Definitions

- **Architecture**: (also instruction set architecture or ISA)
  The parts of a processor design that one needs to understand to write assembly code
  - “What is directly visible to software”
- **Microarchitecture**: Implementation of the architecture
  - CSE 352

- Is cache size “architecture”?
- How about CPU frequency?
- And number of registers?

Turnig C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - Use basic optimizations (~O1)
  - Put resulting machine code in file `p`

| text       | C program (p1.c p2.c) |
| text       | Asm program (p1.s p2.s) |
| binary     | Object program (p1.o p2.o) |
| binary     | Executable program (p) |

Compiling Into Assembly

<table>
<thead>
<tr>
<th>C Code</th>
<th>Generated IA32 Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>`int sum(int x, int y)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>int t = x+y;</td>
<td></td>
</tr>
<tr>
<td>return t;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sum:</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl %ebp</td>
</tr>
<tr>
<td>movl %esp,%ebp</td>
</tr>
<tr>
<td>movl 12(%ebp),%eax</td>
</tr>
<tr>
<td>addl 8(%ebp),%eax</td>
</tr>
<tr>
<td>movi %esp,%esp</td>
</tr>
<tr>
<td>popi %ebp</td>
</tr>
<tr>
<td>ret</td>
</tr>
</tbody>
</table>

Obtain with command `gcc -O1 -S code.c`

Produces file `code.s`
Machine Instruction Example

- **C Code**: add two signed integers
- **Assembly**
  - Add two 4-byte integers
  - “Long” words in GCC speak
  - Same instruction whether signed or unsigned
  - Operands:
    - x: Register %eax
    - y: Memory M[ebp+8]
    - t: Register %eax
  - Return function value in %eax
- **Object Code**
  - 3-byte instruction
  - Stored at address 0x401046

Similar to expression:
- x += y
More precisely:
- int eax;
- int y; 
- eax += ebp[2]

0x401046: 03 45 08

Object Code

- **Code for sum**
  - Add1 8(%ebp),%eax
  - 0x401040: push %ebp; mov %esp,%ebp
  - 0x401041: add 0x8(%ebp),%eax
  - 0x401042: mov %esp,%ebp
  - 0x401043: ret

- **Disassembler**
  - objdump -d p
  - Useful tool for examining object code (man 1 objdump)
  - Analyzes bit pattern of series of instructions (delineates instructions)
  - Produces near-exact rendition of assembly code
  - Can be run on either p (complete executable) or p1.o/p2.o file

Disassembling Object Code

<table>
<thead>
<tr>
<th>Disassembled</th>
<th>Disassembled</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401040 &lt;sum&gt;: push %ebp</td>
<td>0x401040 &lt;sum&gt;: push %ebp</td>
</tr>
<tr>
<td>05</td>
<td>0x55</td>
</tr>
<tr>
<td>1: 89 e5 mov %esp,%ebp</td>
<td>0x401041 &lt;sum+1&gt;: mov %esp,%ebp</td>
</tr>
<tr>
<td>3: 8b 45 0c mov 0xc(%ebp),%eax</td>
<td>0x401043 &lt;sum+3&gt;: mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>6: 03 45 08 add 0x8(%ebp),%eax</td>
<td>0x401046 &lt;sum+6&gt;: add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>9: 89 ec mov %ebp,%esp</td>
<td>0x401049 &lt;sum+9&gt;: mov %ebp,%esp</td>
</tr>
<tr>
<td>b: 5d pop %ebp</td>
<td>0x40104b &lt;sum+11&gt;: pop %ebp</td>
</tr>
<tr>
<td>c: e3 ret</td>
<td>0x40104c &lt;sum+12&gt;: ret</td>
</tr>
</tbody>
</table>

Alternate Disassembly

<table>
<thead>
<tr>
<th>Object</th>
<th>Disassembled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0x401041</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>0x401043</td>
<td>mov 0xc(%ebp),%eax</td>
</tr>
<tr>
<td>0x401046</td>
<td>add 0x8(%ebp),%eax</td>
</tr>
<tr>
<td>0x401049</td>
<td>mov %ebp,%esp</td>
</tr>
<tr>
<td>0x40104b</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>0x40104c</td>
<td>ret</td>
</tr>
</tbody>
</table>

- **Within gdb debugger**
  - gdb p disassemble sum (disassemble function)
  - x/13b sum (examine the 13 bytes starting at sum)
What Can be Disassembled?

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

Next x86 topics

- Move instructions, registers, and operands
- Memory addressing modes
- swap example: 32-bit vs. 64-bit
- Arithmetic operations

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)**

- **%eax**
- **%ecx**
- **%edx**
- **%ebx**
- **%esi**
- **%edi**
- **%esp**
- **%ebp**

General purpose, 32-bits wide

**Origin (mostly obsolete)**
- accumulate
- counter
- data
- base
- source index
- destination index
- stack pointer
- base pointer

---

**x86-64 Integer Registers**

- **%rax**
- **%eax**
- **%rbx**
- **%ebx**
- **%rcx**
- **%ecx**
- **%rdx**
- **%edx**
- **%rsi**
- **%esi**
- **%rdi**
- **%edi**
- **%rsp**
- **%esp**
- **%rbp**
- **%ebp**

64-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

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

- **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

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**

Moving Data: IA32

- **Moving Data**
  - movl *Source, Dest*:

- **Operand Types**
  - *Immediate*: Constant integer data
    - Example: $0x400, $-533
  - *Register*: One of 8 integer registers
    - Example: %eax, %edx
  - *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>Mem</td>
<td>Reg</td>
<td>movl $-147,(%eax)</td>
<td>*p_d = -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

Using Basic Addressing Modes

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

Memory Addressing Modes: Basic

- Indirect (R) Mem[Reg[Reg]]
  - Register R specifies the memory address
  ```
movl (%ecx),%eax
  ```

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

Understanding Swap

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

movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax  # eax = *yp (t1)
movl (%edx),%ebx  # ebx = *xp (t0)
movl %eax, (%edx)  # *xp = eax
movl %ebx, (%ecx)  # *yp = ebx

Understanding Swap

movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax  # eax = *yp (t1)
movl (%edx),%ebx  # ebx = *xp (t0)
movl %eax, (%edx)  # *xp = eax
movl %ebx, (%ecx)  # *yp = ebx

Understanding Swap

movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax  # eax = *yp (t1)
movl (%edx),%ebx  # ebx = *xp (t0)
movl %eax, (%edx)  # *xp = eax
movl %ebx, (%ecx)  # *yp = ebx

Understanding Swap

movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax  # eax = *yp (t1)
movl (%edx),%ebx  # ebx = *xp (t0)
movl %eax, (%edx)  # *xp = eax
movl %ebx, (%ecx)  # *yp = ebx
### Understanding Swap

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>0x120</th>
<th>0x110</th>
<th>0x10c</th>
<th>0x108</th>
<th>0x104</th>
<th>0x100</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebp</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>-4</td>
<td></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>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

%eax 456
%edx 0x124
%ecx 0x120
%ebx 123
%esi
%edi
%esp
%ebp 0x104

### Understanding Swap

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>0x120</th>
<th>0x110</th>
<th>0x10c</th>
<th>0x108</th>
<th>0x104</th>
<th>0x100</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebp</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>-4</td>
<td></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>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

%eax 456
%edx 0x124
%ecx 0x120
%ebx 123
%esi
%edi
%esp
%ebp 0x104

### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>64-bits wide</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
</tr>
<tr>
<td>%rbx</td>
</tr>
<tr>
<td>%rcx</td>
</tr>
<tr>
<td>%rdx</td>
</tr>
<tr>
<td>%rsi</td>
</tr>
<tr>
<td>%rdi</td>
</tr>
<tr>
<td>%rsp</td>
</tr>
<tr>
<td>%rbp</td>
</tr>
<tr>
<td>%r8</td>
</tr>
<tr>
<td>%r9</td>
</tr>
<tr>
<td>%r10</td>
</tr>
<tr>
<td>%r11</td>
</tr>
<tr>
<td>%r12</td>
</tr>
<tr>
<td>%r13</td>
</tr>
<tr>
<td>%r14</td>
</tr>
<tr>
<td>%r15</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 1 (4 Bytes) ↔ Quad word q (8 Bytes)

- New instruction forms:
  - movl → movq
  - addl → addq
  - sall → 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_l1(long int *xp, long int *yp)
{
    long int t0 = *xp;
    long int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Swap Long Ints in 64-bit Mode

```c
void swap_l1(long int *xp, long int *yp)
{
    long int t0 = *xp;
    long int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Arguments passed in registers (why useful?)

- First (xp) in %rdi, second (yp) in %rsi
- 64-bit pointers
- No stack operations required: faster

32-bit data

- Data held in registers %eax and %edx
- movl operation (the 1 refers to data width, not address width)

64-bit data

- Data held in registers %rax and %rdx
- movq operation
- “q” stands for quad-word
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:
  $$D(Rb,Ri,S) \quad \text{Mem}[\text{Reg}[Rb] + S \cdot \text{Reg}[Ri] + D]$$
  - 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 %ebp; %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) Mem[Reg[Rb]+Reg[Ri]] (S=1, D=0)
  - D(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D] (S=1)
  - (Rb,Ri,S) Mem[Reg[Rb]+S\cdot\text{Reg}[Ri]] (D=0)

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 $x + k \cdot i$
    - $k = 1, 2, 4, or 8$

Address Computation Examples

<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>0x100</td>
</tr>
<tr>
<td>(%edx,%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>

Some Arithmetic Operations

- Two Operand (Binary) Instructions:
  - Format
  - Computation
    - add
      - $\text{Dest} = \text{Dest} + \text{Src}$
    - sub
      - $\text{Dest} = \text{Dest} - \text{Src}$
    - imull
      - $\text{Dest} = \text{Dest} \cdot \text{Src}$
    - shll
      - $\text{Dest} = \text{Dest} << \text{Src}$
        - Also called sll
    - sarl
      - $\text{Dest} = \text{Dest} >> \text{Src}$
        - Arithmetic
    - shrl
      - $\text{Dest} = \text{Dest} >> \text{Src}$
        - Logical
    - xorl
      - $\text{Dest} = \text{Dest} \oplus \text{Src}$
    - andl
      - $\text{Dest} = \text{Dest} \& \text{Src}$
    - orl
      - $\text{Dest} = \text{Dest} | \text{Src}$

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

- **One Operand (Unary) Instructions**
  - incl Dest  
    
    Dest = Dest + 1
    
    increment
  - decl Dest  
    
    Dest = Dest - 1
    
    decrement
  - negl Dest  
    
    Dest = ~Dest
    
    negate
  - notl Dest  
    
    Dest = ~Dest
    
    bitwise complement

- See textbook section 3.5.5 for more instructions: mull, cltd, idivl, divl

Using leal for Arithmetic Expressions (IA32)

```
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)

```
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)

**int arith**

\[
\text{(int x, int y, int z)}
\]

\[
\{ \text{int t1 = x+y;}
\text{int t2 = z+t1;}
\text{int t3 = x+4;}
\text{int t4 = y * 48;}
\text{int t5 = t3 + t4;}
\text{int rval = t2 * t5;}
\text{return rval;}
\}
\]

**Stack**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>x</td>
</tr>
<tr>
<td>12</td>
<td>y</td>
</tr>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Old ebp</th>
<th>%ebp</th>
</tr>
</thead>
</table>

\[
\begin{align*}
\text{movl 8(%ebp),%eax} & \quad \# \text{eax = x} \\
\text{movl 12(%ebp),%edx} & \quad \# \text{edx = y} \\
\text{leal (%edx,%eax),%ecx} & \quad \# \text{ecx = x+y (t1)} \\
\text{leal (%edx,%edx,2),%edx} & \quad \# \text{edx = y + 2*y = 3*y} \\
\text{sal $4,%edx} & \quad \# \text{edx = 48*y (t4)} \\
\text{addl 16(%ebp),%ecx} & \quad \# \text{ecx = z+t1 (t2)} \\
\text{leal 4(%edx,%eax),%eax} & \quad \# \text{eax = 4+t4+x (t5)} \\
\text{imull %ecx,%eax} & \quad \# \text{eax = t5*t2 (rval)}
\end{align*}
\]

**Observations about arith**

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:
  - \((x+y+z) * (x+4+48*y)\)

\[
\begin{align*}
\text{movl 8(%ebp),%eax} & \quad \# \text{eax = x} \\
\text{movl 12(%ebp),%edx} & \quad \# \text{edx = y} \\
\text{leal (%edx,%eax),%ecx} & \quad \# \text{ecx = x+y (t1)} \\
\text{leal (%edx,%edx,2),%edx} & \quad \# \text{edx = y + 2*y = 3*y} \\
\text{sal $4,%edx} & \quad \# \text{edx = 48*y (t4)} \\
\text{addl 16(%ebp),%ecx} & \quad \# \text{ecx = z+t1 (t2)} \\
\text{leal 4(%edx,%eax),%eax} & \quad \# \text{eax = 4+t4+x (t5)} \\
\text{imull %ecx,%eax} & \quad \# \text{eax = t5*t2 (rval)}
\end{align*}
\]

### Another Example (IA32)

\[
\text{int logical(int x, int y)}
\]

\[
\{ \text{int t1 = x*y;}
\text{int t2 = t1 >> 17;}
\text{int mask = (1<13) - 7;}
\text{int rval = t2 & mask;}
\text{return rval;}
\}
\]

**logical:**

\[
\begin{align*}
\text{pushl %ebp} & \quad \text{Set Up} \\
\text{movl %esp,%ebp} & \quad \text{Body} \\
\text{movl 8(%ebp),%eax} & \quad \# \text{eax = x} \\
\text{xorl 12(%ebp),%eax} & \quad \# \text{eax = x*y} \\
\text{salr $17,%eax} & \quad \# \text{eax = t1>>17} \\
\text{andl $8185,%eax} & \quad \# \text{eax = t2 & 8185} \\
\text{movl %ebp,%esp} & \quad \text{Finish} \\
\text{popl %ebp} & \quad \text{ret}
\end{align*}
\]

**Stack**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>y</td>
</tr>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%ebp</th>
</tr>
</thead>
</table>
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;
}
```

```assembly
logical:
    pushl %ebp
    movl %esp,%ebp  // Set Up

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

    movl %ebp,%esp
    popl %ebp
    ret

set $t1 = x^y;
set $t2 = t1 >> 17;
set mask = (1<<13) - 7;
set rval = t2 & mask;
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;
}
```

```assembly
logical:
    pushl %ebp
    movl %esp,%ebp  // Set Up

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

    movl %ebp,%esp
    popl %ebp
    ret

set $t1 = x^y;
set $t2 = t1 >> 17;
set mask = (1<<13) - 7;
set rval = t2 & mask;
return rval;
```