WWW.MWFTR.COM

Branching and Looping

EECE416 uC

Fall 2011

Unconditional Jumps

- `jmp`
  - Like a goto in a high-level language
  - Format: `jmp StatementLabel`
  - The next statement executed will be the one at `StatementLabel`:
- `jmp` Encoding
  - Relative **shor**t encodes a single byte signed displacement telling how far forward or backward to jump for the next instruction to execute – the assembler uses this format if possible
  - Relative **near** encodes a signed doubleword displacement – this allows a forward or backward jump essentially anywhere in memory
  - Indirect forms that encode the address of the destination in a register or memory are not often used
- Program Design using `jmp` for 1+2+3+… forever

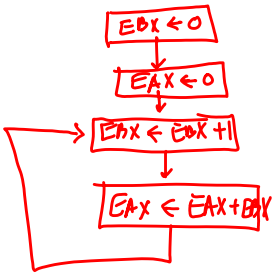
  number := 0;
  sum := 0;
  forever loop
    add 1 to number;
    add number to sum;
  end loop;

Program Code

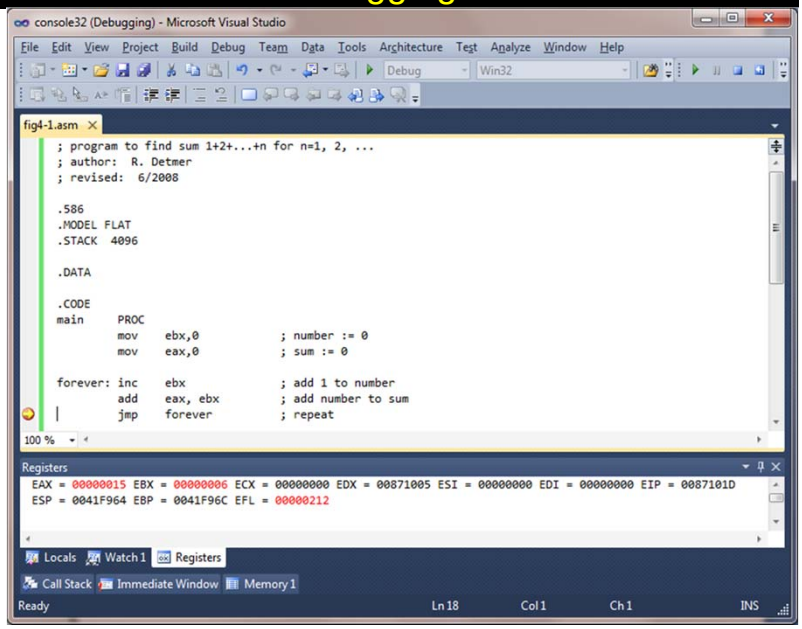
```
; program to find sum 1+2+...+n for n=1, 2, ...
.586
.MODEL FLAT
.STACK  4096
.DATA
.CODE
main     PROC
         mov    ebx,0   ; number := 0
         mov    eax,0   ; sum := 0

forever: inc    ebx     ; add 1 to number
         add    eax, ebx  ; add number to sum
         jmp    forever ; repeat

main     ENDP
END
```



Debugging



Conditional Jumps

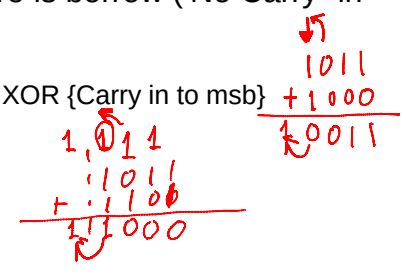
- Format: `j-- targetStatement`
- The last part of the mnemonic identifies the condition under which the jump is to be executed
- If the condition holds, then the jump takes place and the statement executed is at `targetStatement`:
- Otherwise, the next instruction (the one following the conditional jump) is executed
- Used to implement **if** structures, other selection structures, and loop structures in 80x86 assembly language
- Most "conditions" considered by the conditional jump instructions are settings of flags in the flags register.
- Example

```
jz endWhile
```

  - jump to the statement with label endWhile if the zero flag ZF is set to 1
- Conditional jump instructions don't modify flags; they react to previously set flag values

`cmp` Instruction

- Most common way to set flags for conditional jumps
- Format: `cmp operand1, operand2`
- Flags are set the same as for the subtraction operation {`operand1 – operand2`}
- Operands are not changed
- `cmp` does set or clear flags
  - CF: Carry flag (when there is borrow ('No Carry" in subtraction)
  - OF: Overflow (Overflow)
    - OF={ Carry out from msb} XOR {Carry in to msb}
  - SF: msb (Sign bit) is 1
  - ZF: Result is zero



Flag revisited – Example

		op1 - op2	Signed	Unsigned		Flags		
Op1	Op2	Diff	> = <	> = <	CF	OF	SF	ZF
3B	3B	00	op1 = op2	op1 = op2	0	0	0	1
3B	15	26	op1>op2	op1>op2	0	0	0	0
15	3B							
F6	F9	FD	op1>op2	op1<op2	1	0	1	0
F9	F6							
15	F6	1F	op1>op2	op1<op2	1	0	0	0
F6	15							
68	A5							
A5	68	3D	op1<op2	op1>op2	0	1	0	0

Flags – example & Sol

		op1 - op2	Signed	Unsigned	Flags			
Op1	Op2	Diff	> = <	> = <	CF	OF	SF	ZF
3B	3B	00	op1 = op2	op1 = op2	0	0	0	1
3B	15	26	op1>op2	op1>op2	0	0	0	0
15	3B	DA	op1<op2	op1<op2	1	0	1	0
F6	F9	FD	op1>op2	op1<op2	1	0	1	0
F9	F6	03	op1<op2	op1>op2	0	0	0	0
15	F6	1F	op1>op2	op1<op2	1	0	0	0
F6	15	E1	op1<op2	op1>op2	0	0	1	0
68	A5	C3	op1>op2	op1<op2	1	1	1	0
A5	68	3D	op1<op2	op1>op2	0	1	0	0

Conditional Jumps To Use After *Signed* Operand Comparison

mnemonic		jumps if
jg jnle	jump if greater jump if not less or equal	SF=OF and ZF=0
jge jnl	jump if greater or equal jump if not less	SF=OF
jl jnge	jump if less jump if not above or equal	SF=/=OF
jle jng	jump if less or equal jump if not greater	SF=/=OF or ZF=1

Conditional Jumps To Use After *Unsigned* Operand Comparison

mnemonic		jumps if
ja jnbe	jump if above jump if not below or equal	CF=0 and ZF=0
jae jnb	jump if above or equal jump if not below	CF=0
jb jnae	jump if below jump if not above or equal	CF=1
jbe jna	jump if below or equal jump if not above	CF=1 or ZF=1

Some Other Conditional Jumps

mnemonic		jumps if
je	jump if equal	ZF=1
jz	jump if zero	
jne	jump if not equal	ZF=0
jnz	jump if not zero	
js	jump if sign (negative)	SF=1
jc	jump if carry	CF=1
jo	jump if overflow	OF=1

Example of `cmp` and `j--`

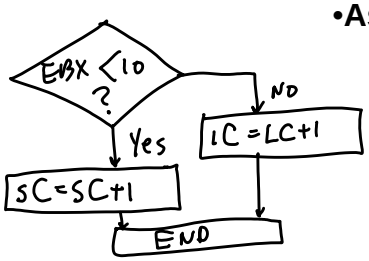
```
cmp  eax, nbr
     jle  smaller
```

- The jump will occur if the value in **eax** is less than or equal to the value in **nbr**, where both are interpreted as signed numbers

if Example 1

Design

if *value* < 10
then
    add 1 to *smallCount*;
else
    add 1 to *largeCount*;
end if;



Code

```
                cmp   ebx, 10
                jnl   elseLarge
                inc   smallCount
                jmp   endValueCheck
elseLarge:      inc   largeCount
endValueCheck:
```

•Assumptions
- **value in EBX**
- **smallCount and largeCount in memory**

if Example 2

Design

if (*total* ≥ 100)
        or (*count* = 10)
then
    add *value* to *total*;
end if;

Draw flow chart(1)

Code

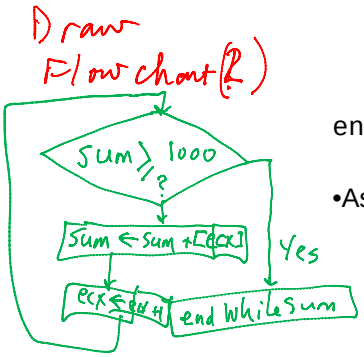
```
                cmp   total, 100
                jge   addValue
                cmp   ecx, 10
                jne   endAddCheck
addValue:       mov   ebx, value
                add   total, ebx
endAddCheck:
```

•Assumptions
- total and value in memory
- count in ECX

While Example

Design

while (*sum* < 1000) loop
    add *count* to *sum*;
    add 1 to *count*;
end while;



Code

```
whileSum:     cmp   sum, 1000
              jnl   endWhileSum
              add   sum, ecx
              inc   ecx
              jmp   whileSum
endWhileSum:
```

•Assumptions
- sum in memory
- count in ECX

Until Example

Design

repeat
    add 2**count* to *sum*;
    add 1 to *count*;
until (*sum* > 1000);

Draw Flowchart (3)

Code

```
repeatLoop:   add   sum, ecx
              add   sum, ecx
              inc   ecx
              cmp   sum, 1000
              jng   repeatLoop
endUntilLoop:
```

•Assumptions
- sum in memory
- count in ECX

Practice (for submission)

- Draw a flowchart for each of the following Assembly Codes

- (a)

```
        cmp ecx, 0
        jne endifa
        mov ecx, value
endifa:
```

- (b)

```
        cmp ecx, value
        jng endifb
        mov ecx, 0
endifb:
```

Practice (for submission)

- Draw flowcharts for the following Assembly Codes
- (c)

```
        mov ebx, a
        add ebx, b
        cmp ebx, c
        jne elsec
        mov al, 'Y'
        jmp endifc
elsec:  mov al, 'N'
endifc:
```

- (d)

```
        cmp edx, -1000
        jle thend
        cmp edx, 1000
        jnge endifd
thend:  mov edx, 0
endifd:
```

Practice (for submission)

- Draw a flowchart for the following Assembly Code
- (e)

```
        cmp  al, 'a'
        jnae else1
        cmp  al, 'z'
        jnbe else1
        inc  lowerCount
        jmp  endif1
else1:  cmp  al, 'A'
        jnae else2
        cmp  al, 'Z'
        jnbe else2
        inc  upperCount
        jmp  endif2
else2:  inc  otherCount
endif2:
endif1:
```

What is this code for? (1) Flowchart & (2) Description (plain English)

```
.586
.MODEL FLAT
.STACK  4096

.DATA
number  DWORD   750

.CODE
main        PROC
            mov   ecx, 0      ; x := 0
            mov   eax, 1      ; twoToX := 1
whileLE:    cmp   eax, number ; twoToX <= number?
            jnle  endWhileLE  ; exit if not
body:       add   eax, eax    ; multiply twoToX by 2
            inc   ecx         ; add 1 to x
            jmp   whileLE     ; go check condition again
endWhileLE:
            dec   ecx         ; subtract 1 from x

            mov   eax, 0      ; exit with return code 0
            ret
main        ENDP
END
```

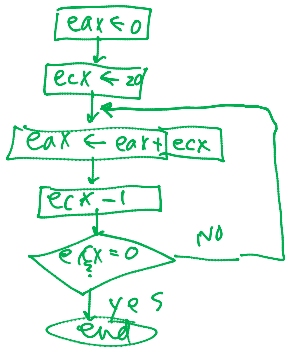
`loop` instruction

- format: `loop` *`statementLabel`*
  - *statementLabel* is the label of a statement which is a short displacement from the loop instruction
- Execution
  - (a) ECX ← ECX – 1
  - (b) If [ECX] =0, Go to next line
  - (c) otherwise, jump to the Label
  - The value in **ECX** is decremented
  - If the new value in ECX is zero, then execution continues with the statement following the loop instruction
  - If the new value in **ECX is non-zero**, then a **jump to the instruction at statementLabel** takes place

example of `loop`

Design

sum := 0
for *count* := 20 downto 1 loop
  add *count* to *sum*;
end for;



Code

```
            mov   eax, 0
            mov   ecx, 20
forCount:   add   eax, ecx
            loop  forCount
```

•Assumptions
- sum in EAX
- count in ECX

Example Code

Draw a flowchart!

```
.586
.MODEL FLAT
.STACK  4096

.DATA

.CODE
main        PROC
            mov   ebx, 1      ;
            mov   eax, 0      ;
            mov   ecx, 0      ;
whilePoor:  cmp   eax, 100000000  ;
            jnl   endLoop     ; exit if not
body:       add   eax, ebx    ; add
            add   ebx, ebx    ; multiply by 2
            inc   ecx         ;
            jmp   whilePoor   ; repeat
endLoop:
            mov   eax, 0      ; exit with return code 0
            ret
main        ENDP
END
```

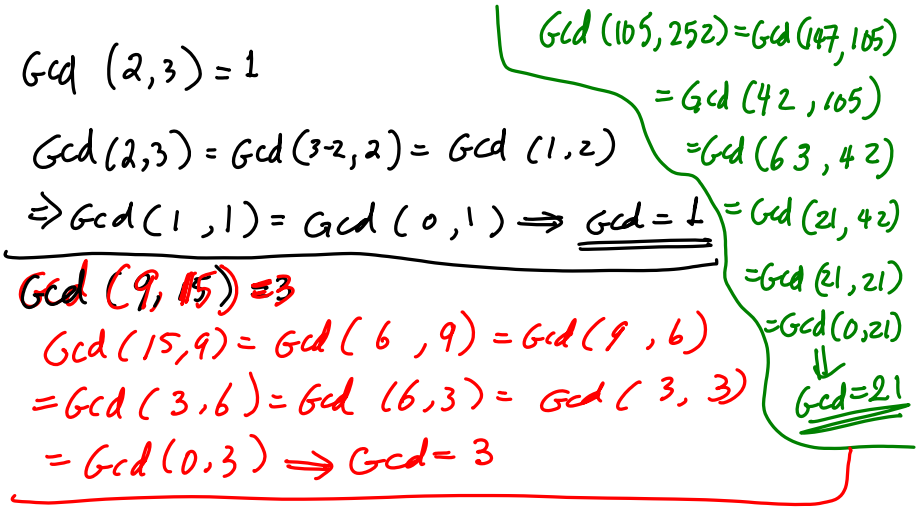
Cautions

- If ECX is initially 0, then 00000000 will be decremented to FFFFFFFF, then FFFFFFFE, etc., for a total of 4,294,967,296 iterations
- The `jecxz` ("jump if ECX is zero") instruction can be used to guard a loop implemented with the `loop` instruction

Coding Assignment -- GCD calculation

- GCD (Greatest common divisor)
- Algorithm?
  - Euclidean Algorithm
  - Bezout's Identity (for n1>n2)
    - GCD(n1, n2) = GCD (n1-n2, n2)
  - Keep the smaller number, and repeat the process
  - Until one of the numbers is 0
  - Then, GCD is the non-zero number

GCD – Euclidean Algorithm



Coding Assignment details

- 1. Draw a flow chart of the Euclidean gcd algorithm (Note: You can use Binary GCD algorithm if you want.)
- 2. Write a Code which calculates GCD of 2 numbers, which all (i.e., 2 inputs and outputs) are to be interactive with users. Note that your code must match with your flow chart (variable name, label, etc)
- 3. Submission
  - Flow Chart (hand delivery): Thursday November 17, 2011 (5:10pm)
  - 80X86 code (email submission) : Thursday November 17, 2011 (5:00pm)
- 4. Importance of the HW?
  - The same weight as Exam 01