**Definitions**

**64bit:** 8 more general purpose registers, 64-bit addresses, supports 256 TB of RAM. **ALU:** Performs arithmetic (+/-) and logic (AND, OR, NOT etc). **Bus [IO, data, control, address]:** Wire that sends signals between parts of the system / Transfers data. [**Control bus** synchronizes actions of all devices attached to system bus] **Caching:** Storing data that might be needed in higher-speed memory in anticipation of use. **Clock**: Synchronizes internal operations of CPU. **Clock Cycle:** 1GHz=1 billion times per second (1 nanosecond clock cycle). **Control Unit:** Coordinates sequencing of steps involved in executing machine instruction. **Directive:** instructions for assembler at compile-time. **Hardware**: Physical components that make up the system. **Instruction:** Executed at runtime. **Linker:** Program that combines object files into executable program. **Software**: Instructions which are executed by hardware. **vonNeumann architecture**: Store programs in memory, execute under control of instruction execution cycle. **Wait State:** Name of time delay in CPU caused by diff between speed of CPU, system bus, memory circuits

**CISC Arch:** *Current machine instruction*: Instruction Register (IR). *Current micro-instruction*: Control Register.

*Protected Mode*: 4GB memory available, separate segments. *Real-Address:*1MiB memory.

**Instruction Execution Cycle:** **1**. CPU fetch the instruction from area of memory called instruction queue + increments the instruction pointer. **2.** decodes instruction by looking at binary bit pattern-might reveal that instruction has operands (input values). **3**. If operands are involved, CPU fetches operands from registers and memory. **4.** executes instruction, using any operand values it fetched. Updates status flags, such as Zero, Carry, and Overflow. **5.**If an output operand was part of instruction, CPU stores result of execution in operand.

**Assemble-Link-Execute Cycle:** 1: text editor to create asci source file. 2: Assembler reads source and produces object file (machine-language translation). 3: Linker reads object and copies any required procedures from the link library, combines them with the object file, and produces the executable file. 4: Loader utility reads executable into memory and branches the CPU to the program’s starting address.

**Metrics:** . B=bytes, b=bits.

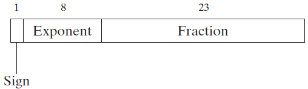
**Registers:** In IA-32, **bus** is 32bits, **gen-purp reg** are 32bits. E[*ABCD*]X, **EBP** (Stack Frame pointer), ESP (Stack Pointer), ESI (Source Index), EDI (Destination Index), EFLAGS, EIP (Instruction Pointer), 16bit segment registers: CS, SS, DS, ES, FS, GS [used in real address mode; indicate base addresses of preassigned memory areas (‘segments’). In protected mode, pointers to segment descriptor tables.

**Status Flags:** **Carry (CF):** set when result of unsigned arithmetic too large to fit into destination. Set when result of an n-bit arithmetic operation sets the (n+1)th bit. **Overflow**(**OF**): Set when signed arithmetic too large or too small to fit. **Sign(SF)** set when result is negative. **Zero(ZF)** when result negative. **Auxilliary Carry (AC)** when carry bit from bit 3 to bit 4 in an 8-bit operand. **Parity(PF)** when least-significant byte in result contains even number of 1 bits – used for error checking.

**Data Types:** Byte: 8bit, WORD: 16bit, DWORD: 32bit, FWORD 48-bit in protected mode, Q word: 64bit, TBYTE: 80-bit, Real4: IEEE short real, REAL8: 64 bit long real, REAL10: 80-bit IEEE extended real.

Maximum unsigned value in n bits is (24bits=.

Range of signed is

**Floating Point Representation: *3 Components***: Sign, Significand, Exponent. : sign is negative, significand is 1.23154, exponent is 5. ***32-bit single precision***: 1 sign, 8 exponent, 23 normalized significand. to approx range. ***64-bit double***: 1, 11, 52. to . ***80-bit double extended*** : 1, 16, 63. to . **Sign:** 1 if neg/0 if pos.

Floating-point binary value 11.1011 is expressed as or . If e is the number of significant bits to the right of the binary point, the decimal denominator is . ***Exponent***: The number’s actual exponent must be added to 127. biased exponent is always positive, between 1 and 254. Actual exponent range is -126 to +127. The range was chosen so the smallest possible exponent’s reciprocal cannot cause an overflow. Normalized: 1.101x=0|01111111|10100000000000000000000 << note that the bits in the significand decimal component are in order, not a converted representation.

Positive/negative infinity – greater/less than any real number. NaN (Not a number) – bit patterns that do not represent any valid real number. *Quiet NaN:* propogate through normal arithmetic. *Signaling NaN:* can be used to signal invalid operation exceptions. Positive 0: all 0’s. Negative 0: 1 followed by all 0’s. Positive infinity is 0 1111111 0’s. Negative infinity is 1 11111111 0’s. QNaN is any x 1111111 1xxx’s, and SNaN is x 11111111 0x’s (and one of the remaining bits must be a 1, otherwise this would be positive infinity).

13.37 - multiply the decimal component (only) by 2:

.37\*2 = 0.74 🡪 0.74\*2 = 1.48 << now drop the 1! 🡪 .48\*2= 0.96 etc. etc.

Proceed until we have filled all 23 bits OR until we get a repeating decimal, then fill in in order: 1101.010111…

To normalize, we move decimal until we have a single 1 on left: 1.101010111… then count the shift (left -> negative 3). 127+(-3) for exponent = 124

So we get 0| 01111100 | 101010111 **Converting other way:**

1 10000011 00001110000000000000000 << 1 means negative, 1000011 is 127+4, rest is decimal

1.00001110000000000000000x2^4

-10000.111 = -16. 0.875 << first convert 10000b to d, then put the negative sign, then ½ +1/4 + 1/8

**Converting Single-Precision Values to Decimal. 1.** If the MSB is 1, the number is negative; otherwise, it is positive. **2.** The next 8 bits represent the exponent. Subtract binary 01111111 (decimal 127), producing the unbiased exponent. Convert the unbiased exponent to decimal. **3.** The next 23 bits represent the significand. Notate a “1.”, followed by the significand bits. Trailing zeros can be ignored. Create a floating-point binary number, using the significand, the sign determined in step 1, and the exponent calculated in step 2. **4.** Denormalize the binary number produced in step 3. (Shift the binary point the number of places equal to the value of the exponent. Shift right if the exponent is positive, or left if the exponent is negative.) **5.** From left to right, use weighted positional notation to form the decimal sum of the powers of 2 represented by the floating-point binary number.

**HAMMING CODE:** add up the bit positions of each of the check bits, that’s where your error is.

First convert the integer to BINARY: 2437d=|\_|\_|1|\_|0|0|1|\_|1|0|0|0|0|1|0|1|

**SYNTAX**

**Identifier Rules:** 1-247 chars, case insensitive, first char must be letter, \_, @, ?, or $. Can use digits later.

**Operands:** May be variables, constants/constant expressions, or registers (depends on the operation)

TITLE Programming Project 1 (Filename)

INCLUDE Irvine32.inc

pressKey EQU <”Press any key to continue…”,0> ; substitutes all between <>

**.data**

newArray DWORD 10,20,30,40,50

BYTE 20 DUP(0) ; 20 bytes, all equal to 0

ArrayCount = ($ - newArray) / 4 ; divide by bytes of the type of variable.

myString BYTE “Hello world”,0 ; make sure to null terminate

prompt BYTE pressKey ; => prompt BYTE “Press any key to continue…”,0

**.code**

**someprocess PROC ; DECLARE A PROCESS**

ret ; RETURN !

**someprocess ENDP ; SIGNALS END**

**main PROC**

move eax, [newArray+4] ; EAX = 20

mov edx, OFFSET myString ; [*label*:] *mnemonic* [*operands*] [;*comment*]

call WriteString

**; If-then:**

cmp eax, ebx ; if eax == ebx

jnz endThen

add eax, ebx ; true code

endThen: ; false code here

**; Unsigned DIVISION DIV**

Mov eax, dividend

mov edx,0 ; extends 0 into EDX for unsigned

Div divisor ; divides edx:eax by operand. Integer result in eax, remainder in edx

**; Signed DIVISION IDIV**

Mov eax, SDividend ;

Cdq ; extends sign in EAX into EDX for signed

**; Unsigned MULTIPLICATION ; 3 versions. Ops must be same size! Result stored in next highest**

MUL *reg/mem8* ; multiplies AL by reg/mem8 MUL reg/mem16 ; ax \* reg/mem16, EAX \* 32bit

**; Signed MULTIPLICATION – 1-operand**

IMUL *reg/mem8* ; **ax** = al \* REG/MEM8mltiplies 8bit AL reg by 8bit operand, stores in 16bit

IMUL *reg/mem16* ; **eax** = ax \* REG/MEM16mltiplies 16bit AX reg by 8bit operand, stores in 32bit

IMUL *reg/mem32* ; **edx:eax** = eax \* REG/MEM32mltiplies 32bit EAX reg by 32bit operand, stores in 64bit

**; Signed MULTIPLICATION – 2-operand Truncates!**

IMUL L*reg16, reg/mem16* ; Lreg16 = L-*reg16* \* R-reg/mem16. OF & CF set if digits lost

**; Signed MULTIPLICATION – 3-operands**

IMUL *reg16, reg/mem16, imm8* ; multiply op2 by op3 and store in op1; sets OF & CF if sigdig’s lost

IMUL *reg16, reg/mem16, imm16* IMUL *reg32, reg/mem32,imm8* IMUL *reg32, reg/mem32,imm32*

**; CLEARING BITS: example, clearing bits 4,3,0 from 8-bit al register:**

And al,11100110b ; and with all 1’s, placing 0 on the bits to clear

; **SETTING BITS: example setting bits**

or AL,00000100b ; set bit 2, leave others unchanged

**main ENDP**

END main ; REQUIRED to tell program the entry point for the program

**SYMBOLIC CONSTANTS**

Symbols use no storage and do not change at runtime. **Equal-Sign Directive:** Associate symbol name with integer expression. Ordinarily 32 bit integer value. EQU directive: *name* EQU *expression* | *symbol* | *<text> .* Useful for values that do not evaluate to an integer. **TEXTEQU:** text macro. <text>, textmacro, %constExpr. Can be redefined at any time.

**MASM INSTRUCTIONS**

**XCHG:** reg,reg | reg,mem | mem,reg.

**ADD**: O\*, S\*, Z\*, A\*, P\*, C\*, D\_, I\_. Valid operands: reg,reg | reg,imm | mem,reg | mem,imm | reg,mem | accum,imm. Source added to dest, sum stored in dest. Must be same size.

**AND**: O\*, S\*, Z\*, A?, P\*, 0. reg,reg | reg,imm | mem,reg | mem,imm | reg,mem | accum,imm.

**CALL**: Push location of next instruction on s tack and transfer to destination location. If procedure is near, only offset of next is pushed, otherwise both segment and offset are pushed.

**CDQ**: Extends sign bit in EAX throughout EDX. No ops. **CLC**: Clear carry. C(0)

**CMP:** OSZAPC\*, DI\_Performs implied subtraction of source from destination. Reg,reg | mem,reg | reg,mem | reg,imm | mem,imm | accum, imm. Destination < SOURCE: ZF 0, CF 1. Destionation > source: ZF 0, CF 0. Destination = Source: ZF 1, CF 0

**DEC:** Subtracts 1 from an operand. Does not affect the Carry flag. OSZAP\*, DIC\_. DEC reg | mem

**DIV:** Unsigned division. If divisor is 8 bits, dividend is AX, Quotient is AL, remainder is AH. 16 bits, Dividend is DX:AX, the Quotient is AX, remainder is DX. 32 bits, Dividend is EDX:EAX, quotient EAX, remainder EDX. Formats: DIV *reg*, DIV *mem*

**IDIV**: Signed int division. Usually IDIV is prefaced by CBW or CWD to sign-extend dividend.

**IMUL**: signed int mult on AL, AX, or EAX. If 8bit multiplier, multiplicand is AL and product is AX. 16 bit, AX -> DX:AX. 32 bits, EAX -> EDX:EAX. Carry and Overflow are set if 16-bit product extends to AH, or 32-bit product extends into DX, or 64 extends into EDX. Formats: (***single op***) IMUL r/mem[8/16/32] : Sign-extends highest bit of lower half of product into upper bits. Stores product in AX/DX/EDX:EAX

(***double op***) r16,r/m16 | r32,r/m32 | r16,imm16 | r16,imm8 | r32, imm8 | r32,imm32.

***3 ops***: r16,r/m16,imm8, OR all 16 OR all 32 OR 32,32,8

**INC**: Add 1 to register. INC reg | INC mem. Does not effect carry! OSZAP\*

**J*cond***: signed: JA/JB & variants, JG, JL, & variants. JZ, JC, JO, JP, JS – jump if flag set

**LAHF:** Load AH from Flags. Flags are copied into AH: SZAPC

**LOOP**: Decrements ECX and jumps to a short label if ECX is not equal to zero. Must be within -128/127 bytes.

**MOV**: both operands must be same size, both cannot be memory operands, and the IP/EIP cannot be destination. Also destination cannot be imm. **MOVZX**: zero extends the destination when an 8 or 16 bit reg/mem is the source. **MOVSX**: Sign extends 🡪if hex, if MSB is >= 7 extends F; binary, extends 1 or 0

**MUL:** Multiplies AL, AX, or EAX by a source operand. If source is 8bits uses AL, 16 uses AX, and 32 bit uses EAX. Results are stored in one tier higher, so AL -> AX, AX->EAX, EAX->EDX:EAX. Can set C and O flags.

**PUSH:**

**POP:** Copies wordor doubleword at current stack pointer location into destination operand and adds 2 (or 4) to (E)SP. Dest can be a reg16/32 or mem16/32 or segreg.

**RET:** Return from procedure. Pops a return address from stock. Can take an 8 bit imm operand to add to the ESP after popping.

**SUB:** Subtracts. Reg,reg | mem,reg | reg,mem | reg,imm | mem,imm | accum, imm

**IRVINE32 LIBRARY**

**Delay**: pause program for miliseconds in EAX. **DumpMem**: Write range of memory in hex. Pass starting addy in ESI, num units in ECX, unit size in EBX. .**isDigit**: Set ZF if AL contains asci code for a decimal digit (0-9)**ReadDec**: Read unsigned 32bit decimal from keyboard, returns in EAX; valid input if CF = 0, 1 signals problem. **ReadInt**: 32bit signed, return in EAX. Optional leading +/- sign; OF set if too big. **Readchar**: read and return single character in the AL register in ASCII format; if extended character pressed, AL set to 0. . **ReadHex**: Read 32bit hex from keyboard, reads up to 8 digits. **ReadFloat**: get keyboard input into ST(0). **ReadKey**: read a single keypress. **ReadString**: Read string until Enter pressed. Pass the offset of a bufferin EDX and set ECX to max # of char’s to enter + 1 to save space for null byte; eax returns number of characters typed. **WriteBin**: Write unsigned 32bit int to console passed in via EAX. **WriteChar**: display a single character stored in AL to console. **WriteDec**: unsigned 32 bit int in decimal with no leading zeros via EAX. **WriteFloat**: display ST(0) contents in floating-point format. **WriteHex**: 32 bit int in hex. B suffix converts to hexadecimal. **WriteInt**: Write 32bit signed integer from EAX to screen with sign. **WriteString**: Write null-terminated string to console window. Move the offset of the string into EDX before calling.

**LITTLE/BIG ENDIAN:**

0xAA 0x8B 0x8C 0x4B BIG- E🡪 AA8B8C4B (Signed/UnSigned) = -1433695157 / 2861272139

0xAA 0x8B 0x8C 0x4B little-e🡪 4B8C8BAA (Signed/UnSigned) = 1267501994 / 126750199

**Address of an Array Element:**

List DWORD 50 DUP(0); a DWORD 25; b DWORD 15; if List is at address 0x0300 then what is the address of the 33rd element? @List[32] is @L + (32 \* TYPE List)

**Filling Array with Random Values in a Range**

.data

loVal DWORD ?

hiVal DWORD ?

randVal DWORD ?

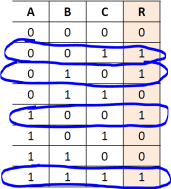
.code

main PROC

call Randomize

; get a loVal and hiVal from use here

Push loVal

Push hiVal

Push offset randVal

Call nextRand

nextRand PROC ; randVal = RandomRange(hiVal – loVal + 1) + loVal

pushad ; or push ebp and other registers needed

mov ebp, esp

mov eax, [ebp+40] ;hiVal in eax

sub eax, [ebp+44] ; subtract loVal

inc eax

call RandomRange ; eax gets value in [0 .. eax-1]

add eax, [ebp+44] ; add the low back in

mov edi, [ebp+36] ; move address of randVal into edit

mov [edi], eax ; move eax into randVal’s address

popad

ret

nextRand ENDP

**ARRAYS**

**OFFSET**: address of a data label. Distance in bytes from beginning of the data segment.

**PTR**: Override declared size of an operand. mov al, BYTE PTR [esi+1] ; if ESI points to 40320b = 0x9D80, convert to little-endian: 80 9D 00 00m then go the the second BYTE. myDouble DWOR 12345678h; (this is 78 56 34 12 little endian, or 5678h 1234h as two-byte chunks. mov ax, WORD PTR myDouble; = 5678h; mov ax, WOR DPTR [myDouble+2] ; 1234h (goes 2 bytes over in the little endian order)

**TYPE**: returns size, in bytes, of a single element of a variable.

**LENGTHOF**: number of elements in array defined by values appearing on same line as label. [Nested DUP will multiply (5 DUP(3 DUP(?)) = 15]. myArray BYTE 10,20,30 \n BYTE 40,50,60 => lengthof myArray = 3; But if you use comma at \n instead and drop second BYTE label then returns 6.

**SIZEOF:** Returns a value equivalent to multiplying LENGTHOF by TYPE.

**Calculating address of an array**: BaseAddress + ElementSize \* [(row # \* elementsPerRow) + column #]

0x0200 matrix DWORD 20 DUP(5 DUP(?)) ; matrx[100] … 20 rows, 5 columns.

0x0390 x DWORD LENGTHOF matrix ; x = 100 (array elements)

0x0394 y DWORD SIZEOF matrix ; y = 400 (bytes used)

The address of matrix[8][2] is = 0x0200 + 4 \* [(8 \* 5) + 2] = 0x0200+(168d=0xA8)=0x2A8

**Strings:**

**Lodsb**: load string byte at [esi] into AL register, inc esi if dir flag 0/dec if 1

**stosb**: store str byte in AL to memory at [edi], increments edi if df=0, dec if =1

**cld**: set dir flag 0, causes esi and edi to be incremented. Move FORWARD

**std**: Set direction flag to 1, causes si and edi to be decremented b ylodsb and stosb. Move backward

**Lower-Level Programming**

All keyboard input is character – digits are 48-57 (‘0’ is 48, ‘1’ is 49…’9’ is 57). ReadInt gets a string of digits and converts digits to their numeric values.

Get str. X = 0. For k=0 to (len(str) – 1), if 48 <= str[k] <= 57 { x = 10 \* x + (str[k] – 48)} else break;

**Reverse Polish Notation**

**Infix -> RPN**: 1. Fully parenthesize infex expression. Parse expression left to right constructing binary tree:

**(** 🡪 go right | **Operand** 🡪 insert | **Operator**(+/-^\*) go up, insert, then go right | **)** go up.

Post-order traversal gives you the RPN: left, right, node (as you pass on right when tracing)

**RPN -> Infix**: Push operands on stack. Hit operator, pop 2 operands and perform the operation, push result onto stack. Single value remaining on stack is value of expression.

**IA-32 Floating Point Processor (FPU)**

Runs in parallel with integer processor. Circuits designed for fast computation on IEEE format numbers. Registers implemented as a pushdown stack. Usually programmed as a 0-address machine. CPU/FPU exchange data through memory. Converts WORD and DWORD to REAL10.

**80-bit registers:** IEEE 754 format. Bit 79[1] = sign bit. Bits 78-64 [15]: biased exponent. Bits 0-63 [64]: Normalized mantissa.

More than 8 values pushed, the “bottom” of the stack will be lost. Operations are defined for the top one or two registers. May be referenced by name ST(*x*). ST(0) – ST(7). ST(0) is top.

**Instruction format**: OPCODE | OPCODE *destination* | OPCODE *destination, source*. One register MUST be ST(0).

**Opcodes: FINIT:** initialize FPU register stack – must be executed before *any other* FPU instructions. **FLD *memVar***: Push ST(i) “down” to ST(i+1) for i=0..6; Load ST(0) with MemVar. **FST *memVar*:** move ***top*** of stack to memVar, 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.

**FSUB, FADD, FMUL, FDIV, FDIVR** (reverses operands), **FSIN** (sine (uses radians), **FCOS, FSQRT, FABS** (absolute value), **FYL2X:** where X is in ST(0) and Y is in ST(1**), FYL2XP1:** . **FISUB** and **FIADD** are for integers.

**Implement G = (A + B x C) / (D - E x F)** in the FPU. RPN equivalent is ABC\*+DEF\*-/

finit fld A fld B fld C fmul fadd fld D fld E fld F fmul fsub fdiv fstp G

**MACROS**

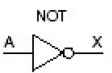
MyMacro MACRO var1, var2

LOCAL test ; create a local label – MASM appends unique integer to end of label once expanded

ENDM

Suppose program requires **2048** bytes of memory. New code will execute **30** times. Macro code requires **48** bytes. If procedure, **128** bytes and each call requires **5** bytes. How many bytes of memory will entire program use if new code added as a macro? 2048 original + (30 expansions \*48 bytes)=3488. How many if added as a procedure? 2048 original + (30 calls \* 5 bytes) + 128 procedure = 2326.

**CIRCUITS AND LOGIC GATES**

Converting a truth table to Boolean: select all rows with result = 1. AND the values in each row using AND for 1 and NOT for 0. OR the resulting terms.