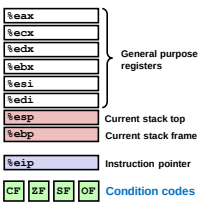
The HW/SW Interface

The x86 ISA: Control Transfer Structures

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


Recap: Condition Codes

- Single bit registers
  - **CF** Carry Flag (for unsigned)
  - **SF** Sign Flag (for signed)
  - **ZF** Zero Flag
  - **OF** Overflow Flag (for signed)
- Implicitly set by arithmetic operations
  - Example: `addl/addq Src,Dest` ↔ `t = a+b`
  - **CF set** if carry 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 `lea` instruction
- Explicitly set by
  - `cmp S2, S1` set condition codes based on `S1-S2`
  - `test S2, S1` set condition codes based on `S1 & S2`



Recap: Condition Codes/Jumps

- setX: reading condition codes
  - set single byte, does not alter remaining 3 bytes
- jX: jump to different part of code depending on condition codes
- cmovX: conditional move

setX dest	jX target	cmovX S, reg	Condition	Description
	jmp		1	
sete	je	cmove	ZF	Equal / Zero
setne	jne	cmovne	~ZF	Not Equal / Not Zero
sets	js	cmovs	SF	Negative
setns	jns	cmovns	~SF	Nonnegative
setg	jg	cmovg	~(SF^OF)&~ZF	Greater (Signed)
setge	jge	cmovge	~(SF^OF)	Greater or Equal (Signed)
setl	jl	cmovl	(SF^OF)	Less (Signed)
setle	jle	cmovle	(SF^OF)|ZF	Less or Equal (Signed)
seta	ja	cmova	~CF&~ZF	Above (unsigned)
setb	jb	cmovg	CF	Below (unsigned)

Recap: if-else

```
if (cond) {
  true statements;
} else {
  false statements;
}
```

```
if (!cond) goto Else;
  true statements;
  goto Exit;
Else:
  false statements;
Exit:
```

```
  evaluate condition
  jX    .else
  true statements
  jmp .exit
.else:
  false statements
.exit:
```


Control Transfer Structures

- **Do-While Loops**
- While-Loops
- For-Loops
- Switch Statements

Acknowledgement: slides based on the cs:app2e material

"Do-While" Loop Example

C Code

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

Goto Version

```
int pcount_do(unsigned x)
{
  int 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

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

- Registers:
  - %edx x
  - %ecx result

```
  movl  $0, %ecx    #   result = 0
.L2:    # loop:
  movl  %edx, %eax
  andl  $1, %eax    #   t = x & 1
  addl  %eax, %ecx  #   result += t
  shrl  %edx        #   x >>= 1
  jne   .L2         #   If !0, goto loop
```


General "Do-While" Translation

C Code

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

Goto Version

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

- Body: { Statement$_1$; Statement$_2$; … Statement$_n$; }
- Test returns integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true

Control Transfer Structures

- Do-While Loops
- **While-Loops**
- For-Loops
- Switch Statements

"While" Loop Example

C Code

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

Goto Version

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

- Is this code equivalent to the do-while version?
  - extra test *before* entering the loop

General "While" Translation

While version

```
while (Test)
  Body
```

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:
```


Control Transfer Structures

- Do-While Loops
- While-Loops
- **For-Loops**
- Switch Statements

"For" Loop Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- Is this code equivalent to other versions?

"For" Loop Form

General Form

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

```
for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned mask = 1 << i;
  result += (x & mask) != 0;
}
```


"For" Loop → While Loop

For Version

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

While Version

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


"For" Loop → … → Goto

For Version

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

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

While Version

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

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


"For" Loop Conversion Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- Initial test can be optimized away

Goto Version

```
int pcount_for_gt(unsigned x) {
  int i;
  int result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               !Test
    goto done;
loop:                             Body
  {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  i++;                            Update
  if (i < WSIZE)                  Test
    goto loop;
done:
  return result;
}
```


Control Transfer Structures

- Do-While Loops
- While-Loops
- For-Loops
- **Switch Statements**

```
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

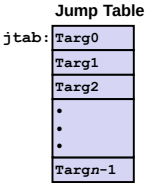
- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

Jump Table Structure

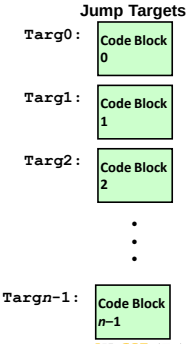
Switch Form

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

Jump Table



Jump Targets



Approximate Translation

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


Switch Statement Example (IA32)

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

What range of values takes default?

Setup:

```
switch_eg:
   pushl   %ebp          # Setup
   movl    %esp, %ebp    # Setup
   movl    8(%ebp), %eax # %eax = x
   cmpl    $6, %eax      # Compare x:6
   ja      .L2           # If unsigned > goto default
   jmp     *.L7(,%eax,4) # Goto *JTab[x]
```

Note that **w** is not initialized here

Switch Statement Example (IA32)

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

Jump table

```
.section    .rodata
  .align 4
.L7:
  .long     .L2 # x = 0
  .long     .L3 # x = 1
  .long     .L4 # x = 2
  .long     .L5 # x = 3
  .long     .L2 # x = 4
  .long     .L6 # x = 5
  .long     .L6 # x = 6
```

Setup:

```
switch_eg:
   pushl   %ebp          # Setup
   movl    %esp, %ebp    # Setup
   movl    8(%ebp), %eax # eax = x
   cmpl    $6, %eax      # Compare x:6
   ja      .L2           # If unsigned > goto default
   jmp     *.L7(,%eax,4) # Goto *JTab[x]
```

Indirect jump →

Assembly Setup Explanation

- Table Structure
  - Each target requires 4 bytes
  - Base address at `.L7`
- Jumping
  - **Direct:** `jmp .L2`
  - Jump target is denoted by label `.L2`
  - **Indirect:** `jmp *.L7(,%eax,4)`
  - Start of jump table: `.L7`
  - Must scale by factor of 4 (labels have 32-bits = 4 Bytes on IA32)
  - Fetch target from effective Address `.L7 + eax*4`
    - Only for $0 \le x \le 6$

Jump table

```
.section    .rodata
  .align 4
.L7:
  .long     .L2 # x = 0
  .long     .L3 # x = 1
  .long     .L4 # x = 2
  .long     .L5 # x = 3
  .long     .L2 # x = 4
  .long     .L6 # x = 5
  .long     .L6 # x = 6
```

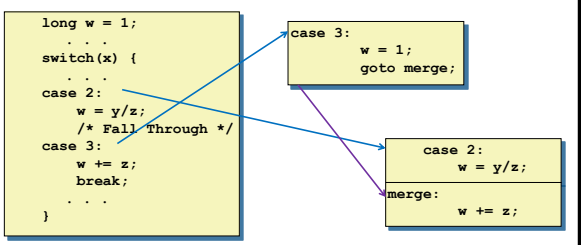

Jump Table

Jump table

```
.section    .rodata
  .align 4
.L7:
  .long     .L2 # x = 0
  .long     .L3 # x = 1
  .long     .L4 # x = 2
  .long     .L5 # x = 3
  .long     .L2 # x = 4
  .long     .L6 # x = 5
  .long     .L6 # x = 6
```

```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
case 2:      // .L4
    w = y/z;
    /* Fall Through */
case 3:      // .L5
    w += z;
    break;
case 5:
case 6:      // .L6
    w -= z;
    break;
default:     // .L2
    w = 2;
}
```

Handling Fall-Through



Code Blocks (Partial)

```
switch(x) {
case 1:       // .L3
      w = y*z;
      break;
 . . .
case 3:       // .L5
    w += z;
    break;
  . . .
default:      // .L2
    w = 2;
}
```

```
.L2:    # Default
  movl  $2, %eax  # w = 2
  jmp   .L8       # Goto done

.L5:    # x == 3
  movl  $1, %eax  # w = 1
  jmp   .L9       # Goto merge

.L3:    # x == 1
  movl  16(%ebp), %eax  # z
  imull 12(%ebp), %eax  # w = y*z
  jmp   .L8       # Goto done
```


Code Blocks (Rest)

```
switch(x) {
  . . .
  case 2:  // .L4
      w = y/z;
      /* Fall Through */
  merge:    // .L9
      w += z;
      break;
  case 5:
  case 6:  // .L6
      w -= z;
      break;
}
```

```
.L4:    # x == 2
  movl  12(%ebp), %edx
  movl  %edx, %eax
  sarl  $31, %edx
  idivl 16(%ebp)  # w = y/z

.L9:    # merge:
  addl  16(%ebp), %eax # w += z
  jmp   .L8       # goto done

.L6:    # x == 5, 6
  movl  $1, %eax       # w = 1
  subl  16(%ebp), %eax # w = 1-z
```


Switch Code (Finish)

```
return w;
```

```
.L8:    # done:
  popl  %ebp
  ret
```

- Noteworthy Features
  - Jump table avoids sequencing through cases
    - Constant time, rather than linear
  - Use jump table to handle holes and duplicate tags
  - Use program sequencing to handle fall-through
  - Don't initialize w = 1 unless really need it

x86-64 Switch Implementation

- Same general idea, adapted to 64-bit code
- Table entries 64 bits (pointers)
- Cases use revised code

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

```
.L3:
  movq    %rdx, %rax
  imulq   %rsi, %rax
  ret
```

Jump Table

```
  .section .rodata
  .align 8
.L7:
  .quad    .L2    # x = 0
  .quad    .L3    # x = 1
  .quad    .L4    # x = 2
  .quad    .L5    # x = 3
  .quad    .L2    # x = 4
  .quad    .L6    # X = 5
  .quad    .L6    # x = 6
```


IA32 Object Code

- Setup
  - Label `.L2` becomes address `0x8048422`
  - Label `.L7` becomes address `0x8048660`

Assembly Code

```
switch_eg:
  . . .
  ja     .L2           # If unsigned > goto default
  jmp    *.L7(,%eax,4) # Goto *JTab[x]
```

Disassembled Object Code

```
08048410 <switch_eg>:
 . . .
 8048419: 77 07         ja     8048422 <switch_eg+0x12>
 804841b: ff 24 85 60 86 04 08  jmp    *0x8048660(,%eax,4)
```

IA32 Object Code (cont.)

- Jump Table
  - Doesn't show up in disassembled code
  - Can inspect using GDB
  - **gdb switch**
  - **(gdb) x/7xw 0x8048660**
    - Examine 7 hexadecimal format "words" (4-bytes each)
    - Use command "**help x**" to get format documentation

```
0x8048660:    0x08048422    0x08048432    0x0804843b    0x08048429
0x8048670:    0x08048422    0x0804844b    0x0804844b
```


IA32 Object Code (cont.)

- Deciphering Jump Table

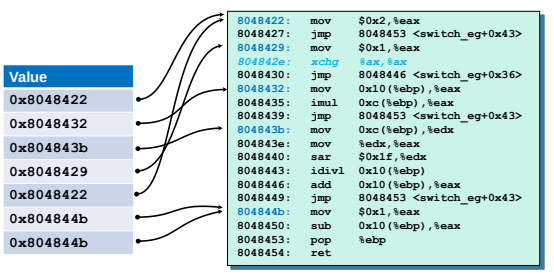
```
0x8048660:    0x08048422    0x08048432    0x0804843b    0x08048429
0x8048670:    0x08048422    0x0804844b    0x0804844b
```

Address	Value	x
0x8048660	0x8048422	0
0x8048664	0x8048432	1
0x8048668	0x804843b	2
0x804866c	0x8048429	3
0x8048670	0x8048422	4
0x8048674	0x804844b	5
0x8048678	0x804844b	6

Disassembled Targets

```
8048422:   b8 02 00 00 00       mov    $0x2,%eax
8048427:   eb 2a                jmp    8048453 <switch_eg+0x43>
8048429:   b8 01 00 00 00       mov    $0x1,%eax
804842e:   66 90                xchg   %ax,%ax # noop
8048430:   eb 14                jmp    8048446 <switch_eg+0x36>
8048432:   8b 45 10             mov    0x10(%ebp),%eax
8048435:   0f af 45 0c          imul   0xc(%ebp),%eax
8048439:   eb 18                jmp    8048453 <switch_eg+0x43>
804843b:   8b 55 0c             mov    0xc(%ebp),%edx
804843e:   89 d0                mov    %edx,%eax
8048440:   c1 fa 1f             sar    $0x1f,%edx
8048443:   f7 7d 10             idivl  0x10(%ebp)
8048446:   03 45 10             add    0x10(%ebp),%eax
8048449:   eb 08                jmp    8048453 <switch_eg+0x43>
804844b:   b8 01 00 00 00       mov    $0x1,%eax
8048450:   2b 45 10             sub    0x10(%ebp),%eax
8048453:   5d                   pop    %ebp
8048454:   c3                   ret
```


Matching Disassembled Targets



Control Transfer Structures: Summary

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