In order to better understand the role of each of the beta control signals, we will work through an example problem that provides us with a partially filled control table for 5 different instructions.

Two of these instructions are existing beta instructions that we must infer from the provided control signals.

The other three are three new instructions that we are adding to our beta by modifying the necessary control signals to produce the desired behavior of each of the operations.

The first instruction that we want to add to our beta is an LDX instruction which is a load that is double indexed.

What this means is that in order to produce the effective address of the load, instead of adding the contents of a register to a constant as is done in the LD instruction, we add the contents of two different registers.

So the address for this load operation is the result of adding together the contents of registers Ra and Rb.

The contents of the memory location pointed to by this effective address are loaded into register Rc.

Finally, the PC is incremented by 4 to point to the next instruction.

The second instruction that we want to add to our beta is a MVZC instruction which is a move constant if zero instruction.

The way this instruction works is that if the contents of register Ra equal zero, then the sign extended version of the literal constant will be loaded into register Rc.

This is followed by incrementing the PC to point to the next instruction.

The third instruction that we want to add to our beta is a STR instruction which is a store relative instruction.

For this instruction, the effective address is computed by sign extending the constant C, multiplying it by 4 and adding it to PC + 4.

The contents of register Rc are then stored at the memory location pointed to by the effective address that was just computed.

As a final step, the PC is incremented by 4 to point to the next instruction.

We are given the partially filled control ROM shown here.

It is our job to fill in all the yellow boxes labeled with a ?.

Let's begin by looking at the top row of this table.

The value that stands out as a bit different in this row is the PCSEL value which is equal to 2.

For most instructions PCSEL equals 0, for branch instructions it equals 1, and for JMP instructions it equals 2.

This means that the instruction described in this row must be a JMP instruction.

Zooming in on the PCSEL control logic from the beta diagram, we see that normally PCSEL = 0 to go to the next instruction.

PCSEL = 1 in order to perform a branch operation, and PCSEL = 2 in order to perform a jump operation where the target of the jump is specified by JT, or the jump target.

This means that the instruction described in this row must be a JMP instruction.

The behavior of a JMP instruction is shown here.

The effective address is calculated by taking the contents of RA and clearing the bottom 2 bits so that the value becomes word aligned.

The address of the next instruction, which is PC + 4, is stored in register Rc in case we need to return to the next instruction in the program.

The PC is then updated with the new effective address in order to actually continue execution at the destination of the JMP instruction.

This dataflow diagram highlights the required dataflow through the beta in order to properly implement the JMP instruction.

Note that no red lines pass through the ALU or memory because the ALU and memory are not used for this instruction.

The control signals that must be set in order to follow this path in the beta are as follows: WDSEL, or write data select, must be set to 0 in order to pass the value of PC + 4 through the WDSEL mux.

WERF, or write enable register file, must be set to 1 in order to enable writing to the register file.

WASEL, or write address select, must be set to 0 in order to write to the Rc register and not to the XP register.

ASEL, BSEL, and ALUFN, are all don't cares for the JMP instruction.

In addition MOE, which stands for memory output enable, is also a don't care because this instruction does not use the memory data.

The one control signal related to the memory that we do need to worry about is the MWR, or memory write read, signal which must be set to 0 so that no value will be written to memory.

Going back to our control ROM and filling in the value of WERF, we see that the control signals for the JMP instruction correspond to the dataflow diagram of the beta that we just looked at.

Moving on to row two of our control ROM, we see that now we have PCSEL = Z in this row.

This suggests that the instruction corresponding to this row is some kind of a branch instruction.

Of our two branch instructions, the one that branches when Z = 1 is BEQ.

This means that this row corresponds to a BEQ operation.

The rest of the control signals for the BEQ operation look just like the ones for the JMP because here too, the ALU and memory are not used so the only ALU and memory related signal that must be set is MWR so we don't write to memory.

Furthermore, like the JMP instruction, the branch instructions also store the return address in register Rc, so the behavior of the control signals related to the register file are all the same.

We now take a look at the third row of the control ROM.

In this row, we are actually told that the corresponding instruction is the newly added LDX instruction.

So it is our job to determine how to set the missing control signals in order to get the desired behavior for this operation.

Recall that the expected behavior of this instruction is that the contents of register Ra and Rb will be added together in order to produce the effective address of the load.

This means that we need to perform an ADD as our ALUFN.

We also need ASEL and BSEL equal to zero in order to pass the values of registers Ra and Rb to the ALU.

The complete dataflow through the register file, ALU, and memory is shown here.

In order to read register Rb rather than Rc, RA2SEL must be set to 0.

As we just mentioned, ASEL and BSEL are set to 0 and ALUFN is set to ADD.

The result of adding registers Ra and Rb is used as the address of the load.

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

In order to enable reading from memory, we set MWR to 0 and MOE to 1.

This sets the read/write functionality to read, and enables an output to be read from the read port of the memory.

On the beta diagram, the read data is labeled MRD, or memory read data.

The data that is read from the memory is then passed along to the register file by setting WDSEL = 2.

In order to write this result to register Rc, WERF = 1, and WASEL = 0.

So the completed control ROM for the LDX operation is shown here.

We now move on to the fourth instruction.

Here we see that the ALUFN just passes operand B through the register file.

We also see that WERF is dependent on the value of Z.

This means that the instruction that corresponds to this row is MVZC which moves a constant into register Rc if the contents of register Ra = 0.

The way this instruction works is that BSEL = 1 in order to pass the constant through as the B operand, and ALUFN = B to pass that constant through the ALU.

WDSEL = 1 so that the output of the ALU is fed back as the write value for the register file.

Because WDSEL = 1 and not 2, we know that the data coming out of the memory will be ignored so MOE can be a don't care.

Of course, MWR still must be set to 0 in order to ensure that we don't write any random values into our memory.

RA2SEL is also a don't care because we don't care whether Register Rb or Register Rc are passed through as the second read argument of the register file, RD2.

The reason we don't care is because the BSEL = 1 will ignore the RD2 value and pass through the constant that comes directly from the instruction after sign extending it.

ASEL is also a don't care because the ALU will ignore the A input when ALUFN = B.

WASEL must be 0 so that the result of the operation is written into register Rc.

Finally, PCSEL = 0 to load PC + 4 into the PC register so that the next instruction will get fetched after this one.

We are now on the last row of our control ROM.

We know that this row must correspond to our third added instruction which is STR, or store relative.

Recall that this instruction writes the contents of register Rc into memory at the address that is computed by the effective address line.

The effective address for this instruction is PC + 4 + 4 * SEXT(C).

The extra adder, just under the instruction memory, is used to calculate PC + 4 + 4 * SEXT(C).

This value is then fed to the ALU via the A operand by setting ASEL = 1.

Setting ALUFN = A passes this value as the output of the ALU in order to be used as the memory address.

This is the address that the store will write to in memory.

The value that will be written to this address in memory is the contents of register Rc.

Register Rc is fed through the register file by setting RA2SEL = 1.

This makes RD2 have the contents of register Rc.

This value then becomes the MWD, or memory write data which is the data that will be stored in the memory address that was produced by the ALU.

In order to enable writing to the memory, MWR must be set to 1.

Since WERF = 0, nothing can be written to the register file.

This means that the value of WDSEL and WASEL are don't cares since the register file won't be affected regardless of their values.

Finally, the PC is incremented by 4 to fetch the next instruction.

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