

ENEE244-010x

Digital Logic Design

Lecture 2

Announcements

- Reminder: Check UTF Office Hours and contact information on Syllabus/Webpage
- Reminder: First homework assigned (see course webpage). Due date: Sept. 10 @ 11:59pm.
- Reminder: Readings for each lecture up on course webpage
- Reminder: First recitation is Wednesday, September 9.

Agenda

- Last time:
 - Positional Number Systems (binary, decimal, octal, hexadecimal) (1.2, 1.4)
 - Number-base conversions (1.3, 1.4)
- This time:
 - Signed numbers and Complements (1.5, 1.6)
 - Addition and Subtraction with Complements (1.5, 1.6)
 - Binary Codes (1.7)

Polynomial method of number conversion

- Convert from base r_1 to base r_2
- Express number as polynomial in base r_1
$$- N = d_2 \times r_1^2 + d_1 \times r_1^1 + d_0 \times r_1^0$$
- Switch each digit symbol d_i to its base r_2 representation and each base symbol r_1 to its base r_2 representation.
- Evaluate the polynomial in base r_2 .

Polynomial Method of Number Conversion

- Example: convert from hexadecimal to decimal
- Hexadecimal number: A78E

$$- A78E = A \times (10_{(16)})^3 + 7 \times (10_{(16)})^2 + 8 \times (10_{(16)})^1 + E \times (10_{(16)})^0$$

$$- A78E = (10) \times (16)^3 + (7) \times (16)^2 + (8) \times (16)^1 + (14) \times (16)^0$$

$$- A78E = 42894$$

- **Use this method when converting a number into decimal form (e.g. binary to decimal)

- Why? Ex: convert 23 from decimal to binary: $2 \times 10_{(10)} + 3 =$

$$\begin{array}{r} 10 \times 1010 + 11 \\ \hline 10100 + 11 \\ \hline 10111 \end{array}$$

Iterative Method of Number Conversion

50.40625

- Convert from base r_1 to base r_2 . 110010.01101
- Perform repeated division by r_2 . The remainder is the digit of the base r_2 number.
- Example: Convert 50 from decimal to binary
 - Divide 50 by 2, get 25 remainder 0 ←
 - Divide 25 by 2, get 12 remainder 1
 - Divide 12 by 2, get 6 remainder 0
 - Divide 6 by 2, get 3 remainder 0
 - Divide 3 by 2, get 1 remainder 1
 - Divide 1 by 2, get 0 remainder 1
- Answer is: 110010
- Can verify using the polynomial method
- **Use when converting from decimal to another base. (e.g. decimal to binary)
- Why?

Iterative Method for Converting Fractions

- Convert from base r_1 to base r_2 .
- Perform repeated **multiplication** by r_2 . The **integer part** is the digit of the base r_2 number.
- Ex: Convert .40625 from decimal to binary
 - Multiply .40625 by 2, get 0 + .8125
 - Multiply .8125 by 2, get 1 + .625
 - Multiply .625 by 2, get 1 + .25
 - Multiply .25 by 2, get 0 + .5
 - Multiply .5 by 2, get 1 + 0
- Answer is: .01101
- Can verify using the polynomial method

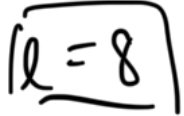
Handwritten diagram illustrating the iterative multiplication method for converting a decimal fraction to binary. It shows a sequence of multiplications by 2, with arrows indicating the progression from one step to the next. The steps are: $0 \times \frac{1}{2}$, $1 \times \frac{1}{2^2}$, $0 \times \frac{1}{2^3}$, and $0 \times \frac{1}{2^4}$. The first step is labeled with a '2' in parentheses, and the subsequent steps are labeled with 2^2 , 2^3 , and 2^4 respectively. Arrows point from the first step to the second, and from the second to the third, and from the third to the fourth.

Special Conversion Procedures

- When converting between two bases in which one base is a power of the other, conversion is simplified.
- Ex: Convert from $1101\ 0110\ 1111\ 1001$ from binary to hexadecimal:
 - $1101 = 13 = D$
 - $0110 = 6$
 - $1111 = 15 = F$
 - $1001 = 9$
- Answer: D6F9
- What about from *binary* to *octal*?

Signed Numbers and Complements

Range of represented numbers

- Let ℓ be the number of binary digits that can be stored. 
- Example: Store data in a single byte (8 bits).
- Using a single byte can represent unsigned numbers from 0 to 255 ($2^8 = 256$ different values).
- Alternatively, can represent the signed numbers from -128 to 127 in same amount of space ($2^7 = 128$).

Signed Numbers and Complements

- How to denote if a number is positive or negative?
 - Use a sign bit: 0_s1001 denotes positive 9, 1_s1001 denotes negative 9. This representation is called the **sign-magnitude representation**.
 - This works, but it will be convenient to use a different representation of negative numbers.
- Two methods: **r 's complement** and **$(r - 1)$'s complement**.

r 's Complement

100000000

$\leftarrow 2^8 - N$

0	0	0	0	1	0	0
1	1	0	1	1	0	0

- r 's complement of $N = r^{\ell} - N = 10_{(2)}^{\ell} - N$
- In our example (one byte of memory, to represent -9, (where $9 = 1001$ in binary), compute $(10)^8 - 1001 = 100000000 - 1001 = 11110111$

flip all bits and add 1.

"flip all bits to the left of the first 1.
all other bits remain the same."
- Notice for negative numbers, most significant bit is always 1. For positive numbers, most significant bit is always 0.
- This bit is therefore called the **sign bit**.
- Aside: This is equivalent to doing arithmetic modulo r^{ℓ} .

Subtraction Terminology

$$\textcircled{10110} - \textcircled{1000}$$

minuend

subtrahend

- Find the 2's complement of the subtrahend
- Perform addition
- Ignore the carry-out bit

Subtraction using r 's complement

- take 2's complement
- Just do addition as usual
- Ignore highest order carry
- This is always correct unless there is **overflow**.

Example of subtraction using r 's complement

- Assume $\ell = 6$:

- Compute:

$$11010 - 00111$$

$$\begin{array}{r} 11 \\ 011010 \\ + 111001 \\ \hline \cancel{X}010011 \end{array}$$

- Compute:

$$000111 - 011010$$

$$\begin{array}{r} 11 \\ 000111 \\ + 100110 \\ \hline 101101 \end{array}$$

Overflow in r 's complement

- Overflow occurs in the following cases:

Operation	Operand A	Operand B	Result
A + B	≥ 0	≥ 0	< 0
A+B	< 0	< 0	≥ 0
A-B	≥ 0	< 0	< 0
A-B	< 0	≥ 0	≥ 0

- These conditions are the same as:
Carry-In to sign position $\neq$ Carry-Out from sign position

Example of Overflow in r 's complement

- Assume $\ell = 6$
- Compute:

$$11100 + 10111$$

1's Complement.

$(r - 1)$'s Complement

- $(r - 1)$'s complement of $N = r^{\ell} - \cancel{r^m} - N = 10^{\ell} - \cancel{10^m} - N$
1
- For integers, $m = 0$, so subtract from $10^n - 1$
- In our example, to represent -9, (where $9 = 1001$ in binary), compute $(10)^8 - 1 - 1001 = 11111111 - 1001 = 11110110$
- This corresponds to flipping the bits of 00001001.
- Again, for negative numbers, nth digit is always 1.
For positive numbers, nth digit is always 0.
- There are now two ways to represent 0:
00000000 or 11111111

}
Same as
before

Subtraction using

$(r - 1)$'s complement

- Take the 1's complement of the subtrahend
- Do addition as usual
- If there is an end carry, add it to the least significant bit. 
- Most significant bit tells you the sign (unless overflow occurs).

Example of subtraction using $(r - 1)$'s complement

- Assume $\ell = 6$

- Compute:

$$11010 - 00111$$

$$\begin{array}{r} 11 \\ 011010 \\ + 111000 \\ \hline 1010010 \end{array} \rightarrow \boxed{010011}$$

(Note: An arrow in the original image points from the last three digits '010' of the sum to the boxed result '011', indicating a carry or adjustment.)

- Compute:

$$00111 - 11010$$

$$\begin{array}{r} 111 \\ 000111 \\ + 100101 \\ \hline \boxed{101100} \end{array}$$

Fast(er) way to compute 2's complement

- To form the 2's complement of 0110 1010:
 - Take the 1s complement: 1001 0101
 - Then add 1: 1001 0110

Ways to Represent Signed Numbers

Decimal	Signed-2's Complement	Signed-1's Complement	Signed Magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	—	1111	1000
-1	1111	1110	1001
-2	1110	1101	1010
-3	1101	1100	1011
-4	1100	1011	1100
-5	1011	1010	1101
-6	1010	1001	1110
-7	1001	1000	1111
-8	1000	—	—

Advantages/Disadvantages of 1's vs. 2's complement

1s complement	2s complement
Easy to compute (just flip bits)	Harder to compute (flip bits and add one)
Harder to manipulate (e.g., for subtraction, need to add in extra carry.)	Easy to manipulate (e.g., subtraction is the same as addition—no extra hardware needed)

Binary Codes

Decimal Codes

- Different ways to represent individual decimal digits in binary
- Adding encoded digits requires a dedicated adder for each code
- As we will see, different encodings can provide different advantages.

BCD Code

Decimal Symbol	BCD Digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Advantage: Decimal addition can be performed over the binary representation. Just need to add 110 to the sum if it is equal to 10 or above.

Other Codes

Decimal Digit	BCD 8421	2421	Excess-3	8, 4, -2, -1
0	0000	0000	0011	0000
1	0001	0001	0100	0111
2	0010	0010	0101	0110
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1011
6	0110	1100	1001	1010
7	0111	1101	1010	1001
8	1000	1110	1011	1000
9	1001	1111	1100	1111
	1010	0101	0000	0001
Unused bit combinations	1011	0110	0001	0010
	1100	0111	0010	0011
	1101	1000	1101	1100
	1110	1001	1110	1101
	1111	1010	1111	1110

Weighted codes: BCD (8421), 2421, 8,4,-2,1

Each binary digit is multiplied by the corresponding weight to obtain the decimal value.

Other Codes

Decimal Digit	BCD 8421	2421	Excess-3	8, 4, -2, -1
0	0000	0000	0011	0000
1	0001	0001	0100	0111
2	0010	0010	0101	0110
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1011
6	0110	1100	1001	1010
7	0111	1101	1010	1001
8	1000	1110	1011	1000
9	1001	1111	1100	1111
	1010	0101	0000	0001
Unused bit combinations	1011	0110	0001	0010
	1100	0111	0010	0011
	1101	1000	1101	1100
	1110	1001	1110	1101
	1111	1010	1111	1110

Excess 3 code: obtained from the binary value plus 3

Other Codes

Decimal Digit	BCD 8421	2421	Excess-3	8, 4, -2, -1
0	0000	0000	0011	0000
1	0001	0001	0100	0111
2	0010	0010	0101	0110
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1011
6	0110	1100	1001	1010
7	0111	1101	1010	1001
8	1000	1110	1011	1000
9	1001	1111	1100	1111
	1010	0101	0000	0001
Unused bit combinations	1011	0110	0001	0010
	1100	0111	0010	0011
	1101	1000	1101	1100
	1110	1001	1110	1101
	1111	1010	1111	1110

Advantages: 2421, Excess-3, 8,4,-2,-1 codes are **self-complementing**
 The 9's complement is obtained by changing 1's to 0's and 0's to 1's.

Other Codes

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

Advantage: Consecutive values differ in exactly one bit

If multiple bits must change when transitioning from one number to the next, they may not switch simultaneously in a physical device, potentially producing incorrect intermediate values.

ASCII Character Code

- American Standard Code for Information Interchange (ASCII)
- Binary code for representing *characters*
- Includes decimal digits (0-9) as before but now also includes
 - Letters (uppercase and lowercase)
 - Special characters (such as \$#@!)
- Uses seven bits to encode 128 characters

ASCII Character Code

$b_4b_3b_2b_1$	$b_7b_6b_5$							
	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	'	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	\$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L		l	
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	-	o	DEL

Control Characters

NUL	Null	DLE	Data-link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End-of-transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete