This Survival Guide is the result of reading the posts for the Hardware Chapters, and addressing issues that came up multiple times.

Project Files

There are .zip files for each chapter/project that can be found in the Project# pages in the Study Plan. These .zip files contain stub files for all the HDL files you need to build, and test scripts and data required to test your HDL implementations.

There are also Lecture slides on the Study Plan page. Click on the icon in the "Lecture" column to get either Power Point slides or a PDF file containing the slides.

Implementation Order

It is **very important** to understand that the project files contain stub HDL files with no implementation. This means that if you implement Xor in terms of And and Or but have not implemented the And and Or chips, your test will fail even if your implementation is correct.

If you implement your chips in the order presented in section 1.2 you will not run into this problem. Every chip in 1.2 can be constructed using only chips presented earlier in section 1.2.

A recommended alternative is to create a subdirectory called stubs and move all the HDL stub files into that directory. You can move them into your working directory as needed.

Note that your HDL file and its associated TST file must be in the same directory. This is because the test script loads the HDL file it is testing from the test script's directory. If you load your HDL into the simulator and then load the test script from a different directory, the test script will load and test a different HDL file. (Proper usage is to load either an HDL file or a TST file, not both.)

HDL Syntax Errors

The Hardware Simulator displays errors on the status bar at the bottom of its window. On some computers with small screens these messages are off the bottom of the screen and are not visible. If you load your HDL and nothing show up in the HDL window in the simulator and you don't see an error message, this is probably what's happening.

Your computer should have a way to move the window to see the message using the keyboard. For example, on Windows use Alt+Space, M, and the arrow keys.

Unconnected Pins

The Hardware Simulator does not consider unconnected pins to be errors. It defaults any unconnected input or output to be *false* (0). This can cause mysterious errors in your chips.

If the output of your chip is always 0, check that your chip's output is connected to the output of one of the parts you are using to implement your chip. Double-check the name of the wires involved in the chip's output. Typographic errors are particularly bad here since the Hardware Simulator doesn't throw errors on disconnected wires. I found this out implementing my FullAdder in chapter two when I had

```
SomeChip(..., sum=sun);
```

And the simulator happily made a wire named *sun* and my chip's *sum* output was always 0.

If the output of a part in your implementation does not appear to be working correctly, check that all of its inputs are connected to something.

Warren Toomey made a **list of all the built-in chips prototypes** in this post (http://tecs-questions-and-answers-forum.32033.n3.nabble.com/Built-in-Chip-Templates-td129514.html). This is an extremely useful post to have bookmarked or printed.

Canonical Representation

The book introduces you to the canonical representation of a Boolean function. This representation can be very useful for chips with small numbers of inputs. As the number of inputs grows, the complexity of the canonical representation grows exponentially.

The canonical representation of Mux has 4 three-variable terms; that of a Mux8Way would have 1024 11-variable terms. Large canonical representations can be reduced with algebra, usually by computer programs. In the case of Mux8Way it can be reduced to 8 four-input terms.

Clearly, this is not a practical approach for TECS. You need to think about how to use the chips you have already made to make the next chip (assuming you're following the recommended order). Often you need the chip that you just made.

Tests Are More Than Pass/Fail

When a chip fails the test script, don't forget to look at the output and compare values for clues to the failure. If you can't see them in the simulator (reported bug on some Macs) you can look at them in a text editor. For *chip*.tst the compare file is *chip*.cmp and the output file is *chip*.out.

If you need to, you can copy the chip.tst file to mychip.tst and change it to give you more information about your chip. Change the output file name in the *output-file* line and delete the *compare-to* line. This will cause the test to always run to completion.

You can also modify the *output-list* line to show your internal wires. The output format specifier is fairly obvious. The format letters are B=binary, D=decimal, X=hexadecimal.

Testing A Chip In Isolation

A some point you may become convinced that your chip is correct, but it is still failing the test. The problem may be with one of the chips that are used to make your new chip. The tests, especially for more complex chips, cannot guarantee that the tested chips are 100% correct.

You can test your new chip by itself, using only the built-in chips, by making a test subdirectory and copying all the *chip*.* files into it. If the new chip passes its test in isolation, there must be a subtle problem with one of your earlier chips. Copy your other chips into this test directory one by one, repeating the test, until you find the problem chip.

HDL Is Not A Programming Language

Go back to one of your chips that uses 3 or 4 parts. Reverse the order of the parts lines. *Will the chip still work?* You may be surprised that the answer is yes.

The reason that the chip still works is that HDL is a hardware *description* language. It describes the wiring connections that are needed to make the chip, not how it operates once power is applied. It makes no difference what order the parts are put into a circuit board. As long as all the parts get placed and connected together correctly, the circuit board will function.

The Hardware Simulator "applies the power" and tests how the chip functions.

An important aspect of this is that there is no such thing as an "uninitialized variable" in HDL. If a wire is connected to an output somewhere in the HDL, it can be connected to any input. This is particularly important to understand for Chapter 3.

Bit Numbering

Hardware bits are numbered from right to left, starting with 0. When a bus is carrying a number, bit n is the bit with weight 2^n.

When the book says 'sel=110', that means 'sel[2]=1, sel[1]=1 and sel[0]=0'. Read Appendix A.5.3 to learn about bus syntax.

Sub-busing

Sub-busing can only be used on buses that are named in the IN and OUT statements of an HDL file, or inputs and outputs of chips used in the PARTS section. If you need a sub-bus of an internal bus, you must create the narrower bus as an output from a chip. For example:

```
        CHIP Foo {
```

```
IN in[16];
OUT out;
PARTS:
Something16 (in=in, out=notIn);
Or8Way (in=notIn[4..11], out=out);
}
```

causes an error on the Or8Way statement. This needs to be coded as

```
Something16 (in=in, out[4..11]=notIn);
Or8Way (in=notIn, out=out);
```

Multiple Outputs

Sometimes you need more than one sub-bus connected to the output of a part. Simply add more than one "out=" connection to the part.

```
CHIP Foo {
IN in[16];
OUT out[8];
PARTS:
Not16 (in=in, out[0..7]=low8, out[8..15]=high8);
Something8 (a=low8, b=high8, out=out);
}
```

This also works if you want to use an output of a chip in further computations.

```
CHIP Foo {
IN a, b, c;
OUT out1, out2;
PARTS:
Something (a=a, b=b, out=x, out=out1);
Whatever (a=x, b=c, out=out2);
}
```