In this problem, we are going to consider several instructions that we want to add to our beta.

For each of these instructions, we will need to decide what the minimum requirement is to add that instruction.

The simplest addition would be a macro that references a single already existing beta instruction.

If our new instruction cannot be implemented by simply defining a macro, then we want to consider whether adding a new opcode, and producing the appropriate control ROM signals for it, will enable the new operation to be executed on our existing Beta datapaths.

Finally, if neither the macro or Control ROM solutions work, then we need to specify that the instruction cannot be implemented without making actual hardware changes to the Beta.

The first instruction that we want to consider adding to our Beta is a SWAPR instruction which swaps the contents of registers Rx and Ry in a single clock cycle.

The constraint that this must be done in a single clock cycle points us to the fact that the Beta hardware does not have the ability to write to two different registers in the same clock cycle.

Thus, in order to add this instruction to the Beta, new hardware would need to be added.

The next instruction that we want to consider adding to our beta is a NEG instruction.

This instruction should take the two's complement negation of register Rx and store it into register Ry.

The first question we want to ask ourselves is whether or not we might be able to implement this using a macro.

Since all we are trying to do is produce the negative of a given value, we can write a macro for this instruction which subtracts Rx from R31 and stores that result into Ry.

Note that this macro will not work for the corner case which is the largest representable negative number because the negation of that number cannot be represented using 32-bit two's complement representation.

For all other cases, however, this macro works as expected.

The next instruction that we want to consider adding to our Beta is a PC-relative store instruction.

The way this instruction works is that it writes the contents of register Rx to a location in memory whose address is computed by adding PC + 4 + 4*SEXT(C).

The only existing store operation in the beta is a store that writes to the address that is computed by adding the

contents of register Ry and the sign extended literal C.

Since this is not equivalent to the store relative instruction's behavior that means that we cannot implement this instruction as a macro.

So next we consider whether or not we can implement this instruction using our existing Beta datapaths.

This beta diagram highlights in red the dataflow through the existing Beta datapaths that would perform the desired PC relative store instruction.

The way this instruction works is that the extra adder under the instruction memory is used to compute the value of the effective address which is PC + 4 + 4*SEXT(C).

The ASEL, or A select signal is then set to 1 to pass that value to the ALU as the A operand.

The ALUFN is then set to A to continue passing that value through the ALU in order for it to be used as the address for the data memory.

This address is labeled MA, or memory address in the beta diagram.

The value that is written to memory is the value of register Rx.

In store operations, the first operand corresponds to register Rc.

So we set RA2SEL = 1 in order to select Rc, which is Rx in this case, as the register whose contents should be written to memory.

The value of this register is made available via the RD2 register file port which then feeds the MWD, or memory write data signal for the memory.

There are a couple other memory related signals that we need to set appropriately.

They are MWR, which stands for memory write read, and controls the write enable of the data memory.

In order to be able to write to the memory, the write enable must be set to 1.

MOE is the memory output enable.

We set this to 0 to specify that no output should be enabled from the memory.

Note that you may think that MOE should be a don't care since we are never making use of the MRD, or memory

read data, signal in our datapath.

However, by setting it to 0 we allow ourselves to potentially use the same databus for the read and write data of the memory.

This is not explicitly shown in our beta diagram but is the reason that MOE is specified as 0 for us.

The other control signal that we must set to 0 is WERF, which stands for write enable register file.

Setting this signal to 0 ensures that no value will be written back into our register file.

This allows us to then set WDSEL and WASEL to don't cares.

The last control signal is BSEL which is also a don't care because the B operand is ignored by the ALU for this instruction.

Finally, the PCSEL = 0 in order to increment the PC by 4 so that the next instruction will be fetched.

So our completed Control ROM for the STR operation is shown here.

The last instruction we want to add to our beta is the BITCLR(Rx, Ry, Rz) instruction.

This instruction performs a bitwise AND of the contents of register Ry with the complement of the contents of register Rx.

There is no existing beta instruction that performs this functionality so using a macro is not an option.

Next, we want to consider whether or not we could implement this instruction using our existing datapaths with changes to our control ROM.

To answer this question, you need to realize that the operation that you are trying to perform here is a boolean operation.

In module 1, when implementing the ALU lab, we learned that the way that the bool module works is that if you set the ALUFN to 10abcd, then the ALU would produce the output defined by this truth table for every pair of bits Bi and Ai.

So for example, to implement the AND function, we simply set a = 1, b = 0, c = 0, and d = 0 as shown in this truth table which is the truth table for an AND function.

The truth table for the BITCLR operation is shown here.

One additional column, NOT(Rx)[i] has been added to show the intermediate step of negating Rx[i].

Then if you take the AND of the Ry[i] column and the Not(Rx)[i] columns you get the result Rz[i].

This means that the ALUFN for the BITCLR operation is 10 followed by 0100.

The rest of the control signals can be determined by looking at this highlighted beta diagram which shows in red the paths that must be followed in order to properly implement the BITCLR operation.

The instruction memory specifies the registers Ra and Rb, in our case Rx and Ry, that are to be used by this operation.

Setting RA2SEL to 0 tells the register file to read Rb, or Ry, as the second operand.

Then setting ASEL and BSEL to 0 passes the values of Rx and Ry to the ALU.

The ALUFN is used to specify the particular boolean operation that we are performing.

Then WDSEL = 1 in order to feed the results of the ALU back to the register file.

The Rc register is Rz and it is the register that the result should be written to.

To make that happen, we set WASEL = 0, and WERF = 1.

To avoid anything being written to the data memory, MWR is set to 0.

MOE can be a don't care because we are not using the memory for reading or writing and setting WDSEL to 1 ignores anything that is on the MRD, or memory read data, line.

Finally, the PCSEL = 0 in order to increment the PC by 4 so that the next instruction will be fetched.

So our completed Control ROM for the BITCLR operation is shown here.