CPSC 313 2012 – Mike Feeley
Assignment #2

Name: Matthew Park
Student Number: 34017103

Problem 1:

Pseudo-code for **rmmovl**

Fetch:
- icode:ifun ← M_1[PC]
- rA:rB ← M_1[PC+1]
- valC ← M_4[PC+2]
- valP ← PC+6

Decode:
- valA ← R[rA]
- valB ← R[rB]

Execute:
- valE ← valC + (valB*ifun)

Memory:
- M_4[valE] ← valA

Write back:

PC update:
- PC ← valP

Pseudo-code for **mrmovl**

Fetch:
- icode:ifun ← M_1[PC]
- rA:rB ← M_1[PC+1]
- valC ← M_4[PC+2]
- valP ← PC+6

Decode:
- valB ← R[rB]

Execute:
- valE ← valC + (valB*ifun)

Memory:
- valM ← M_4[valE]

Write back:
- R[rA] ← valM

PC update:
- PC ← valP

Description of test coverage **movl.s**

According to the comments in the AbstractY86CPU, Y86 CPU can take multiple arguments for the same purpose as proposed in problem 1. They are: 4, 2, 1 and no argument. So the test is split into four parts to test each of the 4 arguments respectively: testx4, testx2, testx1, and testxReg. The workflow is identical for each part; except for the values differ to what is read into memory/register and the constant values to adjust for the offset.

Each part loads a value in a register %ecx that will be inserted into the first two position of an array using the improved rmmovl instruction. It will then take the new value that is stored in memory and place it into a new register using the mrmovl instruction. The workflow will then repeat itself two more times in different positions on memory.

I believe my movl.s has sufficient coverage of the new implementation of rmmovl and mrmovl. During each of the workflows, it inserts different values into different addresses of memory and loads those values into different registers. The workflow is repeated with different values for each argument supported; verifying that each supported argument works and that the intended function works as each position of the array is being overwritten. The test also covers a case of invalid argument for rmmovl and mrmovl, in section "testxInv". When uncommenting those lines of code, Y86 will fail to load movl.s due to an illegal scale.

Test results for **movl.s**

NOTE: The best way to run the test is to have break points at each of labels.
Label "testxInvalid", test invalid arguments for rmmovl and mrmovl.
Uncommenting line 84 should result an "Assembly error: Illegal scale: must be 1, 2, or 4."
Uncommenting line 85 should result an "Assembly error: Illegal scale: must be 1, 2, or 4."

Pretest:
All values of TheArray should be 7.

Values after running "testx4":

```
.pos 0x1000
TheArray:       .long   40
                .long   40
                .long   41
                .long   41
                .long   42
                .long   42
%eax            40
%esi            41
%edi            42
```

Values after running "testx2":

```
.pos 0x1000
TheArray:       .long   20
                .long   20
                .long   21
                .long   21
                .long   22
                .long   22
%eax            20
%esi            21
%edi            22
```

Values after running "testx1":

```
.pos 0x1000
```

```
TheArray:   .long   10
            .long   10
            .long   11
            .long   11
            .long   12
            .long   12
%eax        10
%esi        11
%edi        12
```

Values after running "testxReg":

```
.pos 0x1000
TheArray:   .long   0
            .long   0
            .long   1
            .long   1
            .long   2
            .long   2
%eax        0
%esi        1
%edi        2
```

Problem 2:

Pseudo-code for **I_IOPL**

```
Fetch:
        icode:ifun      ← M1[PC]
        rA:rB           ← M1[PC+1]
        valC            ← M4[PC+2]
        valP            ← PC+6
Decode:
        valB            ← R[rB]
Execute:
        valE            ← valB [+, -, *, /, and, or, xor] valC
Memory:
Write back:
        R[rB]           ← valE
PC Update:
        PC              ← valP
```

Description of test coverage **iopl.s**

iopl.s is divided into seven sections, making sure each of the seven arithmetic functions are working correctly. I believe that iopl.s provides sufficient test coverage because each section tests the arithmetic operation at least twice, while testing specific odd cases that mimics i_opl's operation (such as dividing by zero). The arithmetic operation is then checked using I_opl's untouched implementation and errors are then handled to determine which arithmetic operation is not working correctly.

Test results for **iopl.s**

Simply run the test and check %eax value at the end.

If %eax's value is:

0 = test passed
1 = iaddl error
2 = isubl error
3 = imul error
4 = idivl error
5 = iandl error
6 = ixorl error
7 = imodl error

Problem 3:

Fetch:

icode:ifun	← M_1[PC]
rA:rB	← M_1[PC+1]
valC	← M_4[PC+2]
valP	← PC+6

Decode:

valB	← R[rB]

Execute:

valE	← valC + (valB*ifun)

Memory:

valM	← M_4[valE]
valE	← valP

Write back:

R[rA]	← valE

PC update:

PC	← valM

Description of test coverage **call.s**

I believe this test is sufficient because, call.s checks to see if both the regression method as well as the new implementation works. By having the successfully coming to the halt statement, this shows that both function works correctly.

Test results for **call.s**

Run call.s

If programs comes to a halt it runs successfully, and if:

%eax = 7
%edx= 0x117
%ebx= 8
%ecx= 4