Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

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)

Registers

<table>
<thead>
<tr>
<th>TRAX</th>
<th>HR8</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRB</td>
<td>HR9</td>
</tr>
<tr>
<td>TRC</td>
<td>HR10</td>
</tr>
<tr>
<td>TRD</td>
<td>HR11</td>
</tr>
<tr>
<td>TRSI</td>
<td>HR12</td>
</tr>
<tr>
<td>TRDI</td>
<td>HR13</td>
</tr>
<tr>
<td>TRAP</td>
<td>HR14</td>
</tr>
<tr>
<td>TRBP</td>
<td>HR15</td>
</tr>
</tbody>
</table>

Current stack top: %rip

Instruction pointer

Condition codes: 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 operations
  - Example: addq Src, Dest \(\leftrightarrow t = a+b\)
  - CF set if carry out from most significant bit (unsigned overflow)
  - ZF set if \(t = 0\)
  - SF set if \(t < 0\) (as signed)
  - OF set if two's-complement (signed) overflow
    \(a>0 \land b<0 \land (a-b)<0\) or \(a<0 \land b>0 \land (a-b)>0\)

- Not set by leaq instruction

Condition Codes (Explicit Setting: Compare)

- Explicit Setting by Compare Instruction
  - cmpq Src2, Src1
  - cmpq b, a like computing a-b without setting destination

- CF set if carry out from most significant bit (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
  \(a>0 \land b<0 \land (a-b)<0\) or \(a<0 \land b>0 \land (a-b)>0\)

Condition Codes (Explicit Setting: Test)

- Explicit Setting by Test instruction
  - testq Src2, Src1
    - testq b, a like computing 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 when \(a*b = 0\)
- SF set when \(a*b < 0\)
Reading Condition Codes

- **SetX Instructions**
  - Set low-order byte to 0 or 1 based on condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>setls</td>
<td>SF</td>
<td>Negative (“Sign”)</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>seta</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned &gt;)</td>
</tr>
<tr>
<td>setae</td>
<td>~CF</td>
<td>Above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned &lt;)</td>
</tr>
<tr>
<td>setbe</td>
<td>CF</td>
<td>ZF</td>
</tr>
</tbody>
</table>

x86-64 Integer Registers

<table>
<thead>
<tr>
<th></th>
<th>al</th>
<th>bl</th>
<th>cl</th>
<th>dl</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rbx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rcx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rdx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rsi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r8b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r9b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r10b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r11b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r12b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r13b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r14b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r15b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Can reference low-order byte

Exercise Break: More Conditions

- Every condition can be negated by putting “n” in the mnemonic, for “not”
- We skipped some of these conditions in the previous table, because they were equivalent to others
- Which other conditions are these equivalent to?
  1. setng: not greater than
  2. setnbe: not below or equal

Equivalents of More Conditions

- Intuition: cover three cases: <, =, >
- **setng** not greater than (signed)
  - If not greater, than either less than or equal: **setle**
  - Check conditions:
    - ¬(SF ^ OF) & ¬ZF = ¬SF ^ OF | ¬ZF = (SF ^ OF | ZF
- **setnbe** not below or equal (unsigned)
  - If not below or equal, must be above: **seta**
  - Check conditions:
    - ¬(CF | ZF) = ¬CF & ¬ZF

Logistics announcements

- Data lab is due tonight by 11:55pm, on the Moodle
  - Double-check running driver.pl on a CSE Labs machine
- Assignment I is due Wednesday at the start of class
  - Don’t be late!
  - Write your recitation section time on your paper
- Bomb lab coming soon
  - Bomb lab will only run on CSE Labs machines
  - Due dates for future labs have been adjusted
Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

Jumping

- JX instructions
  - Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>JX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>JNE</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>JG</td>
<td>~(SF^OF)&amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>JGE</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>JL</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>JLE</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>JA</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>JB</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

Conditional Branch Example (Old Style)

- Generation

```
gen -Og -S -fno-if-conversion control.c
```

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

- Expressing with Goto Code

```
long abdiff_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;
}
```

General 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;
go_to Done;
Else:
  val = Else_Expr;
  goto Done;
```

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

Using Conditional Moves

- 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 = Ret
  ? Then_Expr
  : Else_Expr;
result = Then_Expr;
  eval = Else_Expr;
  not = !Test;
  if (nt) result = eval;
  return result;
```

**Goto Version**

```
result = Then_Expr;
  eval = Else_Expr;
  not = !Test;
  if (nt) result = eval;
  return result;
```
Conditional Move Example

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

Bad Cases for Conditional Move

**Expensive Computations**

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

**Risk Computations**

- Both values get computed
- May have undesirable effects
- Computations with side effects

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

“Do-While” Loop Example

**C Code**

```c
do { 
    Statement1;
    Statement2;
    ... 
} while (Test);
```

**Goto Version**

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

“Do-While” Loop Compilation

**Goto Version**

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

General “Do-While” Translation

**C Code**

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

**Goto Version**

```c
long pcount_goto (unsigned long x) {
    long result = 0;
    loop: 
        Body
        if (Test) 
           goto loop
    ```
### General “While” Translation #1

- “Jump-to-middle” translation
- Used with –Og

#### Goto Version

```c
while (Test)
  Body
```

#### Jump to Middle

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

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

### General “While” Translation #2

- “Do-while” conversion
- Used with –O1

#### Do-While Version

```c
if (!Test) goto done;
```

#### Goto Version

```c
if (!Test) goto done;
```

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

### “For” Loop Form

#### General Form

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

#### For Version

```c
define WSIZE *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;
}
```

#### “For” Loop → While Loop

- Initial
- While Version

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

```c
init:
while (Test) {
  Body
  update;
}
```
**For-While Conversion**

```c
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**

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

### Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

### Announcement Break: Bomb Lab Now Out

- Analyze malicious software with a debugger
  - Reverse engineering based on instructions, observation, and experiment
  - Find inputs to “defuse” a bomb program so it does not “explode”
- We’ve covered enough material for you to start working now
  - E.g., control flow structure and arithmetic
  - Will also cover in discussion sections tomorrow
- Like data lab, difficulty increases between parts
  - Last phase especially complex
  - Start early!

### Switch Statement Example

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

**Jump Table Structure**

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

- `jtab: Targ0 Targ1 Targ2 Targn-1`

- `Jump Targets:
  - Targ0: Code Block 0
  - Targ1: Code Block 1
  - Targ2: Code Block 2
  - Targn-1: Code Block n-1`
Switch Statement Example

```c
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  
j a .L8  
j mp +.14, (%rdi,8)
```

What range of values go to .L8?

Note that w not initialized here

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 ≤ x ≤ 6

Jump Table

```
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;
    }
```

Handling Fall-Through

```
long w = 1;
switch(x) {
    case 2:      // .L5
        w = y/z;
        goto merge;
    case 3:      // .L9
        w = z;
        break;
    case 5:      // .L7
        w = z;
        break;
    ...
    merge: w = 1;
    ...
    case 2:      // .L5
        w = y/z;
        goto merge;
    case 3:      // .L9
        w = z;
        break;
    case 5:      // .L7
        w = z;
        break;
    ...
    merge: w = 1;
    ...
```

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)  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Exercise Break: switch Bounds

- Every jump table needs to check that the index is in bounds
  - For each of these code patterns, what indexes are allowed?

```
cmpq $5, %rax
ja .Ldefault
jmp * .L1(%rax,8)

andq $7, %rax
jmp * .L2(%rax,8)

movzbl 8(%rbp), %eax
jmp * .L3(%rax,8)
```

https://chimein.cla.umn.edu/course/view/2021

Summary

- **Today**
  - Control: Condition codes
  - Conditional branches & conditional moves
  - Loops
  - Switch statements

- **Next Time**
  - Stack
  - Call / return
  - Procedure call discipline