Homework 1

Graded

Student

Ashley Leora Cain

Total Points

96 / 100 pts

Question 1

Memory Packing Question **32** / 32 pts

1.1 **0x1003 - 0x1000** **8** / 8 pts

✔ **– 0 pts** Correct `? 0x31 0x22 0x20`

– 0 pts Correct

– 4 pts No Gap between short and char (i.e. N/A 0x11 0x01 0x22)

– 8 pts Incorrect

– 2 pts Typo on one value

– 4 pts Flipped order of bits (i.e. 0x12 instead of 0x21)

– 4 pts Did not use big-endian

– 8 pts Blank / no answer

1.2 **0x1007 - 0x1004** **8** / 8 pts

✔ **– 0 pts** Correct `0x10 0x21 0x00 0x22`

– 0 pts Correct

– 8 pts Incorrect

– 2 pts Typo on one value

– 2 pts Flipped order of bits (i.e. 0x12 instead of 0x21)

– 4 pts Did not use big endian

– 8 pts Blank / no answer

1.3 **0x100B - 0x1008** **8** / 8 pts

✔ **– 0 pts** Correct `0xFE 0xCA ? 0x06`

– 0 pts Correct

– 8 pts Incorrect

– 2 pts Typo on one value

– 2 pts Flipped order of bits (i.e. 0x12 instead of 0x21)

– 4 pts No Gap between short and char (i.e. N/A 0xBE 0xEF 0x56)

– 4 pts Did not use big endian

– 2 pts 0x not on a half-word boundary

– 4 pts One value correct

– 8 pts Blank / no answer

1.4 **0x100F - 0x100C** **8** / 8 pts

✔ **– 0 pts** Correct `? ? 0xCE 0xFA`

– 0 pts Correct

– 8 pts Incorrect

– 2 pts Typo on one value

– 2 pts Flipped order of bits (i.e. 0x12 instead of 0x21)

– 4 pts Did not use big endian

– 4 pts Did not use compaction

– 4 pts Array out of order

– 8 pts Blank / no answer

Question 2

Fill in Assembly **20** / 24 pts

2.1 **(no title)** **1** / 1 pt

✔ **+ 1 pt** Correct

+ 0 pts Incorrect

2.2 **(no title)** **1** / 1 pt

+ 0 pts Incorrect

✔ **+ 1 pt** Correct

2.3 **(no title)** **2** / 2 pts

✔ **+ 2 pts** Correct

+ 0 pts Incorrect

2.4 **(no title)** **0** / 2 pts

+ 2 pts Correct

✔ **+ 0 pts** Incorrect

2.5 **(no title)** **3** / 3 pts

✔ **+ 3 pts** Correct

+ 0 pts Incorrect

2.6 **(no title)** **3** / 3 pts

✔ **+ 3 pts** Correct

+ 1 pt Missing `$t2`

+ 1 pt Missing `$t0`

+ 0 pts Incorrect

2.7 **(no title)** **2** / 2 pts

✔ **+ 2 pts** Correct

+ 1 pt Missing `addi`

+ 1 pt Missing `7`

+ 0 pts Incorrect

2.8 **(no title)** **1** / 3 pts

+ **3 pts** Correct

✔ + **1 pt** Missing `lea`

+ **1 pt** Missing `x`

+ **0 pts** Incorrect

2.9 **(no title)** **3** / 3 pts

✔ + **3 pts** Correct

+ **1 pt** Missing `sw`

+ **1 pt** Missing `$a0`

+ **0 pts** Incorrect

2.10 **(no title)** **4** / 4 pts

✔ + **4 pts** Correct

+ **0 pts** Incorrect

Question 3

Addressing Modes

24 / 24 pts

✔ + **24 pts** Correct

+ **6 pts** Defines "base + offset" (register + offset) / identifies a difference between offset and index

+ **6 pts** Identifies "base + offset" example (elements in a stack frame, elements in a struct, multi-word elements, etc.)

+ **6 pts** Defines "base + index" (register + register) / identifies a difference between index and offset

+ **6 pts** Identifies "base + index" example (arrays, etc.)

+ **0 pts** Incorrect

+ **0 pts** Incorrect / blank / no answer

Question 4

Registers and Main Memory **20** / 20 pts

4.1 **Memory Pros** **10** / 10 pts

✔ **+ 5 pts** Gives two examples of valid data structures (Array lists, Stack, ect.)

✔ **+ 5 pts** Mentions that memory has a much larger capacity/Registers running out of space

+ 3 pts Only mentions one example of a valid data structure

+ 0 pts Mentions how main memory is used to hold data in a more permanent way
(incorrect since RAM clears itself after shutdown)

+ 0 pts Incorrect / blank / no answer

4.2 **Register Pros** **10** / 10 pts

+ 10 pts Describes a task that frequently needs to access or modify data

✔ **+ 10 pts** Describes a scenario where the very fast speed of registers comes into play

+ 7 pts Mentioned the benefit of register speed or benefit for frequently accessed data, but scenario not applicable or not included

+ 0 pts Incorrect / blank / no answer

Q1 Memory Packing Question

32 Points

For the struct defined below, show how a smart compiler might pack the data to **follow natural alignment restrictions** while **minimizing wasted space as possible**. Pack in such a way that each member is naturally aligned based on its data. The compiler **will not reorder fields** of the struct in memory. Memory is byte-addressable and a char is 1 byte, an int is 4 bytes, and a short is 2 bytes. The architecture is **Big-endian** and supports load word, load byte, and load half word instructions, where a memory word is **4 bytes**.

```
struct x {
  short a;     // a = 0x2022
  char b;      // b = 0x31
  int c;   // c = 0x22002110
  char d;      // d = 0x06
  short e[2]; // e = {0xCAFE, 0xFACE}
} magic = {0x2022,0x31,0x22002110,0x06,{0xCAFE,0xFACE}};
```

Presume an instance of our struct x named magic begins at memory address 0x1000. As a work area, the table below represents a memory diagram where each blank represents a byte of memory. The following questions will ask you what **bytes** of data will be present at various addresses of memory **on a single row in this table**. Answer with a hex number in the format **0xAB** or type **?** if unknown data is stored at that address; enter **a space between each byte**.

Keep the order shown in the table below, that is, with +3 on the left and +0 on the right within a row. For example, if there were a fifth row (starting address 0x1010) with 0x37 stored in 0x1010 and no other known values, it should be submitted like `??? 0x37`.

+3	+2	+1	+0	Starting Address
___	___	___	___	0x1000
___	___	___	___	0x1004

+3	+2	+1	+0	Starting Address
___	___	___	___	0x1008
___	___	___	___	0x100C

Q1.1 0x1003 - 0x1000
8 Points

What hex values are stored in the addresses 0x1003 to 0x1000 (the first row)?

? 0x31 0x22 0x20

Q1.2 0x1007 - 0x1004
8 Points

What hex values are stored in the addresses 0x1007 to 0x1004 (the second row)?

0x10 0x21 0x00 0x22

Q1.3 0x100B - 0x1008
8 Points

What hex values are stored in the addresses 0x100B to 0x1008 (the third row)?

0xFE 0xCA ? 0x06

Q1.4 0x100F - 0x100C
8 Points

What hex values are stored in the addresses 0x100F to 0x100C (the fourth row)?

? ? 0xCE 0xFA

[The following question is just food for thought and does not require an answer: Consider how we could better order the struct to minimize the wasted space.]

Q2 Fill in Assembly
24 Points

Fill in the missing lines below. The LC-2200 assembly should emulate the C code that is provided below. Some operands and instructions are given; others you will need to supply.

We have added a PC-relative instruction LEA to LC-2200. It adds the incremented PC and offset to the label and saves result to the destination register, similar to how BEQ calculates the destination address to be stored in the PC, but instead saves it to the destination register. For example, "lea $s0,addr" adds the offset to the label "addr" to the incremented PC and stores that address in $s0.

Follow the comment next to the code for clarification.

C code

```
int x = 0;
int i = 0;
while (i < 28)
{
  x += i;
  i += 7;
}
```

Assembly Code

```
main:
   addi $t0, $zero, ___     ! zero out loop counter
   addi $t1, $zero, ___     ! set loop limit
loop:
   ___ $t0, $t1, end
   ___ $a0, x             ! load the address of x to $a0
   lw $t2, 0x0(___)       ! get the value of x from address
   add ___, $t2, ___
   ___ $t0, $t0, ___
   ___ $a0, ___           ! load address x to $a0
   ___ $t2, 0x0(___)      ! update value of x to its address
   beq ___, ___, ___      ! repeat loop
end:
   halt

x: .fill 0              ! x initially set to 0
```

If the answer is...	Enter
an opcode (e.g. addi)	the opcode in lower case (e.g. addi)
a number (e.g. 22)	the number without spaces (e.g. 22)
a register (e.g. $a0)	the register with the $ (e.g. $a0)
a lable (e.g. loop)	the name of the label (e.g. loop)

If there are multiple blanks, enter the blanks in order from left to right.

Q2.1
1 Point

`addi $t0, $zero, ___`

0

Q2.2
1 Point

`addi $t1, $zero, ___`

28

Q2.3
2 Points

`___ $t0, $t1, end`

beq

Q2.4
2 Points

`___ $a0, x`

lw

Q2.5
3 Points

`lw $t2, 0x0(___)`

$a0

Q2.6
3 Points

`add ___, $t2, ___`

$t2

$t0

Q2.7
2 Points

`___ $t0, $t0, ___`

addi

7

Q2.8
3 Points

`___ $a0, ___`

lw

x

Q2.9
3 Points

`___ $t2, 0x0(___)`

sw

$a0

Q2.10
4 Points

`beq ___, ___, ___`

$zero

$zero

loop

Q3 Addressing Modes

24 Points

Compare "base + offset" and "base + index" addressing modes. **Give one example of** when we might use "base + offset", and **one** when we might use "base + index".

When using base+index, this might be in a situation in which you are accessing an array of a certain type into memory. For example, if this was an array of ints being stored, all with size of 4 bytes, then to access at index 10, the index would take into consideration the size of the data type (int) AND the index that you want to access (10) by multiplying them. For a situation in which you would use base+index, this might be for increasing a PC in order to jump to a desired instruction. For example, if using the BEQ instruction, instead of allowing the PC to increment by 1, an offset is given for the PC to increment to if the values provided are the same in order to access a specific instruction.

Q4 Registers and Main Memory

20 Points

Registers and main memory are two different ways of storing data on for use in a processor.

Q4.1 Memory Pros

10 Points

List two data structures that you would implement in main memory and NOT registers. Why?

We might need to use main memory instead of registers in situations where a lot more memory is needing to be stored since registers can't store as much data. Two data structures that would need this type of memory would be arrays or stacks. In these situations, memory would be useful to store the many values or data points that can be within these data structures, and there would be easier storage and access to these individual components through memory indexing.

Q4.2 Register Pros

10 Points

Give one example where registers make a task run faster? Justify your answer.

If we are reusing data multiple times then registers may be more helpful since we won't have to keep re-accessing the value in memory. This might be helpful in a situation where you are finding the answer to an exponent, like if you need to find 5^6 or something like that. In that situation, it would be really helpful to keep the value of the 5 in a register because you are going to need to multiply with that number multiple times. If this value was saved in memory only, then you would need to access memory each time you wanted to use that number.