Computer Systems Organisation (CS2.201)

Summer 2021, IIIT Hyderabad

18 June, Friday (Lecture 11) – Conditional Branches 2

Taught by Prof. Avinash Sharma

Pipelining

A pipelined architecture refers to one in which the next iteration of the fetch-decode-execute cycle is begun before the current interation is completed. For example, while decoding a certain instruction, the processor might fetch the next one.
This leads to problems when jump statements are included – the processor has already started fetching or decoding the statements which are upcoming sequentially. These instructions must then be discarded and the processor needs to restart from the destination of the jump statement. This is a waste of clock cycles.

Consider the C code

```
long cread(long *xp)
{
    return (xp ? *xp : 0);
}
```

This generates the assembly code

```
movl (%edx), %eax
testl %edx, %edx
movl $0, %edx
cmove %edx, %eax
```

where `movl` is executed after `testl` unconditionally, but `cmove` is executed only of `%edx` is 0.

Loops

Do-While

A do-while loop has the form

```
do
    body;
    while (test);
```

which is equivalent to

```
loop:
    body;
```

```
if (test)
    goto loop;
```

Consider the C code

```
int fact_do(int n)
{
    int result = 1;
    do {
        result *= n;
        n = n-1;
    } while (n > 1)
    return result'
}
```

which generates the assembly code

```
movl 8(%ebp), %edx
movl $1, %eax
.L2
imull %edx, %eax
subl $1, %edx
cmpl $1, %edx
jg .L2
```

While

In C, a while loop has the form

```
while (test)
    body;
```

which is equivalent to

```
if (!test)
    goto done;
do
    body;
    while (test)
done
```

where the do-while loop is converted to `goto` as above.

For example, in C, the following code

```
int fact-while(int n)
{
    int result = 1;
    while (n > 1)
    {
```

```
        result *= n
        n = n-1;
    }
    return result;
}
```

generates this assembly code:

```
movl 8(%ebp), %edx
movl $1, %eax
cmpl $1, %edx
jle .L7
.L10
imull %edx, %eax
subl $1, %edx
cmpl $1, %edx
jg .L10
.L7
```

For

The form of a for loop is

```
for (init; test; update)
    body;
```

which is expressed in terms of a while loop as

```
init;
while (test)
{
    body;
    update;
}
```

For example,

```
int fact_for(int n)
{
    int i;
    int result = 1;
    for (i = 2; i <= n; i++)
        result *= i;
    return result;
}
```

generates

```
movl 8(%ebp), %ecx
movl $2, %edx
```

```
movl $1, %eax
cmpl $1, %ecx
jle .L14
.L17
imull %edx, %eax
addl $1, %edx
cmpl %edx, %ecx
jge .L17
.L14
```

Switch-Case

Consider the C code

```
int switch_eg(int x, int n)
{
    int result = x;
    switch (n)
    {
        case 100: result *= 13;
                  break;
        case 102: result += 10;
        case 103: result += 11;
                  break;
        case 104:
        case 106: result *= result;
                  break;
        default: result = 0;
    }
}
```

The advantage of a switch-case statement over an if-else ladder is that there will be only one `goto` statement, rather than several.

The implementation of this uses an array of pointers, as in the following (where `&&` is nonstandard and refers to the address of a label):

```
int switch_eg_impl(int x, int n)
{
    static void *jt[7] = {&&loc_A, &&loc_def, &&loc_B, &&loc_C, &&loc_D, &&loc_def, &&loc_D]
    unsigned index = n - 100;
    int result;

    if (index > 6) goto loc_def;
    goto *jt[index];

    loc_def:
        result = 0;
```

```
        goto done;
    loc_C:
        result = x;
        goto rest;
    loc_A:
        result = x * 13;
        goto done;
    loc_B:
        result = x + 10;
    rest:
        result += 11;
        goto done;
    loc_D:
        result = x * x;
    done:
    return result;
}
```

The equivalent assembly code is

```
movl 8(%ebp), %edx
movl 12(%ebp), %eax

subl $100, %eax
cmpl $6, %eax
ja .L2
jmp *.L7(,%eax,4 )

-- Default
.L2:
movl $0, %eax
jmp .L8         -- done

.L5:
movl %edx, %eax
jmp .L9

.L3:
leal (%edx, %edx, 2), %eax
leal (%edx, %eax, 4), %eax
jmp .L8

.L4
leal 10(%edx), %eax

.L9
addl $11, %eax
```

```
jmp .L8

.L6
movl %edx, %eax
imull %edx, %eax

.L8
```

`.L7` is the location of the `jt` array, indicated in another part of the code as

```
.section .rodata
.align 4
.L7:
.long .L3
.long .L2
.long .L4
.long .L5
```

The `jmp *` (indirect jump) instruction jumps to the effective address of its argument; for instance, the above line `jmp *.L7(,%eax,4)` jumps to the instruction at `.L7 + 4 * %eax`.