Machine Level Programming: Control

CSCI 237: Computer Organization
15th Lecture, Wednesday, October 9

Kelly Shaw

1

1

Administrative Details

- Lab #3 due Wednesday after reading period at 11pm
  - Any questions?
- Read CSAPP 3.7-3.8
- Weekly quiz opens at 2:30 and due by Friday at 2:30
- Midterm in lab on Wednesday 10/23
  - Closed book and closed notes
  - Includes material from today (including short video)
  - Review session on Monday 10/21 in the evening?
- Apply to be a TA!
- TA feedback forms
- CS Colloquium talk on Friday at 2:35pm in Wege

2

2

Last Time: Machine-Level Programming: Control

- Intro to data-dependent control
  - Condition codes
  - Conditional branches
  - Conditional data
  - Loops
  - Switch Statements

3

3

Today: Machine-Level Programming: Control

- Intro to data-dependent control
  - Condition codes
  - Conditional branches
  - Conditional data
  - Loops
  - Switch Statements
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

4

4

General "Do-While" Translation

C Code

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

Goto Version

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

- Body:

```
{
  Statement1;
  Statement2;
    …
  Statementn;
}
```

5

5

Practice on Your Own

- Convert this assembly code to C code

```
fcn:
        movq $5, %r9
        movq $1, %r8
        movq $0, %rax
L2:     addq (%rdi), %rax
        incq %r8
        addq $8, %rdi
        cmpq %r8, %r9
        jg L2
L3:
        ret
```

%rdi will have a long * passed into it

6

6

General "While" Translation #1

- "Jump-to-middle" translation
- Used with **–Og** compile flag

While version

```
while (Test)
  Body
```

Goto Version

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

7

7

While Loop Example #1

C Code

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

Jump to Middle

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

- Compared to do-while version of function, execution seems to "jump to the middle" (hence the name)
- Initial **goto** starts loop at **test**

8

8

General "While" Translation #2

While version

```
while (Test)
  Body
```

- "Do-while" conversion ("guarded-do")
- Used with `-O1` compile flag

Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

9

9

While Loop Example #2

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;
}
```

- Compared to do-while version of function, looks very similar except for initial test
- Initial conditional "guards" entrance to loop

10

10

"For" Loop Form

General Form

```
for (Init; Test; Update)
    Body
```

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

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

11

11

"For" Loop → While Loop

For Version

```
for (Init; Test; Update)
    Body
```

While Version

```
Init;
while (Test) {
    Body
    Update;
}
```

12

12

For-While Conversion

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;
}
```

13

13

"For" Loop Do-While Conversion

C Code

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

- Initial test can be optimized away since WSIZE is sizeof(int) is never 0

Goto Version

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               !Test
    goto done;
 loop:
  {
    unsigned bit =
      (x >> i) & 0x1;             Body
    result += bit;
  }
  i++;                            Update
  if (i < WSIZE)                  Test
    goto loop;
 done:
  return result;
}
```

14

14

"For" Loop Do-While Conversion

C Code

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

Without { }, the compiler complains about declaration of "bit" (must be at top of function in C).

- Initial test can be optimized away since WSIZE is sizeof(int) is never 0

Goto Version

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               !Test
    goto done;
 loop:
  {
    unsigned bit =
      (x >> i) & 0x1;             Body
    result += bit;
  }
  i++;                            Update
  if (i < WSIZE)                  Test
    goto loop;
 done:
  return result;
}
```

15

15

Practice on Your Own

- Convert this C code to assembly

```
long fcn(long *arr)
{
  long count = 0;

  for(long i = 0; arr[i] != 0; i++){
    count++;
  }

  return count;
}
```

16

16

Today: Machine-Level Programming: Control

- Intro to data-dependent control
  - Condition codes
  - Conditional branches
  - Conditional data
  - Loops
  - Switch Statements

17

17

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Switch Statement Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

18

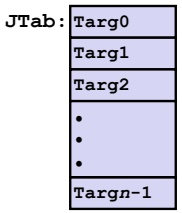
18

Jump Table Structure

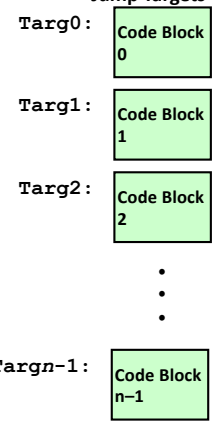
Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table



Jump Targets



Translation (Extended C)

```
goto *JTab[x];
```

19

19

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja      .L8         # x > |table|
    jmp     *.L4(,%rdi,8) # scale
```

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

Note that w not initialized here

20

20

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section    .rodata
  .align 8
.L4:
   .quad    .L8   # x = 0
   .quad    .L3   # x = 1
   .quad    .L5   # x = 2
   .quad    .L9   # x = 3
   .quad    .L8   # x = 4
   .quad    .L7   # x = 5
   .quad    .L7   # x = 6
```

Setup:

```
    switch_eg:
        movq    %rdx, %rcx
        cmpq    $6, %rdi        # x:6
        ja      .L8             # Use default
        jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

Indirect jump

21

21

Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes
  - Base address at `.L4`
- Jumping
  - Direct: `jmp .L8`
  - Jump target is denoted by label `.L8`
  - Indirect: `jmp *.L4(,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective Address `.L4 + x*8`
    - Only for $0 \le x \le 6$

Jump table

```
.section    .rodata
  .align 8
.L4:
   .quad    .L8   # x = 0
   .quad    .L3   # x = 1
   .quad    .L5   # x = 2
   .quad    .L9   # x = 3
   .quad    .L8   # x = 4
   .quad    .L7   # x = 5
   .quad    .L7   # x = 6
```

22

22

Jump Table

Jump table

```
.section    .rodata
  .align 8
.L4:
   .quad    .L8   # x = 0
   .quad    .L3   # x = 1
   .quad    .L5   # x = 2
   .quad    .L9   # x = 3
   .quad    .L8   # x = 4
   .quad    .L7   # x = 5
   .quad    .L7   # x = 6
```

```
switch(x) {
case 1:       // .L3
    w = y*z;
    break;
case 2:       // .L5
    w = y/z;
    /* Fall Through */
case 3:       // .L9
    w += z;
    break;
case 5:
case 6:       // .L7
    w -= z;
    break;
default:      // .L8
    w = 2;
}
```

23

23

Code Blocks (x == 1)

```
 switch(x) {
 case 1:       // .L3
       w = y*z;
       break;
   . . .
}
```

```
.L3:
   movq    %rsi, %rax   # y
   imulq   %rdx, %rax   # y*z
   ret
```

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

24

24

Handling Fall-Through

```
long w = 1;
   . . .
switch(x) {
   . . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
   . . .
}
```

```
case 2:
    w = y/z;
    goto merge;
```

```
case 3:
    w = 1;
merge:
    w += z;
```

25

25