Digital Logic(H)
Southern University of Science and Technology
Mengxuan Wu
12212006

Midterm Review

Mengxuan Wu

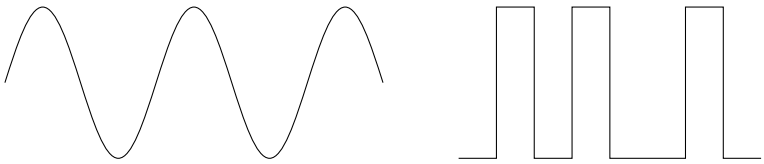
1 Theory

1.1 Binary Number System

1.1.1 Analog vs. Digital Signals

Analog signals change continuously over time. They convert information into electric waves of varying amplitude and record exact waveform.

Digital signals are discrete time signals generated by digital modulation. They are made by sampling along the wave form.



Analog Digital

A digital system is a system that operates on discrete values and performs operations such as logic, arithmetic, and data storage in a binary format.

1.1.2 Common Number Systems

To distinguish different number systems, we use prefix to indicate the base of the number system. It's common to use "0" for octal and "0x" for hexadecimal (sometimes "0b" for binary). For example, $(1011)_2 = (11)_{10} = (013)_8 = (0\text{xB})_{16}$.

To convert a number with $n+m$ digits in base r to decimal, we can use the following formula:

$$\begin{aligned} D &= \overline{d_{n-1}\cdots d_1 d_0 . d_{-1} d_{-2}\cdots d_{-m}} \\ &= d_{n-1}r^{n-1} + \cdots + d_1 r^1 + d_0 r^0 + d_{-1}r^{-1} + \cdots + d_{-m}r^{-m} \\ &= \sum_{i=-m}^{n-1} d_i r^i \end{aligned}$$

To convert a decimal number to base r, we can use the following algorithm:

	Quotient	Remainder	Coefficient
$13 \div 2$	6	1	$a_0 = 1$
$6 \div 2$	3	0	$a_1 = 0$
$3 \div 2$	1	1	$a_2 = 1$
$1 \div 2$	0	1	$a_3 = 1$

Integer Part

	Integer	Fraction	Coefficient
0.375×2	0	0.75	$a_{-1} = 0$
0.75×2	1	0.5	$a_{-2} = 1$
0.5×2	1	0	$a_{-3} = 1$

Fractional Part

To be noticed, the coefficients are read in **different order**. For the integer part, we read the coefficients from bottom to top. For the fractional part, we read the coefficients from top to bottom.

Therefore, $(13.375)_{10} = (1101.011)_2$.

1.1.3 Common Notions

A **bit** is a binary digit, which is either 0 or 1. A **byte** is a group of 8 bits.

The **most significant bit** (MSB) is the bit that has the largest value. The **least significant bit** (LSB) is the bit that has the smallest value. This also applies to bytes.

The prefix for bits and bytes are as follows:

Power	Meaning	Prefix	Abbreviation
2^{10}	1024	Kilo	K
2^{20}	1024^2	Mega	M
2^{30}	1024^3	Giga	G
2^{40}	1024^4	Tera	T
2^{50}	1024^5	Peta	P
2^{60}	1024^6	Exa	E
2^{70}	1024^7	Zetta	Z

1.1.4 Complements

Two types of complements: r's complement (radix complement) and $(r-1)$'s complement (diminished radix complement). For example, 9's complement of 1234 is $9999 - 1234 = 8765$, and 10's complement of 1234 is $10000 - 1234 = 8766$. An easier way to calculate the 10's complement is to add 1 to the 9's complement.

To perform subtraction using complements, we replace the subtraction with addition with this formula:

$$\begin{aligned} A - B &= A + (r^N - B) - r^N \\ &= A + r\text{'s complement of } B - r^N \end{aligned}$$

We demonstrate this with an example:

72532 - 3250(10's complement is 96750)

$A =$	72532
10's complement of $B =$	+96750
Sum =	169282
Discarding the carry =	-100000
$A - B =$	69282

(Discarding the carry when $A \geq B$)

3250 - 72532(10's complement is 27468)

$A =$	3250
10's complement of $B =$	+27468
Sum =	30718
Add "-" to 10's complement of the sum =	-100000
$A - B =$	-69282

(Add "-" to 10's complement of the sum when $A < B$)

1.1.5 Signed Binary Numbers

We use 2's complement to represent signed binary numbers, by make the MSB the sign bit. This is because arithmetic operations holds even when we use 2's complement in operations. The range of a signed binary number is -2^{N-1} to $2^{N-1} - 1$.

To represent a negative number, we first represent the absolute value of the number in binary, then find the 2's complement of the binary number, and finally add a negative sign to the MSB. For example, $(-105)_{10} = (1101001)_2 = (0010111)_{\text{2's Complement}} = (10010111)_{\text{Signed Binary Number}}$.

1.1.6 BCD Code

BCD stands for Binary Coded Decimal. It is a way to represent decimal numbers in binary. Number 0 to 9 are represented by 4 bits, and the remaining 6 combinations are invalid.

In BCD addition, we add the two numbers digit by digit. If the sum is greater than 9, we add 6 to the sum.

BCD subtraction is similar to binary subtraction, we convert the subtrahend to 10's complement and add it to the minuend.

1.1.7 Gray Code

Gray code is a binary code where two successive values differ in only one bit. This is useful in error detection and correction, low power design and Karnaugh maps.

The Gray code for 0-15 is as follows:

Decimal	Gray Code
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

1.1.8 Error-Detecting Codes

A parity bit is a bit added to a string of binary code to ensure that the total number of 1-bits in the string is even or odd. It can only detect odd number of errors.

1.2 Boolean Algebra & Logic Gates

1.2.1 Boolean Axioms and Theorems

- **Distribution Laws**

$$x(y+z) = xy + xz$$
$$x + yz = (x+y)(x+z)$$

- **De Morgan's Laws**

$$(x+y)' = x'y'$$
$$(xy)' = x' + y'$$

- **Simplification**

$$(x+y')y = xy$$
$$xy' + y = x + y$$

1.2.2 Simplify Boolean Function

A **literal** is a variable or its complement. A **product term** is a product of literals. A **sum term** is a sum of literals. For example, $x'y'z + x'yz + xy'$ has 8 literals, 3 product terms and 1 sum term.

A **minterm** is a product term in which each variable appears exactly once in either complemented or uncomplemented form. A **maxterm** is a sum term in which each variable appears exactly once in either complemented or uncomplemented form. Each minterm and maxterm corresponds to a unique row in the truth table. ($M_i = m_i'$)

The value of a minterm is 1 only for the row in which the variables have the values specified by the minterm. The value of a maxterm is 0 only for the row in which the variables have the values specified by the maxterm.

Canonical forms refer to the sum of minterms and product of maxterms. We often use $\sum$ to denote sum of minterms and $\prod$ to denote product of maxterms. Note that $(\sum(a_1, a_2, \cdots, a_n))' = \prod(a_1, a_2, \cdots, a_n)$.

Standard forms refer to the forms that is either SOP or POS.

1.3 Gate-Level Minimization

We can use Boolean algebra or Karnaugh maps to simplify Boolean functions.

In Karnaugh maps, minterms are arranged in Gray code order. Some Karnaugh maps may have don't care conditions, which are represented by "X".

The task of simplification is to find the implicants. A prime implicant is a 1-product term that obtained by combining the maximum possible number of adjacent squares in the map. A prime implicant is essential if it is the only implicant that covers a particular minterm.

Therefore, the simplification steps are as follows:

1. Find all essential prime implicants and cover them.
2. Find a minimum set of prime implicants that cover all the remaining minterms.

Two typical Karnaugh maps are as follows:

A \ BC	00	01	11	10
0	m_0	m_1	m_3	m_2
1	m_4	m_5	m_7	m_6

3-Variable Karnaugh Map

AB \ CD	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}

4-Variable Karnaugh Map

1.4 Two-Level Implementation

1.4.1 Universal Gates

A universal gate is a gate that can implement any Boolean function without need to use any other gate type. The NAND gate is a universal gate, and so is the NOR gate.

In NAND circuits, we can implement other gates as follows:

- Inverter: $x' = \text{NAND}(x, x)$
- AND: $xy = (\text{NAND}(x, y))'$
- OR: $x + y = \text{NAND}(x', y')$

In NOR circuits, we can implement other gates as follows:

- Inverter: $x' = \text{NOR}(x, x)$
- AND: $xy = \text{NOR}(x', y')$
- OR: $x + y = (\text{NOR}(x, y))'$

1.4.2 NAND and NOR Implementations

We can implement any Boolean function using only NAND or NOR gates. If the function is in SOP form, we can use NAND gates. If the function is in POS form, we can use NOR gates.

The expressions are as follows:

$$\begin{aligned} ab + cd &= ((ab + cd)')' \\ &= ((ab)'(cd)')' \\ (a + b)(c + d) &= (((a + b)(c + d))')' \\ &= ((a + b)' + (c + d)')' \end{aligned}$$

1.4.3 Two-Level Implementations

There are 16 possible combinations of two-level implementations and 8 of them degenerate to one-level implementations. The other 8 can be divided into 4 groups:

- AND-OR / NAND-NAND => SOP
- OR-AND / NOR-NOR => POS
- NAND-AND / AND-NOR => AOI (AND-OR-INVERT or Complement of SOP)
- OR-NAND / NOR-OR => OAI (OR-AND-INVERT or Complement of POS)

1.4.4 Exclusive-OR Function

An odd function is a function that is 1 when the number of 1's in the input is odd. We can use the XOR function to implement an odd function by XORing all the inputs. Similarly, an even function is a function that is 1 when the number of 1's in the input is even and can be implemented by XORing all the inputs and then inverting the output.

These two functions are useful in error detection (parity check). For an even parity bit of 3 inputs, we can use $P = x \oplus y \oplus z$. And for an even parity checker of the same message, we know $C = x \oplus y \oplus z \oplus P$.

1.5 Combination Logic

1.5.1 Combinational Circuits and Sequential Circuits

A combinational circuit is a circuit whose outputs depend only on the current inputs. A sequential circuit is a circuit whose outputs depend on the current inputs and the current state (memory elements).

1.5.2 Analysis Procedure

1. Label all gate outputs that are functions of the input variables only. Determine the functions.
2. Label all gate outputs that are functions of the input variables and previously labeled gate outputs, and find the functions.
3. Repeat previous step until all the primary outputs are obtained.

1.6 Standard Components

1.6.1 Decoder

A decoder is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines. We may have an enable input to enable the decoder.

There might be inverters on the output lines or the enable input. If the enable input is inverted, we call it Active Low Enable.

We can use two lower-order decoders with an enable input to implement a higher-order decoder. But we need to make sure the MSB of the input is connected to the enable input of both decoders.

1.6.2 Mutiplexer

A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line.

Likewise, we can use five lower-order multiplexers to implement a higher-order multiplexer. And the higher bits are connected to the select inputs the last multiplexer.

1.6.3 Demultiplexer

A demultiplexer is a combinational circuit that directs a single input line to one of 2^n output lines. A decoder with an enable input can be used as a demultiplexer.

1.6.4 Encoder

An encoder is a combinational circuit that performs the inverse operation of a decoder.

To make sure don't care conditions are not encoded, we can use a priority encoder, which has a priority order for the input lines and a valid output indicator. The truth table of a priority encoder that D_3 has the highest priority is as follows:

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

1.6.5 Tri-state Buffer

A tri-state buffer has truth table as follows:

Inputs		Outputs
E	A	Y
0	X	Z
1	0	0
1	1	1

1.7 Gate Delays

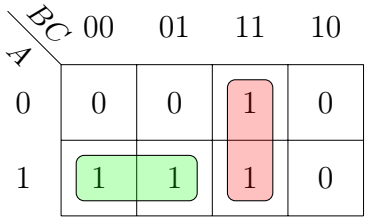
The propagation delay is the time required for the output to change after the input changes.

Because of the propagation delay, we may have glitches in the output. A static 1-hazard is a glitch that the output goes from 1 to 0 and back to 1 when the output should be 1. A static 0-hazard is a glitch that the output goes from 0 to 1 and back to 0 when the output should be 0.

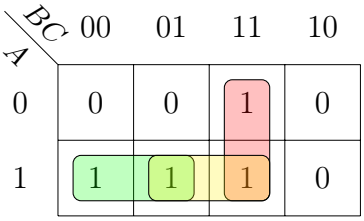
Static 1-hazard Static 0-hazard

To eliminate glitches, we can use a hazard-free circuit. If two adjacent minterms in Karnaugh map are always in some prime implicant, we can eliminate the hazard.

For example:



With Hazard



Without Hazard

2 Lab

2.1 Name of Object

The identifier must begin with an alphabetic character or the underscore character (a-z or A-Z or _). Identifiers may contain alphabetic characters, numeric characters, the underscore, and the dollar sign (a-z or A-Z or 0-9 _ or $). That means name like "1a" is not allowed.

2.2 Numbers

There are four radixes and the syntax should be: <size>'<radix><value>.

The radixes are as follows:

Radix	Meaning
b	Binary
o	Octal
d	Decimal
h	Hexadecimal

Value that exceed the size will be truncated. If the size is omitted, the default size is 32 bits. For example, "6'hCA" is equivalent to "001010", and "hf" is equivalent to "00000000000000000000000000001111".

2.3 Signed vs. Unsigned

There are two types of numbers: signed and unsigned. The default type for integer is signed. The default type for reg and wire is unsigned.

When we assign a shorter number to a longer number, the shorter number will be extended in two possible ways:

- Zero extension: the shorter number is extended with 0's.
- Sign extension: the shorter number is extended with the sign bit.

2.4 Wire vs. Reg

Type	Demo	Wire	Reg
Input of a module	module tx(input a)	Yes	No
Output of a module	module tx(output a)	Yes	Yes
Variable in continuous mode	assign a =	Yes	No
Variable in procedure mode	always @* a =	No	Yes
Variable binds to input port	tx dut(.in(a))	Yes	Yes
Variable binds to output port	tx dut(.out(a))	Yes	No

2.5 Other Important Facts

- Different size of variables must be declared separately. For example, "reg [3:0] a, b" will declare two 4-bit variables.
- The type of variable can be declared later in the code. For example, "output reg [3:0] a" is equivalent to "output [3:0] a; reg [3:0] a".
- When using continuous mode, you cannot assign conflicting value to a variable. For example, "assign a = 1'b1; assign a = 1'b0;" is not allowed.
- Both structural and data-flow modeling cannot be nested in behavior modeling. For example, "always @* begin assign a = b; end" is not allowed.