The Hardware/Software Interface
CSE351 Winter 2013

x86 Programming I
Roadmap

C:
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

Assembly
language:
get_mpg:
pushq %rbp
movq %rsp, %rbp
...
popq %rbp
ret

Machine
code:
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111

Computer
system:

OS:
Windows 8
Mac

Data & addressing
Integers & floats
Machine code & C
x86 assembly
programming
Procedures &
stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Today

- Move instructions, registers, and operands
- Memory addressing modes
- \texttt{swap} example: 32-bit vs. 64-bit
- Arithmetic operations
Why learn assembly language?

- Not to be able to write programs directly in assembly
  - Compilers do that for you
- But to be able to understand the code generated by compilers, so that you can then:
  - Optimize performance of critical sections of code
  - Investigate unexpected or even buggy behavior
  - Understand how security vulnerabilities arise, and how to protect against them
Integer Registers (IA32)

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

32-bits wide

stack pointer
base pointer

general purpose
Three Basic Kinds of Instructions

- **Transfer data between memory and register**
  - *Load* data from memory into register
    - \( \% \text{reg} = \text{Mem}\[\text{address}\] \)
  - *Store* register data into memory
    - \( \text{Mem}\[\text{address}\] = \% \text{reg} \)

- **Perform arithmetic function on register or memory data**
  - \( c = a + b; \)

- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches

Remember: memory is indexed just like an array[]!
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`
    - 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 use
    - 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”

<table>
<thead>
<tr>
<th>%eax</th>
<th>%ecx</th>
<th>%edx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
</table>

Winter 2013
### 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.*
Memory vs. registers

- Why both?
Memory Addressing Modes: Basic

- **Indirect (R)**: `Mem[Reg[R]]`
  - Register R specifies the memory address

  \[ \text{movl} \ (\%ecx), \%eax \]

- **Displacement D(R)**: `Mem[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

  \[ \text{movl} \ 8(\%ebp), \%edx \]
Using Basic Addressing Modes

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

```assembly
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

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

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

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

Stack (in memory)

Register | Value
---|---
%ecx     | yp
%edx     | xp
%eax     | t1
%ebx     | t0
Understanding Swap

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

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>%eax</th>
<th>%edx</th>
<th>%ecx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x124</td>
<td>0x120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
```

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
</tr>
<tr>
<td>0</td>
<td>0x108</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
</tr>
<tr>
<td>456</td>
</tr>
<tr>
<td>0x11c</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x114</td>
</tr>
</tbody>
</table>
Understanding Swap

%eax | Offset | Address
--- | --- | ---
456 | -4 | 0x100
0x124 | 0x120 | 0x108
0x120 | 0x118 | 0x104
0x11c | 0x114 | 0x100
0x118 | 0x110 | 0x100
0x114 | 0x120 | 0x124
0x110 | 0x10c | 0x124
0x108 | Rtn adr | 0x124

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

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

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

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

```

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

 Offset

 yp  12
     0x120
     0x110
 xp  8
     0x124
     0x10c
     4
     Rtn adr
     0
     0

%ebp  0
    -4

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
```

Winter 2013
Understanding Swap

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

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
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>% rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</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:
  - \texttt{movl} \rightarrow \texttt{movq}
  - \texttt{addl} \rightarrow \texttt{addq}
  - \texttt{sall} \rightarrow \texttt{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: \texttt{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:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %esp
    ret
```

Setup

Body

Finish
Swap Ints in 64-bit Mode

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

- **32-bit data**
  - Data held in registers `%eax` and `%edx`
  - `movl` operation (the `l` refers to data width, not address width)
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;
}
```

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:**
  
  \[
  D(Rb,Ri,S) \rightarrow \text{Mem}[\text{Reg}[Rb] + S*\text{Reg}[Ri] + D]
  \]

  - \text{D}: Constant “displacement” 1, 2, or 4 bytes
  - \text{Rb}: Base register: Any of the 8/16 integer registers
  - \text{Ri}: Index register: Any, except for \%esp or \%rsp
    - Unlikely you’d use \%ebp, either
  - \text{S}: Scale: 1, 2, 4, or 8 (*why these numbers?*)

- **Special Cases:** can use any combination of \text{D}, \text{Rb}, \text{Ri} and \text{S}
  
  \[
  (Rb,Ri) \rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]]
  
  D(Rb,Ri) \rightarrow \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D]
  
  (Rb,Ri,S) \rightarrow \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]]
  \]
Address Computation Examples

<table>
<thead>
<tr>
<th>Address</th>
<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>
<td></td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
<td></td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
<td></td>
</tr>
<tr>
<td>0x80(%edx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
<td></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:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addl</code></td>
<td><code>Dest = Dest + Src</code></td>
</tr>
<tr>
<td><code>subl</code></td>
<td><code>Dest = Dest - Src</code></td>
</tr>
<tr>
<td><code>imull</code></td>
<td><code>Dest = Dest * Src</code></td>
</tr>
<tr>
<td><code>sall</code></td>
<td><code>Dest = Dest &lt;&lt; Src</code></td>
</tr>
<tr>
<td><code>sarl</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code></td>
</tr>
<tr>
<td><code>shrl</code></td>
<td><code>Dest = Dest &gt;&gt; Src</code></td>
</tr>
<tr>
<td><code>xorl</code></td>
<td><code>Dest = Dest ^ Src</code></td>
</tr>
<tr>
<td><code>andl</code></td>
<td><code>Dest = Dest &amp; Src</code></td>
</tr>
<tr>
<td><code>orl</code></td>
<td>`Dest = Dest</td>
</tr>
</tbody>
</table>

- **Also called `shll`**
- **Arithmetic**
- **Logical**

- **Watch out for argument order!** (especially `subl`)
- **No distinction between signed and unsigned int (why?)**
Some Arithmetic Operations

- One Operand (Unary) Instructions

  incl $Dest$  
  \[ Dest = Dest + 1 \]

  decl $Dest$  
  \[ Dest = Dest - 1 \]

  negl $Dest$  
  \[ Dest = -Dest \]

  notl $Dest$  
  \[ Dest = \sim Dest \]

- See textbook section 3.5.5 for more instructions: `mull`, `cltd`, `idivl`, `divl`
Using `leal` for Arithmetic Expressions

```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;
}
```

```
pushl %ebp
movl %esp,%ebp

movl 8(%ebp),%eax
movl 12(%ebp),%edx
leal (%edx,%eax),%ecx
leal (%edx,%edx,2),%edx
sall $4,%edx
addl 16(%ebp),%ecx
leal 4(%edx,%eax),%eax
imull %ecx,%eax

movl %ebp,%esp
popl %ebp
ret
```

Body

Set Up

Finish
Understanding 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;
}
```

```
movl 8(%ebp),%eax       # eax = x
movl 12(%ebp),%edx      # edx = y
leal (%edx,%eax),%ecx   # ecx = x+y   (t1)
leal (%edx,%edx,2),%edx  # edx = y + 2*y = 3*y
sall $4,%edx            # edx = 48*y  (t4)
addl 16(%ebp),%ecx      # ecx = z+t1  (t2)
leal 4(%edx,%eax),%eax  # eax = 4+t4+x  (t5)
imull %ecx,%eax         # eax = t5*t2  (rval)
```
Understanding 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;
}
```

```
movl 8(%ebp),%eax      # eax = x
movl 12(%ebp),%edx     # edx = y
leal (%edx,%eax),%ecx   # ecx = x+y (t1)
leal (%edx,%edx,2),%edx  # edx = y + 2*y = 3*y
sall $4,%edx            # edx = 48*y (t4)
addl 16(%ebp),%ecx      # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax   # eax = 4+t4+x (t5)
imull %ecx,%eax         # eax = t5*t2 (rval)
```
Understanding 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;
}
```

```
movl 8(%ebp),%eax   # eax = x
movl 12(%ebp),%edx  # edx = y
leal (%edx,%eax),%ecx # ecx = x+y (t1)
leal (%edx,%edx,2),%edx # edx = y + 2*y = 3*y
sall $4,%edx        # edx = 48*y (t4)
addl 16(%ebp),%ecx  # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax # eax = 4+t4+x (t5)
imull %ecx,%eax     # eax = t5*t2 (rval)
```
Understanding 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;
}
```

```
movl 8(%ebp),%eax     # eax = x
movl 12(%ebp),%edx    # edx = y
leal (%edx,%eax),%ecx # ecx = x+y  (t1)
leal (%edx,%edx,2),%edx # edx = y + 2*y = 3*y
sall $4,%edx          # edx = 48*y (t4)
addl 16(%ebp),%ecx    # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax # eax = 4+t4+x (t5)
imull %ecx,%eax       # eax = t5*t2 (rval)
```

```
Offset

16  z
12  y
  x
  Rtn adr
  Old %ebp
%ebp
```
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;
}
```

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

```assembly
 movl 8(%ebp),%eax  # eax = x
 movl 12(%ebp),%edx  # edx = y
 leal (%edx,%eax),%ecx  # ecx = x+y (t1)
 leal (%edx,%edx,2),%edx  # edx = y + 2*y = 3*y
 sall $4,%edx  # edx = 48*y (t4)
 addl 16(%ebp),%ecx  # ecx = z+t1 (t2)
 leal 4(%edx,%eax),%eax  # eax = 4+t4+x (t5)
 imull %ecx,%eax  # eax = t5*t2 (rval)
```
Another Example

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

```
movl 8(%ebp),%eax  # eax = x
xorl 12(%ebp),%eax  # eax = x^y
sarl $17,%eax  # eax = t1>>17
andl $8185,%eax  # eax = t2 & 8185
```
Another Example

**logical**: 
```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;
}
```

```
movl 8(%ebp),%eax            # eax = x
xorl 12(%ebp),%eax           # eax = x^y (t1)
sarl $17,%eax                # eax = t1>>17 (t2)
andl $8185,%eax              # eax = t2 & 8185
```
```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;
}
```

```
movl 8(%ebp),%eax       eax = x
xorl 12(%ebp),%eax      eax = x^y  (t1)
sarl $17,%eax           eax = t1>>17  (t2)
andl $8185,%eax         eax = t2 & 8185
```
Another Example

```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;
}
```

```
2^{13} = 8192, 2^{13} - 7 = 8185
...0010000000000000, ...0001111111111001
```

```
logical:
pushl %ebp
movl %esp,%ebp

movl 8(%ebp),%eax  ; eax = x
xorl 12(%ebp),%eax  ; eax = x^y (t1)
sarl $17,%eax  ; eax = t1>>17 (t2)
andl $8185,%eax  ; eax = t2 & 8185

movl %ebp,%esp
popl %ebp
ret
```