One last bit of housekeeping, then we're done!

What should our hardware do if for some reason an instruction can't be executed?

For example, if a programming error has led to trying to execute some piece of data as an instruction and the opcode field doesn't correspond to a Beta instruction (a so-called "illop" or illegal operation).

Or maybe the memory address is larger than the actual amount main memory.

Or maybe one of the operand values is not acceptable, e.g., if the B operand for a DIV instruction is 0.

In modern computers, the accepted strategy is cease execution of the running program and transfer control to some error handler code.

The error handler might store the program state onto disk for later debugging.

Or, for an unimplemented but legal opcode, it might emulate the missing instruction in software and resume execution as if the instruction had been implemented in hardware!

There's also the need to deal with external events, like those associated with input and output.

Here we'd like to interrupt the execution of the current program, run some code to deal with the external event, then resume execution as if the interrupt had never happened.

To deal with these cases, we'll add hardware to treat exceptions like forced procedure calls to special code to handle the situation,

arranging to save the PC+4 value of the interrupted program so that the handler can resume execution if it wishes.

This is a very powerful feature since it allows us to transfer control to software to handle most any circumstance beyond the capability of our modest hardware.

As we'll see in Part 3 of the course, the exception hardware will be our key to interfacing running programs to the operating system (OS) and to allow the OS to deal with external events without any awareness on the part of the running program.

So our plan is to interrupt the running program, acting like the current instruction was actually a procedure call to the handler code.

When it finishes execution, the handler can, if appropriate, use the normal procedure return sequence to resume execution of the user program.

We'll use the term "exception" to refer to exceptions caused by executing the current program.

Such exceptions are "synchronous" in the sense that they are triggered by executing a particular instruction.

In other words, if the program was re-run with the same data, the same exception would occur.

We'll use the term "interrupt" to refer to asynchronous exceptions resulting from external events whose timing is unrelated to the currently running program.

The implementation for both types of exceptions is the same.

When an exception is detected, the Beta hardware will behave as if the current instruction was a taken BR to either location 0x4 (for synchronous exceptions) or location 0x8 (for asynchronous interrupts).

Presumably the instructions in those locations will jump to the entry points of the appropriate handler routines.

We'll save the PC+4 value of the interrupted program into R30, a register dedicated to that purpose.

We'll call that register XP ("exception pointer") to remind ourselves of how we're using it.

Since interrupts in particular can happen at any point during a program's execution, thus overwriting the contents of XP at any time, user programs can't use the XP register to hold values since those values might disappear at any moment!

Here's how this scheme works:

suppose we don't include hardware to implement the DIV instruction, so it's treated as an illegal opcode.

The exception hardware forces a procedure call to location 0x4, which then branches to the Illop handler shown here.

The PC+4 value of the DIV instruction has been saved in the XP register, so the handler can fetch the illegal instruction and, if it can, emulate its operation in software.

When handler is complete, it can resume execution of the original program at the instruction following DIV by performing a JMP(XP).

Pretty neat!

To handle exceptions, we only need a few simple changes to the datapath.

We've added a MUX controlled by the WASEL signal to choose the write-back address for the register file.

When WASEL is 1, write-back will occur to the XP register, i.e., register 30.

When WASEL is 0, write-back will occur normally, i.e., to the register specified by the RC field of the current instruction.

The remaining two inputs of the PCSEL MUX are set to the constant addresses for the exception handlers.

In our case, 0x4 for illegal operations, and 0x8 for interrupts.

Here's the flow of control during an exception.

The PC+4 value for the interrupted instruction is routed through the WDSEL MUX to be written into the XP register.

Meanwhile the control logic chooses either 3 or 4 as the value of PCSEL to select the appropriate next instruction that will initiate the handling the exception.

The remaining control signals are forced to their "don't care" values, since we no longer care about completing execution of the instruction we had fetched from main memory at the beginning of the cycle.

Note that the interrupted instruction has not been executed.

So if the exception handler wishes to execute the interrupted instruction, it will have to subtract 4 from the value in the XP register before performing a JMP(XP) to resume execution of the interrupted program.