COMS12200 Labs

Week 11: Build-a-comp decoding

Simon Hollis (simon@cs.bris.ac.uk) Document Version 1.1

Build-a-comp module lab format

This worksheet is intended to be attempted in the corresponding lab session, where there will be help available and we can give feedback on your work.

- The work represented by the lab worksheets will accumulate to form a portfolio: whether complete or not, archive everything you do via https://wwwa.fen.bris.ac.uk/COMS12200/ since it will partly form the basis for assessment.
- You are not necessarily expected to finish all work within the lab. itself. However, the emphasis is on you to catch up with anything left unfinished.
- In build-a-comp labs, you will **work in pairs** to complete the worksheets for the first part of the course. In later labs, with more complex assignments, you will work in bigger groups.
- To accommodate the number of students registered on the unit, the 3 hour lab session is split into two 1.5 hour halves.
- **You should only attend one session**, 9am-10:30am OR 10:30am-12pm. **The slot to which you have been allocated is visible on your SAFE progress page for COMS12200.**

Lab overview

Today, we're going to explore how to make control paths for computing machines, with an emphasis on how to decode states. We'll use a simple state machine example: traffic lights. We'll use several new modules.

New modules:

CLOCK

The Clock module provides you a two-phase non-overlapping clock. It works as shown in lectures, and provides you a predictable stream of clock signals. The clock also outputs a constant ENABLE signal, useful for attaching to latches.

The clock has several features of note:

1. It may be used as a power source, instead of an Input module.
2. It has two modes of operation: Manual and Automatic. In Manual mode, clock pulses are generated by the pressing of the (white) 'MANUAL' button. In Auto mode, clock pulses are generated automatically. The two modes can be switched between using the slide switch furthest away from the buttons.
3. When on automatic, there are two ways to adjust the frequency of the clock pulses: fine adjust (turn the knob) and a x1000 slide switch (nearest the buttons). You can use both to speed up and slow down the clock.
4. The clock contains a RESET (red) button. When pressed, this resets all registers with control inputs connected to the clock outputs to 0000.
5. The clock outputs are shown on green LEDs.

4:1 MULTIPLEXOR

The MUX module works by selecting between four different inputs according to a control input signal. The single output is equal to the selected input. i.e.

Input 1	**Input 2**	**Input 3**	**Input 4**	***C1***	***C0***	**Output**
A	B	C	D	0	0	A
A	B	C	D	0	1	B
A	B	C	D	1	0	C
A	B	C	D	1	1	D

1:4 DEMULTIPLEXOR

The DEMUX module works in inverse, steering a single input to one of four possible outputs. Only one output is active at any one time. i.e.

Input	**C1**	**C0**	**Output 1**	***Output 2***	***Output 3***	**Output 4**
A	0	0	A	0000	0000	0000
A	0	1	0000	A	0000	0000
A	1	0	0000	0000	A	0000
A	1	1	0000	0000	0000	A

4-way OR

The OR module simply performs a logical boolean OR operation between the first bit of each of the inputs, generating a single-bit output as the control out bit c0. The additional control bits are left unconnected, so will be pulled to the default 0 on most other module inputs.

4 main inputs are present on the bottom side, and these are passed-through to the top side outputs (excluding the unused data bits). An additional "carry-through" input is present on the right, which allows chaining of multiple OR modules together when more than 4 inputs are required. The output on the left can be used for control of other modules, or chained into another OR.

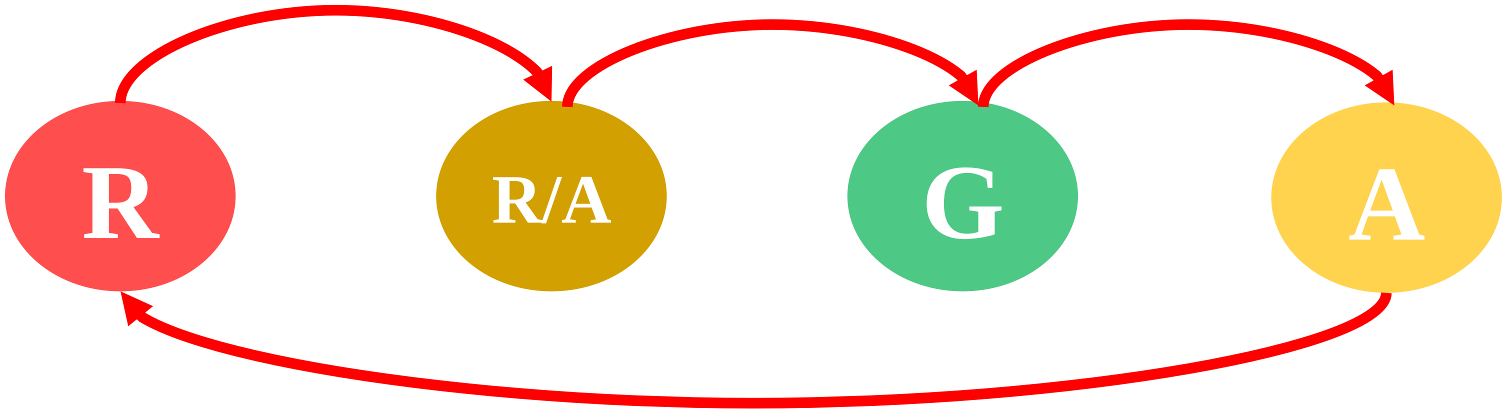
The module is used largely for control logic, where the activation of any one of many control lines may need to trigger a single module, or set of modules (through fan-out).

Split/merge

The splitter/merger module allows splitting or combination of data lines, as indicated on the board's front. Bits designated by a letter ‘A—D’ are passed through; those designated as an ‘X’ are left unconnected.

Task 1: Traffic lights Mk.1 – state detection approach

In this task, you'll build some traffic lights that go through a pre-defined sequence of operations, according to a state machine. As seen in lectures, here is the state transition diagram for UK traffic lights:



Your task is to produce this sequence in response to clock ticks. Since we do not have the various colours available, we'll represent all three colours as single bits of a data value. Use the following scheme for visualising:

d3	***d2***	***d1***	***d0***
Not used	Green	Amber	Red

Therefore, the following data values will represent the above state transitions:

```
0001
0011
0100
0010
<loop>
```

NOTE: For this task, you will need 5 input modules. You Build-a-comp kit contains only 4, so you will need to share between groups. You will only need 5 Input modules for this short task, so once you are done, please restore your kits to containing 4 Input modules.

Task procedure:

1. Take a latch, which you will designate as the “visual output” of your circuit. For this latch, the data LEDs will be interpreted as the various colours according to the table above. Thus, this latch represents your traffic lights.
2. Take four input modules and set each to define one of the four possible outputs. Your goal is to make these patterns appear on the output register.
3. Add in additional components to make the traffic lights cycle through the correct sequence.

Hint coming up… (*if you don’t want a hint about how to go about performing this task, look away now and attempt Task 1 before continuing*):

There are many different ways to implement the traffic lights, but the simplest starts by observing a few things about the problem in hand:

1. There is a sequence of four states that need to be cycled through.
2. Each state produces a pre-defined output.
3. The result can be represented in a single 4-bit data word.

Thus the simplest implementation comprises:

- A mechanism for detecting which of the four states we are in and piping one of four fixed inputs to the output;
- A mechanism for transitioning between the four states.

By now, you probably have a good idea of how to go about implementing the traffic lights***. If you need more help, read the next sentence***, ***otherwise stop and attempt the implementation***. More help would observe that a counter with a fixed increment is an efficient way of representing multiple states. Further, that the LSBs of a counter of size >2 bits cycle just the same as a counter of size 2 i.e. we can ignore the MSBs of a larger counter and still get the effect of a 4 state machine.

Task 2: Traffic lights Mk.2 – Disjunctive Normal Form (DNF) approach

In Task 1, you hopefully created a very simple implementation of traffic lights. This time we'll use a more generic method to create the same end result. We do this since it becomes a generic method.

In this version of the task, you are not permitted to use Input modules to encode the output patterns of the lights. Rather, we will use Boolean algebra and DNF analysis to create the light sequence we need.

DNF primer:

Disjunctive Normal Form comprises Boolean expressions of maxterms and disjunction. Maxterms are Boolean terms that combines variables only with negation and not with conjunction, disjunction is logical ORing.

Example: $a = b + c + \bar{d}$ contains three maxterms: $\{b, c, \bar{d}\}$ and $a = b + c + \bar{d}$ adds in disjunction to create the final DNF expression.

Generic DNF implementation method:

1. Write down the truth table showing the relationship between the states and which lights are on and off.
2. Express each bulb's on-state as an Boolean expression comprising solely of maxterms and disjunction (ORing)
3. Wire up the OR gates to implement the Boolean expression.

Task1 procedure:

Build a new version of the traffic lights that creates the output patterns by:

- Detecting the state that the system is in and asserting one of four signals.
- Combining the signals with OR gates to produce the correct outputs on the three traffic light bulbs.

The task can be completed via three simple steps.

Step 1:

Each light needs its own DNF expression in terms of the state of the system. Create and write down these expressions.

I recommend expressing the states as {S1, S2, S3, S4} rather than {0001, 0011, 0100, 0010}, to make things clearer. This would, for example mean that the Red light can be expressed in DNF as

$$RED = S1 + S2.$$

Step 2

Generate each of the state maxterm signals i.e. you will need to produce a circuit that creates four signals: {S1, S2, S3, S4}. (aside, since only one of these signals is every active at once, the states are now said to be "one-hot" encoded).

Step 3

Use the OR gates to create the necessary outputs from the state signals.

Task 3: Traffic lights Mk.3 – arithmetic approach

So far, we have implemented traffic lights by counting through states and then decoding the state to give a one-hot output value, which then lights the appropriate lights.

In this task, we'll take an alternative approach. Rather than counting states, we'll implement a system that uses arithmetic to calculate the next output value based on the previous state.

We can do this by observing that the output light value sequence has fixed differences between states, when represented as (signed or unsigned) numbers.

State	Difference from previous
0001	
0011	+2
0100	+1
0010	-2
<loop>	-1

Therefore, an implementation of traffic lights could use variable arithemetic operations to implement the sequence.

Task procedure:

- Use an arithmetic unit with variable inputs to produce the required output sequence.

HINT COMING UP… *you will need to make both the data and control paths conditional. Look carefully at the bit patterns to see how to generate your control signals. Producing a truth table showing the relationship of current state to next ALU operation and next constant value will be helpful. You will also have to deal with the fact that, on reset, the output needs to be 0001 not 0000.*

Task 4: Traffic lights Mk.4 – conditional state changes

In our final iteration of the traffic lights, we will deal with the fact that external inputs can alter the operation of our system.

Take an Input module. The d0 switch will now represent a pedestrian who has pressed a button and wishes to cross the road.

We wish to make the most efficient road traffic, so we will take the following approach.

- If the pedestrian button is pressed and the lights are green, they will sequence to red.
- If the button is still pressed, they will remain red until the button is released.
- If the lights are red and the button is not pressed, they will immediately transition to green.
- Green lights without pressed buttons do not change state.

Task procedure

1. Draw an updated state transition diagram to reflect the above rules (you can use the space below for this)

2. Implement the system in the Build-a-comp modules.
3. Add your implementation to your portfolio.

HINT COMING UP… the conditional arithmetic approach works well here, since a system which does not need to transition will add/subtract 0 from the current output).