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 borrow or overflow occurs) – detect unsigned overflow
- OF stands for overflow flag (when MSB change occurs) – detect signed overflow
- 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 0xFFFFFFFF %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, Dat`
  - Compares by subtracting `Src` from `Dat`
- `cmp op1, op2`
  - Compares two operands by subtracting `op1` from `op2` (`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, Dat`
  - `Dat ← Dat - 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 or borrow in occurs`
  - `mov $7, %al`
  - `add 00000001, %al`

  - `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 32 bits 0 to 2,147,483,647
  - Result of an operation on unsigned numbers result 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 to or carry-out of "MSB" occurs— for signed numbers
  - When signed numbers get out of range
    - `mov 0x72 %al`
    - `add 0x60 %al`
  - MSB in `%al` will be 1 a carry in has occurred
    - When you add two signed numbers, MSB will change 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 differs
  - OF is used to detect overflow for signed numbers
  - 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

- 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` (`testb, testw` for different word sizes)
    - 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` and `~SF a&b > 0`
**Flags and operations**

- Assume \( x > y \)
- \( \text{cmp} \ y, x \) 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 \( \text{cmp} \ y, x \) will result in ZF=0 and SF=OF
- If \( X > Y \), then \( \negZF \& \neg(SF\Or OF) \) must be true
- \( \^ \) symbol is for XOR, \( \neg \) is NOT, and \( \& \) is AND ... bitwise
- \( \text{jg} \) needs to check if \( \negZF \& \neg(SF\Or OF) \) 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**

- \( \text{cmp} \ 55, 56 \)
- \( \text{cmp} \ -58, -56 \)
- \( \text{cmp} \ -75, 56 \)

### Add 8-bit numbers

<table>
<thead>
<tr>
<th>S</th>
<th>D</th>
<th>Sub</th>
<th>00110100</th>
<th>SF=0</th>
<th>ZF=0</th>
<th>OF=0</th>
<th>CF=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00110111</td>
<td>00000001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>D</th>
<th>Sub</th>
<th>11000100</th>
<th>SF=0</th>
<th>ZF=0</th>
<th>OF=0</th>
<th>CF=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00110111</td>
<td>00000001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>D</th>
<th>Sub</th>
<th>00110100</th>
<th>SF=0</th>
<th>ZF=0</th>
<th>OF=0</th>
<th>CF=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00111010</td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>D</th>
<th>Sub</th>
<th>11000100</th>
<th>SF=0</th>
<th>ZF=0</th>
<th>OF=0</th>
<th>CF=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00111010</td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>D</th>
<th>Sub</th>
<th>00110011</th>
<th>SF=1</th>
<th>ZF=0</th>
<th>OF=1</th>
<th>CF=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>D</th>
<th>Sub</th>
<th>10110101</th>
<th>SF=1</th>
<th>ZF=0</th>
<th>OF=1</th>
<th>CF=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>D</th>
<th>Sub</th>
<th>00110011</th>
<th>SF=1</th>
<th>ZF=0</th>
<th>OF=1</th>
<th>CF=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>D</th>
<th>Sub</th>
<th>10110101</th>
<th>SF=1</th>
<th>ZF=0</th>
<th>OF=1</th>
<th>CF=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Flags and operations

- Assume \( x < y \)
- \( \text{cmp } y, x \) 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 \( \text{cmp } y, x \) will result in \(ZF=0\) and \(SF \neq OF\)
  - Note: if \( x=y \), then \(ZF=1\) and \(OF=SF=0\)
- Hence, we only need to check \(SF \neq OF\)
- If \( x < y \), then \((SF \oplus OF)\) must be true
  - \( \oplus \) is XOR, … bitwise
- \(Jl\) needs to check if \((SF \oplus OF)\) is true

Example 8-bit numbers

- \(\text{cmp } 56, 55\)
  - Sub 00110111
  - SF=1
  - ZF=0
  - OF=0
  - 11111111

- \(\text{cmp } -56, -58\)
  - 10100101
  - SF=1
  - ZF=0
  - OF=0
  - 11000110

- \(\text{cmp } 56, -75\)
  - 00111000
  - SF=0
  - ZF=0
  - OF=1
  - 01110101

Exercise: fill the entries

<table>
<thead>
<tr>
<th>cmp</th>
<th>SF</th>
<th>ZF</th>
<th>CF</th>
<th>OF</th>
</tr>
</thead>
<tbody>
<tr>
<td>57,56</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101,200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200,101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100,100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-124,-125</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-125,-124</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reading Condition Codes

- SetX Instructions
  - Set single byte to \((0 \text{ OR } 1)\) based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sete D</td>
<td>SF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>Setne D</td>
<td>~SF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>Setn D</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>Setns D</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>Setg D</td>
<td>((SF \oplus OF) \neq SF)</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>Setge D</td>
<td>((SF \oplus OF))</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>Setl D</td>
<td>((SF \oplus OF))</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>Setls D</td>
<td>((SF \oplus OF) \neq SF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>Seta D</td>
<td>~CF &amp; ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>Setb D</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
### 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

### Conditional jumps

- Jcondition target
- Can be used to implement control structures
  - Compare the while loop condition
  - While (something <0 ) { statements}
  - Here we can use
    - beginwhile: cmp something, 0
    - je endwhile
  - statements
  - jmp beginwhile
- endwhile:
  - If temp = 0
  - Printf ("freezing in") We can use je
  - Else
  - Printf ("notbad in"): "notbad in"

### Conditional Branch Example

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

- \_max:
  - pushl \%ebp
  - movl \%esp, \%ebp
  - movl 8(%ebp), \%edx
  - movl 12(%ebp), \%eax
  - cmpl \%eax, \%edx
  - jle L8
  - movl \%edx, \%eax
  - jmp \%edx, \%eax
  - movl \%edx, \%eax
  - movl \%edx, \%eax
  - jmp \%edx, \%eax
  - popl \%ebp
  - ret

- L8:
  - Return value in \%eax

### Jump Instructions

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

<table>
<thead>
<tr>
<th>(jX)</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF)&amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>ja</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

- \(jX\) Instructions
  - Jump to different part of code depending on condition codes
  - Can be used to implement control structures
  - Compare the while loop condition
    - While (something <>0 ) { statements}
    - Here we can use
      - beginwhile: cmp something, 0
      - je endwhile
  - statements
  - jmp beginwhile
  - endwhile:
    - If temp = 0
    - Printf ("freezing in") We can use je
    - Else
    - Printf ("notbad in"): "notbad in"
Conditional Branch Example II

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

Actual assembly code on x86 machine

```assembly
min:
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %edx
movl 12(%ebp), %eax
cmpl %eax, %edx
jge L8
movl %edx, %eax
L8:   movl %ebp, %esp
      popl %ebp
      ret
```

For loop

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

For loop: another implementation

```assembly
max:
pushl %ebp
movl %esp, %ebp
subl $4, %esp
movl 8(%ebp), %eax
cmpl $4, %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

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

For Version

\[ \text{for (Init; Test; Update) \{ Body \} } \]

Do-While Version

\[ \text{Init; \ \text{if (!Test) \ goto done; \ do \{ Body Update \} \ \text{while (Test) \ done; } } \]

Goto Version

\[ \text{Init; \ \text{if (!Test) \ goto done; \ loop: \ \text{Body Update \ if (Test) \ goto loop; \ done: } } \]

Machine-Level Programming III: Control Flow, Stack frame

- Topic
  - Stack frames

Programs & Memory

- The Von Neumann: programs & data are stored in memory
- Recall assembly code converted to object code (Assembler)
- Labels are nothing more then address locations of instructions
  - Here: mov %eax, %edx
  - ...
  - jmp Here

IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register %esp indicates lowest stack address
  - address of top element
- Push element on to stack
- Pop the top most element
Use of stack

- For storing return address of calling procedure
- When a procedure is called
- Program needs to jump to the starting address of the procedure
- After execution of the procedure, return to the next statement following the call statement
- CALL ... Push 0102
- RET ... POP %eip

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: call label ... Push return address on stack; Jump to label
- Return address value
  - Address of instruction beyond call
  - Example from disassembly
  - 804854e: e8 3d 06 00 00 call 8048b90 <main>
  - 8048553: 08 add ...
  - Return address = 0x8048553
- Procedure return:
  - ret ... Pop address from stack; Jump to address

Nested procedures

- On each call, push the return address on stack
- On RET, pop the address to return from the innermost procedure call
- RET -> pop %eip
Passing parameters

- What if the procedure has parameters?
- Use stack to store the values
  - First, push parameters onto stack
  - Where are the parameters?
    - %esp + 4, %esp + 8
  - %esp points to top of stack
  - Second, push return address
  - Top of stack still contains the return address

Stack frames

- To preserve the offset for parameters for every called procedure
- Maintain another register called frame pointer
- %ebp is the frame pointer
- Every time a procedure is entered
  - push %ebp, old frame pointer on to stack
  - mov %esp, %ebp
  - %ebp points to current SP
- When executing within myfunc
  - %ebp? %esp?
  - %ebp+8?, %ebp+12?

Locating Parameters?

- What happens on RET from the innermost procedure?
  - Pop decrements stack pointer
  - Need to remember where the last call stack was

Stack frames

- When myfunc calls otherfunc()
- Push parameters and return address on stack
- Before entering otherfunc()
  - push %ebp, old frame pointer
  - mov %esp, %ebp
  - %ebp points to current SP
- When executing within otherfunc()
  - %ebp? %esp?
  - %ebp+8?
- On RET (clean up)
  - mov %ebp, %esp
  - pop %ebp
What about local variables within procedures?

- Also goes on stack!
- The called procedure pushes local variables onto stack
- Where are the parameters?
  - %ebp + 8, %ebp + 12
- Where are the local variables?
  - %ebp-4, %ebp-8, ...
- z is a local variable in myfunc()

Stack Frame

- Stack has one stack frame per activation
- ESP points to top of current stack frame
- EBP points to bottom of current stack frame
- Stack Frame stores
  - Return address
  - Old call frame pointer
  - Parameters
  - Local variables

Cleaning the stack of arguments

- After mov %ebp, %esp
- pop %ebp
- RET instruction
- pop %esp
- %esp will point to the return address
- %esp is still pointing to the argument stack of the previous frame
- Two ways to clean up

Stack is used for local variables

- z[0] is stored in -40(%ebp)
- On RET from myfunc
- Clean up or LEAVE
- On RET (clean up)
  - mov %ebp, %esp
  - pop %ebp

Cleaning the stack of arguments

- After mov %ebp, %esp
- pop %ebp
- RET instruction
- pop %esp
- %esp will point to the return address
- %esp is still pointing to the argument stack of the previous frame
- Two ways to clean up
Cleaning the stack of arguments

- Cleaning done by the called procedure
- Use RET $n$ instructions
- RET n ➔ pop %eip, and add %esp, $n$
- Called procedure needs to know the size of the argument
- Cleaning done by the calling procedure
- After returning from the procedure
- addl sizeofargument, %esp

Sample program, call stack

```c
main()
{ int x=1, int y=2;
  int z;
  z=f(x,y);
}

int f(int arg1, int arg2)
{ return arg1 + arg2;
}
```

Procedure call stack

```
c:   pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    popl %ebp
    ret
```