Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Fall Term 2018

SYSTEMS PROGRAMMING AND COMPUTER ARCHITECTURE
Sample Solution 5: Assembly

Assigned on: **24th Oct 2018**
Due by: **30th Oct 23:59 2018**

1 Assembly basics

(*Note: the size of data types and assembly code in this assignment assume an x86-64 machine.*)

1.1 Array Basics

	size			access through		
declaration	element	array	address of element i	index	pointer	dereference index=2
`char A[5];`	1	5	$x_A + i$	`A[i]`	`*(A+i)`	`char v = A[2];`
`char *B[3];`	8	24	$x_B + 8i$	`B[i]`	`*(B+i)`	`char v = *(B[2]);`
`char **C[8];`	8	64	$x_C + 8i$	`C[i]`	`*(C+i)`	`char v = **(C[2]);`
`short D[4];`	2	8	$x_D + 2i$	`D[i]`	`*(D+i)`	`short v = D[2] ;`
`short *E[9];`	8	72	$x_E + 8i$	`E[i]`	`*(E+i)`	`short v = *(E[2])`
`int F[4];`	4	16	$x_F + 4i$	`F[i]`	`*(F+i)`	`int v = F[2]`
`int *G[7];`	8	56	$x_G + 8i$	`G[i]`	`*(G+i)`	`int v = *(G[2])`

1.2 Addressing modes

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

Address	Value
0x204	0xFF
0x208	0xCD
0x20C	0x21
0x210	0x11

Register	Value
%rax	0x2
%rcx	0x204
%rdx	0x3

Fill in the following table showing the types (i.e., immediate, register, memory) and the values of the indicated operands:

Operand	Type	Address	Value
%rax	Register	—	0x2
0x210	Memory (absolute address)	0x210	0x11
$0x210	Immediate	—	0x210
(%rcx)	Memory (indirect)	0x204	0xFF
4(%rcx)	Memory (base + displacement)	0x204 + 0x4 = 0x208	0xCD
5(%rcx, %rdx)	Memory (indexed)	0x5 + 0x204 + 0x3 = 0x20C	0x21
519(%rdx, %rax)	Memory (indexed)	0x207 + 0x3 + 0x2 = 0x20C	0x21
0x204(, %rax, 4)	Memory (scaled indexed)	0x204 + 0x2 * 0x4 = 0x20C	0x21
(%rcx, %rax, 2)	Memory (scaled indexed)	0x204 + 0x2 * 0x2 = 0x208	0xCD

1.3 Arithmetic operations

Use the values of the memory addresses and registers from Question 1. Handle the different instructions independently. The result of one of the instructions does not affect the others. 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 %eax, (%rcx)`	0x204	0xFF + 0x2 = 0x101
`subl %edx, 4(%rcx)`	0x208	0xCD - 0x3 = 0xCA
`imull (%rcx, %rax, 4), %eax`	%eax	0x21 * 0x2 = 0x42
`incl 8(%rcx)`	0x20C	0x21 + 0x1 = 0x22
`decl %eax`	%eax	0x2 - 0x1 = 0x1
`subl %edx, %ecx`	%ecx	0x204 - 0x3 = 0x201

1.4 leal and movl

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

Address	Value
0x108	0xFF
0x10C	0xCD
0x110	0x21

Register	Value
%rax	0x100
%rcx	0x4
%rdx	0x1

What is the difference between the two instructions? What value ends up in %ecx? Write the formula!

The movl instruction computes an address and then moves some data.

```
movl 8(%rax, %rdx, 4), %ecx

ecx <- Mem[8 + R[rax] + R[rdx] * 4]
ecx <- Mem[8 + 100 + 1 * 4] = Mem[0x10C] = 0xCD
```

The leal instruction computes an address and stores the computed address in a register. There is no memory access!

```
leal 8(%rax, %rdx, 4), %ecx

ecx <- 8 + R[rax] + R[rdx] * 4]
ecx <- 8 + 100 + 1 * 4 = 0x10C
```

1.5 Condition codes

Consider the instruction `addl` %rax, %rbx. As a side-effect, it sets the condition flags (OF, SF, ZF, CF) according to the result.

Assuming a 4-bit machine, convert the given decimal pairs `(a, b)` to their binary representation and perform the addition. Give both the arithmetical value and the interpreted value (2's complement) of the result. List the condition flags that are set.

- (-1, -1): SF, CF

```
    -1   1111
+   -1   1111
        1111
    ---------
    -2   1110
```

- (+4, -8): SF

```
    +4   0100
+   -8   1000
        0000
    ---------
    -4   1100
```

- (TMax, TMax): SF, OF

```
    +7   0111
+   +7   0111
        0111
    ---------
    -2   1110 (arith. +14)
```

- (TMax, -TMax): CF, ZF

```
    +7   0111
+   -7   1001
        1111
    ---------
     0   0000
```

- (TMin, TMax): SF

```
    -8   1000
+   +7   0111
        0000
    ---------
    -1   1111
```

- (TMin, TMin): ZF, CF, OF

```
    -8   1000
+   -8   1000
        1000
    ---------
     0   0000 (arith. -16)
```

- (-1, TMax): CF

```
    -1   1111
+   +7   0111
        1111
    ---------
    +6   0110
```

- (2, 3): —

```
    +2   0010
+   +3   0011
        0010
    ---------
    +5   0101
```

1.6 Reading Condition Codes with C

As we can see in Volume 1, Section 3.4.3 of the Intel Manual, the CF, ZF, SF, and OF condition codes correspond to bits 0, 6, 7, and 11 of the EFLAGS register. We can thus obtain their value with the following code.

```
struct ccodes getccodes(unsigned eflags)
{
  struct ccodes ccodes;

  ccodes.cf = eflags & 0x1;
  ccodes.zf = (eflags >> 6) & 0x1;
  ccodes.sf = (eflags >> 7) & 0x1;
  ccodes.of = (eflags >> 11) & 0x1;

  return ccodes;
}
```

2 Assembly control flow

2.1 Assembly Code Fragments

a)
```
int f1(int a, int b) {          f1:   pushq   %rbp
                                      movq    %rsp, %rbp
    return a - b;                     movl    %edi, %eax
                                      subl    %esi, %eax
}                                     movq    %rbp, %rsp
                                      popq    %rbp
                                      ret
```

b)
```
int f2(int a) {                 f2:   pushq   %rbp
                                      movq    %rsp, %rbp
    return a*5;                       leal    (%rdi,%rdi,4), %eax
                                      movq    %rbp, %rsp
                                      popq    %rbp
                                      ret
}
```

c)
```
int f3(int a) {                 f3:   pushq   %rbp
                                      movq    %rsp, %rbp
                                      testl   %edi, %edi   # or cmpl $0, %edi
    if (a <= 0)                       movl    %edi, %eax
      return -a;                      jle     .L11
    else                       .L8:   movq    %rbp, %rsp
      return a;                       popq    %rbp
                                      ret
}                              .L11:  negl    %eax
                                      jmp     .L8
```

2.2 Conditional branches

What is the value of %eax, when the last label (respectively `.L3` and `.L17`) is reached? First, annotate the assembly code and then, write the corresponding C-statements!

i) Assume %eax := `a`, %edx := `d`.

```
        ...                         #   eax := a
                                    #   edx := d
        cmpl    %eax, %edx          # if (edx <= eax)
        jle     .L2                 #   goto Else
                                    # Then:
        subl    %eax, %edx          #   edx := edx - eax
        movl    %edx, %eax          #   eax := edx
        jmp     .L3                 #
.L2:                                # Else:
        subl    %edx, %eax          #   eax := eax - edx
.L3:                                # End:
        ...
```

Solution: $\%\mathsf{eax} := |(\mathsf{a} - \mathsf{d})|$

```
    int t;
    if (d > a) {
       t = d - a;
    } else {
       t = a - d;
    }
```

ii) Assume %eax := `1`, %ecx := `N`.

```
        ...                         #   eax := 1
                                    #   ecx := N
        testl   %ecx, %ecx          #   if (ecx <= 0)
        jle     .L17                #     goto End
        xorl    %edx, %edx          #   edx := 0
.L18:                               # Loop:
        incl    %edx                #   edx++
        addl    %eax, %eax          #   eax := eax + eax
        cmpl    %edx, %ecx          #   compare: ecx - edx
        jne     .L18                #   if edx != ecx goto Loop
.L17:                               # End:
        ...
```

Solution: $\%\mathsf{eax} := 2^{\mathsf{N}}$

```
    int t = 1;
    for (int i = 0; i < N; i++) {
        t = t * 2;
    }
```

2.3 For Loop

The following C code corresponds to the assembly code given:

```
int dog (int x, int y) {
    int result = 1;
    for (int i = x; i < y; i = i + 2) {
        result = result * i;
    }
    return result;
}
```

2.4 Switch Statement

The assembly code is hand-coded, i.e. it is not generated by gcc.

- Fragment 3 matches.
- Fragment 1 does not break, and it returns a default value of 0.
- Fragment 2 corresponds to the following C code:

```
int fragment2(int a, int r) {
    int ret = 0;
    switch (a) {
    case 1:
        ret = 4;
        break;
    case 2:
    case 5:
        ret = 7;
        break;
    case 3:
    case 4:
        ret = 11;
        break;
    default:
        ret = 1;
    }
    return ret;
}
```

The compiler-generated assembly code of function woohoo (compiled with `gcc -O1 -S`):

```
woohoo:
    ...                   % edi := a
    subl    $11, %edi     % edi := a - 11
    cmpl    $44, %edi     % cmp: eax - 44
    ja    .L2
    movl    %edi, %edi
    jmp    *.L4(,%rdi,8)
    .section    .rodata
    .align 8
    .align 4
.L4:
    .quad    .L3          # jump_table [0] -> a == 11
    .quad    .L2
```

```
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L7            # jump_table [11] -> a == 22
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L6            # jump_table [22] -> a == 33
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L6            # jump_table [33] -> a == 44
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L2
    .quad    .L7            # jump_table [44] -> a == 55
    .text
.L3:
    movl    $4, %eax
    ret
.L6:
    movl    $11, %eax
    ret
.L2:
    movl    $1, %eax        # default: return 1
    ret
.L7:
    movl    $7, %eax
    ret
```