CSCI2467: Systems Programming Concepts

Slideset 6: Machine Level II: Control

Source: CS:APP Section 3.6, Bryant & O'Hallaron

Course Instructors:

Matthew Toups
Caitlin Boyce

Course Assistants:

Saroj Duwal
David McDonald

Spring 2020

THE UNIVERSITY of NEW ORLEANS

DEPARTMENT OF COMPUTER SCIENCE

Bomblab reminders

- Bomblab in progress!

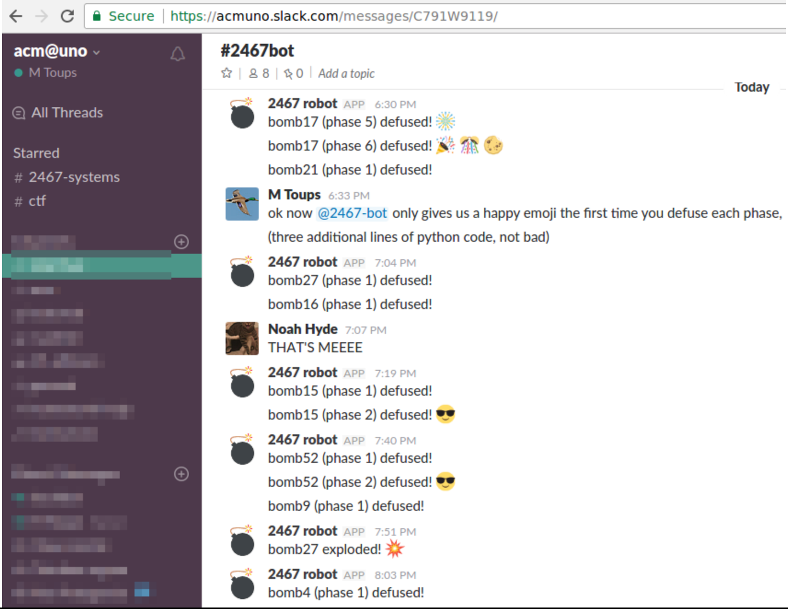
  Due: **Monday** February 17, 11:59pm

  - scoreboard shows score (if positive), no explicit handin required
- breakpoints breakpoints breakpoints!

  with breakpoints set, bomb should *never* (fully) explode!
- more help available: Tues/Thurs 1-4pm (Math 209)

  or, other times at helpdesk (Math 319)

Bomblab live updates via slack



Highlight from previous slides

- `lea` instruction:
  - address computations
  - *or also*, can be used for simple arithmetic
- calling convention (`rdi, rsi, rdx, ...`)
  - in `x86-64`, function arguments go into registers instead of pushed onto stack

Class updates

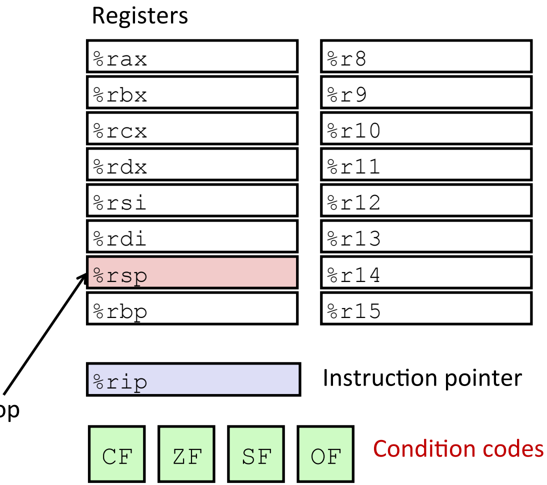
1 Control

- Condition codes
- Conditional branches
- Loops
- Switch statement

Processor state

x86-64 (partial)

- **Information about currently executing program**
  - Temporary data (`%rax`, …)
  - Location of runtime stack (`%rsp`)
  - Location of current code control point (`%rip`, …)
  - Status of recent tests (CF, ZF, SF, OF)



Condition codes are set implicitly

After arithmetic instructions

Single bit registers

CF	Carry Flag (unsigned)	SF	Sign Flag (signed)
ZF	Zero Flag	OF	Overflow Flag (signed)

Implicitly set by arithmetic operations

Example: `add dest,src` $\leftrightarrow t = a + b$

- CF set if carry out from most significant bit (unsigned overflow)
- ZF set if `t == 0`
- SF set if `t` <0 (signed)

Condition codes are set explicitly: `cmp`
Using Compare instruction

Single bit registers

CF	Carry Flag (unsigned)	SF	Sign Flag (signed)
ZF	Zero Flag	OF	Overflow Flag (signed)

Explicitly set by compare instruction

Example: `cmp src1,src2` $\rightarrow a - b$ (without setting destination)

- CF set if carry out from most significant bit (unsigned compare)
- ZF set if `a == b`
- SF set if `(a-b)` <0 (signed)
- OF set if two's complement (signed) overflow

`(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

Condition codes are set explicitly: `test`
Using Test instruction

Single bit registers

CF	Carry Flag (unsigned)	SF	Sign Flag (signed)
ZF	Zero Flag	OF	Overflow Flag (signed)

Explicitly set by test instruction

Example: `test src1,src2` $\rightarrow a\&b$ (without setting destination)

- sets condition codes based on value of `src1` & `src2`
- useful to have one of the operands be a mask

ZF set if `a & b == 0`

SF set if `a & b` <0

Reading condition codes

SetX Instructions

- Set low-order byte of destination to 0 or 1 based on combinations of condition codes
- Does not alter remaining 7 bytes

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

x86-64 registers
with low-order byte

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

- Can reference low-order byte

Reading condition codes

- **SetX Instructions:**
  - Set single byte based on combination of condition codes
- **One of addressable byte registers**
  - Does not alter remaining bytes
  - Typically use `movzbl` to finish job
    - 32-bit instructions also set upper 32 bits to 0

```
int gt (long x, long y)
{
  return x > y;
}
```

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

```
cmp   rdi, rsi  # compare x:y
setg  al        # set when >
movzx eax, al   # zero the rest of rax
ret
```

Conditional jumps

- **jX Instructions**
  - Jump to different part 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)

Conditional branch example

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

```
absdiff:
   cmp     rdi, rsi
   jle     .L2
   mov     rax, rdi
   sub     rax, rsi
   ret
.L2:    # x <= y
   mov     rax, rsi
   sub     rax, rdi
   ret
```

Compiled with:

gcc −Og −S absdiff.c −masm=intel

Register	Use
rdi	argument x
rsi	argument y
rax	return value

Branching with goto

C allows goto statement

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

```
long absdiff_j
  (long x, long y)
{
  long result;
  int ntest = x <= y;
  if (ntest) goto Else;
  result = x-y;
  goto Done;
 Else:
  result = y-x;
 Done:
  return result;
}
```

Conditional expression translation
(using branches)

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
   ntest = !Test;
   if (ntest) goto Else;
   val = Then_Expr;
   goto Done;
Else:
   val = Else_Expr;
Done:
   . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

Conditional Move Instructions

- Instruction supports:

  if (Test) Dest ← Src
- Supported in post-1995 x86 processors
- GCC tries to use them
  - But, only when known to be safe

Why?

- Branches are very disruptive to instruction flow through pipelines
- Conditional moves do not require control transfer

C Code

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Goto Version

```
  result = Then_Expr;
  eval = Else_Expr;
  nt = !Test;
  if (nt) result = eval;
  return result;
```

Conditional move example

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

```
mov     rdx, rdi # x
mov     rax, rsi # y
sub     rdx, rsi # rdx <- x-y
sub     rax, rdi # rax <- y-x
cmp     rdi, rsi # x ? y
cmovg   rax, rdx #
# if >, result= x-y (in rdx)
```

Register	Use
rdi	argument x
rsi	argument y
rax	return value

Bad cases for conditional move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- **Both values get computed**
- **Only makes sense when computations are very simple**

Risky Computations

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

- Both values get computed
- May have undesirable effects

Computations with side effects

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

- Both values get computed
- Must be side-effect free

"Do-While" loop example

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;
}
```

- **Count number of 1's in argument x ("popcount")**
- **Use conditional branch to either continue looping or to exit loop**

“Do-While” loop compilation

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

```
  mov   eax, 0    # result=0
.L2:              # looptop:
  mov   rdx, rdi
  and   edx, 1    # t=x & 0x1
  add   rax, rdx  # result+=t
  shr   rdi  # x >>= 1
  jne   .L2  # if(x) goto L2
  rep   ret  # wtf return
```

Register	Use
rdi	argument x
rax	return value

General "Do-While" translation

C Code

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

Goto Version

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

- **Body:**

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

General "Do-While" translation #1

- "Jump-to-middle" translation
- Used with **-Og**

While version

```
while (Test)
  Body
```

Goto Version

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

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;
}
```

- **Compare to do-while version of function**
- **Initial goto starts loop at test**

General "While" translation #2

While version

```
while (Test)
   Body
```

- "Do-while" conversion
- Used with **-O1**

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:
```

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;
}
```

- Compare to do-while version of function
- Initial conditional guards entrance to loop

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;
}
```

For Version

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

While Version

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

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;
}
```

for loop → do-while conversion

Goto Version

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

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

ıryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

- Class updates

1. Control
   - Condition codes
   - Conditional branches
   - Loops
   - Switch statement

```
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;
}
```

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Switch Statement Example

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

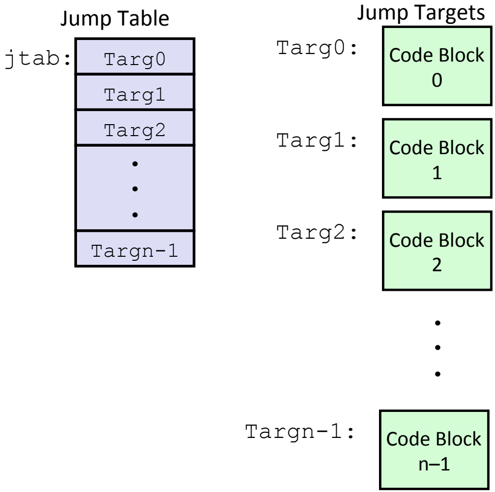
Jump table structure

Switch Form

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

Translation (Extended C)

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



Switch statement example

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

```
cmp   rdi, 6   # x:6
ja    .L8      # default case
jmp  [QWORD PTR .L4[0+rdi*8]]
```

Register	Use
rdi	argument x
rsi	argument y
rdx	argument z
rax	return value

.L8 is default. What values jump there?

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              # Use default
    jmp     *.L4(,%rdi,8)    # goto *JTab[x]
```

Indirect jump

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
```

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 $0 \leq x \leq 6$

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
```

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;
}
```

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

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;
```

Code blocks (x==2, x==3)

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

```
.L5:                          # Case 2
   movq    %rsi, %rax
   cqto
   idivq   %rcx            #  y/z
   jmp     .L6             #  goto merge
.L9:                          # Case 3
   movl    $1, %eax        #  w = 1
.L6:                          # merge:
   addq    %rcx, %rax      #  w += z
   ret
```

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

Code blocks (x==5, x==6, default)

```
switch(x) {
  . . .
  case 5:  // .L7
  case 6:  // .L7
      w -= z;
      break;
  default: // .L8
      w = 2;
}
```

```
.L7:                   # Case 5,6
  movl  $1, %eax    #  w = 1
  subq  %rdx, %rax #  w -= z
  ret
.L8:                   # Default:
  movl  $2, %eax    #  2
  ret
```

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

Summary: control at the machine level

- C Control
  - if-then-else
  - do-while
  - while, for
  - switch
- Assembler Control
  - Conditional jump
  - Conditional move
  - Indirect jump (via jump tables)
  - Compiler generates code sequence to implement more complex control
- Standard Techniques
  - Loops converted to do-while or jump-to-middle form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (if-elseif-elseif-else)