Finally, let's discuss the third class of instructions that let us change the program counter.

Up until now, the program counter has simply been incremented by 4 at the end of each instruction, so that the next instruction comes from the memory location that immediately follows the location that held the current instruction,

i.e., the Beta has been executing instructions sequentially from memory.

But in many programs, such as in factorial, we need to disrupt sequential execution, either to loop back to repeat some earlier instruction, or to skip over instructions because of some data dependency.

We need a way to change the program counter based on data values generated by the program's execution.

In the factorial example, as long as b is not equal to 0, we need to keep executing the instructions that calculate a*b and decrement b.

So we need instructions to test the value of b after it's been decremented and if it's non-zero, change the PC to repeat the loop one more time.

Changing the PC depending on some condition is implemented by a branch instruction, and the operation is referred to as a "conditional branch".

When the branch is taken, the PC is changed and execution is restarted at the new location, which is called the branch target.

If the branch is not taken, the PC is incremented by 4 and execution continues with the instruction following the branch.

As the name implies, a branch instruction represents a potential fork in the execution sequence.

We'll use branches to implement many different types of control structures: loops, conditionals, procedure calls, etc.

Branches instructions also use the instruction format with the 16-bit signed constant.

The operation of the branch instructions are a bit complicated, so let's walk through their operation step-by-step.

Let's start by looking at the operation of the BEQ instruction.

First the usual PC 4 calculation is performed, giving us the address of the instruction following the BEQ.

This value is written to the "rc" register whether or not the branch is taken.

This feature of branches is pretty handy and we'll use it to implement procedure calls a couple of lectures from now.

Note that if we don't need to remember the PC 4 value, we can specify R31 as the "rc" register.

Next, BEQ tests the value of the "ra" register to see if it's equal to 0.

If it is equal to 0, the branch is taken and the PC is incremented by the amount specified in the constant field of the instruction.

Actually the constant, called an offset since we're using it to offset the PC, is treated as a word offset and is multiplied by 4 to convert it a byte offset since the PC uses byte addressing.

If the contents of the "ra" register is not equal to 0, the PC is incremented by 4 and execution continues with the instruction following the BEQ.

Let me say a few more words about the offset.

The branches are using what's referred to as "pc-relative addressing".

That means the address of the branch target is specified relative to the address of the branch, or, actually, relative to the address of the instruction following the branch.

So an offset of 0 would refer to the instruction following the branch and an offset of -1 would refer to the branch itself.

Negative offsets are called "backwards branches" and are usually seen at branches used at the end of loops, where the looping condition is tested and we branch backwards to the beginning of the loop if another iteration is called for.

Positive offsets are called "forward branches" and are usually seen in code for "if statements", where we might skip over some part of the program if a condition is not true.

We can use BEQ to implement a so-called unconditional branch, i.e., a branch that is always taken.

If we test R31 to see if it's 0, that's always true, so BEQ(R31,...) would always branch to the specified target.

There's also a BNE instruction, identical to BEQ in its operation except the sense of the condition is reversed: the branch is taken if the value of register "ra" is non-zero.

It might seem that only testing for zero/non-zero doesn't let us do everything we might want to do.

For example, how would we branch if "a b"?

That's where the compare instructions come in - they do more complicated comparisons, producing a non-zero value if the comparison is true and a zero value if the comparison is false.

Then we can use BEQ and BNE to test the result of the comparison and branch appropriately.

At long last we're finally in a position to write Beta code to compute factorial using the iterative algorithm shown in C code on the left.

In the Beta code, the loop starts at the second instruction and is marked with the "L:" label.

The body of the loop consists of the required multiplication and the decrement of b.

Then, in the fourth instruction, b is tested and, if it's non-zero, the BNE will branch back to the instruction with the label L. Note that in symbolic notation for BEQ and BNE instructions we don't write the offset directly since directly that would be a pain to calculate and would change if we added or removed instructions from the loop. Instead we reference the instruction to which we want to branch, and the program that translates the symbolic code into the binary instruction fields will do the offset calculation for us. There's a satisfying similarity between the Beta code and the operations specified by the high-level FSM we created for

computing factorial in the simple programmable datapath discussed earlier in this lecture.

In this example, each

state in the high-level FSM matches up nicely with a particular Beta instruction.

We wouldn't expect that high degree of correspondence in general, but since our Beta datapath and the example datapath were very similar, the states and instructions match up pretty well.