COSC 458-647
Application Software Security

Today

Assembly Language

Basic Instructions

IA32 Instruction Format

- General format:

  ```
  [prefix] instruction operands
  ```

- Prefix used only in String Functions

- Operands represent the direction of operands
  - Single operand instruction: XXX `<src>`
  - Two operand instruction: XXX `<dest>` `<src>`
    - XXX represents the instruction opcode
    - src & dest represent the source and destination operands respectively

Instruction Format

- General format:

  `[prefix] instruction operands`

- Prefix used only in String Functions

- Operands represent the direction of operands
  - Single operand instruction: **`XXX <src>`**
  - Two operand instruction: **`XXX <dest> <src>`**
    - `XXX` represents the instruction opcode
    - **`src`** & **`dest`** represent the source and destination operands respectively

IA32 Instruction Format (cont’d)

- Source operands can be:
  - Register/Memory reference/Immediate value
- Destination operands can be:
  - Register/Memory reference
- Note:
  - The Intel CPU does NOT allow both source and destination operands to be memory references

Instruction	Description
ADD* reg/memory, reg/memory/constant	Adds the two operands and stores the result into the first operand. If there is a result with carry, it will be set in CF.
SUB* reg/memory, reg/memory/constant	Subtracts the second operand from the first and stores the result in the first operand.
AND* reg/memory, reg/memory/constant	Performs the bitwise logical AND operation on the operands and stores the result in the first operand.
OR* reg/memory, reg/memory/constant	Performs the bitwise logical OR operation on the operands and stores the result in the first operand.
XOR* reg/memory, reg/memory/constant	Performs the bitwise logical XOR operation on the operands and stores the result in the first operand. Note that you can not XOR two memory operands.
MUL reg/memory	Multiplies the operand with the Accumulator Register and stores the result in the Accumulator Register.
DIV reg/memory	Divides the Accumulator Register by the operand and stores the result in the Accumulator Register.
INC reg/memory	Increases the value of the operand by 1 and stores the result in the operand.
DEC reg/memory	Decreases the value of the operand by 1 and stores the result in the operand.
NEG reg/memory	Negates the operand and stores the result in the operand.
NOT reg/memory	Performs the bitwise logical NOT operation on the operand and stores the result in the operand.
PUSH reg/memory/constant	Pushes the value of the operand on to the top of the stack.
POP reg/memory	Pops the value of the top item of the stack in to the operand.
MOV* reg/memory, reg/memory/constant	Stores the second operand's value in the first operand.
CMP* reg/memory, reg/memory/constant	Subtracts the second operand from the first operand and sets the respective flags. Usually used in conjunction with a JMP, REP, etc.
JMP** label	Jumps to label.
LEA reg, memory	Takes the offset part of the address of the second operand and stores the result in the first operand.
CALL subroutine	Calls another procedure and leaves control to it until it returns.
RET	Returns to the caller.
INT constant	Calls the interrupt specified by the operand.

Memory References

- Same as a C/C++ pointer access
- Pointer operands appear within square brackets e.g.
  - **`MOV EAX, [0x00040222h]`**

Contents of memory location 0x00040222

  - Can also have register names instead of hex addresses
    - e.g. **`MOV EAX, [EBX]`**

Refers to the contents of register EBX

Memory References (cont’d)

- Control the size of memory accessed by preceding the memory reference with a size:
  - BYTE PTR: byte access
  - WORD PTR: two byte access
  - DWORD PTR: four byte access

E.g. `MOV EAX, BYTE PTR [0x00001234]`

Memory References (cont’d)

- Invalid Memory Accesses
  - Accessing illegal memory
    - CPU generates a general protection fault: (GPF)
  - Access a memory location that does not exist
    - CPU generates a page fault

NOP

- NOP – No operation
  - Takes no arguments
  - Commonly used by the compiler as padding INSIDE functions so as keep them properly aligned

```
var1:  dd      0FFh
str1:  db      “my dog has fleas”,10
var2:  dd      0

; Here are some simple instructions
               mov      eax, [var1] ; notice the brackets
               mov      edx, str1   ; notice the not brackets
               call     dspmsg
               jmp      done
               mov      ebx, [var2] ; this will never happen
               cmp      ecx, 0x8 ; this also will never happen
done:  nop
```

Stack Manipulation Instructions

- PUSH <argument>
  - Pushes a word/double word on the stack
  - Argument can be a register/memory location/immediate value (hardcoded number)
  - E.g. `push eax;  push msg;  push dword 0x9`

- POP <argument>
  - Pop a word/double word from the stack
  - E.g. `pop ecx; pop ebx`

- Note:
  - PUSH decrements the ESP while POP increments the ESP

Stack Manipulation Instructions (cont’d)

- PUSHAD
  - Push (save) all general purpose registers
- POPAD
  - Pop (restore) all general purpose registers
- Avoids long sequence of PUSH/POP instructions to save/restore the registers
  - Used mainly in system code

Example

```
//Swap the EAX and EBX values. The sequence
//gives you an idea of how it could be done.

        PUSH EAX
        PUSH EBX

        POP EAX
        POP EBX
```

Arithmetic Instructions

- ADD <dest> <src>: <dest> ← <dest> + <src>

```
add <reg>,<reg>
add <reg>,<mem>
add <mem>,<reg>
add <reg>,<con>
add <mem>,<con>
```

E.g.
```
add eax, 10                  ;;— EAX ← EAX + 10
add BYTE PTR [var], 10       ;; add 10 to the single byte stored at memory address var
```

- SUB <dest> <src>: <dest> ← <dest> - <src>

```
sub <reg>,  <reg>
sub <reg>,  <mem>
sub <mem>,  <reg>
sub <reg>,  <con>
sub <mem>,  <con>
```

E.g.
```
sub al, ah          ;; AL ← AL - AH
sub eax, 216        ;; subtract 216 from the value stored in EAX
```

INC, DEC - Increment, Decrement

- The inc instruction increments the contents of its operand by one. The dec instruction decrements the contents of its operand by one.
- Syntax

```
inc <reg>
inc <mem>
dec <reg>
dec <mem>
```

- Examples

```
dec eax              ;;subtract one from the contents of EAX.
inc DWORD PTR [var] ;; add one to the 32-bit integer stored at location var
```

Arithmetic Instructions (cont'd)

- DIV/MUL: Unsigned Division/Multiplication
  - Uses the EDX register to store the high bytes of double-word and higher (64 bit) results.
  - EAX stores the low bytes

- IDIV/IMUL: Signed Division/Multiplication
  - IMUL sometimes has 3 operands:
    - IMUL <dest> <src1> <src2>

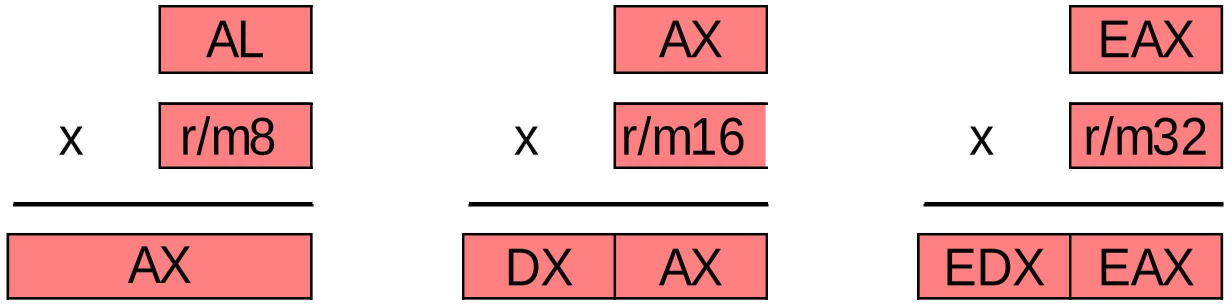
MUL Instruction - (Unsigned Multiply)

- Multiplies an 8-, 16-, or 32-bit operand by either AL, AX or EAX.

```
MUL r/m8
MUL r/m16
MUL r/m32
```

MUL Instruction

- Note that the product is stored in a register (or group of registers) twice the size of the operands.
- The operand can be a register or a memory operand



MUL Examples

```
mov   AL, 5h
mov   BL, 10h
mul   BL
```

AX stores AL*BL ; AX = 0050h, CF = 0

(no overflow - the Carry flag is 0 because the upper half of AX is zero)

MUL Examples

```
.data
  val1  WORD 2000h
  val2  WORD 0100h
.code
  mov  AX, val1
  mul  val2                    ; DX:AX = 00200000h, CF = 1 (CF is 1 because DX is not zero)
```

12345h * 1000h, using 32-bit operands:

```
mov    EAX,    12345h
mov    EBX,    1000h
mul    EBX ;                   ; EDX:EAX = 0000000012345000h, CF=0
```

Your turn

- What will be the hexadecimal values of DX, AX, and the Carry flag after the following instructions execute?

```
mov   AX,     1234h
mov   BX,     100h
mul   BX
```

Read a byte from stdin

```
; read a byte from stdin
      mov    eax, 3 ; sys_read system call
      mov    ebx, 0 ; read from standard input
      mov    ecx, variable ; address to pass to
      mov    edx, 1 ; input length
      int    0x80 ; call the kernel

; write a byte to stdout
      mov    eax, 4 ; sys_write system call
      mov    ebx, 1 ; write to standard output
      mov    ecx, variable ; memory address
      mov    edx, 1 ; output length
      int    0x80 ; call the kernel

; quit the program
      mov    eax, 1 ; sys_exit system call
      mov    ebx, 0 ; sys_exit system call
      int    0x80 ; call the kernel
```

Example

```
//...
int * pInt ;
int iVal ;

// The following instruction sequence is identical to the
// C code: pInt = &iVal ;
  LEA EAX , iVal
  MOV [pInt] , EAX

//..
```

(Example from Debugging Applications by John Robbins)

Another example

```
//....
char szBuff [ MAX_PATH ] ;

// Another example of accessing a pointer through LEA.
// This is identical to the C code:
// GetWindowsDirectory ( szBuff , MAX_PATH ) ;

    PUSH 104h              // Push MAX_PATH as the second parameter.
    LEA  ECX, szBuff       // Get the address of szBuff.
    PUSH ECX               // Push the address of szBuff as the first parameter.
    CALL DWORD PTR             [GetWindowsDirectory]
```

Function Call Instruction

- CALL <argument>
  - argument can be a register \ memory reference \ parameter \ global offset
  - Automatically pushes the return address on the stack and decrements ESP

Function Return Instruction

- RET <optional argument>
  - Argument says how many bytes to pop off the stack (to account for parameters passed to the function)

  - Pops the callers return address off the top of stack and put it in the instruction pointer
    - Return address validity is NOT checked!!!: potential security hazard

Data Manipulation Instructions

- AND <dest> <src> : logical AND
- OR <dest> <src> : logical OR
- NOT <arg>: logical NOT
  - One’s complement negation (Bit Flipping)
- NEG <arg>:
  - Two’s complement negation

Data Manipulation Instructions (cont’d)

- XOR <dest> <src>: logical XOR
  - Fastest way to zero out a register!!!
- INC/DEC <arg> : increment/decrement
  - Often used in speed optimized code (executes in single clock cycle)
  - Directly maps to the C++ operators:
    - ++ : INC
    - -- : DEC

Data Manipulation Instructions (cont’d)

- SHL/SHR <arg> : shift left and Shift right
  - SHL: fastest way to multiply by 2 (C++: <<)
  - SHR: fastest way to divide by 2 (C++: >>)
- MOVZX <dest> <src>: move with zero extend
- MOVSX <dest> <src>: Move with sign extend

Compare Instruction

- CMP <arg1> <arg2>
  - compare arg1 and arg2 and set the appropriate conditional flags in the EFLAGS register

Test Instruction

- TEST <arg1> <arg2> : Bitwise AND of both arguments and sets the appropriate conditional flags
  - PL (SF)
  - ZR (ZF)
  - PE (PF)

Jump Instructions

- JE <label> : Jump if equal
- JL <label> : Jump if less than
- JG <label> : Jump if greater than
- JNE <label> : Jump if not equal to
- JGE <label> : Jump if greater than or equal to
- JLE <label> : Jump if Less than or equal to

Jump Instructions (Cont’d)

- Always follow a CMP/TEST instruction
- JMP condition is always the opposite of the original conditional

Loop Instruction

- Loop <label>
  - Decrement ECX and if ECX isn’t 0, go and re-execute the code sequence marked by <label>

- Rarely used by the VS.NET compiler

Function Calling Conventions

- Specifies how parameters are passed to a function
  - Passed in through stack/registers

- Specifies how stack cleanup occurs upon function return
  - Who performs the cleanup, the caller or the callee?

(Supplied handout has table summarizing the various calling conventions)

Instruction usage examples

- Discuss usage of the previously mentioned instructions
- Generated by VS.NET during compilation
- Examples discussed:
  - Function Entry and Exit
  - Global variable, Local variable and Function parameter access

Function Entry (Prolog)

- Compiler generated at the beginning of a function (before the actual processing code of the function)
- This code sets up the stack for access to the function’s local variables and parameters (the Stack Frame)

Prolog Example

```
// Standard prolog setup
  PUSH EBP              // Save the stack frame register.
  MOV  EBP, ESP         // Set the local function stack frame to ESP.
  SUB  ESP , 20h        // Make room on the stack for 0x20 bytes of local
                        // variables. The SUB instruction appears only if the
                        // function has local variables.
```

Function Exit (Epilog)

- Compiler generated (after the end of the processing code of the function)
- Undoes the operations of the prolog
  - Stack cleanup can be performed here

```
// Standard epilog teardown
MOV ESP , EBP   // Restore the stack value.
POP EBP        // Restore the saved stack frame register.
```

Global Variable Access

- Memory References with a fixed address

e.g.

```
int g_iVal = 0 ;
void AccessGlobalMemory ( void )
{
    __asm
    {
        // Set the global variable to 48,059.
        MOV g_iVal , 0BBBBh

        // If symbols are loaded, the Disassembly window will show:
        // MOV DWORD PTR [g_iVal (4030B4h)],0BBBBh

        // If symbols are NOT loaded, the Disassembly window shows:
        // MOV DWORD PTR [4030B4h],0BBBBh
    }
}
(Example from Debugging Applications by John Robbins)
```

Function Parameter Access

- Function Parameters are positive offsets from EBP (stack frame register)
  - Caller pushes the parameters before calling the function
  - Rule of thumb: “Parameters are positive” -Robbins

Example

```
void AccessParameter ( int iParam )
{
    __asm
    {
        // Move the value if iParam into EAX.
        MOV EAX , iParam

        // If symbols are loaded, the Disassembly window will show:
        // MOV EAX,DWORD PTR [iParam]

       // If symbols are NOT loaded, the Disassembly window shows:
       // MOV EAX,DWORD PTR [EBP+8]
    }
}
```

Caller code pushes iParam onto the stack before calling the function code above:

```
// AccessParameter(0x42);
push     66        ; 00000042H
call     ?AccessParameter@@YAXH@Z  ; AccessParameter
```

Local Variable Access

- Local Variables occur as negative offsets from the EBP (stack frame pointer) register

e.g.

```
void AccessLocalVariable ( void ) {
    int iLocal ;
    __asm {
        // Set the local variable to 23.
        MOV iLocal , 017h

        // If symbols are loaded, the Disassembly window will show:
        // MOV DWORD PTR [iLocal],017h

        // If symbols are NOT loaded, the Disassembly window shows:
        // MOV [EBP-4],017h
    }
}
```

(Example from Debugging Applications By John Robbins)

Improved makefile

```
ASM = nasm –f elf
CC= gcc –o
all: helloWorld
helloWorld: helloWorld.o
                    $(CC) helloWorld helloWorld.o
helloWorld.o: helloWorld.asm
                    $(ASM) helloWorld.asm
```

```
ASM = nasm –f elf
CC= gcc –o
all: helloWorld helloWorld1
helloWorld: helloWorld.o
			$(CC)  helloWorld  helloWorld.o
helloWorld.o: helloWorld.asm
			$(ASM)  helloWorld.asm
helloWorld1: helloWorld1.o
			$(CC)  helloWorld1  helloWorld1.o
helloWorld1.o: helloWorld1.asm
			$(ASM)  helloWorld1.asm
```

Saving registers

- First, C assumes that a subroutine maintains the values of the following registers: EBX, ESI, EDI, EBP, CS, DS, SS, ES.
  - This does not mean that the subroutine can not change them internally. Instead, it means that if it does change their values, it must restore their original values before the subroutine returns. The EBX, ESI and EDI values must be unmodified because C uses these registers for register variables.

- The EBX, ESI and EDI values must be unmodified because C uses these registers for register variables.
  - Usually the stack is used to save the original values of these registers.

Backup slides

Segment Registers

- CS
  - Points to the memory area where your program’s instructions are stored
- DS
  - Points to the memory area where your program’s data is stored
- SS
  - Points to the memory area where your stack is stored
- ES, FS, GS
  - They can be used to point to additional data segments, if necessary

Special Registers

- IP, EIP
  - Instruction pointer, points always to the next instruction that the processor is going to execute
  - Only changed indirectly by branching instructions
- FLAG, EFLAG
  - Flags register, contains individual bits set by different operations (e.g. carry, overflow, zero)
  - Used massively with branch instructions

General Purpose (GP) Registers

- Accumulator (AH, AL, AX, EAX)
  - Accumulates results from mathematical calculations
- Base (BH, BL, BX, EBX)
  - Points to memory locations
- Count (CL, CH, CX, ECX)
  - Counter used typically for loops
  - Can be automatically incremented/decremented
- Data (DL, DH, DX, EDX)
  - Data used in calculations
  - Most significant bits of a 32-bit mul/div operation

CPU State Instructions

- ENTER: save CPU state
- LEAVE: restore CPU state
- Mainly used in interrupt processing

IMUL Instruction - (Signed Multiply)

- Has the same syntax and uses the same operands as the MUL instruction except that it preserves the sign of the product.

IMUL Instruction

IMUL sets the Carry and Overflow flags if the high-order product is not a sign extension of the low-order product.

```
mov   al, 48
mov   bl, 4
imul   bl                  ;AX = 00C0h,  OF = 1
```

AH is not a sign extension of AL, so the Overflow flag is set.

IMUL Instruction

```
mul   al, -4
mov   bl, 4
imul  bl        ; AX = FFF0h, OF = 0
```

AH is a sign extension of AL (the signed result fits within AL), so the Overflow flag is clear.

DIV Instruction - (Unsigned Divide)

- Performs 8-, 16-, and 32-bit division on unsigned integers.

  DIV *r/m8*

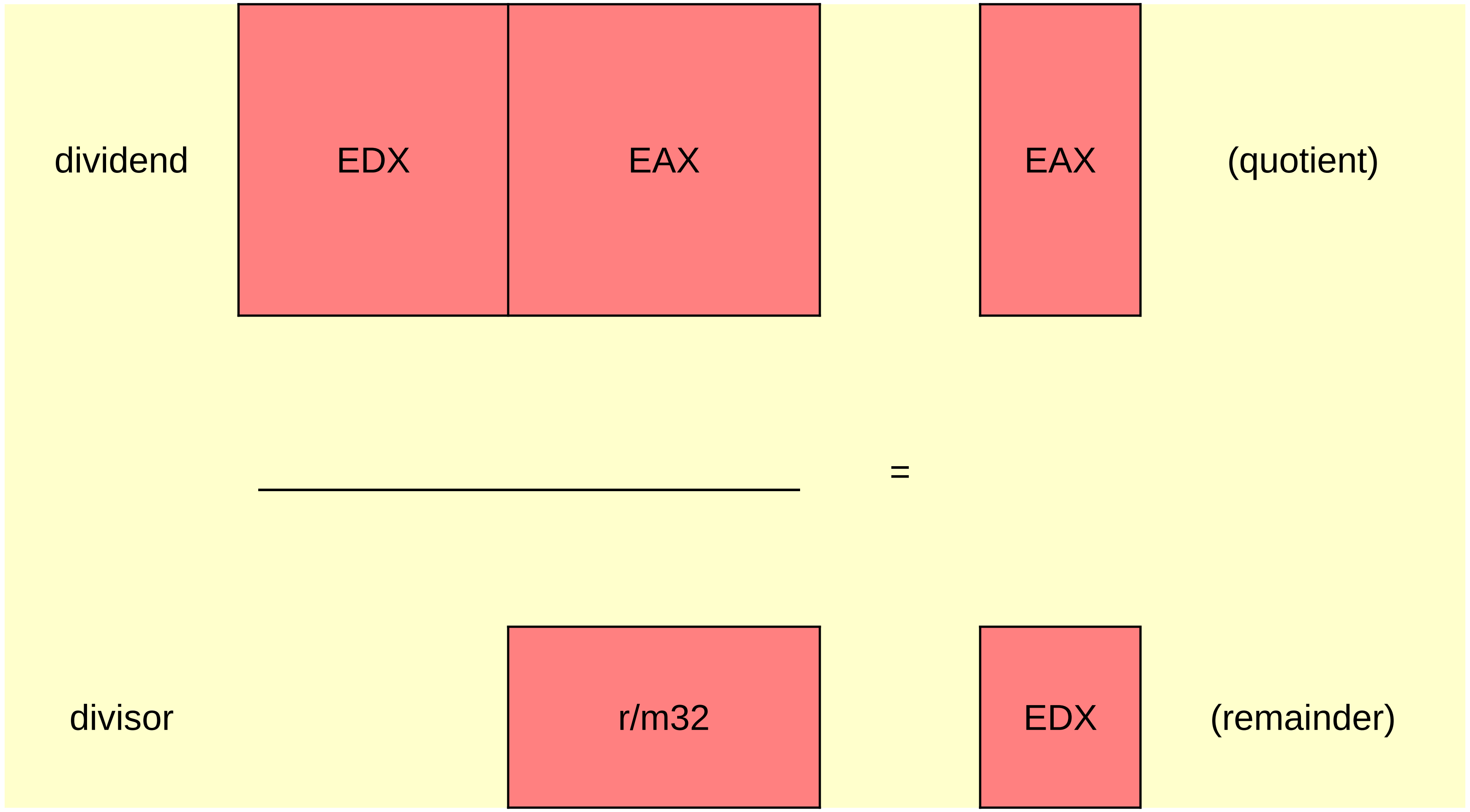
  DIV *r/m16*

  DIV *r/m32*

DIV Instruction

Dividend	**Divisor**	**Quotient**	**Remainder**
AX	*r/m8*	AL	AH
DX:AX	*r/m16*	AX	DX
EDX:EAX	*r/m32*	EAX	EDX

DIV Instruction



DIV Examples - 8-bit Unsigned Division

```
mov    ax,0083h              ; dividend
mov    bl, 2h                ; divisor
div    bl                    ; AL = 41h, AH = 01h
```

Quotient is 41h, remainder is 1

DIV Examples

```
mov   dx, 0           ;clear dividend, high
mov   ax, 8003h       ;dividend, low
mov   cx, 100h        ;divisor
div   cx              ;ax = 0080h, dx = 0003h
```

Quotient = 80h, remainder = 3

Signed Integer Division

- CBW – convert Byte to word
    - Extends the sign bit of AL into the AH register

```
.data
Byteval        SBYTE -65
.code
   mov al,byteval              ; AL = 9Bh
   cbw                         ; AX = FF9Bh
```

Sign Extension Instructions

- CBW
  - -Convert byte to word
- CWD
  - Convert word to double
- CDQ
  - -Convert double to quadword

IDIV Instruction (Signed Division)

- Performs signed integer division, using the same operands as the DIV instruction
- The dividend must be sign-extended into the high order register before IDIV executes.

IDIV Examples

```
.data
Byteval        SBYTE -48
.code
  mov al, byteval        ;dividend
  cbw                    ;extend AL into AH
  mov bl, 5              ;divisor
  idiv  bl               ;AL = -9, AH = -3
```

Divide Overflow

- If the quotient is too large to fit into the destination operand, a divide overflow results. This causes a CPU interrupt, and the current program halts.

```
Mov   ax, 1000h
Mov   bl, 10h
Div   bl                    ;AL cannot hold 100h
```

DIV Overflow

Can use 16-bit divisor to reduce the possibility of divide overflow.

```
Mov   ax, 1000h
Mov   dx, 0                 ;clear DX
Mov   bx, 10h
Div   bx                    ;AX = 0100h
```

HelloASM.asm

- File: included in the Assembly.zip on BB
- Compile

  1. nasm –f elf helloASM.asm
  2. ld –o helloASM_run helloASM.o

To execute

3. ./helloASM_run

Pointer Manipulation Instructions

- LEA: Load Effective Address
  - LEA <dest> <src>
  - Loads the destination register with the address of the source operand
  - Used to emulate pointer access

IA32 Instruction Format (cont’d)

- Source operands can be:
  - Register/Memory reference/Immediate value

- Destination operands can be:
  - Register/Memory reference

- Note:
  - The Intel CPU does NOT allow both source and destination operands to be memory references

Instruction	Description
ADD* reg/memory, reg/memory/constant	Adds the two operands and stores the result into the first operand. If there is a result with carry, it will be set in CF.
SUB* reg/memory, reg/memory/constant	Subtracts the second operand from the first and stores the result in the first operand.
AND* reg/memory, reg/memory/constant	Performs the bitwise logical AND operation on the operands and stores the result in the first operand.
OR* reg/memory, reg/memory/constant	Performs the bitwise logical OR operation on the operands and stores the result in the first operand.
XOR* reg/memory, reg/memory/constant	Performs the bitwise logical XOR operation on the operands and stores the result in the first operand. Note that you can not XOR two memory operands.
MUL reg/memory	Multiplies the operand with the Accumulator Register and stores the result in the Accumulator Register.
DIV reg/memory	Divides the Accumulator Register by the operand and stores the result in the Accumulator Register.
INC reg/memory	Increases the value of the operand by 1 and stores the result in the operand.
DEC reg/memory	Decreases the value of the operand by 1 and stores the result in the operand.
NEG reg/memory	Negates the operand and stores the result in the operand.
NOT reg/memory	Performs the bitwise logical NOT operation on the operand and stores the result in the operand.
PUSH reg/memory/constant	Pushes the value of the operand on to the top of the stack.
POP reg/memory	Pops the value of the top item of the stack in to the operand.
MOV* reg/memory, reg/memory/constant	Stores the second operand's value in the first operand.
CMP* reg/memory, reg/memory/constant	Subtracts the second operand from the first operand and sets the respective flags. Usually used in conjunction with a JMP, REP, etc.
JMP** label	Jumps to label.
LEA reg, memory	Takes the offset part of the address of the second operand and stores the result in the first operand.
CALL subroutine	Calls another procedure and leaves control to it until it returns.
RET	Returns to the caller.
INT constant	Calls the interrupt specified by the operand.

Memory References

- Same as a C/C++ pointer access
- Pointer operands appear within square brackets e.g.
  - `MOV EAX, [0x00040222h]`

Contents of memory location 0x00040222

  - Can also have register names instead of hex addresses
    - e.g. `MOV EAX, [EBX]`

Refers to the contents of register EBX

Memory References (cont'd)

- Control the size of memory accessed by preceding the memory reference with a size:
  - BYTE PTR: byte access
  - WORD PTR: two byte access
  - DWORD PTR: four byte access

E.g. `MOV EAX, BYTE PTR [0x00001234]`

Memory References (cont’d)

- Invalid Memory Accesses
  - Accessing illegal memory
    - CPU generates a general protection fault: (GPF)
  - Access a memory location that does not exist
    - CPU generates a page fault

NOP

- NOP – No operation
  - Takes no arguments
  - Commonly used by the compiler as padding INSIDE functions so as keep them properly aligned

```
var1:  dd      0FFh
str1:  db      “my dog has fleas”,10
var2:  dd      0

; Here are some simple instructions
               mov     eax, [var1] ; notice the brackets
               mov     edx, str1   ; notice the not brackets
               call    dspmsg
               jmp     done
               mov     ebx, [var2] ; this will never happen
               cmp     ecx, 0x8 ; this also will never happen

done:  nop
```

Stack Manipulation Instructions

- PUSH <argument>
  - Pushes a word/double word on the stack
  - argument can be a register/memory location/immediate value (hardcoded number)
- POP <argument>
  - Pop a word/double word from the stack
- Note:
  - PUSH decrements the ESP while POP increments the ESP

Stack Manipulation Instructions (cont’d)

- PUSHAD
  - Push (save) all general purpose registers
- POPAD
  - Pop (restore) all general purpose registers
- Avoids long sequence of PUSH/POP instructions to save/restore the registers
  - Used mainly in system code

Example

```
//Swap the EAX and EBX values. The sequence
//gives you an idea of how it could be done.
      PUSH EAX
      PUSH EBX

      POP EAX
      POP EBX
```

Arithmetic Instructions

- ADD <dest> <src>
  - <dest> ← <dest> + <src>

- SUB <dest> <src>
  - <dest> ← <dest> - <src>

- The result is stored in the destination and the original value in destination is overwritten.

Arithmetic Instructions (cont’d)

- DIV/MUL: Unsigned Division/Multiplication
  - Uses the EDX register to store the high bytes of double-word and higher (64 bit) results.
  - EAX stores the low bytes
- IDIV/IMUL: Signed Division/Multiplication
  - IMUL sometimes has 3 operands:
    - IMUL <dest> <src1> <src2>

Pointer Manipulation Instructions

- LEA: Load Effective Address
  - LEA <dest> <src>
  - Loads the destination register with the address of the source operand
  - Used to emulate pointer access

Example

```
//...
int * pInt ;
int iVal ;

// The following instruction sequence is identical to the
// C code: pInt = &iVal ;
  LEA EAX , iVal
  MOV [pInt] , EAX

//..
```

(Example from Debugging Applications by John Robbins)

Another example

```
//....
char szBuff [ MAX_PATH ] ;

// Another example of accessing a pointer through LEA.
// This is identical to the C code:
// GetWindowsDirectory ( szBuff , MAX_PATH ) ;

   PUSH 104h          // Push MAX_PATH as the second parameter.
   LEA  ECX, szBuff   // Get the address of szBuff.
   PUSH ECX           // Push the address of szBuff as the first parameter.
   CALL DWORD PTR [GetWindowsDirectory]
```

Function Call Instruction

- CALL <argument>
  - argument can be a register \ memory reference \ parameter \ global offset
  - Automatically pushes the return address on the stack and decrements ESP

Function Return Instruction

- RET <optional argument>
  - Argument says how many bytes to pop off the stack (to account for parameters passed to the function)
  - Pops the callers return address off the top of stack and put it in the instruction pointer
    - Return address validity is NOT checked!!!: potential security hazard

Data Manipulation Instructions

- AND <dest> <src> : logical AND
- OR <dest> <src> : logical OR
- NOT <arg>: logical NOT
  - One’s complement negation (Bit Flipping)
- NEG <arg>:
  - Two’s complement negation

Data Manipulation Instructions (cont’d)

- XOR <dest> <src>: logical XOR
  - Fastest way to zero out a register!!!

- INC/DEC <arg> : increment/decrement
  - Often used in speed optimized code (executes in single clock cycle)
  - Directly maps to the C++ operators:
    - ++ : INC
    - -- : DEC

Data Manipulation Instructions (cont'd)

- SHL/SHR <arg> : shift left and Shift right
  - SHL: fastest way to multiply by 2 (C++: <<)
  - SHR: fastest way to divide by 2 (C++: >>)
- MOVZX <dest> <src>: move with zero extend
- MOVSX <dest> <src>: Move with sign extend

Compare Instruction

- CMP <arg1> <arg2>
  - compare arg1 and arg2 and set the appropriate conditional flags in the EFLAGS register

Test Instruction

- TEST <arg1> <arg2> : Bitwise AND of both arguments and sets the appropriate conditional flags
  - PL (SF)
  - ZR (ZF)
  - PE (PF)

Jump Instructions

- JE <label> : Jump if equal
- JL <label> : Jump if less than
- JG <label> : Jump if greater than
- JNE <label> : Jump if not equal to
- JGE <label> : Jump if greater than or equal to
- JLE <label> : Jump if Less than or equal to

Jump Instructions (Cont’d)

- Always follow a CMP/TEST instruction
- JMP condition is always the opposite of the original conditional

Loop Instruction

- Loop <label>
  - Decrement ECX and if ECX isn’t 0, go and re-execute the code sequence marked by <label>

- Rarely used by the VS.NET compiler

Function Calling Conventions

- Specifies how parameters are passed to a function
  - Passed in through stack/registers
- Specifies how stack cleanup occurs upon function return
  - Who performs the cleanup, the caller or the callee?

(Supplied handout has table summarizing the various calling conventions)

Instruction usage examples

- Discuss usage of the previously mentioned instructions
- Generated by VS.NET during compilation
- Examples discussed:
  - Function Entry and Exit
  - Global variable, Local variable and Function parameter access

Function Entry (Prolog)

- Compiler generated at the beginning of a function (before the actual processing code of the function)
- This code sets up the stack for access to the function’s local variables and parameters (the Stack Frame)

Prolog Example

```
{
  // Standard prolog setup
  PUSH EBP          // Save the stack frame register.
  MOV  EBP, ESP     // Set the local function stack frame to ESP.
  SUB  ESP , 20h    // Make room on the stack for 0x20 bytes of local
                    // variables. The SUB instruction appears only if the
                    // function has local variables.
}
```

Function Exit (Epilog)

- Compiler generated (after the end of the processing code of the function)
- Undoes the operations of the prolog
  - Stack cleanup can be performed here

```
// Standard epilog teardown
MOV ESP , EBP   // Restore the stack value.
POP EBP      // Restore the saved stack frame register.
```

Global Variable Access

- Memory References with a fixed address

e.g.

```
int g_iVal = 0 ;
void AccessGlobalMemory ( void )
{
    __asm
    {
        // Set the global variable to 48,059.
        MOV g_iVal , 0BBBBh

        // If symbols are loaded, the Disassembly window will show:
        // MOV DWORD PTR [g_iVal (4030B4h)],0BBBBh

        // If symbols are NOT loaded, the Disassembly window shows:
        // MOV DWORD PTR [4030B4h],0BBBBh
    }
}
(Example from Debugging Applications by John Robbins)
```

Function Parameter Access

- Function Parameters are positive offsets from EBP (stack frame register)
  - Caller pushes the parameters before calling the function
  - Rule of thumb: “Parameters are positive” -Robbins

Example

```
void AccessParameter ( int iParam )
{
    __asm
    {
        // Move the value if iParam into EAX.
        MOV EAX , iParam

        // If symbols are loaded, the Disassembly window will show:
        // MOV EAX,DWORD PTR [iParam]

       // If symbols are NOT loaded, the Disassembly window shows:
       // MOV EAX,DWORD PTR [EBP+8]
    }
}
```

Caller code pushes iParam onto the stack before calling the function code above:

```
// AccessParameter(0x42);
push     66        ; 00000042H
call     ?AccessParameter@@YAXH@Z  ; AccessParameter
```

Local Variable Access

- Local Variables occur as negative offsets from the EBP (stack frame pointer) register

e.g.

```
void AccessLocalVariable ( void ) {
    int iLocal ;
    __asm {
        // Set the local variable to 23.
        MOV iLocal , 017h

        // If symbols are loaded, the Disassembly window will show:
        // MOV DWORD PTR [iLocal],017h

        // If symbols are NOT loaded, the Disassembly window shows:
        // MOV [EBP-4],017h
    }
}
(Example from Debugging Applications By John Robbins)
```

helloWorld.asm

```
;; These externals are all from the standard C library:
extern printf       ; Notify linker that we're calling printf

[SECTION .data]     ; Section containing initialised data
    msg db “Hello, world!”, 0xa, 0    ; our dear string

[SECTION .text]
    global          main      ; Required so linker can find entry point for C
main:
        push     msg
        call     printf
        add      esp, byte 4          ; Clean up stack after printf call
return:
        mov eax, 0
        ret                   ; Return control to Linux
```

Make file

```
helloWorld: helloWorld.o
	gcc -o helloWorld helloWorld.o
helloWorld.o: helloWorld.asm
	nasm -f elf helloWorld.asm
```

The -f Option: Specifying the Output File Format

press "tab" to input it

Improved makefile

```
ASM = nasm –f elf
CC= gcc –o
all: helloWorld
helloWorld: helloWorld.o
                $(CC) helloWorld helloWorld.o
helloWorld.o: helloWorld.asm
                $(ASM) helloWorld.asm
```

helloWorld1.asm – improved helloWorld.asm

```
    extern printf

[SECTION .data]
                msg db “Hello, world!”, 0xa,0;our dear string

[SECTION .text]
                global    main
main:
                push ebp
                mov   ebp,esp
                push dword msg
                call printf
                add esp, byte 4
return:
                mov esp,ebp          ; Destroy stack frame before returning
                pop ebp
                mov eax,0
                ret
```

```
ASM = nasm –f elf
CC= gcc –o
all: helloWorld helloWorld1
helloWorld: helloWorld.o
			$(CC)  helloWorld  helloWorld.o
helloWorld.o: helloWorld.asm
			$(ASM)  helloWorld.asm
helloWorld1: helloWorld1.o
			$(CC)  helloWorld1  helloWorld1.o
helloWorld1.o: helloWorld1.asm
			$(ASM)  helloWorld1.asm
```

Saving registers

- First, C assumes that a subroutine maintains the values of the following registers: EBX, ESI, EDI, EBP, CS, DS, SS, ES.
  - This does not mean that the subroutine can not change them internally. Instead, it means that if it does change their values, it must restore their original values before the subroutine returns. The EBX, ESI and EDI values must be unmodified because C uses these registers for register variables.

- The EBX, ESI and EDI values must be unmodified because C uses these registers for register variables.
  - Usually the stack is used to save the original values of these registers.

Backup slides

Segment Registers

- CS
  - Points to the memory area where your program’s instructions are stored
- DS
  - Points to the memory area where your program’s data is stored
- SS
  - Points to the memory area where your stack is stored
- ES, FS, GS
  - They can be used to point to additional data segments, if necessary

Special Registers

- IP, EIP
  - Instruction pointer, points always to the next instruction that the processor is going to execute
  - Only changed indirectly by branching instructions
- FLAG, EFLAG
  - Flags register, contains individual bits set by different operations (e.g. carry, overflow, zero)
  - Used massively with branch instructions

General Purpose (GP) Registers

- Accumulator (AH, AL, AX, EAX)
  - Accumulates results from mathematical calculations
- Base (BH, BL, BX, EBX)
  - Points to memory locations
- Count (CL, CH, CX, ECX)
  - Counter used typically for loops
  - Can be automatically incremented/decremented
- Data (DL, DH, DX, EDX)
  - Data used in calculations
  - Most significant bits of a 32-bit mul/div operation

CPU State Instructions

- ENTER: save CPU state
- LEAVE: restore CPU state
- Mainly used in interrupt processing