3666 hw 2

Casey Provitera

February 2024

ANSWERS IN BLUE

1

Suppose A and B are word arrays. The following C loop increments elements in A by 4 and saves the results into B.
for (i = 0; i < 100; i += 1)
B[i] = A[i] + 4;
The following table shows the mapping between variables and registers.

Register	s1	s2	s3
Variable/value	i	Address of A	Address of B

We will study two implementations in RISC-V.

(a) The first implementation is based on the array copy code we discussed in lecture. We just need to revise it slightly. What changes do we need? How many instructions will be executed for the loop? Note that we do not need to jump to the condition test before the first iteration because we are sure the condition is true at the beginning.

i. main:
ii. add s1, x0, x0 # counter for loop
iii. lui s2, 0x10010 # starting address of A
iv. addi s3, s2, 512 # starting address of B
v. addi t1, x0, 100 # limit of loop
key: s1=i, s2=A address, s3=B address, t2=offset, t3=base+offset
key: t4=where we store loaded value
vi. loop:
calculation
vii. slli t2, s1, 2 # t2=i*4
viii. add t3, s2, t2 # adding offset to A address
ix. lw t4, 0(t3) # t4 loaded value from A

x. addi t4, t4, 4 # t4+=4
xi. add t3, s3, t2 # adding offset to B address
xii. sw t4, 0(t3) # t4 stored in B
xiii. # looping
xiv. addi s1, s1, 1 # incitement i
xv. blt s1, t1, loop

804 instructions are executed by the above code.

(b) Loop unrolling is an optimization technique to improve the performance of programs. In the second implementation, we unroll the loop and process four array elements in A in each iteration. The unrolled loop in C is shown below. Translate the loop to RISC-V instructions. Try to minimize the number of instructions that are executed. Explain your code. How many instructions will be executed for the new loop?
for (i = 0; i < 100; i += 4) {
B[i] = A[i] + 4;
B[i+1] = A[i+1] + 4;
B[i+2] = A[i+2] + 4;
B[i+3] = A[i+3] + 4;}

i. main:
ii. add s1, x0, x0 # counter for loop
iii. lui s2, 0x10010 # starting address of A
iv. addi s3, s2, 512 # starting address of B
v. addi t1, x0, 100 # limit of loop
key: s1=i, s2=A address, s3=B address, t2=offset, t3=base+offset
key: t4=where we store loaded value
vi. loop:
vii. # calculation
viii. slli t2, s1, 2# t2=i*4
ix. add t3, s2, t2 # adding offset to A address
x. lw t4, 0(t3) # t4 loaded value from A
xi. addi t4, t4, 4 # t4+=4
xii. add t5, s3, t2 # adding offset to B address
xiii. sw t4, 0(t3) # t4 stored in B
unpacking uses the found values of t3 and t5 to store values quicker
by increasing the actual offset value
xiv. lw t4, 4(t3)
xv. addi t4, t4, 4
xvi. sw t4, 4(t5)
xvii. lw t4, 8(t3)

```
xviii.      addi t4, t4, 4
 xix.       sw t4, 8(t5)
  xx.       lw t4, 12(t3)
 xxi.       addi t4, t4, 4
xxii.       sw t4, 12(t5)
      #looping
xxiii.      addi s1, s1, 4 # increment i
xxiv.       blt s1, t1, loop
      429 instructions are executed by the above code.
```

2

A two-dimensional array in C (and some other languages) can be considered as an array of one-dimensional array. For example, the following define T as an 16x8 array in C.
int T[16][8];
The two-dimensional array can be considered as an array of 16 elements, each of which is a one-dimensional array of 8 integers/words. In total there are 128 words. The words are stored in memory in the following order:

T[0][0], T[0][1], . . . , T[0][6], T[0][7],
T[1][0], T[1][1], . . . , T[1][6], T[1][7],
. . .
T[14][0], T[14][1], . . . , T[14][6], T[14][7],
T[15][0], T[15][1], . . . , T[15][6], T[15][7]

Row 0, consisting of T[0][0], T[0][1],. . . , and T[0][7], goes first. Row i is stored right after row i – 1, for i = 1, 2, . . . , 15. For example, T[1][0] is stored right after T[0][7]. If T[0][0] is located at address 1000, T[0][7] is located at address 1028 = 1000 + 7 * 4. And T[1][0] is located at address 1032. Similarly, we can calculate that T[2][0] is located at 1064, T[3][0] is located at 1096, and so on.
Translate the following C code to RISC-V instructions. Assume T's address is already in s9. As a practice of accessing two-dimensional arrays, do not use pointers. Explain your code, especially how you implement the loops and how you calculate T[i][j]'s address.
for (i = 0; i < 16; i += 1) {
for (j = 0; j < 8; j += 1)
T[i][j] = 256 * i + j;}
}
My code works by having a few values that go into calculating the offset. First we have j stored in t2 and secondly i stored in t1, the offset is a combination of i and j. Specifically it is 32*i+4*j. The reasoning for 4*j is simple, each value needs to be 4 bytes also known as a word apart. The reasoning for 32*i is a little more complicated, i is the index of which 1D array we are visiting in our

2D array. Since we are storing a 2D array as a 1D array and each array in the 2D array has 8 values each 1D array in out 2D array is 32 bytes apart. Each time we increment i we are going to the next 1D array location. I will use an example to validate my reasoning.
EX. Imagine a 2D array where each 1D array inside it has 4 integer values, that would mean each 1D array needs 16 bytes of space to store. When you jump from the first 1D array to the second you reset the index of the 1D array to be 0 since you want to work with the first element in the new array. If you did not increase the offset some other way you would end up writing over the first 1D array. In this example we would need a value to be 16*(the 1D array number we are in right now) so we do not write over previous arrays.

1. main:
2. addi t1, x0, -1 # this will be the counter for the outer loop
   # start at -1 so we can increment every time we enter outloop
3. addi t3, x0, 16 # this is the limit for the outer loop
4. add t2, x0, x0 # this will be the counter for the inner loop
5. addi t4, x0, 8 # this is the limit for the inner loop
6. lui s9 0x10000 # just so s9 has some value
   # key t1=i, t2=j, t3=16, t4=8, t5=temp sum, t6=offset, s1=32*j, s2=4i, s3 = 32j+4i
7. outloop:
8. add t2, x0, x0 # resetting the value of t2 to 0
9. addi t1, t1, 1 # increment value of t1
10. blt t1, t3, inloop # if i¡16 do the calculation
11. beq x0, x0, exit # else exit program
12. inloop:
    # actual calculation start here
13. slli t5, t1, 8 # 256*i is the same as slli i, i, 8
14. add t5, t5, t2 # t5 is now 256 * i + j
    # now we need to figure out where to store t5 in memory
    # if we think about the 2D array as being smushed into a 1D array then we just
    # need to adjust offset but the length of a word for every iteration
15. slli s1, t1, 5 # s1 = 32*i
16. slli s2, t2, 2 # s2 = 4*j

```
17.     add s3, s1, s2 # s3 =31j+4i
    # s9 holds the address of T increase by 4 bytes or a single word every time we store a value
    # when we get to a new array in the 2d array i increases which adds 32 to the offset
    # this is because j was just 8 but it went back to 0 but we need to keep moving up in memory
18.     add t6, s3, s9
19.     sw t5 0(t6)
    # looping mechanics
20.     addi t2, t2, 1 # increment j
21.     beq t2, t4, outloop # if j=8 go to outer loop
22.     beq x0, x0, inloop # else go to inner loop
23. exit:
```

3

Decimal strings. Write RISC-V code to add two (non-negative) numbers whose decimal representations are stored in strings. Skeleton code is provided. The program reads two numbers from the console (stdin) and prints their sum. The inputs to the program are two decimal numbers of the same length. The numbers have at least one but less than 100 decimal digits. There is no sign. We also keep the same number of digits in the sum, which means the carry generated from the highest place is discarded. Examples of input and output are shown below. In the submitted PDF file, include the code you write (not the skeleton code provided) and explain how you do addition of digits stored in memory as characters.

```
13240825609812370976 16
38764545336936939732 40
52005370946749310708 56
35621077363125825162106806220138723213 3
84620726513202257031423344081443621939 2
20241803876328082193530150301582345152 5
```

CODE:

```
1. add s4, x0, x0 # setting s4 to 0 to be used as a counter
2. strlen:
3. slli t2, s4, 0 # t2 is s4*1
```

4. add t2, t2, s1
5. lb t1, 0(t2)
6. addi s4, s4, 1 # increasing count of str1 len
   # if t1 is less than 0 it is not a numeric character so we should stop doing calculations
7. blt t1, a4, leave
8. beq x0, x0, strlen
9. leave:
   # remove 2 from s4 makes the random characters at the end of the final destination disappear
10. addi s4, s4, -2
11. # key: len of strings=s4, t1=str1 element, t2=offset, t3=str2 element, t6=9(limit of addition)
12. addi t6, x0, 9 # limit of addition before carry is needed
13. addi t4, x0, 0 # carry variable will be used when t1+t1>9
14. calc:
    # Note that we assume str1, str2, and dst have the same number of decimal digits.
15. slli t2, s4, 0 # shift for offset 1 bit
16. add t2, t2, s1 # str1 address
17. lb t1, 0(t2) # str1 element loaded
18. slli t2, s4, 0
19. add t2, t2, s2 # srt2 address
20. lb t3, 0(t2) # str2 element loaded
21. sub t1, t1, a4 # numerical value of t1 now converted
22. sub t3, t3, a4 # numerical value of t3 now converted
23. add t5, t1, t3, # t1 and t3 added
24. add t5, t5, t4 # t5 now including carry variable
25. addi t4, x0, 0 # reset carry variable to 0
26. bgt t5, t6, carry # if the sum of the two elements if more than 9 we need to keep track of a carry

27. beq x0, x0, skip # skip carry part if sum is not greater than 9
28. carry:
    # this spot will be used to lower t5 and keep track of a carry
29. sub t5, t5, a5 # lowers t5 by 10
30. addi t4, x0, 1 # make carry variable 1
    # We then write a loop to add str1 and str2, and save the result in dst also known as s3.
31. skip:
32. add t5, t5, a4 changing t5 back into a char
33. slli t2, s4, 0 shifting offset
34. add t2, t2, s3 final address stored in t2/offset
35. sb t5, 0(t2) storing the now char value back into s3
36. addi s4, s4, -1 decramenting s4 by 1
37. bge s4, x0, calc
    # Remember that dst should have a terminating NULL.
    # this is taken care of automatically since after we find the length of the input strings we remove 2
    # find length of output
38. add s4, x0, x0
39. outputlen:
40. slli t2, s4, 0 # t2 is s4*1
41. add t2, t2, s3
42. lb t1, 0(t2)
43. addi s4, s4, 1 # increasing count of str1 len
    # if t1 is less than 0 it is not a numeric character so we should stop doing calculations
44. blt t1, a4, leavefin
45. beq x0, x0, outputlen
46. leavefin:

First we need to load the variables from memory. Then since the numbers are stored as strings we need to subtract the ASCII character of '0' this converts the strings to integers for us to use addition on. We are loading the values from both str1 and str2 at the same time to add simply with an add call. We then check if the value is above 9 if so we subtract 10 from it and add 1 to a carry variable to be used in the next addition. Right beofre we store the value back in memory we convert it back to a string since the memory needed for a string is much smaller than an integer. Finally we store the value back into memory.

4

Encoding. For each RISC-V instruction, find out its encoding format, the bits in each field, and the machine code as 8 hexadecimal digits. Pay attention to the number of bits in each field.

or s1, s2, s3
or has the format of R - funct7, rs2, rs1, funct3, rd, opcode
rd - s1 - x9 - 01001 | rs1 - s2 - x18 - 10010 | rs2 - s3 - x19 - 10011
funct7 - 0000000| funct3 - 110 | opcode - 0110011
final in binary - 0b 0000 0001 0011 1001 0110 0100 1011 0011
final in hex - 0x013964b3

slli t1, t2, 16
slli has the format of I - imm[11:0], rs1, funct3, rd, opcode
rd - t1 - x6 - 00110 | rs1 - t2 - x7 - 00111 | imm - 16 - 0000 0001 0000
funct3 - 001 | opcode - 0010011
final in binary - 0b 0000 0001 0000 0011 1001 0011 0001 0011
final in hex - 0x01039313

xori x1, x1, -1
xori has the format of I - imm[11:0], rs1, funct3, rd, opcode
rd - x1 - 00001 | rs1 - x1 - 00001 | imm - (-1) - 1111 1111 1111
funct3 - 100 | opcode - 0010011
final in binary - 0b 1111 1111 1111 0000 1100 0000 1001 0011
final in hex 0xfff0c093

lw x2, -100(x3)
lw has the format of I - imm[11:0], rs1, funct3, rd, opcode
rd - x2 - 00010 | rs1 - x3 - 00011
imm - offset - (-100) - 2s compliment is 1100100 - flip bits 0011011 - add 1 0011100 - add leading ones 111110011100
funct3 - 010 | opcode - 0000011
final in binary - 0b 1111 1001 1100 0001 1010 0001 0000 0011
final in hex - 0xf9c1a103

5

Decoding. Each 8-digit hexadecimal number in the following table represents a RISC-V instruction. For each machine code, find its encoding format, bits in each field, and then decode it into a RISC-V instruction. Use register numbers (like x0 instead of zero). Any immediate or displacement (offset) should be in decimal.

	Machine Code
A	0xfeaca823
B	0x04020713
C	0x00557bb3
D	0x414fdf13

A in binary - 0b 1111 1110 1010 1100 1010 1000 0010 0011
Opcode is always last 7 bits = 0100011 this means format of instruction is S
Formatted into being separated according to chart for type of instruction:
imm[11:5] - 1111111 -
rs2 - 01010 - x10
rs1 - 11001 - x25
funct3 - 010 denotes the call as sw
imm[4:0] - 10000
full imm - 111111110000 - 2s counter part = 10000 - (-16)
opcode - 0100011
call equates to - sw x10, -16(x25)

B in binary - 0b 0000 0100 0000 0010 0000 0111 0001 0011
Opcode is always last 7 bits = 001 0011 this means format of instruction is I
Formatted into being separated according to chart for type of instruction:
imm[11:0] - 000001000000 - 64
rs1 - 00100 - x4
funct3 - 000 - denotes the call as addi
rd - 01110 - x14
opcode - 0010011
call equates to - addi x14, x4, 64

C in binary - 0b 0000 0000 0101 0101 0111 1011 1011 0011
Opcode is always last 7 bits = 011 0011 this means format of instruction is R
Formatted into being separated according to chart for type of instruction:
funct7 - 0000000
rs2 - 00101 - x5
rs1 - 01010 - x10
funct3 - 111 - denotes the call as and

rd - 10111 - x23
opcode - 0110011
call equates to - and x23, x10, x5

D in binary - 0b 0100 0001 0100 1111 1101 1111 0001 0011
Opcode is always last 7 bits = 001 0011 this means format of instruction is I
Formatted into being separated according to chart for type of instruction:
imm[11:0] - 010000010100 - 4+16+1024=1044
rs1 - 11111 - x31
funct3 - 101 - denotes the call as srai
rd - 11110 - x30
opcode - 0010011
call equates to - srai x30, x31, 1044