ECE291

Lecture 4

Jumping in head first

Lecture outline

- Multiplication and Division
- Program control instructions
- Unconditional jumps
- Conditional branching
- Numerical comparison
- Looping
- High-level program constructs

Multiplication

- The product after a multiplication is always a double-width product
  - if we multiply two 16-bit numbers, they generate a 32-bit product
  - unsigned: $(2^{16} - 1) * (2^{16} - 1) = 2^{32} - 2 * 2^{16} + 1 < (2^{32} - 1)$
  - signed: $(-2^{15}) * (-2^{15}) = 2^{30} < (2^{31} - 1)$
  - overflow cannot occur
- Modification of Flags
  - Most flags are undefined after multiplication
  - **O** and **C** flags clear to 0 if the result fit into half-size register
  - if the most significant 16 bits of the product are 0, both flags **C** and **O** clear to 0

Multiplication

- Two different instructions for multiplication
  - MUL Unsigned integer multiplication
  - IMUL Singed integer multiplication
- MUL Multiplication is performed on bytes, words, or doubles
- Size of the multiplier determines the operation
- The multiplier can be any register or any memory location
- For MUL, the multiplicand is always AL, AX or EAX

```
mul    cx              ; AX * CX (unsigned result in DX:AX);
imul   word [si]       ; AX * [word content of memory location
                       ; addressed by SI] (signed product in  DX:AX)
```

Unsigned multiplication

- **16-bit multiplication**

Multiplicand	AX
Multiplier	16-bit register or 16-bit memory variable
Product	High word in DX, Low word in AX

Unsigned multiplication

- **8-bit multiplication**
  - Multiplicand AL
  - Multiplier 8-bit register or 8-bit memory variable
  - Product AX
- **32-bit multiplication**
  - Multiplicand EAX
  - Multiplier 32-bit register or 32-bit memory variable)
  - Product High word in EDX : Low word in EAX)

Signed multiplication

- Not available in 8088 and 8086
- For 2's Complement signed integers only
- Four forms
  - imul reg/mem – assumes AL, AX or EAX
  - imul reg, reg/mem
  - imul reg, immediate data
  - imul reg, reg/mem, immediate data

Signed multiplication

- Examples
  - imul bl ; BL * AL → AX
  - imul bx ; BX * AX → DX:AX
  - imul cl, [bl] ; CL * [DS:BL] → CL
    ; overflows can occur
  - imul cx, dx, 12h ; 12h * DX → CX

Binary Multiplication

- Long multiplication is done through shifts and additions

```
          0 1 1 0 0 0 1 0    (98)
        x 0 0 1 0 0 1 0 1    (37)
    ---------------------

          0 1 1 0 0 0 1 0
        0 1 1 0 0 0 1 0 - -
    0 1 1 0 0 0 1 0 -- -- -  (3626)
```

- This works if both numbers are positive
- To multiply negative numbers, the CPU will store the sign bits of the numbers, make both numbers positive, compute the result, then negate the result if necessary

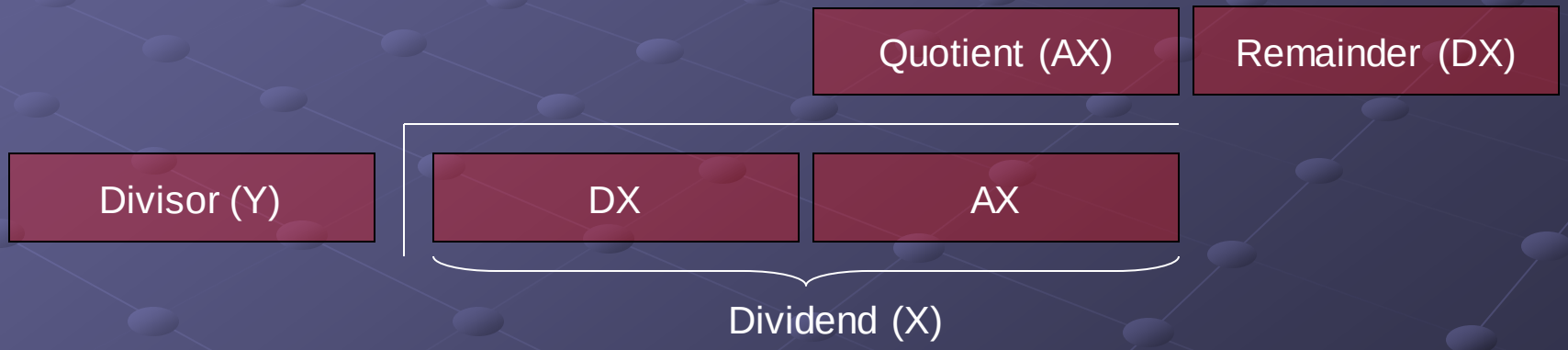
Division

- Two different instructions for division
  - DIV Unsigned division
  - IDIV Signed division (2's complement)
- Division is performed on bytes, words, or double words
- the size of the divisor determines the operation
- The dividend is always a double-width dividend that is divided by the operand (divisor)
- The divisor can be any register or any memory location

Division

32-bit/16-bit division - the use of the AX (and DX) registers is implied

Dividend	high word in DX, low word in AX
Divisor	16-bit register or 16-bit memory variable
Quotient	AX
Remainder	DX



Division

- **16-bit/8-bit**
  - Dividend AX
  - Divisor 8-bit register or 8-bit memory variable
  - Quotient AL
  - Remainder AH
- **64-bit/32-bit**
  - Dividend high double word in EDX, low double word in EAX
  - Divisor 32-bit register or 32-bit memory variable)
  - Quotient EAX
  - Remainder EDX

Division

- Division of two equally sized words
  - *Unsigned numbers:* move zero into high order-word
  - *Signed numbers:* use signed extension to fill high-word with ones or zeros
  - **CBW** (convert byte to word)

    AX = xxxx xxxx snnn nnnn (before)

    AX = ssss ssss snnn nnnn (after)
  - **CWD** (convert word to double)

    DX:AX = xxxx xxxx xxxx xxxx snnn nnnn nnnn nnnn (before)

    DX:AX = ssss ssss ssss ssss snnn nnnn nnnn nnnn (after)
  - **CWDE** (convert double to double-word extended)

Division

- Flag settings
  - none of the flag bits change predictably for a division
- A division can result in two types of errors
  - divide by zero
  - divide overflow (a small number divides into a large number), e.g., 3000 / 2
    - AX = 3000
    - Devisor is 2 => 8 bit division is performed
    - Quotient will be written to AL => but 1500 does not fit into AL
    - consequently we have divide overflow
    - in both cases microprocessor generates interrupt (interrupts are covered later in this course)

Example

Division of the byte contents of memory NUMB by the contents of NUMB1

Unsigned

```
MOV AL, [NUMB]      ;get NUMB
MOV AH, 0           ;zero extend
DIV byte [NUMB1]
MOV [ANSQ], AL      ;save quotient
MOV [ANSR], AH      ;save remainder
```

Signed

```
MOV AL, [NUMB]      ;get NUMB
CBW                 ;signed-extend
IDIV byte [NUMB1]
MOV [ANSQ], AL      ;save quotient
MOV [ANSR], AH      ;save remainder
```

Division

- What do we do with remainder after division?
  - use the remainder to round the result
  - drop the remainder to truncate the result
  - if the division is unsigned, rounding requires that double the remainder is compared with the divisor to decide whether to round up the quotient
  - Example, sequence of instructions that divide AX by BL and round the result

```
        DIV     BL
        ADD     AH, AH  ;double remainder
        CMP     AH, BL  ;test for rounding
        JB      .NEXT
        INC     AL
.NEXT
```

Unconditional jumps

- Basically an instruction that causes execution to transfer to some point other than the next sequential instruction in your code.
- Essentially a “goto” statement.
- Forget all the nasty things you heard about using “goto’s” since you were a kid programming Quick Basic in middle school.
- Assembly language couldn’t exist without unconditional jumps.

Unconditional jumps

- **Short jump** – 2-byte instruction that allows jumps to memory locations within +127 and -128 bytes from the memory location following the jump statement.

  JMP SHORT *label*

- **Near jump** – 3-byte instruction that allows jumps within +/- 32KB from the instruction in the current code segment

  JMP *label*

- **Far jump** – 5-byte instruction that allows a jump to any memory location within the entire memory space

  JMP *label*

- In protected mode, the near jump and far jump are the same

Conditional jumps

- Logic and arithmetic instructions set flags
- Flags provide state information from previous instructions
- Using flags we can perform conditional jumping. For example, transfer program execution to some different place within the program
- if condition is true
  - jump back or forward in a code to the location specified
  - instruction pointer (IP) gets updated to point to the instruction to which the program will jump
- if condition is false
  - continue execution at the following instruction
  - IP gets incremented as usual

Conditional jumps

- Take the form J + some condition or combination of conditions
- L, G, A, B, Z, E, S, C, N are some of the more common conditions
- We'll have a closer look in a few more slides

Conditional jumps

- Conditional jumps are always short jumps in the 8086-80286
  - the range of the jump is +127 bytes and -128 bytes from the location following the conditional jump
- In the 80386 and above, conditional jumps are either short or near jumps
- If you want to do a near jump you *must* include the NEAR keyword after the jump instruction and before the label
- Conditional jumps test: sign (S), zero (Z), carry (C), parity (P), and overflow (O)
- An FFh is **above** 00h in the set of unsigned numbers
- An FFh (-1) is **less** than 00h for signed numbers

Numerical comparison

- **CMP A, B** compares A to B
  - a subtraction that *only changes the flag bits*
  - useful for checking the entire contents of a register or a memory location against another value
  - usually followed by a conditional jump instruction

```
CMP AL, 10h   ;compare with 10h (contents of AL do not change)
JAE foo       ;if 10h or above then jump to memory location foo
```

- **SUB** A,B calculates difference A - B
  - saves result to A and sets flags

Numerical comparison

- CMP, SUB instructions always set flags the same way regardless if you are interpreting the operands as signed or unsigned.
- There are different flags you look at depending on what kind of data you're comparing

Numerical comparison

CMP A, B

Unsigned operands

Z: equality/inequality

C: A < B (C = 1)

A > B (C = 0)

S: No meaning

O: No meaning

Signed operands

Z: equality/inequality

C: No meaning

S and 0 together:

if S xor O = 1 then

A < B

else

A > B

Signed comparisons

- Consider CMP AX,BX computed by the CPU
  - The Sign bit (S) will be set if the result of AX-BX has a 1 at the most significant bit of the result
  - The Overflow flag (O) will be set if the result of AX-BX produced a number that was out of range [-32768, 32767].
- Case 1: Sign = 1 and Overflow = 0
  - AX-BX looks negative, and
  - No overflow so result was within range and is valid
  - Therefore, AX must be less than BX

Signed comparisons

- Case 2: Sign = 0 and Overflow = 1
  - AX-BX looks positive, but
  - Overflow is set, so result is out of range so the sign bit is wrong
  - Therefore AX must still be less than BX
- Case 3: Sign = 0 and Overflow = 0
  - AX-BX is positive, and
  - There was no overflow, meaning sign bit is correct
  - So AX must be greater than BX
- Case 4: Sign = 1 and Overflow = 1
  - AX-BX looks negative, but
  - There was an overflow so the sign is incorrect
  - AX is actually greater than BX

Signed comparisons

- Difference in JS (jump on sign) and JL (jump less than)
  - JS looks at the sign bit (S) of the last compare (or subtraction) only. If S = 1 then jump.
  - JL looks at S *XOR* O of the last compare
    - REGARDLESS of the value AX-BX, i.e., even if AX-BX causes overflow, the JL will be correctly executed

Signed comparisons

- JL is true if the condition: S xor O is met
- JL is true for two conditions:
- S=1, O=0: (AX-BX) was negative and (AX-BX) did not overflow
  - Example (8-bit): CMP -5, 2

    (-5) - (2) = (-7)

    Result (-7) has the sign bit set

    Thus (-5) is less than (2).

Signed comparisons

- S=0, O=1: **Overflow!, Sign bit of the result is wrong!**
- Consider the following case:
  - AX is a large negative number (-)
  - BX is a positive number (+).
  - The subtraction of (-) and (+) is the same as the addition of (-) and (-)
  - The result causes negative overflow, and thus cannot be represented as a signed integer correctly (O=1).
  - The result of AX-BX appears positive (S=0).
- Example (8-bit): (-128) - (1) = (+127)
  - Result (+127) overflowed. Ans should have been -129.
  - Result appears positive, but overflow occurred
  - Thus (-128) is less than (1), the condition is **TRUE for JL**

Conditional jumps

- Terminology used to differentiate between jump instructions that use the carry flag and the overflow flag
  - Above/Below unsigned compare
  - Greater/Less signed (+/-) compare
- Names of jump instructions
  - J => Jump
  - N => Not
  - A/B G/L => Above/Below Greater/Less
  - E => Equal

Conditional jumps

Command	Description	Condition
JA=JNBE	Jump if above Jump if not below or equal	C=0 & Z=0
JBE=JNA	Jump if below or equal	C=1 | Z=1
JAE=JNB=JNC	Jump if above or equal Jump if not below Jump if no carry	C=0
JB=JNA=JC	Jump if below Jump if carry	C=1
JE=JZ	Jump if equal Jump if Zero	Z=1
JNE=JNZ	Jump if not equal Jump if not zero	Z=0
JS	Jump Sign (MSB=1)	S=1

Conditional jumps

Command	Description	Condition
JNS	Jump Not Sign (MSB=0)	S=0
JO	Jump if overflow set	O=1
JNO	Jump if no overflow	O=0
JG=JNLE	Jump if greater Jump if not less or equal	S=O & Z=0
JGE=JNL	Jump if greater or equal Jump if not less	S=O
JL=JNGE	Jump if less Jump if not greater or equal	S^O
JLE=JNG	Jump if less or equal Jump if not greater	S^O | Z=1
JCXZ	Jump if register CX=zero	CX=0

Looping

- The LOOP instruction is a combination of the DEC and JNZ instructions
- LOOP decrements CX and if CX has not become zero jumps to the specified label
- If CX is zero, the instruction following the LOOP is executed

Loop example

```
ADDS
        mov cx, 100             ;load count
        mov    si, BLOCK1
        mov    di, BLOCK2
.Again
        lodsw                   ;get Block1 data
                                ; AX = [SI]; SI = SI + 2

        add    AX, [ES:DI]      ;add Block2 data

        stosw                   ;store in Block2
                                ; [DI] = AX; DI = DI + 2

        loop   .Again           ;repeat 100 times

        ret
```

If…then…else clauses

```
‘ Visual Basic Example
if ax > bx then
    ‘true instructions
else
    ‘false instructions
end if
```

```
cmp ax, bx
ja .true
    ;false instructions
    jmp .done
.true
    ;true instructions
.done
```

Switch…case clauses

```
/* C code example */
switch (choice) {

    case ‘a’:
        /* a instructions */
        break;

    case ‘b’:
        /* b instructions */
        break;

    default:
        /* default instructions */
}
```

```
cmp al, ‘a’
jne .b
    ; a instructions
    jmp .done

.b
cmp al, ‘b’
jne .default
    ; b instructions
    jmp .done

.default
    ; default instructions
 .done
```

Do…while clause

```
do {
   // instructions
while (A=B);
```

```
.begin
   ; instructions
mov ax, [A]
cmp ax, [B]
je .begin
```

While…do clause

```
while (A=B) {
    // instructions
}
```

```
.begin
    mov ax, [A]
    cmp ax, [B]
    jne .done
    ; instructions
jmp .begin
```

For loops

```
for (i = 10; i > 0; i--) {
    // instructions
}
```

```
mov cx, 10
.begin
    ; instructions
loop .begin
```

Examples

```
;    if  (J <= K) then
;            L := L + 1
;    else    L := L - 1
;            J, K, L are signed words
            mov     ax, [J]
            cmp     ax, [K]
            jnel    .DoElse
            inc     word [L]
            jmp     .ifDone
.DoElse:
            dec     word [L]
.ifDone:
```

```
;    while (J >= K) do  begin
;            J := J - 1;
;            K := K + 1;
;            L := J * K;
;    end;
.WhlLoop:
            mov     ax, [J]
            cmp     ax, [K]
            jnge    .QuitLoop
            dec     word [J]
            inc     word [K]
            mov     ax, [J]
            imul    [K]
            mov     [L], ax
            jmp     .WhlLoop
.QuitLoop:
```

LOOPNE

- The LOOPNE instruction is useful for controlling loops that stop on some condition or when the loop exceeds some number of iterations
- Consider String1 that contains a sequence of characters that end with the byte containing zero
- we want to convert those characters to upper case and copy them to String2

```
…..
String1  DB  “This string contains lower case characters”, 0
String2  128 DB 0
……..
```

Example (LOOPNE)

```
   mov si, String1          ;si points to beginning of String1
   lea di, String2          ;di points to beginning of String2
   mov cx, 127              ;Max 127 chars to String2
.StrLoop:
   lodsb                    ;get char from String1;
   AL=[DS:SI]; SI = SI + 1
   cmp al, 'a'              ;see if lower case
   jb .NotLower             ;chars are unsigned
   cmp al, 'z'
   ja .NotLower
   and al, 5Fh              ;convert lower -> upper case
                            ;bit 6 must be 0
.NotLower:
   stosb                    ;[ES:DI] = AL; DI = DI + 1
   cmp al, 0                ;see if zero terminator
   loopne .StrLoop          ;quit if AL or CX = 0
```