Today we're going to describe the datapath and control logic needed to execute Beta instructions.

In an upcoming lab assignment, we'll ask you to build a working implementation using our standard cell library.

When you're done, you'll have designed and debugged a 32-bit reduced-instruction set computer! Not bad…

Before tackling a design task, it's useful to understand the goals for the design.

Functionality, of course; in our case the correct execution of instructions from the Beta ISA.

But there are other goals we should think about.

An obvious goal is to maximize performance, as measured by the number of instructions executed per second.

This is usually expressed in MIPS, an acronym for "Millions of Instructions Per Second".

When the Intel 8080 was introduced in 1974, it executed instructions at 0.29 MIPS or 290,000 instructions per second as measured by the Dhrystone benchmark.

Modern multi-core processors are rated between 10,000 and 100,000 MIPS.

Another goal might be to minimize the manufacturing cost, which in integrated circuit manufacturing is proportional to the size of the circuit.

Or we might want have the best performance for a given price.

In our increasingly mobile world, the best performance per watt might be an important goal.

One of the interesting challenges in computer engineering is deciding exactly how to balance performance against cost and power efficiency.

Clearly the designers of the Apple Watch have a different set of design goals then the designers of high-end desktop computers.

The performance of a processor is inversely proportional to the length of time it takes to run a program.

The shorter the execution time, the higher the performance.

The execution time is determined by three factors.

First, the number of instructions in the program.

Second, the number of clock cycles our sequential circuit requires to execute a particular instruction.

Complex instructions, e.g., adding two values from main memory, may make a program shorter, but may also require many clock cycles to perform the necessary memory and datapath operations.

Third, the amount of time needed for each clock cycle, as determined by the propagation delay of the digital logic in the datapath.

So to increase the performance we could reduce the number of instructions to be executed.

Or we can try to minimize the number of clock cycles needed on the average to execute our instructions.

There's obviously a bit of a tradeoff between these first two options:

more computation per instruction usually means it will take more time to execute the instruction.

Or we can try to keep our logic simple, minimizing its propagation delay in the hopes of having a short clock period.

Today we'll focus on an implementation for the Beta ISA that executes one instruction every clock cycle.

The combinational paths in our circuit will be fairly long, but, as we learned in Part 1 of the course, this gives us the opportunity to use pipelining to increase our implementation's throughput.

We'll talk about the implementation of a pipelined processor in some upcoming lectures.

Here's a quick refresher on the Beta ISA.

The Beta has thirty-two 32-bit registers that hold values for use by the datapath.

The first class of ALU instructions, which have 0b10 as the top 2 bits of the opcode field, perform an operation on two register operands (Ra and Rb), storing the result back into a specified destination register (Rc).

There's a 6-bit opcode field to specify the operation and three 5-bit register fields to specify the registers to use as source and destination.

The second class of ALU instructions, which have 0b11 in the top 2 bits of the opcode, perform the same set of operations where the second operand is a constant in the range -32768 to +32767.

The operations include arithmetic operations, comparisons, boolean operations, and shifts.

In assembly language, we use a “C” suffix added to the mnemonics shown here to indicate that the second operand is a constant.

This second instruction format is also used by the instructions that access memory and change the normally sequential execution order.

The use of just two instruction formats will make it very easy to build the logic responsible for translating the encoded instructions into the signals needed to control the operation of the datapath.

In fact, we’ll be able to use many of the instruction bits as-is!

We’ll build our datapath incrementally, starting with the logic needed to perform the ALU instructions, then add additional logic to execute the memory and branch instructions.

Finally, we’ll need to add logic to handle what happens when an exception occurs and execution has to be suspended because the current instruction cannot be executed correctly.

We’ll be using the digital logic gates we learned about in Part 1 of the course.

In particular, we’ll need multi-bit registers to hold state information from one instruction to the next.

Recall that these memory elements load new values at the rising edge of the clock signal, then store that value until the next rising clock edge.

We’ll use a lot of multiplexers in our design to select between alternative values in the datapath.

The actual computations will be performed by the arithmetic and logic unit (ALU) that we designed at the end of Part 1.

It has logic to perform the arithmetic, comparison, boolean and shift operations listed on the previous slide.

It takes in two 32-bit operands and produces a 32-bit result.

And, finally, we’ll use several different memory components to implement register storage in the datapath and also for main memory, where instructions and data are stored.

You might find it useful to review the chapters on combinational and sequential logic in Part 1 of the course.

The Beta ISA specifies thirty-two 32-bit registers as part of the datapath.

These are shown as the magenta rectangles in the diagram below.

These are implemented as load-enabled registers, which have an EN signal that controls when the register is loaded with a new value.

If EN is 1, the register will be loaded from the D input at the next rising clock edge.

If EN is 0, the register is reloaded with its current value and hence its value is unchanged.

It might seem easier to add enabling logic to the clock signal, but this is almost never a good idea since any glitches in that logic might generate false edges that would cause the register to load a new value at the wrong time.

Always remember the mantra: NO GATED CLOCKS!

There are multiplexers (shown underneath the registers) that let us select a value from any of the 32 registers.

Since we need two operands for the datapath logic, there are two such multiplexers.

Their select inputs (RA1 and RA2) function as addresses, determining which register values will be selected as operands.

And, finally, there's a decoder that determines which of the 32 register load enables will be 1 based on the 5-bit WA input.

For convenience, we'll package all this functionality up into a single component called a "register file".

The register file has two read ports, which, given a 5-bit address input, deliver the selected register value on the read-data ports.

The two read ports operate independently.

They can read from different registers or, if the addresses are the same, read from the same register.

The signals on the left of the register file include a 5-bit value (WA) that selects a register to be written with the specified 32-bit write data (WD).

If the write enable signal (WE) is 1 at the rising edge of the clock (CLK) signal, the selected register will be loaded with the supplied write data.

Note that in the BETA ISA, reading from register address 31 should always produce a zero value.

The register file has internal logic to ensure that happens.

Here's a timing diagram that shows the register file in operation.

To read a value from the register file, supply a stable address input (RA) on one of read ports.

After the register file's propagation delay, the value of the selected register will appear on the corresponding read data port (RD). [CLICK]

Not surprisingly, the register file write operation is very similar to writing an ordinary D-register.

The write address (WA), write data (WD) and write enable (WE) signals must all be valid and stable for some specified setup time before the rising edge of the clock.

And must remain stable and valid for the specified hold time after the rising clock edge.

If those timing constraints are met, the register file will reliably update the value of the selected register. [CLICK]

When a register value is written at the rising clock edge, if that value is selected by a read address, the new data will appear after the propagation delay on the corresponding data port.

In other words, the read data value changes if either the read address changes or the value of the selected register changes.

Can we read and write the same register in a single clock cycle?

Yes! If the read address becomes valid at the beginning of the cycle, the old value of the register will be appear on the data port for the rest of the cycle.

Then, the write occurs at the *end* of the cycle and the new register value will be available in the next clock cycle.

Okay, that's a brief run-though of the components we'll be using.

Let's get started on the design!