DEFINITIONS

"**hardware**" refers to the tangible physical devices that comprise a computer system.

"**software**" refers to the instructions that control the hardware.

Cross-assemblers (software) can be used to convert a machine language to another machine language.

Virtual machines (software) can be used to simulate another computer's architecture.

hardware's **instruction set architecture** (ISA) provides a micro-program for each machine instruction (CISC*) or direct execution (RISC*)

DEFINING DATA

Variable – *[name (**no strt w/ #**)] directive initializer [, initializer]… -* ***Extra initializers would be for array elements***

String – *name* BYTE *"Text.",0* (\ continues string to the next line)

DUP – store multiple of a thing *name DATATYPE # DUP (thingToStore#of)*

Constants: *name = expression* (all name replaced by expression) OR *name EQU expression / symbol / <text> (no redefinition*)

Current Location: $ returns current memory offset *selfPtr DWORD $; contains its own memory add*

MISCELLANEOUS

MASM Identifier syntax: not case sensitive, start with letter, _, @, $; remaining letter, digit, or _

Little Endian: data stored from LSB to MSB

Sign Extension: Use CBW, CWD, and CDQ to extend sign before doing signed division.

Stack: Memory arr managed by ESP w/ 32 bit values (LIFO)

PROCS: Define proc w/ procName PROC … ret procNam ENDP

USES – *USES reg reg …* - just following PROC generates push and pops for all regs specified this way

Sizes: Ki = 2^{10}, Mi = 2^{20}, Gi = 2^{30} (*for B, multiply by 8*)

CISC: protected / real-address modes, integer & FP units (2in1)

DATA REPRESENTATION

Num bits needed to represent unsigned int n: ceiling (log2n)

Unsigned int range for n bits: 0 – (2n – 1)

Signed int range for n bits: -2(n-1) – (2(n-1) – 1)

Hex 2's complement: subtract all digits from 15, then add 1

Signed Hex: MSB >= 8 means negative

FLAGS

Carry:—unsigned int overflow, **Overflow**: signed int OF, **Zero**: result zero, **Auxillary carry**: 1 bit carries out of LSB, **Sign**: negative result, **Parity**: even # of 1 bits

x86 Floating Point: sign, significand, and an exponent

MICROPROCESSOR DESIGN

CPU: registers, high frequency clock, control unit, ALU

- * Clock synchronizes CPU, CU coordinates steps for executing instructions, ALU performs arithmetic / logic
- * *Cache*: fast temp storage for mem copied from RAM

Memory Storage Unit: hold instructions / data while prog running (CPU <-> RAM).

Busses (parallel wires for data transfer, 32 Bits wide)

- * Data: CPU <-> Memory
- * I/O: CPU <-> I/O devices
- * Control: synchronize all devices attached to system bus
- * Address: instruction / data addresses when CPU <-> RAM
- * Speed depends on width (num bits it can transfer simul taneously).
- * Must be able to deal with arbitration / multiple busses

VonNeuman: prog stored in mem, exec. by OS using **EEC**.

•**Argument** (actual parameter) is a value or reference **passed to** a procedure.

•**Parameter** (formal parameter) is a value or reference **received by** a procedure.

•**Return value** is a value determined by the procedure, and **communicated back** to the calling procedure.

An **input parameter** is data passed by a calling program to a procedure.

•The called procedure is not expected to modify the corresponding argument variable, and even if it does, the modification is confined to the procedure itself.

•An **output parameter** is created by passing the **address** of an argument variable when a procedure is called.

•The "address of" a variable is the same thing as a "**pointer** to" or a "**reference** to" the variable. In MASM, we use **OFFSET**.

•The procedure does not use any existing data from the variable, but it fills in new contents before it returns.

•An **input-output parameter** is the **address** of an argument variable which contains input that will be both used and modified by the procedure.

•The content is modified at the memory address passed by the calling procedure.

MASM Data Types

Type	**Used for:**
BYTE	Character, string, 1-byte integer
WORD	2-byte integer, address
DWORD	4-byte unsigned integer, address
FWORD	6-byte integer
QWORD	8-byte integer
TBYTE	10-byte integer
REAL4	4-byte floating-point
REAL8	8-byte floating-point
REAL10	10-byte floating-point

INSTRUCTION EXECUTION CYCLE (EEC)

1. Fetch instruction (IP → IR).
2. Increment instruction pointer (IP).
3. Decode instruction in instruction register (IR)
4. If needed, fetch operands from memory / registers.
5. Execute micro-program / update status flags
6. To to 1. (If necessary, store result in output

CMP Results	Flags
Destination < source	SF ≠ OF
Destination > source	SF = OF
Destination = source	ZF = 1

Computer languages/hardware by "levels"

Level 4: Problem solution in natural language
–Description of algorithm, solution design
–Programmer translates to .....

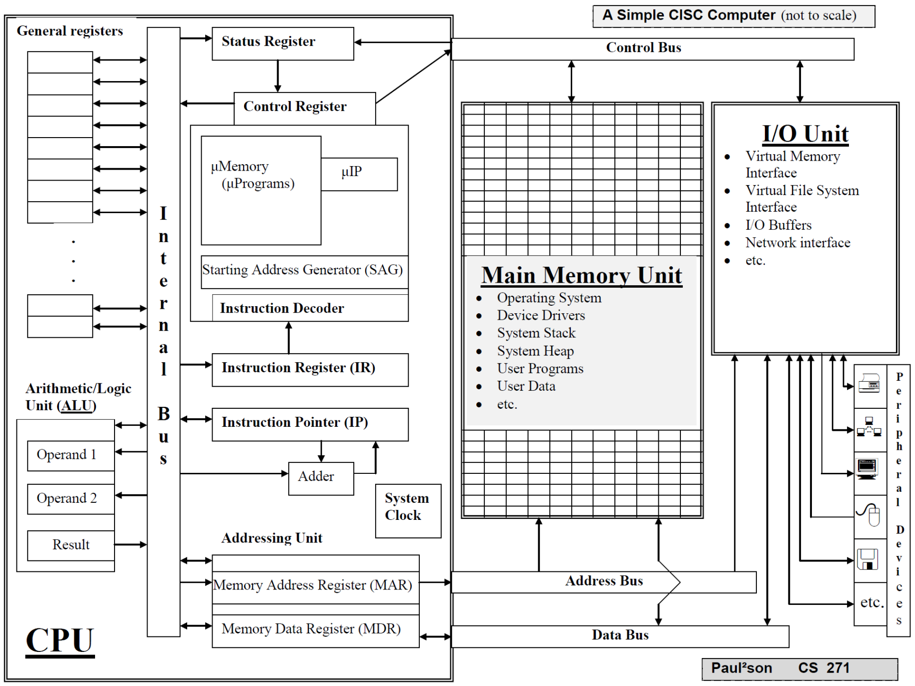
Level 3: Computer program in high-level computer programming language
–Source code (machine independent)
–Compiler translates to ...

Level 2: Program in assembly language
–Machine specific commands to control hardware components
–Assembler translates to ...

Level 1: Program in machine code
–Object code (binary)
–Linker / loader sets up ...

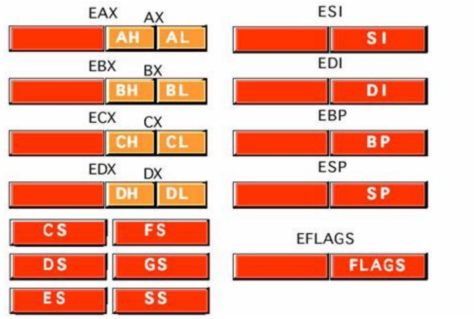
Level 0: Actual computer hardware
–Program in electronic form

CMP Results	ZF	CF
Destination < source	0	1
Destination > source	0	0
Destination = source	1	0



IRVINE32 LIBRARY

Clrscr – clear the screen
Crlf – New line
Delay – pause prog for num miliseconds in eax
GetMaxXY – num col/row in console buff (dl = col, dh = row)
GetMseconds – get number of milliseconds since midnight (eax)
Gotoxy – locate cursor at DL = col & DH = row
IsDigit – is char in AL a valid decimal digit
Random32 – EAX = random int from 0 – FFFFFFFFh
Randomize – seed random number generator
RandomRange – start w/ int in eax, returns int from 0 – (eax – 1)
ReadChar – get char from intput and store in AL (al = asci code)
ReadDec – get unsigned int to eax, if cf=0 means valid num entered and sf=sign, cf=1, invalid entry & eax=0
ReadHex – read 32 bit hex to eax from input
ReadInt – read integer from keyboard, store in eax
ReadString – Read string from keyboard, store in edx, store len of str in eax (offset of mem must be in EDX and size in ecx)
SetTextColor – in eax, bits0-3=foreground, bits4-7= background
WriteInt / WriteDec – write integer in eax to screen (int w/ +/-)
WriteString – Write null term string (offset in edx) to screen
ReadFloat – get keyboard input into ST(0)
WriteFloat – write float in ST(0) to screen



Typical Uses of General-Purpose Registers

Register	Size	Typical Uses
EAX	32-bit	Accumulator for operands and results
EBX	32-bit	Base pointer to data in the data segment
ECX	32-bit	Counter for loop operations
EDX	32-bit	Data pointer and I/O pointer
EBP	32-bit	Frame Pointer - useful for stack frames
ESP	32-bit	Stack Pointer - hardcoded into PUSH and POP operations
ESI	32-bit	Source Index - required for some array operations
EDI	32-bit	Destination Index - required for some array operations
EIP	32-bit	Instruction Pointer
EFLAGS	32-bit	Result Flags - hardcoded into conditional operations

MASM INSTRUCTIONS

ADD: add source to destination
CALL: push ret address on stack, copies proc address into IP
CBW – Convert byte to word AX = AL
CDQ: DWORD -> QWORD extend sign from EDX:EAX = EAX
CWD – convert word to double DX:AX = AX
CWDE – convert word extended double EAX = AX
CMP – compare two operands (used for jumps)
DIV – unsigned divide, EAX = EDX:EAX / op
IDIV – signed divide (Save as DIV)
LAHF – load status flags into AH
LEA – load effective address (OFFSET for stack operations)
LOOP – *loop destination* – sub 1 frm ecx, if exc not zero, jump to destination
MOV – mov src to dest (*no mem to mem*)
MOVZX—UNSIGNED
MOVSX – *movsx dest, src* – move src to dest with sign extended
MOVZH – *movzh dest, src* – move src to dest w/ 0 extended
MUL – EDX:EAX = EAX * op
PUSHAD / POPAD – push / pop 32 bit registers
PUSHFD / POPFD – push / pop 32 bit EEFLAGS reg
SAHF – store AH into low byte of EEFLAGS
SUB – subtract source from destination
TEST – *test dest, src* – implied AND btwn dest and src (set flags)
XCHG – exchange two operands (*no mem to mem*)
AND clears 1 or more bits in operand without effecting other bits (masking) ALWAYS clears OF/CF. can mod SF, ZF, PF
AND ops must be same size. Reg,reg   reg,mem   reg,imm   mem, reg   mem, imm
OR sets 1 or more bits in an operand without effecting other bits. Always clears CF/OF
NOT will reverse all bits (the complement set)
XOR bool exclusive-or inverts bits with 1.

- Example: 6.2 ≅ 110.001100110011…
- Method:

6 = 110   (Integral part: convert in the usual way)
.2 x 2 = 0.4
.4 x 2 = 0.8   (Fractional part: successive multiplication by 2)
.8 x 2 = 1.6   (Stop when fractional part repeats or size
.6 x 2 = 1.2   is exceeded)
.2 x 2 = 0.4

- 110.0011 0011 $\overline{0011}$

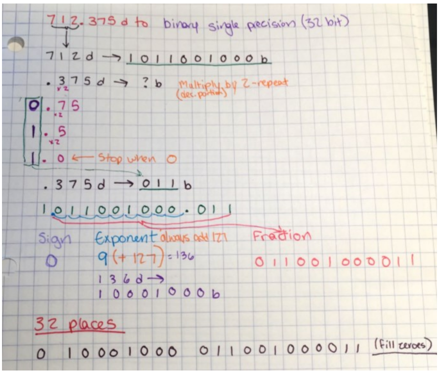
IEEE 754 Standard	Single-precision	Double-precision	Extended
Bits	32	64	80
Sign Bit	1	1	1
“Biased” Exponent	8	11	16
“Normalized” Mantissa	23	52	63

JUMPS

JE – jump if equal
JNE – jump is not equal
JZ – jump is zero
JNZ – jump if not zero
JG (JA) – jump if greater
JGE (JAE) – jump if greater or equal
JL (JB) – jump if less
JLE (JBA) – jump if less or equal
JNG (JNA) – jump of not greater
JNGE (JNAE) – jump if not greater or equal
JNL – jump if not less
JNLE – jump of not less or equal
JO – jump if overflow
JNO – jump if no overflow
JS – jump if sign (= negative)

DATA OPERATORS

OFFSET – returns distance in bytes of label from beginning of enclosing data segment
PTR – override default type of label (mov, ax, WORD PTR mydble)
TYPE – size in bytes of a single element of a data declaration
LENGTHOF – returns the number of items in a single data declaration
SIZEOF – returns value equivalent LENGTHOF * TYPE


712.375 d to binary single precision (32 bit)
712 d → 1011001000 b
. 3 7 5 d → ? b   Multiply by 2-repeat (dec. portion)
0 . 75
1 . 5
1 . 0 ← Stop when 0
. 3 7 5 d → 011 b
1011001000 . 011
Sign   Exponent always add 127   Fraction
0   9 (+ 127) = 136   01100100001 1
136 d → 10001000 b
32 places
0 10001000 0110010000011 (fill zeroes)

- 6.25 in IEEE single precision

6.25 (decimal) = 110.01 (binary)

Move the radix point until a single 1 appears on the **left**, and multiply by the corresponding power of 2

$= 1.1001 \times 2^2$

… so the sign bit is 0 (positive)

… the “biased” exponent is 2 + 127 = 129 = 10000001

… and the “normalized” mantissa is 1001 (drop the 1, and zero-fill).

0 10000001 10010000000000000000000

0100 0000 1100 1000 0000 0000 0000 0000

= 0x40C80000

STACK FRAMES / STACK PARAMETERS

Stack Frame (activation record): Area of stack set aside for args, subroutine return addresses, locals, saved registers.
Steps for creating Stack Frame:

1. Arguments, if any, pushed to stack
2. Subroutine called, return address pushed to stack
3. EBP pushed to stack
4. EBP set equal to ESP (EBP = base reference)
5. If local vars, ESP decremented to reserve space
6. Any saved registers pushed on stack

Access Stack Parameters: [EBP + 8], [EBP + 12], etc…
Uses of stack: pass arguments to subroutine, temp storage for local variables, temp save register values, CALL: CPU saves return address on stack
LOCAL: MASM generates code (ex: DWORD)
push ebp
mov ebp, esp
sub esp, 4   reserves space on stack
Subroutine values received: parameters, passed: arguments (by value, reference (offset))

CALL proc
1) pushes the offset of the next instruction in the calling procedure onto the system stack.(EIP register onto stack)
2) copies the address of the called procedure into EIP
3) Executes the called procedure until RET 4) RET pops the top of the stack into EIP
POP can’t be used with immediate (literal) but you can **PUSH** immediate
PUSH decrements ESP by 4.
POP increments ESP by 4
EIP will point to next instruction when call -> next instruction on stack (esp pointing to address containing it [esp] = next addy) When ret, EIP gets the next addy from the esp esp is decremented.

HAMMING CODES

Parity: sum of one check bit and its selected data bits
number required is **log_2n+1**