Computer Architecture Project 1 - MIPS Disassembler

10% of Course Grade, Due: in three weeks (Oct. 3); 10% off each week late; Not accepted after Oct. 17.
Keep your source code. It may be used later in project three.
email addresses to send it : hendrick@bu.edu AND kspalmer@bu.edu **Please put "CS472"in the subject line to avoid it getting overlooked!**

****
* SHARE IDEAS, BUT NOT CODE. NOT *ONE* LINE! BASING YOUR WORK ON *
* SOMEONE ELSE'S WILL GET YOU AN "F" FOR THE COURSE. DON'T USE *
* OTHERS WORK "TO GET AN IDEA OF HOW TO DO IT" OR FOR ANY OTHER *
* REASON. NO EXCUSES. NO EXCEPTIONS. NO POSTING SOLUTIONS, EVER! *
****

Your project is to write a partial disassembler for MIPS instructions. That is, your input will be the 32-bit machine instructions that a compiler or assembler produces. Your program then figures out what the original source instructions were that created those 32-bit machine instructions and outputs them. The possible source instructions that you must be able to disassemble are: add, sub, and, or, slt, lw, sw, beq, bne. Ignore all other MIPS instructions.

The specific machine instructions that you will disassemble (one after another in this exact order) are: 0x032BA020, 0x8CE90014, 0x12A90003, 0x022DA822, 0xADB30020, 0x02697824, 0xAE8FFFF4, 0x018C6020, 0x02A4A825, 0x158FFFF7, 0x8ECDFFF0

That is, the above 32-bit instructions will be the input to your program. (Eight hex digits are 32 binary bits.) Feel free to embed them in the program itself so you can avoid typing them in each time. Your program will then analyze a 32-bit instruction and figure out what the opcode, register operands and other fields in the instruction are and then print out the assembly language instruction that produced it. Assume that the first instruction begins at address hex 9A040 and the rest follow right after that one. **You must output the address along with the instruction.**

For example, if your program determines that the first 32-bit machine instruction above is the instruction lw $10, 12 ($20) (it isn't, but if it were) then your output for that instruction would be:

9A040 lw $10, 12 ($20)

You'll then go on and do the next 32-bit instruction, specifying its address in hex (the address for an instruction immediately following one at 9A040) and what instruction caused those 32-bits. The instruction should show the correct syntax so that an assembler could correctly evaluate it (with the exception of the branch instruction detailed below). Output the numerical registers (e.g., $7, $0) as opposed to the symbolic descriptions (e.g., $s3, $t1). For any load or store instructions, show the offset value as a **signed** decimal number.

I suggest using a 16-bit variable (a **short)** for the I-format offset so that if the number is negative, it will be accurately handled. For example, if you take the number -4, it is represented in the 16-bit offset as xFFFC. When you do the bit-wise AND on the 32-bit instruction to remove the other 16 bits, you'll correctly get xFFFC if you put it in a **short** but incorrectly get x0000FFFC if you put it in an **int**. If you print out the former for a load/store, you'll correctly show -4, but if you print out the latter you'll get a large positive number (64K-4), which is an error. This makes it so you don't have to do sign-extension.

For the branch instructions (beq, bne), don't try to invent a label for the destination to branch to. Just indicate the address of the destination. So if you've disassembled a beq $7, $8 instruction at 9A05C that is branching to an instruction at address 9A080, then your output should be:

9A05C beq $7, $8, address 9A080

You do not need to worry about branches that are an absurd distance away, which could cause problems with the "**short**" approach I suggested above.

The key to this project is using bitwise AND operations and logical (not arithmetic) shifts. **This is mandatory!** A bitwise AND operation can be used to zero out all but some desired bits. A shift can get those bits into the desired position. In case you haven't used these operations before, here's an example in C/C++. If the least significant bit is called bit 0 and the most significant bit is 31 and you want to look just at bits 7-10, you would do the following:

Bits_7_to_10 = All_32_bits & 0x00000780;

The bitwise AND (the & operation) does a bit-by-bit AND between the value in All_32_bits and the constant (known as a mask) hex 00000780. Hex 00000780 in binary is 0000 0000 0000 0000 0000 0111 1000 0000, which is zeros in all bits except bits 7, 8, 9, and 10. Any bit ANDed with a 0 is turned into 0, while any bit ANDed with a 1 is left as it is (1 AND 1 yields 1 while 0 AND 1 yields 0).

If you then wanted to take bits 7-10 and get them into the least significant position, you could shift by seven bits (note: in java, use ">>>" not ">>"):

Shifted_bits_7_to_10 = Bits_7_to_10 >> 7;

So if All_32_bits was hex F2C7AE82 (binary 1111 0010 1100 0111 1010 1110 1000 0010), then Bits_7_to_10 would be hex 00000680 (binary 0000 0000 0000 0000 0000 0110 1000 0000) and Shifted_bits_7_to_10 would be hex 0000000D (binary 0000 0000 0000 0000 0000 0000 0000 1101).

Other Details: The "shamt" field in R-type instructions can always be assumed to be all zeroes.

Don't feel that you have to "convert" hex numbers to decimal or to a string. You shouldn't have to convert anything. Say you have the number hex 40, which is decimal 64, which is also binary 0100 0000. You don't have to convert the numbers to get from one to the other since they are the same number. It's just a question of how you output the number. If you're using C, then if your printf statement has a %d, then it will print out 64. If instead it has a %x (for example, printf("var= %x", var)), then it'll print out the hex number 40. **They are the same number.** It's just a question of how to show it. The same holds true for signed numbers. If you have the two's complement representation of -1 (FFFF FFFF or all binary 1s) in a signed integer, then doing a printf with %d you'll get -1 printed out. No conversion is necessary.

Taking another example, if you've set up an int for your first instruction (you'll actually want to use an array of instructions, but I want to keep this simple) like this: int FirstInstruction = 00x022DA822;
Then for C++ you can do: cout << "The instruction shown in hex is: " << hex << FirstInstruction;
For java: System.out.println("The instruction shown in hex is: " + Integer.toHexString((firstinstruction)));

Finally, if you download the SPIM Simulator to play around with MIPS instructions, be aware that it does some non-standard things with internal 32-bit formats on some instructions. Those things are fine for a simulator, but wouldn't be on real hardware. As a result, the book's description of the 32-bit instructions is the correct version even if it disagrees with some 32-bit instructions you might see on the SPIM Simulator. You must follow the book's approach.

Reminder: Show addresses in hex. All other values should be in decimal, which is what an assembler defaults to.

Suggestion: Before you start writing your program, figure out by hand what the correct instructions should be so that you are sure you understand how it's done before writing your program. Of course, your program must then figure out those correct results in an algorithm that works in general (not hardcoded to work only for the results you calculated), but you'll be able to proceed confidently from that point. Also, make sure you understand the bitwise AND description above since it's fundamental to the process.

Last note: Your output MUST reflect your source code. Otherwise there's a minimum 40% deduction. You can't edit your output to get rid of errors. Your program must create the correct output itself!