[CSED211] Introduction to Computer Software Systems

Lecture 5: Control

Prof. Jisung Park

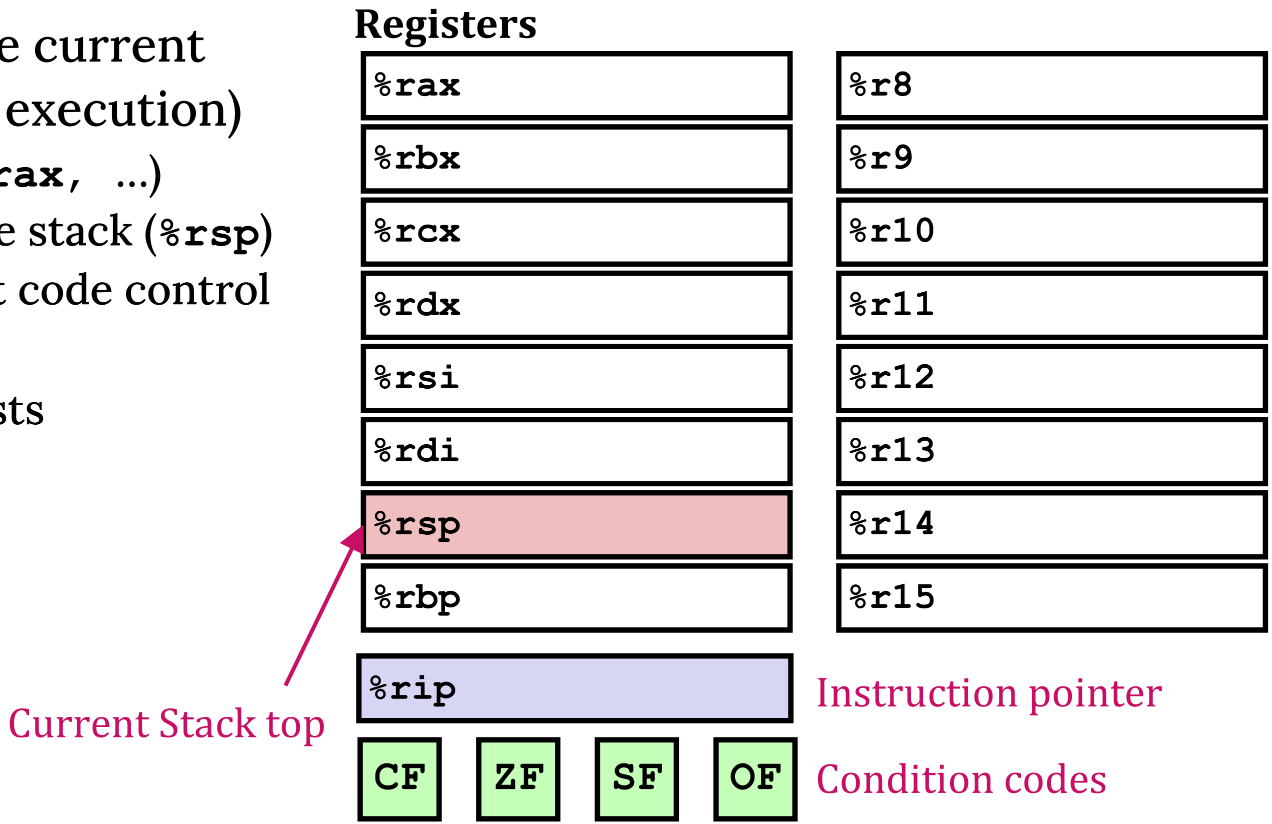
2023.09.25

Today

- Control: Condition Codes
- Conditional Branches
- Loops
- Switch Statements

Processor State (x86-64, Partial)

- Information about the current CPU status (program execution)
  - 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: Implicit Setting

- Single bit registers
  - **`CF`**: Carry Flag (for unsigned)
  - **`SF`**: Sign Flag (for signed)
  - **`ZF`**: Zero Flag
  - **`OF`**: Overflow Flag (for signed)

- Implicitly set (think of it as side effect) by arithmetic instructions

  Example: **`addq src, dst`** → **`t = a + b`**

  `CF` set if carry/borrow out from most significant bit (unsigned overflow)

  `ZF` set if **`t == 0`**

  `SF` set if **`t < 0`** (as signed)

  `OF` set if two's-complement (signed) overflow

  ```
  (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
  ```

- Not set by **`leaq`** instruction

Condition Codes: Explicit Setting - Compare

- Explicit setting by compare instruction
  - **`cmpq src2, src1`**
  - **`cmpq b,a`** is like computing **`(a-b)`** without setting destination

    `CF` set if carry/borrow out from the MSB (used for unsigned comparisons)

    `ZF` set if `(a-b) == 0`, i.e., **`a = b`**

    `SF` set if **`(a-b)`** `< 0` (as 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: Explicit Setting - Test

- Explicit setting by test instruction
  - **`testq src2, src1`**
  - **`testq b,a`** is like computing **`(a&b)`** without setting destination
    - Useful to have one of the operands be a mask

    `ZF` set when **`a&b == 0`**

    `SF` set when **`a&b < 0`**

    `CF`, **`OF`** clear

- Can be used to check if the content is zero: **`testq %rax, %rax`**

Condition Codes: Explicit Reading - Set

- Explicit reading by **`setX`** instructions
  - **`setX dst`**: sets the least-significant byte of destination **`dst`** to `0` or `1` based on combinations of condition codes
  - Does not alter the remaining 7 bytes of **`dst`**

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 Integer Registers

- Can reference the least-significant byte

%rax	%al
%rbx	%bl
%rcx	%cl
%rdx	%dl
%rsi	%sil
%rdi	%dil
%rps	%spl
%rbp	%bpl

%r8	%r8b
%r9	%r9b
%r10	%r10b
%r11	%r11b
%r12	%r12b
%r13	%r13b
%r14	%r14b
%r15	%r15b

Explicit Reading Condition Codes (Cont.)

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

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

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

```
cmpq    %rsi, %rdi     # Compare x and y
setg    %al            # Set when x > y
movzbl  %al, %eax.     # Zero rest of %rax
ret
```

Explicit Reading Condition Codes (Cont.)

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

Beware weirdness `movzbl` (and others, i.e., `movxyz`)

`movzbl %al, %eax`

%rax	%eax	%al

%rax	Return value

```
cmpq    %rsi, %rdi     # Compare x and y
setg    %al            # Set when x > y
movzbl  %al, %eax.     # Zero rest of %rax
ret
```

Today

- Control: Condition Codes
- Conditional Branches
- Loops
- Switch Statements

Jumping

- `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: Old Style

- Generation

```
$ gcc –Og -S –fno-if-conversion control.c
```

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

```c
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
```

Expressing with Goto Code

- C supports `goto` statement
  - Jump to position designated by label

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

```
long absdiff(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;
}
```

General Conditional Expression Translation

- C code using branch

```
val = test ? then_expr : else_expr;
```

- 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 Moves

- Conditional move instructions
  - Instruction supports:

    ```
    if (test) dst = 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

Branch Version

```
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;
}
```

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

```
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
```

Conditional Move Example

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

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

Bad Cases for Conditional Move

- Expensive computations

  ```
  val = test(x) ? hard1(x) : hard2(x);
  ```

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

Bad Performance

- Risky computations

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

  - Both expressions get computed
  - May have undesirable effects (e.g., null-pointer exception)

Unsafe

- Computations with side effects

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

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

Illegal

Exercise

```
xorq    %rax, %rax
subq    $1, %rax
cmpq    $2, %rax
setl    %al
movzbl  %al, %eax
```

%rax	SF	CF	OF	ZF

- **`cmpq b, a` → `(a-b)`** w/o setting dst

  `CF` set if carry/borrow out from MSB
  (used for unsigned comparisons)

  `ZF` set if `a == b`

  `SF` set if `(a-b) < 0` (as signed)

  `OF` set if two's-complement (signed) overflow
- **`test b, a` → `(a&b)`** w/o setting dest

  `SF`, `ZF` set based on result

  `CF`, `OF` cleared
- `setl` and `movzbl` do not modify condition codes

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)

Exercise

```
xorq     %rax, %rax
subq     $1, %rax
cmpq     $2, %rax
setl     %al
movzbl   %al, %eax
```

%rax	SF	CF	OF	ZF
0000 0000 0000 0000	0	0	0	1

- **cmpq b, a** → **(a-b)** w/o setting dst
  CF set if carry/borrow out from MSB
  (used for unsigned comparisons)
  ZF set if **a == b**
  SF set if **(a-b) < 0** (as signed)
  OF set if two's-complement (signed) overflow
- **test b, a** → **(a&b)** w/o setting dest
  SF, **ZF** set based on result
  CF, **OF** cleared
- **setl** and **movzbl** do not modify condition codes

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)

Exercise

```
xorq    %rax, %rax
subq    $1, %rax
cmpq    $2, %rax
setl    %al
movzbl  %al, %eax
```

%rax	SF	CF	OF	ZF
0000 0000 0000 0000	0	0	0	1
FFFF FFFF FFFF FFFF	1	1	0	0

- **`cmpq b, a`** → **`(a-b)`** w/o setting dst

  `CF` set if carry/borrow out from MSB (used for unsigned comparisons)

  `ZF` set if **`a == b`**

  `SF` set if **`(a-b) < 0`** (as signed)

  `OF` set if two's-complement (signed) overflow
- **`test b, a`** → **`(a&b)`** w/o setting dest

  `SF`, **`ZF`** set based on result

  `CF`, **`OF`** cleared
- **`setl`** and **`movzbl`** do not modify condition codes

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)

Exercise

```
xorq    %rax, %rax
subq    $1, %rax
cmpq    $2, %rax
setl    %al
movzbl  %al, %eax
```

%rax	SF	CF	OF	ZF
0000 0000 0000 0000	0	0	0	1
FFFF FFFF FFFF FFFF	1	1	0	0
FFFF FFFF FFFF FFFF	1	0	0	0

- **cmpq b, a** → **(a-b)** w/o setting dst
  CF set if carry/borrow out from MSB
  (used for unsigned comparisons)
  ZF set if **a == b**
  SF set if **(a-b) < 0** (as signed)
  OF set if two's-complement (signed) overflow
- **test b, a** → **(a&b)** w/o setting dest
  SF, **ZF** set based on result
  CF, **OF** cleared
- **setl** and **movzbl** do not modify condition codes

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)

Exercise

```
xorq     %rax, %rax
subq     $1, %rax
cmpq     $2, %rax
setl     %al
movzbl   %al, %eax
```

%rax	SF	CF	OF	ZF
0000 0000 0000 0000	0	0	0	1
FFFF FFFF FFFF FFFF	1	1	0	0
FFFF FFFF FFFF FFFF	1	0	0	0
FFFF FFFF FFFF FF01	1	0	0	0

- **`cmpq b, a` → `(a-b)`** w/o setting dst
  - `CF` set if carry/borrow out from MSB (used for unsigned comparisons)
  - `ZF` set if `a == b`
  - `SF` set if `(a-b) < 0` (as signed)
  - `OF` set if two's-complement (signed) overflow
- **`test b, a` → `(a&b)`** w/o setting dest
  - `SF`, `ZF` set based on result
  - `CF`, `OF` cleared
- **`setl`** and **`movzbl`** do not modify condition codes

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)

Exercise

```
xorq    %rax, %rax
subq    $1, %rax
cmpq    $2, %rax
setl    %al
movzbl  %al, %eax
```

%rax	SF	CF	OF	ZF
0000 0000 0000 0000	0	0	0	1
FFFF FFFF FFFF FFFF	1	1	0	0
FFFF FFFF FFFF FFFF	1	0	0	0
FFFF FFFF FFFF FF01	1	0	0	0
0000 0000 0000 0001	1	0	0	0

- **`cmpq b, a`** → **`(a-b)`** w/o setting dst

  `CF` set if carry/borrow out from MSB

  (used for unsigned comparisons)

  `ZF` set if **`a == b`**

  `SF` set if **`(a-b) < 0`** (as signed)

  `OF` set if two's-complement (signed) overflow
- **`test b, a`** → **`(a&b)`** w/o setting dest

  `SF`, **`ZF`** set based on result

  `CF`, **`OF`** cleared
- **`setl`** and **`movzbl`** do not modify condition codes

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)

Today

- Control: Condition Codes
- Conditional Branches
- Loops
- Switch Statements

Do-While Loop Example

Do-While Version

```
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

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
   ret
```

General Do-While Loop Translation

Do-While Version

```
do
  body_expr
  while(test);
```

Goto Version

```
Loop:
  body_expr
  if(test) goto Loop;
```

body_expr:

```
{
  statement_1;
  statement_2;
    …
  statement_n;
}
```

General While Translation#1

- Jump-to-middle translation
- Used with `-Og`

While Version

```
while(test)
  body_expr
```

Goto Version

```
  goto Test;
Loop:
  body_expr
Test:
  if(test) goto Loop;
```

While Loop Example#1

While Version

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

Goto (Jump-to-Middle) Version

```
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 the function
- Initial `goto` starts loop at `Test`

General While Translation #2

- Do-while conversion
- Used with `-O1`

While Version

```
while(test)
  body_expr
```

Do-While Version

```
  if(!test)
    goto Done;
  do
    body_expr
    while(test)
Done:
```

Goto Version

```
  if(!test)
    goto Done;
Loop:
  body_expr
  if(test) goto Loop;
Done:
```

While Loop Example#1

While Version

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

Goto Version (via Do-While Conversion)

```
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 the function
- Initial `goto` starts loop at `Test`

While Loop Example#1

Goto (Jump-to-Middle) Version

```
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;
}
```

Goto Version (via Do-While Conversion)

```
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 the function
- Initial `goto` starts loop at `Test`

General For Loop Translation

For Loop Version

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

```
#define WSIZE 8*sizeof(long)
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;
}
```

General For Loop Translation: While Conversion

For Loop Version

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

```
#define WSIZE 8*sizeof(long)
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;
}
```

While Version

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

```
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++;
  }
  …
```

For Loop Do-While Conversion

For Loop Version

```
#define WSIZE 8*sizeof(long)
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;
}
```

Goto Version (via Do-While Conversion)

```
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;                   // body
  i++;  // update
  if(i < WSIZE) goto Loop; // test
Done:
  return result;
}
```

For Loop Do-While Conversion

For Loop Version

```c
#define WSIZE 8*sizeof(long)
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;
}
```

Goto Version (via Do-While Conversion)

```c
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;                  // body
  i++;  // update
  if(i < WSIZE) goto Loop; // test
Done:
  return result;
}
```

- Initial test can be optimized away

Today

- Control: Condition Codes
- Conditional Branches
- Loops
- Switch Statements

Switch Statement Example

- Multiple case labels
  - e.g., cases 5 & 6
- Fall through cases
  - e.g., case 2
- Missing cases
  - e.g., case 4

```
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;
}
```

Jump Table Structure

Switch Form

```
switch(x) {
  case val_0:
    cblock0
  case val_1:
    cblock1
     …
  case val_n-1:
    cblock(n-1)
}
```

Translation (Extended C)

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

Jump Table

```
JTab: Targ0
      Targ1
      Targ2
      ...
      Targ(n-1)
```

Jump Targets

```
Targ0:      Code Block 0
Targ1:      Code Block 1
Targ2:      Code Block 2
...
Targ(n-1):  Code Block (n-1)
```

Switch Statement Example

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

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

Setup

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

Note that w is not initialized here

Switch Statement Example

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

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

Setup

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

Indirect Jump

Jump table

```
.section    .rodata # read only
  .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)`
    - Must scale by factor of 8: an address (pointer) is 8 bytes
    - Fetch target from the effective address `.L4 + x*8`, only for $0 \le x \le 6$

Jump table

```
.section    .rodata # read only
  .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: with the Example C Code

```
long switch_eg(long x, long y, long z){
  long w = 1;
  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:       // .L7
  case 6:       // .L7
    w -= z; break;
  default:      // .L8
    w = 2;
  }
  return w;
}
```

Jump table

```
.section    .rodata # read only
  .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
```

Code Blocks: x==1

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

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

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

Handling Fall-Through

```
switch(x) {
…
case 2:       // .L5
  w = y/z;    // Fall through
case 3:       // .L9
  w += z; break;
```

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

```
case 3:
  w = 1;
```

```
Merge:
  w += z;
```

Code Blocks: `x == 2` and `x == 3`

```
switch(x) {
…
case 2:       // .L5
  w = y/z;   // Fall through
case 3:       // .L9
  w += z; break;
```

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

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

Instruction	Effect	Description
imulq S	R[%rdx]:R[%rax] ← S × R[%rax]	Signed full multiply
mulq S	R[%rdx]:R[%rax] ← S × R[%rax]	Unsigned full multiply
cltq	R[%rax] ← SignExtend(R[%eax])	Convert %eax to quad word
cqto	R[%rdx]:R[%rax] ← SignExtend(R[%rax])	Convert to oct word
idivq S	R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] ÷ S	Signed divide
divq S	R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] ÷ S	Unsigned divide

Code Blocks: `x == 5`, `x == 6` and `default`

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

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

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

Finding Jump Table in Binary

```
 00000000004005e0 <switch_eg>:
4005e0:       48 89 d1                mov    %rdx,%rcx
4005e3:       48 83 ff 06             cmp    $0x6,%rdi
4005e7:       77 2b                   ja     400614 <switch_eg+0x34>
4005e9:       ff 24 fd f0 07 40 00    jmpq   *0x4007f0(,%rdi,8)
4005f0:       48 89 f0                mov    %rsi,%rax
4005f3:       48 0f af c2             imul   %rdx,%rax
4005f7:       c3                      retq
4005f8:       48 89 f0                mov    %rsi,%rax
4005fb:       48 99                   cqto
4005fd:       48 f7 f9                idiv   %rcx
400600:       eb 05                   jmp    400607 <switch_eg+0x27>
400602:       b8 01 00 00 00          mov    $0x1,%eax
400607:       48 01 c8                add    %rcx,%rax
40060a:       c3                      retq
40060b:       b8 01 00 00 00          mov    $0x1,%eax
400610:       48 29 d0                sub    %rdx,%rax
400613:       c3                      retq
400614:       b8 02 00 00 00          mov    $0x2,%eax
400619:       c3                      retq
```

Finding Jump Table in Binary

```
             00000000004005e0 <switch_eg>:
            4005e0:      48 89 d1                mov    %rdx,%rcx
            4005e3:      48 83 ff 06             cmp    $0x6,%rdi
            4005e7:      77 2b                   ja     400614 <switch_eg+0x34>
            4005e9:      ff 24 fd f0 07 40 00    jmpq   *0x4007f0(,%rdi,8)
x==1        4005f0:      48 89 f0
            4005f3:      48 0f af c2
            4005f7:      c3
x==2        4005f8:      48 89 f0
            4005fb:      48 99
            4005fd:      48 f7 f9
            400600:      eb 05                   jmp    400607 <switch_eg+0x27>
x==3        400602:      b8 01 00 00 00          mov    $0x1,%eax
            400607:      48 01 c8                add    %rcx,%rax
            40060a:      c3                      retq
x==5, x==6  40060b:      b8 01 00 00 00          mov    $0x1,%eax
            400610:      48 29 d0                sub    %rdx,%rax
            400613:      c3                      retq
default     400614:      b8 02 00 00 00          mov    $0x2,%eax
            400619:      c3                      retq
```

```
 % gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:     0x0000000000400614     0x00000000004005f0
0x400800:     0x00000000004005f8     0x0000000000400602
0x400810:     0x0000000000400614     0x000000000040060b
0x400820:     0x000000000040060b     0x2c646c25203d207
```

Finding Jump Table in Binary

```
             00000000004005e0 <switch_eg>:
             4005e0:   48 89 d1               mov    %rdx,%rcx
             4005e3:   48 83 ff 06            cmp    $0x6,%rdi
             4005e7:   77 2b                  ja     400614 <switch_eg+0x34>
             4005e9:   ff 24 fd f0 07 40 00   jmpq   *0x4007f0(,%rdi,8)
x==1         4005f0:   48 89 f0               mov    %rsi,%rax
             4005f3:   48 0f af c2            imul   %rdx,%rax
             4005f7:   c3                     retq
x==2         4005f8:   48 89 f0               mov    %rsi,%rax
             4005fb:   48 99                  cqto
             4005fd:   48 f7 f9               idiv   %rcx
             400600:   eb 05                  jmp    400607 <switch_eg+0x27>
x==3         400602:   b8 01 00 00 00         mov    $0x1,%eax
             400607:   48 01 c8               add    %rcx,%rax
             40060a:   c3                     retq
x==5, x==6   40060b:   b8 01 00 00 00         mov    $0x1,%eax
             400610:   48 29 d0               sub    %rdx,%rax
             400613:   c3                     retq
default      400614:   b8 02 00 00 00         mov    $0x2,%eax
             400619:   c3                     retq
```

Summary

- 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)

[CSED211] Introduction to Computer Software Systems

Lecture 5: Control

Prof. Jisung Park

COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.09.25