EE234

Microprocessor Systems

Midterm Exam

Oct. 14, 2020. (2:10pm – 3pm)

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

Name:

WSU ID:

Problem	Points	
1	10	
2	10	
3	10	
4	10	
5	10	
6	10	
7	20	
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) or draw a graph of (R1 vs. R2). Here, “arithmetic” means something like addition, subtraction, multiplication, division (quotient), division (remainder), square root, transcendental functions, etc.

EOR R2, R1, #0x03

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. Explain what it does (i.e., briefly explain the meaning of the data stored in R2 in terms of arithmetic operations) or draw a graph of (R1 vs. R2). Here, “arithmetic” means something like addition, subtraction, multiplication, division (quotient), division (remainder), square root, transcendental functions, etc.

OR R2, R1, #0x80

Problem #3 (Bit manipulation, 10 points)

Suppose R# is an <u>8-bit register</u>. R1 ($x_7\ x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0$) has an input data. We want to generate the following signal from R1.

R2 = ($x_7\ \overline{x_6}\ 1\ 0\ x_3\ \overline{x_2}\ 1\ 0$)

Use the instructions in the instruction sheet to generate R2. Try to minimize # instructions you use. (≤3 instructions: 10 points, 4 instructions: 5 points, ≥5 instructions: 2 points)

Problem #4 (Bit manipulation, 10 points)

Suppose R# is an 8-bit register. R1 ($x_7\ x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0$) has an input data. We want to generate the following signal from R1.

$$\text{R2} = (x_3\ x_2\ x_1\ x_0\ x_7\ x_6\ x_5\ x_4)$$

Use the instructions in the instruction sheet to generate R2. Use only R1 and R2 (i.e., don't use any other registers.) You don't need to preserve the input data. Try to minimize # instructions you use. (≤3 instructions: 10 points, 4 instructions: 5 points, ≥5 instructions: 2 points)

Problem #5 (ARM assembly, 10 points)

What is the value of the data stored in R2 when the program ends?

```
  MOV R1, #0
  MOV R2, #0
loop1:
  ADD R1, R1, #1
  MOV R3, R1
  AND R4, R3, #0x01
  LSR R3, R3, #1
  AND R4, R4, R3
  LSR R3, R3, #1
  AND R4, R4, R3
  ADD R2, R2, R4
  CMP R1, #255
  BLT loop1
end:
    // end of code
```

Problem #6 (ARM assembly, 10 points)

What is the value of the data stored in R2 when the program ends?

```
  MOV R1, #0
  MOV R2, #1
  MOV R3, #2
loop1:
  ADD R4, R1, R2
  ADD R3, R3, #1
  MOV R1, R2
  MOV R2, R4
  CMP R3, #10
  BNE loop1
end:
   // end of code
```

Problem #7 (ARM assembly, 20 points)

Let's use the 32-bit ARM architecture, i.e., R# is a 32-bit register and the register file has 16 registers (you can use R0~R12 only). R0 has a positive number (given to you). We want to check whether the number in R0 is an integer multiple of 3 (i.e., $3n$) or not. If it is, we set R1 to 1. If not, we set R1 to 0. Here is an algorithm for that.

1) If R0 is 0, R1 = 1. Done.

2) If R0 is 1, R1 = 0. Done.

3) If R0 is 2, R1 = 0. Done.

4) If not (i.e., R0 $\geq$ 3), subtract 3 from R0 (i.e., R0 = R0 – 3).

5) Go back to 1).

Write an assembly code running the above algorithm. Use only the instructions shown in the instruction sheet (but do not use LDR and STR). 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>