Homework 1

Graded

Student

Vidit Dharmendra Pokharna

Total Points

100 / 100 pts

Question 1

Memory Packing Question **32** / 32 pts

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

- ✔ **– 0 pts** Correct `0x20 0x22 ? 0x99`
- **– 4 pts** No Gap between short and char (i.e. `? 0x20 0x22 0x99`)
- **– 2 pts** Typo on one value
- **– 4 pts** Did not use little-endian (i.e. `0x22 0x20 ? 0x99`)
- **– 2 pts** Wrong address order, thought lower addresses were on the left (i.e. `0x99 ? 0x22 0x20`)
- **– 4 pts** Flipped order of bits (e.g. 0x02 instead of 0x20)
- **– 8 pts** Incorrect
- **– 8 pts** Blank / no answer

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

- ✔ **– 0 pts** Correct `0x22 0x00 0x21 0x10`
- **– 2 pts** Typo on one value
- **– 2 pts** Flipped order of bits (e.g. 0x12 instead of 0x21)
- **– 2 pts** Wrong address order, thought lower addresses were on the left (i.e. `0x10 0x21 0x00 0x22`)
- **– 4 pts** Did not use little-endian (i.e. `0x10 0x21 0x00 0x22`)
- **– 8 pts** Blank / no answer
- **– 8 pts** Incorrect

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

- ✔ **– 0 pts** Correct `0xCA 0xFE ? 0x06`
- **– 2 pts** Typo on one value
- **– 2 pts** Flipped order of bits (e.g. 0xAC instead of 0xCA)
- **– 2 pts** Wrong address order, thought lower addresses were on the left (i.e. `0x06 ? 0xFE 0xCA`)
- **– 4 pts** Incorrect Gap
- **– 4 pts** No Gap between short and char (i.e. `? 0xCA 0xFE 0x06`)
- **– 4 pts** Did not use little endian (i.e. `0xFE 0xCA ? 0x06`)
- **– 4 pts** Array out of order (i.e. `0xFA 0xCE ? 0x06`)
- **– 8 pts** Blank / no answer
- **– 8 pts** Incorrect

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

✔ **– 0 pts** Correct `? ? 0xFA 0xCE`

– 2 pts Typo on one value

– 2 pts Flipped order of bits (e.g. 0xAF instead of 0xFA)

– 2 pts Wrong address order, thought lower addresses were on the left (i.e. `0xCE 0xFA ? ?`)

– 4 pts Did not use little-endian (i.e. `? ? 0xCE 0xFA`)

– 4 pts Array out of order (i.e. `? ? 0xCA 0xFE`)

– 8 pts Blank / no answer

– 8 pts Incorrect

Question 2

Fill in Assembly **24** / 24 pts

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

✔ **+ 1 pt** Correct (0)

+ 0 pts Incorrect

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

✔ **+ 1 pt** Correct (40)

+ 0 pts Incorrect

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

✔ **+ 2 pts** Correct (beq)

+ 0 pts Incorrect

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

✔ **+ 2 pts** Correct (lea)

+ 0 pts Incorrect

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

✔ **+ 3 pts** Correct ($a0)

+ 1 pt Used (a0) instead of ($a0)

+ 0 pts Incorrect

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

✔ **+ 3 pts** Correct ($a1; $t0)

+ 2 pts Missing $ prefix for one or both registers

+ 1 pt Missing `$a1`

+ 1 pt Missing `$t0`

+ 0 pts Incorrect

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

✔ **+ 2 pts** Correct (addi; 8)

+ 1 pt Missing `addi`

+ 1 pt Missing `8`

+ 1 pt Incorrect formatting i.e. $addi

+ 0 pts Incorrect

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

✔ **+ 3 pts** Correct (lea; x)

+ 1 pt Missing `lea`

+ 1 pt Missing `x`

+ 0 pts Incorrect

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

✔ **+ 3 pts** Correct (sw; $a0)

+ 2 pts Incorrect formatting i.e. $sw

+ 1 pt Missing `sw`

+ 1 pt Missing `$a0`

+ 0 pts Incorrect

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

✔ **+ 4 pts** Correct ($zero; $zero; loop)

+ 3 pts Incorrect formatting i.e. $addi

+ 2 pts Missing one

+ 1 pt Missing two

+ 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 Memory

20 / 20 pts

4.1 Memory or Register?

6 / 6 pts

✔ **+ 6 pts** Correct (Linked-List; Queue)

+ 5 pts 1 missing/extra item

+ 4 pts 2 missing/extra items

+ 3 pts 3 missing/extra items

+ 2 pts 4 missing/extra items

+ 1 pt 5 missing/extra items

+ 0 pts Incorrect

4.2 Explanation

14 / 14 pts

✔ **+ 14 pts** Mentions that memory has a much larger capacity/registers running out of space

+ 0 pts Incorrect

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 **Little-endian** and supports load word, load byte, and load half word instructions, where a memory word is **4 bytes**.

```
struct x {
  char b;      // b = 0x99
  short a;     // a = 0x2022
  int c;   // c = 0x22002110
  char d;      // d = 0x06
  short e[2]; // e = {0xCAFE, 0xFACE}
} magic = {0x99, 0x2022,0x22002110,0x06,{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)?

0x20 0x22 ? 0x99

Q1.2 0x1007 - 0x1004
8 Points

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

0x22 0x00 0x21 0x10

Q1.3 0x100B - 0x1008
8 Points

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

0xCA 0xFE ? 0x06

Q1.4 0x100F - 0x100C
8 Points

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

? ? 0xFA 0xCE

[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 i = 0;
int x = 0;
while (i < 40)
{
  x += i;
  i += 8;
}
```

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 $a1, 0x0(___)       ! get the value of x from address
   add ___, $a1, ___
   ___ $t0, $t0, ___
   ___ $a0, ___           ! load address x to $a0
   ___ $a1, 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, ___`

40

Q2.3
2 Points

`___ $t0, $t1, end`

beq

Q2.4
2 Points

`___ $a0, x`

lea

Q2.5
3 Points

`lw $a1, 0x0(___)`

$a0

Q2.6
3 Points

`add ___, $a1, ___`

$a1

$t0

Q2.7
2 Points

`___ $t0, $t0, ___`

addi

8

Q2.8
3 Points

`___ $a0, ___`

lea

x

Q2.9
3 Points

`___ $a1, 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".

"base + offset" is used when you want to access elements at fixed offsets within a data structure, while "base + index" is used when you need to access elements based on dynamic/variable indices that may change during program execution.

Suppose we have an array of integers starting at memory location 0x1000, and we want to access the 3rd element (index 2) of the array. You can use "base + offset" addressing in this case.

Suppose we have an array, and we want to access a specific value, which is represented by an index. We would use "base + index" addressing, where the index is an offset from the base address. The effective address of the memory location is calculated by adding the contents of the base register and the index register.

Q4 Registers and Main Memory

20 Points

Some data should be stored in memory, and some should be stored in registers. Consider the different types of data in Q4.1 and choose which data is most fitting to be held in memory rather than in a register. Then, in Q4.2, explain why you chose that type of data.

Q4.1 Memory or Register?

6 Points

Which data should be stored in memory rather than in a register:

- [ ] Loop Counter
- [ ] Return Value
- [x] Linked-List
- [ ] Function Parameters
- [x] Queue
- [ ] Return Address

Q4.2 Explanation

14 Points

Explain why the data you selected in 4.1 should be stored in memory.

Linked list: Long lists would require a lot of registers, so storing them in memory with the reference of the head of the list as a register is more practical

Queue: They are pretty dynamic and storing in memory allows for efficient allocation and deallocation, and thus flexibility