EE234

Microprocessor Systems

Midterm Exam

Oct. 25, 2019. (3:10pm – 4pm)

Instructor: Dae Hyun Kim (daehyun@eecs.wsu.edu)

Name:

WSU ID:

Problem	Points	
1	10	
2	10	
3	10	
4	10	
5	10	
6	15	
7	15	
Total	80	

Problem #1 (Bit manipulation, 10 points)

Suppose R# is an 8-bit register. The data stored in R# is treated as an unsigned binary number. The following instruction performs an arithmetic operation. Explain what it does (i.e., briefly explain the meaning of the data stored in R2 in terms of arithmetic operations). Here, “arithmetic” means something like addition, subtraction, multiplication, division (quotient), division (remainder), square root, transcendental functions, etc.

AND R2, R1, #0x0F

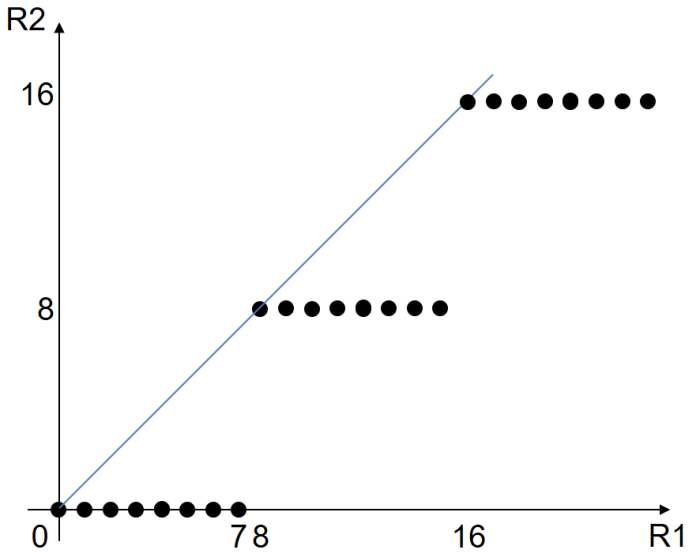
The first four bits of R1 are set to 0, so it performs the modulo-16 operation, i.e., R2 has the remainder of R1 % 16.

Problem #2 (Bit manipulation, 10 points)

Suppose R# is an 8-bit register. The data stored in R# is treated as an unsigned binary number. The following instruction performs an arithmetic operation. Draw a graph for the following instruction. The x-axis should be the value stored in R1 and the y-axis should be the value stored in R2.

AND R2, R1, #0xF8

The three rightmost bits of R1 are set to zero, so it performs $8 \times \left\lfloor \frac{R1}{8} \right\rfloor$ where $\lfloor\ \rfloor$ is the floor function (the largest integer less than or equal to).

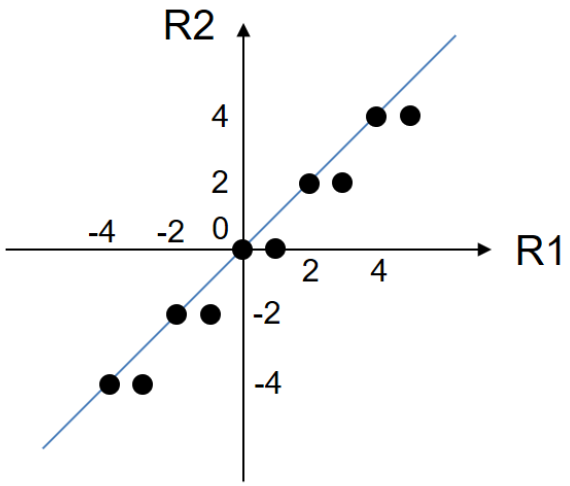


Problem #3 (Bit manipulation, 10 points)

All the data stored in the registers are two's complement binary numbers. Draw a graph for the following instruction. The x-axis should be the value stored in R1 and the y-axis should be the value stored in R2.

```
LSR R2, R1, #1
LSL R2, R2, #1
```

Logically shifting R1 to the right by one bit and then to the left by one bit just sets the LSB to zero. If R1 is a positive number or zero, R2 = R1 if R1 is even and R2 = R1 – 1 if R1 is odd. If R1 is a negative number, R2 = R1 if the LSB of R1 is zero. However, if R1 is a negative number and its LSB is 1, it is also subtracting 1. For example, suppose R1 = 1X1 where X is a string of zeros and ones. The absolute value of this number is $\bar{X} + 1$. If the LSB is set to zero (=1X0), its absolute value becomes $\bar{X} + 2$. Thus, the graph will look like this:



Problem #4 (ARM assembly, 10 points)

Suppose R# is an 8-bit register. 1) What is the value of the data stored in R2 when the program ends? 2) What is the meaning of the code? Explain briefly.

```
    MOV R1, #250
    MOV R2, #0
loop1:
    AND R3, R1, 0x01
    AND R4, R1, 0x02
    LSR R4, #1
    OR R3, R3, R4
    INV R3
    AND R3, #0x01
    ADD R2, R2, R3
    SUB R1, #1
    CMP R1, #0
    BEQ end
    B loop1
end:
    // end of code
```

For given R1, R3 is $0000000x_0$ and R4 is $000000x_10$. Then, R4 is shifted to the right by one bit and ORed with R3, so R3 is $0000000(x_0|x_1)$ where | is logical OR. Then, R3 is inverted, so R3 becomes $1111111\overline{(x_0|x_1)}$. Then, R3 is ANDed with 0x01, so R3 becomes $0000000\overline{(x_0|x_1)}$. As a result, R3 is #1 if $x_0 = 0$ and $x_1 = 0$ (otherwise, R3 is #0). Then, it is added to R2. We repeat this for all the numbers from 250 to 1. Thus, R2 has 62.

The meaning of this code is to count the number of the multiples of 4 in the range of [1, 250].

Problem #5 (ARM assembly, 10 points)

Suppose R# is an 8-bit register. Write an assembly code to count the number of 1's in R1. The result must be stored in R2. For example, suppose R1 = 0110 1010 (given). Then, R2 will be 0000 0100 (#4) after the execution of your code. Use only the instructions shown in the instruction sheet. You can also use subroutines. The performance of the code doesn't matter as long as the code works.

```
    // R1 is given.
    MOV R2, #0
    MOV R3, #8
loop1:
    AND R4, R1, #0x01
    ADD R2, R2, R1
    LSR R1, #1
    SUB R3, #1
    CMP R3, #0
    BNE loop1
    // R2 has # 1's.
```

Problem #6 (ARM assembly, 15 points)

In Code 1, we call subroutine “func2” in the “main” routine. Inside “func2”, we call subroutine “func3” without backing-up the data stored in LR. Thus, when “func3” is done, we can come back to “func2”, but we cannot return back to the main routine from “func2”.

In Code 2, we call subroutine “func2” in the “main” routine. Inside “func2”, we still call subroutine “func3” without backing-up the data stored in LR. Thus, when “func3” is done, we come back to “func2”, then we use the unconditional jump instruction (“B”) to come back to the main routine. Will Code 2 work? Explain why it can work (or why it cannot work).

```
main:
   SUB ...
   BL func2
func1:
   ADD ...
   B end

func2:
   ...
   BL func3
   BX LR

func3:
   ...
   BX LR

end:
   // end of code
```
<Code 1>

```
main:
   SUB ...
   BL func2
func1:
   ADD ...
   B end

func2:
   ...
   BL func3
   B func1

func3:
   ...
   BX LR

end:
   // end of code
```
<Code 2>

Yes, it will work. When we come back from func3 to func2, we don’t have the return address to go back to the main routine (which is the ADD instruction). However, we have the address label “func1” for the ADD instruction, so we can directly jump to the instruction (although this is not recommended in real applications because this does not support subroutine calls from a different subroutine.)

Problem #7 (ARM assembly, 15 points)

Let's use the 32-bit ARM architecture, i.e., R# is a 32-bit register and the register file has 16 registers (and R13 is the stack pointer, R14 is the link register, and R15 is the program counter). R5 has a positive number (given to you). Write an assembly code to find out whether the number in R5 is a prime number or not. If it is a prime number, set R6 to 1. If not, set R6 to 0. Use only the instructions shown in the instruction sheet (but do not use LDR and STR). You can also use subroutines. The performance of the code doesn't matter as long as the code works.

```
    // R5 is given.
    MOV R6, #0
    CMP R5, #1
    BEQ end  // exception handling
    MOV R2, #2
loop1:
    CMP R2, R5
    BEQ prime
    MOV R3, R2
    MOV R7, #0
    BL check
    CMP R7, #0
    BEQ end  // not a prime
    ADD R2, #1
    B loop1
prime:
    MOV R6, #1
    B end
end:
    // end
```

```
check:
    ADD R3, R3, R2
    CMP R3, R5
    BEQ check_return_no_prime
    BLT check
    MOV R7, #1  // not a multiple
    BX LR
check_return_no_prime:
    BX LR
```