Bryan Martinez

CSE 3666

10 February 2024

1)

a)

```
1          .globl  main
2
3          .text
4
5   main:  addi    s4, x0, 100
6          addi    s1, x0, 0
7
8   loop:  slli    t0, s1, 2       # t0 = i * 4 - convert to word size
9          add     t2, t0, s2      # compute address of A[i]
10         lw      t1, 0(t2)       # loads the cotents at A[i] to t1
11         addi    t1, t1, 4       # increments the contents at A[i] by 4
12         add     t3, t0, s3      # Comptues address of B[i]
13         sw      t1, 0(t3)       # saves the contents of A[i] to B[i]
14         addi    s1, s1, 1       # incremeant i counter
15
16  test:  bne     s1, s4, loop
```

The change that needed to be made was the addition of A[i] with 4. This was done on line 11, where the contents at the address of A[i] are added by 4 and then saved to B[i]. As there are 8 instructions that are ran with every iteration, and there will be 100 iterations, that will be 800 instructions, plus the first two instructions that run in the beginning of the program. In total, there will be 802 instructions executed in total.

b)

```
       .globl  main

       .text

main:  addi    s4, x0, 100
       addi    s1, x0, 0

loop:  slli    t0, s1, 2       # t0 = i * 4 - convert to word size
       add     t2, t0, s2      # compute address of A[i]
       lw      t1, 0(t2)       # loads the cotents at A[i] to t1
       addi    t1, t1, 4       # increments the contents at A[i] by 4
       add     t3, t0, s3      # Comptues address of B[i]
       sw      t1, 0(t3)       # saves the contents of A[i] to B[i]

       lw      t1, 4(t2)       # A[i] already saved to t2, so A[i+1] is offset of 4
       addi    t1, t1, 4       # add 4 to contents
       sw      t1, 4(t3)       # repeat same procedure with B[i+1]

       lw      t1, 8(t2)       # repeat the pattern
       addi    t1, t1, 4
       sw      t1, 8(t3)

       lw      t1, 12(t2)      # repeat the pattern
       addi    t1, t1, 4
```

```
        sw      t1, 8(t3)

        addi    s1, s1, 4       # incremeamnt i counter

test:   bne     s1, s4, loop
```

To load and store the contents at A[i] and B[i] are the same as 1a, but when doing the next index of A[i+1], an offset of 4 is used when loading words and saving words. So, for B[i+1] and A[i+1], an offset of 4 is used. Then, A[i+2] and B[i+2] use an offset of 8. Thus, to find the offset is done by multiplying the number being added to i by 4. As such, 17 instructions are executed for each iteration, and there will be 25 iterations. Thus, it will be 425 plus the two instructions that run at the beginning of the program. In total, 427 instructions are executed.

2)

```
main:

        addi    s0, x0, 0       # i counter set to 0
        addi    s1, x0, 0       # j counter set to 0

        addi    s2, x0, 16      # set max that i can be
        addi    s3, x0, 8       # set max that j can be

for:    bge     s0, s2, exit    # if i >= 16, exit
        blt     s1, s3, nested  # if j < 8, go to nested label

        addi    s1, x0, 0       # reset j

        addi    s0, s0, 1       # increment i by 1
        beq     x0, x0, for     # move back to beginning of for loop

nested:
        slli    t0, s0, 8       # multiply i by 256
        add     t0, t0, s1      # add by j and save to t0

        slli    t1, s0, 5       # multiply i by 32 - the first index represents i * 32
        slli    t2, s1, 2       # second index of array represents j * 4

        add     t3, t1, t2      # add t1 and t2 to get the address of the nested array
        add     s9, s9, t3      # move s9 to specific address of T[i][j]

        sw      t0, 0(s9)       # saved calculated answer from t0 to the memory address at s9

        addi    s1, s1, 1       # incremeamnt j by 1
        beq     x0, x0, for     # return back to beginning of for loop


exit:   addi    a7, x0, 10      # exit
        ecall
```

In main, all the counters were created for the nested for loops. The maximum number of iterations was also created for each for loop. For the first for loop, a greater than or equal branch is used to exit the loop when the conditions are met. Then, it moves on directly to another branch that checks to see if j is under 8. If it is, then it moves on to performing calculations to find the memory address of the 2-d array and save information to said address. If the nested for loop reaches past 8 iterations, then it moves on to resetting j back to 0, incrementing i by 1, and returning to the beginning of the for loop. When inside the nested for loop, a left bit shift of 8 bits is performed on I in order to multiply it by 256. Then this number is added with j and saved to t0 register, which becomes the information that will be stored at the memory address that is to be calculated. Two-bit shifts to the left are performed. The first is 5 bits to the left, thus multiplying i by 32 as i represents the row of element. Thus, an entire row takes up 32 bytes as each element in the row is 4 bytes. Then the second bit shifts to the left is 2 bits, thus it multiplies j by 4. Then, both calculations are then added together and saved to register t3 where it is added to s9 to move s9 to that specific memory address that was calculated. Then, we save t0 to s9 by using sw t0, 0(s9). J is then incremented by 1 and a equal branch is used to move back to the beginning of the for loop.

3)

```
        add     t1, s1, x0      # create a copy of s1
        add     t2, s2, x0      # create copy of s2

        addi    t0, x0, -1      # i
        addi    s4, x0, 0       # carry tracker

loop:   lb      t3, 0(t1)       # loads values from str1 into t3
        blt     t3, a4, adding  # if the loaded value is less than the value of '0', leave loop
        addi    t1, t1, 1       # moves t1 by 1 to next value
        addi    t0, t0, 1       # adds one to counter of t0
        beq     x0, x0, loop    # loops to beginning


adding: blt     t0, x0, print   # if t0 is equal to null, go to print

        add     t1, s1, t0      # memory address of s1 at s1 offset by t0
        add     t3, s2, t0      # memory address of s2 at s2 offset by t0
        add     t6, s3, t0      # memoyr address of s3 at s3 offset by t0

        lb      t2, 0(t1)       # get ascii at that index
        lb      t4, 0(t3)       # get ascii at index


        sub     t2, t2, a4      # subtract the ascii value of t1 by the ascii of 0 to get the number
        sub     t4, t4, a4      # convert to decimal by subtracting ascii value of 0

        add     t5, t2, t4      # sum of the two digits
        add     t5, t5, s4      # add the remainder
        add     s4, x0, x0      # reset remainder

        bge     t5, a5, carry   # if the sum is greater than or equal to 10, deal with carry

        add     t5, t5, a4      # convert back to ascii

        sb      t5, 0(t6)       # stores byte back into t6

        addi    t0, t0, -1      # decrease counter by 1

        beq     x0, x0, adding  # loop back


carry:  sub     t5, t5, a5      # subtract 10 to get singular digit
        addi    s4, x0, 1       # add one to the remainder counter
        add     t5, t5, a4      # convert t5 back to ascii
        sb      t5, 0(t6)       # save the ascii character back to t6
        addi    t0, t0, -1      # decrease counter i by 1
        beq     x0, x0, adding  # go back to loop


print:
        addi    a0, s3, 0
        addi    a7, x0, 4
        ecall

        # exit
        addi    a7, x0, 10
        ecall
```

In order to do addition of digits stored in memory as characters, I had to do conversions between ASCII and decimals. This was done by subtracting the loaded byte by the ascii value of ‘0’. After converting into a decimal from both str1 and str2, they are added together, taking into account any remainders from previous iterations. If the sum is greater than 10, then the remainder is dealt with by subtracting 10 and then setting s4 in the program to 1 for the next iteration. If the sum is less than 10, then s4 is added in case there is a 1, and then it is reset back to 0.  Then, the total sum of the two digits is converted back to an ASCII by adding the ascii value of ‘0’ to the sum and then loaded to s3 after it was offset by t0.

4)

a) Instruction: or s1, s2, s3

Register Values: or x9, x18, x19

R-type:

| Funct7 | rs2 | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|---|
| 0000000 | 10011 | 10010 | 110 | 01001 | 0110011 |

Machine code (Binary): 0000 0001 0011 1001 0110 0100 1011 0011
Machine code (Hex): 0x013964b3

b) Instruction: slli t1, t2, 16
Register Values: slli x6, x7, 16

I-type

| Funct7 | Imm[0:4] | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|---|
| 0000000 | 10000 | 00111 | 001 | 00110 | 0010011 |

Machine code (Binary): 0000 0001 0000 0011 1001 0011 0001 0011

Machine code (Hex): 0x01039313

c) Instruction: xori x1, x1, -1

Register values: xori x1, x1, -1

I-type

| Imm[11:0] | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|
| 111111111111 | 00001 | 100 | 00001 | 0010011 |

Machine code (Binary): 1111 1111 1111 0000 1100 0000 1001 0011

Machine code (Hex): 0xfff0c093

d) Instruction: lw x2, -100(x3)

Register Values: lw x2, -100(x3)

I-type

| Imm[11:0] | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|
| 111110011100 | 00011 | 010 | 00010 | 0000011 |

Machine code (Binary): 1111 1001 1100 0001 1010 0001 0000 0011

Machine code (Hex): 0xf9c1a103

5)

**A.** S-type

| Imm[11:5] | rs2 | rs1 | funct3 | Imm[4:0] | opcode |
|---|---|---|---|---|---|
| 1111111 | 01010 | 11001 | 010 | 10000 | 0100011 |

Hex: 0xfeaca823

Binary: 1111 1110 1010 1100 1010 1000 0010 0011

Instruction: sw x10, -16(x25)

**B.** I-type

| Imm[11:0] | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|
| 000001000000 | 00100 | 000 | 01110 | 0010011 |

Hex: 0x04020713

Binary: 0000 0100 0000 0010 0000 0111 0001 0011

Instruction: addi x14, x4, 64

**C.** R-type

| Funct7 | rs2 | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|---|
| 0000000 | 00101 | 01010 | 111 | 10111 | 0110011 |

Hex: 0x00557bb3

Binary: 0000 0000 0101 0101 0111 1011 1011 0011

Instructions: and x23, x10, x5

**D.** I-type

| Funct7 | Imm[0:4] | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|---|
| 0100000 | 10100 | 11111 | 101 | 11110 | 0010011 |

Hex: 0x414fdf13

Binary: 0100 0001 0100 1111 1101 1111 0001 0011

Instructions: srai x30 x31, 20