Our first task is to work on the datapath logic needed to execute ALU instructions with two register operands.

Each instruction requires the same processing steps: Fetch, where the 32-bit encoded instruction is read from main memory from the location specified by the program counter (PC).

Decode, where the opcode field (instruction bits [31:26]) is used to determine the values for the datapath control signals.

Read, where the contents of the registers specified by the RA and RB fields (instruction bits [20:16] and [15:11]) are read from the register file.

Execute, where the requested operation is performed on the two operand values.

We'll also need to compute the next value for the PC.

And Write-back, where the result of the operation is written to the register file in the register specified by the RC field (instruction bits [25:21]).

The system's clock signal is connected to the register file and the PC register.

At the rising edge of the clock, the new values computed during the Execute phase are written to these registers.

The rising clock edge thus marks the end of execution for the current instruction and the beginning of execution for the next instruction.

The period of the clock, i.e., the time between rising clock edges, needs to be long enough to accommodate the cumulative propagation delay of the logic that implements the 5 steps described here.

Since one instruction is executed each clock cycle, the frequency of the clock tells us the rate at which instructions are executed.

If the clock period was 10ns, the clock frequency would be 100 MHz and our Beta would be executing instructions at 100 MIPS!

Here's a sketch showing the hardware needed for the Fetch and Decode steps.

The current value of the PC register is routed to main memory as the address of the instruction to be fetched.

For ALU instructions, the address of the next instruction is simply the address of the current instruction plus 4.

There's an adder dedicated to performing the "PC+4" computation and that value is routed back to be used as the next value of the PC.

We've also included a MUX used to select the initial value for the PC when the RESET signal is 1.

After the memory propagation delay, the instruction bits (ID[31:0]) are available and the processing steps can begin.

Some of the instruction fields can be used directly as-is.

To determine the values for other control signals, we'll need some logic that computes their values from the bits of the opcode field.

Now let's fill in the datapath logic needed to execute ALU instructions with two register operands.

The instruction bits for the 5-bit RA, RB and RC fields can be connected directly to the appropriate address inputs of the register file.

The RA and RB fields supply the addresses for the two read ports and the RC field supplies the address for the write port.

The outputs of the read data ports are routed to the inputs of the ALU to serve as the two operands.

The ALUFN control signals tell the ALU what operation to perform.

These control signals are determined by the control logic from the 6-bit opcode field.

For specificity, let's assume that the control logic is implemented using a read-only memory (ROM), where the opcode bits are used as the ROM's address and the ROM's outputs are the control signals.

Since there are 6 opcode bits, we'll need 2^6 = 64 locations in the ROM.

We'll program the contents of the ROM to supply the correct control signal values for each of the 64 possible opcodes.

The output of the ALU is routed back to the write data port of the register file, to be written into the RC register at the end of the cycle.

We'll need another control signal, WERF ("write-enable register file"), that should have the value 1 when we want to write into the RC register.

Let me introduce you to Werf, the 6.004 mascot, who, of course, is named after her favorite control signal, which she's constantly mentioning.

Let's follow the flow of data as we execute the ALU instruction.

After the instruction has been fetched, supplying the RA and RB instruction fields, the RA and RB register values appear on the read data ports of the register file.

The control logic has decoded the opcode bits and supplied the appropriate ALU function code.

You can find a listing of the possible function codes in the upper right-hand corner of the Beta Diagram handout.

The result of the ALU's computation is sent back to the register file to be written into the RC register.

Of course, we'll need to set WERF to 1 to enable the write.

5.oo Here we see one of the major advantages of a reduced-instruction set computer architecture: the datapath logic required for execution is very straightforward!

The other form of ALU instructions uses a constant as the second ALU operand.

The 32-bit operand is formed by sign-extending the 16-bit two's complement constant stored in the literal field (bits [15:0]) of the instruction.

In order to select the sign-extended constant as the second operand, we added a MUX to the datapath.

When its BSEL control signal is 0, the output of the register file is selected as the operand.

When BSEL is 1, the sign-extended constant is selected as the operand.

The rest of the datapath logic is the same as before.

Note that no logic gates are needed to perform sign-extension - it's all done with wiring!

To sign-extend a two's complement number, we just need to replicate the high-order, or sign, bit as many times as necessary.

You might find it useful to review the discussion of two's complement in Lecture 1 of Part 1 of the course.

So to form a 32-bit operand from a 16-bit constant, we just replicate it's high-order bit (ID[15]) sixteen times as we make the connection to the BSEL MUX.

During execution of ALU-with-constant instructions, the flow of data is much as it was before.

The one difference is that the control logic sets the BSEL control signal to 1, selecting the sign-extended constant as the second ALU operand.

As before, the control logic generates the appropriate ALU function code and the output of the ALU is routed to the register file to be written back to the RC register.

Amazingly, this datapath is sufficient to execute most of the instructions in the Beta ISA!

We just have the memory and branch instruction left to implement.

That's our next task.