WEEK 1

ALU = Arithmetic/Logic Unit

CACHE = an area of fast temporary storage

IP = Instruction Pointer

IR = Instruction Register

Instruction Execution Cycle

1. Fetch next instruction
2. Increment IP to point to next inst
3. Decode instruction in IR
4. If instr requires mem access
   1. Determine Mem Add
   2. Fetch operand from memory into register
5. Execute micro-program for instr
6. Go to step 1

Protected Mode – 4 GB available

Real-address mode – 1 MB

wait state – time delay due to differences between the CPU, system bus, etc.

3 types of buses – Data, Address, Control

Parts of instruction from left to right:

* Label, mnemonic, operand, comment

Declare an array as follows:

Data Types:

|  |  |
| --- | --- |
| Type | Used For |
| BYTE | Character, string, 1-byte int |
| WORD | 2 byte int, address |
| SWORD | 2 byte signed int |
| DWORD | 4 byte unsigned int, addres |
| SDWORD | 4 byte signed int |
| FWORD | 6-byte int |
| QWORD | 8 byte int |
| TBYTE | 10-byte int |
| REAL4 | 4-byte floating-point |
| REAL8 | 8-byte floating-point |
| REAL10 | 10-byte floating-point |

someBytes WORD 42 DUP(0)

It means they are initialized to 0.

Integer Information:

* A signed integers stores the sign in the most significant bit.
* The integer range of ASCII codes is 0 to 127
* 32 bit signed integer range:
* 231 – 1 to -231

**MOVZX = 0 extend move**

**MOVSX = sign-extend move**

Irvine Library:

Clrscr – Clear the screen

* + Pre: none
  + Post: screen cleared and cursor at upper left

Crlf – New line

* + Pre: none
  + Post: cursor is at beg of next line

ReadInt – Reads an integer from keyboard, terminated by Enter key

* + Pre: none
  + Post: value entered is in EAX

ReadString – Reads a string from keyboard, terminated by the Enter key

* + Pre: OFFSET of memory destination in EDX, size of memory destination in ECX
  + Post: String entered is in memory, Length of string entered is in EAX

WriteInt, WriteDec – Writes an integer to the screen

* + Pre: value in EAX
  + Post: value displayed,
  + WriteInt displays +/-

WriteString – Writes a null-terminated string to the screen

* + Pre: OFFSET of memory location in EDX
  + Post: string displayed

WriteChar – Writes a character

* + Pre: mov character to al
  + Post: Character displayed

Constants:

* Two ways to define a constant, however do both before .data
* PI = 3.1416 or PI EQU <3.1416>
* NAME EQU <”Kevin Lewis”, 0>

$ = Current location in data segment

Two’s Complement

* Change every bit to its opposite then at 1 to the result.

Conversion:

* To binary – Divide by 2 until you get 0, remainders are binary code
* To Hex – Divide by 16 until you get 0, remainders are hex code.

16 Bit sign vs unsigned range:

* Unsigned = 0 - 65535
* Signed = -32768 to 32767

Flags:

Carry (CF)

* + Number is larger than the size of the holder. 16 bit number in an 8 bit reg.
  + Or if a negative number is produced on with an unsigned subtraction.
  + INC instruction does not affect it

Overflow (OF)

* + Sum of two numbers with sign bits off yields a result number with the sign bit on.
  + Sum of two number with the sign bits on yields a result number with the sign bit off (doesn’t care if signed or unsigned)

Push and Pop:

Push – decrements the stack pointer and copies the operand into the stack at the location pointed to by the stack pointer.

ESP – points to the last value to be added to or pushed on the top of stack

Linker – Combines object files into an executable file.

WEEK 3:

Big Endian – Bytes ordered from left to right (most significant to least)

Little Endian – Bytes ordered least significant to most significant (left to right)

Floating Point:

* + Decimals in 2-1, 2-2, 2-3, cont..
  + Convert integral part in the usual way
  + Fractional part in successive multiplication by 2, when the remainder (.x) part multiplied by two is greater than 1, record 1.
  + 3 parts
* 1 sign bit
* biased exponent (single: 8 bit, double: 11 bit, extended: 15 bits)
* normalized mantissa (single: 23 bits, double: 52 bits, extended: 64 bits)
* You need to drop the 1 in the mantissa, becomes part of the exponent

Hamming Code:

* Required number of parity bits is log2m + 1

Test: does an AND operation sets CF to zero, SF to MSB and ZF, only if it is zero afterwards

AND = if both are 1 then 1

OR = Either one is One

XOR = they are different

WEEK 4:

CALL

* + pushes the offset of the next instruction in the calling procedure onto the system stack.
  + Copies the address of the called procedure into EIP
  + Executes the called procedure until RET

RET

* + Pops the top of stack into EIP
  + Syntax RET n, n causes n to be added to the stack pointer after EIP is assigned a value (for variables passed to the stack)

PUSH

* + Decrements the stack pointer by 4
  + Actual decrements depends on operand

POP

* + Copies value at ESP into a register or variable

Week 5

Activation record:

* + Area of the stack used for a procedure’s return address, passed parameters, saved registers, and local variables
  + Created by the following steps:
* Calling program pushes arguments onto the stack and calls the procedure
* The called procedure pushes EBP onto the stack, and sets EBP to ESP

Addressing Modes:

* + Register Indirect: Access memory through address in a register
* mov [edx+12], eax
  + Indexed: array name, with “distance” to element in a register
* mov list[edi], eax
  + Base-indexed: starting address in one register, offset in another; add and access memory
* mov eax, [edx + ecx]

Randomize procedure: must be called once at the beginning of the program.

RandomRange – Generates a random number in [0 .. N – 1]

* + Pre: N > 0 in eax
  + Post: random integer in [0 .. N-1] in eax
  + Range = hi – lo + 1

Week 6

OFFSET

* + returns the distance in bytes, of a label from the beginning of its enclosing segment.

PTR

* + Overrides the default type of a label, provides the flexibility to access part of a variable
  + EX:
* myDouble DWORD 12345678h
* mov ax, mydouble -🡪 error
* mov ax, WORD PTR myDouble ->> 5678h
* mov WORD PTR myDouble, 1357h 🡪 saves 1357h

Little Endian order is used when storing data in memory: in memory 78h 56h 34h 12h

mov al, BYTE PTR [myDouble + 1] = 56h

TYPE

* + Returns the size, in bytes, of a single element of a data declaration
  + var1 BYTE
  + move eax, TYPE var1 ;1

LENGTHOF

* + Counts the number of elements in a single data declaration
  + List1 WORD 30 DUP (?) ;30
  + Byte1 BYTE 10, 20, 30 ;3
  + digitStr BYTE “1234567”,0 ;8

SIZEOF

* + operator retuns a value that is equivalent to multiplying LENGTHOF by TYPE

A data declaration spans multiple lines if each line ends with a comma.

mov edx, listD[esi \* TYPE listD]

Note: you can declare a pointer variable that contains the offset of another variable

Example:

List DWORD 100 DUP(?)

Ptr DWORD list ;Contains OFFSET list

Two Dimensional Arrays:

Example:

Matrix DWORD 5 DUP (3 DUP (?)) ; 15 elements

* + An elements address is calculated as the base address plus an offset

BaseAddress + elementSize \*[(row# \* elementsPerRow) + column#]

String Primitives:

lodsb

* + Moves byte at [esi] into the AL register
  + Increments esi if direction flag is 0
  + Decrements esi if the direction flag is 1

stosb

* + Moves byte in the AL register to memory at [edi]
  + Increments edi if direction flag is 0
  + Decrements edi if direction flag is 1

cld

* + Sets the direction flag to 0
  + Causes esi and edi to be incremented by lodsb an stosb
  + Use for moving “forward” through an array

std

* + Sets direction flag to 1
  + Causes esi and edi to be decremented by lodsb and stosb
  + Used for moving “backward” through an array

ReadInt Algorithm:

Get str

X = 0

for k = 0 to (len(str) – 1)

if 48 <= str[k] <= 57

x = 10 \* x + (str[k] – 48)

else

break

Floating Point:

* + Pushdown stack
  + Operations are defined for the “top” one or two registers
  + Registers referenced by name ST(x)
  + ST = ST(0) = top of stack
  + Instruction Format
* OPCODE
* OPCODE destination
* OPCODE destination, source
  + FINIT initialize FPU register stack
  + FLD MemVar
* Push ST(i) “down” to ST(I + 1) for I = 0 .. 6
* Load ST(0) with Mem Var
  + FST MemVar
* Move top of stack to memory
* Leave result in ST(0)
  + FSTP MemVar
* Pop top of stack to memory
* Move ST(i) “up” to ST(i-1) for i=1..7
  + FADD: Addition (pop top two, add, push result)
  + FSUB: Subtraction
  + FMUL: Multiplication
  + FDIV: Division
  + FDIVR: Division (reverses operands)
  + FSIN: Sine (uses radians)
  + FCOS: Cosine (uses radians
  + FSQRT: Square Root
  + FABS: Absolute Value
  + FYL2X: Y\*log2(X) X is in ST(0), Y is in ST(1))
  + FYL2XP1: Y \* log2(X) + 1

Week 7:

* + Procedure
* During assembly, procedure code is translated once
* During execution, control is transferred to the procedure at each call, may be called many times.
  + Macro
* Once defined, it can be invoked one or more times
* During assembly, entire macro code is substituted for each call
* A macro must be defined before it can be invoked

Macroname MACRO [param-1, param-2 ..]

Statement-list

ENDM

mWriteStr Macro buffer

push edx

move dx, OFFSET buffer

call WriteString

pop edx

ENDM

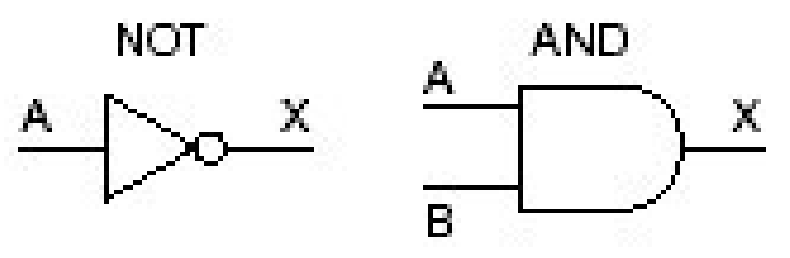
* + Should specify that a label is LOCAL

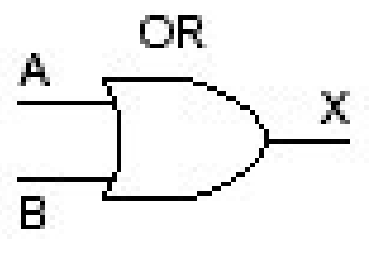
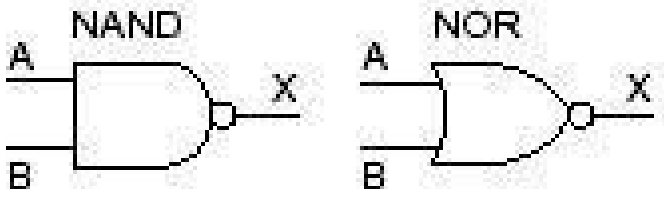
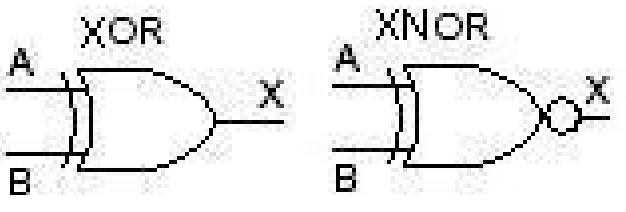
Macro vs Procedure:

* + Macros are very convenient
  + Macros execute faster than procedure
  + Macros are invoked by name
  + If macro is called many times, the assembler produces “fat code”
  + Use a macro for short code that is called “a few” times and uses only a few registers
  + Use a procedure for more complex code or code that is called “many” times

Boolean Expressions

|  |  |  |
| --- | --- | --- |
| Func | Log | Boolean |
| NOT(A) | ~A | A/ |
| AND(A,B) | A AND B | AB |
| OR(A,B) | A OR B | A**+**B |
| XOR(A,B) | A XOR B | A⊕B |
| NAND(A,B) | A NAND B | AB/ |
| NOR(A,B) | A NOR B | A+B/ |
| XNOR(A,B) | A XNOR B | A⊕B |



Function of n binary variables has 2n possible combinations of values for the variables

Week 8:

* + Internal Bus
* Control Unit, ALU, Registers, Addressing Unit communicate via a bus.
* Speed depends on bus width and bus length

Random access memory (RAM)

Read Only Memory (ROM)

* + Clock Cycles
* Near light speed
* Clock cycle length determines CPU speed (mostly)

RISC – Reduced Instruction Set Computer

* + Clock Cycles
  + Instruction executed directly by hardware
  + Only LOAD and STORE instuctions reference memory

**Multiprocessor parallelism** – Shared memory

**Multicomputer Parallelism** – distributed memory

* + Multi-Processor
* Difficult to build
* Relatively Easy to Program
  + Multi-Computer
* Easy to build
* Extremely difficult to program

Amdahl’s Law

Speedup = n / (1 + (n – 1) f)

Total time = f\*T + (1-f)\*T / n

Max speed up = 1/f