Q1 Memory Packing Question

32 Points

For the struct defined below, show how a smart compiler might pack the data to **minimize wasted space** and follow alignment restrictions. Pack in such a way that each member is naturally aligned based on its data. Assume the compiler **will not reorder fields** of the struct in memory. Assume a char is 1 byte, int is 4 bytes, and a short is 2 bytes. Moreover. assume the architecture is **Big-endian** and its granularity supports load word, load byte, and load half word, where a memory word is 4 bytes.

```
struct x {
    short a;    // a = 0x1100
    char b;     // b = 0x41
    int c;      // c = 0x12345678
    char d;     // d = 0x61
    short e[2]; // e = {0x8765, 0x4321}
}
```

Presume an instance of our struct x with the values listed above begins at memory address 0x1000. As an example, the table below represents a memory diagram where each "o" represents a byte of memory. The following questions will ask you what **bytes** of data will be stored at various addresses of memory **on a single row in this table**. Answer with a hex number in the format **0xAB** or type **N/A** if no data is stored at that address with **a space between each address**.

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 values, it would be submitted like `N/A N/A N/A 0x37`.

+3	+2	+1	+0	Starting Address
o	o	o	o	0x1000
o	o	o	o	0x1004
o	o	o	o	0x1008
o	o	o	o	0x100C

Q1.1 0x1003 - 0x1000

8 Points

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

N/A 0x41 0x00 0x11

Q1.2 0x1007 - 0x1004

8 Points

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

0x78 0x56 0x34 0x12

Q1.3 0x100B - 0x1008

8 Points

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

0x65 0x87 N/A 0x61

Q1.4 0x100F - 0x100C

8 Points

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

N/A N/A 0x21 0x43

(The note below is just some food for thought and **will not** affect your answers)
In the future, consider how to better order your structs to minimize the wasted space.

Q2 Registers and Main Memory

24 Points

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

Q2.1 Memory Pros

12 Points

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

Array and Stack. Data structures are used to store many instances of data. The REG-FILE (at least in LC-2200) contains 16, 32-bit registers compared to the main memory that contains a lot more space (the majority of those registers can't hold temp values too); it would be impossible to store a lot of memory in registers, so they should be implemented in main memory.

Q2.2 Register Pros

12 Points

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

When you are allocating/deallocating data from the stack in main memory, it is much faster to keep a register at hand to keep the stack pointer and just using the add immediate version of the ADD instruction to keep track of the offset when allocating/deallocating. If you didn't have a register for the stack pointer, you'd have to keep loading it from memory into the ALU and then add a number to it, and it is much faster to obtain data from a register than from main memory.

Q3 Calling Convention

20 Points

Explain what the Frame Pointer is and its relevance to the LC-2200 calling convention.

The Frame Pointer points to the top of the current activation record ALWAYS. NOT to be confused with the stack pointer. The difference between the two is that the stack pointer moves as memory is allocated/deallocated from the stack but the frame pointer always points to the top of the current activation record. The reason we have this is that we need a reference to be able to access local variables. There is too much overhead in trying to find the offset of a certain local variable every time a new variable is added or removed to the stack.

Q4 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".

If you wanted to access the contents of a struct, it would be helpful to reference the starting point of the struct and then the offset of a certain field of the struct based on the data types of the fields. It would therefore be advantageous to the base+offset mode.

When trying to obtain the jth element of an index, it is easiest to keep the starting address of the array in one register and keep the offset of the jth element in a second array. This example uses the base+index addressing mode.

Homework 1

GRADED

STUDENT
Eric Anders Gustafson

TOTAL POINTS
88 / 100 pts

QUESTION 1

Memory Packing Question		**32** / 32 pts
1.1	0x1003 - 0x1000	**8** / 8 pts
1.2	0x1007 - 0x1004	**8** / 8 pts
1.3	0x100B - 0x1008	**8** / 8 pts
1.4	0x100F - 0x100C	**8** / 8 pts

QUESTION 2

Registers and Main Memory		**24** / 24 pts
2.1	Memory Pros	**12** / 12 pts
2.2	Register Pros	**12** / 12 pts

QUESTION 3

Calling Convention	**20** / 20 pts

QUESTION 4

Addressing Modes	**12** / 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