## What Is A Register (again)?

- A location in the CPU that stores a small amount of data, which can be accessed very quickly *(once every clock cycle)*.

- Registers have names, not addresses.

- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but especially x86.

## Integer Registers (IA32)

<table>
<thead>
<tr>
<th>Register</th>
<th>Origin (mostly obsolete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>accumulate</td>
</tr>
<tr>
<td>%ecx</td>
<td>counter</td>
</tr>
<tr>
<td>%edx</td>
<td>data</td>
</tr>
<tr>
<td>%ebx</td>
<td>base</td>
</tr>
<tr>
<td>%esi</td>
<td>source index</td>
</tr>
<tr>
<td>%edi</td>
<td>destination index</td>
</tr>
<tr>
<td>%esp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>%ebp</td>
<td>base pointer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Purpose</td>
<td>Accumulate</td>
</tr>
<tr>
<td></td>
<td>Counter</td>
</tr>
<tr>
<td></td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td>Base</td>
</tr>
<tr>
<td></td>
<td>Source Index</td>
</tr>
<tr>
<td></td>
<td>Destination Index</td>
</tr>
<tr>
<td></td>
<td>Stack Pointer</td>
</tr>
<tr>
<td></td>
<td>Base Pointer</td>
</tr>
</tbody>
</table>

### Origin (mostly obsolete)

- 32-bits wide
**Assembly Data Types**

- "Integer" data of 1, 2, 4 (IA32), or 8 (just in x86-64) bytes
  - Data values
  - Addresses (untyped pointers)

- Floating point data of 4, 8, or 10 bytes

- What about "aggregate" types such as arrays?
  - Just contiguous memory locations

**Three Basic Kinds of Instructions**

- Transfer data between memory and register
  - **Load** data from memory into register
    - %reg = Mem[address]
  - **Store** register data into memory
    - Mem[address] = %reg

  Remember: memory is indexed just like an array[] of bytes!

- Perform arithmetic function on register or memory data
  - c = a + b; z = x << y; i = h & g;

- Transfer control: what instruction to execute next
  - Unconditional jumps to/from procedures
  - Conditional branches

---

### Integer Registers (IA32)

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax, %ah, %al</td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx, %ch, %cl</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx, %dh, %dl</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx, %bh, %bl</td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
</tr>
</tbody>
</table>

- 16-bit virtual registers (backwards compatibility)

---

### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax, %edx</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx, %edx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx, %edx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi, %edi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
</tr>
</tbody>
</table>

- 64-bit wide
- Extend existing registers, and add 8 new ones; all accessible as 8, 16, 32, 64 bits.
Moving Data: IA32

- **Moving Data**
  - `movx Source, Dest`
  - *x* is one of {b, w, l}

- `movl Source, Dest`:
  - Move 4-byte “long word”

- `movw Source, Dest`:
  - Move 2-byte “word”

- `movb Source, Dest`:
  - Move 1-byte “byte”

- Lots of these in typical code

### confounding historical terms…
not the current machine word size

#### Operands Types

- **Immediate**: Constant integer data
  - Example: `$0x400, $-533`
  - Like C constant, but prefixed with `$`
  - Encoded with 1, 2, or 4 bytes

- **Register**: One of 8 integer registers
  - Example: `%eax, %edx`
  - But `%esp` and `%ebp` reserved for special uses
  - Others have special uses for particular instructions

- **Memory**: 4 consecutive bytes of memory at address given by register
  - Simplest example: (`%eax`)
  - Various other “address modes”

#### movl Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>var_a = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl $-147,%eax</td>
<td>*p_a = -147;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td>var_d = var_a;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
<td>var_d = *p_a;</td>
</tr>
</tbody>
</table>

*Cannot do memory-memory transfer with a single instruction.*

**How would you do it?**

Memory vs. registers

- **What is the main difference?**
- **Addresses vs. Names**
- **Big vs. Small**
Memory Addressing Modes: Basic

- **Indirect** \((R)\) \(\text{Mem}[\text{Reg}[R]]\)
  - Register \(R\) specifies the memory address
  
  \[
  \text{movl} \ (%\text{ecx}),%\text{eax}
  \]

- **Displacement** \(D(R)\) \(\text{Mem}[\text{Reg}[R]+D]\)
  - Register \(R\) specifies a memory address
    - (e.g. the start of some memory region)
  - Constant displacement \(D\) specifies the offset from that address
  
  \[
  \text{movl} \ 8(%\text{ebp}),%\text{edx}
  \]

Using Basic Addressing Modes

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```assembly
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```
Understanding Swap

```
Address
123  0x124
  456  0x120
  0x11c  0x118
  0x114  0x110
  0x10c  0x108
  0x104  0x100

Offset

%ebp  0
%edi  4
%esi  8
%edi  12
%esp  16
%ebp  20

movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax, (%edx)
movl %ebx, %ecx
movl %eax, (%edi)
movl %ebx, (%esi)
movl %eax, (%esi)
movl %ebx, (%edi)
movl %eax, (%edi)
movl %ebx, (%edi)
movl %eax, (%esi)
movl %ebx, (%esi)
movl %eax, (%edi)
movl %ebx, (%edi)
movl %eax, (%esi)
movl %ebx, (%esi)
movl %eax, (%edi)
movl %ebx, (%edi)

# ecx = yp
# edx = xp
# eax = *yp (t1)
# ebx = *xp (t0)
# *xp = eax
# *yp = ebx
```
### Understanding Swap

<table>
<thead>
<tr>
<th>&amp;easx 456</th>
<th>0x124</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;edx 0x124</td>
<td>0x120</td>
</tr>
<tr>
<td>&amp;ecx 0x120</td>
<td>0x11c</td>
</tr>
<tr>
<td>&amp;ebx 123</td>
<td>0x118</td>
</tr>
<tr>
<td>&amp;esi</td>
<td>0x114</td>
</tr>
<tr>
<td>&amp;edi</td>
<td>0x110</td>
</tr>
<tr>
<td>&amp;esp</td>
<td>0x10c</td>
</tr>
<tr>
<td>&amp;ebp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

#### Address Table

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
<tr>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td>8</td>
<td>0x110</td>
</tr>
<tr>
<td>12</td>
<td>0x112</td>
</tr>
</tbody>
</table>

#### Code Snippet

```
movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax,(%edx)
movl %ebx,(%ecx)
```

### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>&amp;rax</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>&amp;rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>&amp;rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>&amp;rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>&amp;rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>&amp;rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>&amp;rbp</td>
<td>%ebp</td>
</tr>
</tbody>
</table>

- Extend existing registers, and add 8 new ones; all accessible as 8, 16, 32, 64 bits.

### 32-bit vs. 64-bit operands

- **Long word** $l$ (4 Bytes) $\leftrightarrow$ **Quad word** $q$ (8 Bytes)

- **New instruction forms:**
  - `movl` $\rightarrow$ `movq`
  - `addl` $\rightarrow$ `addq`
  - `sall` $\rightarrow$ `salq`
  - etc.

- x86-64 can still use 32-bit instructions that generate 32-bit results
  - Higher-order bits of destination register are just set to 0
  - Example: `addl`
Swap Ints in 32-bit Mode

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Swap Ints in 64-bit Mode

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Swap Long Ints in 64-bit Mode

```c
void swap_l (long int *xp, long int *yp)
{
    long int t0 = *xp;
    long int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Complete Memory Addressing Modes

- Remember, the addresses used for accessing memory in mov (and other) instructions can be computed in several different ways
- Most General Form:
  \[ D(Rb,Ri,S) \quad Mem[Reg[Rb] + S*Reg[Ri] + D] \]
  - \( D \): Constant "displacement" value represented in 1, 2, or 4 bytes
  - \( Rb \): Base register: Any of the 8/16 integer registers
  - \( Ri \): Index register: Any, except for %esp or %rsp; %ebp unlikely
  - \( S \): Scale: 1, 2, 4, or 8 (*why these numbers?*)
- Special Cases: can use any combination of \( D \), \( Rb \), \( Ri \) and \( S \)
  - \( (Rb,Ri) \quad Mem[Reg[Rb]+Reg[Ri]] \quad (S=1, D=0) \)
  - \( D(Rb,Ri) \quad Mem[Reg[Rb]+Reg[Ri]+D] \quad (S=1) \)
  - \( (Rb,Ri,S) \quad Mem[Reg[Rb]+S*Reg[Ri]] \quad (D=0) \)
### Address Computation Examples

<table>
<thead>
<tr>
<th>%edx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%edx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>0x80(%dx,%ecx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(%edx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

### Address Computation Instruction

- **leal Src, Dest**
  - `Src` is address mode expression
  - Set `Dest` to address computed by expression
    - (lea stands for *load effective address*)
  - Example: `leal (%edx,%ecx,4), %eax`

- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form `x + k*i`
    - `k = 1, 2, 4, or 8`

### Some Arithmetic Operations

- **Two Operand (Binary) Instructions:**
  - **Format**
    - `addl Src, Dest`  
    - `subl Src, Dest`
    - `imull Src, Dest`  
    - `shll Src, Dest`  
    - `xorl Src, Dest`  
    - `andl Src, Dest`  
    - `orl Src, Dest`
  - **Computation**
    - `Dest = Dest + Src`
    - `Dest = Dest - Src`
    - `Dest = Dest * Src`
    - `Dest = Dest << Src`
    - `Dest = Dest ^ Src`
    - `Dest = Dest & Src`
    - `Dest = Dest | Src`

- **One Operand (Unary) Instructions**
  - **incl Dest**  
  - **decl Dest**  
  - **negl Dest**  
  - **notl Dest**
  - **Computation**
    - `Dest = Dest + 1`
    - `Dest = Dest - 1`
    - `Dest = ~Dest`
    - `Dest = ~Dest`
  - **Arithmetic**
    - `Arithmetic`
  - **Logical**
    - `Logical`

- **Watch out for argument order! (especially subl)**
  - **No distinction between signed and unsigned int (why?)**
    - except arithmetic vs. logical shift right

- **See textbook section 3.5.5 for more instructions:** `mull, cld, idivl, divl`
Using `lea`l for Arithmetic Expressions (IA32)

```c
int arith(int x, int y, int z) {
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

Understanding `arith` (IA32)

```c
int arith(int x, int y, int z) {
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```
Understanding arith (IA32)

```c
int arith(int x, int y, int z)
{
    int t1 = x + y;
    int t2 = z + t1;
    int t3 = x + 4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

Observations about arith

```c
int arith(int x, int y, int z)
{
    int t1 = x + y;
    int t2 = z + t1;
    int t3 = x + 4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

Another Example (IA32)

```c
int logical(int x, int y)
{
    int t1 = x ^ y;
    int t2 = t1 >> 17;
    int mask = (1 << 13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

```c
movl 8(%ebp), %eax
# eax = x
xorl 12(%ebp), %eax
# eax = x ^ y (t1)
```

```c
movl %ebp, %esp
# eax = x
xorl 12(%ebp), %eax
# eax = x ^ y (t1)
```

```c
logical:
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax
xori 12(%ebp), %eax
sarl $17, %eax
andl $8185, %eax
movl %ebp, %esp
popl %ebp
ret
```

```c
logical:
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax
xori 12(%ebp), %eax
sarl $17, %eax
andl $8185, %eax
movl %ebp, %esp
popl %ebp
ret
```
Another Example (IA32)

```c
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

```
set up
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
movl %ebp,%esp
popl %ebp
ret
```

```
body
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
```

```
finish
movl %ebp,%esp
```

```
2^{13} = 8192,
2^{13} - 7 = 8185
...
```

topics: control flow

- Condition codes
- Conditional and unconditional branches
- Loops

Conditionals and Control Flow

- A conditional branch is sufficient to implement most control flow constructs offered in higher level languages
  - if (condition) then {...} else {...}
  - while (condition) {...}
  - do {...} while (condition)
  - for (initialization; condition; iterative) {...}

- Unconditional branches implement some related control flow constructs
  - break, continue

- In x86, we’ll refer to branches as “jumps” (either conditional or unconditional)
Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes
  - Takes address as argument

<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>

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** (think of it as side effect) 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** (beware!)

- **Full documentation (IA32):** [http://www.jegerlehner.ch/intel/IntelCodeTable.pdf](http://www.jegerlehner.ch/intel/IntelCodeTable.pdf)

Condition Codes (Explicit Setting: Compare)

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

- **Explicit Setting by Compare Instruction**
  - `cmpl cmpq Src2,Src1`
  - `cmp l 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 & & b<0 & & (a-b)<0) │ │ (a<0 & & b>0 & & (a-b)>0)

Processor State (IA32, Partial)

- **Information about currently executing program**
  - Temporary data (%eax, ...)
  - Location of runtime stack (%ebp, %esp)
  - Location of current code control point (%eip)
  - Status of recent tests (CF, ZF, SF, OF)

<table>
<thead>
<tr>
<th></th>
<th>General purpose registers</th>
<th>Current stack top</th>
<th>Current stack frame</th>
<th>Instruction pointer</th>
<th>Condition codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CF, ZF, SF, OF</td>
</tr>
<tr>
<td>%ecx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%eip</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Condition Codes (Explicit Setting: Test)

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

- Explicit Setting by Test instruction
  - `testl`/`testq` `Src2`, `Src1`
  - `testl` `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 if `a & b == 0`
  - SF set if `a & b < 0`

  - `testl` `%eax`, `%eax`
    - Sets SF and ZF, check if eax is +0,-

Reading Condition Codes

- SetX Instructions
  - Set a single byte to 0 or 1 based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sete</code></td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td><code>setne</code></td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td><code>sets</code></td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td><code>setns</code></td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td><code>setg</code></td>
<td>(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td><code>setge</code></td>
<td>(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td><code>setl</code></td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td><code>setle</code></td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td><code>seta</code></td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td><code>setb</code></td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

Reading Condition Codes (Cont.)

- SetX Instructions:
  - Set single byte to 0 or 1 based on combination of condition codes
- One of 8 addressable byte registers
  - Does not alter remaining 3 bytes
  - Typically use `movzbl` to finish job

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

Body: y at 12(%ebp), x at 8(%ebp)

```assembly
movl 12(%ebp), %eax
cmpl %eax, 8(%ebp)
setg %al
movzbl %al, %eax
```

What does each of these instructions do?

Body: y at 12(%ebp), x at 8(%ebp)

```assembly
movl 12(%ebp), %eax # eax = y
cmpl %eax, 8(%ebp) # Compare x and y
setg %al # al = x > y
movzbl %al, %eax # Zero rest of %eax
 Movem (x - y)```
Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes
  - Takes address as argument

<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

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

```assembly
absdiff:  
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L7
    subl %eax, %edx
    movl %edx, %eax
    .L8:
    leave
    ret
    .L7:
    subl %edx, %eax
    jmp .L8
```

#### Conditional Branch Example (Cont.)

```c
int goto_ad(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
    Exit:
    return result;
    Else:
    result = y-x;
    goto Exit;
}
```

```assembly
goto_ad:  
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L7
    subl %eax, %edx
    movl %edx, %eax
    .L8:
    leave
    ret
    .L7:
    subl %edx, %eax
    jmp .L8
```

- C allows "goto" as means of transferring control
  - Closer to machine-level programming style
  - Generally considered bad coding style
### Conditional Branch Example (Cont.)

```c
int goto_ad(int x, int y) {
    int result;
    if (x <= y) goto Else;
    result = x-y;
    Exit:
    return result;
Else:
    result = y-x;
    goto Exit;
}
```

```assembly
absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L7
    subl %eax, %edx
    movl %edx, %eax
.L8:
    leave
    ret
.L7:
    subl %edx, %eax
    jmp .L8
```

### Conditional Branch Example (Cont.)

```c
int goto_ad(int x, int y) {
    int result;
    if (x <= y) goto Else;
    result = x-y;
    Exit:
    return result;
Else:
    result = y-x;
    goto Exit;
}
```

```assembly
absdiff:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L7
    subl %eax, %edx
    movl %edx, %eax
.L8:
    leave
    ret
.L7:
    subl %edx, %eax
    jmp .L8
```
General Conditional Expression Translation

C Code

```c
if (Test)
    val = Then-Expr;
else
    val = Else-Expr;
result = x>y ? x-y : y-x;
```

Goto Version

```c
nt = !Test;
if (nt)
    goto Else;
val = Then-Expr;
Done:
    ...
Else:
    val = Else-Expr;
goto Done;
```

- Test is expression returning integer
  - 0 interpreted as false
  - ≠0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one
- How might you make this more efficient?

Conditionals: x86-64

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

- Conditional move instruction
  - cmovC src, dest
  - Move value from src to dest if condition C holds
  - Why is this good?

PC Relative Addressing

```
0x100     cmp  r2, r3     0x1000
0x102     je   0x70       0x1002
0x104     ...               0x1004
          ...               ...
0x172     add  r3, r4     0x1072
```

- PC relative branches are relocatable
- Absolute branches are not

```
absdiff: # x in %edi, y in %esi
movl  %edi, %eax # eax = x
movl  %esi, %edx # edx = y
subl  %esi, %eax # eax = x-y
subl  %edi, %edx # edx = y-x
cmpl  %esi, %edi # x:y
cmovle %edx, %eax # eax=edx if <=
ret
```

- Conditional move instruction
  - cmovC src, dest
  - Move value from src to dest if condition C holds
  - More efficient than conditional branching (simple control flow)
  - But overhead: both branches are evaluated
Compiling Loops

How to compile other loops should be straightforward
- The only slightly tricky part is to be sure where the conditional branch occurs: top or bottom of the loop

```
C/Java code:
while ( sum != 0 ) {
    <loop body>
}

Machine code:
loopTop:  cmpl $0, %eax
          je  loopDone
          <loop body code>
          jmp  loopTop
loopDone:
```

- Use backward branch to continue looping
- Only take branch when “while” condition holds

“Do-While” Loop Example

```
C Code
int fact_do(int x)
{
    int result = 1;
    do {
        result *= x;
        x = x-1;
    } while (x > 1);
    return result;
}

Goto Version
int fact_goto(int x)
{
    int result = 1;
    loop:
    result *= x;
    x = x-1;
    if (x > 1) goto loop;
    return result;
}
```

“Do-While” Loop Compilation

```
Goto Version
int fact_goto(int x)
{
    int result = 1;
    loop:
    result *= x;
    x = x-1;
    if (x > 1) goto loop;
    return result;
}

Assembly
fact_goto:
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax
    movl 8(%ebp),%edx

    .L11:
        imull %edx,%eax
        decl %edx
        cmpl $1,%edx
        jg .L11
    movl %ebp,%esp
    popl %ebp
    ret

Translation?
```

```
Goto Version
int fact_goto(int x)
{
    int result = 1;
    loop:
    result *= x;
    x = x-1;
    if (x > 1) goto loop;
    return result;
}

Assembly
fact_goto:
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax
    movl 8(%ebp),%edx

    .L11:
        imull %edx,%eax
        decl %edx
        cmpl $1,%edx
        jg .L11
    movl %ebp,%esp
    popl %ebp
    ret
```
General “Do-While” Translation

C Code

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

Goto Version

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

- **Body:**
  ```c
  { 
  Statement;
  ... 
  Statement;
  }
  ```
- **Test** returns integer
  - 0 interpreted as false
  - ≠ 0 interpreted as true

“While” Loop Translation

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  }
  return result;
}
```

Goto Version

```
int fact_while_goto(int x)
{
  int result = 1;
  goto middle;
  loop:
    result *= x;
    x = x-1;
  middle:
    if (x > 1)
      goto loop;
  return result;
}
```

- Used by GCC for both IA32 & x86-64
- First iteration jumps over body computation within loop straight to test

“While” Loop Example

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x--;
  }
  return result;
}
```

“While” Loop Example: Square-and-Multiply

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int p)
{
  int result;
  for (result = 1; p != 0; p = p>>1) {
    if (p & 0x1)
      result *= x;
    x = x*x;
  }
  return result;
}
```

- **Algorithm**
  - Exploit bit representation: 
    \[ p = p_0 + 2p_1 + 2^2p_2 + \ldots + 2^{n-1}p_{n-1} \]
  - Gives: 
    \[ x^p = x^{p_0} \cdot (x^{2})^{p_1} \cdot (x^{4})^{p_2} \cdot \ldots \cdot (x^{2^{n-1}})^{p_{n-1}} \]
  - Complexity \( O(\log p) = O(\text{sizeof}(p)) \)

“For” Loop Example: Square-and-Multiply

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int p)
{
  int result;
  for (result = 1; p != 0; p = p>>1) {
    if (p & 0x1)
      result *= x;
    x = x*x;
  }
  return result;
}
```

- **Algorithm**
  - Exploit bit representation: 
    \[ p = p_0 + 2p_1 + 2^2p_2 + \ldots + 2^{n-1}p_{n-1} \]
  - Gives: 
    \[ x^p = x^{p_0} \cdot (x^{2})^{p_1} \cdot (x^{4})^{p_2} \cdot \ldots \cdot (x^{2^{n-1}})^{p_{n-1}} \]
  - Complexity \( O(\log p) = O(\text{sizeof}(p)) \)

### Example

\[ 3^{10} = 3^2 \cdot 3^8 \]

\[ = 3^2 \cdot (3^2)^2 \]
ipwr Computation

/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int p)
{
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x*x;
    }
    return result;
}

before iteration | result | x=3 | p=10
-----------------|--------|-----|-----
1                | 1      | 3   | 10=1010₂
2                | 1      | 9   | 5=101₂
3                | 9      | 81  | 2=10₂
4                | 9      | 6561| 1=1₂
5                | 59049  | 43046721| 0₂

“For” Loop Example

```c
int result;
for (result = 1; p != 0; p = p>>1) {
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```

General Form
```
for (initialize; test; update) 
    Body
```
```
Init  | Test  | Update          | Body
------|-------|-----------------|---------------------
result = 1 | p != 0 | p = p >> 1     |
```

“For” → “While”

**For Version**
```
for (initialize; test; update) 
    Body
```

**Goto Version**
```
initialize;
goto middle;
loop:
    body
    update;
middle:
    if (test)
goto loop;
done:
```

**While Version**
```
initialize;
while (test) {
    body
    update;
}
```

For-Loop: Compilation

**For Version**
```
for (initialize; test; update) 
    Body
```
```
result = 1;
goto middle;
loop:
    body
    update;
middle:
    if (test)
goto loop;
done:
```

**Goto Version**
```
initialize;
goto middle;
loop:
    body
    update;
middle:
    if (test)
goto loop;
done:
```
Switch Statement Example

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

- Lots to manage, we need a *jump table*

Jump Table Structure

C code:

```c
long switch_eg (unsigned 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

<table>
<thead>
<tr>
<th>Code Blocks</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We can use the jump table when `x <= 6`:

```c
if (x <= 6)
    target = JTab[x];
goto target;
else
goto default;
```

Jump Table Structure

<table>
<thead>
<tr>
<th>C code:</th>
</tr>
</thead>
</table>

```c
switch(x) {
    case 1: <some code>
    break;
    case 2: <some code>
    case 3: <some code>
    break;
    case 5:
    case 6: <some code>
    break;
    default: <some code>
}
```

Jump Table (IA32)

<table>
<thead>
<tr>
<th>Jump table</th>
</tr>
</thead>
<tbody>
<tr>
<td>declaring data, not instructions</td>
</tr>
<tr>
<td>4-byte memory alignment</td>
</tr>
</tbody>
</table>

```
.switch .rodata
    .align 4
    .L62:
      .long   .L56  # x = 0
      .long   .L56  # x = 1
      .long   .L57  # x = 2
      .long   .L58  # x = 3
      .long   .L61  # x = 4
      .long   .L60  # x = 5
      .long   .L60  # x = 6

switch(x) {
    case 1: // .L56
        w = y*z;
        break;
    case 2: // .L57
        w = y/z;
        /* Fall Through */
    case 3: // .L58
        w += z;
        break;
    case 5:
    case 6: // .L60
        w -= z;
        break;
    default: // .L61
        w = 2;
}
```
Switch Statement Example (IA32)

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

Assembly Setup Explanation (IA32)

- **Table Structure**
  - Each target requires 4 bytes
  - Base address at .L62

- **Jumping: different address modes for target**
  - **Direct**: jmp .L61
    - Jump target is denoted by label .L61
  - **Indirect**: jmp *.L62(,%edx,4)
    - Start of jump table: .L62
    - Must scale by factor of 4 (labels are 32-bits = 4 bytes on IA32)
    - Fetch target from effective address .L62 + edx*4
      - target = JTab[x]; goto target; (only for 0 ≤ x ≤ 6)

Code Blocks (Partial)

```c
switch(x) {
    ....
    case 2: // .L57
        w = y/z;
        /* Fall Through */
    case 3: // .L58
        w += z;
        ....
    default: // .L61
        w = 2;
    }
    return w;
}
```

Jump table
```
.L62: .align 4
.long .L61 # x = 0
.long .L56 # x = 1
.long .L57 # x = 2
.long .L58 # x = 3
.long .L61 # x = 4
.long .L60 # x = 5
.long .L60 # x = 6
```
The compiler might choose to pull the return statement in to each relevant case rather than jumping out to it.
IA32 Object Code (cont.)

- Jump Table
  - Doesn’t show up in disassembled code
  - Can inspect using GDB
    - `gdb asm-cntl (gdb) x/7xw 0x080488dc`
      - Examine 7 hexadecimal format “words” (4-bytes each)
      - Use command “help x” to get format documentation

Disassembled Targets

```
0x08048630:  bb 02 00 00 00    mov $0x2, %ebx
0x08048635:  89 d8          mov %ebx, %eax
0x08048637:  5b            pop %ebx
0x08048638:  c9            leave
0x0804863a:  c3            ret
0x0804863d:  8b 45 0c      mov 0xc(%ebp), %eax
0x0804863e:  99            cltd
0x08048640:  f7 f9         idiv %ecx
0x08048642:  01 cb         add %ecx, %ebx
0x08048644:  89 d8         mov %ebx, %eax
0x08048646:  5b            pop %ebx
0x08048647:  c9            leave
0x08048648:  c3            ret
0x0804864a:  29 cb         sub %ecx, %ebx
0x0804864b:  89 d8         mov %ebx, %eax
0x0804864d:  5b            pop %ebx
0x0804864e:  c9            leave
0x0804864f:  c3            ret
0x08048650:  8b 5d 0c      mov 0xc(%ebp), %ebx
0x08048653:  0f af d9      imul %ecx, %ebx
0x08048655:  89 d8         mov %ebx, %eax
0x08048656:  5b            pop %ebx
0x08048658:  c9            leave
0x0804865a:  c3            ret
```

Matching Disassembled Targets

```
0x08048630:  bb 02 00 00 00    mov $0x2, %ebx
0x08048635:  89 d8          mov %ebx, %eax
0x08048637:  5b            pop %ebx
0x08048638:  c9            ret
0x0804863a:  8b 45 0c      mov 0xc(%ebp), %eax
0x0804863e:  99            cltd
0x08048640:  f7 f9         idiv %ecx
0x08048642:  01 cb         add %ecx, %ebx
0x08048644:  89 d8         mov %ebx, %eax
0x08048646:  5b            pop %ebx
0x08048647:  c9            leave
0x08048648:  c3            ret
0x0804864a:  29 cb         sub %ecx, %ebx
0x0804864b:  89 d8         mov %ebx, %eax
0x0804864d:  5b            pop %ebx
0x0804864e:  c9            leave
0x0804864f:  c3            ret
```

Question

- Would you implement this with a jump table?

```c
switch(a) {
    case 0: <some code>
        break;
    case 10: <some code>
        break;
    case 52000: <some code>
        break;
    default: <some code>
        break;
}
```

- Probably not:
  - Don’t want a jump table with 52001 entries for only 4 cases (too big)
  - about 200KB = 200,000 bytes
  - text of this switch statement = about 200 bytes
Quick Review

### x86-64 vs. IA32
- Integer registers: 16 x 64-bit vs. 8 x 32-bit
- `movq`, `addq`, ... vs. `movl`, `addl`, ...
  - `movq` -> "move quad word" or 4*16-bits
- x86-64: better support for passing
  function arguments in registers

### Complete memory addressing mode
- Immediate (constant), Register, and Memory Operands
  - `subl %eax, %ecx` # `ecx = ecx + eax`
  - `sall $4,%edx` # `edx = edx << 4`
  - `addl 16(%ebp),%ecx` # `ecx = ecx + Mem[16+ebp]`
  - `imull %ecx,%eax` # `eax = eax * ecx`

### Control
- 1-bit condition code registers
- Set as side effect by arithmetic instructions or by `cmp`, `test`
- Used:
  - Read out by `setx` instructions (`setg`, `setle`, ...)
  - Or by conditional jumps (`jle .L4`, `je .L10`, ...)
  - Or by conditional moves (`cmovle %edx, %eax`)
- Arithmetic operations also set condition codes
  - `subl`, `addl`, `imull`, `shrl`, etc.
- Load Effective Address does NOT set condition codes
  - `leal 4(%edx,%eax),%eax` # `eax = 4 + edx + eax`

### Do-While loop

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

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

### While-Do loop

```c
While version
while (Test)
  Body
```

```c
Do-While Version
if (!Test) 
  goto done;
  do
    Body
  while (Test);
  done:
```

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

### Control Flow Summary

- **C Control**
  - If-then-else
  - Do-while
  - While, for
  - Switch

- **Assembler Control**
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler
  - Must generate assembly code
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 (see textbook)

- **Conditions in CISC**
  - CISC machines generally have condition code registers