211: Computer Architecture
Summer 2016

Liu Liu

Topic:
  ▪ Assembly Programming
Recap

- **Data representation:**
  - Endian / Complement
  - Floating number
  - ASCII
  - Add / Sub on unsigned / signed numbers
  - Overflow Detection

- **Assembly Programming:**
  - Relationship between Assembly, Hardware and High-Lvl language
  - Basic components: ALU / Registers / Memory / PC / Control Logic
  - Assembly Instruction Format
Today’s Topic

- Assembly Programming:
  - addressing mode
  - Data Movement Instruction: movl
  - Addressing Instruction: leal
  - Arithmetic Instruction: addl / subl / imull / idivl …
Assembly Programmer’s View

CPU

- ALU
- Control Logic
- PC
- Registers

Memory

(OS code & data)
Object Code
Program Data

Addresses
Data
Instructions
Memory Access: Read

CPU

Memory

Addresses

00 00101100
01 10001000
02 11111111
03 01010101
04 00000000
05 11000001
06 00000000
07 11111001
08 11111000
09 00110000
0A 00000000
0B 00000000
0C 00000000
0D 11000011
0E 00011001
0F 00000000

Storage
Memory Access: Write

Memory Addresses

CPU

Memory

Storage

00 00101100
01 10001000
02 11111111
03 01010101
04 00000000
05 11000001
06 00000000
07 11111001
08 11111000
09 00110000
0A 00000000
0B 00000000
0C 00000000
0D 11000011
0E 00011001
0F 00000000
Memory Access: Write

![Diagram showing memory access and CPU connections with hexadecimal addresses and values.]

- CPU
- Memory Addresses
- Storage
- Addresses:
  - 00: 00101100
  - 01: 10001000
  - 02: 11111111
  - 03: 01010101
  - 04: 01010101
  - 05: 11000001
  - 06: 11111001
  - 07: 11111000
  - 08: 00110000
  - 09: 00000000
  - 0A: 00000000
  - 0B: 00000000
  - 0C: 00000000
  - 0D: 11000011
  - 0E: 00011001
  - 0F: 00000000
Processor: ALU & Registers

\[ C = F_S(A, B) \]

F includes
- Arithmetic: +, -, *, /, ~, etc.
- Logical: <, >, =, etc.

![Diagram of ALU](image)

Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>00101100</td>
</tr>
<tr>
<td>R1</td>
<td>10001000</td>
</tr>
<tr>
<td>R2</td>
<td>11111111</td>
</tr>
<tr>
<td>R3</td>
<td>01010101</td>
</tr>
</tbody>
</table>
Putting It All Together

CPU

- Condition Codes
- Program Counter (PC)
- ALU
- Control Logic
- Registers

Memory

Addresses

- Addresses:
  - 00: 00101100
  - 01: 10001000
  - 02: 11111111
  - 03: 01010101
  - 04: 01010101
  - 05: 11000001
  - 06: 00000000
  - 07: 11111001
  - 08: 11111000
  - 09: 00110000
  - 0A: 00000000
  - 0B: 00000000
  - 0C: 00000000
  - 0D: 11000011
  - 0E: 00011001
  - 0F: 00000000

Storage
Putting It All Together

CPU

Condition Codes

Program Counter (PC)

ALU

Control Logic

Registers

Memory

Storage

Addresses

00 00101100
01 10001000
02 11111111
03 01010101
04 01010101
05 11000001
06 00000000
07 11111001
08 11111000
09 00110000
0A 00000000
0B 00000000
0C 00000000
0D 11000011
0E 00011001
0F 00000000
Putting It All Together

CPU

Condition Codes

Program Counter (PC)

ALU

Control Logic

Registers
R0: x
R1: y

Memory

Addresses

Storage

00 00101100
01 10001000
02 11111111
03 01010101
04 01010101
05 11000001
06 00000000
07 11111001
08 11111000
09 00110000
0A 00000000
0B 00000000
0C 00000000
0D 11000011
0E 00011001
0F 00000000
Putting It All Together

CPU

Condition Codes

Program Counter (PC)

ALU

Control Logic

Registers
R0: x
R1: y

Memory

Addresses
00  00101100
01  10001000
02  11111111
03  01010101
04  01010101
05  11000001
06  00000000
07  11111001
08  11111000
09  00110000
0A  00000000
0B  00000000
0C  00000000
0D  11000011
0E  00011001
0F  00000000

Storage
Putting It All Together

CPU

Condition Codes

Program Counter (PC)

ALU

Control Logic

Registers

R0: x
R1: y
R2: z

Memory

Storage

Addresses

00 00101100
01 10001000
02 11111111
03 01010101
04 01010101
05 11000001
06 00000000
07 11111001
08 11111000
09 00110000
0A 00000000
0B 00000000
0C 00000000
0D 11000011
0E 00011001
0F 00000000

Rutgers University

Liu Liu

15
Basic CPU Function

- FETCH[PC++]
- DECODE
- EXECUTE

Arithmetic: +, -, *, /  
Logic: bre, jmp
Data Formats

Byte: 8 bits
- E.g., char

Word: 16 bits (2 bytes)
- E.g., short int

Double Word: 32 bits (4 bytes)
- E.g., int, float

Quad Word: 64 bits (8 bytes)
- E.g., double

Instructions can operate on any data size
- `movl, movw, movb`
  - Move double word, word, byte, respectively
- End character specifies what data size to be used
x86 Registers

General purpose registers are 32 bit
- Although operations can access 8-bits or 16-bits portions

Originally categorized into two groups with different functionality
- Data registers (EAX, EBX, ECX, EDX)
  - Holds operands
- Pointer and Index registers (EBP, ESP, EIP, ESI, EDI)
  - Holds references to addresses as well as indexes

Now, the registers are mostly interchangeable

Segment registers
- Holds starting address of program segments
  - CS, DS, SS, ES
# x86 Registers

<table>
<thead>
<tr>
<th>16 BITS</th>
<th>8 BITS</th>
<th>32 BITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>AX</td>
<td>AH</td>
</tr>
<tr>
<td>ECX</td>
<td>CX</td>
<td>CH</td>
</tr>
<tr>
<td>EDX</td>
<td>DX</td>
<td>DH</td>
</tr>
<tr>
<td>EBX</td>
<td>BX</td>
<td>BH</td>
</tr>
</tbody>
</table>

- **ESP** — Stack Pointer
- **EBP** — Base register of current stack frame
- **ESI** — Source index for string operations
- **EDI** — Destination index for string operations
Addressing

Accessing various data from various locations:

- access data from register
- access immediate value (constant)
- access data from memory
  - primitive data type
  - array element
  - ,etc
Register Mode Addressing

Use % to denote register
- E.g., %eax

Source operand: use value in specified register

Destination operand: use register as destination for value

Examples:
- movl %eax, %ebx
  - Copy content of %eax to %ebx
Immediate Addressing

Operand is immediate

- Operand value is found immediately following the instruction
- $ in front of immediate operand
- E.g., movl $0x4040, %eax

```
0x4040
```
Direct Addressing

Address of operand is found immediately after the instruction
- Also known as direct addressing or absolute address
- movl %eax, 0x0000f
Indirect Mode Addressing

Content of operand is an memory address
  ▪ Designated as parenthesis around operand

Offset can be specified as immediate mode

Examples:
  ▪ movl (%ebp), %eax
    • Copy value from memory location whose address is in ebp into eax
  ▪ movl -4(%ebp), %eax
    • Copy value from memory location whose address is -4 away from content of ebp into eax
Indexed Mode Addressing

Add content of two registers to get address of operand

- movl (%eax, %esi), %eax
  - Copy value at (address = eax + esi) into eax
- movl 8(%eax, %esi), %eax
  - Copy value at (address = 8 + eax + esi) into eax

Useful for dealing with arrays

- If you need to walk through the elements of an array
- Use one register to hold base address, one to hold index
  - E.g., implement C array access in a for loop
Scaled Mode Addressing

get address with scale factor:
- (displacement, (base, index, scale))
- movl (%eax(%esi, %ebx, 2)), %eax
  - Copy value at (address = eax + esi + ebx*2) into eax
  - e.g. int a[10][10]
  - a = 0x0010
  - how to get a[1][1]’s memory address?
    - displacement = 0x0010
    - base = 10 * 4 = 0x0028 (address of the second row)
    - index = 1
    - scale = 4
    - Final value = 0x0010 + 0x0028 + 1*4 = 0x003c
Scaled Mode Addressing

get address with scale factor:

- (displacement, (base, index, scale))
- movl (%eax(%esi, %ebx, 2)), %eax
  - Copy value at (address = eax + esi + ebx*2) into eax

- e,g struct mystr a[10]
- a = 0x0010
- how to get a[1].field1 ‘s memory address?
  - displacement = offset of the field in structure
  - base = 0x0010(pointer to starting addr)
  - index = 1
  - scale = sizeof(mystr)
# Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>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>(%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>

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

Most common instruction is data transfer instruction

- mov S, D
  - Copy value at S to D

Used to copy data from: (what’s missing?)

- Memory to register
- Register to memory
- Register to register
- Constant to register
### movl Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Imm</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg</td>
</tr>
<tr>
<td>Mem</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg</td>
</tr>
</tbody>
</table>

- Cannot do memory-memory transfers with single instruction
Stack Operations

By convention, %esp is used to maintain a stack in memory
  ▪ Used to support C function calls

%esp contains the address of top of stack

Instructions to push (pop) content onto (off of) the stack
  ▪ pushl %eax
    • esp = esp – 4
    • Memory[esp] = eax
  ▪ popl %ebx
    • ebx = Memory[esp]
    • esp = esp + 4

Where does the stack start? We’ll discuss later
Example: Swap

```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
```

Body

Set Up (Prologue)

Finish (Epilogue)
Function Call

When calling a procedure, caller needs to:

1) save the registers whose value needs to be preserved
2) pass arguments
3) provide return address
4) retrieve return value
x86 Registers

Caller-save registers

- caller-saved registers are used to hold temporary quantities that need not be preserved across calls.

Callee-save registers

- callee-saved registers are used to hold long-lived values that should be preserved across calls.

IA32 convention

- caller-save: %eax, %ecx, %edx (save before call)
- callee-save: %ebx, %esi, %edi (save before use)
Argument

How to pass arguments to a function?

- How about using registers?
- Problems?
  - nested? >32bit?

Parameters are pushed to stack by “caller” before making the function call
Function Call(Caller)

When calling a procedure, caller needs to:
1) *push caller-save registers if needed
2) push arguments onto stack
3) push return address onto stack
4) finally, execute “callee”
After calling a procedure, caller needs to:
1) *push caller-save registers if needed
2) push arguments onto stack (right to left, why?)
3) push return address onto stack
4) finally, execute "callee"
After calling a procedure, caller needs to:
1) *push caller-save registers if needed
2) push arguments onto stack
3) push return address (current %eip) onto stack (where should the program continue after returned)
4) finally, execute “callee”

These 2 steps are done by instruction “call”

Good to go
After calling a procedure, caller needs to:
1) *push caller-save registers if needed
2) push arguments onto stack
3) push return address (current %eip) onto stack (where should the program continue after returned)
4) finally, execute "callee"

Function Call: Call Instruction

Call Instruction

. . .
push $0x120 #yp
push $0x124 #xp
call swap

Good to go
Before executing the body, callee needs to:

1) save the previous %ebp (why?)
2) update the %ebp
3) push "callee-save" registers to stack (so that the callee could use those registers)

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

Set Up (Prologue)
Beginning of a procedure: Prologue

Before executing the body, callee needs to:

1) save the previous %ebp (why?)
2) update the %ebp
3) push “callee-save”
   registers to stack (so that the callee could use those registers)

swap:

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

Set Up (Prologue)
Beginning of a procedure: Prologue

Before executing the body, callee needs to:

1) save the previous %ebp (why?)
2) update the %ebp
3) push “callee-save” registers to stack (so that the callee could use those registers)

swap:

pushl %ebp
movl %esp,%ebp
pushl %ebx

Set Up (Prologue)
```c
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
```
Procedure Body

\[
\begin{align*}
\text{movl } & 12(\%ebp), \%ecx \quad \# \text{ecx} = \text{yp} \\
\text{movl } & 8(\%ebp), \%edx \quad \# \text{edx} = \text{xp} \\
\text{movl } & (\%ecx), \%eax \quad \# \text{eax} = *\text{yp} \ (t1) \\
\text{movl } & (\%edx), \%ebx \quad \# \text{ebx} = *\text{xp} \ (t0) \\
\text{movl } & \%eax, (\%edx) \quad \# \ast\text{xp} = \text{eax} \\
\text{movl } & \%ebx, (\%ecx) \quad \# \ast\text{yp} = \text{ebx}
\end{align*}
\]
Procedure Body

```
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
```
**Procedure Body**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

**Labels:***
- yp
- xp

**Instructions:**
- `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

**Address Table:**

<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>0</td>
<td>0x108</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

**Variables:**
- Old %ebp
- Old %ebx
- %esp
- %ebp
Procedure Body

```assembly
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
```
Procedure Body

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
### Procedure Body

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

#### Offset

<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>0x110</td>
</tr>
<tr>
<td>0</td>
<td>0x108</td>
</tr>
</tbody>
</table>

#### Instruction

- `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
Procedure Body

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</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
```
Before return: Epilogue

After calling a procedure, caller needs to:
1) restore the "caller save" register
2) clear up the stack
3) reset %ebp
4) finally return

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret

Finish (epilogue)
Before return: Epilogue

After calling a procedure, caller needs to:

1) restore the “caller save” register
2) clear up the stack
3) reset %ebp
4) finally return

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Finish (epilogue)
Epilogue

After calling a procedure, caller needs to:

1) restore the "caller save" register
2) clear up the stack
3) reset %ebp
4) finally return

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Finish (epilogue)
Epilogue: RET instruction

The last step of a function call, returning to where it made the call:
1) restore the “caller save” register
2) clear up the stack
3) reset %ebp
4) finally return (ret instruction takes the return address and do the rest of clean up)

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret

Finish (epilogue)
Address Computation Instruction

leal: compute address using addressing mode without accessing memory

leal src, dest

- src is address mode expression
- Set dest to address specified by src

Use

- Computing address without doing memory reference
  - E.g., translation of \( p = \&x[i] \);

Example:

- leal 7(%edx, %edx, 4), %eax
  - \( eax = 4*edx + edx + 7 = 5*edx + 7 \)
## Some Arithmetic Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addl</code> <code>Src,Dest</code></td>
<td><code>Dest = Dest + Src</code></td>
</tr>
<tr>
<td><code>subl</code> <code>Src,Dest</code></td>
<td><code>Dest = Dest - Src</code></td>
</tr>
<tr>
<td><code>imull</code> <code>Src,Dest</code></td>
<td><code>Dest = Dest * Src</code></td>
</tr>
<tr>
<td><code>sall</code> <code>Src,Dest</code></td>
<td><code>Dest = Dest &lt;&lt; Src (left shift)</code></td>
</tr>
<tr>
<td><code>sarl</code> <code>Src,Dest</code></td>
<td><code>Dest = Dest &gt;&gt; Src (right shift)</code></td>
</tr>
<tr>
<td><code>xorl</code> <code>Src,Dest</code></td>
<td><code>Dest = Dest ^ Src</code></td>
</tr>
<tr>
<td><code>andl</code> <code>Src,Dest</code></td>
<td><code>Dest = Dest &amp; Src</code></td>
</tr>
<tr>
<td><code>orl</code> <code>Src,Dest</code></td>
<td>`Dest = Dest</td>
</tr>
</tbody>
</table>
## Some Arithmetic Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>incl <em>Dest</em></td>
<td><em>Dest</em> = <em>Dest</em> + 1</td>
</tr>
<tr>
<td>decl <em>Dest</em></td>
<td><em>Dest</em> = <em>Dest</em> - 1</td>
</tr>
<tr>
<td>negl <em>Dest</em></td>
<td><em>Dest</em> = - <em>Dest</em></td>
</tr>
<tr>
<td>notl <em>Dest</em></td>
<td><em>Dest</em> = ~ <em>Dest</em></td>
</tr>
</tbody>
</table>
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;
}
```

```assembly
arith:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    addl 16(%ebp),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax
    movl %ebp,%esp
    popl %ebp
    ret
```
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;
}
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

```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;
}
```

\[2^{13} = 8192, \quad 2^{13} - 7 = 8185\]

```
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
```
What’s missing?

Control Flow (Excluded from Midterm)
Questions about Project?