MILESTONE 6

Controller Implementation

6.1 Procedure

Fig. 6.1 illustrates the procedure for controller design.

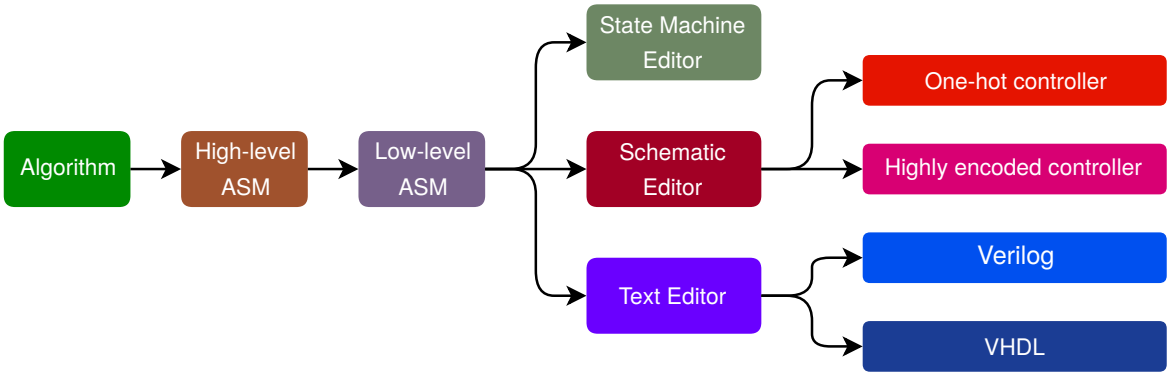

Figure 6.1: Procedure for controller design.

- **Algorithm** describes the problem at high level. This is usually given as C code fragment or pseudo-code.
- **High-level ASM** converts the algorithm to an ASM chart that closely follows the algorithm using the *Register Transfer Language (RTL)* notation. All variable names used in the algorithm are maintained.
- **Low-level ASM** replaces the RTL statements to *control* and *status* signals as defined in the datapath unit.
- The controller is finally implemented using one-hot or one of the highly encoded state assignments. For full marks, submit a implementation based on the one-hot approach. For quick prototyping and for partial marks, you can use the State Machine Editor. The highly encoded state assignment is not recommended because the large number of inputs to the next state logic.

6.2 High-Level ASM

The high-level ASM for the naive multiplier is given in Fig. 6.2.

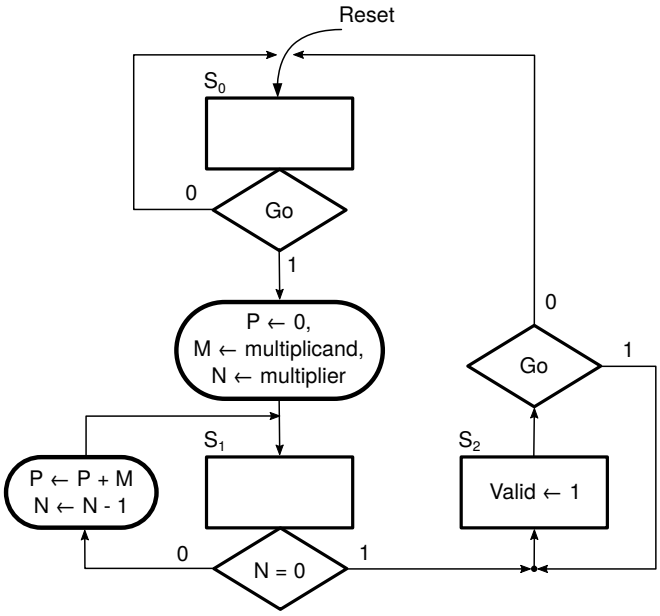

Figure 6.2: High-level ASM for naïve multiplier.

Convert the high-level ASM chart to the low-level version first.

6.3 Low-Level ASM

From the low-level ASM chart, you can complete the design directly using one-hot method or you can experiment with State Machine Editor first. The low-level ASM for the naive multiplier is given in Fig. 6.3.

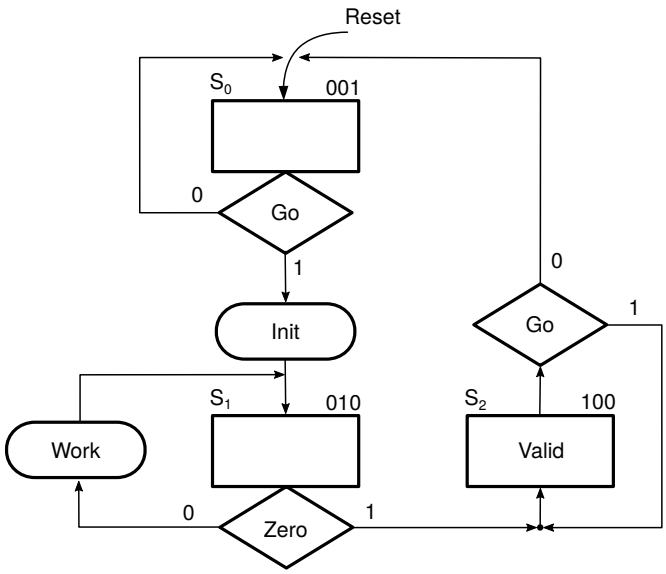

Figure 6.3: Low-level ASM for naïve multiplier.

From the low-level ASM chart, you can complete the design directly or you can experiment with State Machine Editor for first.

Based on the low-level ASM, the next state equations are:

$$S_0^+ = S_0 \cdot Go' + S_2 \cdot Go'$$
$$S_1^+ = S_1 \cdot Zero' + S_0 \cdot Go$$
$$S_2^+ = S_1 \cdot Zero + S_2 \cdot Go$$

The three outputs of the controller are:

$$Init = S_0 \cdot Go$$
$$Work = S_1 \cdot Zero'$$
$$Valid = S_2$$

The controller can now be entered using the schematic editor. The result is shown Fig. 6.4.

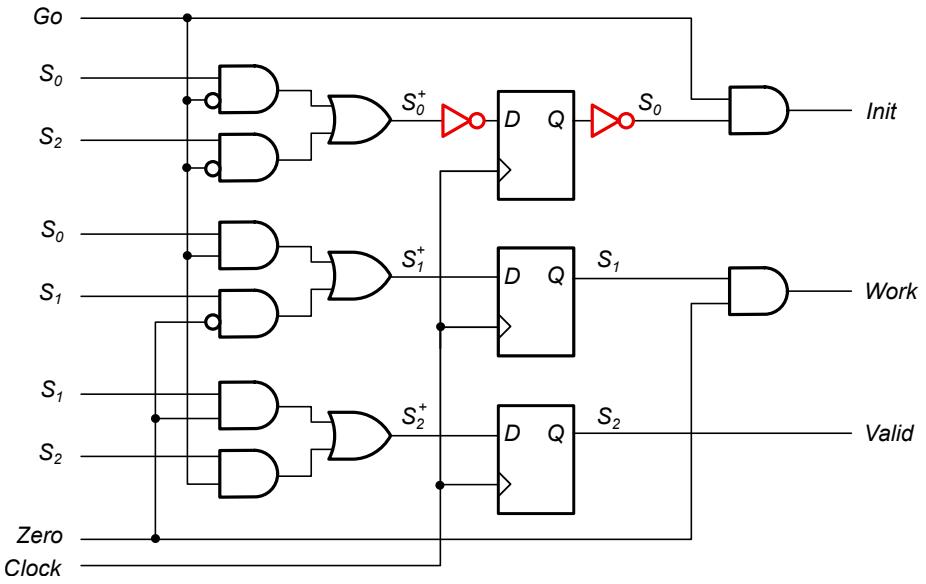

Figure 6.4: Controller for naïve multiplier. The inverted first flip-flop method is used to automatically start the one-hot controller upon power-on-reset.

6.4 System Integration

Create a symbol for the controller. Then create the actual top level entity in Quartus where you can import both the controller and datapath units. The top level entity should now look like the one in Fig. 6.5.

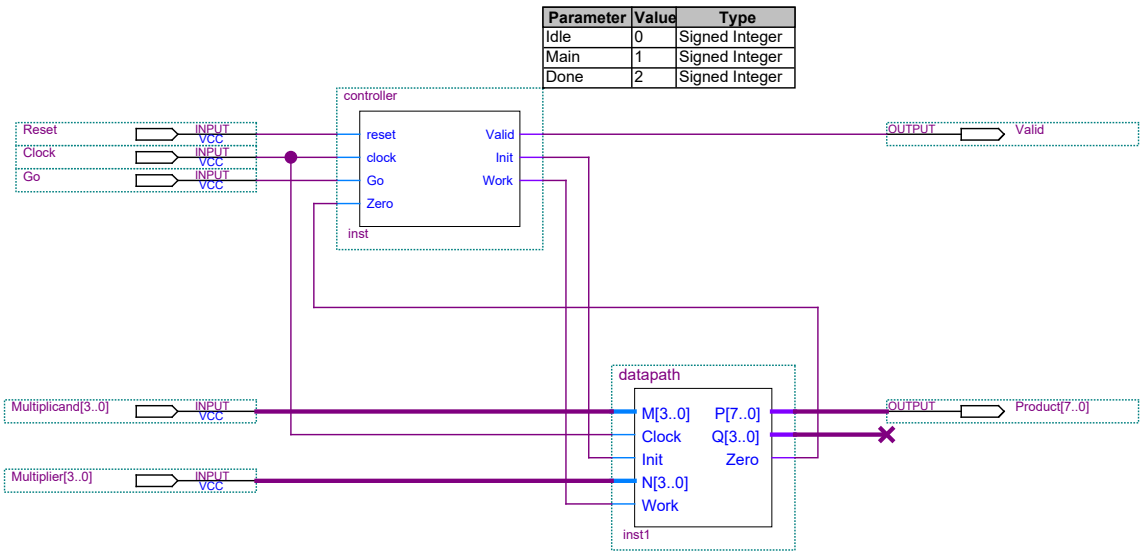

Figure 6.5: Top level entity for the project.

Test it by giving some numbers to multiply such as 0×0, 1×1, 2×5, and 15×15. In the waveform editor, put the numbers you want to multiply in the Multiplicand and Multiplier bus signals, then give a short high pulse on the Go signal to load the data into the datapath and start the multiplication.

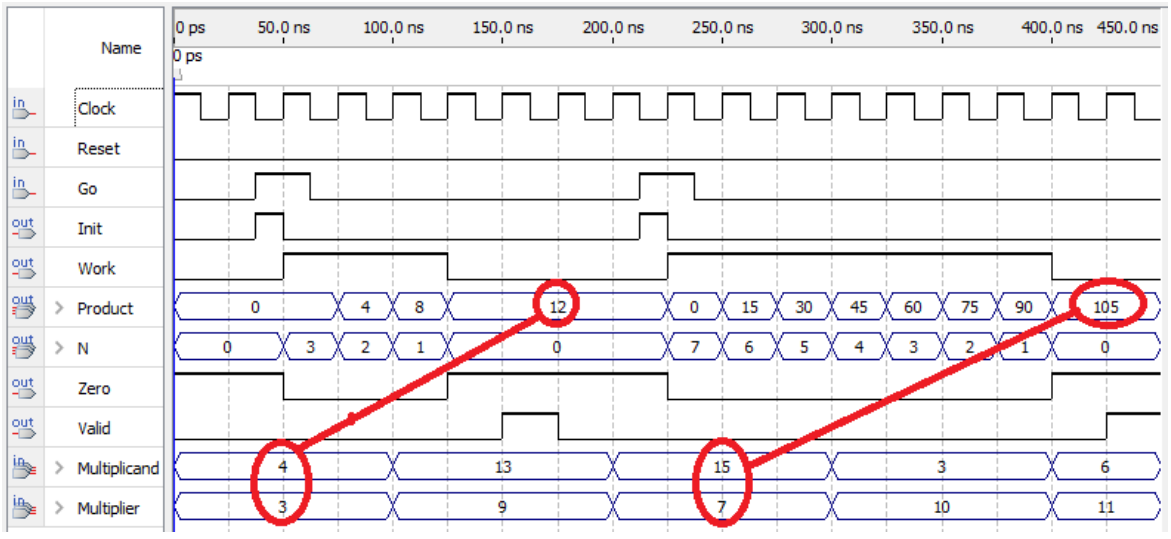

Figure 6.6: Simulation of 4×3 and 15×7 at the module level. Annotations helps the reader look at the important parts of the waveform.

6.5 Demonstration

After the simulations have shown the circuit to work correctly, add the necessary input and output modules such as in Fig. 6.7. The number of DIP switches and LED displays depends on the bit width used in your particular circuit.

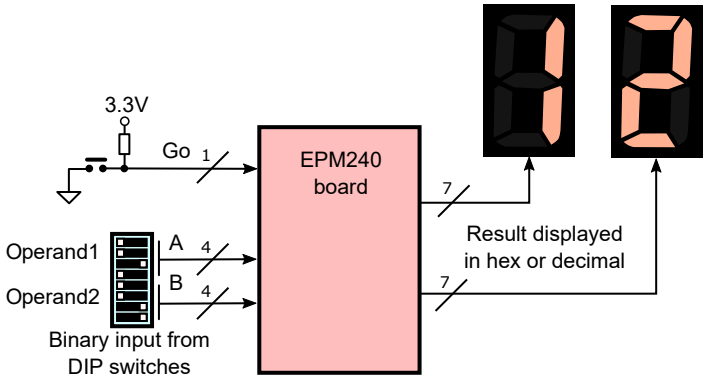

Figure 6.7: Demonstrating the CU+DU with decimal output. Shown here, $A = 4, B = 3$ and $P = 4 \times 3 = 12$.

To demonstrate, enter the operands using the DIP switches, and press *Go*. After some processing, the output should be visible on the displays. You may want to use a slow clock so that intermediate results can be made visible.

You can also use the hexadecimal number system to display the output. For example, the biggest value generated by the multiplier is $15 \times 15 = 225$ which requires three digits in decimal, but only two digits in hexadecimal. However, you must replace the 7447 decimal-to-seven-segment-display code converter with your own converter.