28. Review

Dec. 10, 2018

- Algorithms
- Using building blocks
- Binary system
- Conversion between binary and decimal
- Sign extension and overflow
- Other bases

REVIEW SESSION
DEC. 16 (SUNDAY) 2:00 PM --
(HERE) EER 1.516

TA OFFICE HOURS (EXTRA)
MON. DEC. 17
EER 4.704 2-5 PM
6 PM---

Converting Binary (2's C) to Decimal

1. If leading bit is one, take two's complement to get a positive number.
2. Add powers of 2 that have "1" in the corresponding bit positions.
3. If original number was negative, add a minus sign.

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

$$X = 01101000_{two}$$
$$= 2^6+2^5+2^3 = 64+32+8$$
$$= 104_{ten}$$

Assuming 8-bit 2's complement numbers.

Converting Decimal to Binary (2's C)

First Method: *Division*

1. Find magnitude of decimal number. (Always positive.)
2. Divide by two – remainder is least significant bit.
3. Keep dividing by two until answer is zero, writing remainders from right to left.
4. Append a zero as the MS bit; if original number was negative, take two's complement.

$X = 104_{ten}$

104/2	=	52 r0	*bit 0*
52/2	=	26 r0	*bit 1*
26/2	=	13 r0	*bit 2*
13/2	=	6 r1	*bit 3*
6/2	=	3 r0	*bit 4*
3/2	=	1 r1	*bit 5*
1/2	=	0 r1	*bit 6*

$X = 01101000_{two}$

Sign Extension

To add two numbers, we must represent them with the same number of bits.

If we just pad with zeroes on the left:

<u>4-bit</u>	<u>8-bit</u>
`0100` (4)	`00000100` (still 4)
`1100` (-4)	`00001100` (12, not -4)

Instead, replicate the MS bit -- the sign bit:

<u>4-bit</u>	<u>8-bit</u>
`0100` (4)	`00000100` (still 4)
`1100` (-4)	`11111100` (still -4)

Overflow

If operands are too big, then sum cannot be represented as an *n*-bit 2's comp number.

```
  01000  (8)          11000  (-8)
+ 01001  (9)        + 10111  (-9)
  10001  (-15)        01111  (+15)
```

We have overflow if:

- **signs of both operands are the same, and**
- **sign of sum is different.**

Another test -- easy for hardware:

- **carry into MS bit does not equal carry out**

↳ MOST SIGNIFICAND

Converting from Binary to Hexadecimal

Every four bits is a hex digit.

- **start grouping from right-hand side**

```
 011 1010 1000 1111 0100 1101 0111
  ⇓   ⇓    ⇓    ⇓    ⇓    ⇓    ⇓
  3   A    8    F    4    D    7
```

This is not a new machine representation, just a convenient way to write the number.

Examples of Logical Operations

AND USEFUL AS "MASKS"

- **useful for clearing bits**
  - ➢ **AND with zero = 0**
  - ➢ **AND with one = no change**

```
      11000101
AND   00001111
      00000101
```

OR

- **useful for setting bits**
  - ➢ **OR with zero = no change**
  - ➢ **OR with one = 1**

```
      11000101
OR    00001111
      11001111
```

NOT

- **unary operation -- one argument**
- **flips every bit**

```
NOT   11000101
      00111010
```

Fractions: Fixed-Point

How can we represent fractions?

- **Use a "binary point" to separate positive from negative powers of two -- just like "decimal point."**
- **2's comp addition and subtraction still work.**
  - ➢ **if binary points are aligned**

$2^{-1} = 0.5$

$2^{-2} = 0.25$

$2^{-3} = 0.125$

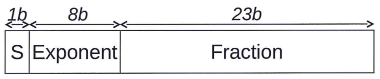
```
  00101000.101 (40.625)
+ 11111110.110 (-1.25)
  00100111.011 (39.375)
```

No new operations -- same as integer arithmetic.

Very Large and Very Small: Floating-Point

Large values: 6.023 x 10^{23} -- requires 79 bits
Small values: 6.626 x 10^{-34} -- requires >110 bits

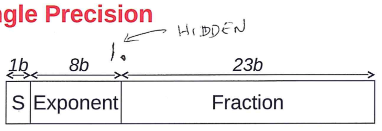
Use equivalent of “scientific notation”: F x 2^{E}
Need to represent F (*fraction*), E (*exponent*), and sign.
IEEE 754 Floating-Point Standard (32-bits):



$$N=(-1)^{S}\times 1.\text{fraction}\times 2^{\text{exponent}-127},\ 1\le \text{exponent}\le 254$$
$$N=(-1)^{S}\times 0.\text{fraction}\times 2^{-126},\ \text{exponent}=0$$

IEEE 754 Floating Point Standard

Single Precision



$$N=(-1)^{S}\times 1.\text{fraction}\times 2^{\text{exponent}-127},\ 1\le \text{exponent}\le 254$$
$$N=(-1)^{S}\times 0.\text{fraction}\times 2^{-126},\ \text{exponent}=0$$

Floating Point Example

Single-precision IEEE floating point number:

$$\underbrace{1}_{\text{sign}}\underbrace{01111110}_{\text{exponent}}\underbrace{10000000000000000000000}_{\text{fraction}}$$

- **Sign is 1 – number is negative.**
- **Exponent field is 01111110 = 126 (decimal).**
- **Fraction is 0.100000000000… = 0.5 (decimal).**

Value = -1.5 x $2^{(126-127)}$ = -1.5 x 2^{-1} = -0.75.

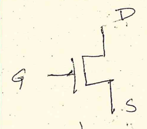
Text: ASCII Characters

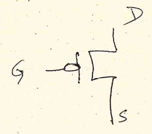
ASCII: Maps 128 characters to 7-bit code.
- **both printable and non-printable (ESC, DEL, …) characters**

00	nul	10	dle	20	sp	30	0	40	@	50	P	60	`	70	p
01	soh	11	dc1	21	!	31	1	41	A	51	Q	61	a	71	q
02	stx	12	dc2	22	"	32	2	42	B	52	R	62	b	72	r
03	etx	13	dc3	23	#	33	3	43	C	53	S	63	c	73	s
04	eot	14	dc4	24	$	34	4	44	D	54	T	64	d	74	t
05	enq	15	nak	25	%	35	5	45	E	55	U	65	e	75	u
06	ack	16	syn	26	&	36	6	46	F	56	V	66	f	76	v
07	bel	17	etb	27	'	37	7	47	G	57	W	67	g	77	w
08	bs	18	can	28	(	38	8	48	H	58	X	68	h	78	x
09	ht	19	em	29	)	39	9	49	I	59	Y	69	i	79	y
0a	nl	1a	sub	2a	*	3a	:	4a	J	5a	Z	6a	j	7a	z
0b	vt	1b	esc	2b	+	3b	;	4b	K	5b	[	6b	k	7b	{
0c	np	1c	fs	2c	,	3c	<	4c	L	5c	\	6c	l	7c	|
0d	cr	1d	gs	2d	-	3d	=	4d	M	5d	]	6d	m	7d	}
0e	so	1e	rs	2e	.	3e	>	4e	N	5e	^	6e	n	7e	~
0f	si	1f	us	2f	/	3f	?	4f	O	5f	_	6f	o	7f	del

CMOS

↳ TRANSISTORS TO (LOGIC) GATES





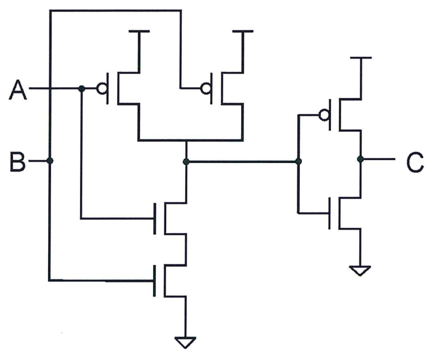
n-channel
~~ON~~ CONDUCTS IF G=1V (EXAMPLE) LOGIC 1

p-channel
CONDUCTS IF G=0V ~~LOGIC~~ (LOGIC 0)

G: GATE
S: SOURCE
D: DRAIN

LIKE A SWITCH

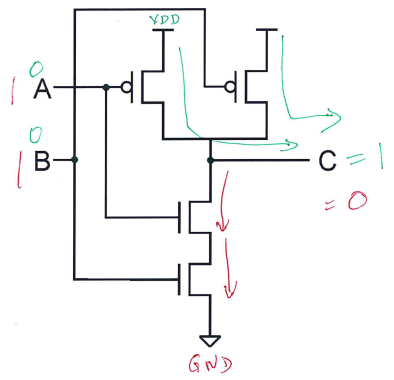
AND Gate

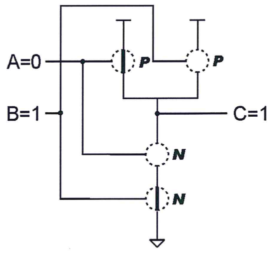


A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Add inverter to NAND.

NAND Gate (AND-NOT)



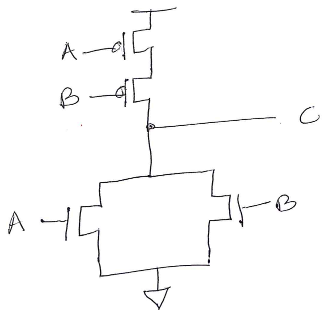


A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

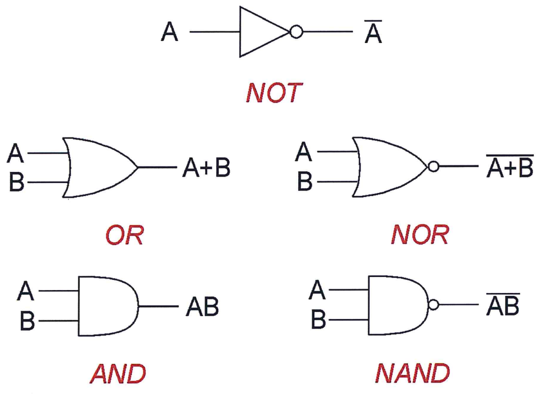
Note: Parallel structure on top, serial on bottom.

NOR GATE

(OR-NOT)



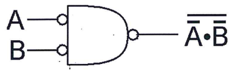
Basic Logic Gates



DeMorgan's Law

Converting AND to OR (with some help from NOT)

Consider the following gate:



A	B	$\overline{A}$	$\overline{B}$	$\overline{A} \cdot \overline{B}$	$\overline{\overline{A} \cdot \overline{B}}$
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

Same as A+B!

To convert AND to OR (or vice versa), invert inputs and output.

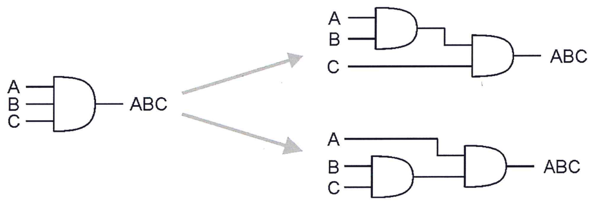
$\overline{(X + Y)} = \overline{X} \cdot \overline{Y}$

More than 2 Inputs?

AND/OR can take any number of inputs.

- **AND = 1 if all inputs are 1.**
- **OR = 1 if any input is 1.**
- **Similar for NAND/NOR.**

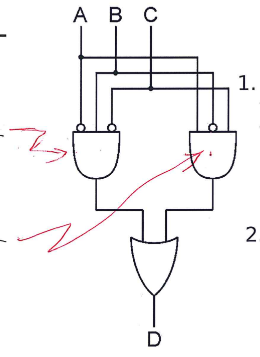
Can implement with multiple two-input gates, or with single CMOS circuit.



Logical Completeness

Can implement ANY truth table with AND, OR, NOT.

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



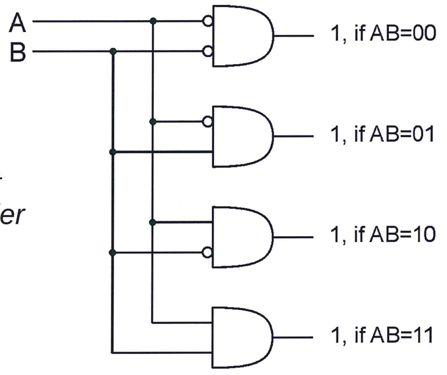
1. AND combinations that yield a "1" in the truth table.

2. OR the results of the AND gates.

Decoder

***n* inputs, 2^n outputs**

- **exactly one output is 1 for each possible input pattern**



2-bit decoder

ENCODER: 2^n INPUTS (ONE-HOT), n OUTPUTS

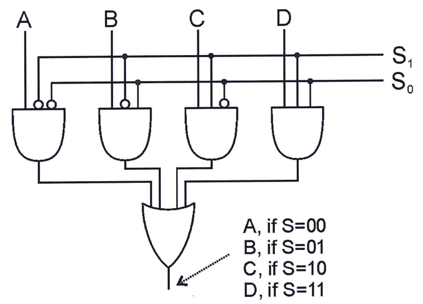
W	X	y	Z	A B
0	0	0	1	0 0
0	0	1	0	0 1
0	1	0	0	1 0
1	0	0	0	1 1

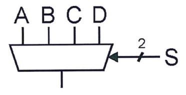
INVERSE OF DECODER

Multiplexer (MUX)

***n*-bit selector and *2^n* inputs, one output**

- **output equals one of the inputs, depending on selector**





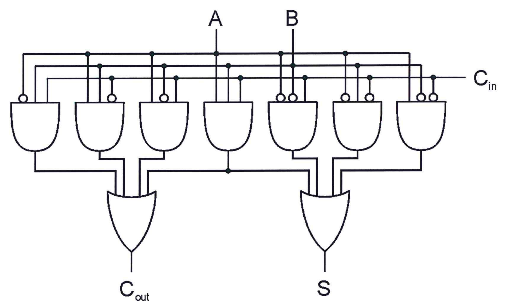
4-to-1 MUX

PRIORITY ENCODER

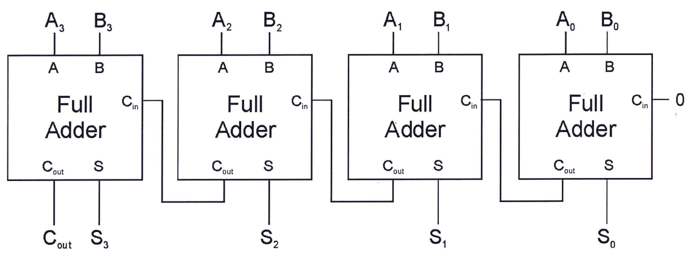
HIGH w	x	y	LOW z	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Full Adder

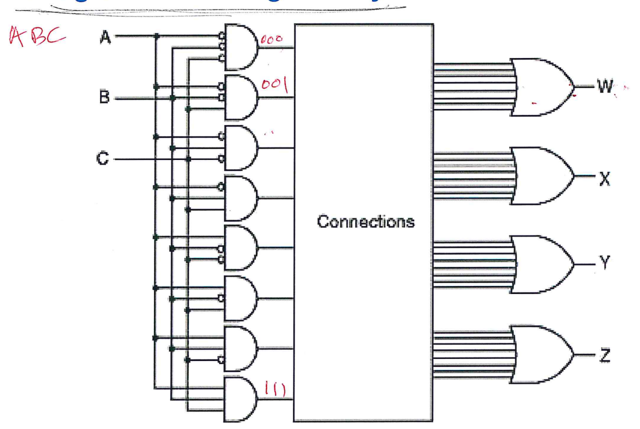
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



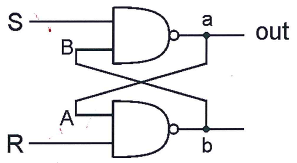
Four-bit Adder



Programmable Logic Array

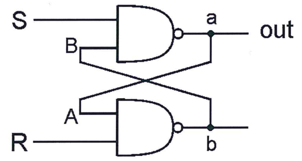


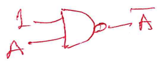
R-S Latch: Simple Storage Element

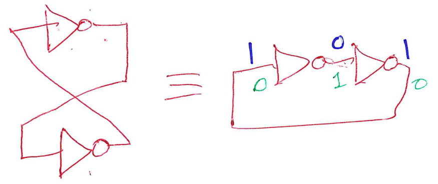


S	R	OUT
1	1	PREV. VAL.
0	1	1
1	0	0
0	0	NA

0



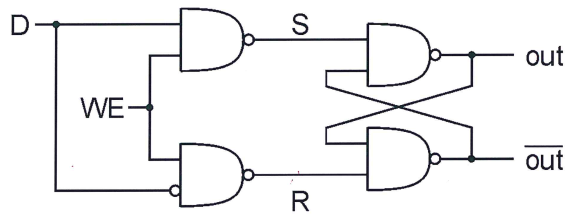




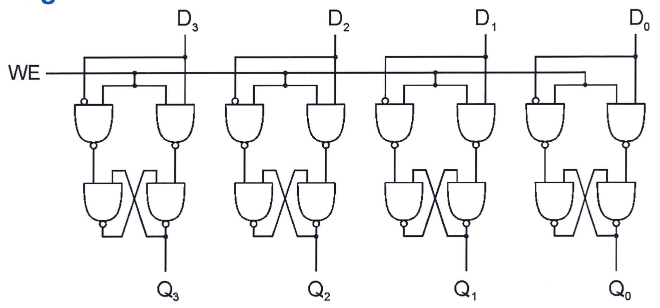
Gated D-Latch

Two inputs: D (data) and WE (write enable)

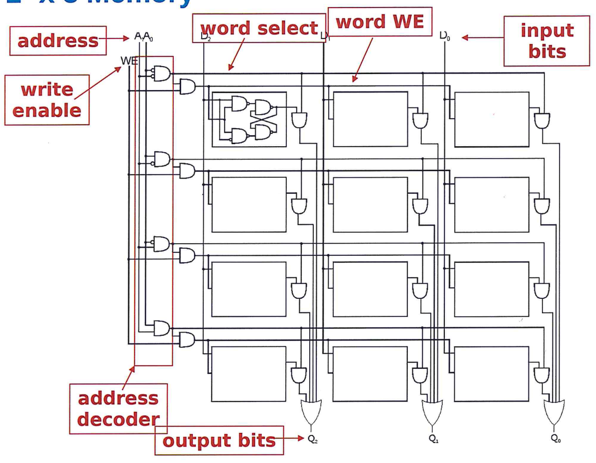
- **when WE = 1, latch is set to value of D**
  - ➢ **S = NOT(D), R = D**
- **when WE = 0, latch holds previous value**
  - ➢ **S = R = 1**



Register



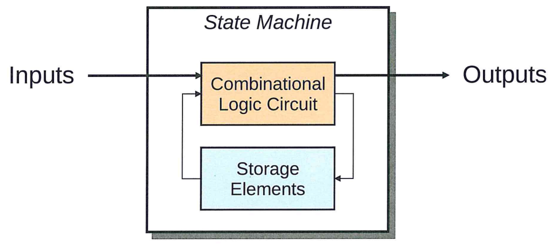
2^2 x 3 Memory



State Machine

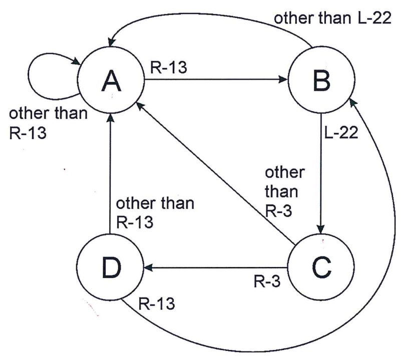
Another type of sequential circuit

- **Combines combinational logic with storage**
- **"Remembers" state, and changes output (and state) based on inputs and current state**



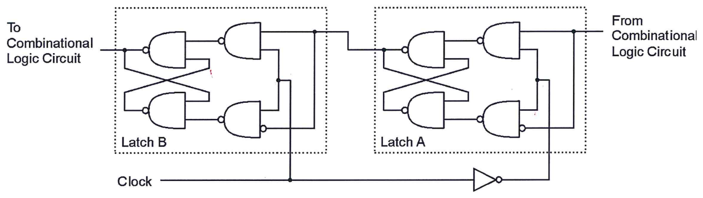
State Diagram

Shows states and actions that cause a transition between states.



Storage: Master-Slave Flipflop

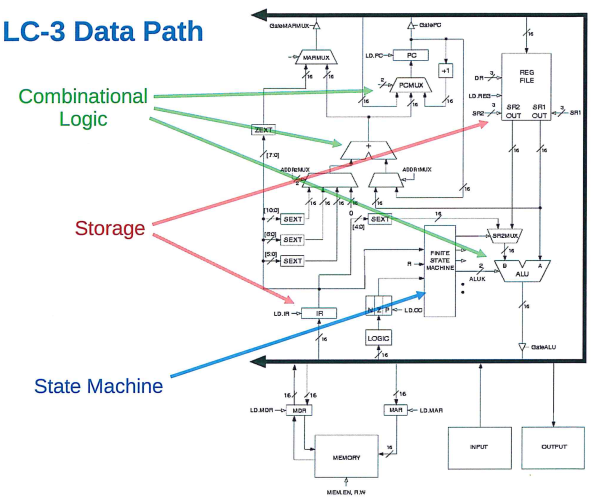
A pair of gated D-latches, to isolate *next* state from *current* state.


To Combinational Logic Circuit
From Combinational Logic Circuit
Latch B
Latch A
Clock

During 1st phase (clock=1), previously-computed state becomes *current* state and is sent to the logic circuit.

During 2nd phase (clock=0), *next* state, computed by logic circuit, is stored in Latch A.

LC-3 Data Path


Combinational Logic
Storage
State Machine
GateMARMUX
GatePC
MARMUX
LD.PC
PC
PCMUX
+1
REG FILE
DR
LD.REG
SR2
SR2 OUT
SR1 OUT
SR1
ZEXT
[7:0]
ADDR1MUX
ADDR2MUX
[10:0]
[8:0]
[5:0]
SEXT
[4:0]
FINITE STATE MACHINE
R
SR2MUX
ALU
ALUK
IR
LD.IR
N Z P
LD.CC
LOGIC
GateALU
MDR
LD.MDR
MAR
LD.MAR
MEMORY
MEM.EN, R.W
INPUT
OUTPUT