Structural Hints for the TayMag MkI Washer

In designing this circuit, I broke it into a number of parts:

TayMagMkI.v – this is the TLDE for the project, and as such, must have all inputs and outputs that will be used by the board. It's the primary interface between the controller, and the displays and such.

washer_control.v – this is the only device instantiated by the TLDE, and it is responsible for actually instantiating all other pieces. It's largely a copy of the signal set of the TLDE, but contains actual instantiations of other modules.

I have identified 13 stages in the sequence, as follows:

```
/* Map of Stages

0 - IDLE, advance on start button
1 - PREWASH, filling with user-selected water temperature - advance on FULL sensor
2 - WASH, agitating - advance on wash_done signal
3 - POSTWASH, draining - advance on EMPTY sensor
4 - PRERINSE, filling with cold water - advance on FULL sensor
5 - RINSE, agitating - advance on rinse_done signal
6 - POSTRINSE, draining - advance on EMPTY sensor (if no extra_rinse, go to 10)
7 - PREXRINSE, filling with cold water - advance on FULL sensor
8 - XRINSE, agitating, advance on rinse_done signal
9 - POSTXRINSE, draining - advance on EMPTY sensor
10 - SPIN, spinning - advance on spin_done signal
11 - POST_SPIN, draining - advance on EMPTY sensor
12 - DONE, sound alert - no advance, wait for RESET

*/
```

The washer control unit instatiates a valve control unit, a cycle counter, an advance MUX, a wash decoder, a binary-to-dual-7-segment display driver, and a timer display MUX (which allows multiple timers to have access to the same set of digits in the display.) It also instantiates three timers, for wash, rinse and spin.

I use a vector of advancement signals which is connected to the advance MUX, which tells it when to move to the next stage. For example, stage 0 advances on the **start** signal, whereas stage 1 advances on the **full** signal, etc. For every stage, there is a signal assignment that defines when it can move to the next stage – except for the final stage (12), which requires the **reset** signal to reset the unit.

The timer display MUX is defined as an "inner module" inside this module; its purpose is to route the appropriate timer data (from each digit) to the timer bus, which will be used to actually make the digital display show the countdowns. It uses behavioural Verilog, and on certain stages (where washing, rinsing or spinning is being done) routes the display data from the appropriate timer. Its default state is to route a predefined "null" bus, which causes the displays to be blank in any stage that doesn't require a timer.

A cycle counter (**cycle_counter.v**) provides the cycle count; it steps through all the stages. If reset is pressed, it resets to stage 0; likewise, if stage 12 completes, it goes to reset. If there is no extra rinse, then it skips the stages associated with extra rinse. Otherwise, when triggered, it increments the cycle count. It takes inputs of "next" (advance), "extra" (extra rinse switch) and "reset" (reset button) and provides a 4-bit bus representing the cycle.

A wash decoder (**wash_decoder.v**) takes the cycle count (the stage values 0 to 12), and returns a vector called the "activity selector"; it is also used to assign the output signals which drive the components in the washer (valves, cold override, agitate, spin, pump and alert.) It is essentially a 4:16 decoder, with the last 3 lines unused, making it a 4:13 decoder. It is expressed behaviourally.

The advance MUX (**advance_mux.v**) takes the cycle count (a 4-bit selector) as well as the assigned advance signal vector from the washer control, and behaviourally assigns an output "next" based on the selected advancement signal.

The valve control (**valve_control.v**) unit is nothing more than combinational logic; it uses dataflow Verilog to assign which valve or valves are open. It accepts as input the cold, lukewarm and boiling selection inputs, as well as a valve_open command signal, and a cold_override signal. When valve_open is asserted, it provides two outputs (cold_valve_open and hot_valve_open), the values of which depend on the three selection inputs together with the override input. Note that only one of cold, lukewarm and boiling will be asserted at any time; if cold_override is asserted, then only the cold valve will open.

The binary to dual seven segment decoder (**BINtoDual7Seg.v**) takes a four-bit value and displays its corresponding decimal value on two 7-segment displays. It uses positive logic (0 is off, 1 is on) – this will be inverted by the TLDE before being routed to physical hardware (that's why we use a TLDE, to modify any signals as needed before presentation to the physical hardware.) It can format the decimal equivalents of one nybble value (range 0..15) for routing to two 7-segment displays.

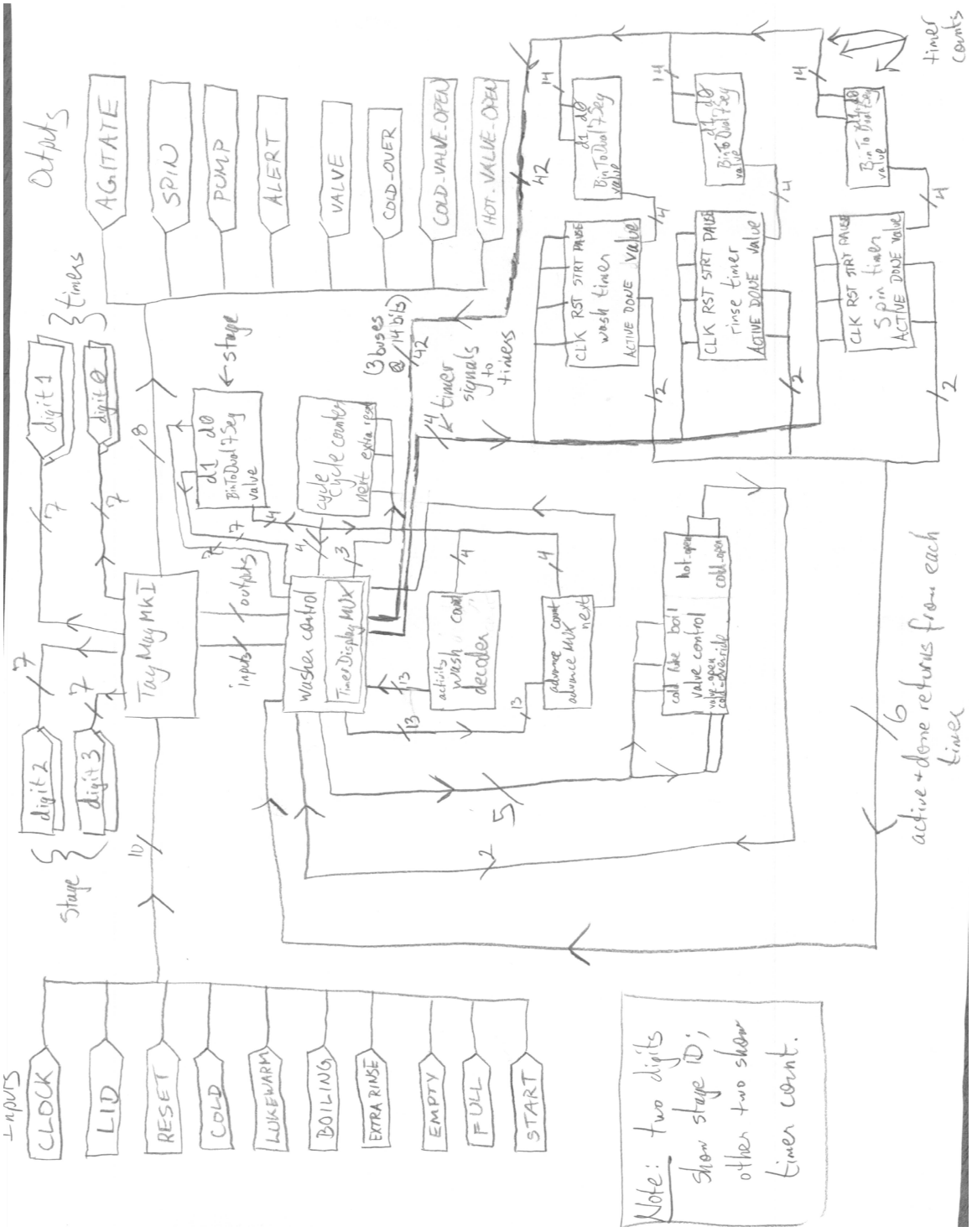
The wash timer, rinse timer and spin timer modules are precisely that which was given to you in the problem definition – the only thing differing being the module IDs (**wash_timer.v**, **rinse_timer.v** and **spin_timer.v**) and the differing starting count values (according to the timer type.) Yes, this could have been done as a single module with a configurable starting count, but to match the problem description, there are three modules. They accept a CLOCK signal (which comes from the onboard 50MHz clock of the board), master_reset, start and pause signals. They provide active and done signals, as well as the signals to drive the 7-segment displays (using the binary to dual seven segment decoder, defined above.)

Summary

The accompanying diagram sketch (not a fully-baked diagram, just a rough sketch of the concept) looks complex; however, you must remember that everything expressed there is a *parameter* in Verilog. This is why I strongly (waving arms) suggest you use Verilog, and not schematic capture, because that diagram would actually look a lot like this, and would be *ugly – very ugly.* There is a reason why developers created abstraction languages instead of sticking with schematics!

This is how *I* solved the problem. You may find a different solution, and that's just fine. I can see inefficiencies in my development – for example, regardless of the fact that the problem specifies different timer modules, I could still have made them more parameterized, and I could have put the decoding of the timer outputs in the washer control module, bussing back only the four "value" lines for the timer count, rather than the 14 bits per module of actual display output. In fact, I could have built a timer return MUX that would take the active, done and value signals from each timer, and depending on the timer selected, pass back only 6 lines. These would be things that a later design could improve on.

I am not expecting perfection – I am expecting you to work on this with your partners, discuss the issues, and make your best effort at coming up with a working design.



Obviously, this is a scan of a hand sketch . . . but it gives you an idea of what I've described here. Each of these modules is actually not all that complex – the washer control is really the most complex. The Timer Display MUX simply selects between the 14-bit outputs of each timer, and that could have been made simpler by putting the decoder in that inner module, and passing only 4 bits back per unit . . . hindsight is always 20/20!