Course Overview

URL: http://www.ee.nmt.edu/~erives/classes.php

Texts:

- Class Notes (Available online)
- Freescale Databooks on the MC9S12 (Available online)
- **The HCS12/9S12: An Introduction to Software and Interfacing, 2nd Edition** by Han-Way Huang (Not required)

Grading:

- 15%: Homework
- 15%: Quizzes (given regularly every Friday)
- 30%: Two midterms exams
- 15%: Final exam
- 25%: Laboratory grade

Late work will have a 25% penalty. You need to pass the Laboratory to be able to pass the course.

- Introduction to digital electronics
- Introduction to the MC9S12 Microcontroller
- Binary and Hexadecimal Numbers
- Assembly Language Programming
- C Language Programming
- Introduction to MC9S12 Internal Peripherals
  - The MC9S12 Timer Subsystem
  - Interrupts using the Timer Subsystem
  - The MC9S12 Pulse Width Modulator Subsystem
- The MC9S12 Expanded Mode
  - Address and Data Buses and Timing
  - Adding Memory and External Peripherals
  - Interfacing to the MC9S12
- More MC9S12 Internal Peripherals
  - The A/D Converter Subsystem
  - The Serial Peripheral Interface
  - The Serial Communications Interface
- Using the MC9S12 in a Control Application

Lab Overview

• **No labs this week, you will be notified in advance.**

• Lab handouts will be posted starting the following week.

• The 9S12 evaluation kits will be passed out in lab.

• You need to bring a bound lab notebook to the first lab.

• There will be a prelab for each lab. This must be done and turned in at the start of your lab section. The lab TA will verify that you have completed the prelab.

• Be prepared to answer questions about the pre-lab when you come to lab.

• If you do not complete the prelab before coming to lab, you will lose a high percentage of the points for that lab.

- **Introduction to Digital Electronics.**

Binary Numbers

A decimal integer is expressed by an n-tuple comprising n decimal digits

$$D = d_{n-1}d_{n-2}\cdots d_1 d_0$$

which represents the value

$$V(D) = d_{n-1} \times 10^{n-1} + d_{n-2} \times 10^{n-2} + \cdots + d_1 \times 10^1 + d_0 \times 10^0$$

This is referred to as the *positional number representation*

In the binary number system, the same positional number representation is used so that

$$B = b_{n-1}b_{n-2}\cdots b_1 b_0$$

represents an integer that has the value

$$V(B) = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$
$$= \sum_{i=0}^{n-1} b_i \times 2^i$$

For example the binary number 1101 represents the value

$$V = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$(1101)_2 = (13)_{10}$$

The following table shows the first 15 positive integers and their binary representations

Decimal representation	Binary representation
00	0000
01	0001
02	0010
03	0011
04	0100
05	0101
06	0110
07	0111
08	1000
09	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Converting a decimal number into a binary number is not quite straightforward

Convert $(857)_{10}$

			Remainder	
$857 \div 2$	=	428	1	LSB
$428 \div 2$	=	214	0	
$214 \div 2$	=	107	0	
$107 \div 2$	=	53	1	
$53 \div 2$	=	26	1	
$26 \div 2$	=	13	0	
$13 \div 2$	=	6	1	
$6 \div 2$	=	3	0	
$3 \div 2$	=	1	1	
$1 \div 2$	=	0	1	MSB

Result is $(1101011001)_2$

Truth Table

Logic operations can be defined in the form of a table

x_1	x_2	x_3	$x_1 \cdot x_2 \cdot x_3$	$x_1 + x_2 + x_3$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Three basic logic operations are AND, OR, and NOT. A complex function may require hundreds or thousands of these basic operations for its implementation

Another gate is the XOR gate

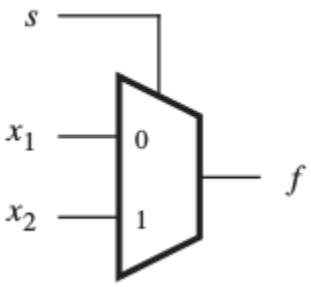
The truth table for the XOR function is

x	y	L
0	0	0
0	1	1
1	0	1
1	1	0

Multiplexer circuit

In computer systems it is often necessary to choose data form exactly one of a number of sources

A circuit that implements this function is called a *multiplexer circuit*



s	$f(s, x_1, x_2)$
0	x_1
1	x_2

Positional Number Representation

The most practical representations are:

- Decimal
- Binary
- Octal
- Hexadecimal

Decimal	Binary	Octal	Hexadecimal
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10
17	10001	21	11
18	10010	22	12

In computers the dominant number system is **binary.**

The reason for using octal and hexadecimal systems is that they serve as a useful shorthand notation for binary numbers.

For example the binary number 101011010111 is convected to octal by making groups of 3 bits

$$\underbrace{101}_{5}\quad\underbrace{011}_{3}\quad\underbrace{010}_{2}\quad\underbrace{111}_{7}$$

$$(101011010111)_2 = (5327)_8.$$

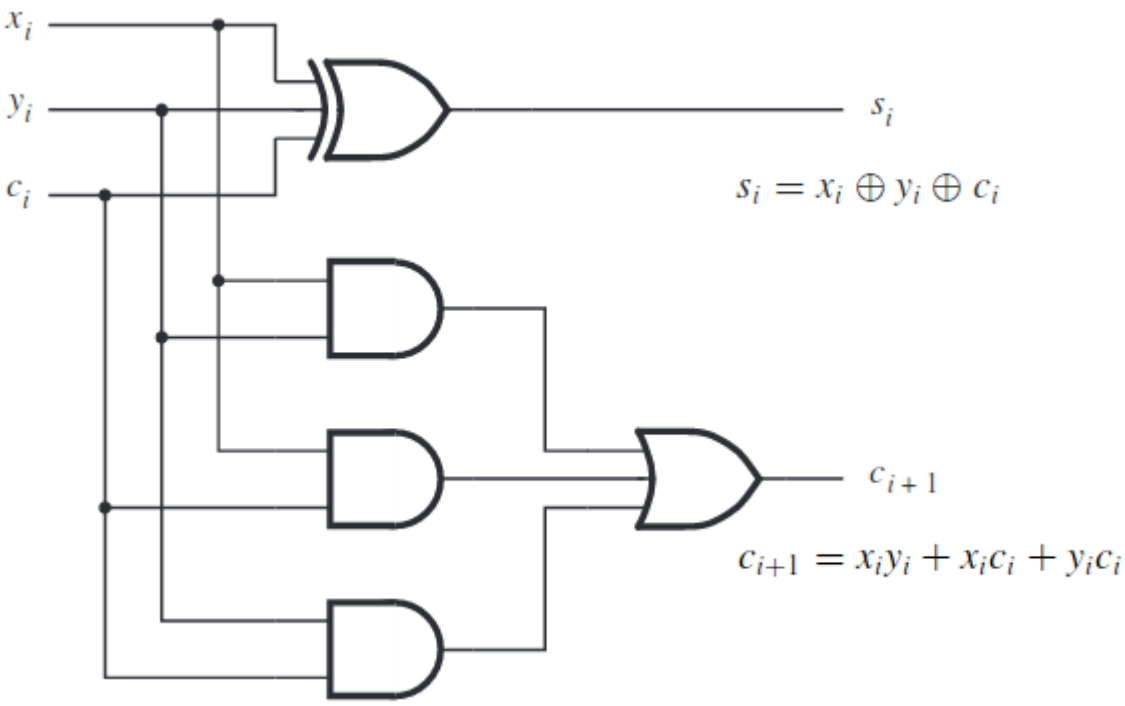
For example a 16-bit number is represented by 4 hex digits

$$\underbrace{1010}_{\text{A}}\quad\underbrace{1111}_{\text{F}}\quad\underbrace{0010}_{2}\quad\underbrace{0101}_{5}$$

$$(1010111100100101)_2 = (\text{AF25})_{16}$$

Addition of binary numbers

The additions of a binary number can be accomplished with XOR and AND gates



Generated carries →	1 1 1 0	
$X = x_4 x_3 x_2 x_1 x_0$	0 1 1 1 1	$(15)_{10}$
$+ Y = y_4 y_3 y_2 y_1 y_0$	+ 0 1 0 1 0	+ $(10)_{10}$
$S = s_4 s_3 s_2 s_1 s_0$	1 1 0 0 1	$(25)_{10}$

Signed Numbers

One representation of binary negative numbers is the **1's complement** scheme

The 1's complement negative number can be obtained simply by complementing each bit of the number, including the sign bit, which in signed numbers it is the MSB (most significant bit).

For example to convert +5 to a negative, we will have:

$(+5)_{10} = (0101)_2$ and $(-5)_{10} = (1010)_2$

Signed number are usually expressed using **2's complement representation.** A simple way of finding a 2's complement of a number is add 1 to its 1's complement.

For example to convert +5 to a negative, we will have:

$(+5)_{10} = (0101)_2$ and $(-5)_{10} = (1010 + 1)_2 = (1011)2$

$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	–0	–7	–8
1001	–1	–6	–7
1010	–2	–5	–6
1011	–3	–4	–5
1100	–4	–3	–4
1101	–5	–2	–3
1110	–6	–1	–2
1111	–7	–0	–1

Addition and Subtraction

An obvious advantage of the 1's complement representation is that a negative number is generated simply by complementing all bits.

However, addition of 1's complement numbers MAY or MAY NOT be simple, depending on the numbers being added.

Lets consider the following cases:

(+5)	0101	(–5)	1010
+ (+2)	+ 0010	+ (+2)	+ 0010
(+7)	0111	(–3)	1100

(+5)	0101	(–5)	1010
+ (–2)	+ 1101	+ (–2)	+ 1101
(+3)	10010	(–7)	10111
	→ 1		→ 1
	0011		1000

In **2's complement addition**, if there is a carry-out from the sign-bit position, it is simply *ignored*

Lets consider the following cases:

(+5)	0101	(–5)	1011
+ (+2)	+ 0010	+ (+2)	+ 0010
(+7)	0111	(–3)	1101

(+5)	0101	(–5)	1011
+ (–2)	+ 1110	+ (–2)	+ 1110
(+3)	10011	(–7)	11001
	↑ ignore		↑ ignore

In **2's complement subtraction**, the easies way of performing subtraction is done by finding the 2's complement of the subtrahend and then performing addition.

(+5)	0101		0101
– (+2)	– 0010	⇨	+ 1110
(+3)			10011
			↑ ignore
(–5)	1011		1011
– (+2)	– 0010	⇨	+ 1110
(–7)			11001
			↑ ignore
(+5)	0101		0101
– (–2)	– 1110	⇨	+ 0010
(+7)			0111
(–5)	1011		1011
– (–2)	– 1110	⇨	+ 0010
(–3)			1101

Arithmetic Overflow

The result of addition and subtraction is supposed to fit within the significant bits used to represent the numbers.

If n bits are used to represent signed numbers, then the result must be in the range -2^{n-1} to $2^{n-1}-1$.

If the result DOES NOT fit in this range, then we say that an **arithmetic overflow** has occurred.

So to ensure the correct operation of an arithmetic circuit, it is important to be able to detect the occurrence of overflow.

The following examples show the occurrence of overflow:

(+7)	0 1 1 1	(–7)	1 0 0 1
+ (+2)	+ 0 0 1 0	+ (+2)	+ 0 0 1 0
(+9)	1 0 0 1	(–5)	1 0 1 1
	$c_4 = 0$		$c_4 = 0$
	$c_3 = 1$		$c_3 = 0$
(+7)	0 1 1 1	(–7)	1 0 0 1
+ (–2)	+ 1 1 1 0	+ (–2)	+ 1 1 1 0
(+5)	1 0 1 0 1	(–9)	1 0 1 1 1
	$c_4 = 1$		$c_4 = 1$
	$c_3 = 1$		$c_3 = 0$

When the numbers have opposite signs, there is no overflow

But if both numbers have the same sign, overflow occurs

Therefore we can observe that:

$$\begin{aligned}\text{Overflow} &= c_3\bar{c}_4 + \bar{c}_3 c_4 \\ &= c_3 \oplus c_4\end{aligned}$$

So in general, overflow can be computed by using the following equation:

$$\text{Overflow} = c_{n-1} \oplus c_n$$

Binary-Coded-Decimal Representation

In digital systems it is possible to represent decimal numbers simply by encoding each digit in binary form

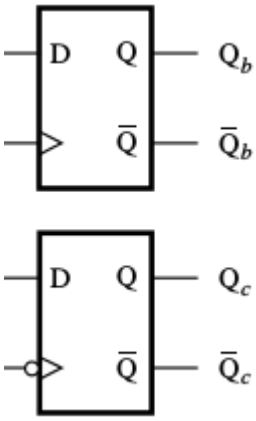
This is called binary-coded-decimal (BCD) representations

Because there are 10 digits to encode, it is necessary to use 4 bits/digit

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

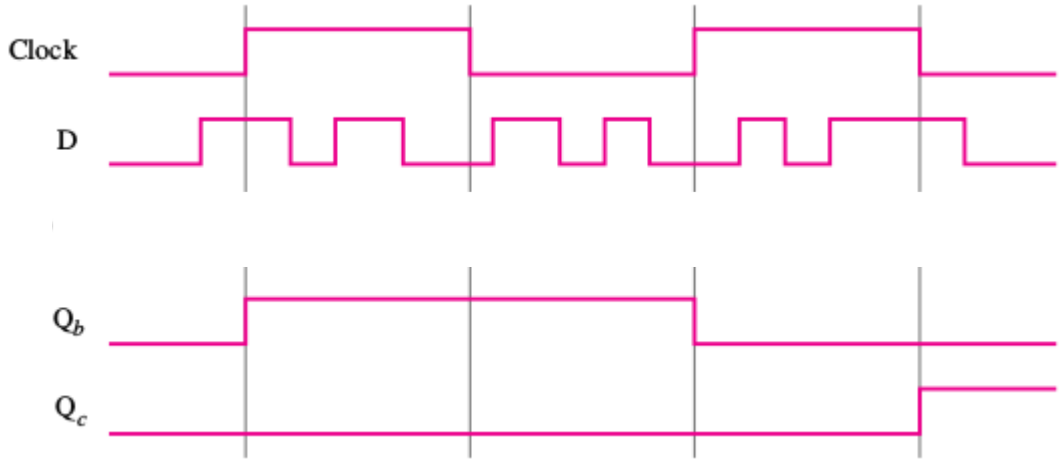
Flip-flops, Registers, and Counters

The term flip-flop denotes a **storage element** that changes its output state at the edge of a controlling **clock signal**.



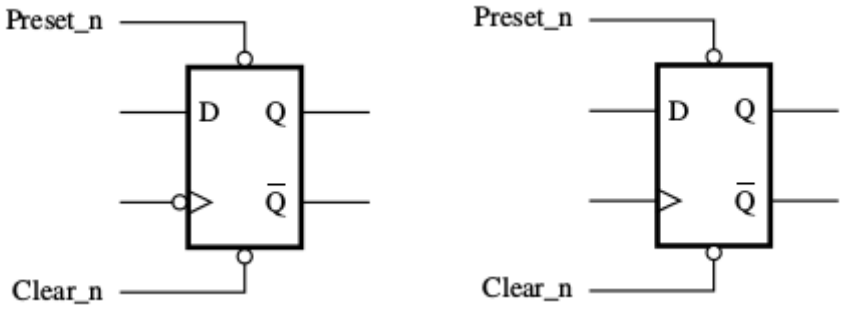
In the symbols we use the > mark to denote that the flip-flop responds to the active edge of a **clock signal**.

A *bubble* on the clock input indicates that the active edge for that particular circuit is the *negative edge* (versus the *positive-edge* triggered D flip-flop).

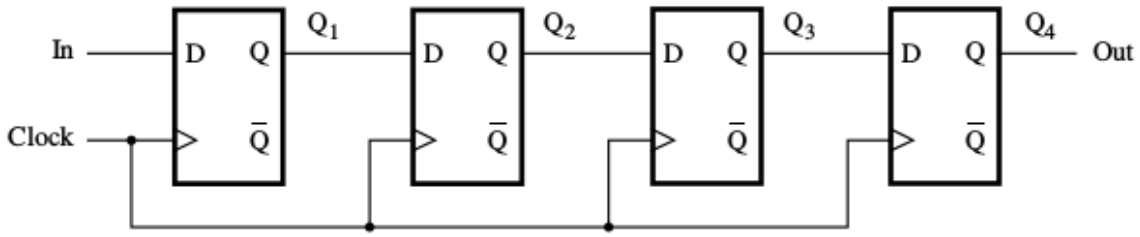


To make flip-flops more flexible, two other inputs are required:

- preset
- reset (clear)



To transfer an n-bit data item serially, we can use a SHIFT REGISTER



	In	Q_1	Q_2	Q_3	Q_4 = Out
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1