Having talked about the storage resources provided by the Beta ISA, let's design the Beta instructions themselves.

This might be a good time to print a copy of the handout called the "Summary of Beta Instruction Formats" so you'll have it for handy reference.

The Beta has three types of instructions: compute instructions that perform arithmetic and logic operations on register values, load and store instructions that access values in main memory, and branch instructions that change the value of the program counter.

We'll discuss each class of instructions in turn.

In the Beta ISA, all the instruction encodings are the same size: each instruction is encoded in 32 bits and hence occupies exactly one 32-bit word in main memory.

This instruction encoding leads to simpler control-unit logic for decoding instructions.

And computing the next value of the program counter is very simple: for most instructions, the next instruction can be found in the following memory location.

We just need to add 4 to the current value of program counter to advance to the next instruction.

As we saw in Part 1 of the course, fixed-length encodings are often inefficient in the sense that the same information content (in this case, the encoded program) can be encoded using fewer bits.

To do better we would need a variable-length encoding for instructions, where frequently-occurring instructions would use a shorter encoding.

But hardware to decode variable-length instructions is complex since there may be several instructions packed into one memory word, while other instructions might require loading several memory words.

The details can be worked out, but there's a performance and energy cost associated with the more efficient encoding.

Nowadays, advances in memory technology have made memory size less of an issue and the focus is on the higher-performance needed by today's applications.

Our choice of a fixed-length encoding leads to larger code size, but keeps the hardware execution engine small and fast.

The computation performed by the Beta datapath happens in the arithmetic-and-logic unit (ALU).

We'll be using the ALU designed in Part 1 of the course.

The Beta ALU instructions have 4 instruction fields.

There's a 6-bit field specifying the ALU operation to be performed - this field is called the opcode.

The two source operands come from registers whose numbers are specified by the 5-bit "ra" and "rb" fields.

So we can specify any register from R0 to R31 as a source operand.

The destination register is specified by the 5-bit "rc" field.

This instruction format uses 21 bits of the 32-bit word, the remaining bits are unused and should be set to 0.

The diagram shows how the fields are positioned in the 32-bit word.

The choice of position for each field is somewhat arbitrary, but to keep the hardware simple, when we can we'll want to use the same field positions for similar fields in the other instruction encodings.

For example, the opcode will always be found in bits [31:26] of the instruction.

Here's the binary encoding of an ADD instruction.

The opcode for ADD is the 6-bit binary value 0b100000 - you can find the binary for each opcode in the Opcode Table in the handout mentioned before.

The "rc" field specifies that the result of the ADD will be written into R3.

And the "ra" and "rb" fields specify that the first and second source operands are R1 and R2 respectively.

So this instruction adds the 32-bit values found in R1 and R2, writing the 32-bit sum into R3.

Note that it's permissible to refer to a particular register several times in the same instruction.

So, for example, we could specify R1 as the register for both source operands AND also as the destination register.

If we did, we'd be adding R1 to R1 and writing the result back into R1, which would effectively multiply the value in R1 by 2.

Since it's tedious and error-prone to transcribe 32-bit binary values, we'll often use hexadecimal notation for the

binary representation of an instruction.

In this example, the hexadecimal notation for the encoded instruction is 0x80611000.

However, it's *much* easier if we describe the instructions using a functional notation, e.g., "ADD(r1,r2,r3)".

Here we use a symbolic name for each operation, called a mnemonic.

For this instruction the mnemonic is "ADD", followed by a parenthesized list of operands, in this case the two source operands (r1 and r2), then the destination (r3).

So we'll understand that ADD(ra,rb,rc) is shorthand for asking the Beta to compute the sum of the values in registers ra and rb, writing the result as the new value of register rc.

Here's the list of the mnemonics for all the operations supported by the Beta.

There is a detailed description of what each instruction does in the Beta Documentation handout.

Note that all these instructions use same 4-field template, differing only in the value of the opcode field.

This first step was pretty straightforward - we simply provided instruction encodings for the basic operations provided by the ALU.

Now that we have our first group of instructions, we can create a more concrete implementation sketch.

Here we see our proposed datapath.

The 5-bit "ra" and "rb" fields from the instruction are used to select which of the 32 registers will be used for the two operands.

Note that register 31 isn't actually a read/write register, it's just the 32-bit constant 0, so that selecting R31 as an operand results in using the value 0.

The 5-bit "rc" field from the instruction selects which register will be written with the result from the ALU.

Not shown is the hardware needed to translate the instruction opcode to the appropriate ALU function code - perhaps a 64-location ROM could be used to perform the translation by table lookup.

The program counter logic supports simple sequential execution of instructions.

It's a 32-bit register whose value is updated at the end of each instruction by adding 4 to its current value.

This means the next instruction will come from the memory location following the one that holds the current instruction.

In this diagram we see one of the benefits of a RISC architecture: there's not much logic needed to decode the instruction to produce the signals needed to control the datapath.

In fact, many of the instruction fields are used as-is!