EECS-140/141 Introduction to Digital Logic Design
Lecture 6: Common Combinational Logic Circuits

I. MULTIPLEXERS (MUXES)

I.A Review

I.A.1 Basic Definition

Recall basic 2:1 mux: 2 data inputs a_1 and a_0, select input s

and 4:1 mux: 4 data inputs, 2 select inputs

I.A.2 Muxes from Muxes

We can build bigger muxes from smaller ones, and *any* size mux can be built with 2:1 muxes.

Example: 4:1 from 2:1

I.B Muxes in FPGAs

I.B.1 Field Programmable Gate Arrays (FPGAs)

This is a specific type of Programmable Logic Device (PLD) characterized by:

- High gate density
- Flexible logic blocks
- Rich interconnection capability

Logic blocks are implemented as:

A LUT can be *programmed* to implement *any* of the 2^{2^n} possible logic functions for *n* inputs.

I.B.2 Mux Implementation of a LUT

2-input LUT can be implemented as:

I.B.3 LUT Inputs

Muxes can also be used to select LUT inputs from several possibilities (rich interconnection).

I.C Mux Implementation of General Logic Functions

I.C.1 Introduction

We saw in the last section that we can implement *any* logic function as a LUT using muxes.

But for a *specific* logic function, a more efficient implementation may be possible.

Example: 2-input "match" function: $f = ab + \bar{a}\bar{b}$

Since there are 2 inputs, we *could* implement with:

But can be done with 2:1 mux as well (simpler):

So can implement with 2:1 mux as:

Could *also* be done with *b* as select input (you try).

This simple TT-based method is not very general: works best only for leftmost input (the "msb") of TT being the mux select.

I.C.2 Shannon's Expansion Theorem

I.C.2.a Introduction

To use a 2:1 mux for a given logic function f with x_i as select input, we need to get it in the form:

Shannon's Expansion Theorem says this is:

Any *n*-input Boolean function can be written as::

Can expand this way in terms of *any* of the x_i.

f_{x_i} is called a *cofactor* with respect to x_i.

Above is good for 2:1 mux implementations, but what about 4:1?

Do it *again* for *each* of $f_{\bar{x}_1}$ and f_{x_1} in terms of *another* variable.

I.C.2.b Methodology

Shannon's Theorem says the cofactors ($f_{\bar{x}_1}$ and f_{x_1}) exist, but doesn't tell you how to *find* them (and *neither* does your textbook!).

One way is to write out a TT, similar to what we did earlier. Here is another (better) way.

— Express f in *any* SoP form (need not be CSoP)

— Choose expansion variable x_i

— For *each* term of f:

- If term contains $\bar{x}_i$ literal, that term with $\bar{x}_i$ *removed* is a term of $f_{\bar{x}_i}$.
- If term contains x_i literal, that term with x_i *removed* is a term of f_{x_i}.
- If term does not contain $\bar{x}_i$ or x_i literal, that term is a term of *both* $f_{\bar{x}_i}$ and f_{x_i}, since:

— Simplify cofactors if possible.

I.C.3 Examples

I.C.3.a 2-Input Match Function (Simple)

I.C.3.b $f(a,b,c) = a\bar{b} + \bar{b}\bar{c} + \bar{a}bc$

Expand via *a*:

Or expand via *b*:

Or expand via *c*:

Expansion via *b* is simplest: show circuit:

Could even use $\overline{(a+\bar{c})} = \bar{a}c$:

4-input mux implementation?

Expand cofactors in:

Use *b* expansion above, then expand $f_{\bar{b}}$ and f_b via *c*:

Then the 4:1 mux implementation is as follows:

II. DECODERS

II.A Introduction

We already know that an *n*-bit word can represent 2^n different *things*. A binary decoder represents this as follows:

- It has *n* input bits representing an *n*-bit unsigned binary number (or code).
- It has 2^n outputs, each representing one possible "decoded" value. *Only one* output can have value 1 at any time.
- It optionally may have an Enable (*En*) input that forces all outputs to 0 when *En* = 0.

This is an *n* to 2^n decoder.

Example: 2 to 4 decoder:

II.B Decoder Circuits

II.B.1 Logic Circuit for 2 to 4 Decoder

II.B.2 Big Decoders from Smaller Decoders

We can use a decoder with higher-order bits to control a bank of decoders for lower-order bits, using *En* inputs. See Figure 6.18.

II.C Decoder Applications

II.C.1 Mux Implementation

Think about how you could implement a 4:1 mux using a 2 to 4 decoder (exercise).

II.C.2 Decoder as Demux

Demux does the opposite of a mux: 1 data input goes to one of 2^n outputs under control of n select inputs.

II.C.3 Memory (Data Storage) Access

Computer systems store data as *n*-bit words (e.g., *n*=32). A coded *address* (*m* bits long) is used to designate a particular memory (data storage) location. Can use an *m* to 2^m decoder to generate "enable" signals for 2^m memory locations.

III. ENCODERS

III.A Basic Encoder

An encoder does the opposite of a decoder: output is *n*-bit code corresponding to one of 2^n inputs that has value 1. *Assumes* exactly one input "active" at a time.

III.B Priority Encoder

III.B.1 Definition

This version *allows* multiple inputs to be active (have value 1) at one time, but outputs the code of the input with:

Example: 4 to 2 Priority Encoder with priority same as input index. "Compressed" TT below:

III.B.2 Implementation

Instead of using our previous techniques, we will use a design taken directly from the compressed TT. First, represent the last 4 rows as 4 intermediate variables i_0 to i_3:

Then:

IV. COMPARATOR CIRCUITS

Often useful to *compare* one number (a) to another number (b).

IV.A Unsigned Comparator

Simple case: two 4-bit inputs (a and b) in *unsigned* binary.

3 outputs:

Note: one and only one output will be active (value of 1) for *any* a and b.

Design?? 8 inputs and 3 outputs, so 3 TT, each with $2^8 = 256$ rows. How would we do an 8-input K-map???

Instead, we will think through the design logically.

IV.A.1 *AeqB* Output

It should be clear that $a = b$ if all bits are the same:

But $a_k = b_k$ if both are 0 or both are 1, so define intermediate variables i_k as:

Then:

IV.A.2 *AgtB* Output

Since bit 3 is MSB, if $a_3 > b_3$ (that is, if $a = 1$ and $b = 0$), then $a > b$.

Also, if $a_3 = b_3$ (matching condition i_3), and if $a_2 > b_2$, then $a > b$.

Keep following the pattern and use the intermediate variables:

IV.A.3 *AltB* Output

We *could* do a similar thing for *AltB* as we did for *AgtB*, but *better* to use the fact that:

So,

IV.B 2-Input Sorter

To illustrate how we can build up a design using existing blocks/circuits, we can use a comparator as part of a circuit that *sorts* 2 inputs: the larger of *a* and *b* goes to the *Max* outputs, and the smaller goes to the *Min* outputs.

How to make a crossover switch? Muxes!!