Design Principles for Large Systems

Introduction

So far, we have encountered lots of different basic components, such as:

- basic logic gates
- multiplexers
- adders and other arithmetic logic
- flip-flops
- counters
- shift registers
- ROMs and PLAs
- etc.

We have used different techniques to build these:

- K-maps and algebraic manipulation to minimize or reduce logic
- sequential design to avoid race hazards
- state machines (Moore machines and Mealy machines) to provide control

So, the question we might have now is, “What do we do with all these different components and techniques?”

- **The answer is that we use them to build larger systems.**

This is probably the first time you will be called upon to do real engineering – i.e. solving problems with the knowledge and tools that we already have, as well as the knowledge and tools we must acquire.

Since we are engineers, we like to have a methodical approach to design, rather than just trusting to luck. We’ll cover one such approach in the next section.
Design Approach

We can view our design approach as having three major aspects.

1. **Design from the top down, implement from the bottom up.**
   - This essentially means that we start by considering the overall specification of what we are supposed to design, such as the highest level input and output. Then, we figure out what parts or components we will need to meet the overall specification. For each of those parts, we then figure out what will be needed to meet *their* individual requirements, and so on until the component parts we have are simple enough that we can implement them without too much trouble.
   - Then, we can start implementing them at the bottom level, putting together the simplest components, then joining those together into larger ones, and those into larger ones still, until we have the whole system implemented.

2. **Design using separate control and datapath/architecture.**
   - The datapath (or architecture) is the part of the circuit that actually carries out the action required in the specification.
   - The controller (as you might imagine) controls the datapath as needed.
3. **Be aware of dependencies.**

- The specification affects every aspect of your design.
- The design of the datapath affects the controller, as do any hardware constraints (e.g. ROMs can be used or can’t, limited to 8 states only, certain components unavailable, or else surplus that must be used).
- The datapath is affected by hardware constraints to some extent.

To use these preceding principles, we need:

- clearly understood specifications.
- knowledge of available building blocks.
- time and effort (especially for the datapath).
- a language to describe the control algorithm.

- This is our next topic.
- The language is something like a flowchart that changes state every clock cycle.
- Likewise, it is sort of like a state diagram, but more descriptive in that it can handle multiple inputs and outputs, and the “signals” can be values or instructions.
- Hence, we use *ASM charts.*
Algorithmic State Machine (ASM) Charts

There are three components used in ASM charts:

1. State boxes

   ![ASM State Box Diagram]

2. Decision boxes

   ![ASM Decision Box Diagram]

3. Conditional boxes

   ![ASM Conditional Box Diagram]
With respect to the above diagrams, we define the following terms:

**Condition**: an input signal name
   The value of the signal during the state determines which exit path is taken from the decision box.

**Action**: an output signal name
   The named signal is asserted (driven high) during this state (this clock tick).

**Name**: an arbitrary name given to a state
   The name given to a state box. May have meaningful context (eg. IDLE or RESET) or may be arbitrary (STATE0 or S0).

**Code**: the binary representation of the state
   This can be assigned after the algorithm is designed. Usually use D-type flip-flops to store the state. Can try to use minimal number of flip-flops (eg. 4 states, 2 flip-flops, state codes would be 00, 01, 10, 11) or use one-hot design (one flip-flop per state – eg. 3 states, 3 flip-flops, state codes would be 001, 010, 100).

Lastly, we define the concept of an **ASM block**.

**ASM block** refers to a state box and all of the decision and conditional boxes connected to its exit path, until it reaches the next state. An ASM block represents the state of the system for exactly one (1) clock tick.
An ASM block includes exactly one state box and all decision and conditional boxes until the next state.

**Note:**
Often, the “action” signals (the outputs of the control algorithm) might be commands being sent to the datapath.
- If the command connects to an asynchronous input of a datapath component, the effect of the command will be immediate (for example, asynchronous clear of a register).
- More usually, the command will be connecting to a synchronous control input. In this case, the effect of the command will occur on the next clock edge.

**Relating State Diagrams and ASM Charts**

ASM charts can easily represent state diagrams, as seen in the next two examples.

**Example 1:** Moore machine with one input (X) and one output (Y).
Example 2: Mealy machine with one input (X) and one output (Y).

When representing Moore machines as ASM charts, the outputs are asserted in the state boxes. When represent Mealy machines, the outputs are asserted in conditional boxes.

**From ASM Chart to Circuit**

We might be wondering what good these ASM charts are. After all, they allow us to describe the control algorithm that we want, but how do we implement that algorithm? That is, how do we get a circuit from an ASM chart?

We develop a circuit from an ASM chart by following a process similar to the “classical” sequential design process you already know.

- Assign state variables.
- Write the *state sequences* (also called the state sequence table), which is like a circuit excitation table but only includes inputs that affect transitions in the current state.
- From the state sequence table, we develop the state equations by reading off the conditions for the state transition where the state variable becomes a 1.
- Once we have state equations, we can try to simplify algebraically, and implement using D-type flip-flops for the state variables, and Boolean logic (less common) or ROMs (more common) for the generation of the next state.
Example: Generating circuit for the previous Moore machine example.

<table>
<thead>
<tr>
<th>Current State</th>
<th>$B_k$</th>
<th>$A_k$</th>
<th>Next State</th>
<th>$B_{k+1}$</th>
<th>$A_{k+1}$</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>0</td>
<td>0</td>
<td>S1</td>
<td>0</td>
<td>1</td>
<td>$X'$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S2</td>
<td>1</td>
<td>0</td>
<td>$X$</td>
</tr>
<tr>
<td>S1</td>
<td>0</td>
<td>0</td>
<td>S1</td>
<td>0</td>
<td>0</td>
<td>$X'$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S0</td>
<td>0</td>
<td>0</td>
<td>$X'$</td>
</tr>
<tr>
<td>S2</td>
<td>1</td>
<td>0</td>
<td>S1</td>
<td>0</td>
<td>1</td>
<td>$X$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S2</td>
<td>1</td>
<td>0</td>
<td>$X'$</td>
</tr>
</tbody>
</table>

To generate the state equations for each state variable, we look for wherever the next value of that variable is a 1. We then take the corresponding condition, AND it with the state, and that becomes part of our state equation.

So $B_{k+1} = S0 \cdot X + S2 \cdot X'$ and $A_{k+1} = S0 \cdot X' + S1 \cdot X + S2 \cdot X$, in both cases a function of the current state and the current input. This is what allows us to handle multiple inputs and outputs.

Next, we replace the state names with the corresponding values of the state variables. Thus, our final state equations are:

$B_{k+1} = B_k \cdot A_k' \cdot X + B_k \cdot A_k \cdot X'$

$A_{k+1} = B_k \cdot A_k' \cdot X' + B_k \cdot A_k \cdot X + B_k \cdot A_k \cdot X$

These can be algebraically simplified (at least a little bit) and we can implement with basic logic or (as is more often the case) ROMs.
Design Example: Traffic Light Controller

Consider a 4-way intersection.

Traffic analysis shows that a much greater amount of traffic travels along the East-West road than the North-South. No one ever seems to make a turn, so there is no need to install turn signals. There is no need to account for pedestrians crossing the intersection. And everyone who uses the intersection is an ideal driver, so we don’t need to worry about leaving a few seconds after a yellow light before turning the other direction green, or anything like that.

The traffic planners have come up with the following control scheme, using a sensor to detect if cars are waiting at a red light in the North-South direction. Note that when one direction is green or yellow, obviously the other direction is red.

Starting as the N-S lights turn green:
1. The N-S lights stay green for 20 seconds. Proceed to step 2.
2. The N-S lights turn yellow for 7 seconds. Proceed to step 3.
3. The N-S lights turn red for 20 seconds. If a car is waiting, the N-S lights stay red for an additional 7 seconds, then turn green (go back to step 1). Otherwise, the lights stay red until either a car arrives and waits or 180 seconds have passed (whichever comes first), then stay red for an additional 7 seconds, then turn green (go back to step 1).

Starting as the E-W lights turn red:
1. The E-W lights stay red for 20 + 7 seconds. Proceed to step 2.
2. The E-W lights turn green for 20 seconds. If there is a car waiting at that point, proceed to step 3. If no car is waiting then, the light stays green until either a car shows up and waits, or 180 seconds have passed (whichever comes first), then proceed to step 3.
3. The E-W lights turn yellow for 7 seconds. Go back to step 1.

The traffic planners want you to design a controller for this intersection, that meets the above specification. The lights and sensor are already installed. The sensor asserts signal CAR_WAITING when it detects a car waiting at the N-W intersection. The lights are controlled by the signals NS_GREEN, NS_YELLOW, NS_RED, EW_GREEN, EW_YELLOW, and
EW_RED, which are self-explanatory. Currently, only the lights and the sensor comprise the datapath, but you may have to add some more components to complete your design.

Consider: What will my controller need to know?

- From the specification, we notice that certain actions take place after a certain amount of time – 7 seconds yellow, 20 seconds for a green or red, of 180 seconds for an extended green.
- This suggests a timer might exist in the datapath. We can create a timer using just a counter and a 1 Hz clock.
- So the controller would need to be told when those times have passed: TIME_7SEC, TIME_20SEC, and TIME_180SEC will be signals generated by the datapath and sent to the controller.
- The controller will need to be able to start the timer, so START_TIMER will be a signal generated by the controller and sent to the datapath.

Thus, we can do a top-level system diagram.
Consider the datapath:

We are told the traffic lights are directly controlled by the six self-explanatory control signals. (In that respect, they’re sort of like the LEDs on the lab board). Thus, no design is needed to deal with this aspect of the datapath.

Similarly, the sensor circuit is pre-installed and simply provides us with just the input signal CAR_WAITING. Thus, no design is needed.

As we concluded above, the datapath needs to count seconds and inform the controller. This obviously leads us to use a counter.

- We need to count up to 180 seconds, therefore we need an 8-bit counter. (We can either assume we have access to an 8-bit counter, or build it out of 2 4-bit counters as on page 476 of your text).
- We use a 1 Hz clock signal, so that the counter will count seconds.
- We need to generate signals for 7 seconds, 20 seconds, and 180 seconds.
- We will use the START_TIMER command from the controller to load zeros into the counter.
Generate control signals as follows:

Note:
Often, a design may call for components like an 8-bit shift register or 8-bit counter, or some other component that does a common operation but that you might not be sure whether it exists or not in the form you need. In the real world, you would be expected to do research and find suitable components with which to implement your design. In a test or quiz, it is generally permissible to assume such a component exists, so long as you assume reasonable I/O. For example, like above, I might assume that I have an 8-bit counter with parallel load. Reasonably, it should have a clock input, a clear input, and a load enable input, as well as the parallel data lines for loading. So, on a test, I could just assume the 8-bit counter exists, and complete the question using that. But in real life, I’d either have to find such a counter, or build it from two smaller ones.

Now consider the controller:

We know that when the lights in one direction are green or yellow, the lights in the other direction are red. Thus, we should concern ourselves with the behaviour of the green and yellow states, rather than the red states.

So, we can see five obvious states: NS_GREEN, NS_YELLOW, EW_GREEN, EW_GREEN_EXT, and EW_YELLOW. EW_GREEN_EXT is used to represent the extended green in the East-West direction. (For this example, we will assume that we start automatically in state 000, so we won’t worry about unused states).

See the ASM chart for the traffic light controller on the next page.
Call the state variables $C_k$, $B_k$, and $A_k$.

<table>
<thead>
<tr>
<th>Current state</th>
<th>$C_k$</th>
<th>$B_k$</th>
<th>$A_k$</th>
<th>Next state</th>
<th>$C_{k+1}$</th>
<th>$B_{k+1}$</th>
<th>$A_{k+1}$</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS_GREEN</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NS_YELLOW</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>TIME_20SEC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NS_GREEN</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>TIME_20SEC’</td>
</tr>
<tr>
<td>NS_YELLOW</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>NS_YELLOW</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>TIME_7SEC’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EW_GREEN</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>TIME_7SEC</td>
</tr>
<tr>
<td>EW_GREEN</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>EW_GREEN</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>TIME_20SEC’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EW_YELLOW</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>TIME_20SEC·CAR_WAITING</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EW_GREEN_EXT</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>TIME_20SEC·CAR_WAITING’</td>
</tr>
<tr>
<td>EW_YELLOW</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>EW_YELLOW</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>TIME_7SEC’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NS_GREEN</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>TIME_7SEC</td>
</tr>
<tr>
<td>EW_GREEN_EXT</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>EW_GREEN_EXT</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>TIME_180SEC·CAR_WAITING’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EW_YELLOW</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>TIME_180SEC·CAR_WAITING</td>
</tr>
</tbody>
</table>

So we write out the state equations, and replace the states with their variables:

$$C_{k+1} = EW\_GREEN\cdot TIME\_20SEC\cdot CAR\_WAITING’ + EW\_GREEN\_EXT\cdot TIME\_180SEC’\cdot CAR\_WAITING’$$

$$= C_k\cdot B_k\cdot A_k\cdot TIME\_20SEC\cdot CAR\_WAITING’ + C_k\cdot B_k\cdot A_k\cdot TIME\_180SEC’\cdot CAR\_WAITING’$$

$$= CAR\_WAITING’(C_k\cdot B_k\cdot A_k\cdot TIME\_20SEC + C_k\cdot B_k\cdot A_k\cdot TIME\_180SEC’$$

$$B_{k+1} = NS\_YELLOW\cdot TIME\_7SEC + EW\_GREEN\cdot TIME\_20SEC’ + EW\_GREEN\_EXT\cdot TIME\_180SEC + EW\_GREEN\_EXT\cdot TIME\_180SEC + CAR\_WAITING$$

$$= C_k\cdot B_k\cdot A_k\cdot TIME\_7SEC + C_k\cdot B_k\cdot A_k\cdot TIME\_20SEC’ + C_k\cdot B_k\cdot A_k\cdot TIME\_20SEC\cdot CAR\_WAITING + C_k\cdot B_k\cdot A_k\cdot TIME\_7SEC + C_k\cdot B_k\cdot A_k\cdot TIME\_180SEC + CAR\_WAITING)$$
A_{k+1} = NS\_GREEN\cdot TIME\_20SEC + NS\_YELLOW\cdot TIME\_7SEC' + EW\_GREEN\cdot TIME\_20SEC\cdot CAR\_WAITING + EW\_YELLOW\cdot TIME\_7SEC' + EW\_GREEN\_EXT\cdot (TIME\_180SEC + CAR\_WAITING)

= C_k'\cdot B_k'\cdot A_k'\cdot TIME\_20SEC + C_k'\cdot B_k'\cdot A_k'\cdot TIME\_7SEC' + C_k'\cdot B_k'\cdot (A_k' + A_k) + C_k'\cdot (TIME\_180SEC + CAR\_WAITING)

The outputs are easily generated, since all but one of them is dependent only on the state. In the following assignments, the name to the left of the “=” sign is the output name; the names to the right are state names.

\[
\begin{align*}
\text{NS\_GREEN} &= \text{NS\_GREEN} = C_k'\cdot B_k'\cdot A_k' \\
\text{NS\_YELLOW} &= \text{NS\_YELLOW} = C_k'\cdot B_k'\cdot A_k \\
\text{NS\_RED} &= \text{EW\_GREEN} + \text{EW\_YELLOW} + \text{EW\_GREEN\_EXT} \\
&= C_k'\cdot B_k'\cdot A_k' + C_k'\cdot B_k'\cdot A_k + C_k'\cdot B_k'\cdot A_k' \\
&= C_k'\cdot B_k'\cdot (A_k' + A_k) + C_k'\cdot (C_k'\cdot B_k' + C_k'\cdot B_k') \\
\text{EW\_GREEN} &= \text{EW\_GREEN} + \text{EW\_GREEN\_EXT} \\
&= C_k'\cdot B_k'\cdot A_k' + C_k'\cdot B_k'\cdot A_k' = A_k'\cdot (C_k'\cdot B_k' + C_k'\cdot B_k') \\
\text{EW\_YELLOW} &= \text{EW\_YELLOW} = C_k'\cdot B_k'\cdot A_k \\
\text{EW\_RED} &= \text{NS\_GREEN} + \text{NS\_YELLOW} \\
&= C_k'\cdot B_k'\cdot A_k' + C_k'\cdot B_k'\cdot A_k \\
&= C_k'\cdot B_k'\cdot (A_k' + A_k) = C_k'\cdot B_k' \\
\text{START\_TIMER} &= \text{NS\_GREEN\cdot TIME\_20SEC} + \text{NS\_YELLOW\cdot TIME\_7SEC} \\
&= \text{NS\_GREEN\cdot TIME\_20SEC} + \text{NS\_YELLOW\cdot TIME\_7SEC} + \text{EW\_GREEN\_EXT\cdot (TIME\_180SEC + CAR\_WAITING)} \\
&= C_k'\cdot B_k'\cdot A_k'\cdot TIME\_20SEC + C_k'\cdot B_k'\cdot A_k'\cdot TIME\_7SEC + C_k'\cdot B_k'\cdot A_k'\cdot (TIME\_180SEC + CAR\_WAITING)
\end{align*}
\]

We can implement the state equations and outputs as a circuit, or using a ROM or PLA.