Today

• Comparisons
• Branching (basics)
• if and while statements

Assignments

• HW6 out
Conditional Statements

Recall condition codes (eflags/rflags register):

- ZF: Zero Flag – most recent a/l (arithmetic/logic) instruction yielded zero
- SF: Sign Flag – most recent a/l instruction yielded negative value
- OF: Overflow Flag – most recent a/l instruction caused 2’s-comp overflow
- CF: Carry Flag – most recent a/l instruction had a carry out or borrow

Note that CF is only useful for unsigned values

**cmp instruction**

```assembly
    cmps src_1, src_2 ... src_2 - src_1
```

- after subtracting, just sets the above condition codes
- i.e., same as sub, but nothing stored (only condition code changes)

**example:**

```assembly
    movl $42, %eax
    movl $27, %ebx
    cmpl %eax, %ebx  # 27 - 42 = -15 ... eg, sets SF (negative)
```
Jump instructions

jmp Label

- simplest form of the jmp instruction
- unconditionally jumps to location at label (sets IP to label address)

Example:

```
xorl %eax, %eax
L1:  incl %eax
    jmp L1          # infinite loop!
```

je Label

- Jump to address at label if zero flag set

Example:

```
  cmpl %eax, %ebx      # set flags
  je    L1            # jump if zero flag (ZF)
  jmp   L2            # unconditional jump
L1:    do something ...
L2:    do something else ...
```
Other jump variants: ... e.g., for cmp b, a

je Label ... equal (ZF flag set)

jne Label ... not equal (not ZF flag set)

js Label ... < 0 (SF flag set)

jns Label ... > 0 (SF flag not set)

jl Label ... a < b (signed)

jle Label ... a <= b (signed)

jg Label ... a > b (signed)

jge Label ... a >= b (signed)

jb Label ... a < b (unsigned)

jbe Label ... a <= b (unsigned)

ja Label ... a > b (unsigned)

jae Label ... a >= b (unsigned)
Exercise: Convert the C code to corresponding assembly:

```assembly
... # c in eax, x in ebx
if (c < 97) {
    x = x + 1;
}
c = x;
```

Exercise: Rewrite to have only one jump instruction:

```assembly
... # c in eax, x in ebx
if (c < 97) {
    x = x + 1;
}
c = x;
```

Exercise: Convert the C code to corresponding assembly:

```assembly
... # c in eax, x in ebx
if (c < 97) {
    x = x + 1;
} else if (c > 122) {
    x = x - 1;
} c = x;
```
Looping

- no new instructions needed for loop variants (do while, while do, for)
- instead of just jumping “forward”, we also can jump “back” in the program

One way to think of this is via goto statements

- which exist in C/C++

- e.g., the original C program:

```c
int x = 0; // which is equivalent to:
int i = 0;
while (i < 10) {
    x = x + 2;
    ++i;
}
```

- can be rewritten using goto:

```c
L1: int x = 0;
    int i = 0;
    if (i < 10) {
        x = x + 2;
        ++i;
        goto L1;
    }
```

- thus, we can “remove” loop constructs using goto
- and a goto is just a jump instruction
Exercise: Convert the C code to corresponding assembly:

```c
int x = 0;
int i = 0;
while (i < 10) {
    x = x + 2;
    ++i;
}
```

```assembly
# x in eax, i in ecx
xorl %eax, %eax
xorl %ecx, %ecx

while:  
    cmpl $10, %ecx
    jl body
    incl %ecx
    jmp while

body:  
    addl $2, %eax
    jmp while

done: ...
```
Two methods for converting while loops

The general form of a while loop:

```plaintext
while (test-expr)
    body-statement
```

In "jump to middle", jump first to test block:

```plaintext
goto test
loop:
    body-statement
test:
    t = test-expr
    if (t)
        goto loop
```

In "guarded do", test and then do-while:

```plaintext
t = test-expr
if (!t)
    goto done
loop:
    body-statement
t = test-expr
if (t)
    goto loop
done:
```

Note in guarded-do, the loop block is a do-while statement

```plaintext
do
    body-statement
while (test-expr)
```
Exercise: Convert the C code using the *jump-to-middle* translation:

```c
int x = 0;
int i = 0;
while (i < 10) {
    x = x + 2;
    ++i;
}
```

```asm
# x in eax, i in ecx
xorl %eax, %eax
xorl %ecx, %ecx
jmpl test

loop:
    addl $2, %eax
    incl %ecx
test:
    cmpl $10, %ecx
    jll loop
```

Exercise: Convert the C code using the *guarded-do* translation:

```c
int x = 0;
int i = 0;
while (i < 10) {
    x = x + 2;
    ++i;
}
```

```asm
# x in eax, i in ecx
xorl %eax, %eax
xorl %ecx, %ecx
cmpl $10, %ecx
jge done

loop:
    addl $2, %eax
    incl %ecx
cmpl $10, %ecx
    jl loop
done:
```
Conditional Moves

\texttt{cmovcs} \texttt{src, dst}

- move \texttt{src} to \texttt{dst} based on a comparison
- \texttt{c} denotes the comparison check (e.g., \texttt{e}, \texttt{le}, \texttt{ge}, etc.)
- \texttt{s} denotes the data sizes to move (byte, word, double word, etc.)

Example:

\begin{verbatim}
movl $3, %ebx
movl $4, %ecx
cmpl $1, %eax
cmovgl %ebx, %ecx    # ecx = 3 only if eax > 1
\end{verbatim}
Test Instruction

\texttt{test} op1, op2

- performs AND of op1 and op2, setting condition flags
- $s$ denotes the data sizes to move (byte, word, double word, etc.)
- Typically of the form \texttt{test} $\%rax, \%rax$ to see if $\%rax$ negative, zero, or positive.
- Example:

\begin{verbatim}
  testq  \%rax, \%rax  # e.g.: if (x) { ... }
  jnz   body
\end{verbatim}
Set Instruction

\texttt{setc \texttt{dst}}

- like compare, but sets \texttt{dst} (byte) register to 0 or 1 (false or true)
- \texttt{c} denotes the comparison check (e.g., e, le, ge, etc.)
- Can be used to store or return boolean expression result

- Example:

  \begin{verbatim}
  cmpq $1, %rax
  setg %bl           # bl = rax > 1
  \end{verbatim}

- Example of where this is useful:

  \begin{verbatim}
  bool x = y > 1;     # store result in variable
  ...
  ...
  return y > 1;      # return boolean value
  \end{verbatim}