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

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

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
```

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
```

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.

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>

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.

Assembly Instructions

R# is a register. (# = 0 ~ 12)

<table>
<tr><th>Instruction</th><th>Meaning</th></tr>
<tr><td>INV Rd</td><td>Bitwise inversion.
<table><tr><td>Before</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>After</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr></table></td></tr>
<tr><td>AND Rd, Ra, Rb
AND Rd, Ra, #imm
AND Rd, #imm</td><td>Bitwise AND. (Rd = Ra AND Rb), (Rd = Ra AND #imm), (Rd = Rd AND #imm)
<table><tr><td>Ra</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr><tr><td>Rb</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>Rd</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr></table></td></tr>
<tr><td>OR Rd, Ra, Rb
OR Rd, Ra, #imm
OR Rd, #imm</td><td>Bitwise OR. (Rd = Ra OR Rb), (Rd = Ra OR #imm), (Rd = Rd OR #imm).
<table><tr><td>Ra</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>Rb</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>Rd</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td></tr></table></td></tr>
<tr><td>EOR Rd, Ra, Rb
EOR Rd, Ra, #imm
EOR Rd, #imm</td><td>Bitwise exclusive-OR. (Rd = Ra ⊕ Rb), (Rd = Ra ⊕ #imm), (Rd = Rd ⊕ #imm)
<table><tr><td>Ra</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>Rb</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>Rd</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr></table></td></tr>
<tr><td>LSR Rd, Ra, #imm
LSR Rd, #imm</td><td>Logical shift right by (#imm) bits. (Rd = Rd >> #imm), (Rd = Rd >> #imm)
Ex) #imm = 3
<table><tr><td>Before</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr><tr><td>After</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table></td></tr>
<tr><td>LSL Rd, Ra, #imm
LSL Rd, #imm</td><td>Logical shift left by (#imm) bits. (Rd = Ra << #imm), (Rd = Rd << #imm)
Ex) #imm = 3
<table><tr><td>Before</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr><tr><td>After</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table></td></tr>
<tr><td>MOV Rd, Ra
MOV Rd, #imm</td><td>(Rd = Ra)
(Rd = #imm)</td></tr>
<tr><td>ADD Rd, Ra, Rb
ADD Rd, Ra, #imm
ADD Rd, #imm</td><td>(Rd = Ra + Rb)
(Rd = Ra + #imm)
(Rd = Rd + #imm)</td></tr>
<tr><td>SUB Rd, Ra, Rb
SUB Rd, Ra, #imm
SUB Rd, #imm</td><td>(Rd = Ra - Rb)
(Rd = Ra - #imm)
(Rd = Rd - #imm)</td></tr>
<tr><td>CMP Rd, #imm
CMP Rd, Ra</td><td>Set Z = 1 if Rd == #imm. Otherwise, Z = 0. (Z is the Zero field of the CPSR.)
Set Z = 1 if Rd == Ra. Otherwise, Z = 0.
Notice that N != V is Rd < #imm or Rd < Ra.</td></tr>
<tr><td>BEQ [addr]</td><td>Branch to [addr] if Z = 1. Ex) CMP R1, R2. BEQ tar → Go to tar if R1 == R2.</td></tr>
<tr><td>BNE [addr]</td><td>Branch to [addr] if Z = 0. Ex) CMP R1, R2. BNE tar → Go to tar if R1 != R2.</td></tr>
<tr><td>BLT [addr]</td><td>Branch to [addr] if N != V. Ex) CMP R1, R2. BLT tar → Go to tar if R1 < R2.</td></tr>
<tr><td>LDR Rd, [Ra, #imm]</td><td>Load the data stored at [Ra + #imm] to Rd.</td></tr>
<tr><td>STR Rd, [Ra, #imm]</td><td>Store the data stored in Rd to [Ra + #imm].</td></tr>
</table>