CEG 3155 Summary Sheet

Fall 2025

University of Ottawa
Owen Daigle

Contents

Basically so far it is about asm design, and some fsm design. :-/

Also, this course has a lot of processes that in my opinion are quite hard to explain. They need to be done to understand them. So I don't think putting them here would really help me learn. I will just put a very high level description of what was done.

1 Miscellaneous Topics

Really just a review of ITI1100 and CEG2136

Some concepts that are assumed to be known for the other sections are:

- K Maps
- Function Minimization
- Multiple Output circuits (if n outputs, then do n k maps)
- Don't Cares

1.1 Big K maps

For K maps, if we want to do a K map for a 5 variable function, say $x_4x_3x_2x_1x_0$, we can always fix one variable at 1 and then subsequently 0, say x_4, and then solve two k maps one of which with $x_4 = 0$ and one with $x_4 = 1$. Then we get an equation like the following that can be simplified down potentially.

$$f = x_4(...) + \overline{x_4}(...)$$

If the K map is bigger, we often either have to rely on some hacky ways, or we simply cannot solve it using k maps and need to use computers.

1.2 Multi Level Synthesis

FPGAs often only have a certain number of inputs to each gate. They may only have AND2 and not AND4 gates. If we need to have 4 inputs into an AND gate, then we need to break it up into multiple levels. In the below example we break 1 AND4 into 3 AND2:

$$(x_1x_2x_3x_4) + x_0 \equiv ((x_1x_2)(x_3x_4)) + x_0$$

We can also decompose functions into multiple sub functions to make it simpler.

$$f = (x_1x_2) + (x_3x_4) \equiv f = g + h \text{ where } g = x_1x_2, h = x_3x_4$$

2 VHDL

VHDL is a hardware description language. It is used to model hardware.

We can built at multiple levels such as the structural level, and the behaviour level. The behavioural level is very high level and uses `if/else`, and `while` loops. It is easy to program, but it is not very efficient when it compiles it to the actual gates to be used in hardware.

At the structural level, we only use basic `AND` gates, `OR` gates, `NOT` gates, `XOR` gates, etc. We connect the wires between gates. We can create **components** which have inputs and outputs, and are also made up of gates (behaviourally coded, or structurally coded).

We have two main parts of the vhdl file, the entity (inputs and outputs) and the behaviour (what it actually does).

```
LIBRARY ieee; -- allows us to use STD_LOGIC and other stuff
USE ieee.std_logic_1164.ALL; -- see above note

ENTITY name IS
        PORT (
                X, Y, Z: IN STD_LOGIC; -- 3, 1 bit inputs
                A: IN STD_LOGIC_VECTOR(7 DOWNTO 0); -- 8 bit input
                B, C: OUT STD_LOGIC -- 1 bit output
                );
END name;

ARCHITECTURE arch_name OF name IS
BEGIN
        -- actual behaviour is modeled here
        B <= X OR Y; -- B is X or Y
        C <= Y AND Z;
END ARCHITECTURE arch_name;
```

We could also do it in behavioural

```
ARCHITECTURE arch2 OF name IS
BEGIN
        -- actual behaviour is modeled here
        test : PROCESS IS
        BEGIN
                if X='1' then
                        C <= Y;
        END PROCESS test;
END ARCHITECTURE arch2;
```

We use the keyword `SIGNAL` to define internal connections, we use `COMPONENT` for already declared entity/architecture instances, and we use `PORT␣MAP` for a component to connect signals to ports.

```
...
ARCHITECTURE struct OF name IS
        COMPONENT d_latch -- define what the port map of d_latch is.
            This has an ENTITY/ARCHITECTURE pair somewhere else
                port(d, clk: IN STD_LOGIC; q: OUT STD_LOGIC);
```

```
        END COMPONENT;

        SIGNAL int_clk, d0, q0: STD_LOGIC; --this is just an internal
            signal not mapped to outputs or inputs

BEGIN
        bit0: ENTITY work.d_latch(name)
                PORT MAP (
                        CLK => int_clk,
                        d => d0,
                        q => q0
                );
END ARCHITECTURE struct;
...
```

We typically design a testbench for each vhdl file that we make. These testbenches run through tests to ensure the component works fine.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY name IS
-- blank on purpose since no inputs or outputs in a testbench
END ENTITY name;

ARCHITECTURE test OF name IS
        signal sigs: STD_LOGIC;
BEGIN
        dut : ENTITY work.name(main)
        PORT MAP (
                -- a => a
                );

        clk_process : PROCESS IS
        BEGIN
                -- pulses clk every 2ns,
                -- assumes CLK is defined as input
                -- will run in parallel with test_process
                CLK <= '1'; WAIT FOR 1 NS; CLK <= '0'; WAIT FOR 1 NS;
        END PROCESS clk_process;

        test_process : PROCESS IS
        BEGIN
                -- test code here

                WAIT;
        END PROCESS test_process;
END ARCHITECTURE test;
```

Notation	Intended Operation	Notation	Intended Operation
$X \leftarrow Y$	Transfer from Y into X	$X \leftarrow M[1234]$	Set X to memory at 1234
$X \leftarrow 0$	Clear X	$X \leftarrow Y \vee Z$	Put Y OR Z (bitwise) into X
$X++$	Increment X	$X \leftarrow Y + Z$	Add Y and Z and put into X

Table 1: RTL Summary

VHDL is quite a powerful and complicated language, but is better explained through examples.

2.1 Other Languages

Verilog is another popular hardware description language, as is SystemC.

Verilog is a simpler language which has simpler types. It is easier to use, but may not have as efficient end designs. Verilog is also based on C so it is quite familiar.

SystemC is an extension of C++ that adds basic hardware modeling to the language. While it is not as efficient as verilog of vhdl, it integrates well with existing C++ code, especially non hardware code.

3 ASM Design

Algorithmic State Machine is a method to take a design and turn it into a circuit. We have two main parts, the datapath and the control path. The control path just simply takes in the inputs and states, and generates control signals for each state we are in. These signals are used by the datapath and the controlpath.

Some signals could be to shift a register, load a flip flop, enable a certain flip flop, etc.

The datapath is where the actual data goes. It goes in, comes out, and takes in all the control signals.

To do this, we need to know Register Transfer Language (RTL). This is a language that literally symbolizes data going into and out of registers.

We have five major steps for the ASM process:

1. Create an algorithm using pseudocode to describe the operation of the device
2. Convert the pseudocode into an ASM chart using RTL
3. Design the datapath based on the ASM chart
4. Create a detailed ASM chart based on the datapath (convert RTL to actual signals that need to be active at each stage)

5. Design control path using detailed ASM chart (We use 1 Flip Flop per state that we need)

This looks complicated, but in practice it is actually quite simple.

Ex. Example found in Appendix 11.1.

4 Arithmetic

A lot of this is review of previous courses.

To recap, we store negative numbers in the twos complement form. We can then (for addition and subtraction) treat them the same as any other positive number.

We have overflow and carry. These happen when we end up with too many bits. If we are working with signed numbers, then we only pay attention to the overflow bit. If we are working with unsigned numbers, then we pay attention to only the carry bit.

4.1 Carry Lookahead Adder

The standard adder works very well, but if we need to do like a 32 bit adder, then we need to wait for the first adder to generate a carry to do the second, and then that carry to do the third, and so on until the 32nd has to wait for the 31st to finish to get its carry.

A Carry Lookahead Adder basically uses math to tell us that we can find all the carrys quite early on. We just need to do some combinatorial logic, and then we have all the inputs we require for all the 32 adders to be able to calculate simultaneously.

Basically, we end up with a much faster adder, but it uses up much more space and power due to the additional gates for the carry calculating logic.

4.2 Multiplication

Usually multiplication is just implemented using shifting and the adder. We can however make a more complicated and efficient circuit, but just like the carry lookahead adder they use more space and power.

Dividing is also typically done using shifting and the adder.

4.3 Floating Point Arithmetic

This is best shown with an example.

Ex. Express $(36.5625)_{10}$ as a 32 bit floating point number using the IEEE standard.

The value in binary is 100100.1001 and normalized is 1.001001001×2^5.

1. Sign: 0
2. Exponent: 5+127 = 132 = 10000100

3. Mantissa: 00100100100000000000000

So the 32 bit number is: 0 10000100 00100100100000000000000 (without the spaces ofc)

5 FSM Design

Finite State Machine is a method to take a design with a finite number of states and turn it into a circuit. It is very similar to ASM, but it works better for small scale circuits, and falls flat with more complex ones.

FSM has 8 steps:

1. Obtain problem specification
2. Create a state diagram
3. Create a state table
4. Minimize the number of states if possible
5. Choose a state assignment for the states
6. Create a transition table
7. Get the design equations using k maps and the transition table
8. Implement the circuit using the equations from the previous step as the flip flop inputs

Ex. Example found in Appendix 11.2.

5.1 Mealy vs Moore

When designing these circuits, we should know the difference between a mealy and a moor machine.

A Mealy machine has outputs that depend on the current state, and the input variables. These machines have very fast response times since they can quickly change output right when the input changes.

Moore machines have outputs that **only** depend on the current state. These machines are simpler, but they have slightly slower response times, since if the inputs change, they have to wait for the next clock cycle to change state and then finally change output.

5.2 State Minimization

To minimize states, we use the class equivalence method.

1. Divide all states based on output
2. For each state in a grouping, for the next states, if they all go to the same output GREAT, if not, divide based on output
3. Repeat Step 2 for each grouping
4. Once nothing changes, stop.

We can have don't cares in the state outputs, but we need to assume they are either all 1, or all 0. This is just for the algorithm to work. We can try each and see which gives a smaller circuit.

Ex. A quick example is found in Appendix 11.2.

5.3 State Assignment

There are often better and worse ways to assign values to each state. This is a simple way to do this that may give a slightly better state assignment.

We make groupings of states for Rule 1, Rule 2, and Rule 3.

1. Two or more current states have the same next state for a given input combination
2. For any current state, and two adjacent input combinations, take the two next states.
3. Two or more current states that generate the same output for a given output. This should only be applied on **either** 0 or 1, not both

Then we need to make as many of these states as possible (with priority to Rule 1 over Rule 2/3, and Rule 2 over Rule 3) in a K map. This is a decent state assignment.

Ex. Find an efficient state assignment for this machine.

Present State	Next State		Out
	x=0	x=1	
A	B	A	0
B	A	C	1
C	D	A	0
D	D	E	1
E	C	D	1

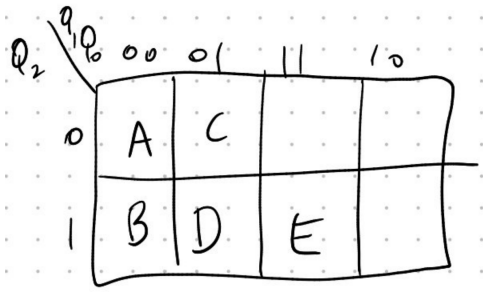
We need to apply the three rules.

Rule 1: `(C,D), (A,C)`

Rule 2: `(A,B), (A,C), (A,D), (D,E), (C,D)`

Rule 3: Select 1 - `(B,D,E)`

Now I need to make a k map and put these together so definitely Rule 1 are together, probably Rule 2, and if possible Rule 3.



So this means that we have the state assignments of: `A: 000, B: 100, C: 001, D: 101, E: 111`.

6 Asynchronous Circuit Design

Asynchronous circuits are very efficient since they do not have a clock. They do not need to wait for the next clock edge to move on, they just move on when they are ready. But the big problem is that only one bit can change at a time or bad stuff will happen.

Race conditions, encoding, and communication are just a few of the challenges with asynchronous design. So usually we stick to synchronous for simplicity and reliability.

When we have the state table, we add a new mechanism to this table. If we are in say state A, and we go to state A for input 00, then we say that state A is stable for 00. We denote this with a circle on the transition table.

	Next State			
Present State	00	01	10	11
A	(A)	(A)	B	(A)
B	(B)	A	(B)	A

Table 2: Asynchronous State Table Example

6.1 State Reduction

We can either use the standard state equivalence method, or use the new state fusion method.

The state fusion method says that two states can potentially be combined if for each input combination **one** of the following is true:

1. Both states have the same successor
2. Both states are stable
3. The successor of **one state** is undefined

Then we create a diagram of all the states, and then we can combine the ones that are adjacent **and** form a closed loop. Also, note that each state can only be in ONE loop (obviously).

Ex. We want to minimize the following machine:

Present	Next State				Output
State	00	01	10	11	z
A	(A)	F	C	-	0
B	A	(B)	-	H	1
C	G	-	(C)	D	0
D	-	F	-	(D)	1
E	G	-	(E)	D	1
F	-	(F)	-	K	0
G	(G)	B	J	-	0
H	-	L	E	(H)	1
J	G	-	(J)	-	0
K	-	B	E	(K)	1
L	A	(L)	-	K	1

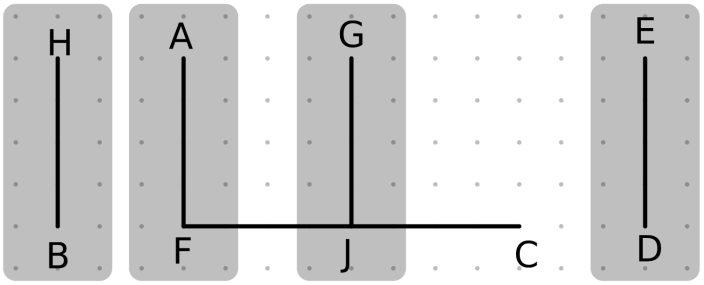
First use the state equivalence method on all these states: We get that `BL` and `HK` can be combined.

Present	Next State				Output
State	00	01	10	11	z
A	(A)	F	C	-	0
B	A	(B)	-	H	1
C	G	-	(C)	D	0
D	-	F	-	(D)	1
E	G	-	(E)	D	1
F	-	(F)	-	H	0
G	(G)	B	J	-	0
H	-	B	E	(H)	1
J	G	-	(J)	-	0

Now I need to apply the three rules for state fusion.

I see that: `(A,F), (B,H), (F,J), (C,J), (G,J), (D,E)`

Now I need to create the diagram. There are not really any closed loops in the traditional sense, but any two pair is a closed loop for sure. So we choose `BH, AF, JG, DE`, but we could just as easily pick `BH, AF, JC, DE`.



And now we reduce the table again:

Present	Next State				Output
State	00	01	10	11	z
A	(A)	(A)	C	B	0
B	A	(B)	D	(B)	1
C	G	-	(C)	D	0
D	G	A	(D)	(D)	1
G	(G)	B	(G)	-	0

Note that a lot of the undefined states now become defined. If we have two states, one is undefined and one is defined that are fused, then the undefined becomes the defined.

We can then do state fusion again. Here we will find that `CG` are compatible, so we can fuse those.

Present	Next State				Output
State	00	01	10	11	z
A	(A)	(A)	C	B	0
B	A	(B)	D	(B)	1
C	(C)	B	(C)	D	0
D	C	A	(D)	(D)	1

6.2 State Assignment

When assigning states, we cannot have any states where going from one state to the other more than one bit changes. So we could not for example go from `000` to `110`. We could however go from `000` to `010` to `110`.

To easily do this, we start by numbering all the stable states with a unique number, and then using those numbers on other states in the same column. Then, we create a transition diagram using those numbers as the transitions. Here we do not want to have any diagonals. If we have a diagonal we either have to eliminate it (by going along a different path) or add more states to account for it.

Finally, once we have an acceptable state diagram we assign states so that each adjacent state only has 1 bit changing.

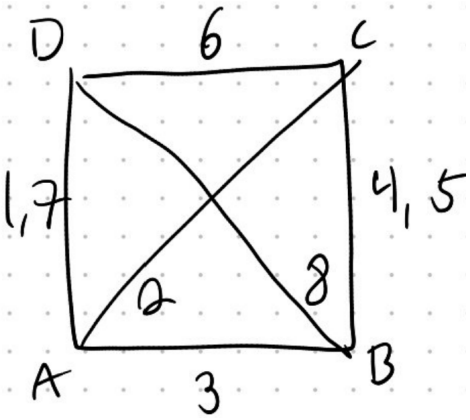
Ex. Find a state assignment for the following table:

Present State	Next State				Output z			
	00	01	10	11	00	01	10	11
A	(A)	B	D	(A)	0	-	1	1
B	C	(B)	(B)	D	0	1	0	0
C	(C)	(C)	B	A	0	1	0	1
D	A	C	(D)	(D)	-	-	1	0

I need to go through the table and assign states.

Present State	Next State				Output z			
	00	01	10	11	00	01	10	11
A	(1)	3	7	(2)	0	-	1	1
B	5	(3)	(4)	8	0	1	0	0
C	(5)	(6)	4	2	0	1	0	1
D	1	6	(7)	(8)	-	-	1	0

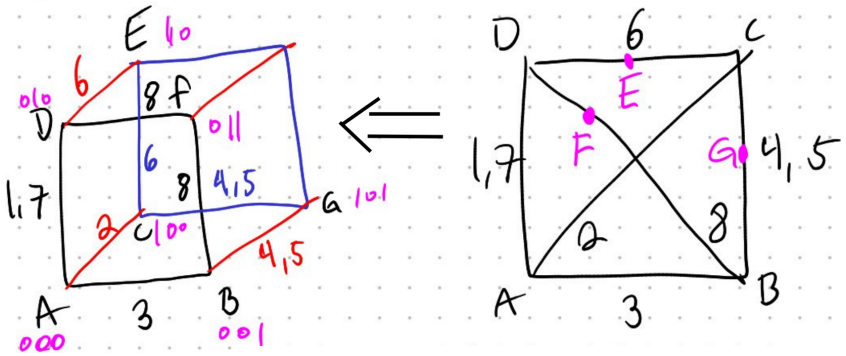
Now I go ahead and create the transition diagram.



I see that I cannot eliminate any of the diagonals by going on a different path.

Note that *if* I had for example the 2 transition on the AB and BC transition, then I could eliminate the AC transition by first going from AB, then BC.

So I need to add more states. I will add a state E, F, and G. Then I will assign states to the state diagram where each state is only



Now I can apply those 3 new states to get the new state table:

Present State	Next State				Output z			
	00	01	10	11	00	01	10	11
A	(A)	B	D	(A)	0	-	1	1
B	G	(B)	(B)	F	0	1	0	0
C	(C)	(C)	G	A	0	1	0	1
D	A	E	(D)	(D)	-	-	1	0
E	-	C	-	-	-	1	-	-
F	-	-	-	D	-	-	-	0
G	C	-	B	-	0	-	0	-

Note that for these extra states, we also have to apply the outputs. If we go from 0→0, we have 0. 1→1 we have 1, or 0→1 / 1→0, does not matter.

6.3 Hazards

Static hazards are when a signal briefly changes values and then goes back to the original value. Dynamic hazards are when a signal switches a few times before settling on a new value.

Figure 1: Hazard Types

We want to avoid both types of hazards. We can do this by trying to include more implicants in the K maps, and by using circuits with only two levels of gates.

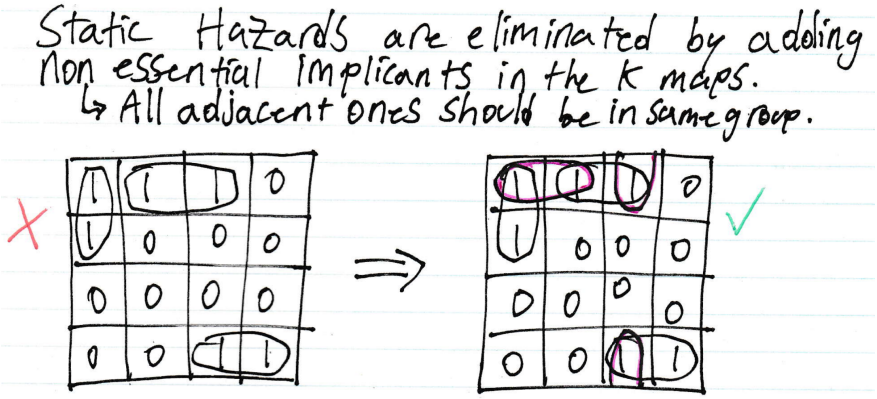


Figure 2: Avoiding Static Hazards

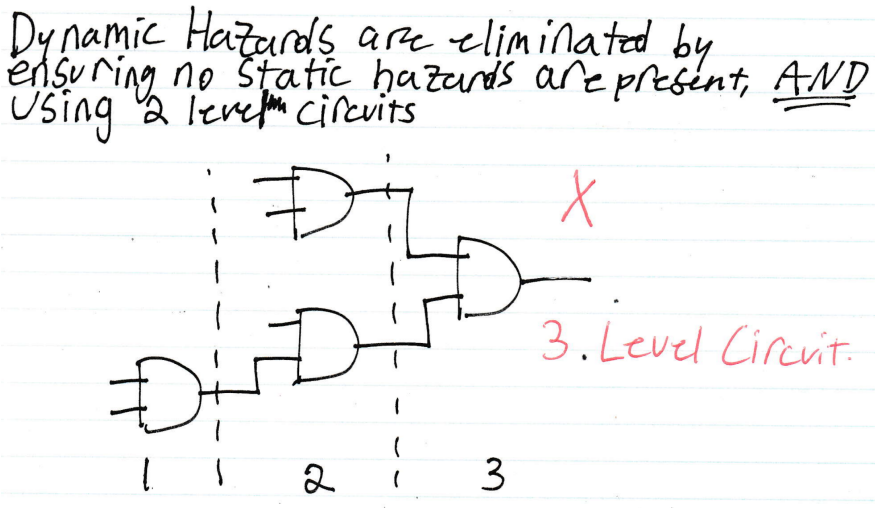


Figure 3: Avoiding Dynamic Hazards

7 Implementation Technologies

There were many different ways of implementing circuits before today's FPGAs.

7.1 Programmable Logic Array (PLA)

This is basically an AND plane and an OR plane. This lets us implement any SOP form functions. We use eFuses to enable and disable connections on these AND planes and OR planes.

7.2 Programmable Array Logic (PAL)

These are similar to PLAs, but they get rid of the programmable OR plane, and make that fixed, but they keep the AND plane. This makes the product much cheaper to make.

7.3 Simple Programmable Logic Device (SPLD)

This is a more advanced PAL where the outputs from the OR gates can be sent through a flip flop, or kept in there regular form.

7.4 Complex Programmable Logic Device (CPLD)

This is a device that contains a lot of PALs per device to make it more efficient. Each block contains something like 16 macrocell, where each macrocell contains lots of OR gates, XOR gate, D Flip Flop, and a MUX. Wires allow us to connect these blocks together.

7.5 Field Programmable Gate Array (FPGA)

FPGAs are very large in scale, and they typically use Look Up Tables (LUTs) which are basically truth tables, rather than any actual gates for the combinatorial circuits. These LUTs are buffered by an optional flip flop, and all these modules are connected together with optional connections in a matrix.

7.6 Application Specific Integrated Circuit (ASIC)

These are much more powerful and bigger than FPGAs, and they are custom chips. They are custom, and usually require very large and expensive orders to make.

8 Interfaces and Controllers

Not on exam

These are how we interface with the computer from the outside such as a keyboard/mouse, disks, and displays.

Most of this is done on a bus which allows us to easily add multiple devices to the bus. But there are bus access conflicts, and bandwidth issues since everything has to share the bus.

For busses, we have 3 main types:

- Processor Memory - Very fast and short
- Backplane - Decently fast, and standardized (PCI, or ISA)
- I/O - Slower, standardized, and lengthy (USB or SCSI)

These busses can be either synchronous, or asynchronous. Synchronous busses must be short to avoid clock skew. This is used for processor memory busses. Most busses are asynchronous since these allow many devices, and can be large. They use handshakes to coordinate data transfers.

To arbiter the bus, we can either have a centralized arbitration (needs arbiter), daisy chain arbitration (position on bus determines priority), or distributed arbitration (all devices arbiter themselves).

9 Processor Design

Not on exam

We can either have a single cycle processor (all instructions take one long clock cycle) or a multi cycle processor (instructions take different number of shorter clock cycles). Multi cycle are more efficient since faster instructions spend less time doing nothing, but they are a lot more complicated.

We have RISC (Reduced Instruction set), or CISC (Complex Instruction set) architectures, RISC is lower power, but has a slower clock, while CISC is higher power and has a higher clock. CISC has more instructions, allowing us to do more with less memory. RISC has much less instructions which need more memory to store, but they are quicker (require less clock cycles).

10 Digital Testing

Not on exam

When getting circuits fabricated at the plant, sometimes faults occur in fabrication. We need to test for these faults and if one exists, the physical piece of hardware needs to be discarded.

To test circuits, we ideally want to test all the different input combinations and see if the resulting output is correct to ensure no wire is stuck at 1, or stuck at 0.

We can be smart and reduce the number of tests we need to run by testing out each wire that exists in the system, and we can often test more wires than the sum of multiple tests through combinations. We find the minimal test set to get the coverage that we want (ideally 100% coverage).

Often we have built in tests (BIST or BILBO) to the circuits, so it does not have to be done with external equipment. These tests generate test vectors (vectors of all the inputs), send them into the circuit, and then verify the outputs. This is done by generating a signature which is checked to be valid. If invalid, the hardware has a fault.

11 Appendix

11.1 ASM Design Example

In this example I will come up with an ASM design for a weird light controller. The problem description is below:

`Design a light display controller that adheres to the following pseudocode. There will be 8 lights, and two switches. The left switch causes the LEDs to go from right to left, and the right switch causes the LEDs to go from left to right. If both are combined, then both lights are combined.`

```
DISPLAY <= 00000000; LMASK <= 00000001; RMASK <= 10000000;
if (LEFT and RIGHT)
        DISPLAY <= LMASK OR RMASK;
        LMASK <= LMASK << 1;
        RMASK <= RMASK >> 1;
else if (LEFT)
        DISPLAY <= LMASK;
        LMASK <= LMASK << 1;
else if (RIGHT)
        DISPLAY <= RMASK;
        RMASK <= RMASK >> 1;
else
        DISPLAY <= 00000000;
```

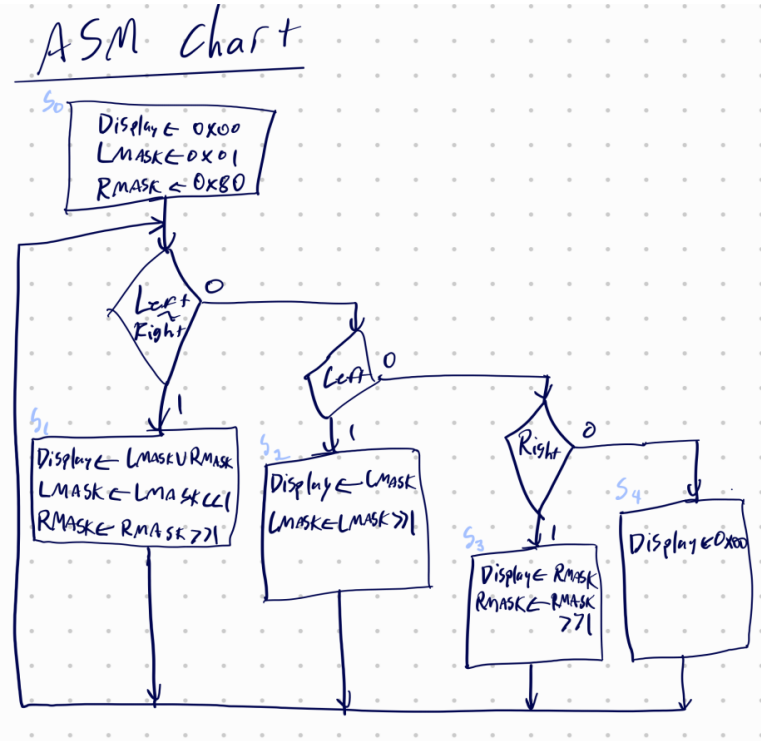


Figure 4: ASM Chart

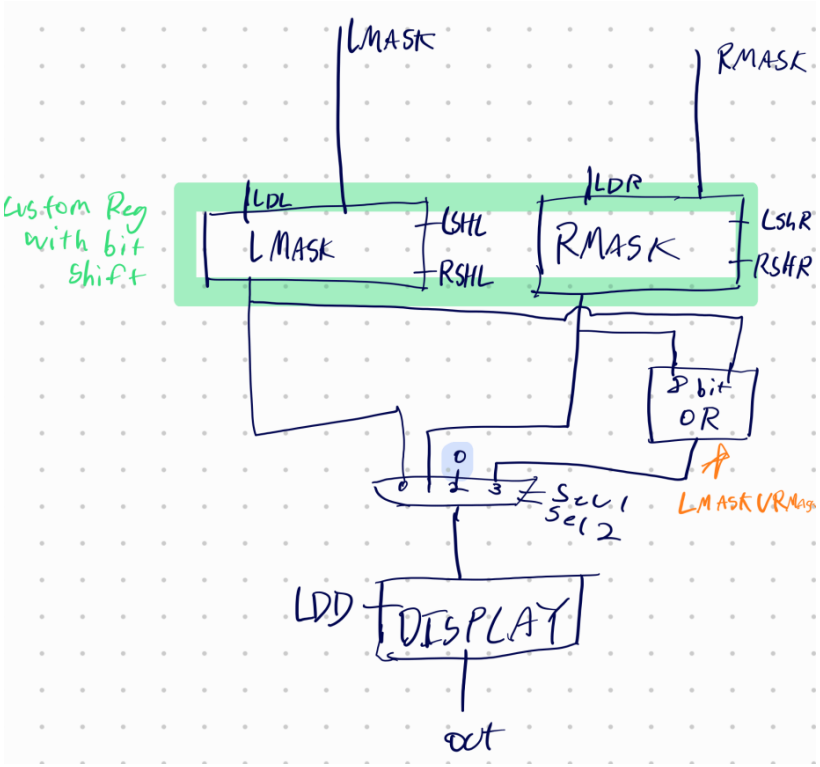


Figure 5: Datapath

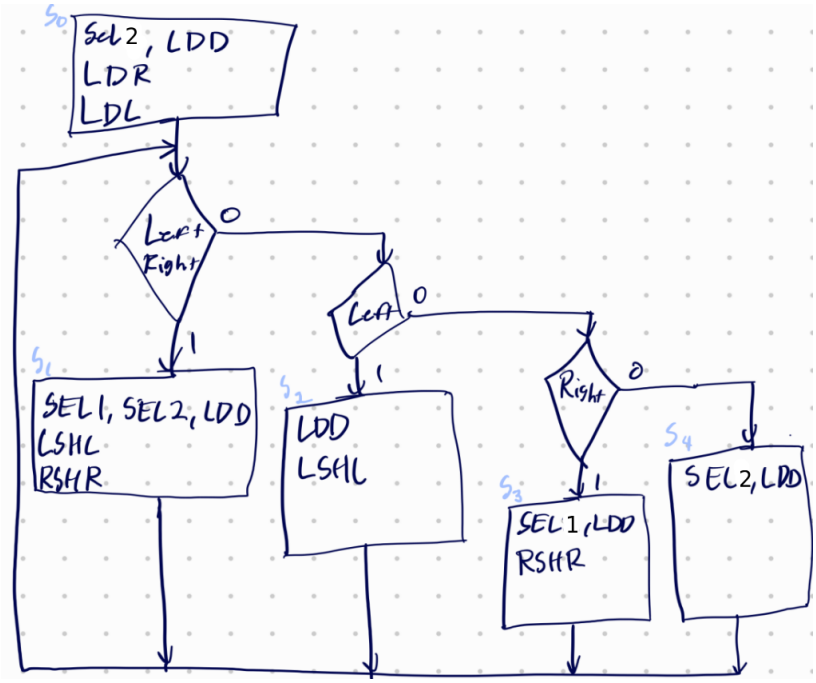


Figure 6: Detailed ASM Chart

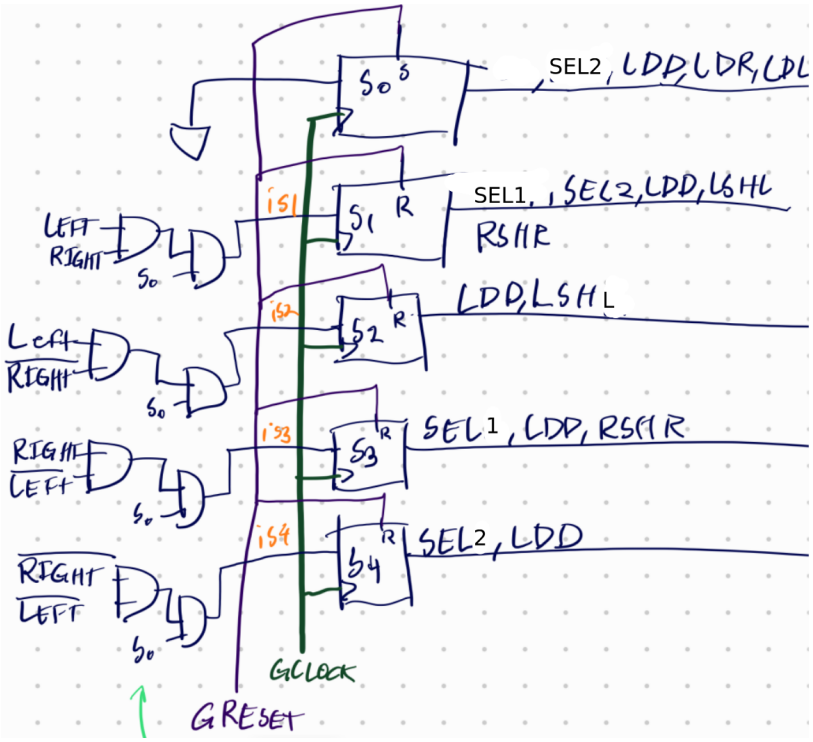


Figure 7: Control Path

11.2 FSM Design Example

In this example I will come up with an FSM design for a traffic light controller for two streets.

`Here we have two traffic lights, Main (M) and Side (S). By default the Main is Green (MG) and second is red (SR). When a car comes onto the side street sensor, then upon the timer going off, M goes to Y, and SR still. Then MR SG, then after a set time MR SY, followed by MG SR again until the timer goes off again. Then it repeats checking the sensor.`

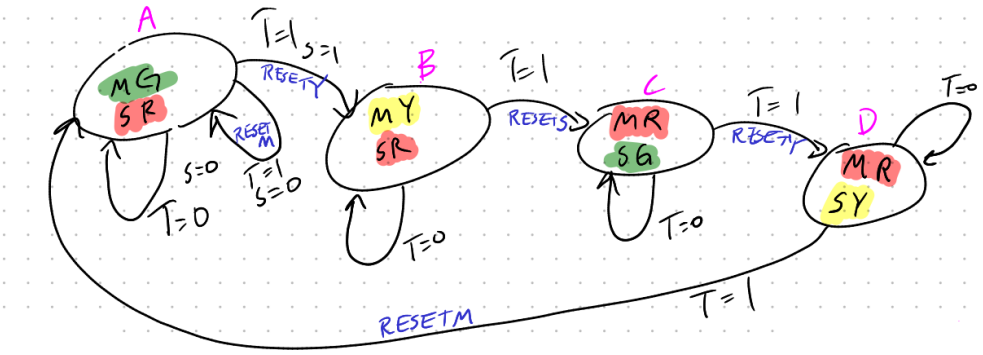


Figure 8: FSM State Diagram

Note that `T` is the timer, `S` is the sensor, `RESETY`, `RESETM`, `RESETS` represent resetting the timer for Yellow, Main light, and Secondary light respectively. Resetting involves loading the timer at the specified value such as Yellow which is hardcoded, or the main or side timer which is brought in as an input.

PS	NS T0S0	T1S0	T0S1	T1S1	Outputs M (2b)	S (2b)
A	A	A	A	B	01	10
B	B	C	B	C	01	10
C	C	D	C	D	10	01
D	D	A	D	A	10	01

Figure 9: FSM State Table

When minimizing, I end up finding that it cannot be minimized after applying the minimization algorithm.

$$(ABCD)M_2 = 1 \implies (AB)(CD)$$
$$(AB)T = 1, S = 0 \implies (AD) \text{ Since in different partitions A and B are distinct}$$

$(CD)T = 1, S = 0 \implies (DA)$ Since in different partitions, C and D are distinct
$(A)(B)(C)(D)$

Now I can assign states:

$$A = 00$$
$$B = 01$$
$$C = 11$$
$$D = 10$$

Now I can make the state transition table (Figure 10). I will use 8 k maps for this to get the output equations, and next state equations.

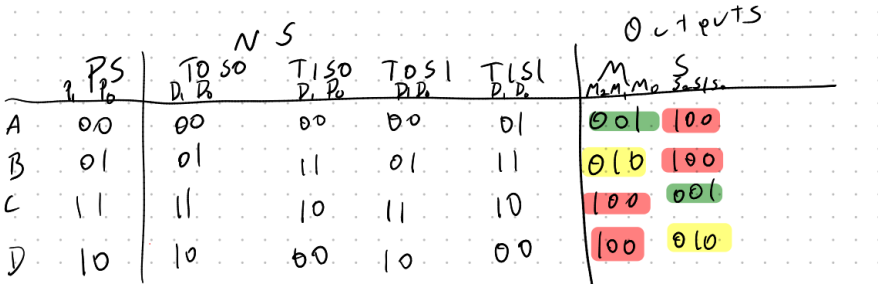


Figure 10: FSM State Transition Table

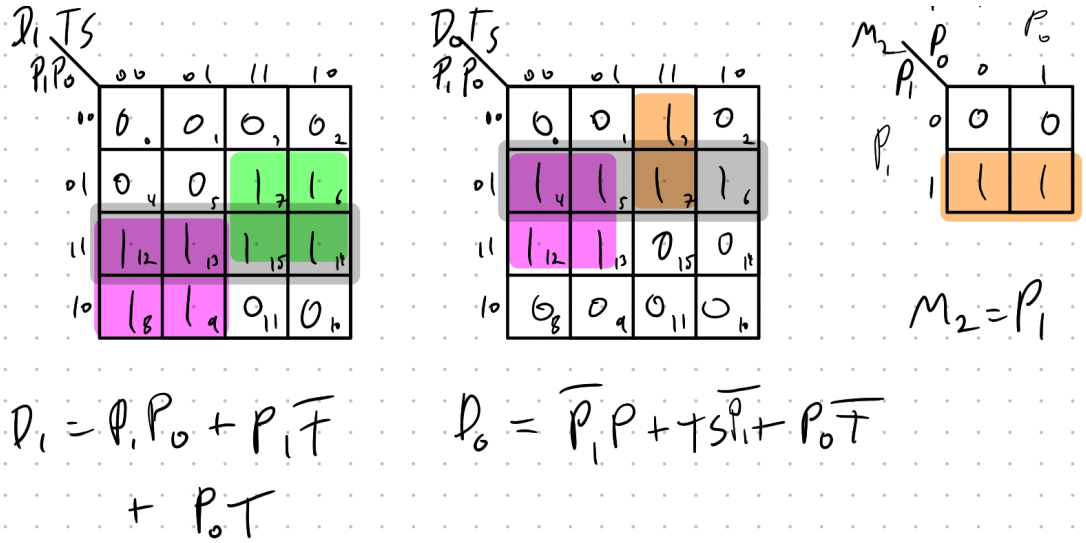


Figure 11: FSM K Maps 1

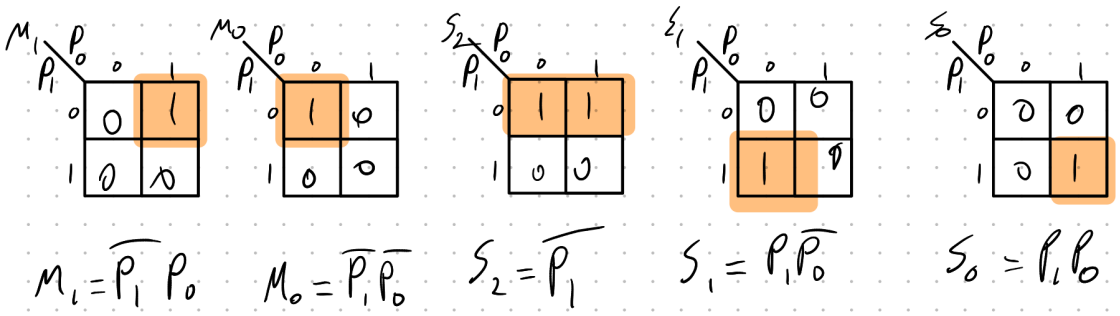


Figure 12: FSM K Maps 2

I also need the actual timer input since this will be different depending on which state we are in (as shown in Figure 8).

$$RM = (\overline{P}_1\overline{P}_0 \cdot \overline{S}) + (P_1\overline{P}_0 \cdot S) \text{ State A with sensor or State state D without}$$
$$RS = (\overline{P}_1 P_0) \text{ State B}$$
$$RY = (\overline{P}_1\overline{P}_0 \cdot S) + (P_1 P_0) \text{ State A with sensor or State state C}$$

Putting these into an equation with the actual starting inputs for the Main, Side, and Yellow timer (`MSC`, `SSC`, `YELLOWC`) in a MUX type circuit we get:

$$TIMER\_INPUT = (YELLOWC \cdot RY) + (MSC \cdot RM) + (SSC \cdot RS)$$

The load input to the counter can be given by simply `T`. This is because we only want it to load whenever the timer expires. Due to the `TIMER_INPUT` equation being the input to the timer, we will always have the correct timer input sitting at the counter input.

The final circuit is found in Figure 13. This circuit is technically not complete as it does not have the `OR` and `AND` gates for the traffic light outputs (`M2`, `M_1`, ..., `S1`, `S0`) but the idea makes sense. We just create that combinatorial logic going into the corresponding input (of state flip flops) or output (for both traffic lights). The timer input is from the above combinatorial equation of `TIMER_INPUT`.

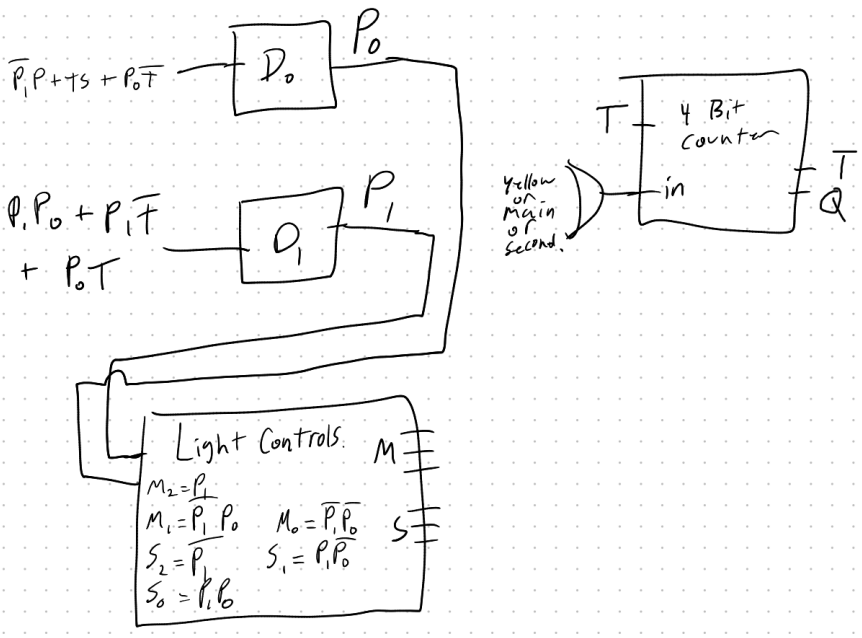


Figure 13: FSM Generated Circuit