EE234

Microprocessor Systems

Midterm Exam 1

Oct. 8, 2021. (2:10pm – 3pm)

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

Name:

WSU ID:

Problem	Points	
1	10	
2	10	
3	20	
4	20	
5	30	
6	30	
Total	120	

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. R1 has an input data. The following two instructions perform 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.

```
AND R2, R1, #0xFD

ORR R2, R2, #0x01
```

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. R1 has an input data. 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.

AND R2, R1, #0xBF

Problem #3 (ARM assembly, 20 points)

What is the value of the data stored in R1 when the following program ends?

```
  MOV R1, #0
  MOV R2, #0
loop1:
  CMP R2, #200
  BGE end
  AND R3, R2, #0x07
  CMP R3, #2
  BNE loop1_end
  ADD R1, R1, #1
loop1_end:
  ADD R2, R2, #1
  B loop1
end:
  // end of code
```

Problem #4 (ARM assembly, 20 points)

What is the value of the data stored in R3 when the program ends?

```
  MOV R3, #0
  MOV R1, #0
loop1:
  CMP R1, #5
  BGE loop1_end
  MOV R2, #0
loop2:
  CMP R2, #5
  BGE loop2_end
  AND R4, R1, R2
  ADD R3, R3, R4
  ADD R2, R2, #1
  B loop2
loop2_end:
  ADD R1, R1, #1
  B loop1
loop1_end:
// end of code
```

Problem #5 (ARM assembly, 30 points)

Make an assembly code for the following C code.

```
int a, b, c;  // a in R0, b in R1, c in R2

if ( (a == 0) && (b == 3) ) {
  c++;
}
else if ( (b == 2) || (c == 4) ) {
  a++;
}
else if ( (a == 5) && (c != 6) ) {
  b++;
}
else {
  a--;
}
```

- Use the assembly instructions listed in the last page only.
- a is in R0, b is in R1, and c is in R2.
- The exit point (the end of the if statement) could be just an address label.

Problem #6 (ARM assembly, 30 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 a square number (i.e., n^2) 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 = 0. Done.

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

3) If R0≥ 2, try to compare 1^2 with R0, 2^2 with R0, ..., n^2 with R0. If n^2==R0, R1 = 1. Done. If $(n-1)^2$<R0 and n^2>R0, R1 = 0. Done.

Simply speaking, suppose 35 is given (in R0). Then, $1^2 = 1 < 35, 2^2 = 4 < 35, 3^2 = 9 < 35, 4^2 = 16 < 35, 5^2 = 25 < 35, 6^2 = 36 > 35$, so we know that 35 is not a square number.

Write an assembly code running the above algorithm. Use only the instructions shown in the instruction sheet. Assume that R0 has a given number. The performance of the code doesn't matter as long as the code works. You can't use multiply instructions, so you should use ADD to compute n^2.

Assembly Instructions

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

<table>
<tr><th>Instruction</th><th>Meaning</th></tr>
<tr><td>MVN Rd, Ra</td><td>Bitwise inversion. (Rd = Bitwise-NOT Ra)
<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</td><td>Bitwise AND. (Rd = Ra AND Rb), (Rd = Ra 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>ORR Rd, Ra, Rb
ORR Rd, Ra, #imm</td><td>Bitwise OR. (Rd = Ra OR Rb), (Rd = Ra 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</td><td>Bitwise exclusive-OR. (Rd = Ra ⊕ Rb), (Rd = Ra ⊕ #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>MOV Rd, Ra, LSR #imm</td><td>Logical shift right by (#imm) bits. (Rd = Ra >> #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>MOV Rd, Ra, LSL #imm</td><td>Logical shift left by (#imm) bits. (Rd = Ra << #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</td><td>(Rd = Ra + Rb)
(Rd = Ra + #imm)</td></tr>
<tr><td>SUB Rd, Ra, Rb
SUB Rd, Ra, #imm</td><td>(Rd = Ra - Rb)
(Rd = Ra - #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>B [addr]</td><td>Jump to [addr] unconditionally</td></tr>
<tr><td>BEQ, BNE, BLT, BGT, BGE, BLE [addr]</td><td>Branch to [addr] if (BEQ: R1 == R2), (BNE: R1 != R2), (BLT: R1 < R2), (BGT: R1 > R2), (BGE: R1 >= R2), (BLE: 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>