Combinatorial Circuits

Note to Anyone Reading

I've left out a lot of stuff because I did it last year in CS1110 and CS1111. If you need more info, you can look at those notes.

Inputs and outputs

Inputs can be artbitrarily assigned to specific states, and the output is the result/observation (whatever follows).

A simple example is a switch in a circuit with a light – we control the switch which determines the light's state, so the switch is the input and the light is the output.

Outputs depend on inputs but inputs do not depend on outputs.

Distribution

A.(B + C) = A.B + A.C

A+(B . C) = A+B . B+C

The Majority Decoder

A box with three inputs (A, B, C) and one output (Y), where Y follows the majority input.

Applications

- The carry digit in a half or full adder
- A security system with multiple sensors

Note

The XOR function cannot be simplified and is the nastiest switching function.

Multiplexor

With a 2-1 MUX, you can express it as `B.C + A.C'`. An optional additional term is `A.B` – it makes no difference to the truth table. However, it makes a difference in circuit design:

- If `A = 1`, `B = 1`, and C changes from 1 to 0, the gate delay caused by the NOT gate will mean that for a brief interval, neither A nor B is selected. The output, instead of remaining at 1, will dip before coming back to it. This causes a glitch.
- Adding an `A.B` gate will fix this.

Simplifying Canonical Sums of Products and Products of Sums

As `A.A = A` and `A + A = A`, you can duplicate existing terms without changing the result. So you can pair off one term with others multiple times in order to cancel things.

You can either cancel terms using distribution and other rules, or you can work it out in English.

Karnaugh Map

Grey Coding

Reorganise the truth table so that every pair of adjacent lines (including the first and last) differ only by one digit.

[missed stuff here]

Karnaugh Map with "Don't-Care" Entries

Take a function with four inputs A, B, C, and D. Interpret A and B as two digits of one number, and the same for C and D. We want the circuit to output 1 when the number formed by C, D is bigger than the number formed by A, B.

[see handwritten notes; need to scan them]

Next Assignment

A 2-to-1 MUX which also has a clamp zero case. this will require two control lines C and D, but one of the four combinations of C and D will be a "don't-care" entry, as we are only using three of the cases. We'll get more details by Monday.

BCD-to-7-Segment Decoder (getting back to this in the future)

We want 10 different outputs (we're ignoring the possibility of including letters). To do that, we need 4 inputs in our switching function (3 is not enough), which will give us 16 cases. In practice, the LEDs are connected so that the first 10 rows of the truth table are used, and the last 6 rows are then "don't-care" entries.

Implication

A	B	Y
0	0	1
0	1	1
1	0	0
1	1	1

E.g. A is whether it rains and B is whether the road is wet.

[missed a bit here]

Karnaugh Map with 6 Inputs

Still possible with 6 inputs. With more inputs computers are required.

Karnaugh Map with More Inputs

Terminology for Quine McCluskey Algorithm

An *implicant* is a minterm of a switching function, but also terms from simplified solutions, e.g. `A.B` from the majority decoder.

A *prime implicant* is an implicant of the form `A.B.C.D.…`, where removing any of the variables causes it to no longer be an implicant. I.e. it is a block of maximum size – it is simplified as far as possible.

Note that the SSOP is not the sum of all prime implicants, but only of some of them. Take for example the following Karnaugh Map:

A B	0	1
0 0	1	0
0 1	1	1
1 1	0	1
1 0	0	0

You can draw three 2x1 rectangles to cover the ones in that Karnaugh Map, but you could also do it with just two of them. One of the three is not required.

Instead the SSOP is the minimum cover of all prime implicants.

An *essential prime implicant* is one that is present in the SSOP.

Computer algorithm solutions (including the Quine McCluskey Algorithm) involve finding all prime implicants and then finding a minimum cover from them.

DeMorgan's Theorem

Our next assignment will be something like: here's an expression, develop two simplified circuits for this expression, one using only NAND gates, one using only NOR gates.

[I've left out a fair bit here; see CS1110 or possibly CS1111]

```
A . B' + C' . D = ((A . B')' . (C' . D)')'
```

In this module, you're allowed to use NOT gates/symbols when making NAND-only and NOR-only circuits/expressions, as you can easily make NOT gates from NAND and NOR gates, and that's considered an unimportant detail.

Remember if you end up with an expression that has an OR or AND gate (rather than a NOR or NAND gate), to represent that gate in a circuit diagram, you will need to use a NOR or NAND gate followed by a NOT gate.

XOR with NANDs

First, a rule:

```
A . B' = A . (A . B)'
```

Apply DeMorgan's theorem to the right-hand side to prove it:

```
A . B' = A . (A' + B')
A . B' = A.A' + A.B'
```

You can use this to change the representation of XOR with NAND gates from this:

```
((A . B')' . (A' . B)')'
```

To this:

```
((A . (A . B)')' . (B . (A . B)')')'
```

This looks more complicated, but because you can reuse the `(A . B)'` input, this only takes 4 gates in a circuit diagram, while the previous form took five. This is a level 3 circuit, as (not counting inverters) the longest path is through 3 gates.

XNOR with NORs

Starting with XOR as `(A + B').(A' + B)`, you can use DeMorgan's theorem to change it to:

```
((A + B')' + (A' + B)')'
```

There's a similar rule to before:

```
A + B' = A + (A + B)'
```

(You can prove this by using DeMorgan's theorem on the bracketed part of the right-hand side, and then distributing `+` across `.`.)

This gives this expression for XNOR, which has a simpler circuit solution than our last form:

```
((A + (A + B)')' + ((A + B)' + B)')'
```

Again the circuit realisation saves 1 gate by using `(A + B)'` twice. Again this is a level 3 circuit.

Multiplexors

[see CS1110]

Multiplexors are used in serial ports, e.g. USB, to send different signals over the same wire.

Remember you can use a 4-1 mux to represent any switching function of two variables by feeding the truth table to the inputs and the variables to the select lines.

You can buy multiplexors which are set up initially with every input connected to both 1 and 0, and you can use an electric pulse to destroy the connections you don't want, in order to create the switching function you want.

Majority Decoder with MUX

You can construct a (3-input) majority decoder with an 8–1 mux in the way described above.

However, you can also construct one with a 4–1 mux, by using functions of C as the inputs and A and B as the control lines.

There are 4 switching functions of one variable C: constant 0, constant 1, C and C'.

Here's the truth table for the majority decoder:

A	B	C	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

If you look at two lines at a time, so that the value of A and B are fixed, you can see that:

- When `(A, B) = (0, 0)`, you want constant 0 at the output.
- When `(A, B) = (0, 1)`, you want C at the output.
  - `0` if `C = 0` and `1` if `C = 1`
- When `(A, B) = (1, 0)`, you want C at the output.

- When `(A, B) = (1, 1)`, you want constant 1 at the output.

2-bit Comparator

A circuit that takes 4 inputs, A and B which represent a 2-bit number p, and C and D which represent a second 2-bit number q.

If the first number is bigger, we output 1. If they are equal or the second is bigger, we output 0.

We can solve this with a 16–1 mux, which is straightforward. We can also solve it with an 8–1 mux, or a 4–1 mux.

Solution with an 8–1 Mux

8 inputs (D0–D7) to the mux, which can be connected to any function of D. A, B, and C are used as the control inputs.

D0

A	B	C	D	y
0	0	0	0	0
0	0	0	1	0

This table above represents the cases where A, B, and C are 0. Since we want 0 in both of these cases, the function of D that we use is constant 0.

D1

A	B	C	D	y
0	0	1	0	0
0	0	1	1	0

In this second table, `p = 0`, and `q = 0` and `q = 1`. So we still want `constant 0`.

D2

A	B	C	D	y
0	1	0	0	1
0	1	0	1	0

For D2 we now want `D'` at the output.

D3

A	B	C	D	y
0	1	1	0	0
0	1	1	1	0

`Constant 0` here as q is always greater than p.

D4

A	B	C	D	y
1	0	0	0	1
1	0	0	1	1

`Constant 1` here as p is always greater than q.

D5

A	B	C	D	y
1	0	1	0	0
1	0	1	1	0

`Constant 0` here as q is always greater than p.

D6

A	B	C	D	y
1	1	0	0	1
1	1	0	1	1

`Constant 1` here.

D7

A	B	C	D	y
1	1	1	0	1
1	1	1	1	0

Output of `D'` here.

Solution with a 4–1 Mux

Now we connect `A` and `B` to the control lines, and functions of `C` and `D` to the data inputs (`D0` – `D3`).

D0

A	B	C	D	y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0

As the four outputs are 0, our switching function of `C` and `D` is `constant 0`.

D1

A	B	C	D	y
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0

Our function of `C` and `D` here is `C' . D' = (C + D)'`.

D2

A	B	C	D	y
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0

Our function here is `C'`.

D3

A	B	C	D	y
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Our function here is `C' + D' = (C . D)'`.