We can now generate programs that execute linear sequences of instructions: access registers and memory, 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.

**Registers**

<table>
<thead>
<tr>
<th>%rax</th>
<th>%r8</th>
</tr>
</thead>
<tbody>
<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

**Condition codes**

| CF | ZF | SF | OF |

Don’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**
cmpq Src2, 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**
testq Src2, 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.

How could you use testq to jump if the value in \%rbx is even?

```assembly
  testq $1, %rbx
  je even
  odd:
```
Reading Condition Codes

**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>sets</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (signed)</td>
</tr>
<tr>
<td>setge</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>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>%rax</th>
<th>%al</th>
<th>%r8</th>
<th>%r8b</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%bl</td>
<td>%r9</td>
<td>%r9b</td>
</tr>
<tr>
<td>%rcx</td>
<td>%cl</td>
<td>%r10</td>
<td>%r10b</td>
</tr>
<tr>
<td>%rdx</td>
<td>%dl</td>
<td>%r11</td>
<td>%r11b</td>
</tr>
<tr>
<td>%rsi</td>
<td>%eil</td>
<td>%r12</td>
<td>%r12b</td>
</tr>
<tr>
<td>%rdi</td>
<td>%di</td>
<td>%r13</td>
<td>%r13b</td>
</tr>
<tr>
<td>%rsp</td>
<td>%spl</td>
<td>%r14</td>
<td>%r14b</td>
</tr>
<tr>
<td>%rbp</td>
<td>%bpl</td>
<td>%r15</td>
<td>%r15b</td>
</tr>
</tbody>
</table>

Can reference the low-order byte.

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^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>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

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

**Register Use(s)**

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

**Absdiff Assembly**

<table>
<thead>
<tr>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpq %rsi, %rdi # x:y</td>
</tr>
<tr>
<td>jle .L4</td>
</tr>
<tr>
<td>movq %rdi, %rax</td>
</tr>
<tr>
<td>subq %rsi, %rax</td>
</tr>
<tr>
<td>retq .L4: # x &lt;= y</td>
</tr>
<tr>
<td>movq %rsi, %rax</td>
</tr>
<tr>
<td>subq %rdi, %rax</td>
</tr>
<tr>
<td>retq</td>
</tr>
</tbody>
</table>

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.

```c
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;
}
```

C Code:

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

Example:

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

Goto Version:

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

- Create separate code regions for then and else expressions.
- Execute the appropriate one.
Conditional Move Instructions

- 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.

<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>cmovng</td>
<td>Negative</td>
</tr>
<tr>
<td>cmovns</td>
<td>cmovnge</td>
<td>Not negative</td>
</tr>
<tr>
<td>cmovg</td>
<td>cmovnle</td>
<td>Greater (signed)</td>
</tr>
<tr>
<td>cmovge</td>
<td>cmovnl</td>
<td>Greater or equal (signed)</td>
</tr>
<tr>
<td>cmovl</td>
<td>cmovnge</td>
<td>Less (signed)</td>
</tr>
<tr>
<td>cmovle</td>
<td>cmovng</td>
<td>Less or equal (signed)</td>
</tr>
<tr>
<td>cmova</td>
<td>cmovnbe</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>cmovae</td>
<td>cmovnb</td>
<td>Above or equal (unsigned)</td>
</tr>
<tr>
<td>cmovb</td>
<td>cmovnae</td>
<td>Below (unsigned)</td>
</tr>
<tr>
<td>cmovbe</td>
<td>cmovna</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
long absdiff (long x, long y)
{
    long result;
    if (x > y)
        result = x - y;
    else
        result = y - x;
    return result;
}
```

Goto Version

```c
val = Test
    ? Then_Mexpr
        : Else_Mexpr

result = Then_Mexpr;
val = Else_Mexpr;
nt = !Test;
if (nt) result = eval;
return result;
```

Conditional Move Example

```asm
abdiff:
    movq %rdi, %rax  # x
    subq %rsi, %rax  # result = x - y
    movq %rsi, %rdx  # y
    subq %rdi, %rdx  # eval = y - x
    cmpq %rsi, %rdi  # x : y
    cmovle %rdx, %rax  # if <=, result = eval
    retq
```

Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>
Bad Cases for Conditional Move

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;
}
```

Goto Version:

```c
long pcount goto
(unsigned long x) {
    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:

```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);
```

Goto Version:

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

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

### General While Translation #1

- **While version**
  ```c
goto test;
loop:
  Body
test:
    if (Test)
      goto loop;
done:
```

- **Goto version**
  ```c
  while (Test)
  Body
test:
  if (Test)
    goto loop;
done:
```

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

- **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;
  }
  ```

- **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

- **While Loop Example #1**
  ```c
  long pc_count
  (unsigned long x)
  {
    long result = 0;
    while (x) {
      result += x & 0x1;
      x >>= 1;
    }
    return result;
  }
  ```

- **C Code**
  ```c
  long pc_count
  (unsigned long x) {
  long result = 0;
  goto test;
  loop:
    result += x & 0x1;
    x >>= 1;
  test:
    if (x) goto loop;
  return result;
  }
  ```

- **Jump to Middle**
  ```c
  long pc_count
  (unsigned long x) {
  long result = 0;
  goto test;
  loop:
    result += x & 0x1;
    x >>= 1;
  test:
    if (x) goto loop;
  return result;
  }
  ```

- **Do-While version**
  ```c
  long pc_count
  (unsigned long x) {
  long result = 0;
  if (!x) goto done;
  loop:
    result += x & 0x1;
    x >>= 1;
  if (x) goto loop;
  done:
  return result;
  }
  ```

### While Loop Example #2

- **C Code**
  ```c
  long pc_count
  (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
  }
  ```

- **Goto Version**
  ```c
  if (!Test)
    goto done;
  loop:
    Body
  if (Test)
    goto loop;
done:
  ```

- **Do-While version**
  ```c
  long pc_count
  (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 goto starts loop at test**

Are all three versions semantically equivalent?
For Loop Form

**General Form**

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

**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;
}
```

---

For Loop to While Loop

**For version**

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

translates to:

**While version**

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

---

For-While Conversion Example

**Init**

```
i = 0
```

**Test**

```
i < WSIZE
```

**Update**

```
i++
```

**Body**

```
    { unsigned bit =
      (x >> i) & 0x1;
      result += bit;
      i++;
    }
    return result;
```

---

For Loop Do-While Conversion

**C Code: Goto version**

```
long pcount_for_goto_dw (unsigned long x) {
    size_t i;
    long result = 0;
    i = 0;
    if (!$i < WSIZE) # drop
goto done; # drop
loop:
{ unsigned bit =
  (x >> i) & 0x1;
  result += bit;
  i++;
} if ($i < WSIZE)
goto 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:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

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

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
}
```

**Jump Table**

- **JTab**: Targ0, Targ1, Targ2, ..., Targn-1

**Translation (Extended C)**

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

Jump Targets

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

Switch Example

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

Setup:

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

**Register Use(s)**

- %rdi: Argument x
- %rsi: Argument y
- %rdx: Argument z
- %rax: return value

Note that w is not initialized here.

Jump table

```c
cmpl %rdx, %rcx
    cmpl $6, %rdi # x : 6
    ja .L8 # use default
    jmp *.L4(, %rdi, 8) # goto *JTAB[x], # indirect jump
```

CS429 Slideset 8: 33 Instruction Set Architecture III

CS429 Slideset 8: 34 Instruction Set Architecture III

CS429 Slideset 8: 35 Instruction Set Architecture III

CS429 Slideset 8: 36 Instruction Set Architecture III
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), but 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
```

Code Blocks ($x == 1$)

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

Handling Fall-Through

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

Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</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>
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.