Control Flow

CSE 361S

Assembly Control Flow

- Unconditional Jump –

```
        jmp    [label]

e.g.,
        jmp    L1
               …
    L1: target instruction
               …

    jmp       *%eax       indirect, dest in %eax
    jmp       *(%eax)     indirect, dest in M[%eax]
```

Conditional Control Flow

- In x86, separate expression eval and cond branch inst.
- Compare –

```
    cmpl   b, a
      {r,m,i}, {r,m}
```

- Perform operation temp = a – b, throw away temp and set flags based on results of subtraction operation.
- Flags can also be set as a result of normal arithmetic and/or logical operations.

Conditional Jumps

```
        j[cond]       [label]

e.g.,
        jge           j_loop
```

- There are three classes of conditionals:
  - General
  - Unsigned
  - Signed

General Conditionals

JZ	zero	alternate name JE
JNZ	not zero	alternate name JNE
JC	carry	
JNC	no carry	

Signed Conditionals

JG	greater than (a > b)	alternate name JNLE
JL	less than (a < b)	alternate name JNGE
JGE	greater or equal (a ≥ b)	alternate name JNL
JLE	less or equal (a ≤ b)	alternate name JNG

Unsigned Conditionals

JA	above (a > b)	alternate name JNBE
JB	below (a < b)	alternate name JNAE
JAE	above or equal (a ≥ b)	alternate name JNB
JBE	below or equal (a ≤ b)	alternate name JNA

Loop Instruction

```
        loop        [label]

e.g.,
        movl        $15, %ecx     /* # of iterations */
.L1     .
        .
        .
        loop        .L1
```

- Operation is as follows:
  - decrement %ecx
  - test %ecx for zero, if not zero jump to [label]

Control Flow in C

if … then …

```
if ([cond expr]) {
        [true body]
}
[main body]
```

e.g.,

```
if ( var1 > var2 ) {
    var1 = var1 + var2;
    var2 = 0;
}
…
```

if … then

if … then …

```
if ([cond expr]) {
        [true body]
}
[main body]
```

Assembly:

```
    cmpl  [cond exp opers]
    j[!cond]  main_body
    [true body]
main_body:
    [main body]
```

if … then

```
if ( var1 > var2 ) {
    var1 = var1 + var2;
    var2 = 0;
}
…
```

```
    movl  var1, %eax
    cmpl  var2, %eax
    jle   main_body
    addl  var2, %eax
    movl  %eax, var1
    movl  $0, var2
main_body:
    …
```

if … then … else

```
if ([cond expr]) {
    [true body]
}
else {
    [false body]
}
[main body]
```

```
    cmpl       [cond exp opers]
    j[!cond]   false_body
    [true body]
    jmp        main_body
false_body:
    [false body]
main_body:
    [main body]
```

if … then … else

```
if ( var1 > var2 ) {                      movl   var1, %eax
   var1 = var1 + var2;                    cmpl   var2, %eax
   var2 = 0;                              jle    false_body
}                                         addl   var2, %eax
else {                                    movl   %eax, var1
   var2 = var2 + var1;                    movl   $0, var2
   var1 = var2;                           jmp    main_body
}                                  false_body:
                                          addl   %eax, var2
                                          movl   var2, %eax
                                          movl   %eax, var1
                                   main_body:    …
```

Conditional if … then … else

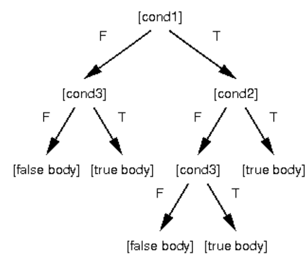
```
if (([cond1] && [cond2]) || [cond3]) {
        [true body]
}
else {
        [false body]
}
[main body]
```

- Note: evaluation order of compound expression is left to right, only conditions that need to be evaluated are evaluated

Conditional if … then … else

if (([cond1] && [cond2]) || [cond3])

Evaluation order for above compound expression:



if (([cond1] && [cond2]) || [cond3])

```
            cmpl          [cond1]
            j[!cond1]     check_cond3
            cmpl          [cond2]
            j[cond2]      true_body
    check_cond3:
            cmpl          [cond3]
            j[cond3]      true_body
            [false body]
            jmp           main_body
    true_body:
            [true body]
    main_body:
            [main body]
```

for loop

```
for ([ind var] = [init val]; [cond expr]; [update ind var] ) {
        [loop body]
}
[main body]
```

e.g.,

```
for (i=0; i<24; i++) {
    mask = 1 << i;
    status_bit[i] = status & mask;
    status_bit[i] >>= i;
}
```

for loop

```
for ([ind var] = [init val]; [cond expr]; [update ind var] ) {
        [loop body]
}
[main body]
```

- Assembly

```
            movl        [init val], [ind var]
for_loop:   cmpl        [cond expr]
            j[!cond]    loop_exit
            [loop body]
            [update ind var]
            jmp         for_loop
loop_exit:
            [main_body]
```

while loop

```
while ([cond expr]) {
        [loop body]
}
[main body]
```

- Assembly

```
while_loop:
        cmpl        [cond expr oper]
        j[!cond]    exit_while
        [loop body]
        jmp         while_loop
exit_while:
        [main body]
```

Notes for Java Programmers

- Declare index variable before for loop

```
int i;
for (i=0; i<n; i++) { }
```

vs.

```
for (int i=0; i<n; i++) { }
```

- Uninitialized variables

```
int main (int argc, char* argv[]) {
    int i;
    factorial(i);
    return 0;
}
```

Notes for Java Programmers

- Error handling:
  - No exceptions, must look at return values
  - E.g.,

```
int open_file(filename) {
        /* attempt to open filename */
        if (error) return -1;
        else return 0;
}
```

Command line arguments in C

int main(int argc, char * argv[])

- argc
  - **Number of arguments (including program name)**
- argv
  - **Array of char*s (that is, an array of 'C' strings)**
  - argv[0]**: = program name**
  - argv[1]**: = first argument**
  - …
  - argv[argc-1]**: last argument**

Program echoargs.c

```
#include <stdio.h>

int main(int argc, char* argv[])
{
   int i;

   printf("%d arguments\n", argc);
   for(i = 0; i < argc; i++)
       printf("  %d: %s\n", i, argv[i]);
   return 0;
}
```

```
> ./echoargs Humankind cannot stand very much reality
7 arguments
 0: ./echoargs
 1: Humankind
 2: cannot
 3: stand
 4: very
 5: much
 6: reality
>
```