Computer Architecture
Machine Representation of Programs: Control

Cain Susko

Queen's University
School of Computing

February 15, 2022

Conditional Branches

Equivalent to the 'if' statement that computers use. There are many types of conditional branches:

Jumps Known as jX instructions, these make the program jump to different parts of code depending on condition codes.

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

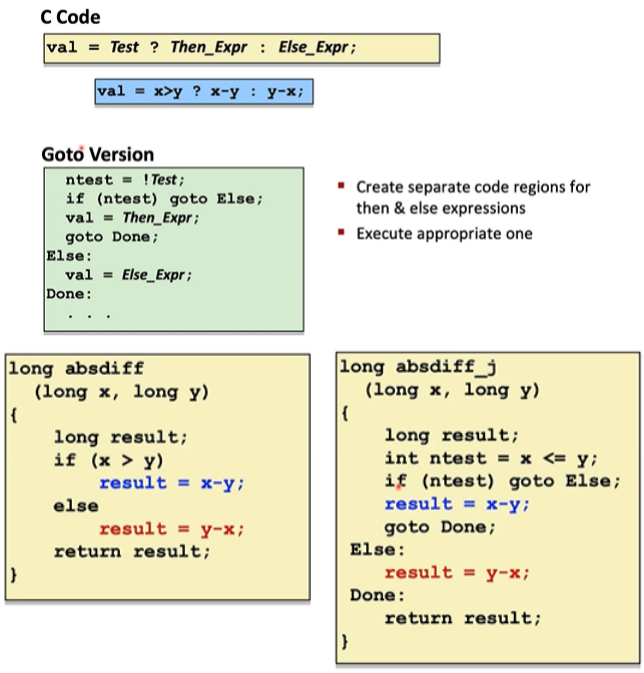
An example of an 'Old Sytle' Conditional Branch

```
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
   cmpq    %rsi, %rdi  # x:y
   jle     .L4
   movq    %rdi, %rax
   subq    %rsi, %rax
   ret
.L4:       # x <= y
   movq    %rsi, %rax
   subq    %rdi, %rax
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

This is old style because The GoTo Function was implemented in C which allows for the following:



The above programs are the old C code on the left compared to the New GoTo C code on the right. This makes the C code more functionally similar to the Assembly code. But because it is a hardcoded jump rather than a branch (like an if statement) the computer cannot pipeline (predict the program) efficiently which is a drain on performance.

Conditional Move

The following is an example of a Conditional Move (which is preferable to Goto):

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
absdiff:
   movq    %rdi, %rax  # x
   subq    %rsi, %rax  # result = x-y
   movq    %rsi, %rdx
   subq    %rdi, %rdx  # eval = y-x
   cmpq    %rsi, %rdi  # x:y
   cmovle  %rdx, %rax  # if <=, result = eval
   ret
```

Note that this is a different representation of the example on page 2 of this note which uses conditional moves rather than jumps.

Bad Cases There are a few cases where a conditional jump is not good to use. First, if the computation for the if statement is very complex the computational benefit for using a conditional statement is outweighed by the computational load used to compute the conditions.

```
val = Test(x) ?  HardComp1(x) :  HardComp2(x);.
```

When a computation is risky, a conditional branch may cause undefined or undesirable behaviour.

```
val = p ?  *p :  0;.
```

Finally, if the computations in each branch have 'side effects' then a conditional move should not be used.

```
val = x > 0 ?  x*=7 :  x+=3;.
```

This is because both of the values get computed and causes more computational resources to be used.

Conditional Branch - if statement summary

we now know that we can use two different methods of implementing if statements in code. They are:

- conditional Jumps
- conditional Moves

Loops

Loops can be implemented with simply with an inf statement. in C code, there are 3 categories of loops.

Do While

The first is the 'Do While' loop which does something while it is true. In C code this can be represented with either an if statement or using the Goto function:

C Code

```
long pcount_do
  (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

These are both equivalent; however the goto version makes the code unnecissarily complicated. The assembly code for this loop is:

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

```
   movl    $0, %eax    #  result = 0
.L2:                   # loop:
   movq    %rdi, %rdx
   andl    $1, %edx    #  t = x & 0x1
   addq    %rdx, %rax  #  result += t
   shrq    %rdi        #  x >>= 1
   jne     .L2         #  if (x) goto loop
   rep; ret
```

Below is the general form for **do while** loops in C:

C Code

```
do
  Body
  while (Test);
```

- **Body:**

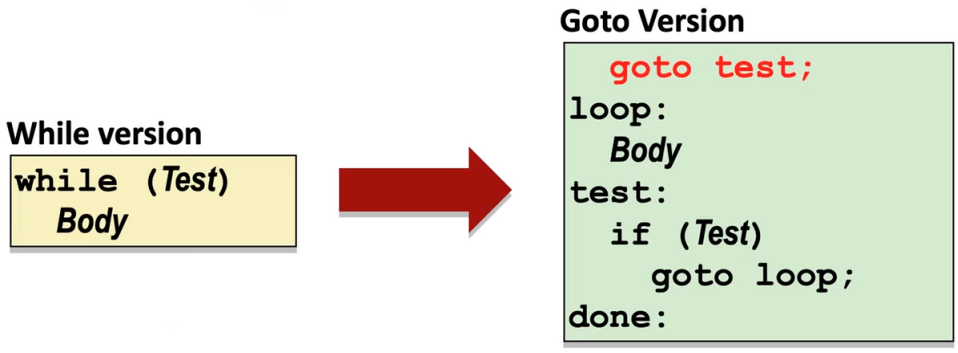
```
{
  Statement1;
  Statement2;
    ...
  Statementn;
}
```

Goto Version

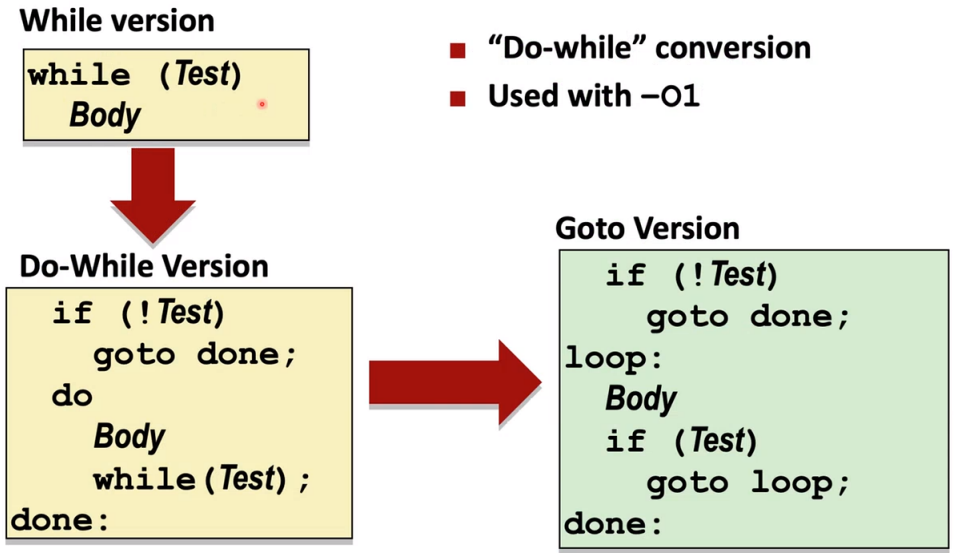
```
loop:
  Body
  if (Test)
    goto loop
```

While

While loops are similar to do-while for obvious reasons. The general form for a **while** loop in C is:



We can also generalize the while loop in terms of the do while loop like so:



Using this generalization we can reform a while loop to be a ‘do while’ for loop:

C Code

```
long pcount_while
  (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Do-While Version

```
long pcount_goto_dw
  (unsigned long x) {
  long result = 0;
  if (!x) goto done;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
 done:
  return result;
}
```

For

The for loop has the general form in C (and most other languages) is :

```
for (i = InitVal; i < Test; i++).
```

To convert a for loop to a ‘for while’ loop the general form is:

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

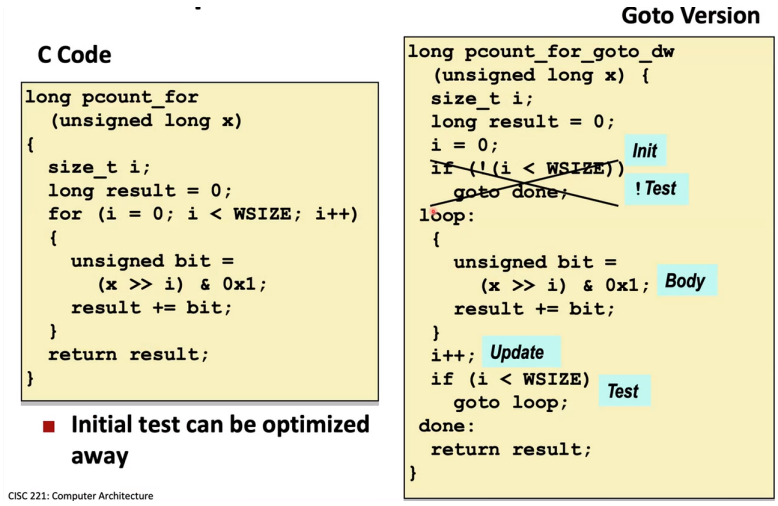
```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

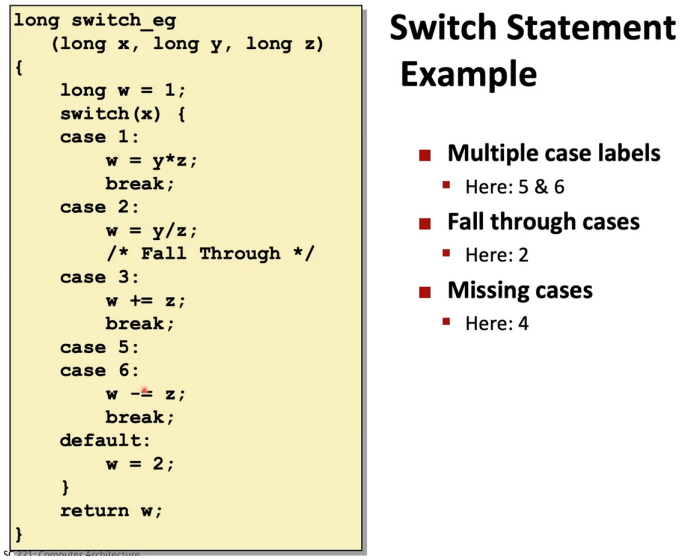
```
long pcount_for_while
  (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

We can also convert from for loop to a do while loop like so:

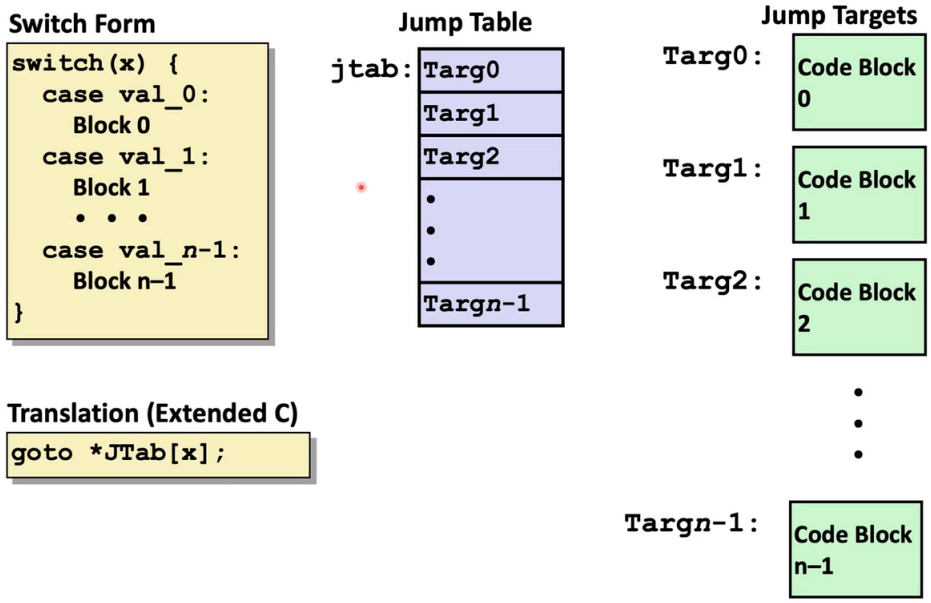


Switch Statement

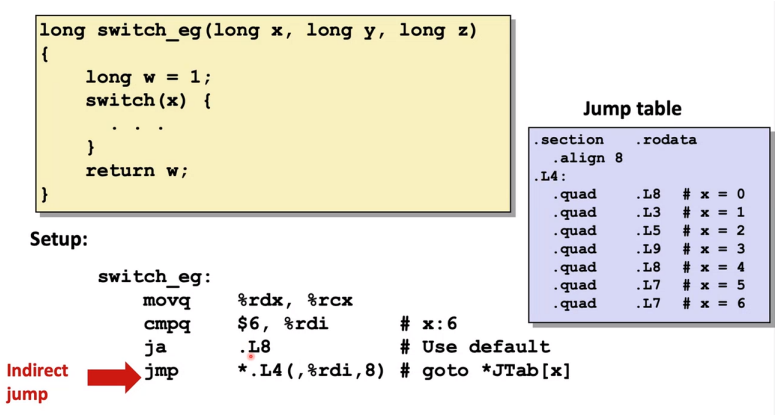
The final control structure we will cover is the switch statement.



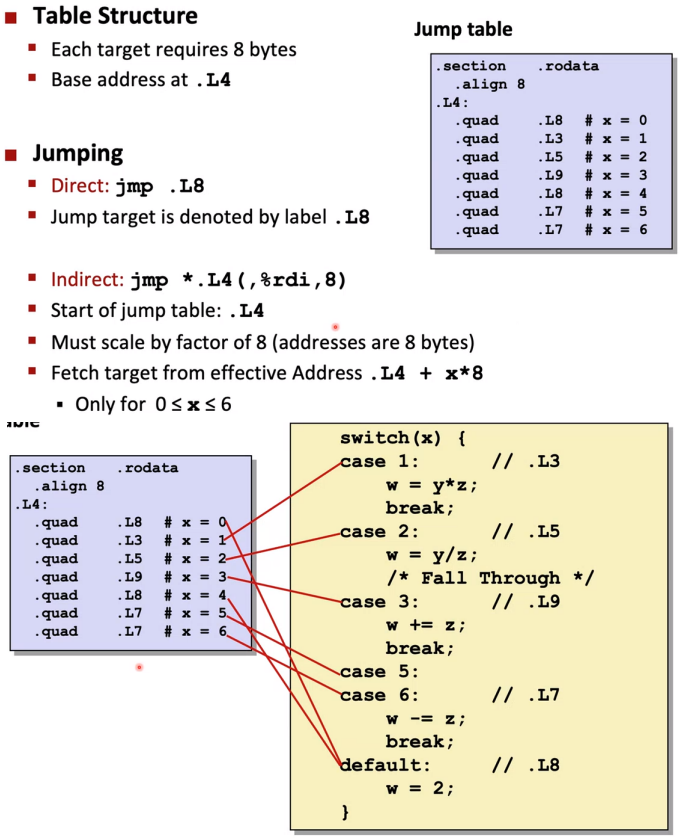
This structure is acheived by using a Jump Table with different identifiers. This is done like so:



Showing this with the example from before:



And thus, the general setup in assembly for a switch statement is:



Summary

In C we can implement Control by using:

- if-then-else
- do-while
- while, for
- switch

Within the Assembler there are 4 ways of achieveing these:

- conditional Jump

- conditional Move
- indirect jump (via jump table)
- compiler generates code sequence to implement more complex control

Some standard techniques for operating on control structures are:

- converting loops to do-while or 'lump-to-middle' form
- large switch statements use jump tables
- sparse switch statements may use desision trees (if-elseif-elseif-...)