1. What is the address of the first instruction that will be executed when running the program?
   1. 0x00400000
2. Where did you find this information?
   1. In the Text Segment under the Execute tab where it says Address for the first line of code that was run.
3. Which memory locations contain the value for the variable val?
   1. Memory 0x1001000
4. Diagram and label a process' address space in Mars. By this, I mean, draw a column labeled 0x00000000 at the bottom and 0xffffffff at the top. Then, show which portions of this address space are used used for instructions, user data, the stack, kernel data, etc. Hint: You will have to look in different MARS windows to find these different areas. Also, the stack grows "down", meaning that each data item added to the stack has a smaller address than the previous item.

|  |  |
| --- | --- |
| Memory Location | Usage |
| 0x10000000 - 0x10010000 | extern |
| 0x10010000 - 0x10040000 | data |
| 0x10040000 - 0x10008000 | Heap |
| 0x10008000 - 0x00400000 | Stack |
| 0x00400000 - 0x90000000 | Text |
| 0x90000000 - 0xFFFF0000 | Kernal |
| 0xFFFF0000 - 0xFFFFFFFF | Memory Mapped I/O |

1. What is the maximum program size for this configuration? (In other words, how many instructions can your program have before they overflow into data memory?)
   1. There are 0x0FC00000 addresses available between the .text storage and the .data storage. Since instructions are spaces out by four places, we divide this number by 4 to get 0x03F00000 maximum instructions or 66,060,288.
2. What lines of machine language does the addi pseudo-instruction on line 11 produce?
   1. It produces the ‘lui’, then the ‘ori’, and finally the ‘and’
3. What does "lui" stand for? (Hint: Look on page A-57 in the SPIM guide.)
   1. Load Upper Immediate - the immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.
4. Why do the two addi instructions result in different numbers of actual instructions? (Hint: Write each constant out in hex.)
   1. The first ‘addi’ is loading 32767 into v0, which currently has no value. When 65538 is then added to v0, the sum requires more bits than the register has available. So the ‘overflow’ is being pushed into the ‘lui’
5. Explain how the MIPS assembler applies the "Make the common case fast" principle to addi.
   1. It is assuming that the common case is going to be that the numbers being added will result in a sum that fits into the bits of the register. You can see this with the way that the first ‘addi’ works. This would result in the fastest way to add. So then it takes the longer route of moving bits with ‘lui’ for the outlier cases.
6. What is the hex representation of the "immediate" parameter to the lui instruction generated for line 12?
   1. 0x0000 1001
7. Why is this the immediate value that is used (i.e., what does it represent)? Hint: Look in the Data Segment window.
   1. It is taking the 1001 and does a ‘lui’ to put the values into the upper bits of the address to get the memory address of val1.
8. Now, look at the second machine instruction generated for line 12 (the first lw instruction). Notice that this instruction has three parameters. Describe the function of all three parameters. (Hint: Examine the machine language generated for lines 13 and 15 as well as the description of lw on page A-67.)
   1. This is loading the value from the memory address generated in the previous line into the register. The $1 is the memory address where the data was saved, the 0x0000 0000 is the offset, and the $8 is the register that the data is being loaded into.
9. Are all three parameters necessary in order for lw to be able to access the entire 4GB memory space; or, could you eliminate the offset parameter? (In other words: If lw did not have an offset parameter, would there be memory locations that could not be specified using the remaining two lw parameters?) If so, give an example. If not, give a sequence of machine instructions that could be used to load val2 into $t1 with a 2-parameter version of lw.
   1. If you removed the offset, the command would work. However, you would lose access to the remaining 28 bits of memory per address.
10. Explain why the three-parameter version of lw is useful. Include an explanation of how can it be used to "make the common case fast." (In other words, how it can be used to reduce the number of instructions needed by the program.) (Hint: Look for redundant code in the execute window for exampleCIT-3.s.)
    1. The three parameter assumes for the common case of accessing deeper into the memory that just the first memory address. It allows the computer to load the memory address using smaller faster commands that fit in 32 bits.
11. Explain the cost of the three-parameter version of lw. In particular, include an explanation of how the third (i.e., offset) parameter can potentially slow the computer.
    1. The third parameter forces you to do not only send the memory location, but also do an add to get the correct memory location. Additionally, this also requires more hardware, which can leed to more gate delays.
12. How is the li pseudo-instruction implemented? In other words, which "real" instructions are used to implement the li pseudo-instruction? (Remember, register 0 always contains the value 0.)
    1. It performs the ‘addiu’ (Add Immediate Unsigned’) loading into $2, with $0 and the value defined by the users.
13. How does MIPS implement the move pseudo-instruction?
    1. It calls the ‘addu’ (Add Unsigned) into the designated register by adding the value to move with 0.
14. Would a built-in move be faster than the MIPS implementation? Why or why not? Consider the effects on both the time for the individual instruction, and the overall speed of the processor.
    1. The individual instruction would be faster, but the overall speed of the machine would suffer because of the complexity of implementing a specific move. By using add, the number of commands for the machine is kept at a minimum.
15. For each instruction in add\_xy.s marked with a "\*" in the comment (17, 18, 29, 30, 41, 42, and 72) complete a table showing how the assembly-language instruction is mapped into a machine-language instruction. For pseudo-instructions, create one table for each machine instruction produced by the assembler.

Instruction 17: la $a0, string1

Machine Instruction: 0x3C011001

B\_Machine Instruction: 001111 00000 000010001000000000001

Instruction Field (Decimal):15 0 69633

Field Function: op rx label

Instruction 17(2):

Machine Instruction: 0x34240000

B\_Machine Instruction: 001101 00001 00100 0000 0000 0000 0000

Instruction Field (Decimal):5 1 4

Field Function: op rt rs imm

Instruction 18: li, $v0, 4

Machine Instruction: 0x24020004

B\_Machine Instruction: 001001 00000 00010 0000 0000 0000 0100

Instruction Field (Decimal):9 0 2 4

Field Function: op rt rs imm

Instruction 29: lw $s2, quitVal

Machine Instruction: 0x3C011001

B\_Machine Instruction: 001111 00000 00001 0001 0000 0000 0001

Instruction Field (Decimal): 15 unused 1 4097

Field Function: op rs rt imm

Instruction 29(2):

Machine Instruction: 0x8C32005C

B\_Machine Instruction: 100011 00001 10010 0000 0000 0101 1100

Instruction Field (Decimal):35 1 18 92

Field Function: op rs rt imm

Instruction 30: beq $s0, $s2, exit

Machine Instruction: 0x12120018

B\_Machine Instruction: 000100 10000 10010 0000 0000 0001 1000

Instruction Field (Decimal):4 16 18 24

Field Function: op rs rt imm

Instruction 41: syscall

Machine Instruction: 0x0000000C

B\_Machine Instruction: 0000 0000 0000 0000 0000 0000 0000 1100

Instruction Field (Decimal):12

Field Function: op

Instruction 42: move $s1, $v0

Machine Instruction: 0x00028821

B\_Machine Instruction: 000000 00000 00010 10001 00000 100001

Instruction Field (Decimal):0 0 2 17 0 33

Field Function: op rs rt rd unused func

Instruction 72: j top

Machine Instruction: 0x08100004

B\_Machine Instruction: 000010 00 0001 0000 0000 0000 0000 0100

Instruction Field (Decimal):2 1048580

Field Function: op addr

1. Why does the la pseudo-instruction in line 17 generate two assembly instructions while the li pseudo instruction in line 18 generates only one?
   1. Because 17 is loading a variable, while 18 is loading a literal.
2. What is the value of the immediate parameter for the beq instruction on line 30?
   1. 0x00000018
3. Where does this number come from (i.e., how does the assembler calculate it)?
   1. It comes from converting the hex number in the machine code to binary and then parsing the ‘imm’ section (the last 16 bits).
4. What is the value of the immediate parameter for the j instruction on line 72? Be careful, you need to look at the actual hex value of the instruction, not the number in the "Basic" column.
   1. 0x00100004
5. Where does this number come from (i.e, how does the assembler calculate it)?
   1. It comes from converting the hex number in the machine code to binary and parsing the last 26 bits to find the address.