Machine-Level Representation of Programs #1 — 4190.308 Computer Architecture

Due Date: Wednesday, September 25, 2014, 23:59 **Solution**

Submission: **Please update your e-mail address and mobile phone number on the eTL** in paper form.
There will be a drop off box in class and in front of the CSAP Lab in building 301, room 419.

Question 1

Load effective address

Suppose register `%eax` holds value `x` and `%ecx` holds value `y`. Fill in the table below with formulas indicating the value that will be stored in register `%edx` for each of the given assembly code instructions:

Instruction	Result
`leal 8(%eax), %edx`	x + 8
`leal (%eax,%ecx), %edx`	x + y
`leal (%eax,%ecx,8), %edx`	x + 8y
`leal 10(%eax,%eax,4), %edx`	5x + 10
`leal 0xF(,%ecx,4), %edx`	4y + 16
`leal 4(%eax,%ecx,4), %edx`	x + 4y + 4

Question 2
Unary and Binary Operations

Assume the following values are stored at the indicated memory addresses and register:

Address	Value
0x100	0xDF
0x104	0xCB
0x108	0x25
0x10C	0x14

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x2

Fill in the following table showing the effects of the following instructions, both in terms of the register or memory location that will be updated and the resulting value:

Instruction	Destination	Value
addl %ecx,(%eax)	0x100	0xE0
subl %edx,4(%eax)	0x104	0xC9
imull $15,(%eax,%edx,4)	0x108	0x250
incl 0xC(%eax)	0x10c	15
decl %ecx	%ecx	0
subl %edx,%eax	%eax	0xFE

Question 3
Shift Operations

Suppose we want to generate assembly code for the following C function:

```
int fun(int x, int y, int n)
{
    x <<= 4;
    y <<= 2;
    return ((x+y)<<n);
}
```

The code that follows is a portion of the assembly code that performs the actual shifts and leaves the final value in register `%eax`. Two key instructions have been omitted. Parameters `x, y,` and `n` are stored at memory locations with offsets `0x8`, `0xc`, and `0x10` respectively, relative to the address in register `%ebp`.

`x` at `%ebp+0x8`, `y` at `%ebp+0xc`, `n` at `%ebp+0x10`

```
mov    0x8(%ebp),%eax
mov    0xc(%ebp),%edx
mov    0x10(%ebp),%ecx
shl    $0x4,%eax
lea    (%eax,%edx,4),%eax
shl    %cl,%eax
ret
```

Fill in the missing instructions.

Question 4
Arithmetic Operations

In the following variant of the function, the expressions have been replaced by blanks:

```
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x<<2;
    int t4 = t3+t2;
    int t5 = t1*t4;
    return t5;
}
```

The portion of the generated assembly code implementing these expressions is as follows:

`x` at `%ebp+0x8`, `y` at `%ebp+0xc`, `z` at `%ebp+0x10`

```
mov    0x8(%ebp),%eax
mov    0xc(%ebp),%edx
mov    0x10(%ebp),%ecx
add    %eax,%edx
add    %edx,%ecx
lea    (%ecx,%eax,4),%eax
imul   %edx,%eax
ret
```

Based on this assembly code, fill in the missing portions of the C code.

Question 5
Condition Codes

The CPU maintains a set of single-bit *condition code* registers describing attributes of the most recent arithmetic of logical operation. These registers can then be tested to perform conditional branches.
Fill in the omitted entries in the following table.

CF	Carry Flag	The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.
ZF	Zero Flag	The most recent operation yielded zero
SF	Sign Flag	The most recent operation yielded a negative value
OF	Overflow Flag	The most recent operation caused a two's-complement overflow—either negative or positive

Question 6
Translating Conditional Branches

Starting with C code of the form

```
int branch(int x, int y) {
    int val;
    if (x < -3) {
        val = x*y;
    } else if (x > 2) {
        val = x-y;
    } else {
        val = x+y;
    }
    return val;
}
```

or

```
int branch(int x, int y) {
    int val;
    if (x < -3) {
        val = x*y;
    } else if (x < 3) {
        val = x+y;
    } else {
        val = x-y;
    }
    return val;
}
```

GCC generates the following assembly code:

```
 3: mov    0x8(%ebp),%edx
 6: mov    0xc(%ebp),%ecx
 9: cmp    $0xfffffffd,%edx
 c: jge    15 <branch+0x15>
 e: mov    %ecx,%eax
10: imul   %edx,%eax
13: jmp    21 <branch+0x21>
15: mov    %edx,%eax
17: sub    %ecx,%eax
19: add    %edx,%ecx
1b: cmp    $0x3,%edx
1e: cmovl  %ecx,%eax
21: pop    %ebp
22: ret
```

Fill in the missing expressions in the C code.

Question 7
if statements

A function *fun* has the following overall structure and the GCC C compiler has generated the following assembly code for it:

```
int fun(int x, int y, int z)
{
  int val = 0;

  if (x>0) {

    val = 2*x + 12;

  } else if(y>z+1) {

    val = x + y + 4;

  } else if(z>5) {

    val = x + z;

  } else {

    val = x + y + z;

  }

  return val;

}
```

```
x at %ebp+8, y at %ebp+12, z at %ebp+16

fun:
  movl  8(%ebp), %eax
  movl  12(%ebp), %ecx
  movl  16(%ebp), %edx
  testl %eax, %eax
  jle .L2
  leal  12(%eax,%eax), %eax
  jmp .L3
.L2:
  leal  1(%edx), %ebx
  cmpl  %ecx, %ebx
  jge .L4
  leal  4(%eax,%ecx), %eax
  jmp .L3
.L4:
  addl  %eax, %ecx
  leal  (%edx,%eax), %eax
  addl  %edx, %ecx
  cmpl  $6, %edx
  cmovl %ecx, %eax
.L3:
  popl  %ebx
  popl  %ebp
  ret
```

Use the assembly code version to fill in the missing parts of the C code.

Question 8
for loops

A function *fun* has the following overall structure and the GCC C compiler has generated the following assembly code for it:

```
int fun(unsigned x) {
  int val = 0;
  int i;
  for (i = 1; i < x+1; i++) {

    if(i%2 == 0) {

      val += i;
    }
  }
  return val;
}
```

```
x at %ebp+8

fun:
  pushl %ebp
  xorl  %eax, %eax
  movl  %esp, %ebp
  movl  $1, %edx
  pushl %ebx
  movl  8(%ebp), %ebx
  addl  $1, %ebx
  cmpl  $1, %ebx
  ja  .L7
  jmp .L2
.L5:
  leal  (%eax,%edx), %ecx
  testb $1, %dl
  cmove %ecx, %eax
.L7:
  addl  $1, %edx
  cmpl  %edx, %ebx
  ja  .L5
.L2:
  popl  %ebx
  popl  %ebp
  ret
```

Reverse engineer the operation of this code and then do the following:

a) Use the assembly code version to fill in the missing parts of the C codes.

b) Describe what this function computes.

Sum of all even numbers i ($0 < i < x+1$)

Question 9
switch statements

For the following C function `switcher` GCC generates assembly code and a jump table as shown below:

```
int switcher(int a, int b, int c)
{
  int answer;
  switch(a) {

    case 4: /* Case A */

      c = b & 15;

      /* Fall through */

    case 7: /* Case B */

      answer = c - 138;
      break;

    case 5: /* Case C */

    case 0: /* Case D */

      answer = (c + 3) * b;
      break;

    case 2: /* Case E */

      answer = 4;
      break;
    default:

      answer = b * a;
  }
  return answer;
}
```

```
a at %ebp+8, b at %ebp+12, c at %ebp+16

switcher:
  movl  8(%ebp), %edx
  movl  12(%ebp), %ecx
  movl  16(%ebp), %eax
  cmpl  $7, %edx
  jbe .L10
.L2:
  movl  %ecx, %eax
  imull %edx, %eax
  ret
.L10:
  jmp *.L7(,%edx,4)
.L5:
  movl  %ecx, %eax
  andl  $15, %eax
.L6:
  subl  $138, %eax
  ret
.L8:
  movl  $4, %eax
  ret
.L3:
  addl  $3, %eax
  imull %ecx, %eax
  ret
.L7:
  .long .L3
  .long .L2
  .long .L8
  .long .L2
  .long .L5
  .long .L3
  .long .L2
  .long .L6
```

Fill in the missing parts of the C code. Except for the ordering of case labels `C` and `D`, there is only one way to fit the different cases into the template.