We're on the home stretch now.

For all the instructions up until now, the next instruction has come from the location following the current instruction - hence the "PC+4" logic.

Branches and JMPs change that by altering the value in the PC.

The JMP instruction simply takes the value in the RA register and makes it the next PC value.

The PCSEL MUX in the upper left-hand corner lets the control logic select the source of the next PC value.

When PCSEL is 0, the incremented PC value is chosen.

When PCSEL is 2, the value of the RA register is chosen.

We'll see how the other inputs to the PCSEL MUX are used in just a moment.

The JMP and branch instructions also cause the address of the following instruction, i.e., the PC+4 value, to be written to the RC register.

When WDSEL is 0, the "0" input of the WDSEL MUX is used to select the PC+4 value as the write-back data.

Here's how the data flow works.

The output of the PC+4 adder is is routed to the register file and WERF is set to 1 to enable that value to be written at the end of the cycle.

Meanwhile, the value of RA register coming out of the register file is connected to the "2" input of the PCSEL MUX.

So setting PCSEL to 2 will select the value in the RA register as the next value for the PC.

The rest of the control signals are "don't cares", except, of course for the memory write enable (MWR), which can never be "don't care" lest we cause an accidental write to some memory location.

The branch instruction requires an additional adder to compute the target address by adding the scaled offset from the instruction's literal field to the current PC+4 value.

Remember that we scale the offset by a factor of 4 to convert it from the word offset stored in the literal to the byte offset required for the PC.

The output of the offset adder becomes the "1" input to the PCSEL MUX, where, if the branch is taken, it will

become the next value of the PC.

Note that multiplying by 4 is easily accomplished by shifting the literal two bits to the left, which inserts two 0-bits at the low-order end of the value.

And, like before, the sign-extension just requires replicating bit ID[15], in this case fourteen times.

So implementing this complicated-looking expression requires care in wiring up the input to the offset adder, but no additional logic!

We do need some logic to determine if we should branch or not.

The 32-bit NOR gate connected to the first read port of the register file tests the value of the RA register.

The NOR's output Z will be 1 if all the bits of the RA register value are 0, and 0 otherwise.

The Z value can be used by the control logic to determine the correct value for PCSEL.

If Z indicates the branch is taken, PCSEL will be 1 and the output of the offset adder becomes the next value of the PC.

If the branch is not taken, PCSEL will be 0 and execution will continue with the next instruction at PC+4.

As in the JMP instruction, the PC+4 value is routed to the register file to be written into the RC register at end of the cycle.

Meanwhile, the value of Z is computed from the value of the RA register while the branch offset adder computes the address of the branch target.

The output of the offset adder is routed to the PCSEL MUX where the value of the 3-bit PCSEL control signal, computed by the control logic based on Z, determines whether the next PC value is the branch target or the PC+4 value.

The remaining control signals are unused and set to their default "don't care" values.

We have one last instruction to introduce: the LDR or load-relative instruction.

LDR behaves like a normal LD instruction except that the memory address is taken from the branch offset adder.

Why would it be useful to load a value from a location near the LDR instruction?

Normally such addresses would refer to the neighboring instructions, so why would we want to load the binary encoding of an instruction into a register to be used as data?

The use case for LDR is accessing large constants that have to be stored in main memory because they are too large to fit into the 16-bit literal field of an instruction.

In the example shown here, the compiled code needs to load the constant 123456.

So it uses an LDR instruction that refers to a nearby location C1: that has been initialized with the required value.

Since this read-only constant is part of the program, it makes sense to store it with the instructions for the program, usually just after the code for a procedure.

Note that we have to be careful to place the storage location so that it won't be executed as an instruction!

To route the output of the offset adder to the main memory address port, we'll add ASEL MUX so we can select either the RA register value (when ASEL=0) or the output of the offset adder (when ASEL=1) as the first ALU operand.

For LDR, ASEL will be 1, and we'll then ask the ALU compute the Boolean operation "A", i.e., the boolean function whose output is just the value of the first operand.

This value then appears on the ALU output, which is connected to the main memory address port and the remainder of the execution proceeds just like it did for LD.

This seems a bit complicated!

Mr. Blue has a good question: why not just put the ASEL MUX on the wire leading to the main memory address port and bypass the ALU altogether?

The answer has to do with the amount of time needed to compute the memory address.

If we moved the ASEL MUX here, the data flow for LD and ST addresses would then pass through two MUXes, the BSEL MUX and the ASEL MUX, slowing down the arrival of the address by a small amount.

This may not seem like a big deal, but the additional time would have to be added the clock period, thus slowing down every instruction by a little bit.

When executing billions of instructions, a little extra time on each instruction really impacts the overall performance of the processor.

By placing the ASEL MUX where we did, its propagation delay overlaps that of the BSEL MUX, so the increased functionality it provides comes with no cost in performance.

Here's the data flow for the LDR instruction.

The output of the offset adder is routed through the ASEL MUX to the ALU.

The ALU performs the Boolean computation "A" and the result becomes the address for main memory.

The returning data is routed through the WDSEL MUX so it can be written into the RC register at the end of the cycle.

The remaining control values are given their usual default values.