Controlling Program Execution

- We can now generate programs that execute linear sequences of instructions
  - Access registers and storage
  - Perform computations
- But what about loops, conditions, etc.?
- Need ISA support for:
  - comparing and testing data values
  - directing program control
    - jump to some instruction that isn’t just the next one in sequence
    - Do so based on some condition that has been tested.

Processor State (x86-64, Partial)

Information about currently executing program.

<table>
<thead>
<tr>
<th>Registers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%r8</td>
</tr>
<tr>
<td>%rbx</td>
<td>%r9</td>
</tr>
<tr>
<td>%rcx</td>
<td>%r10</td>
</tr>
<tr>
<td>%rdx</td>
<td>%r11</td>
</tr>
<tr>
<td>%rsi</td>
<td>%r12</td>
</tr>
<tr>
<td>%rdi</td>
<td>%r13</td>
</tr>
<tr>
<td>%rsp</td>
<td>%r14</td>
</tr>
<tr>
<td>%rbp</td>
<td>%r15</td>
</tr>
</tbody>
</table>

%rip Instruction pointer

PC-relative Addressing

In general, you shouldn’t use %rip as a general purpose register.

However, the compiler may generate **PC-relative addressing**.

```
jmp 0x10(%rip)
```

The effective address for a PC-relative instruction address is the offset parameter added to the address of the next instruction. This offset is signed to allow reference to code both before and after the instruction.

Can you guess why the compiler might generate such code?
**Condition Codes (Implicit Setting)**

**Single bit registers**
- CF: carry flag (for unsigned)
- ZF: zero flag
- SF: sign flag (for signed)
- OF: overflow flag (for signed)

**Implicitly set by arithmetic operations**
E.g., addq Src, Dest

C analog: \( t = a + b \);
- 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 overflow:
  \( (a > 0 && b > 0 && (a - b) < 0) || (a < 0 && b < 0 && (a - b) >= 0) \)

Condition codes not set by lea instruction.

**Setting Condition Codes (Explicit Setting)**

**Explicitly set by Compare instruction**

\( \text{cmpq } S\text{rc2, Src1} \)
- \( \text{cmpq } b, a \) is like computing \( a - b \) without setting destination.
- CF set if carry out from most significant bit; used for unsigned computations.
- ZF set if \( a == b \)
- SF set if \( (a-b) < 0 \)
- OF set if two's complement (signed) overflow:
  \( (a>0 && b>0 && (a-b)<0) || (a<0 && b<0 && (a-b)>=0) \)

**Setting Condition Codes: Test Instruction**

**Explicitly set by Test instruction**

\( \text{testq } S\text{rc2, Src1} \)
- Sets condition codes based on value of \( \text{Src1} \& \text{Src2} \).
- Often useful to have one of the operands be a mask.
- \( \text{testq } b, a \) is like computing \( a\&b \), without setting a destination.
- ZF set iff \( a \& b == 0 \)
- SF set iff \( a \& b < 0 \)
- CF and OF are set to 0.

**SetX Instructions**

Set low order bytes of destination to 0 or 1, based on combinations of condition codes.

Does not alter remaining 7 bytes.

<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>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF&amp;OF)&amp;ZF</td>
<td>Greater (signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF&amp;OF)</td>
<td>Greater or equal (signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF&amp;OF)</td>
<td>Less (signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF&amp;OF) &amp; ZF</td>
<td>Less or equal (signed)</td>
</tr>
<tr>
<td>seta</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
### x86-64 Registers: Least Significant Byte

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td>return value</td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td>Argument x</td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>Argument y</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td></td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td></td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td></td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td></td>
</tr>
<tr>
<td><code>%rbp</code></td>
<td></td>
</tr>
</tbody>
</table>

Can reference the low-order byte.

### Reading Condition Codes

**SetX instructions**
- Set single byte based on combinations of conditions codes.

**Argument is one of addressable byte registers.**
- does not alter remaining bytes;
- typically use `movzbl` to finish the job (will also zero 4 high order bytes).

```plaintext
cmpq %rsi, %rdi # compare x:y
setg %al # Set if >
movzbl %al, %eax # Zero rest of %rax
retq
```

### Jumping

**jX Instructions:** Jump to different parts of the code depending on condition codes.

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<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>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>(SF&lt;OF) &amp; ~ZF</td>
<td>Greater (signed)</td>
</tr>
<tr>
<td>jge</td>
<td>(SF&lt;OF)</td>
<td>Greater or equal (signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF&lt;OF)</td>
<td>Less (signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF&lt;OF)</td>
<td>Less or equal (signed)</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: `gcc -Og -fno-if-conversion control.c`

```plaintext
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)
{
    cmpq %rsi, %rdi # x:y
    jle .L4
    movq %rdi, %rax
    subq %rsi, %rax
    retq
    .L4: # x <= y
    movq %rsi, %rax
    subq %rdi, %rax
    retq
}
```

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>Argument x</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>Argument y</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>return value</td>
</tr>
</tbody>
</table>
Converting C into Assembly

Machine Models

Data
1) char
2) int, float
3) double
4) struct, array
5) pointer

Control
1) loops
2) conditionals
3) switch
4) proc. call
5) proc. return

Memory Processor

C

ALU
Regs
Stack
Mixery
Processor

A common compilation strategy is to take a C construct and rewrite it into an equivalent C version that is closer to assembly, as an intermediate step toward assembly.

Expressing with Goto Code

C allows “goto” as a means of transferring control.

- Jump to position designated by label.
- Generally considered bad coding style in high level language.

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

C Code:

```
val = Test ? Then.Expr : Else.Expr;
```

Example:

```
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: ...
```

- Refer to generically as “cmovXX”
- Based on values of condition codes
- Conditionally copy value from source to destination.
- Can be used to eliminate conditional jump.

Conditional Move Instructions

General Condition Expression Translation

Create separate code regions for then and else expressions.

Execute the appropriate one.
### Conditional Move Instructions

<table>
<thead>
<tr>
<th>Inst.</th>
<th>Synonym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmove</td>
<td>cmovz</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>cmovne</td>
<td>cmovnz</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>cmovs</td>
<td>cmovnle</td>
<td>Greater (signed)</td>
</tr>
<tr>
<td>cmovns</td>
<td>cmovnl</td>
<td>Greater or equal (signed)</td>
</tr>
<tr>
<td>cmovg</td>
<td>cmovnge</td>
<td>Less (signed)</td>
</tr>
<tr>
<td>cmovge</td>
<td>cmovnle</td>
<td>Less or equal (signed)</td>
</tr>
<tr>
<td>cmovl</td>
<td>cmovnb</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>cmovle</td>
<td>cmovnbe</td>
<td>Above or equal (unsigned)</td>
</tr>
<tr>
<td>cmova</td>
<td>cmovnbe</td>
<td>Below (unsigned)</td>
</tr>
<tr>
<td>cmovae</td>
<td>cmovnae</td>
<td>Below or equal (unsigned)</td>
</tr>
</tbody>
</table>

### 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 safe

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

#### C Code
```c
val = Test ? Then_Expr : Else_Expr
```

#### Goto Version
```c
result = Then_Expr;
val = Else_Expr;
nt = !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;
}
```

#### Expensive Computations:
```c
val = Test(x) ? Hard1(x) : Hard2(x);
```
- Both values get computed
- Only makes sense when computations are very simple

#### Risky Computations:
```c
val = p ? *p : 0;
```
- Both values get computed
- May have undesirable side effects.

#### Computations with Side Effects:
```c
val = x > 0 ? x *= 7 : x += 3;
```
- Both values get computed
- Must be side effect free
Do-While Loop Example

Following our strategy of rewriting a C construct into a semantically equivalent C version that is closer to assembly.

C Code:

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

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

Goto Version:

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

General Do-While Translation

C Code:

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

- Body can be any C statement, typically is a compound statement.
- Test is an expression returning an integer.
  - If it evaluates to 0, that's interpreted as false.
  - If it evaluates to anything but 0, that's interpreted as true.

Goto Version:

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

General While Translation #1

C Code:

```c
while (Test);
```

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

Goto version

```c
goto test;
loop:
    Body
test:
    if (Test) goto loop;
done:
```
While Loop Example #1

C Code

```c
long pcount_while (unsigned long s) {
    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

Jump to Middle

```c
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;
}
```

While Loop Example #2

C Code

```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 conditional guards entrance to loop

Do-While version

```c
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;
}
```

General While Translation

- Which gets compiled as if it were:
  - Goto Version
    ```c
    if (!Test)
        goto done;
    loop:
        if (Test)
            goto loop;
    done:
    return result;
    ```
  - Are all three versions semantically equivalent?

For Loop Form

```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;
}
```

- Init
  ```c
  i = 0
  ```
- Test
  ```c
  i < WSIZE
  ```
- Update
  ```c
  i++
  ```
- Body
  ```c
  { unsigned bit = (x >> i) & 0x1;
    result += bit;
  }
```
For Loop to While Loop

**For version**

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

**Translates to:**

**While version**

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

For Loop Do-While Conversion

**C Code: Goto version**

```c
long pcount_for_goto (unsigned long x) {
    size_t i;
    long result = 0;
    i = 0;
    if (!(i < WSIZE)) # drop
go to done; # drop loop:
    { unsigned bit =
        (x >> i) & 0x1;
        result += bit;
    }
    i++;
    if (i < WSIZE)
go to loop;
done:
    return result;
}
```

Note that the initial test is not needed. Why?

**Switch Statement Example**

```c
long switch_eq (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:
        break;
    default:
        w = 2;
    }
    return w;
}
```

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

For-While Conversion Example

**Init**

```c
i = 0
```

**Test**

```c
i < WSIZE
```

**Update**

```c
i++
```

**Body**

```c

unsigned bit =
    (x >> i) & 0x1;
result += bit;
```
### Jump Table Structure

#### Switch Form

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

#### Translation (Extended C)

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

#### Jump Table

<table>
<thead>
<tr>
<th>JTab:</th>
<th>Targ0</th>
<th>Targ1</th>
<th>Targ2</th>
<th>...</th>
<th>Targn-1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Code</td>
<td>Code</td>
<td>Code</td>
<td></td>
<td>Code</td>
</tr>
<tr>
<td></td>
<td>Block</td>
<td>Block</td>
<td>Block</td>
<td></td>
<td>Block</td>
</tr>
</tbody>
</table>

#### Jump Targets

- **Targ0:** Code Block 0
- **Targ1:** Code Block 1
- **Targ2:** Code Block 2
- **...**
- **Targn-1:** Code Block n-1

### Switch Example

#### Long Switch Statement Example

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

#### Setup:

```asm
switch_eq:
  movq  %rdx, %rcx
  cmpq  $6, %rdi      # x:6
  ja    .L8
  jmp   *.L4(, %rdi, 8)
```

Note that `w` is not initialized here.

### Assembly Setup Explanation

- 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`), but only for `0 ≤ x ≤ 6`
Jump Table:

```
.long switch_eq
(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;
    default:
        w = 2;
    }
    return w;
}
```

Handling Fall-Through

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

Code Blocks (x == 1)

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

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

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

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

```
.long w = 1;
...
.switch (x) {
    ...
    case 2:
        w = y/z;
        goto merge;
    ...
    case 3:
        w += z;
        merge:
        w += z;
        ...
}
```

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

```
Register Use(s)
%rdi Argument x
%rsi Argument y
%rdx Argument z
%rax return value
```
Suppose you have a set of switch labels that are “sparse” (widely separated).

In this case, it doesn’t make sense to use a jump table.
- If there are only a few labels, simply use a nested if structure.
- If there are many, build a balanced binary search tree.

The compiler decides the appropriate thresholds for what’s “sparse,” what are “a few,” etc.

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

---

### Summarizing

**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