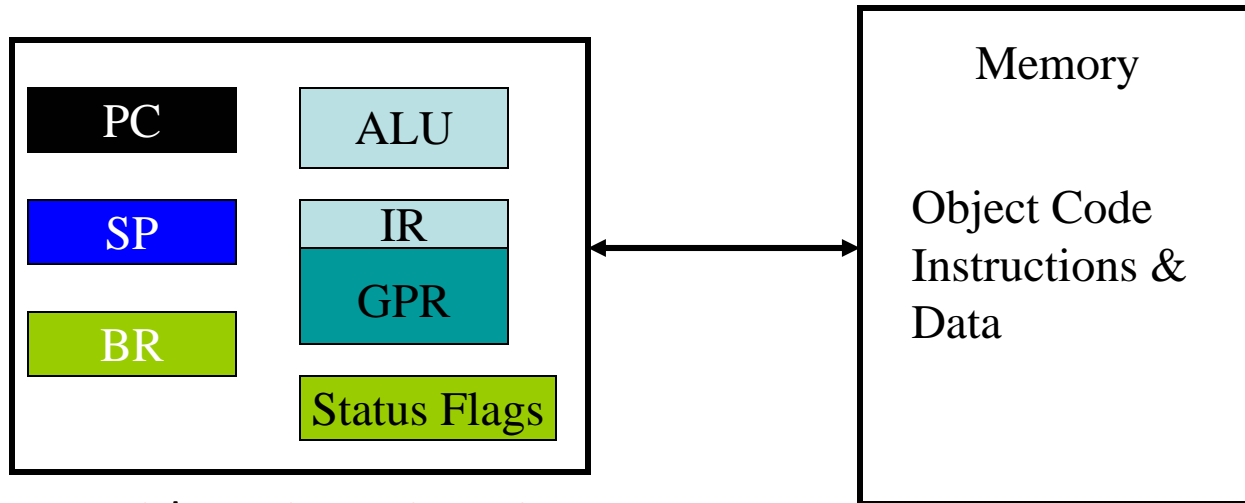


# Machine-Level Programming II: Control Flow

- Topics
  - Condition Codes
    - Setting
    - Testing
  - Control Flow
    - If-then-else
    - While, for loop

# Assembly programmer view



- ALU: Arithmetic Logic Unit
- IR: Instruction register
- GPR: General Purpose Registers
- PC: Program Counter
- SP: Stack Pointer
- BR: Base Register

## Status flags

- There are 1 bit flags or condition codes that are set in a status register
- ZF stands for zero flag
- SF stands for sign flag
- CF stands for carry flag (when carry out occurs)
- OF stands for overflow flag (when carry in to MSB occurs)
- PF parity flag
- AF auxiliary flag

## Status flags (ZF or zero flag)

- Status flags or condition codes are set when the processor executes Arithmetic operations
- ZF: Zero Flag is set to 1 if an operation results in a zero
  - Example 1
    - `mov 0xF %eax`
    - `add 1 %eax`
  - Example 2
    - `mov 1 %eax`
    - `dec %eax` (decrements register by 1)
- Operations that set zero flag are:
- `cmp`, `inc`, `dec`, `sub`, `add` etc

# cmp operation

- `cmp Src, Dst`
  - Compares by subtracting `Dst` from `Src`
- `cmp op1, op2`
- Compares two operands by subtracting `op2` from `op1` (`op2-op1`)
- Sets the status flags based on the result
- Does not alter `op2`
- Subtract operation or `sub` instruction stores the result in `op2`
- `sub Src, Dst`
- `Dst ← Dst - Src`
- `sub op1, op2`    `op2-op1` result stored in `op2`
  - `move $5 %eax`
  - `cmp $5 %eax`
  - `%eax` will still contain 5 but `ZF` will be set to 1
  - `move $5 %eax`
  - `sub $5 %eax`
  - `%eax` will have 0 and `ZF` is set to 1

## SF (Sign Flag)

- As a result of arithmetic operation, the copy of the sign bit is in SF
  - `mov $5 %eax`
  - `sub $6 %eax`
  - SF is set to 1
- If the number is positive, MSB is 0 then SF is set to 0
  - `mov $15 %eax`
  - `add $10 %eax`
  - SF is set to 0

# Carry flag (CF)

- CF is set when a carry out occurs (number is too big)
  - `mov ffff,%eax`
  - `add 0001, %eax`
- Unsigned numbers
  - For 8 bits the range is 0 to 255
  - For 16 bits the range is 0 to 65, 535
  - For 32 bits the range is 0 to 4, 294, 967, 295
- Result of an operation on unsigned numbers results in an overflow
  - `mov $5, %eax`
  - `sub $6 %eax`
- also sets CF – operation generates a borrow into MSB
- Number is too small to be represented using signed numbers

# Overflow flag (OF)

- When a “carry in” occurs
- When signed numbers get out of range
  - `mov 7200H %eax`
  - `add 0E00H %eax`
- MSB in %eax will be 1 , a carry in has occurred
  - When you add two unsigned numbers, MSB will be 1 if the result is out of range
- OF is set if there is a carry-in to MSB or carry out of MSB
- OF is set if MSB changes
- Recall 2’s complement addition
  - Based on carry-in (OF) and carry out (CF) decide if the operation was in error

## How does the processor know?

- unsigned vs signed is not distinguished
- Any number is just a binary number
- Sets CF and OF based on carry in and carry out
- Left to implementation to interpret flags accordingly

## Compare instruction

- `cmp o1, o2` or `cmp Src, Dst`
- Sets flag based on the result of `o2-o1` or `Dst-Src`
- Unlike `sub` operation, `cmp` does not change the value of `o2` or `Dst`

# Setting Condition Codes (cont.)

- Explicit Setting by Compare Instruction

`cmpl Src, Dst`

- `cmpl b, a` like computing  $a - b$  without setting destination
- CF set if carry out from most significant bit or borrow into MSB
  - Used for unsigned comparisons
- ZF set if  $a == b$
- SF set if  $(a - b) < 0$
- OF set if two's complement overflow

`(a > 0 && b < 0 && (a - b) < 0) || (a < 0 && b > 0 && (a - b) > 0)`

OF is used for signed comparisons

## Setting Condition Codes (cont.)

- Explicit Setting by Test instruction

```
testl Src2,Src1
```

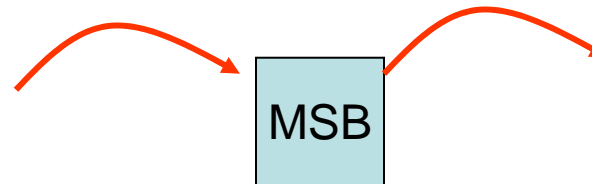
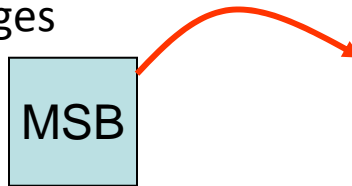
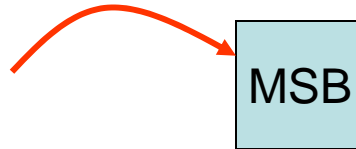
- Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask
- `testl b,a` like computing `a&b` without setting destination
- ZF set when `a&b == 0`
- SF set when `a&b < 0`

# Flags and operations

- Assume  $x > y$
- `cmp y, x` is equivalent to checking result of  $x-y$
- Since  $x > y$ ,  $x-y$  has to be positive and non-zero
- Hence,  $SF=0$  and  $ZF=0$  and  $OF=0$
- However, if  $SF=1$  then overflow has occurred, i.e.,  $OF=1$
- If  $X > Y$ , then `cmp y, x` will result in  $ZF=0$  and  $SF=OF$
- If  $X > Y$ , then  $\sim ZF \ \& \ \sim(SF \wedge OF)$  must be true
- $\wedge$  symbol is for XOR,  $\sim$  is NOT, and  $\&$  is AND ... bitwise
- `jb` needs to check if  $\sim ZF \ \& \ \sim(SF \wedge OF)$  is true

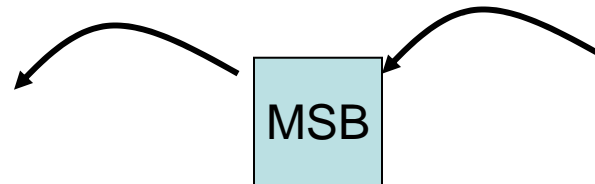
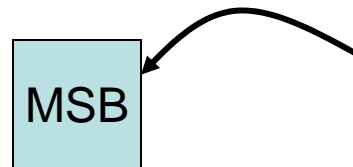
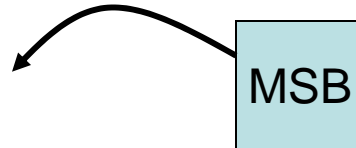
# OF/CF for subtract operation

- CF and OF are set
- **Borrow-in to MSB**
  - CF
  - OF because MSB changes
- Only OF flag is set
- **Borrow-out of MSB**
  - OF because MSB changes
- Only CF is set
- Borrow-in to MSB, borrow-out of MSB
  - MSB does not change



# OF/CF for add operation

- CF and OF are set
- **Carry-out of MSB**
  - CF
  - OF because MSB changes
- Only OF flag is set
- **carry-in to MSB**
  - OF because MSB changes
- Only CF is set
- Carry-out of MSB, carry-in to MSB
  - MSB does not change



## Example 8-bit numbers

- `cmp 55, 56`

Sub	00111000	SF=0
	00110111	ZF=0
	-----	OF=0
	00000001	CF=0

- `cmp -58, -56`

sub	11001000	SF=0	
	11000110	ZF=0	add 11001000
	-----	OF=0	00111010
	00000010	CF=0	-----
			00000010

- `cmp -75, 56`

sub	00111000	SF=1	add 00111000
	10110101	ZF=0	01001011
	-----	OF=1	-----
	10000011	CF=1	10000011

# Flags and operations

- Assume  $x < y$
- `cmp y, x` is equivalent to checking result of  $x-y$
- Since  $x < y$ ,  $x-y$  has to be negative and non-zero
  - Hence,  $SF=1$  and  $ZF=0$  and  $OF=0$
- However, if  $SF=0$  then overflow has occurred, i.e.,  $OF=1$ 
  - If  $x < y$ , then `cmp y, x` will result in  $ZF=0$  and  $SF \neq OF$
  - Note: if  $X=Y$ , then  $ZF=OF=SF=0$
- Hence, we only need to check  $SF \neq OF$
- If  $X < Y$ , then  $(SF \wedge OF)$  must be true
- $\wedge$  is XOR, ... bitwise
- `Jl` needs to check if  $(SF \wedge OF)$  is true

## Example 8-bit numbers

- `cmp 56, 55`

Sub 00110111	SF=1
00111000	ZF=0
-----	OF=0
11111111	CF=1

- `cmp -56, -58`

11000110	SF=1
11001000	ZF=0
-----	OF=0
11111110	CF=1

- `cmp 56, -75`

10110101	SF=0
00111000	ZF=0
-----	OF=1
01111101	CF=0

## Exercise: fill the entries

cmp	SF	ZF	CF	OF
57,56				
101, 200				
200,101				
100, 100				
-124, -125				
-125, -124				

# Reading Condition Codes

- SetX Instructions
  - Set single byte based on combinations of condition codes

<b>SetX</b>	<b>Condition</b>	<b>Description</b>
<b>Sete D</b>	<b>ZF</b>	<b>Equal / Zero</b>
<b>Setne D</b>	<b>~ZF</b>	<b>Not Equal / Not Zero</b>
<b>Sets D</b>	<b>SF</b>	<b>Negative</b>
<b>Setns D</b>	<b>~SF</b>	<b>Nonnegative</b>
<b>Setg D</b>	<b>~(SF^OF) &amp; ~ZF</b>	<b>Greater (Signed)</b>
<b>Setge D</b>	<b>~(SF^OF)</b>	<b>Greater or Equal (Signed)</b>
<b>Setl D</b>	<b>(SF^OF)</b>	<b>Less (Signed)</b>
<b>Setle D</b>	<b>(SF^OF)   ZF</b>	<b>Less or Equal (Signed)</b>
<b>Seta D</b>	<b>~CF &amp; ~ZF</b>	<b>Above (unsigned)</b>
<b>Setb D</b>	<b>CF</b>	<b>Below (unsigned)</b>

# Jmp Instruction

- A JMP instruction forces the program counter to point to the address specified in the instruction
  - `jmp target`
  - `jmp (%edx)`
  - `edx` contains `0080H`
- After execution of `jmp` instruction the program counter or `%eip` points to contents of `edx`
- In other words, the program starts executing from that address

# Jumping

- jX Instructions
  - Jump to different part of code depending on condition codes

<b>jX</b>	<b>Condition</b>	<b>Description</b>
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	$\sim ZF$	Not Equal / Not Zero
js	SF	Negative
jns	$\sim SF$	Nonnegative
jg	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
jge	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
jl	$(SF \wedge OF)$	Less (Signed)
jle	$(SF \wedge OF) \   \ ZF$	Less or Equal (Signed)
ja	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
jb	CF	Below (unsigned)

# Conditional jumps

- Jcondition target
- Can be used to implement control structures
- Compare the while loop condition
- While (something <>0 ) {}
- Here we can use jz endwhile
- jmp to endwhile if zero flag ZF is set

- If temp = 0
- Printf (“freezing \n”)
- Else
- Printf (“notbad \n”);

We can use jnz

```

    cmp temp 0
    jnz elsepart
    Print freezing
    Jmp endif
Elsepart: Print notbad
Endif:

```

# Conditional Branch Example

```
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

```

_max:
    pushl %ebp          }
    movl  %esp,%ebp    } Set Up
                        }
    x  movl  8(%ebp),%edx
    y  movl  12(%ebp),%eax
    x-y?  cmpl %eax,%edx
        jle L8
        movl %edx,%eax
                        } Body
L8:   movl  %ebp,%esp
        popl %ebp
        ret
                        } Finish

```

Return value in %eax

# Conditional Branch Example II

```
int min(int x, int y)
{
    if (x < y)
        return x;
    else
        return y;
}
```

```

_min:
    pushl %ebp          }
    movl %esp,%ebp     } Set Up
                        }
    x  movl 8(%ebp),%edx
    y  movl 12(%ebp),%eax
    x-y? cmpl %eax,%edx }
        jge L8          } Body
        movl %edx,%eax
L8:    movl %ebp,%esp
        popl %ebp
        ret              } Finish

```

Return value in %eax

# Actual assembly code on x86 machine

```

min:
pushl %ebp
movl %esp, %ebp
subl $4, %esp
movl 8(%ebp), %eax
cmpl 12(%ebp), %eax
jge .L2
movl 8(%ebp), %eax
movl %eax, -4(%ebp)
jmp .L3
.L2:
movl 12(%ebp), %eax
movl %eax, -4(%ebp)
.L3:
movl -4(%ebp), %eax
leave
ret

```

```

max:
pushl %ebp
movl %esp, %ebp
subl $4, %esp
movl 8(%ebp), %eax
cmpl 12(%ebp), %eax
jle .L2
movl 8(%ebp), %eax
movl %eax, -4(%ebp)
jmp .L3
.L2:
movl 12(%ebp), %eax
movl %eax, -4(%ebp)
.L3:
movl -4(%ebp), %eax
leave
ret

```



# For loop

- Test
- If test is true execute
- For loop body
- Update loop variable
- Else exit

```
for ( Init; Test; Update )
    Body
```

```
for ( I=0; I < 211; i++ )
{
```

```
    Body
}
```

```

        mov $0 , %esi
startfor: cmp $211, %esi
        jge exitfor
        body
        inc %esi
        jmp startfor
Exitfor:
```

# For loop: another implementation

- Jmp to Test
- For loop
- body
- Update loop variable
- Test: if true jmp to body

```
for ( Init; Test; Update )
    Body
```

```
for ( I=0; I < 211; i++ )
{
    Body
        mov $0 , %esi
        jmp fortest
startfor:
        body
        inc %esi
fortest: cmp $211, %esi
        jl startfor
```

# Implementing for loop

## For Version

```
for ( Init; Test; Update )  
    Body
```

## While Version

```
Init;  
while ( Test ) {  
    Body  
    Update ;  
}
```

## Do-While Version

```
Init;  
if (!Test)  
    goto done;  
do {  
    Body  
    Update ;  
} while ( Test )  
done:
```

## Goto Version

```
Init;  
if (!Test)  
    goto done;  
loop:  
    Body  
    Update ;  
    if ( Test )  
        goto loop;  
done:
```