>].
- This creates an effective address

Check for Signed Overflow (with + and -)
- If two positive operands generates a negative sum
- If two negative operands generate a positive sum

Stack
- LIFO
- Not directly modified, typically, instead using POP PUSH RET and CALL...
- Push decrements ESP; pop increments ESP
- ESP points to the last pushed data to the stack; EBP points to base of the stack.

Procedures
- <name> PROC ... ret <name> ENDP
- When CALL is called, the next address to be executed is pushed to stack.
- When RET is called, the element on the stack is popped back into EIP and execution is completed.

Pointers
- Generated as in mov esi, OFFSET <var>
- Used as mov al, [esi]
- Note: inc [esi] requires size note, so inc BYTE PTR [esi] - err operand must have size

To add a constant to an operand, do:
- Constant[register]
- [register + constant]

TYPEDEFs: PBYTE TYPEDEF PTR BYTE

Stack Frames and Calling Conventions
- C Calling Convention: follow call with stack cleanup by forced manipulation of ESP.
- STDCALL: pass the number of bytes used to ret, which does cleanup.

Binary Operations:
- Add: 1+1 = 10
- Sub: 2's Complement, then add.
- Mult: Repeatedly add
- Div: Repeated subtraction or Shifting: i.e. 45/8; 8=2^3; 45 = 00101101. Shift 3 to the right: 00101 | 101 where left is q and right is r = 5r5.

Floating Point
- Decimal point => "radix point"

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
4	2	1	0.5	0.25	0.125

- 4.5 dec -> 100.1bin
- Some rationals are irrational in binary.
- IEEE754 Standard

Name	Total Bits	Sign Bits	Exponent	Mantissa
Single precision	32	1	8	23
Double Precision	64	1	11	52
Extended (AKA Double Extended)	80	1	15	64

- Conversion to IEEE754:

a. Convert 6.25 to single precision FP
b. 6.25 dec -> 110.01 bin
c. Move radix to the left until there is only one "1" left of the radix, mult by appropriate power of 2 for places that were moved: 110.01 -> 1.1001 * 2^2
d. Determine Sign Bit. Add that to the start. In this case it is POS == 0:
e. Calc Biased Exponent: 127 + moves taken in (c). In this case: B.E. == 2 + 127 = 129 = 10000001
f. Normalized Mantissa: Drop 1 left of radix (1.1001 -> 1001), and pad to length of mantissa (in this case 23): MANTISSA == 10010000000000000000000
g. Combine Sign Bit + B.E. + Mantissa: 01000000110010000000000000000000 = 0x40c80000

- To undo, undo steps above; don't forget sign.

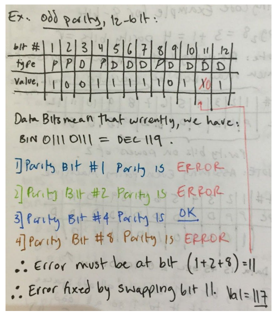
Hamming Codes and Assoc Learning
- Parity
  - The number of "1" bits in a certain set of binary code.
  - Even Parity Bit (first bit) example: 1 11010110
  - Even Parity Bit (first bit) example: 0 11010110
- Hamming Codes
  - For a n bit code, n=m+r where r = parity bits, m = data bits.
  - There are 2n combinations of bits. Only 2m are correct.
  - Parity Bit Count = r = log2m + 1
  - There are 2r invalid codes
  - Use log2m + 1 hamming codes rounded up.
- 8-bit hamming code example
  - n = m + r -> n = (8) + (log28 + 1 = 4). n = 12.
  - Data is 45dec = 00101101bin, parity bits are on powers of 2 byte#'s

1	2	3	4	5	6	7	8	9	10	11	12
X	X	0	X	0	1	0	X	1	1	0	1

    - P.B. 1 controls all odd spaces
    - P.B. 2 controls all spaces with a 1 in the 2's place (so 3,6,7,10,11)
    - P.B. 4 controls all spaces with a 1 in the 4's place (so 5,6,7,12)
    - P.B. 8 controls (9,10,11,12)
  - For an even parity system:

1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	0	1	0	1	1	1	0	1

- Undoing Hamming Codes:



Ex.. odd parity, 12-bit:

bit #	1	2	3	4	5	6	7	8	9	10	11	12
type	P	P	D	P	D	D	D	P	D	D	D	D
Value,	1	0	0	1	1	1	1	1	0	1	~~1~~ 0	1

Data Bits mean that currently, we have:
BIN 0111 0111 = DEC 119 .
1] Parity Bit #1 Parity is ERROR
2] Parity Bit #2 Parity is ERROR
3] Parity Bit #4 Parity is OK
4] Parity Bit #8 Parity is ERROR
∴ Error must be at bit (1+2+8) = 11
∴ Error fixed by swapping bit 11. Val = 117

-

Program Sections:

```
• TITLE Program Template      (template.asm)
  ; Author:
  ; Course / Project ID                Date:
  ; Description:
• INCLUDE Irvine32.inc
  ; (insert constant definitions here)
• .data
  ; (insert variable definitions here)
• .code
  main PROC
  ; (insert executable instructions here)
• exit
  ; exit to operating system
  main ENDP
• ; (insert additional procedures here)
• END main
```

Declaring Text

- name BYTE “text”,0

Hand Selected Opcodes

- LAHF: Load Status Flags into AH
- SAHF: Save Data Flags from AH
- XCHG: exchanges the two operands
- PUSHFD, POPFD: Push and pop EFLAGS.
- PUSHAD,POPAD: pushes (or pops) in the order: EAX, ECX, EDX, EBX, ESP [val before pushad], EBP, ESI, EDI. POPAD pops in reverse order (per stack lifo)
- MUL: (only one operand, other, see set ->) returns product. Result is stored in {ax for al; dx:ax for ax; edx:eax for eax. Carry flag set if the upper (first) half is used.
- IMUL: same operands as mul, just sx. Can also take one reg and one [*] to mult, stores result in reg. Also can be reg, [*], [*] and multiplies the last two into reg.
- DIV: (one operand). {quotient, remainder for input; al, ah for ax; ax, dx for dx:ax; eax, edx for edx:eax.
- CDQ, CWD, CBW: sign extensions for operations.
- IDIV: signed version of DIV.
- PUSHA, POPA pushes (or pops) selected 16-bit registers in the order AX, CX, DX, BX, SP, BP, SI, DI.
- LOOP: sub’s 1 from ecx, them cmp to 0.
- INC: doesn't care about carry flag.
- CMP a, b: {a=b: ZF=1, CF=0; a>b: (ZF|CF)=0; a<b: ZF=0, CF=1}
- SHR: shift right: CF accepts rightmost, displaced bit. 1001 -> 0100 + CF=1
- TEST al,00001001b will test if bits 0,3 are set. If either is, ZF will be set. Otherwise,

Bin <==> Dec:

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}

Bin <==> Hex:

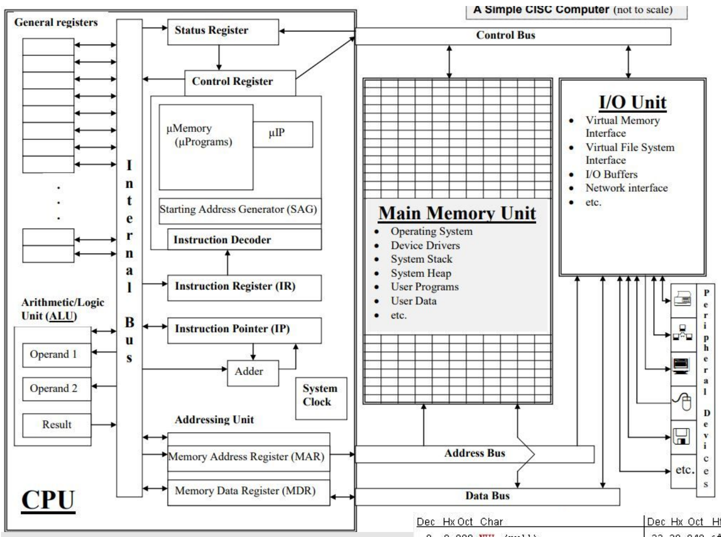
1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	16	ff
0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	0001 0110	11111111

Dec <==> Hex Ex: 578 -> 2 4 2

4096	256	16	1
16^3	16^2	16^1	16^0

Hamming Code Template

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100



6. (10 pts) Using the Intel IA32 instruction set please write a program that accepts an **unsigned** binary integer in the AX register and pushes the decimal ASCII equivalent value onto the system stack. For example, suppose that the AX register initially contains: 0b1001011001101011 (decimal value 38507). After your program has fully executed, the system stack should contain the following numbers:

55 (the ASCII representation of 7)
48 (the ASCII representation of 0)
53 (the ASCII representation of 5)
56 (the ASCII representation of 8)
51 (the ASCII representation of 3)

Note that the number at the bottom of the stack represents the 10,000's place, the number above it represents the 1,000's place, the number above it represents the 100's place, the number above it represents the 10's place and the top number on the stack represents the 1's place.

Hints: 1) As a reminder, the instruction **DIV BX** results in the following behavior: AX <- DX:AX / BX and DX <- remainder

2) Since the AX register can only hold 2 bytes, its maximum unsigned value is 65,535. Therefore, one way of approaching this assignment is to start by dividing AX by 10000, pushing the ASCII representation of the quotient onto the stack and then dividing the remainder by 1000. You can use this general idea to solve for the 10000's, 1000's, 100's, 10's and 1's places.

There are many ways to write this code but the example on the next page illustrates one approach.

```
; AX already contains the unsigned integer
MOV BX, 10000 ; [illegible]
MOV CX, 10 ; We are going to keep dividing the divisor by 10
           ; First the divisor will be 10000, then it will be 1000,
           ; then 100, then 10

loopAgain:
  MOV DX, 0
  ; [illegible]
  ; [illegible]
  ; remainder in DX
  DIV BX ; the quotient is in AX and the remainder is in DX
  ADD AX, 48 ; Convert the quotient into ASCII [illegible]
             ; [illegible]
  PUSH AX ; Push the ASCII value onto the stack

  ; [illegible]
  CMP BX, 1
  JE finished

  ; Now we need to calculate the next divisor by dividing the existing
  ; divisor by 10
  MOV AX, BX ; Copy the current divisor into AX
  DIV CL     ; Divide AX (the current divisor) by 10
             ; [illegible]
  MOVZX BX, AL ; Move AL into BX and fill the top byte with zeros

  ; [illegible]
  MOV AX, DX ; [illegible]

JMP loopAgain ; Repeat the loop
```

```
mov   [illegible]       ; immediate value
mov   eax, ebx          ; register to register
mov   eax, [illegible]  ; immediate addressing
mov   [illegible]       ; indirect addressing
mov   eax, [illegible]  ; indirect with offset
mov   eax, [illegible]  ; [illegible] addressing

; assume ebx ptr Vector2
mov   [ebx].Vector2.x, [illegible]
mov   [illegible]
mov   (Vector2 ptr [ebx]).z, 0

add   eax, [illegible]  ; [illegible]
sub   [illegible]       ; [illegible]
inc   eax               ; [illegible]
and   eax, ebx          ; [illegible]
or    [illegible]       ; [illegible]

imul  ebx               ; [illegible]
imul  [illegible]       ; [illegible]
imul  eax, ebx, 4       ; [illegible]

[illegible]
div   ebx               ; [illegible]
```

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL