3666 hw 1

Casey Provitera

January 2024

Answers will be in blue

1

For each instruction in the table, write 8 hexadecimal digits that represent the 32 bits in the destination register after the instruction is executed. Assume s0 is 0x98ABCD6A, s1 is 0x20FF5A98.
Find out the answers by working on bits/hexadecimal digits. Do not convert large numbers from hexadecimal to decimal.
The submission should include the steps you take to find the answers. Check your answers in RARS.

Convert Hex values to binary -
s0 = 0x98ABCD6A -> 0b 1001 1000 1010 1011 1100 1101 0110 1010
s1 = 0x20FF5A98 -> 0b 0010 0000 1111 1111 0101 1010 1001 1000

add t0, s0, s1 =
0b 1001 1000 1010 1011 1100 1101 0110 1010
0b 0010 0000 1111 1111 0101 1010 1001 1000
=
0b 1011 1001 1010 1011 0010 1000 0000 0010 convert to Hex -> 0xB9AB2802

and t1, s0, s1 =
0b 1001 1000 1010 1011 1100 1101 0110 1010
0b 0010 0000 1111 1111 0101 1010 1001 1000
=
0b 0000 0000 1010 1011 0100 1000 0000 1000 convert to Hex -> 0x00A4808

or t2, s0, s1 =
0b 1001 1000 1010 1011 1100 1101 0110 1010
0b 0010 0000 1111 1111 0101 1010 1001 1000
=
0b 1011 1000 1111 1111 1101 1111 1111 1010 convert to Hex -> 0xB8FFDFFA

xor t3, s0, s1 =
0b 1001 1000 1010 1011 1100 1101 0110 1010
0b 0010 0000 1111 1111 0101 1010 1001 1000
=
0b 1011 1000 0101 0100 1001 0111 1111 0010 convert to Hex -> 0xB85497F2

addi t4, s0, 0x2FA =
0b 1001 1000 1010 1011 1100 1101 0110 1010
0b 0000 0000 0000 0000 0000 0010 1111 1010
=
0b 1001 1000 1010 1011 1101 0000 0110 0100 convert to Hex -> 0x98ABD064

andi t5, s0, -16 =
16 in binary is 0b 0000 0000 0000 0000 0000 0000 0001 0000 flip bits to get
0b 1111 1111 1111 1111 1111 1111 1110 1111 add 0b1 to get
-16 = 0b 1111 1111 1111 1111 1111 1111 1111 0000
andi t5, s0, -16 =
0b 1001 1000 1010 1011 1100 1101 0110 1010
0b 1111 1111 1111 1111 1111 1111 1111 0000
=
0b 1001 1000 1010 1011 1100 1101 0110 0000 convert to Hex -> 0x98ABCD60

slli t6, s0, 12 =
0b 1001 1000 1010 1011 1100 1101 0110 1010
shift 12 bits left (fill with 0)=
0b 1011 1100 1101 0110 1010 0000 0000 0000 convert to Hex -> 0xBCD6A000

srai s2, s0, 8 =
0b 1001 1000 1010 1011 1100 1101 0110 1010
shift right 8 bits (fill with sign bit) =
0b 1111 1111 1001 1000 1010 1011 1100 1101 convert to Hex -> 0xFF98ABCD

Instructions	Dest. reg. in 8 hexadecimal digits
add t0, s0, s1	0xB9AB2802
and t1, s0, s1	0x00A4808
or t2, s0, s1	0xB8FFDFFA
xor t3, s0, s1	0xB85497F2
addi t4, s0, 0x2FA	0x98ABD064
andi t5, s0, -16	0x98ABCD60
slli t6, s0, 12	0xBCD6A000
srai s2, s0, 8	0xFF98ABCD

2

Write RISC-V instructions to reverse the order of bytes in register s2 and save the results in s4. For example, if s2 is 0x12345678, the four bytes in s2 are 0x12, 0x34, 0x56, and 0x78. Register s4 should be 0x78563412 after the execution of the instruction sequence. Use temporary registers like t0 and t1 to save intermediate values.
Explain your strategy and test your code in RARS.
Always write brief comments with your code. See the examples in slides and the code given in the next questions.
Use shifts to move bits over, and "ands" to get specific bits. Make t2 = s2. To get the last 8 bits (which is the same as 2 bytes) of a 32 bit (s2) value execute "and t3, s2, 0b 1111 1111", this will only give the last 8 bites of the 32 bit value. Put the last 8 bits of t2 which are now stored in t3 into t4, shift t2 right 8 bits, then shift t4 left 8 bits. repeat until t2 is 0. When t2 is 0 make s4 = t4.

```
# loading s2 as 0c12345678
    lui s2, 0x12345
    addi s2, s2, 0x678
# printing s2 in hex format
    addi a7, x0, 34
    add a0, s2, x0
    ecall
loop:
    slli t2, t2, 8                  # shift t2 left 8 bits
    andi t3, s2, 0xFF               # getting the last 8 bits of s2
    add t2, t2, t3                  # add last 8 bits of s2 to t2
    srli s2, s2, 8                  # shift s2 right 8 bits
    andi t3, s2, 0xFF               # getting the last 8 bits of s2
    bne t3, x0, loop                # continue loop if t3 is not 0
    add s4, t2, x0                  # storing t2 in s4
# print newline
    addi a0, x0, 10
    addi a7, x0, 11
    ecall
```

```
20. # printing s4 in hex format
21.     addi a7, x0, 34
22.     add a0, s4, x0
23.     ecall
```

3

The following RISC-V instructions calculate the Hamming weight (the number of 1's) of s0. The result is saved in register s1.

```
1.      addi s1, x0, 0                                  s1 = 0
2.      addi t0, x0, 1          Use t0 as mask to test each bit in s0
3. loop:
4.      and t1, s0, t0                      extract a bit with the mask
5.      beq t1, x0, skip           if the bit is 0, do not increment s1
6.      addi s1, s1, 1                           increment the counter
7. skip:
8.      slli t0, t0, 1                          shift mask to left by 1
9.      bne t0, x0, loop                 if the mask is not 0, continue
```

(a) If s0 is 0xFF00FF00, how many instructions are executed? Does the number of executed instructions depend on the number of 1's in s0? Does it depend on the location of 1's? Explain your answers.
0xFF00FF00 to binary
= 0b 1111 1111 0000 0000 1111 1111 0000 0000
number of instructions is equal to =
2 (initial instructions before loop)
+ 4*(number of digits in the given value)
+ 1*(number of 1s) each time line 5 is true 6 runs
for 0b 1111 1111 0000 0000 1111 1111 0000 0000 =
2+4*32+1*16 = 2+128+16=146
The number of executions depends on the number of 1s in s0 but it does not depend on their location. For every 1 in s0 line 6 is run 1 time. But there is no other line of code that depends on where the 1s are located in s0.

(b) There are many ways to compute Hamming weight. We could test the most significant bit (bit 31) of s0. For example, extract bit 31 with an AND instruction, and compare it with 0. This is similar to the method in the code given. However, we can save one instruction. If we treat s0 as a 2's complement number, s0 is less than 0 if and only if bit 31 in s0 is 1. Using this method, write RISC-V instructions to compute the Hamming weight of s0. Explain your method in comments. We can start with the following two instructions. How many instructions are executed if s0 is 0xFF00FF00? Explain how you get the answers.

```
        addi s1, x0, 0,                                   # s1 = 0
        add t0, x0, s0,     # t0 is copy of s0 # this call is used to see if s0 is negative
        lui t1, 0x8     # same as 0b 1000 000 0000 0000 0000 0000 0000 0000
        and t2, t0, t1                      # t1 now tells if s0 is negative or not
        beq t2, t1, negative
        beq x0, x0, loop
negative: # this code works like finding the reciprocal number of a negative
        xori t2, t2, -1                      # this works like running not on t2
        addi t2, t2, 1
# same code from part a
loop:
        and t1, s0, t0
        beq t1, x0, skip
        addi s1, s1, 1
skip
        slli t0, t0, 1
        bne t0, x0, loop
```

This method flips the bits and adds 1 to a number if it was negative which in some cases can be quicker than the method from part a.
If s0 is 0xFF00FF00 we first flip the bits to get 0x00FF00FF, then we add 1 to get 0x00FF0100 in binary = 0b 0000 0000 1111 1111 0000 0001 0000 0000. In our method in part be this takes
7 initial instructions
4*(number of digits in the value)
1*(number of 1s)
for 0b 0000 0000 1111 1111 0000 0001 0000 0000 =
7+4*32+1*9 = 144

4

Translate the following C code to RISC-V assembly code. Assume that a, i, and r are stored in registers s1, s2, and s3, respectively, and their values are already set in these registers. All the variables are signed. Write brief comments in your code. Clearly mark the instructions that control the loop (for example, using different colors), the instructions in if branch, and instructions in else branch. Use at most 12 instructions.

```
1. for (i = 0; i < a; i+ = 1)
2.     if ((i & 0xA5) != 0)
3.         r ^= i << 8;
4.     else
5.         r += i >> 4;
```

Bold Underlined texts shows what commands control the loops and if statements

```
1. loop:
2.     andi t4, s2, 0xA5       # and of s2 and hex A5
3.     bne t4, x0, if          # if the and is not 0 go to if
4. else:
5.     srai t3, s2, 4          # store s2 shifted right arithmetically 4 bits in t3
6.     add s3, s3, t3          # add t3 to s3
7.     addi s2, s2, 1          # add 1 to s2
8.     blt s2, s1, loop        # if s2 less than s1 go to loop
9.     bne x0, x0 exit         # used when loop is done to skip if block
```

```
if:
    slli t3, s2, 8          # store s2 shifted left logically 8 bits in t3
    xor s3, s2, t3                       # xor of s2 and t3 stored in s3
    addi s2, s2, 1                                         # add 1 to s2
    blt s2, s1, loop                         # if s2 less than s1 go to loop
exit:
```

5

Read the following Wikipedia page about Collatz conjecture. The link is clickable.
`https://en.wikipedia.org/wiki/Collatz_conjecture`
Write a RISC-V program in RARS. The program reads a positive integer n and prints out the total stopping time of n, i.e., the number of times we need to apply function f on n to reach 1. Function f is defined on the Wikipedia page. For example, if n is 1, the program outputs 0. If n is 9, the program outputs 19. There is no newline character after the number. You can find more expected results on the Wikipedia page.
Note that we cannot use MUL and DIV instructions, which are not in RV32I

store n in s0
need a counter for number of steps, use s1
everytime when number is even we shift right one bit
when number is odd we add it to itself 3 times and add 1
example of multiple by 3 and add one s0 is 7
add t0, s0, s0 = t0 is 14
add t0, t0, s0 = t0 is 21
addi t0, t0, 1 = t0 is 22
to check if odd or even check last bit if last bit is 0 number is even if last bit is 1 number is odd perform andi with original number and 1 if return is 1 number is odd if return is 0 number is even

code :

```
addi s1, x0, 1 # storing 1 inside of s1
add s2, x0, x0 # storing 0 in s2, s2 will be used as a counter
# getting number to run function on from user
addi a7, x0, 5
ecall
```

```
6. add s0, a0, x0
7. add t0, s0, x0 # storing value in t0
8. beq t0, s1, end
9. # t1 is only ever used for andi and branch statments
10. # t0 is the actual number that is being changed
11. andi t1, t0, 1 # getting the and of the value and 1 to see if it's even or
    odd
12. beq t1, x0, even # chceking even
13. beq t1, s1, odd # checking odd
14. even:
15. addi s2, s2, 1 # increase counter
16. srai t0, t0, 1 # shifting t0 1 bit to the right
17. beq t0, s1, end # checking if t0 is 1
18. andi t1, t0, 1 # getting the and of the value and 1 to see if it's even or
    odd
19. beq t1, x0, even
20. beq t1, s1, odd
21. odd:
22. addi s2, s2, 1 # increase counter
23. add t3, t0, t0 # t3 now equal 2*t0
24. add t3, t3, t0 # t0 now equals 3*t0
25. addi t3, t3, 1 # t0 now equal s 3*t0+1
26. add t0, t3, x0 # moving value back to t0
27. andi t1, t0, 1 # getting the and of the value and 1 to see if it's even or
    odd
28. beq t1, x0, even
29. beq t1, s1, odd
30. end:
31. add a0, s2, x0 # storing counter value in a0 to be printed
```

32. addi a7, x0, 1 # putting system call 1(print integer) in a7
33. ecall # printing a0
34. addi a7, x0, 10 # sysetm call to end program
35. ecall