Review of
Combinational Circuits

Review of Combinational Circuits

1. Fundamental Gates

- AND and NAND
- OR and NOR
- XOR and XNOR
- Buffers (regular, tri-state, open-collector)

2. Combinational Logic Circuits

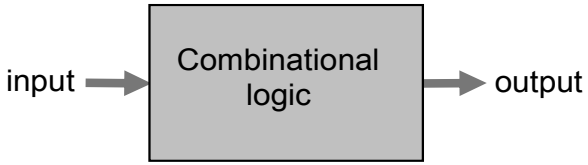
- Multiplexer
- Encoders and decoders
- Parity detector
- Priority encoder

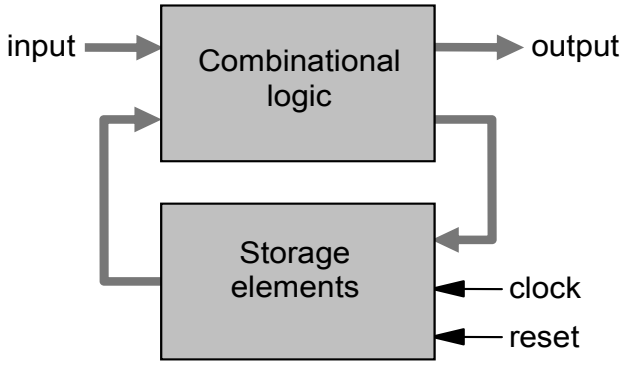
3. Combinational Arithmetic Circuits

- Basic adders
- Fast adders
- Signed adders/subtracters
- Comparators
- ALU (arithmetic-logic unit)
- Multipliers
- Dividers

Introduction

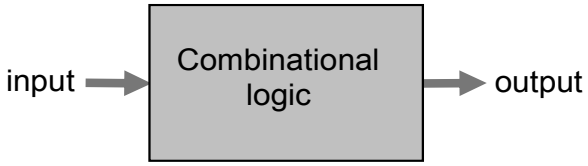
Combinational versus Sequential

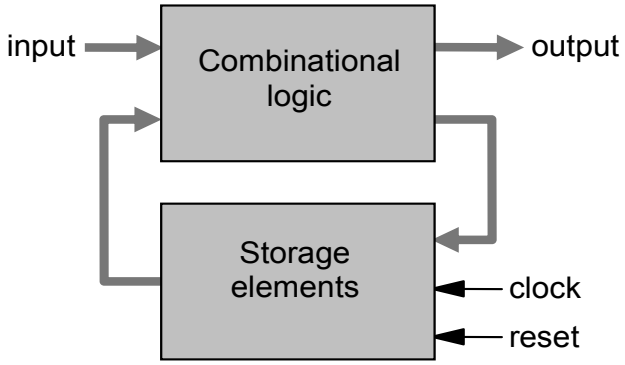


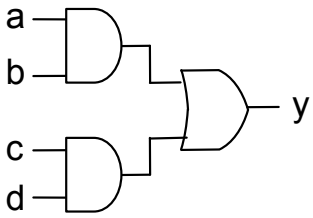


Introduction

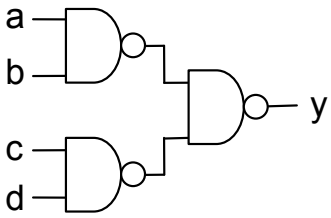
Combinational versus Sequential





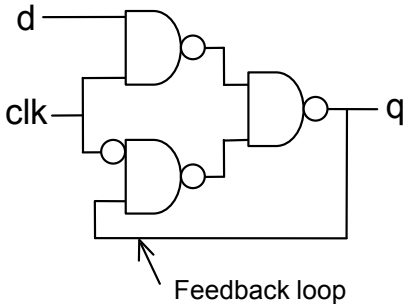


y = a.b + c.d



y = a.b + c.d

Non-recursive



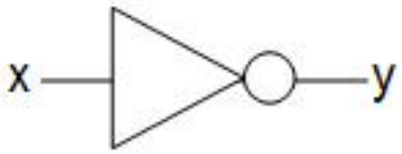
q = d.clk + q.clk′

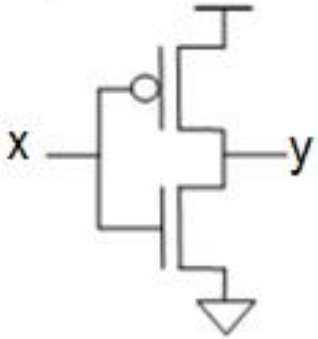
Recursive → Has memory!

1. Fundamental gates

- NOT
- AND and NAND
- OR and NOR
- XOR and XNOR
- Buffers

NOT



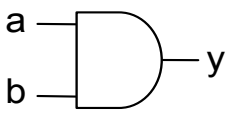


y = x’

y = NOT x

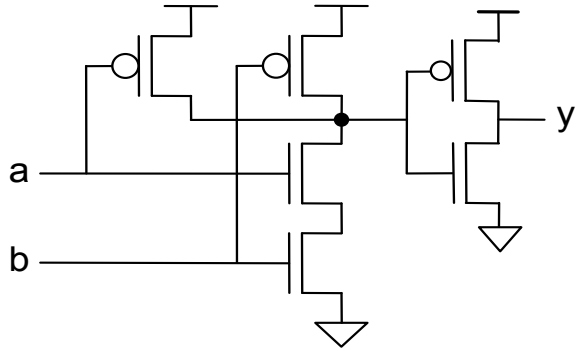
AND and NAND

AND

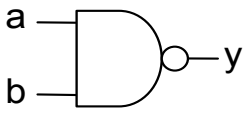


$y=a \cdot b$

a b	y
0 0	0
0 1	0
1 0	0
1 1	1

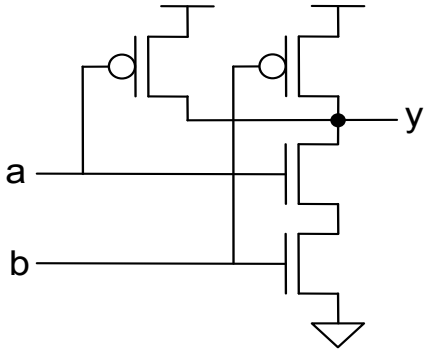


NAND



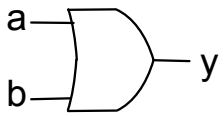
$y=(a \cdot b)'$

a b	y
0 0	1
0 1	1
1 0	1
1 1	0



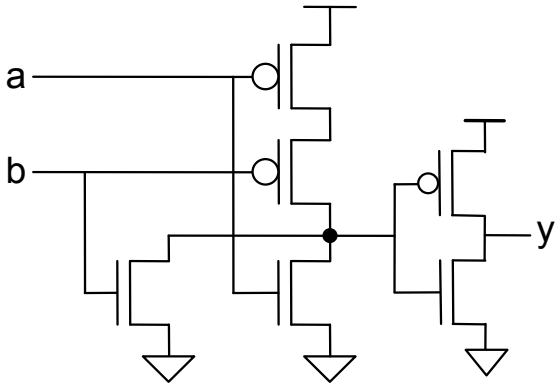
OR and NOR

OR

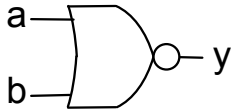


y=a+b

a b	y
0 0	0
0 1	1
1 0	1
1 1	1

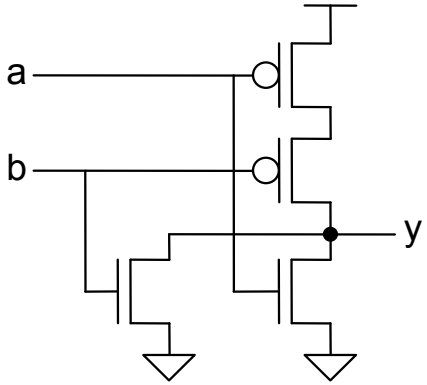


NOR



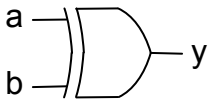
y=(a+b)′

a b	y
0 0	1
0 1	0
1 0	0
1 1	0



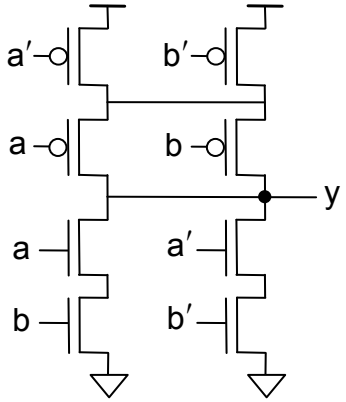
XOR and XNOR

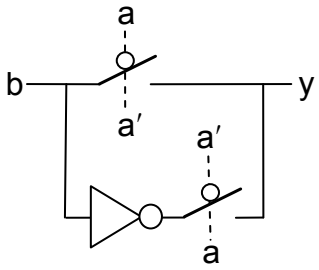
XOR



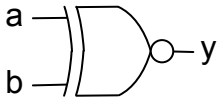
$y = a \oplus b$

a b	y
0 0	0
0 1	1
1 0	1
1 1	0



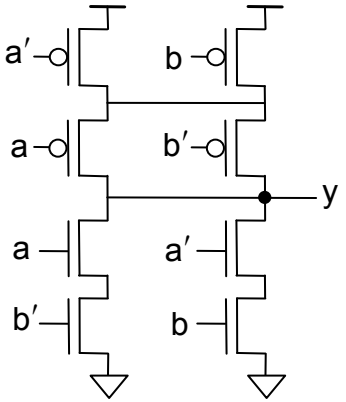


XNOR

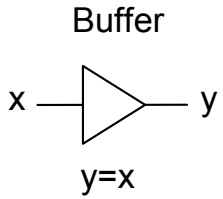


$y = (a \oplus b)'$

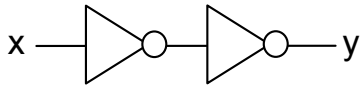
a b	y
0 0	1
0 1	0
1 0	0
1 1	1

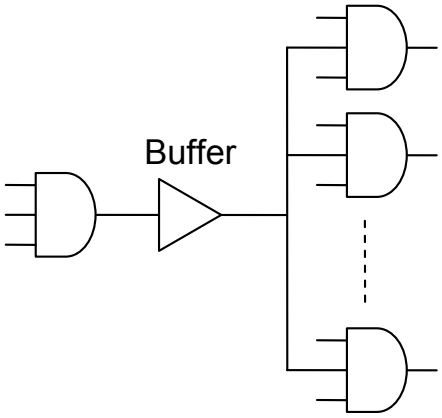


Regular buffer

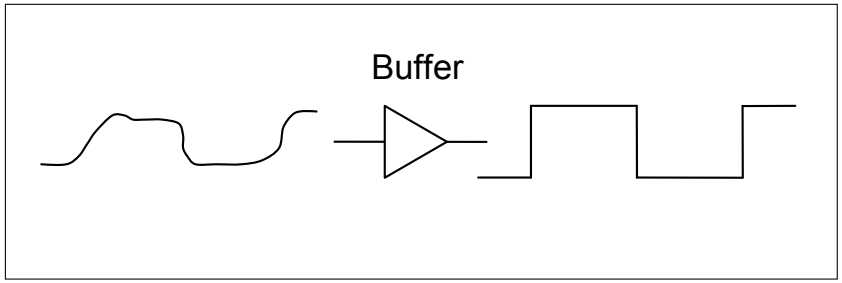


x	y
0	0
1	1





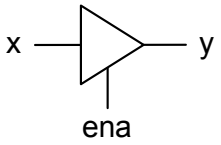
Example 1 (Purpose?)



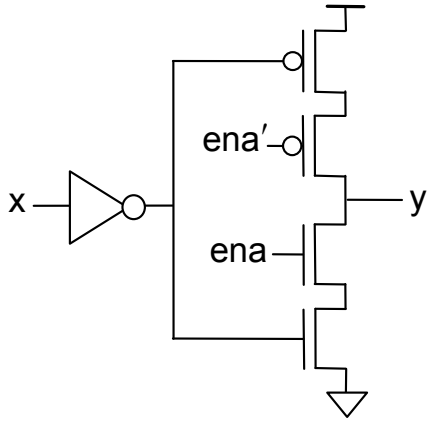
Example 2 (Purpose?)

Tri-state buffer

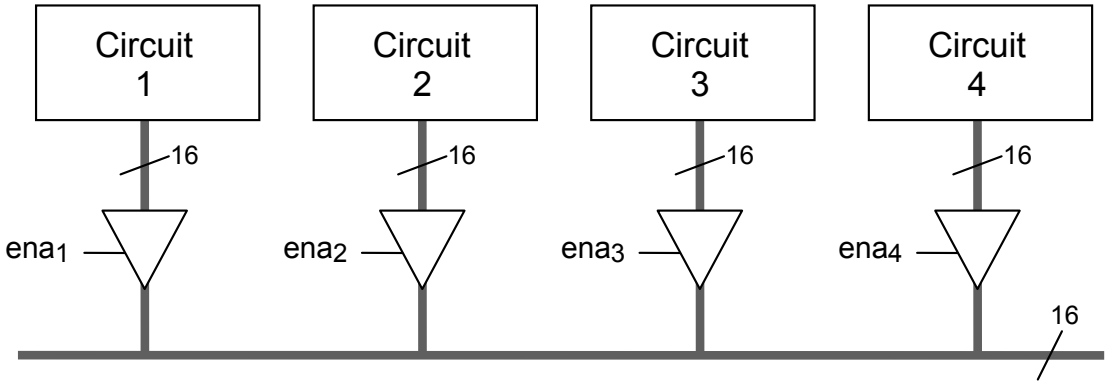
Tri-state buffer



ena	y
0	Z
1	x



$y = ena'.Z + ena.x$



Example (purpose?)

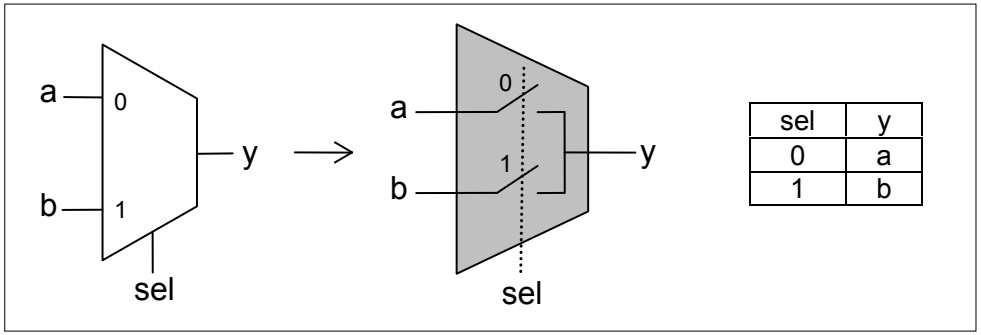
2. Combinational *Logic* Circuits

- Multiplexer
- Encoders / Decoders
- Parity detector
- Priority encoder

Multiplexer

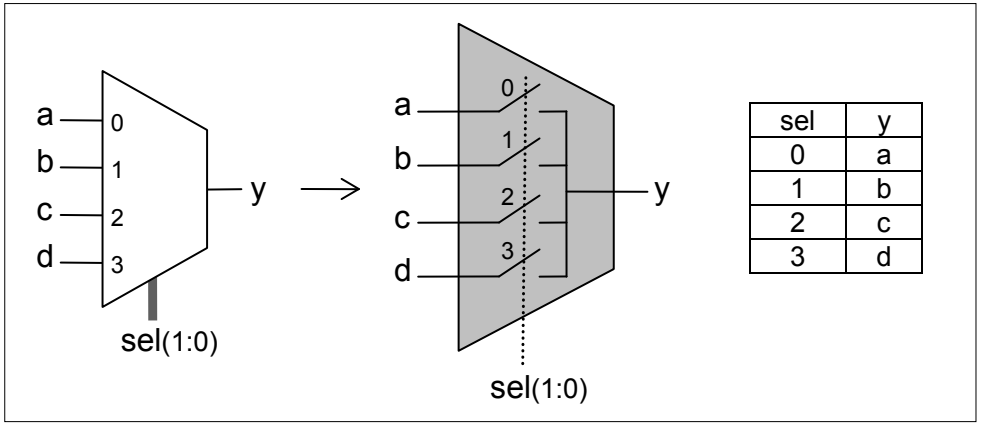
Concept

2x1 mux



sel	y
0	a
1	b

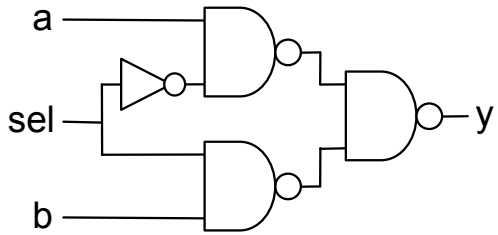
4x1 mux



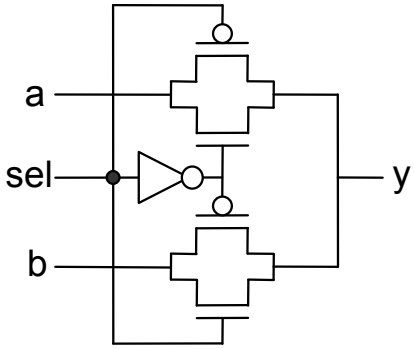
sel	y
0	a
1	b
2	c
3	d

Multiplexer

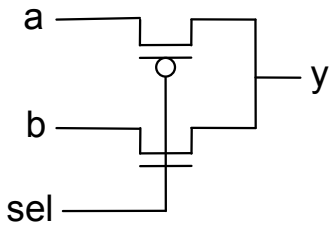
Implementation examples 2x1 mux



NAND-based



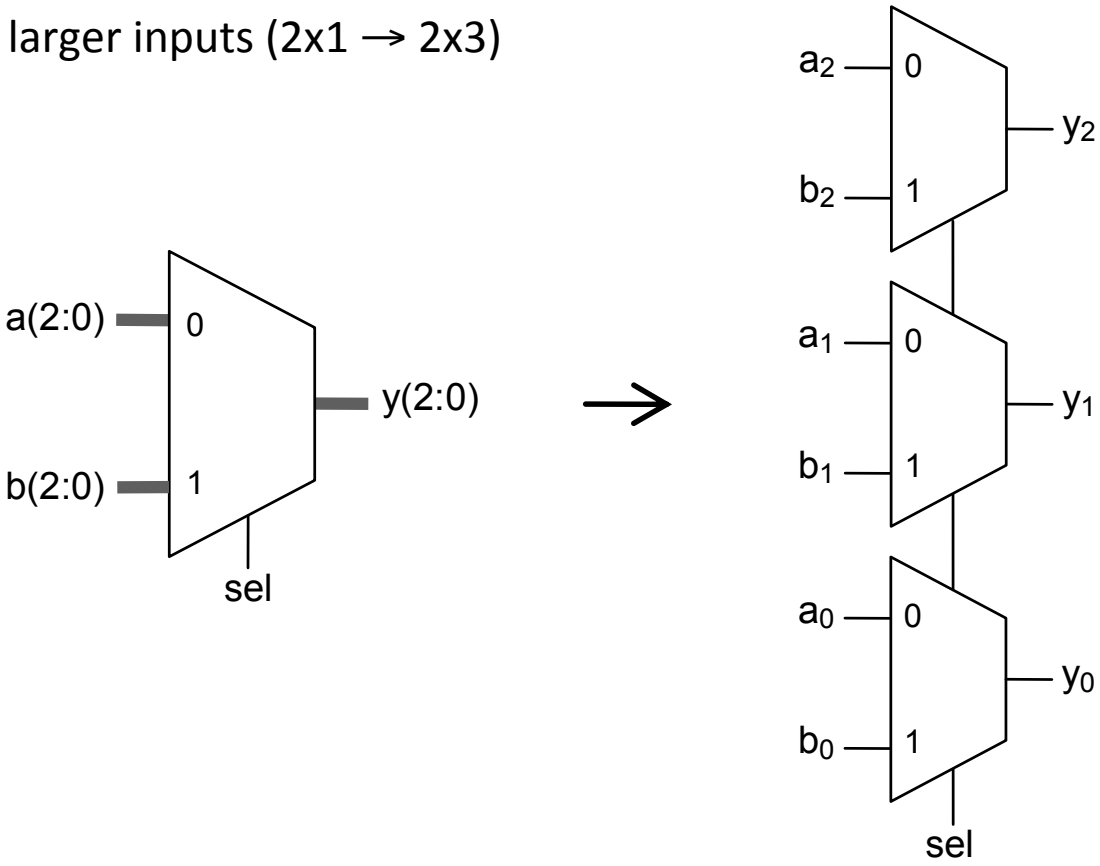
TG-based



PT-based

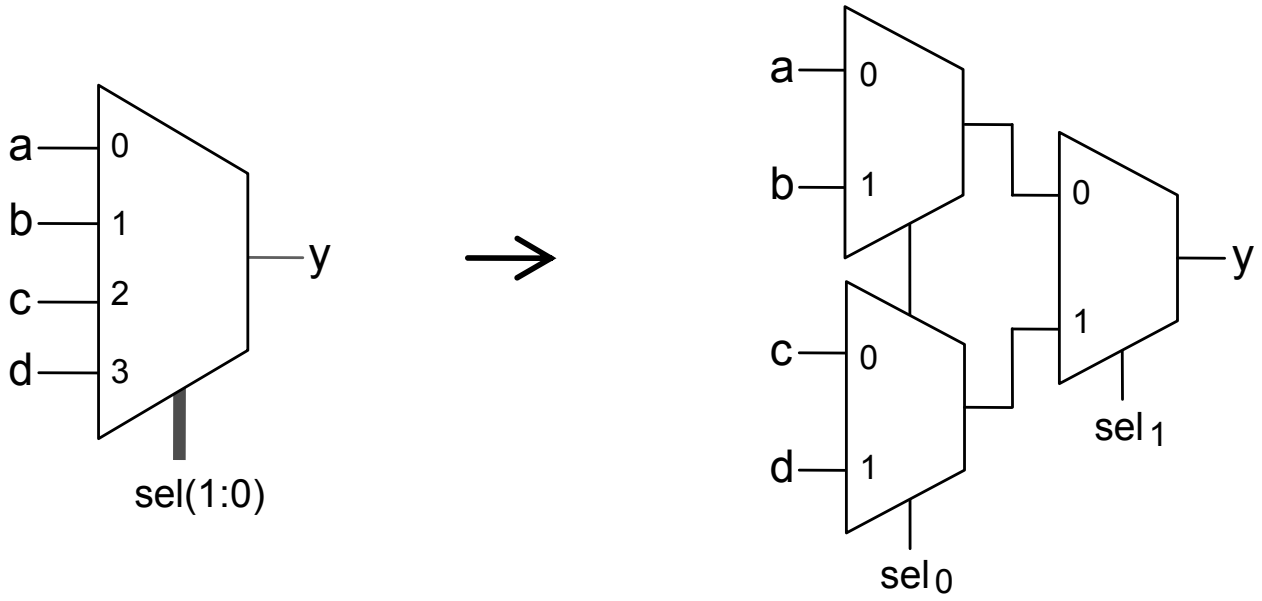
Multiplexer

Mux with larger inputs (2x1 → 2x3)



Multiplexer

Mux with more inputs (2x1 → 4x1)

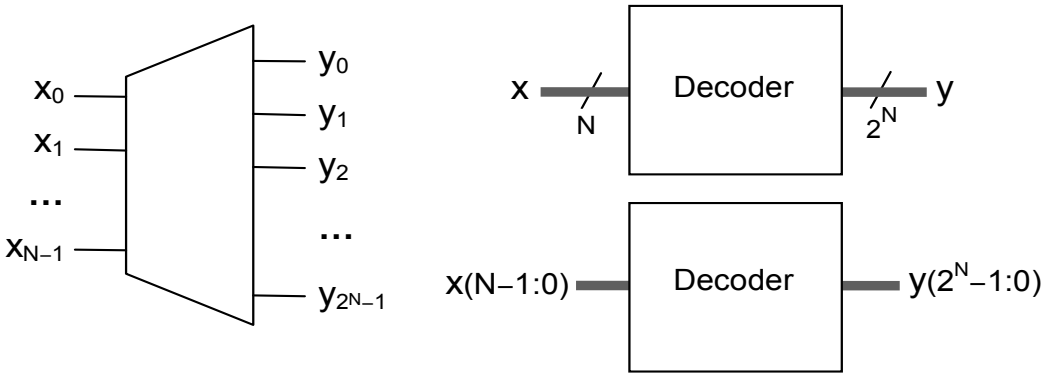


2. Combinational *Logic* Circuits

- Multiplexer
- **Encoders / Decoders**
- Parity detector
- Priority encoder

Encoders / Decoders

Address decoder



Symbols

x	y
000	00000001
001	00000010
010	00000100
011	00001000
100	00010000
101	00100000
110	01000000
111	10000000

Truth table (‘one-hot’ code)

Encoders / Decoders

Address decoder

Implementation examples (for N=2)

$y0 = x1'.x0'$ (SOP) or $y0 = (x1 + x0)'$ (POS)

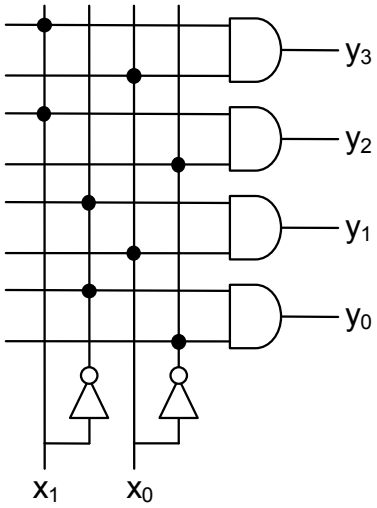
$y1 = x1'.x0$ (SOP) or $y1 = (x1 + x0')'$ (POS)

$y2 = x1.x0'$ (SOP) or $y1 = (x1' + x0)'$ (POS)

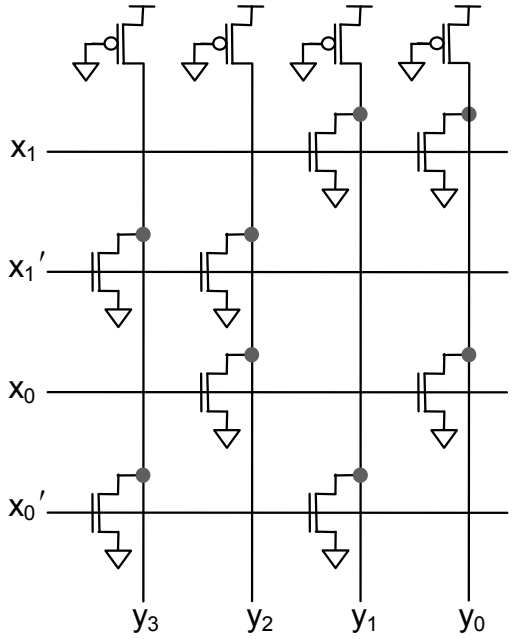
$y3 = x1.x0$ (SOP) or $y1 = (x1' + x0')'$ (POS)

x	y
00	0001
01	0010
10	0100
11	1000

Truth table (for N=2)



SOP-based implementation with AND gates

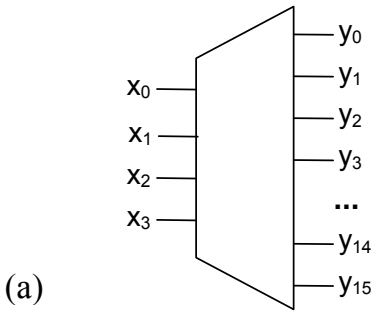


POS-based implementation (columns are NOR gates)

Encoders / Decoders

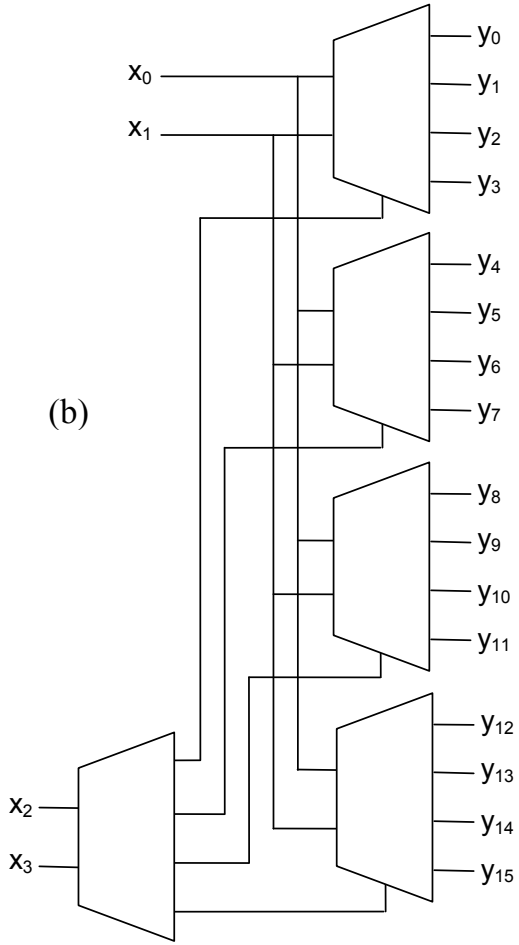
Address decoder

Larger address decoder (2-bit → 4-bit)



(a)

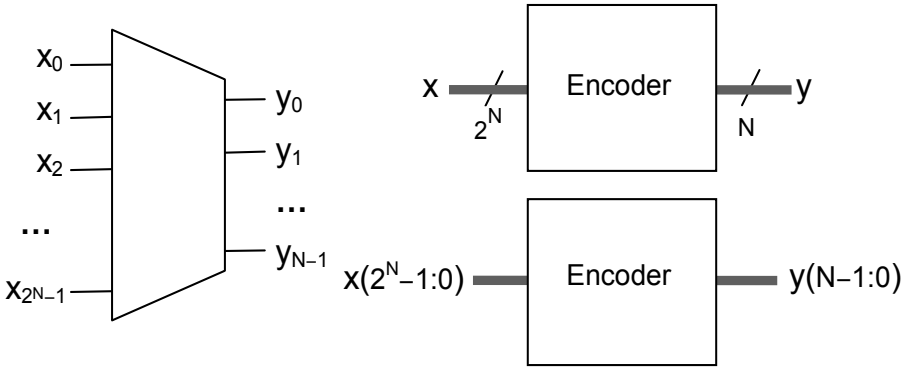
$x_3\ x_2\ x_1\ x_0$	$y_{15}\ y_{14}\ y_{13}\ \ldots\ y_2\ y_1\ y_0$
0 0 0 0	0 0 0 ... 0 0 0 0 1
0 0 0 1	0 0 0 ... 0 0 0 1 0
0 0 1 0	0 0 0 ... 0 0 1 0 0
0 0 11	0 0 0 ... 0 1 0 0 0
0 1 0 0	0 0 0 ... 1 0 0 0 0
...	...
1 1 1 0	0 1 0 ... 0 0 0 0 0
1 1 1 1	1 0 0 ... 0 0 0 0 0



(b)

Encoders / Decoders

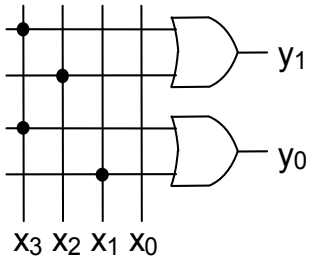
Address encoder



Symbols

x	y
0001	00
0010	01
0100	10
1000	11

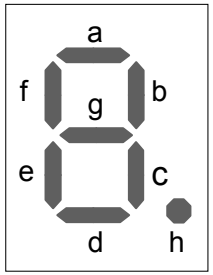
Truth table



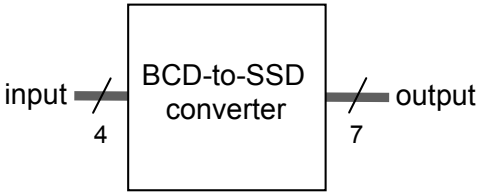
Implementation with OR gates

Encoders / Decoders

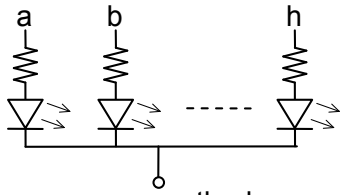
SSD (seven-segment display) decoder (or encoder)



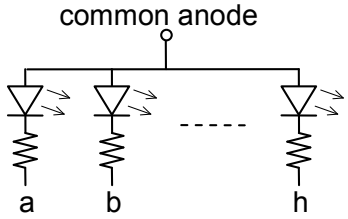
(a)



(d)



(b)



(c)

(e)

input		output (7 bits)		output (8 bits)	
ABCD	decimal	abcdefg	decimal	abcdefgh	decimal
0000	0	1111110	126	11111100	252
0001	1	0110000	48	01100000	96
0010	2	1101101	109	11011010	218
0011	3	1111001	121	11110010	242
0100	4	0110011	51	01100110	102
0101	5	1011011	91	10110110	182
0110	6	1011111	95	10111110	190
0111	7	1110000	112	11100000	224
1000	8	1111111	127	11111110	254
1001	9	1111011	123	11110110	246
others	10-15	don't care		don't care	

(a)

CD \ AB	00	01	11	10
00	1	0	X	1
01	0	1	X	1
11	1	1	X	X
10	1	1	X	X

a = A + C + B.D + B′.D′

(b)

CD \ AB	00	01	11	10
00	1	1	X	1
01	1	0	X	1
11	1	1	X	X
10	1	0	X	X

b = B′ + C.D + C′.D′

(c)

CD \ AB	00	01	11	10
00	1	1	X	1
01	1	1	X	1
11	1	1	X	X
10	0	1	X	X

c = (A′.B′.C.D′)′

(d)

C D \ A B	00	01	11	10
00	1	0	X	1
01	0	1	X	1
11	1	0	X	X
10	1	1	X	X

d = A + B′.C + B′.D′ + C.D′ + B.C′.D

(e)

CD \ AB	00	01	11	10
00	1	0	X	1
01	0	0	X	0
11	0	0	X	X
10	1	1	X	X

e = B′.D′ + C.D′

(f)

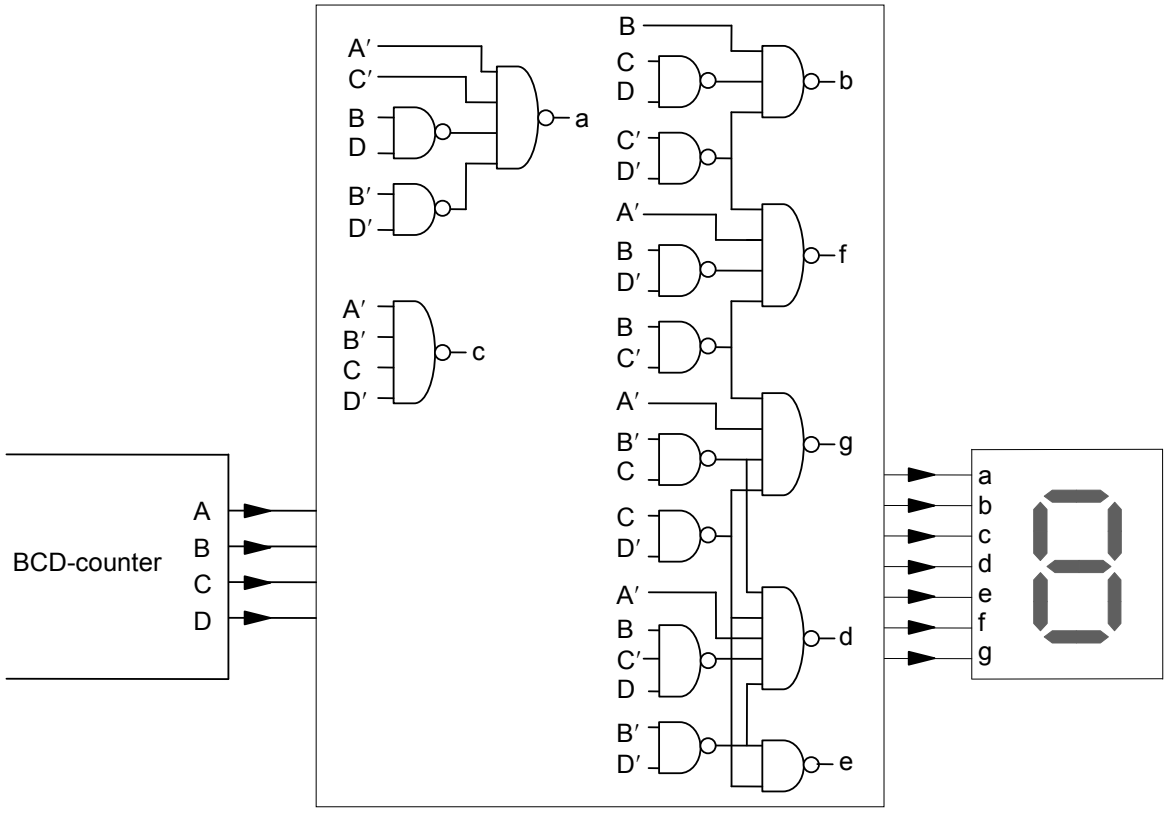
C D \ A B	00	01	11	10
00	1	1	X	1
01	0	1	X	1
11	0	0	X	X
10	0	1	X	X

f = A + B.C′ + B.D′ + C′.D′

(g)

C D \ A B	00	01	11	10
00	0	1	X	1
01	0	1	X	1
11	1	0	X	X
10	1	1	X	X

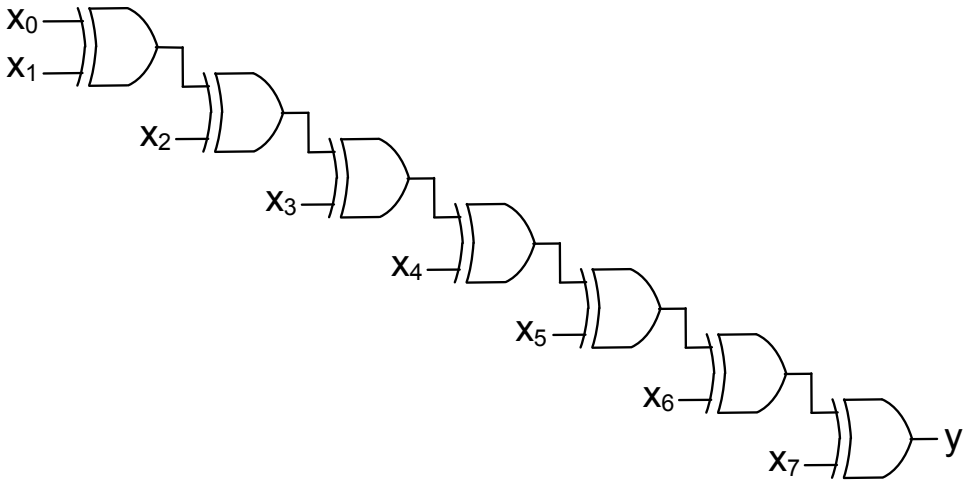
g = A + B.C′ + B′.C + C.D′

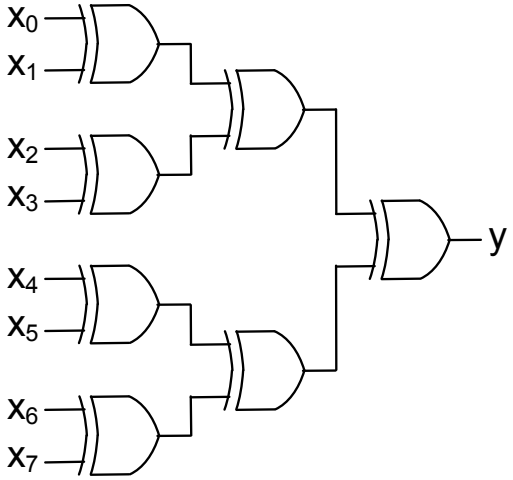


2. Combinational *Logic* Circuits

- Multiplexer
- Encoders / Decoders
- **Parity detector**
- Priority encoder

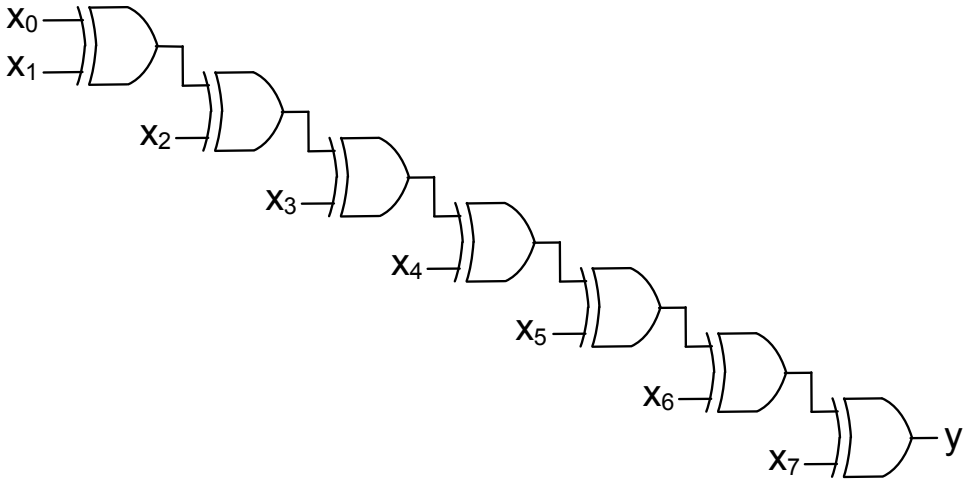
Parity detector

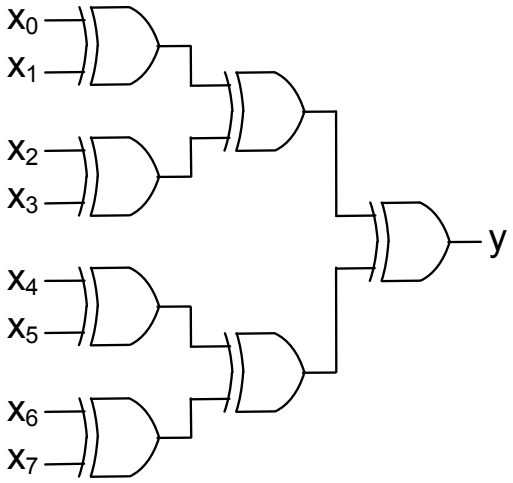




- Odd or even parity?
- Linear or logarithmic size?
- Linear or logarithmic propagation delay?

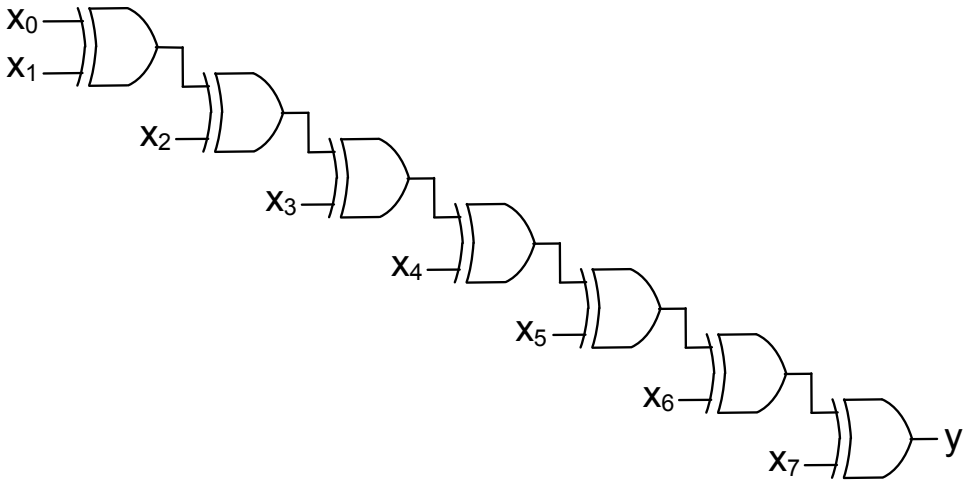
Parity detector

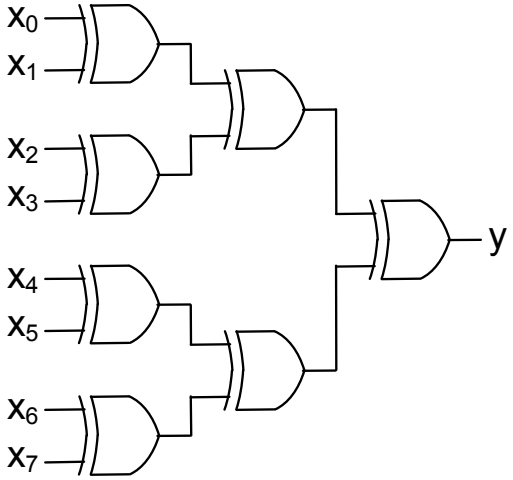




- Odd or even parity? Odd
- Linear or logarithmic size?
- Linear or logarithmic propagation delay?

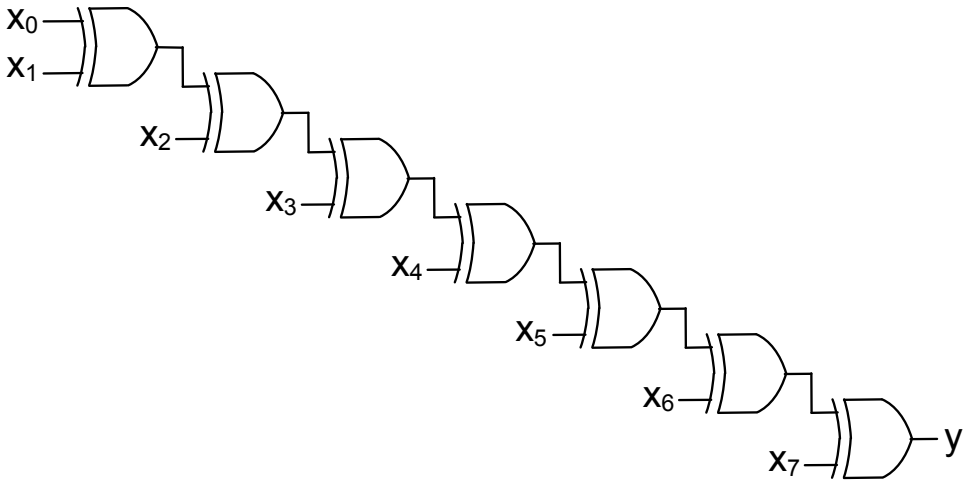
Parity detector

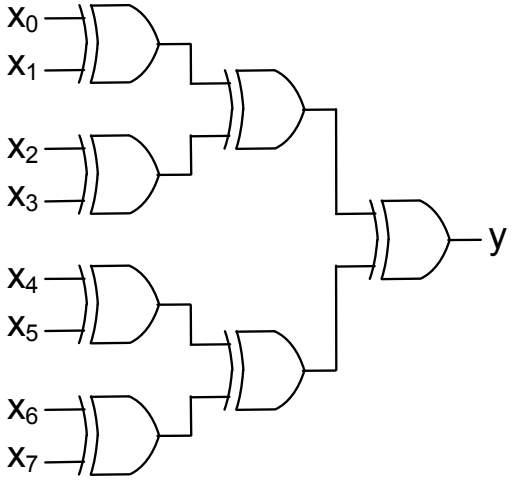




- Odd or even parity? Odd
- Linear or logarithmic size? Linear (N-1 gates)
- Linear or logarithmic propagation delay?

Parity detector





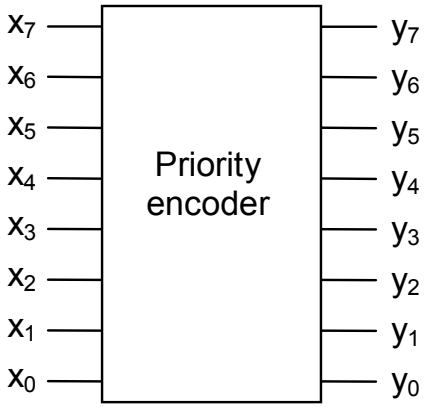
- Odd or even parity? Odd
- Linear or logarithmic size? Linear (N-1 gates)
- Linear or logarithmic propagation delay? Log (log2(N) logic layers)

2. Combinational *Logic* Circuits

- Multiplexer
- Encoders / Decoders
- Parity detector
- **Priority encoder**

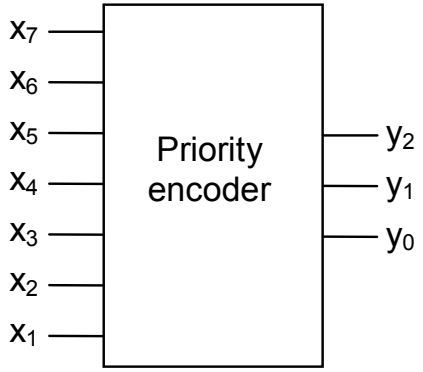
Priority encoders

Output = one-hot address of highest-priority input



$x_7 \ldots x_0$	$y_7 \ldots y_0$
1xxxxxxx	10000000
01xxxxxx	01000000
001xxxxx	00100000
0001xxxx	00010000
00001xxx	00001000
000001xx	00000100
0000001x	00000010
00000001	00000001
00000000	00000000

Output = conventional binary address of highest-priority input



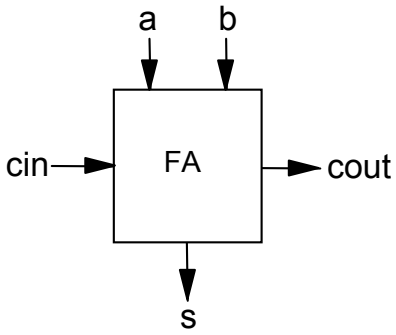
$x_7 \ldots x_1$	$y_2\ y_1\ y_0$
1xxxxxx	111
01xxxxx	110
001xxxx	101
0001xxx	100
00001xx	011
000001x	010
0000001	001
0000000	000

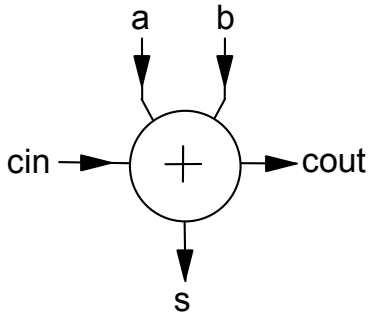
3. Combinational *Arithmetic* Circuits

- Basic adders
- Fast adders
- Signed adders/subtracters
- Comparators
- ALU (arithmetic-logic unit)
- Multipliers
- Dividers

Basic adders

Full-adder (FA) unit





Symbols

cin	a b	s	cout
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

Truth table

$s = a \oplus b \oplus cin$ → Odd-parity function

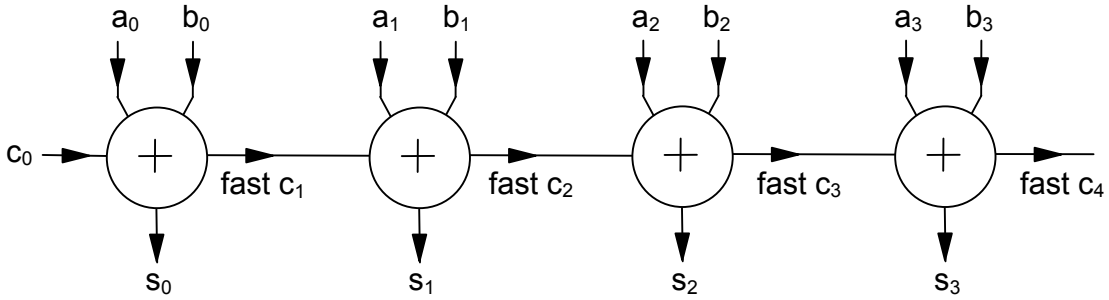
$cout = a.b + a.cin + b.cin$ → Majority function

3. Combinational *Arithmetic* Circuits

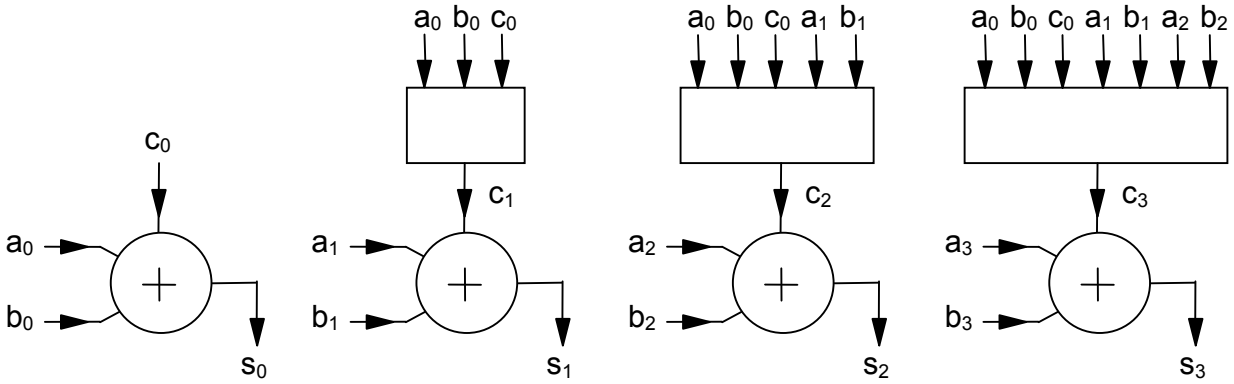
- Basic adders
- **Fast adders**
- Signed adders/subtracters
- Comparators
- ALU (arithmetic-logic unit)
- Multipliers
- Dividers

Fast Adders

General approaches to the design of fast adders



(a) Faster carry propagation

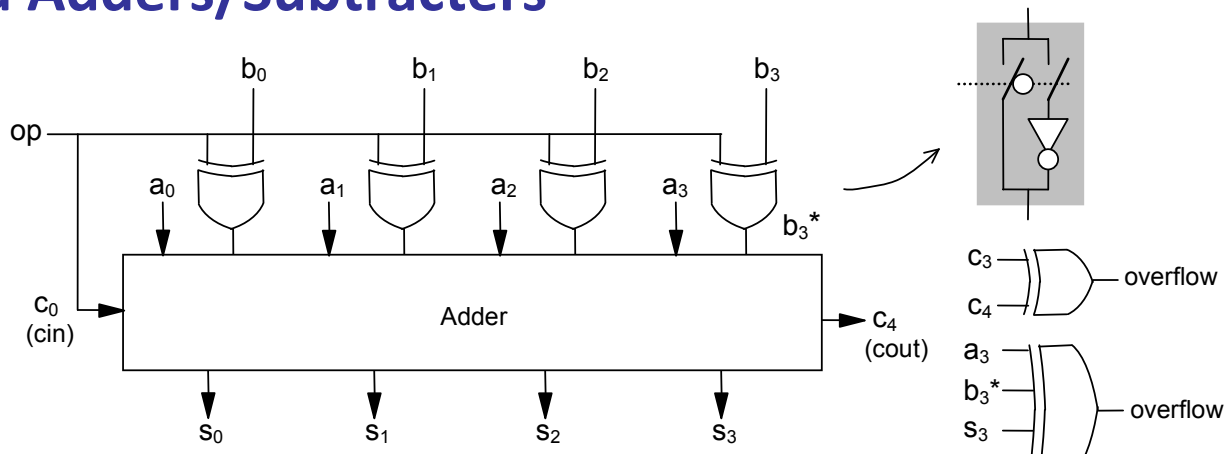


(b) Faster carry generation

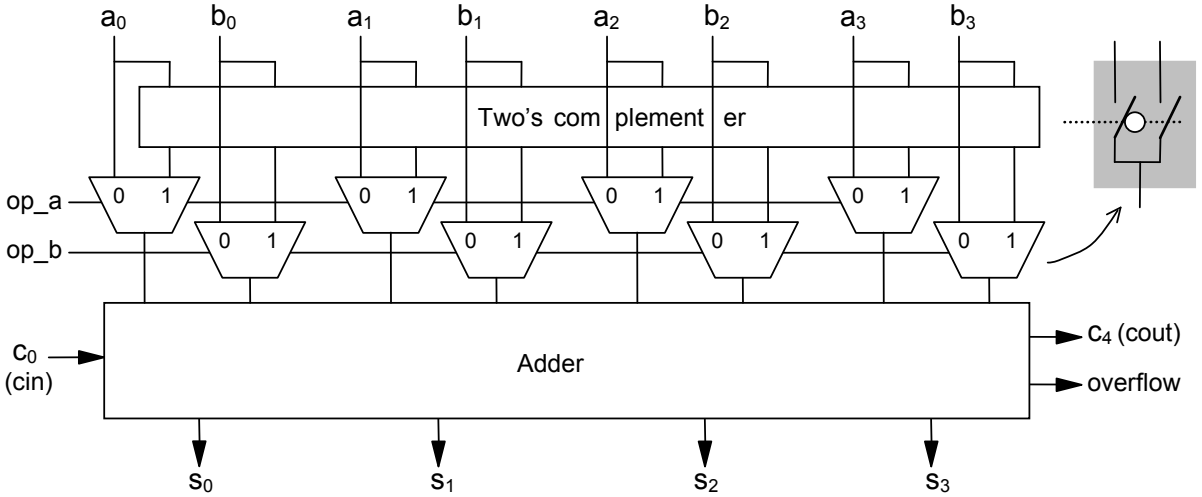
3. Combinational *Arithmetic* Circuits

- Basic adders
- Fast adders
- **Signed adders/subtracters**
- Comparators
- ALU (arithmetic-logic unit)
- Multipliers
- Dividers

Signed Adders/Subtracters



(a) a ± b

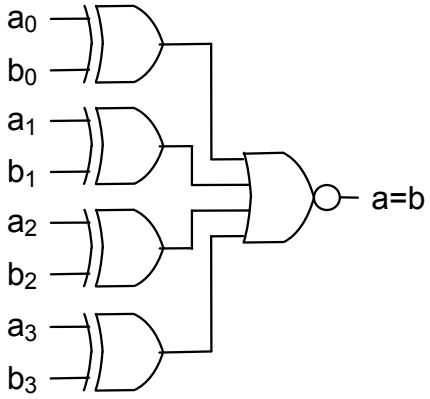


(b) ±a ± b

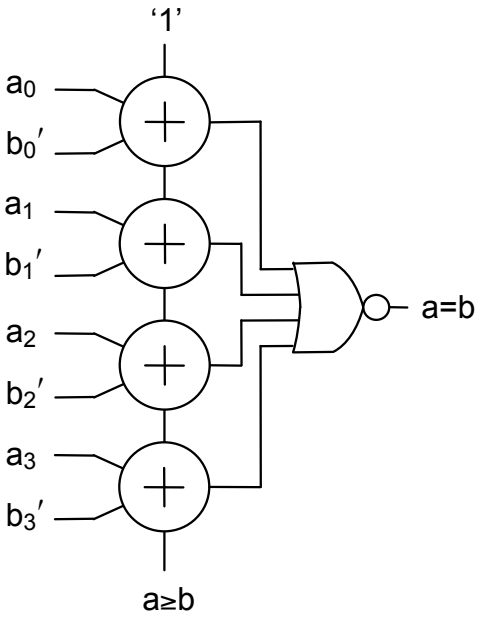
3. Combinational *Arithmetic* Circuits

- Basic adders
- Fast adders
- Signed adders/subtracters
- **Comparators**
- ALU (arithmetic-logic unit)
- Multipliers
- Dividers

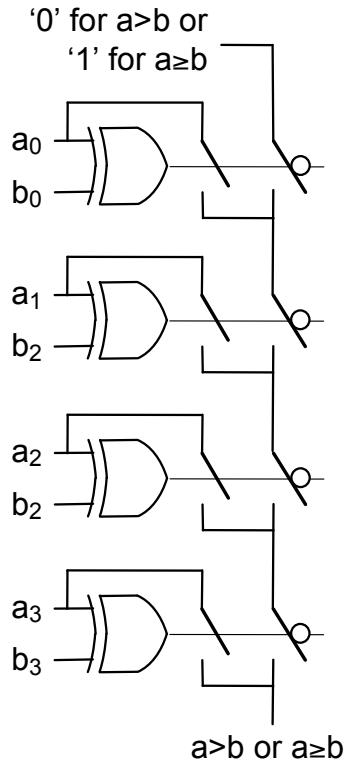
Comparators



XOR-based
equality comparator



Adder-based
equality and magnitude
comparator

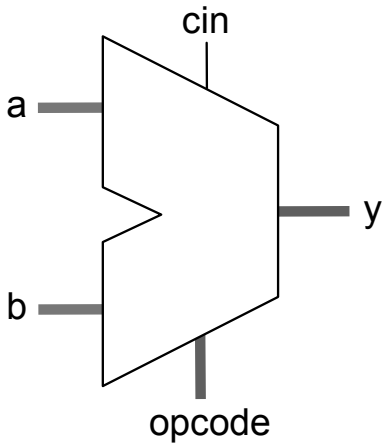


Mux-based
equality and magnitude
comparator

3. Combinational *Arithmetic* Circuits

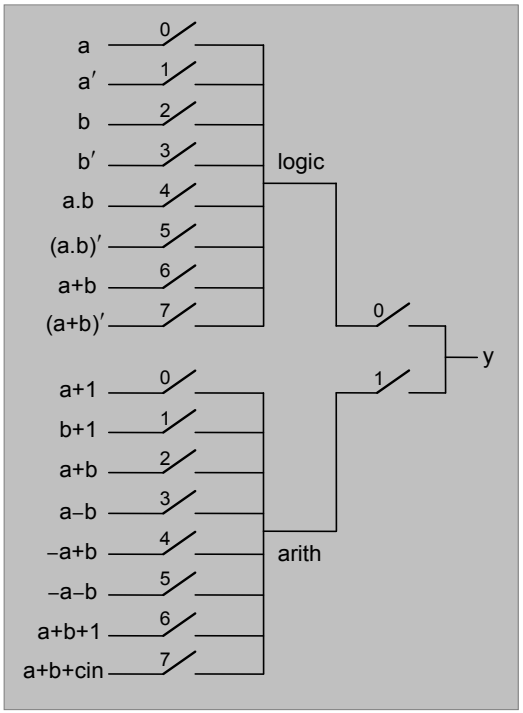
- Basic adders
- Fast adders
- Signed adders/subtracters
- Comparators
- **ALU (arithmetic-logic unit)**
- Multipliers
- Dividers

ALU



Unit	Instruction	Operation	opcode
Logic	Transfer a	y = a	0000
	Complement a	y = a′	0001
	Transfer b	y = b	0010
	Complement b	y = b′	0011
	AND	y = a.b	0100
	NAND	y = (a.b)′	0101
	OR	y = a+b	0110
	NOR	y = (a+b)′	0111
Arithmetic	Increment a	y = a+1	1000
	Increment b	y = b+1	1001
	Add a and b	y = a+b	1010
	Sub b from a	y = a–b	1011
	Sub a from b	y = –a+b	1100
	Add negative	y = –a–b	1101
	Add with 1	y = a+b+1	1110
	Add with carry	y = a+b+cin	1111

ALU

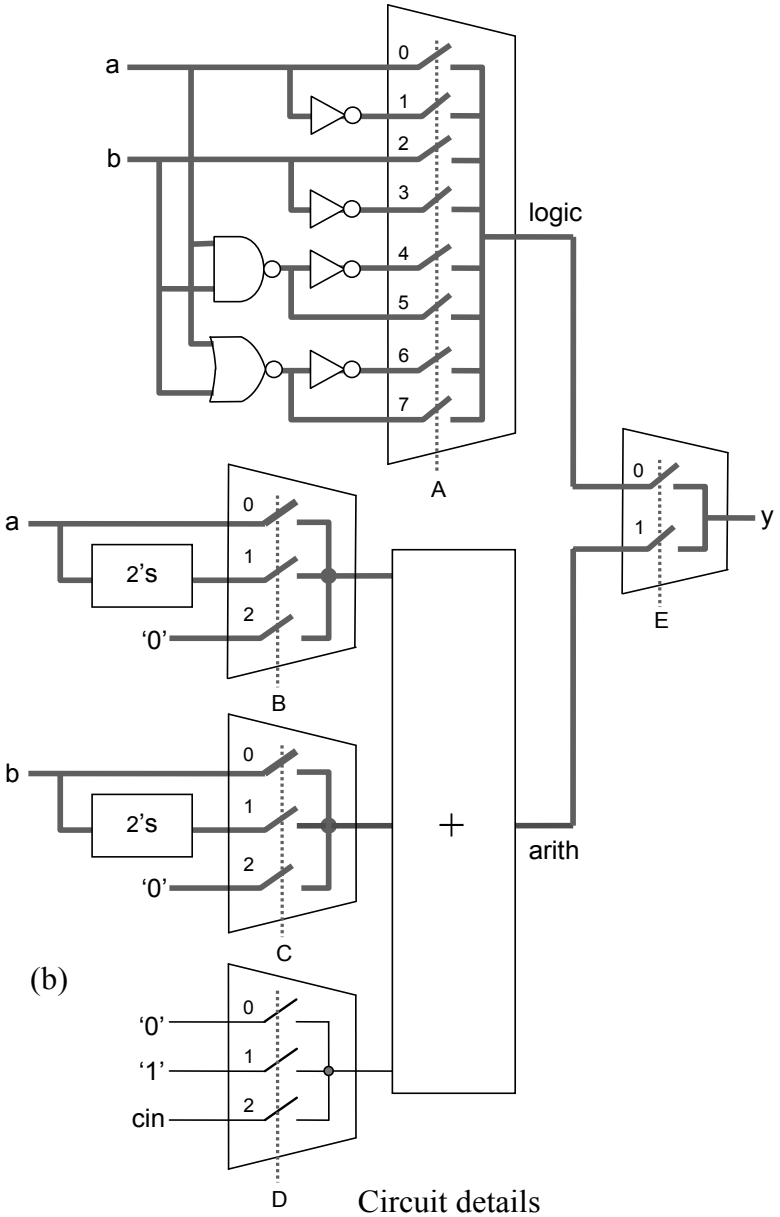


Conceptual circuit

Decoder for logic section	
opcode	closed switches
0000	A0 E0
0001	A1 E0
0010	A2 E0
0011	A3 E0
0100	A4 E0
0101	A5 E0
0110	A6 E0
0111	A7 E0

Decoder for arithmetic section	
opcode	closed switches
1000	B0 C2 D1 E1
1001	B2 C0 D1 E1
1010	B0 C0 D0 E1
1011	B0 C1 D0 E1
1100	B1 C0 D0 E1
1101	B1 C1 D0 E1
1110	B0 C0 D1 E1
1111	B0 C0 D2 E1

Instructions decoder



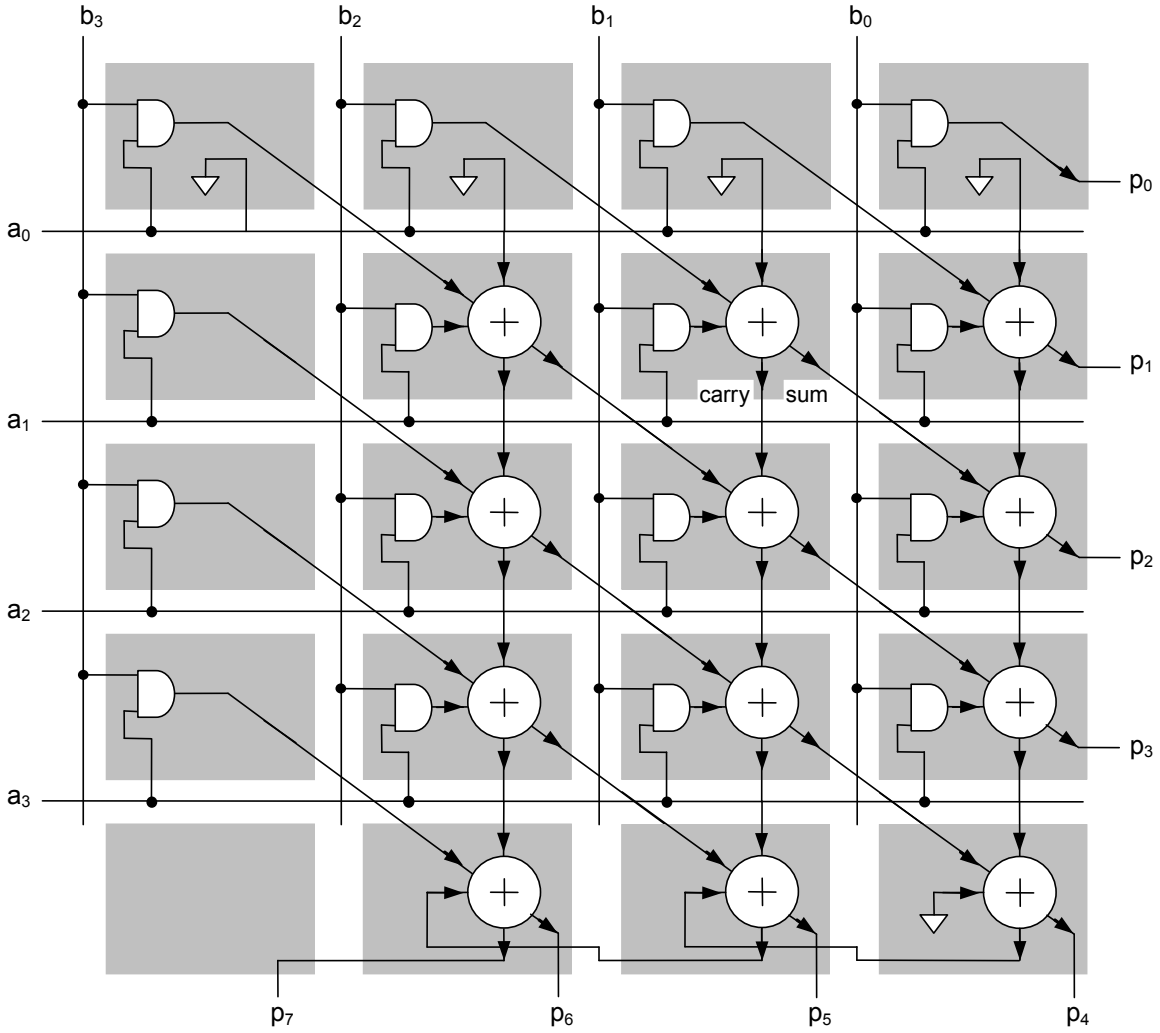
Circuit details

3. Combinational *Arithmetic* Circuits

- Basic adders
- Fast adders
- Signed adders/subtracters
- Comparators
- ALU (arithmetic-logic unit)
- **Multipliers**
- Dividers

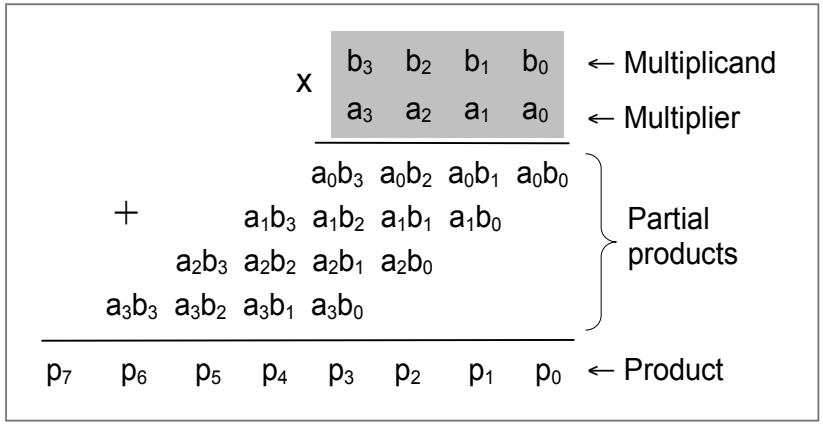
Multipliers

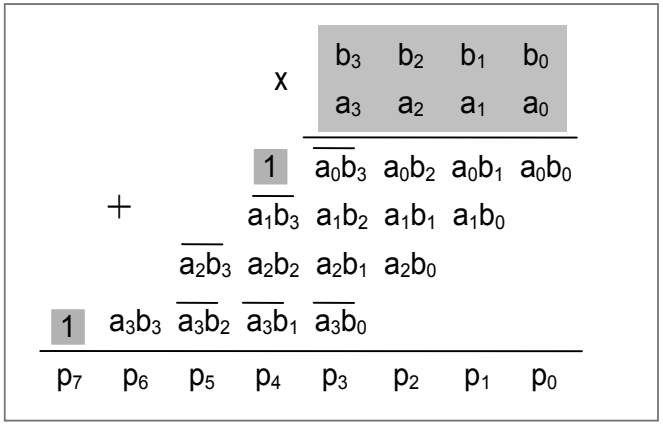
Parallel unsigned multiplier (*array multiplier*)



Multipliers

Parallel signed multiplier (*modified Baugh-Wooley multiplier*)

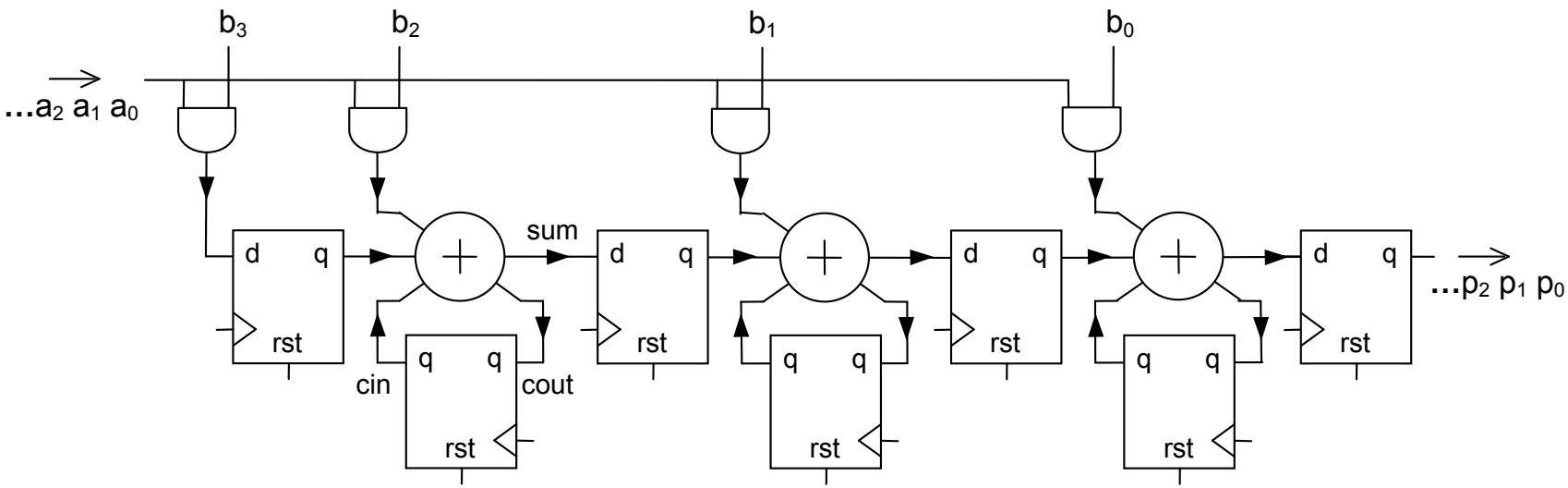




ALU-based –> not combinational

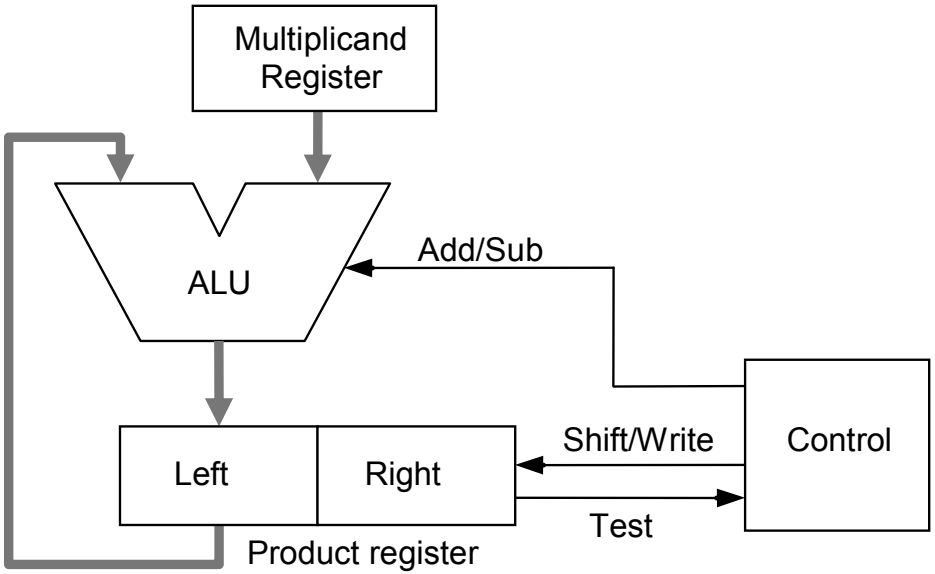
Multipliers

Parallel-serial multiplier (not combinational)



Multipliers

ALU-based multiplier (not combinational)



Multipliers

ALU-based multiplier (not combinational – employs Booth's algorithm)

Iteration	Procedure	Multiplicand	Product Left Right Extra
0	Initialization (Multiplier loaded into ProdRight)	10010	00000 0111 0 0
1	Bits=00 → No operation Shift right arith		00000 0111 0 0 00000 0011 1 0
2	Bits=10 → ProdLeft – Multiplicand Shift right arith		01110 0011 1 0 00111 0001 1 1
3	Bits=11 → No operation Shift right arith		00111 0001 1 1 00011 1000 1 1
4	Bits=11 → No operation Shift right arith		00011 1000 1 1 00001 1100 0 1
5	Bits=01 → ProdLeft + Multiplicand Shift right arith		10011 11000 11001 11100

3. Combinational *Arithmetic* Circuits

- Basic adders
- Fast adders
- Signed adders/subtracters
- Comparators
- ALU (arithmetic-logic unit)
- Multipliers
- **Dividers**

Dividers

ALU-based divider (not combinational)

Iteration	Procedure	Divisor	Remainder Left Right
0	Initialization (Dividend is loaded into RemRight) Shift Rem left with '0' in empty position	0101	0000 1101 0001 1010
1	RemLeft – Divisor Bit=1 → RemLeft + Divisor Bit=1 → Shift Rem left with '0'		1 100 1010 0001 1010 0011 0100
2	RemLeft – Divisor Bit=1 → RemLeft + Divisor Bit=1 → Shift Rem left with '0'		1 110 0100 0011 0100 0110 1000
3	RemLeft – Divisor Bit=0 → No operation Bit=0 → Shift Rem left with '1'		0 001 1000 0001 1000 0011 0001
4	RemLeft – Divisor Bit=1 → RemLeft + Divisor Bit=1 → Shift Rem left with '0'		1 110 0001 0011 0001 0110 0010
(*)	Shift RemLeft to the right with '0'		0011 0010
(*) After the last iteration the left half of the remainder must be shifted to the right.			