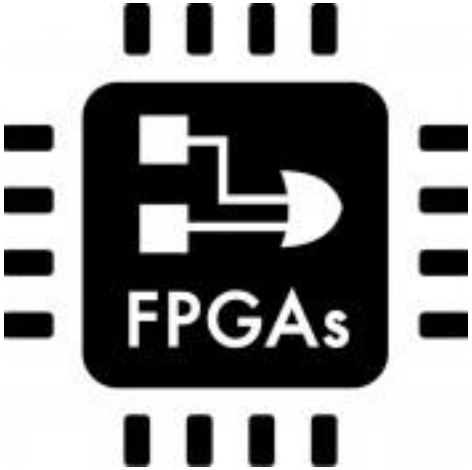


In-House Chip Design

Pravin Zode

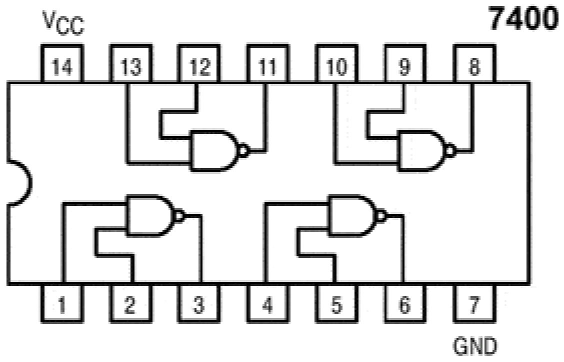
Anything's easy if you know how to do it.

Bill Vaughn

Outline

- Introduction to Design
- Older Ways of design
- Modern Digital Design
- Programmable Logic Devices (CPLD and FPGA)
- Demonstration of In-House Chip design

Older ways of Digital Design



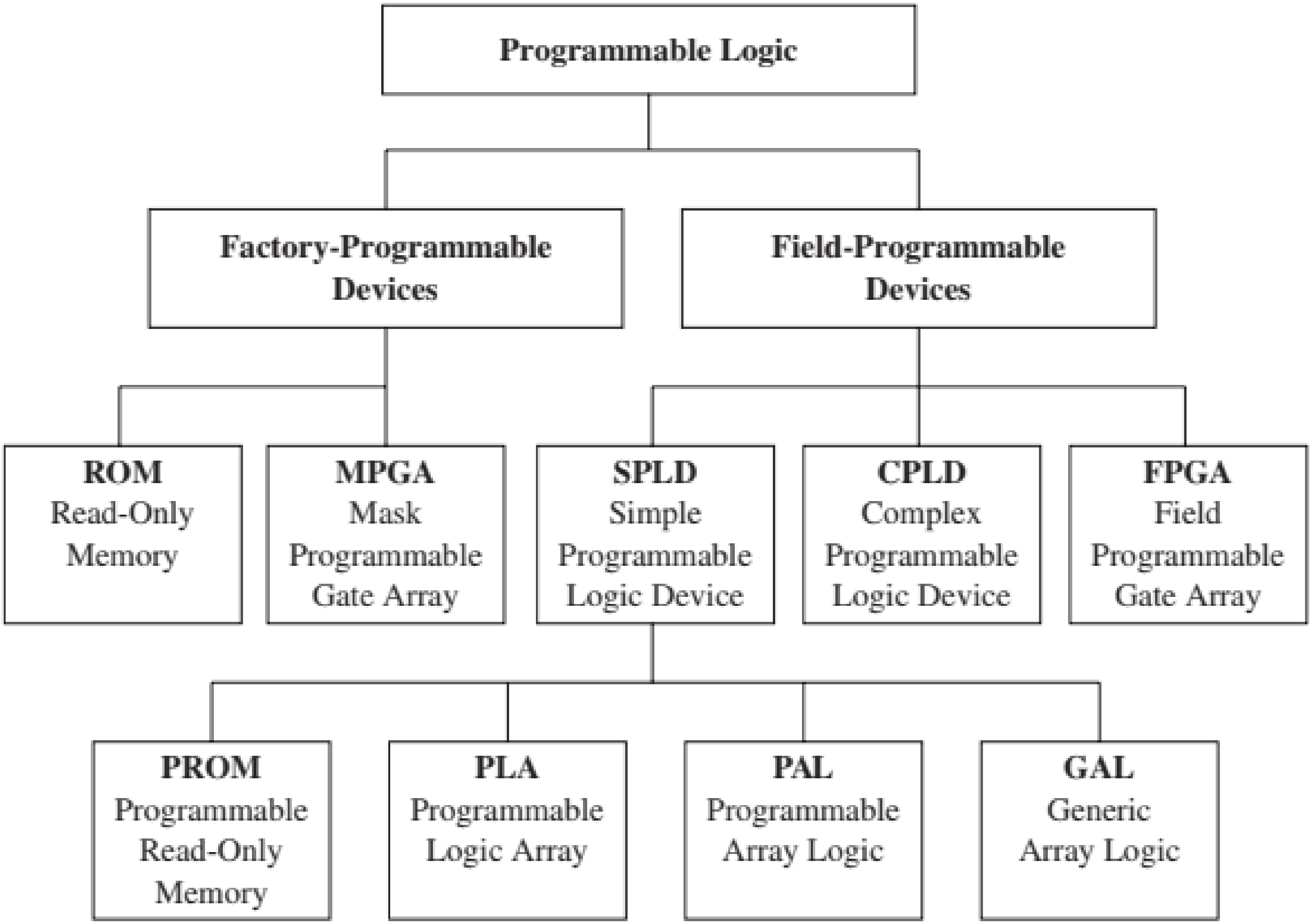
- Discrete Logic : Can build small circuits only
- De-Morgan’s Theorem : Theoretically we need only two input NAND and NOR gates
- Tedious , Expensive , slow, prone to wiring errors

Motivation

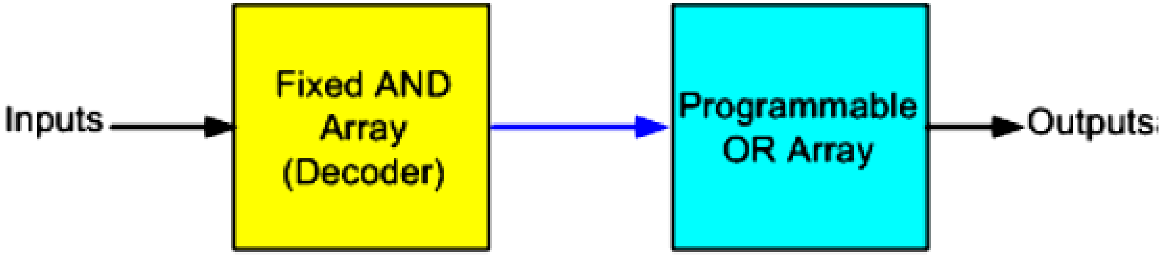
Why Programmable Logic Devices (PLDs) are Important in VLSI

- **Flexibility:** One hardware platform, multiple designs.
- **Faster Prototyping:** No need to fabricate silicon for each design iteration.
- **Lower Development Cost:** Avoids expensive mask sets required for ASIC.
- **Shorter Time-to-Market:** Quickly implement and test designs in FPGA/CPLD.
- **Easy Reconfiguration:** Designs can be upgraded in-field (firmware update instead of hardware change).
- **Supports Hardware-Software Co-Design:** Can integrate processors + custom logic

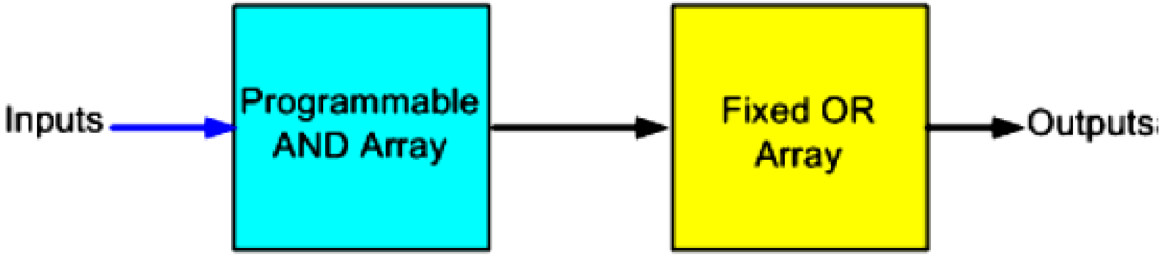
Programmable Logic Devices


Programmable Logic
Factory-Programmable Devices
Field-Programmable Devices
ROM Read-Only Memory
MPGA Mask Programmable Gate Array
SPLD Simple Programmable Logic Device
CPLD Complex Programmable Logic Device
FPGA Field Programmable Gate Array
PROM Programmable Read-Only Memory
PLA Programmable Logic Array
PAL Programmable Array Logic
GAL Generic Array Logic

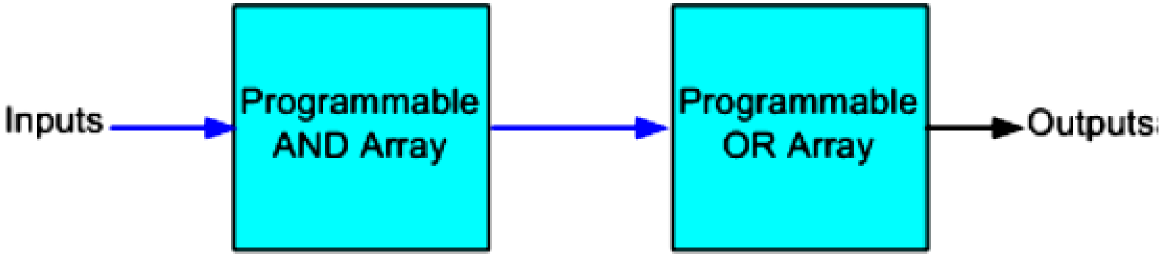
Fundamental PLD



Programmable Read Only Memory (PROM)



Programmable Array Logic (PAL) Device



Programmable Logic Array (PLA) Device

Fundamental PLD

SPLD (Simple PLD)

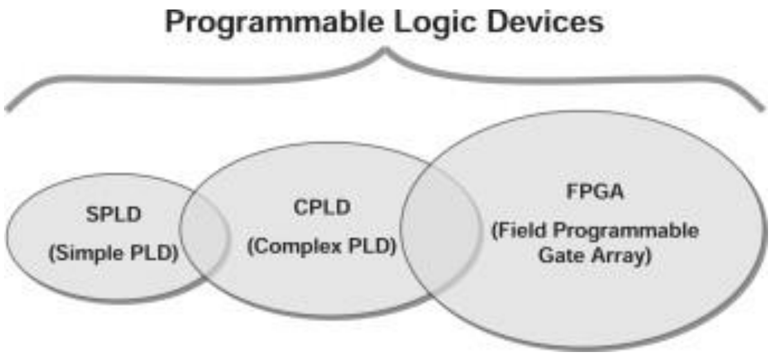
- Basic building block (PAL, PLA)
- Used for small, simple logic functions

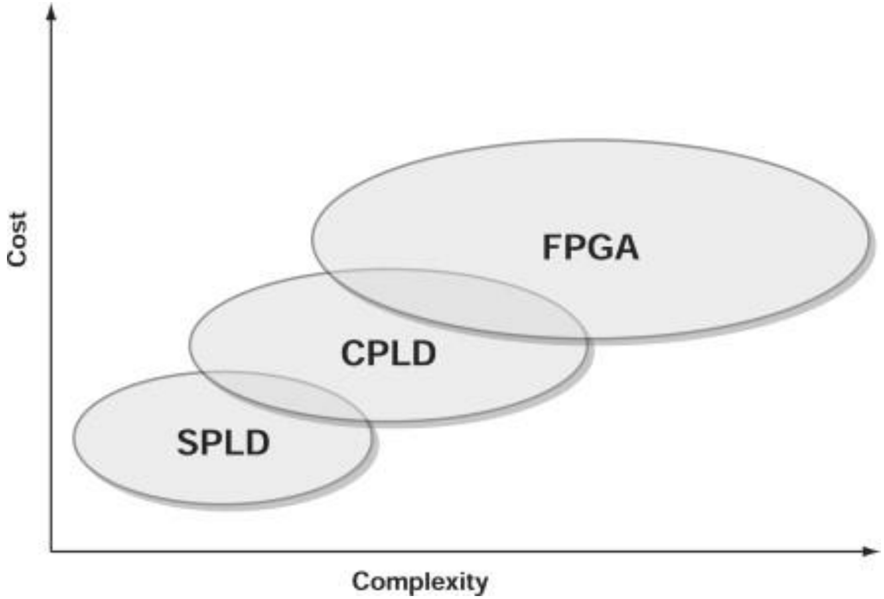
CPLD (Complex PLD)

- Multiple SPLD-like blocks with predictable timing
- Non-volatile, suitable for control logic

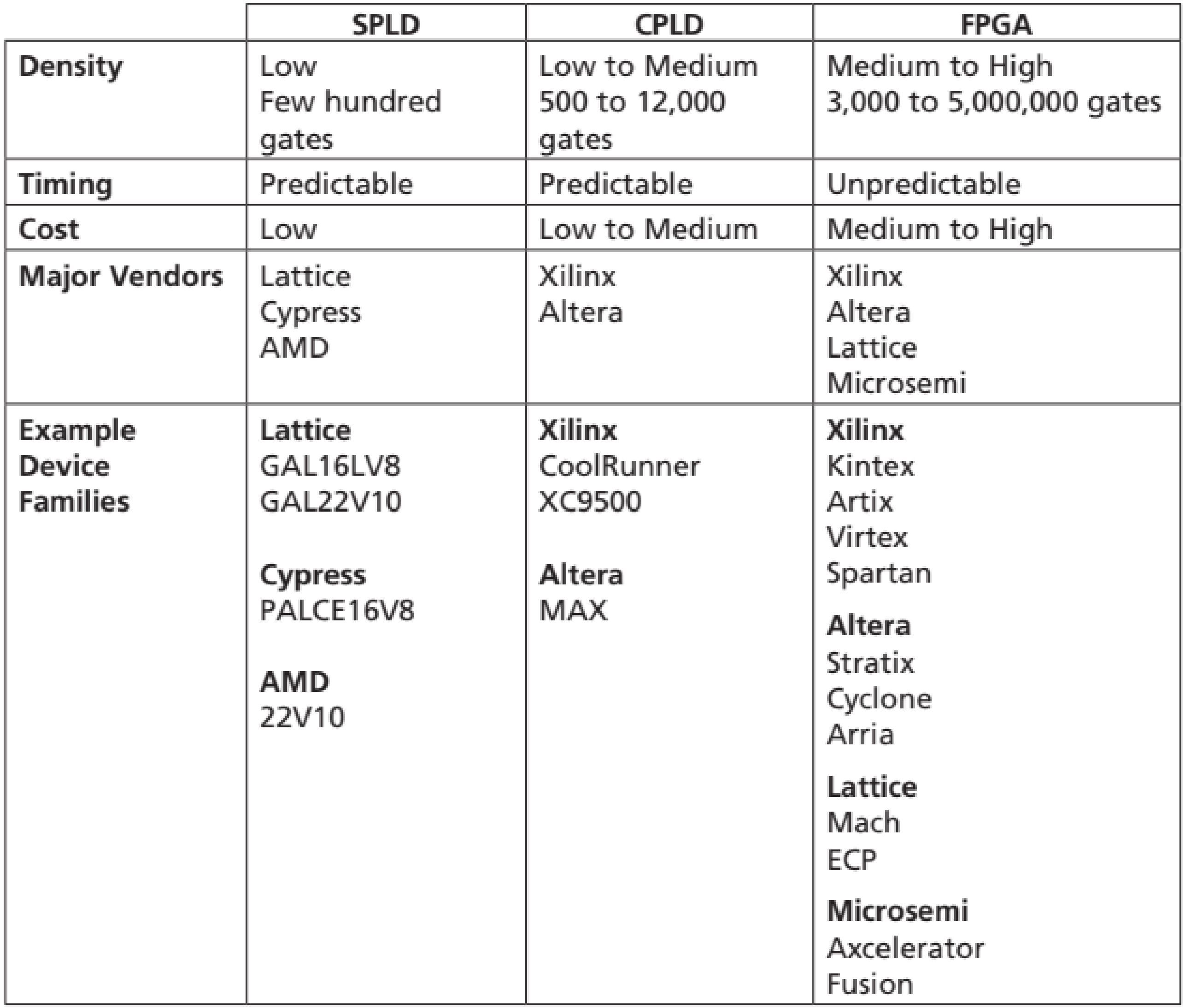
FPGA (Field Prog. Gate Array)

- Large array of configurable logic blocks (high density)
- SRAM-based, reconfigurable for complex designs



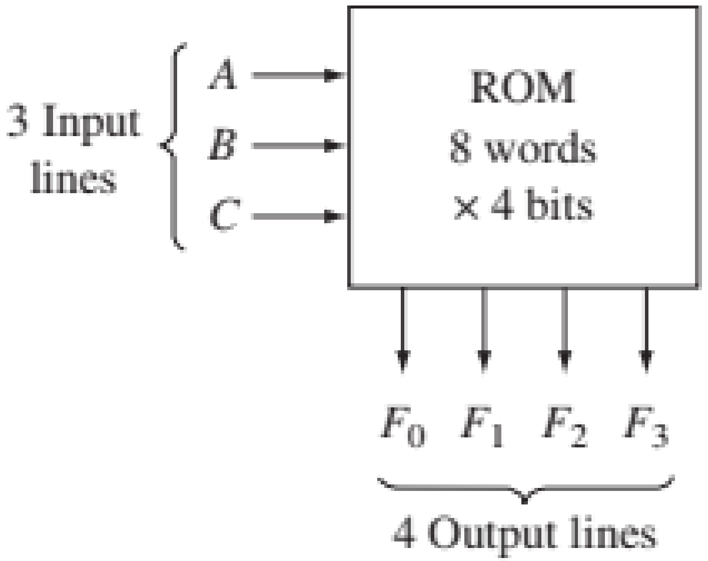


PLD Comparison



	SPLD	CPLD	FPGA
Density	Low Few hundred gates	Low to Medium 500 to 12,000 gates	Medium to High 3,000 to 5,000,000 gates
Timing	Predictable	Predictable	Unpredictable
Cost	Low	Low to Medium	Medium to High
Major Vendors	Lattice Cypress AMD	Xilinx Altera	Xilinx Altera Lattice Microsemi
Example Device Families	**Lattice** GAL16LV8 GAL22V10 **Cypress** PALCE16V8 **AMD** 22V10	**Xilinx** CoolRunner XC9500 **Altera** MAX	**Xilinx** Kintex Artix Virtex Spartan **Altera** Stratix Cyclone Arria **Lattice** Mach ECP **Microsemi** Axcelerator Fusion

ROM Based Design



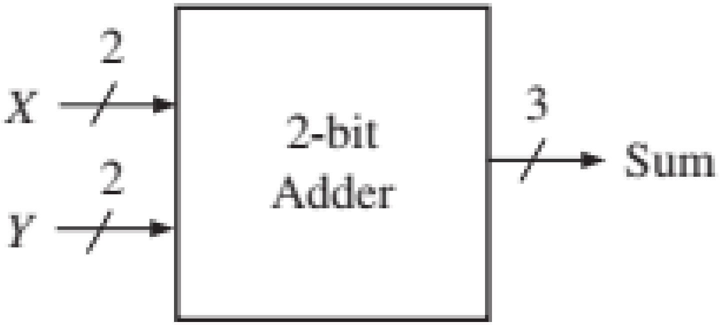
(a) Block diagram

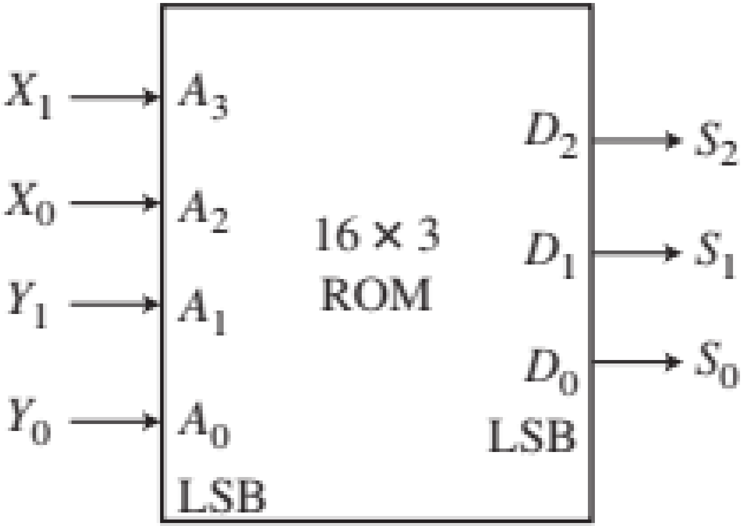
A	B	C	F_0	F_1	F_2	F_3
0	0	0	1	0	1	0
0	0	1	1	0	1	0
0	1	0	0	1	1	1
0	1	1	0	1	0	1
1	0	0	1	1	0	0
1	0	1	0	0	0	1
1	1	0	1	1	1	1
1	1	1	0	1	0	1

Typical data stored in ROM (2^3 words of 4 bits each)

(b) Truth table for ROM

ROM Based Design (2-bit Adder)

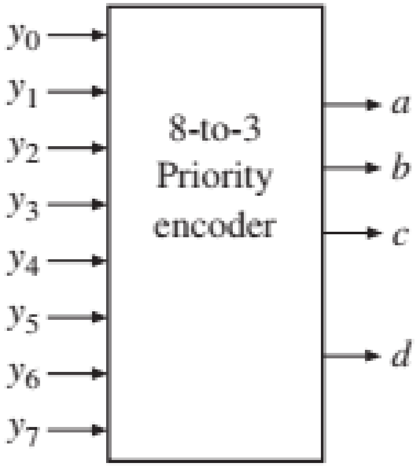




X_1	X_0	Y_1	Y_0	S_2	S_1	S_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

ROM Based Design (8:3 Priority Encoder)

Compute the size of the ROM required to implement an 8-to-3 priority encoder.

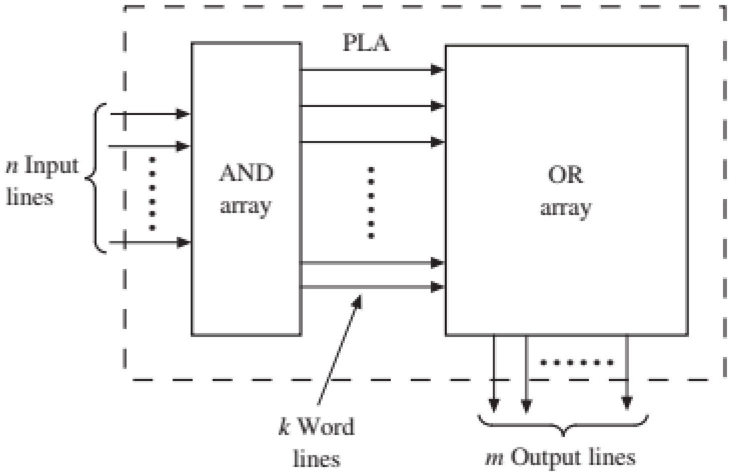


y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	a	b	c	d
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
X	1	0	0	0	0	0	0	0	0	1	1
X	X	1	0	0	0	0	0	0	1	0	1
X	X	X	1	0	0	0	0	0	1	1	1
X	X	X	X	1	0	0	0	1	0	0	1
X	X	X	X	X	1	0	0	1	0	1	1
X	X	X	X	X	X	1	0	1	1	0	1
X	X	X	X	X	X	X	1	1	1	1	1

it needs a $2^8 \times 4$ bit ROM.

Programmable Logic Arrays (PLAs)

Compute the size of the ROM required to implement an 8-to-3 priority encoder.



Programmable Logic Arrays (PLAs)

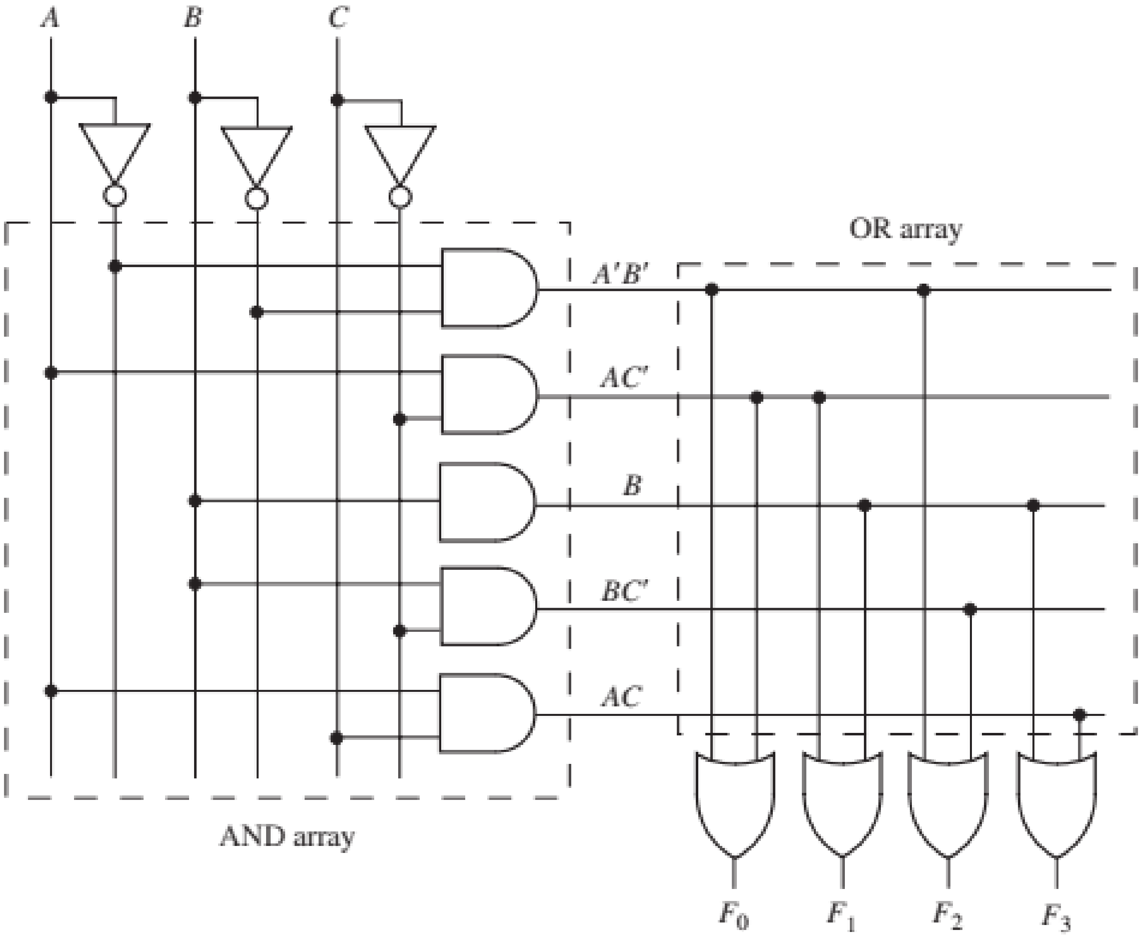
$$F_0 = \Sigma m(0, 1, 4, 6) = A'B' + AC$$

$$F_1 = \Sigma m(2, 3, 4, 6, 7) = B + AC'$$

$$F_2 = \Sigma m(0, 1, 2, 6) = A'B' + BC'$$

$$F_3 = \Sigma m(2, 3, 5, 6, 7) = AC + B$$

Product	Inputs			Outputs			
Term	A	B	C	F_0	F_1	F_2	F_3
$A'B'$	0	0	–	1	0	1	0
AC'	1	–	0	1	1	0	0
B	–	1	–	0	1	0	1
BC'	–	1	0	0	0	1	0
AC	1	–	1	0	0	0	1



Programmable Logic Arrays (PLAs)

- Realize the following functions using a PLA

$$F_1 = \Sigma m(2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$$

$$F_2 = \Sigma m(2, 3, 5, 6, 7, 10, 11, 14, 15)$$

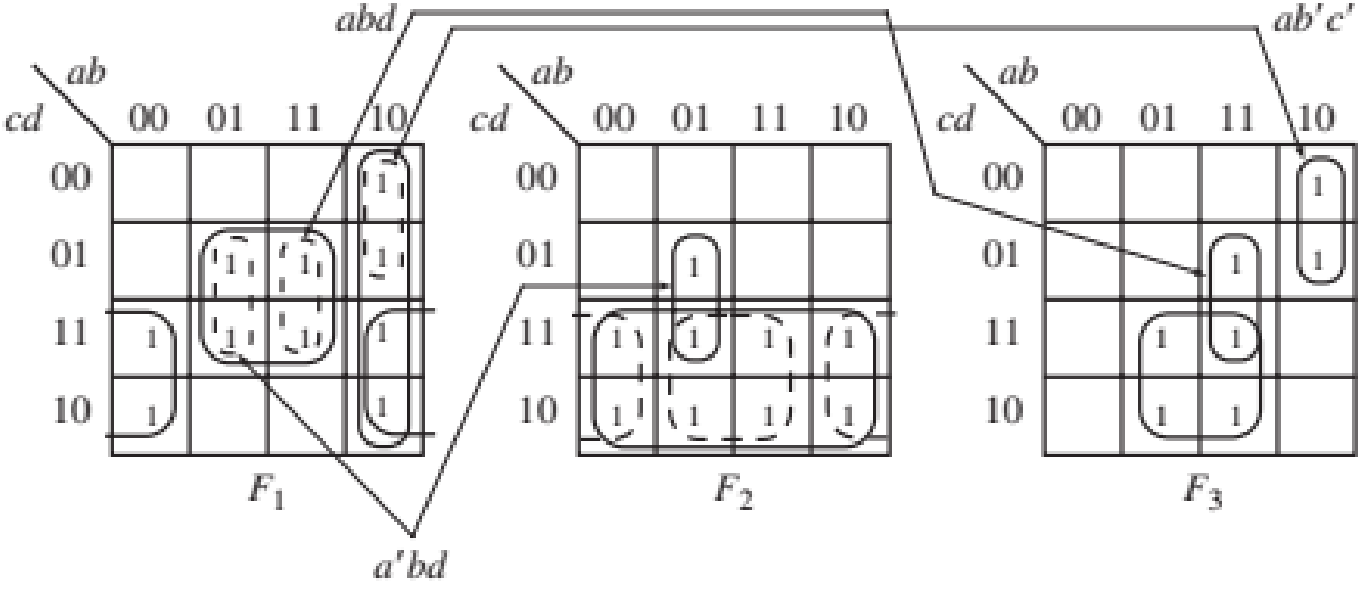
$$F_3 = \Sigma m(6, 7, 8, 9, 13, 14, 15)$$

$$F_1 = bd + b'c + ab'$$

$$F_2 = c + a'bd$$

$$F_3 = bc + ab'c' + abd$$

Programmable Logic Arrays (PLAs)



F_1

cd \ ab	00	01	11	10
00				1
01		1	1	1
11	1	1	1	1
10	1			1

F_2

cd \ ab	00	01	11	10
00				
01		1		
11	1	1	1	1
10	1	1	1	1

F_3

cd \ ab	00	01	11	10
00				1
01			1	1
11		1	1	
10		1	1	

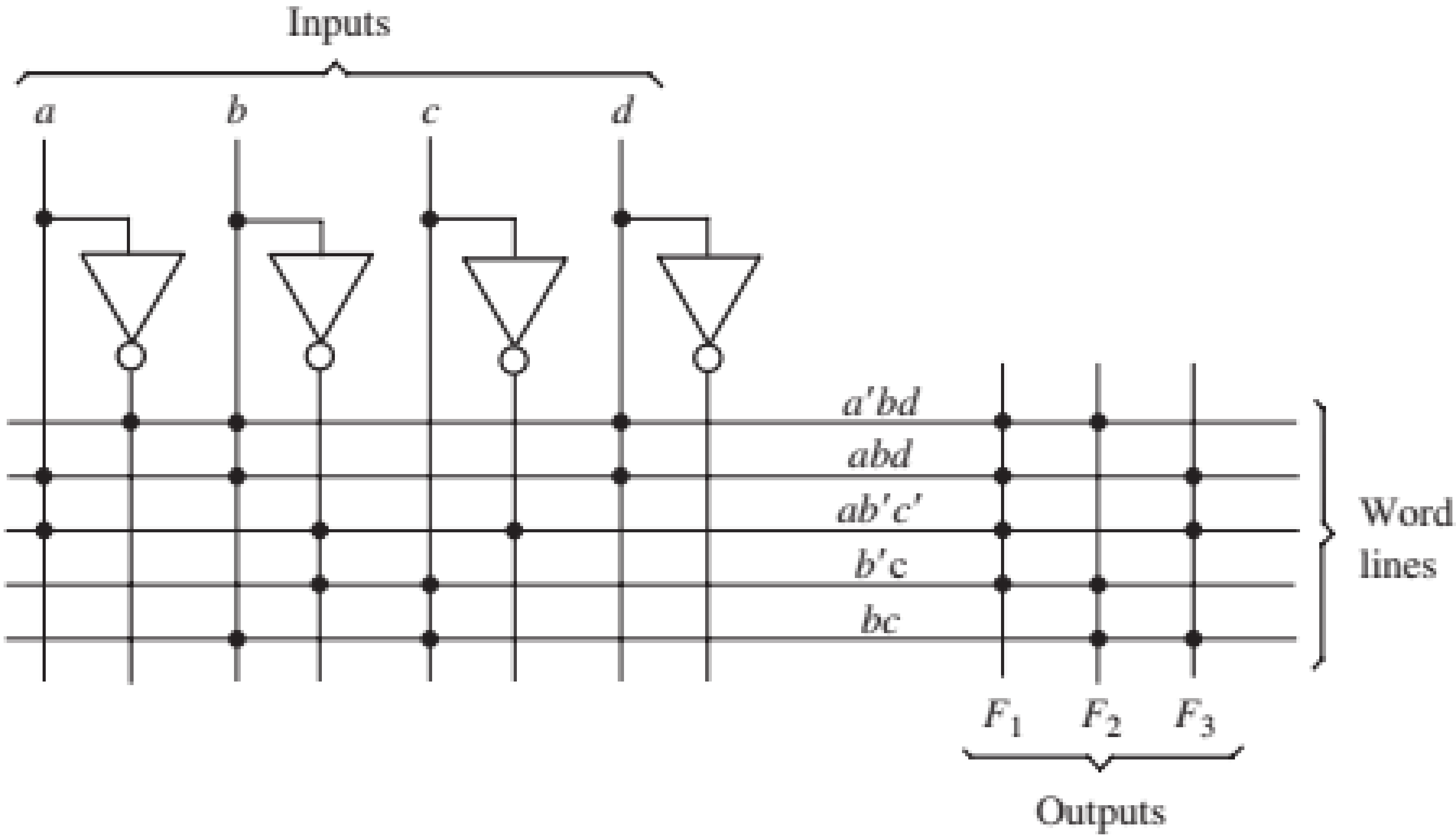
$$F_1 = a'bd + abd + ab'c' + b'c$$

$$F_2 = a'bd + b'c + bc$$

$$F_3 = abd + ab'c' + bc$$

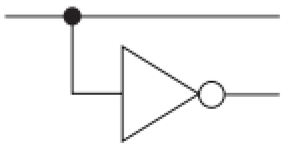
a	b	c	d	F_1	F_2	F_3
0	1	–	1	1	1	0
1	1	–	1	1	0	1
1	0	0	–	1	0	1
–	0	1	–	1	1	0
–	1	1	–	0	1	1

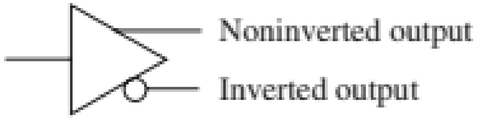
Programmable Logic Arrays (PLAs)

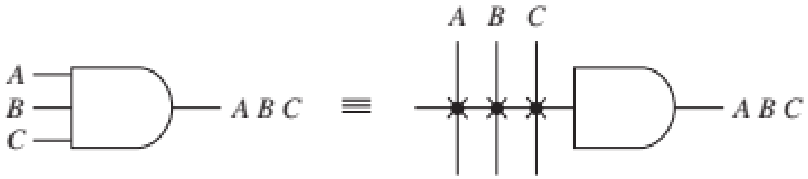


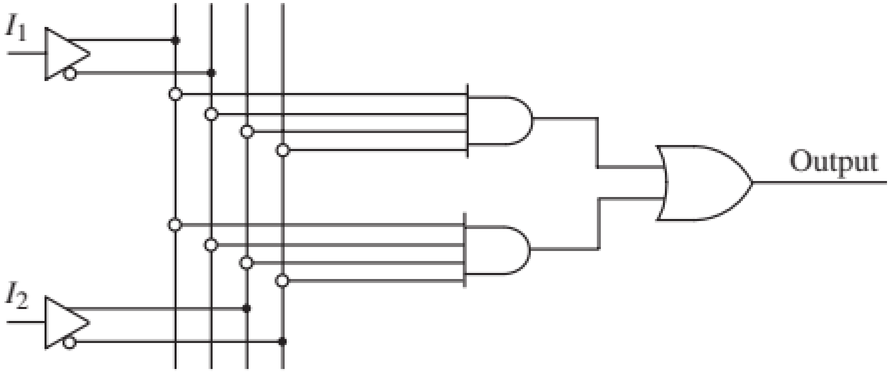
Programmable Array Logic (PAL)

- AND array is programmable, and the OR array is fixed

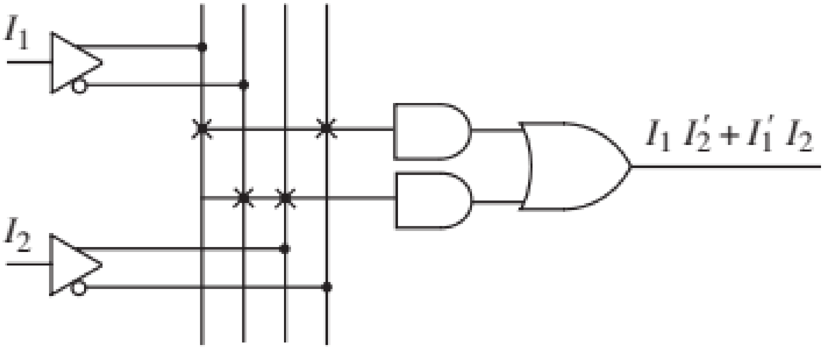








Unprogrammed



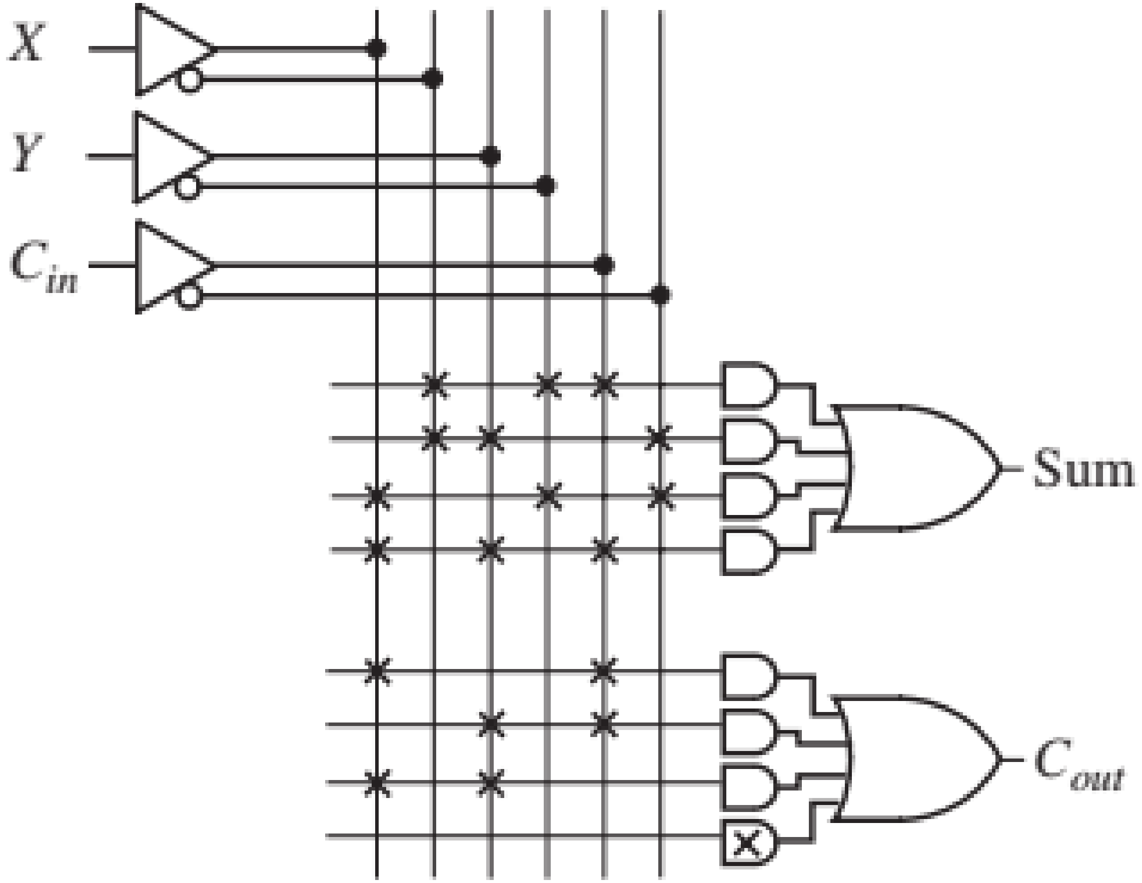
Programmed

Programmable Array Logic (PAL)

- Full Adder

$$Sum = X'Y'C_{in} + X'YC_{in}' + XY'C_{in}' + XYC_{in}$$

$$C_{out} = XC_{in} + YC_{in} + XY$$



Generic Array Logic (GAL)

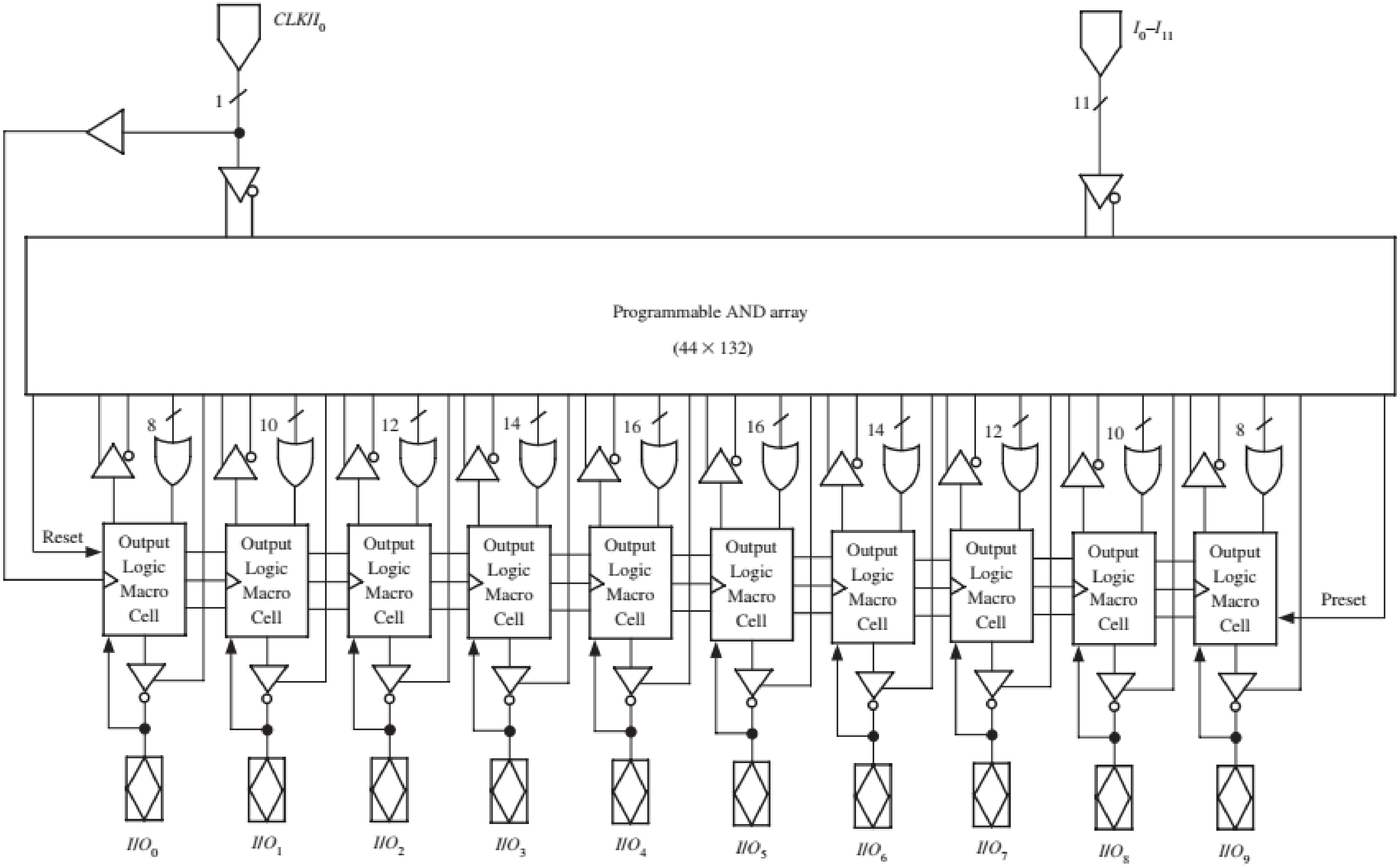
- PALs and PLAs were widely used for small circuitry and interface logic.
- With advances in IC technology, newer types of PLDs became available
- Traditional PALs were not reprogrammable → once programmed, fixed permanently.
- Now there are flash erasable/reprogrammable PALs, also called: PLDs ,GALs (Generic Array Logic)
- Example device: 22CEV10 (a CMOS electrically erasable PLD)
  - Can implement both combinational and sequential circuits.
  - Contains AND–OR arrays (like PALs) plus macroblocks.
  - Macroblocks include multiplexers and extra programmability.

Generic Array Logic (GAL)

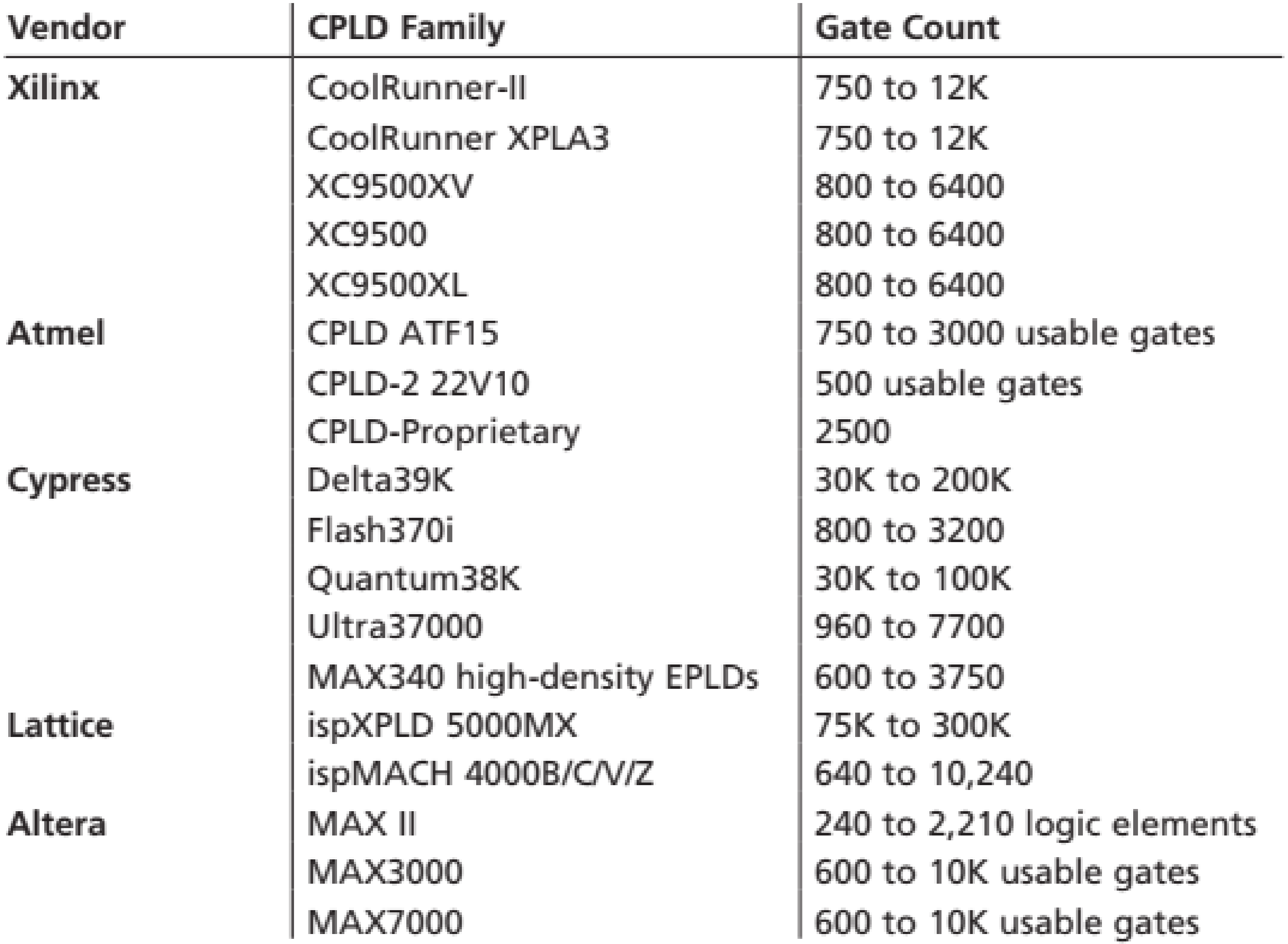
22CEV10 Naming Breakdown

- "22" → Total pins on the device (22-pin package).
- "C" → CMOS technology implementation.
- "E" → Electrically Erasable / Reprogrammable (EEPROM/Flash-based).
- "V" → Advanced version / enhanced features (vendor-specific, typically indicates extra macrocell flexibility compared to older PALs).
- "10" → Number of macrocells (outputs)Each macrocell can be configured as registered (with D flip-flop) or combinational output. Can also be configured as I/O pin.

Generic Array Logic (GAL)

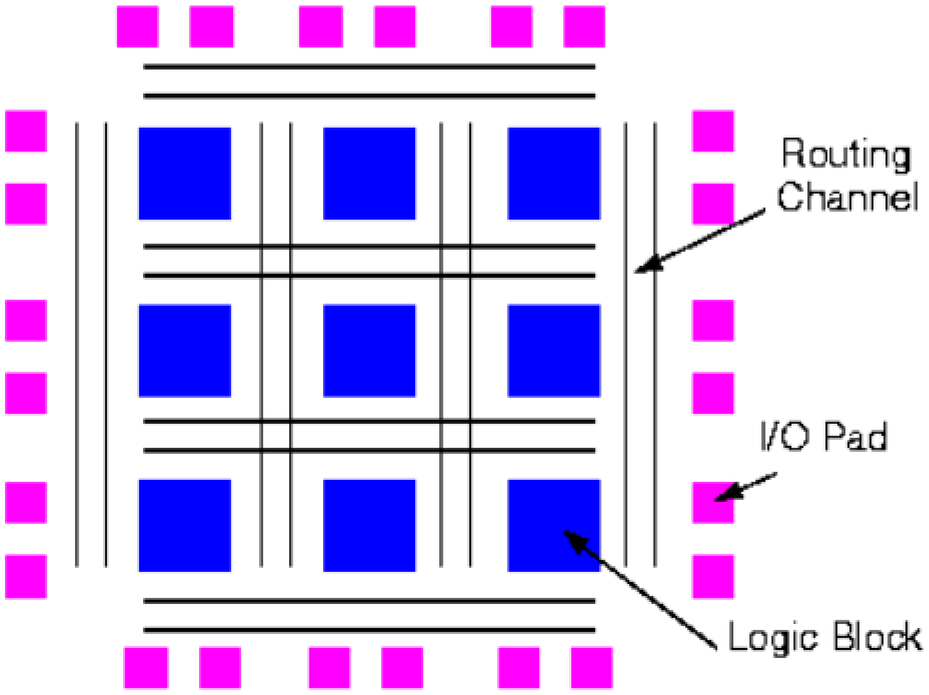


CPLD Chips



Vendor	CPLD Family	Gate Count
Xilinx	CoolRunner-II	750 to 12K
	CoolRunner XPLA3	750 to 12K
	XC9500XV	800 to 6400
	XC9500	800 to 6400
	XC9500XL	800 to 6400
Atmel	CPLD ATF15	750 to 3000 usable gates
	CPLD-2 22V10	500 usable gates
	CPLD-Proprietary	2500
Cypress	Delta39K	30K to 200K
	Flash370i	800 to 3200
	Quantum38K	30K to 100K
	Ultra37000	960 to 7700
	MAX340 high-density EPLDs	600 to 3750
Lattice	ispXPLD 5000MX	75K to 300K
	ispMACH 4000B/C/V/Z	640 to 10,240
Altera	MAX II	240 to 2,210 logic elements
	MAX3000	600 to 10K usable gates
	MAX7000	600 to 10K usable gates

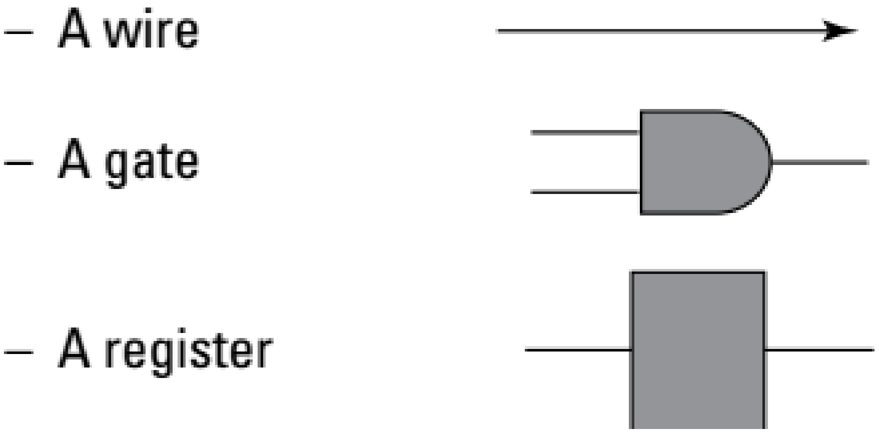
Field Programmable Grid Array



- Combined idea of PLD and gate arrays
- First introduced by Xilinx in 1985
- Array of logic blocks
- Lot of programmable wiring
- Very flexible I/O interfacing
- Two dominant FPGA makers
  - Xilinx and Altera
- Other specialist makers
  - Actel and Lattice Logic

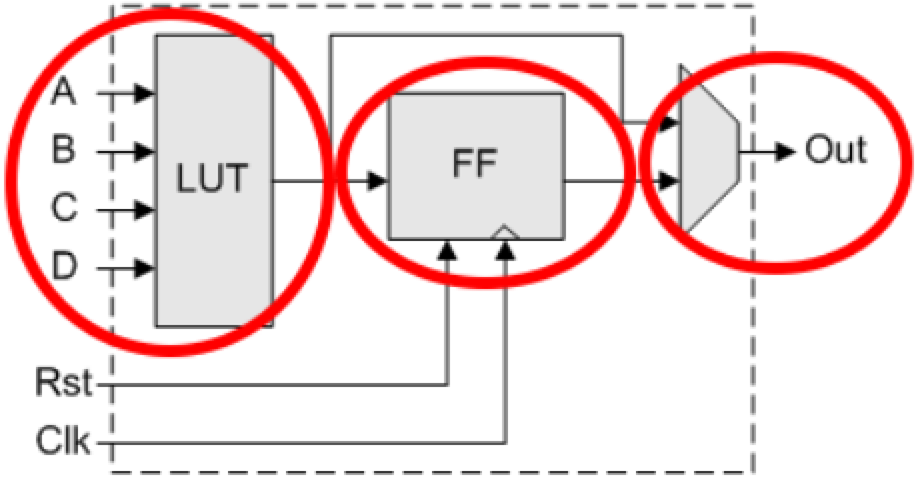
Building blocks of FPGA

- A *register* remembers a piece of information until it is told to remember something else
- A *logic gate* performs simple logic operations on signals
- *A wire* connects these other pieces

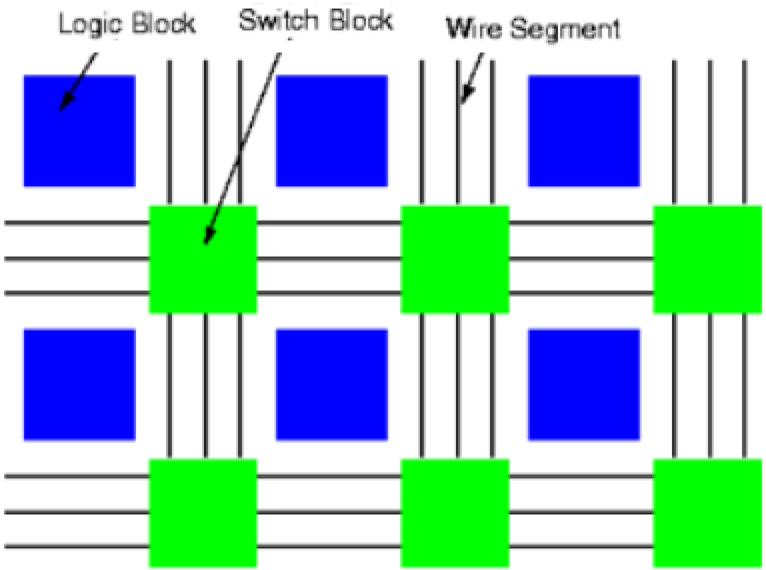


Older ways of Digital Design

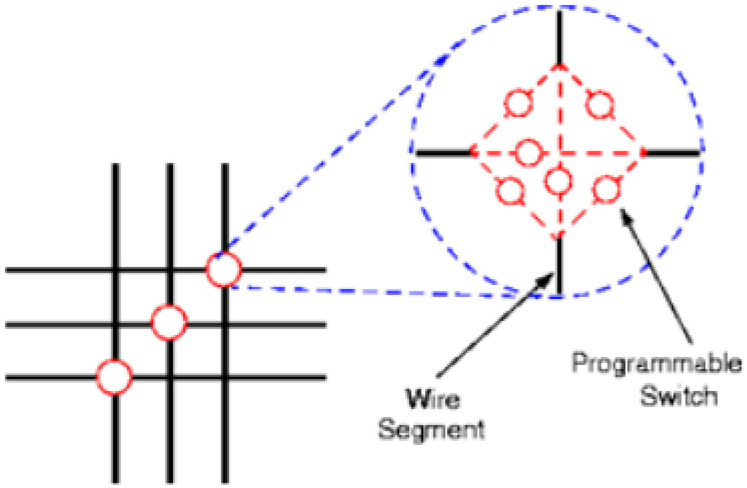
- Based on Lookup Table ,most common 4 input
- Optional D-FF at the output
- 4-input LUT can implement any Boolean function
- Special circuits for cascading logic blocks (carry chains)

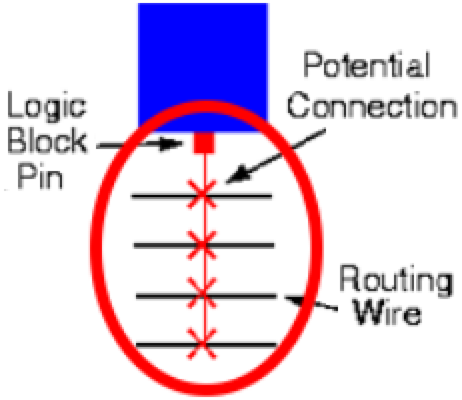


Programmable Routing



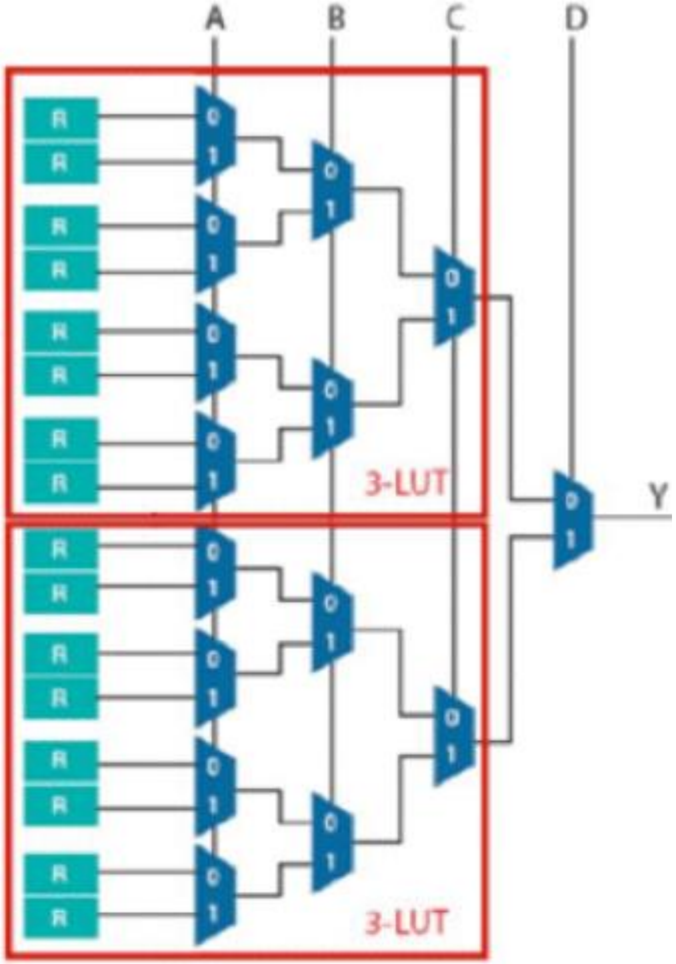
- Wiring channels between rows and column
- These are programmable
- Each wiring segment can be connected in one or many ways





Idea of configuring FPGA

- Programming FPGA is not same as microprocessor
- We download bitstream (not a program) to an FPGA
- Programming FPGA is known as configuration
- LUTs are configured to implement the Boolean logic

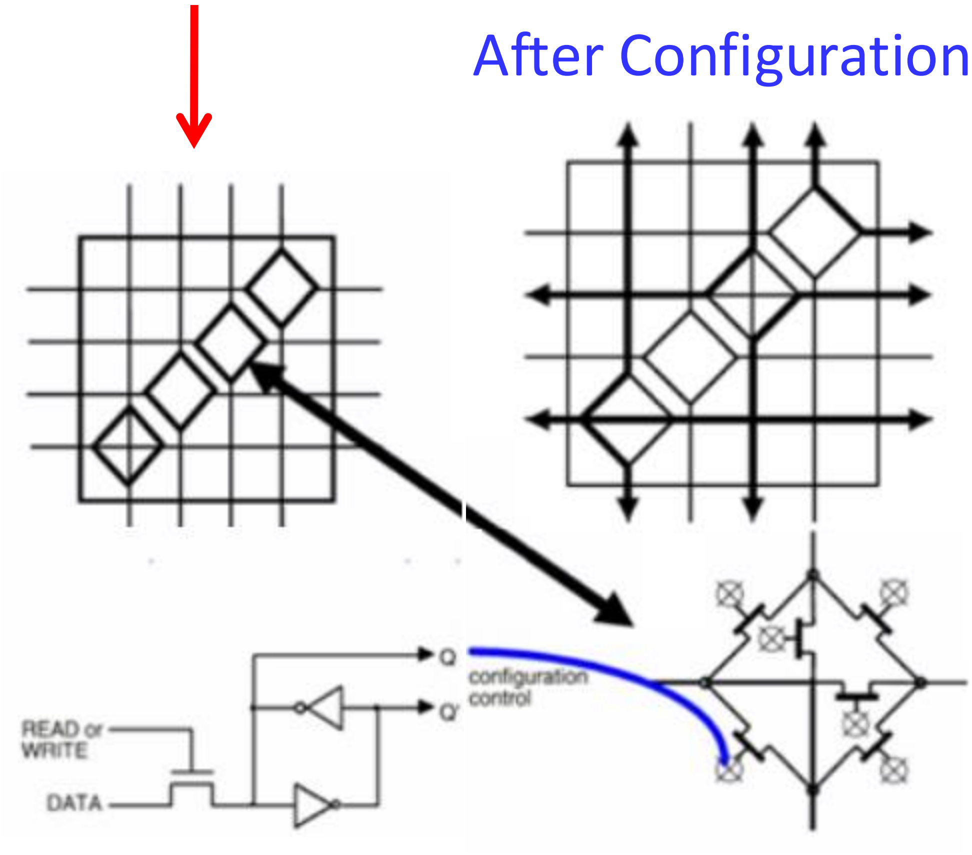


Configuring the Routing of FPGA

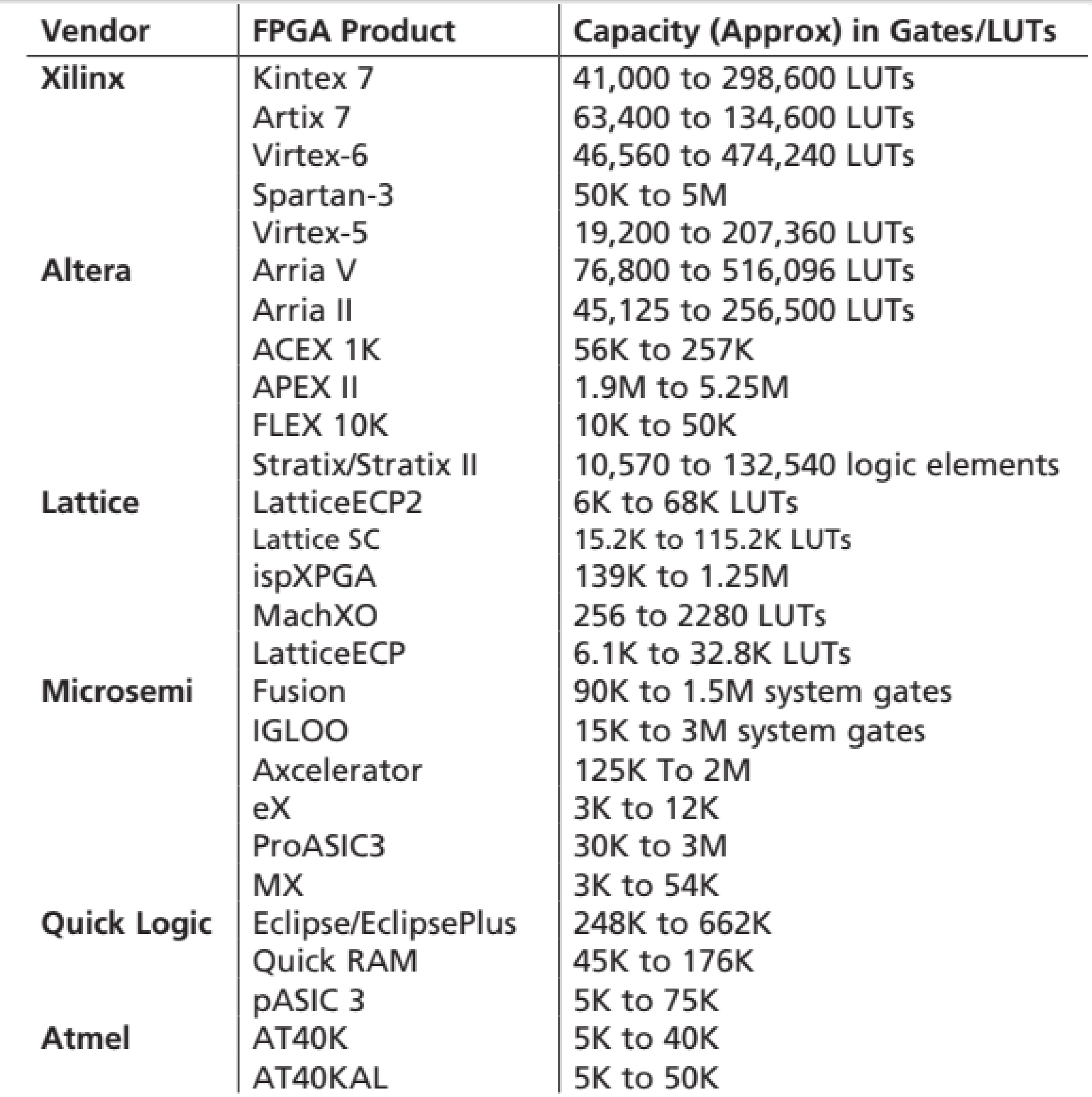
- At each interconnect transistor switch OFF state
- Each switch is controlled by output of 1-bit configuration register
- Configuring the routing simply to put a ‘1’ or ‘0’in register
- Bitstream is either stored on local flash memory or download via computer
- Configuration happens on power-up

Before Configuration

After Configuration



FPGA

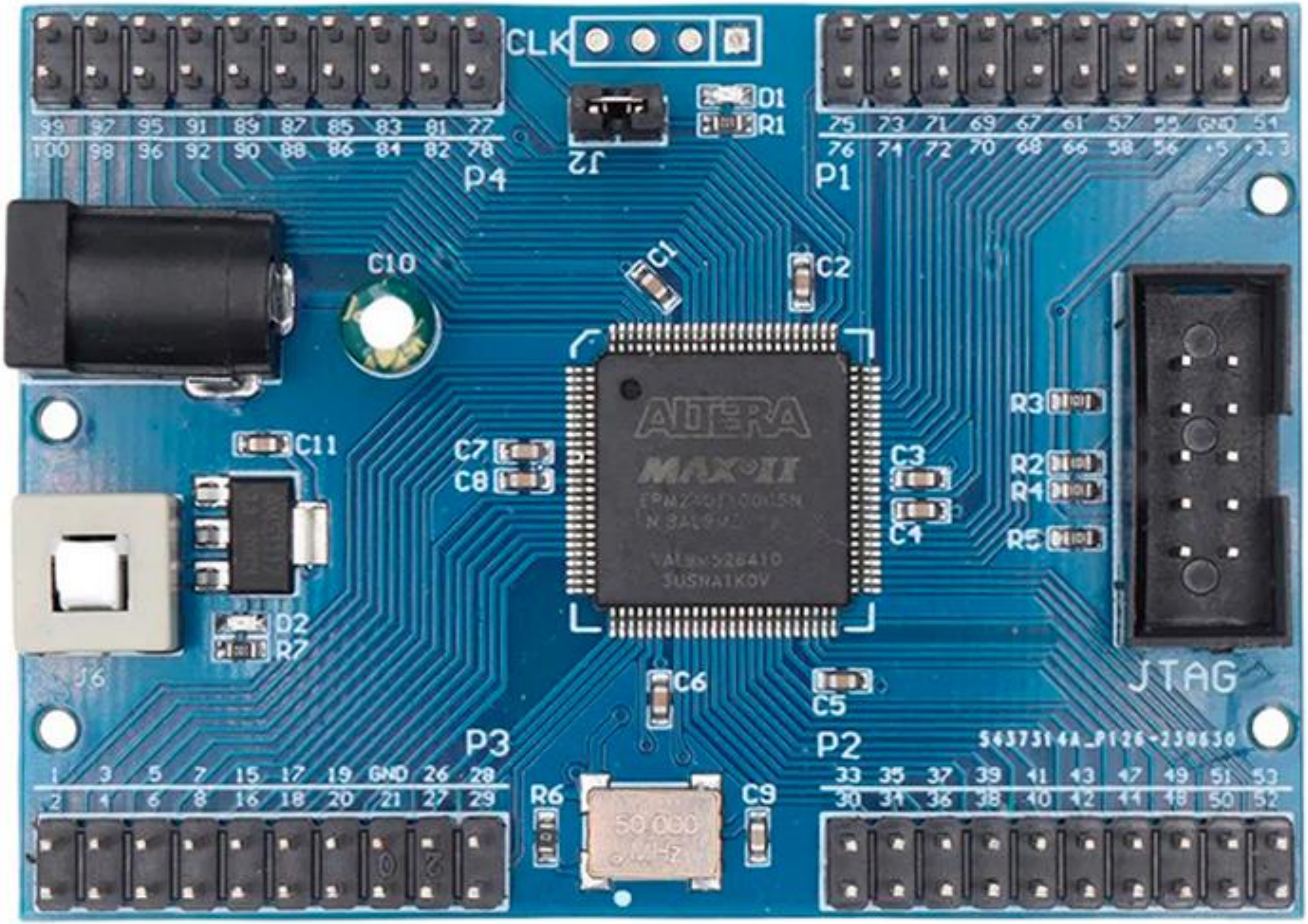


Vendor	FPGA Product	Capacity (Approx) in Gates/LUTs
Xilinx	Kintex 7	41,000 to 298,600 LUTs
	Artix 7	63,400 to 134,600 LUTs
	Virtex-6	46,560 to 474,240 LUTs
	Spartan-3	50K to 5M
	Virtex-5	19,200 to 207,360 LUTs
Altera	Arria V	76,800 to 516,096 LUTs
	Arria II	45,125 to 256,500 LUTs
	ACEX 1K	56K to 257K
	APEX II	1.9M to 5.25M
	FLEX 10K	10K to 50K
	Stratix/Stratix II	10,570 to 132,540 logic elements
Lattice	LatticeECP2	6K to 68K LUTs
	Lattice SC	15.2K to 115.2K LUTs
	ispXPGA	139K to 1.25M
	MachXO	256 to 2280 LUTs
	LatticeECP	6.1K to 32.8K LUTs
Microsemi	Fusion	90K to 1.5M system gates
	IGLOO	15K to 3M system gates
	Axcelerator	125K To 2M
	eX	3K to 12K
	ProASIC3	30K to 3M
	MX	3K to 54K
Quick Logic	Eclipse/EclipsePlus	248K to 662K
	Quick RAM	45K to 176K
	pASIC 3	5K to 75K
Atmel	AT40K	5K to 40K
	AT40KAL	5K to 50K

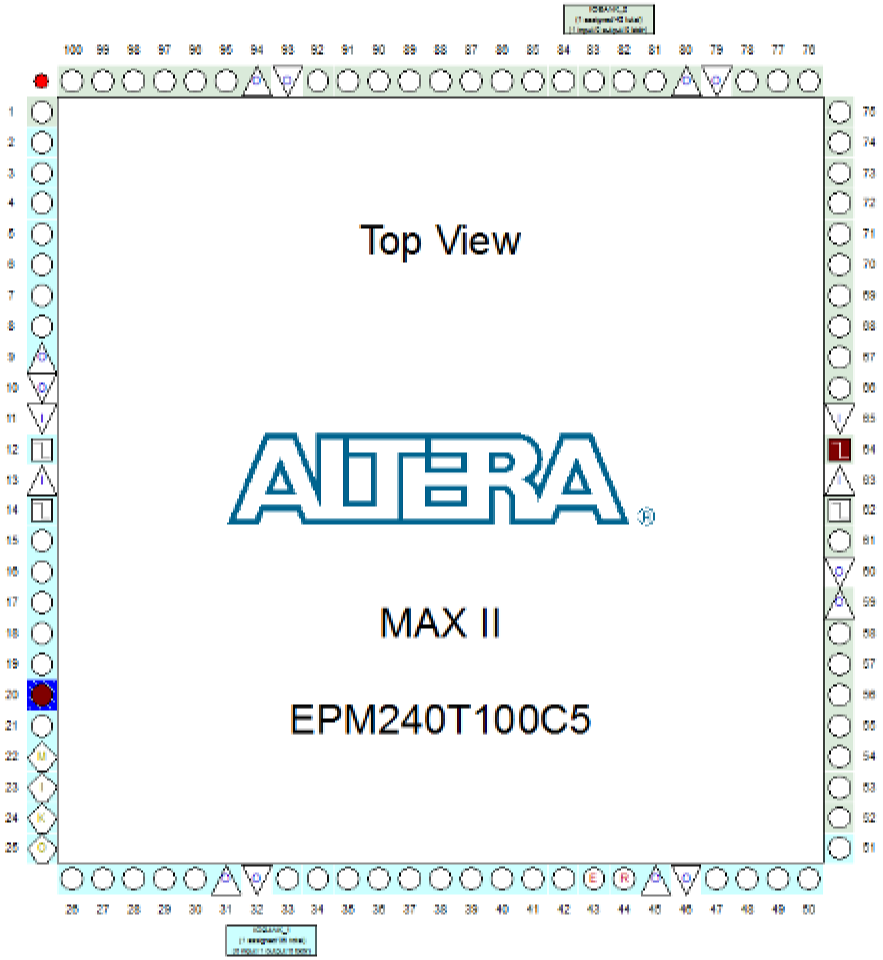
FPGA Vs Microcontroller

- With a microcontroller you write software
- The biggest limitation of software is that you can only do one thing at a time!
- With FPGAs you are not creating software
- You are designing the hardware!
- Instead of writing code to run on a fixed processor with fixed peripherals, you get to design your own circuit.
- If you really want to you can even create your very own processor and write software to run on it!
- A huge benefit of working with FPGAs is that every element of your design runs independently of each other
- That means one part of your design can be reading in some serial data, while another part is controlling a servo, while another is reading some sensors, and yet another is controlling a display

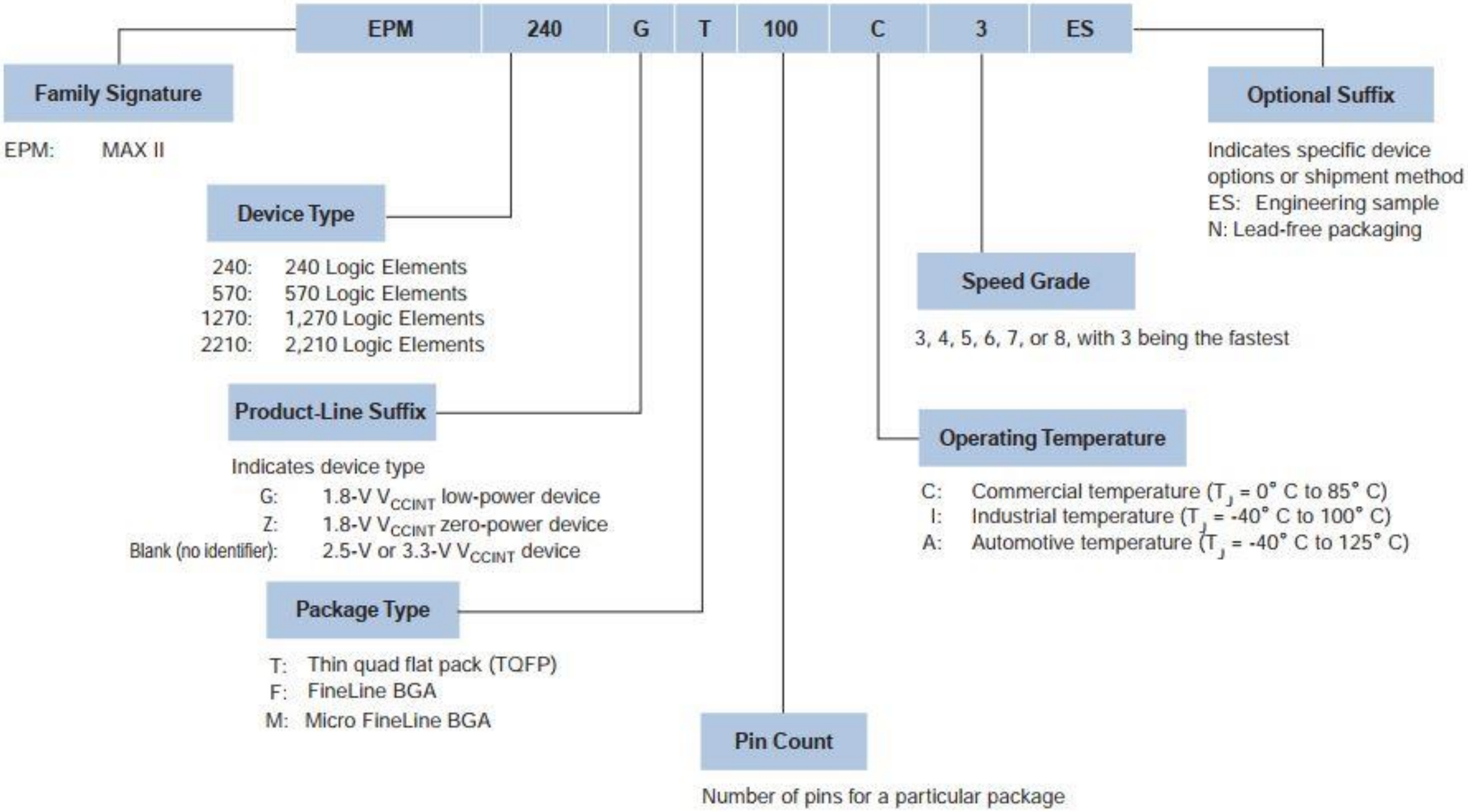
CPLD Development Board



CPLD Development Board



CPLD Board



EPM	240	G	T	100	C	3	ES

Family Signature

EPM:	MAX II

Device Type

240:	240 Logic Elements
570:	570 Logic Elements
1270:	1,270 Logic Elements
2210:	2,210 Logic Elements

Product-Line Suffix

Indicates device type

G:	1.8-V V_{CCINT} low-power device
Z:	1.8-V V_{CCINT} zero-power device
Blank (no identifier):	2.5-V or 3.3-V V_{CCINT} device

Package Type

T:	Thin quad flat pack (TQFP)
F:	FineLine BGA
M:	Micro FineLine BGA

Pin Count

Number of pins for a particular package

Operating Temperature

C:	Commercial temperature (T_J = 0° C to 85° C)
I:	Industrial temperature (T_J = -40° C to 100° C)
A:	Automotive temperature (T_J = -40° C to 125° C)

Speed Grade

3, 4, 5, 6, 7, or 8, with 3 being the fastest

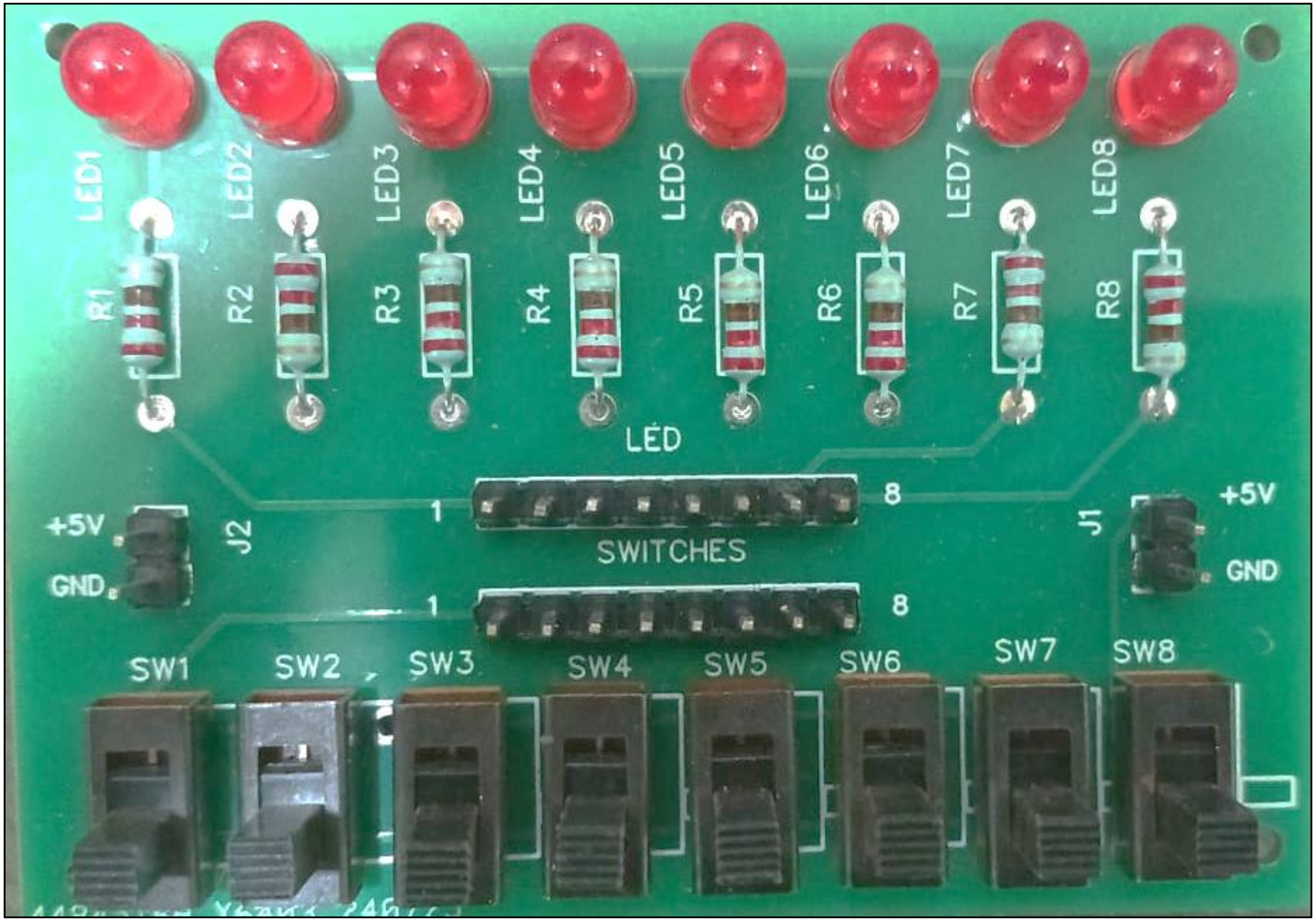
Optional Suffix

Indicates specific device options or shipment method

ES:	Engineering sample
N:	Lead-free packaging

EPM240T100C5

Extension Kit



HDL Programming

Verilog Code NOT Gate

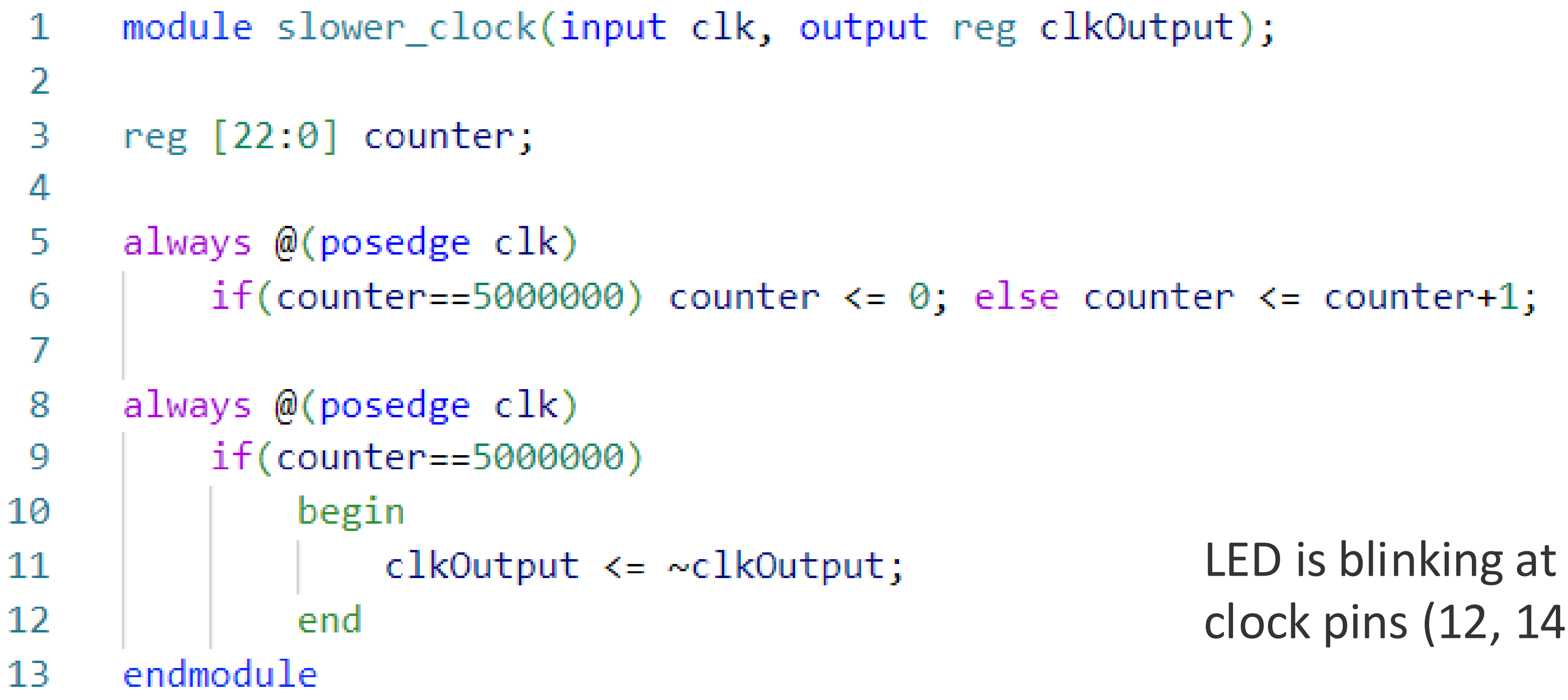
Verilog Code

```
module inverter ( input wire a, output wire y );

        assign y = ~a;

endmodule
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity inverter is
    Port (
        a : in STD_LOGIC;   -- Input signal
        y : out STD_LOGIC   -- Output signal
    );
end inverter;

architecture Behavioral of inverter is
begin
    y <= not a;  -- Output is the negation
(inversion) of the input
end Behavioral;
```

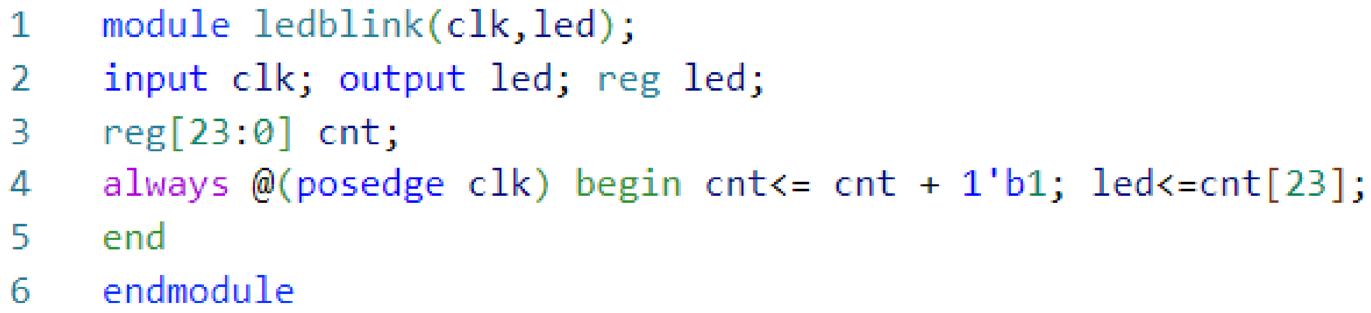
VHDL Code

Slower Clock



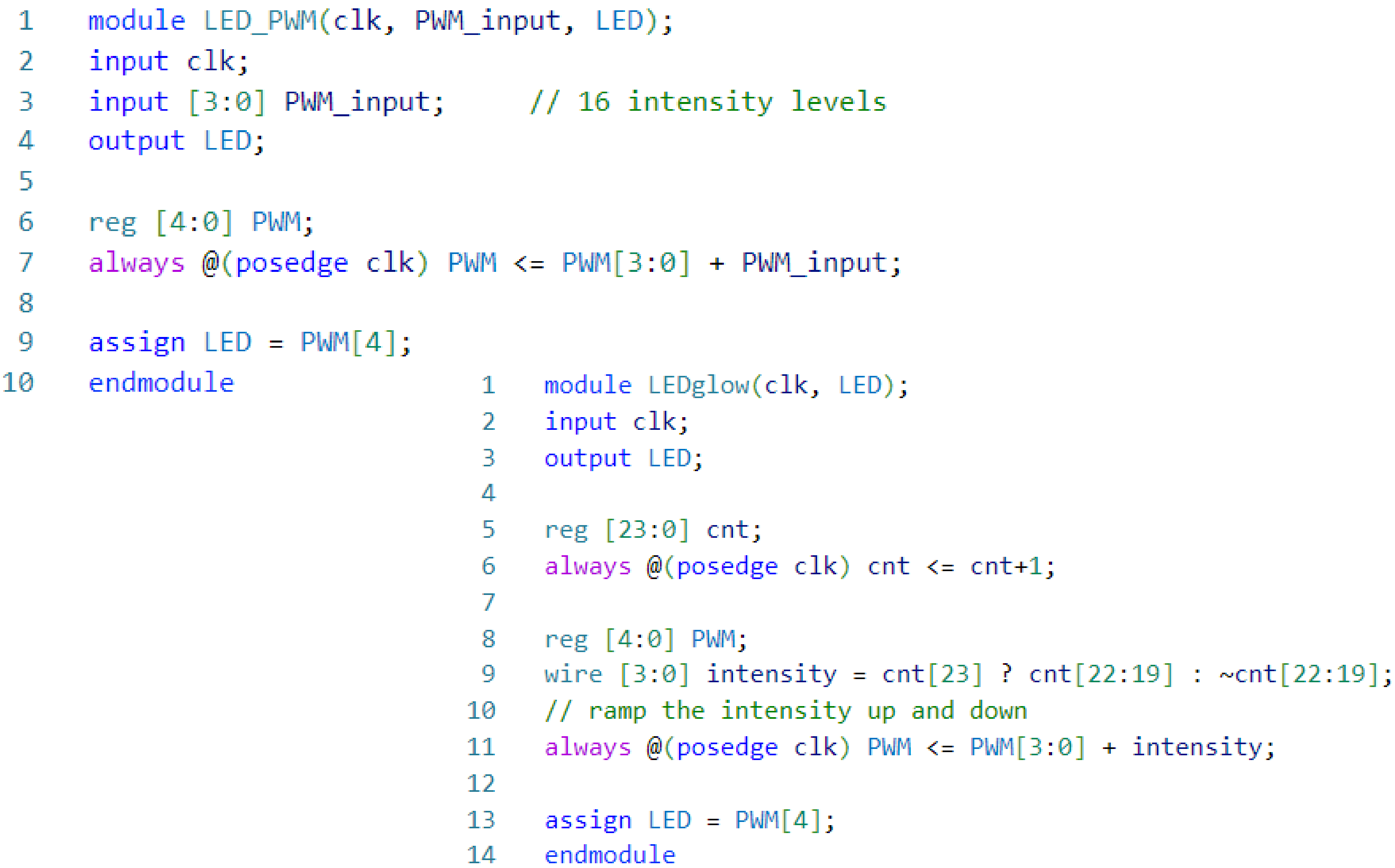
```
module slower_clock(input clk, output reg clkOutput);

reg [22:0] counter;

always @(posedge clk)
    if(counter==5000000) counter <= 0; else counter <= counter+1;

always @(posedge clk)
    if(counter==5000000)
        begin
            clkOutput <= ~clkOutput;
        end
endmodule
```

LED is blinking at 5Hz
clock pins (12, 14, 62)



```
module ledblink(clk,led);
input clk; output led; reg led;
reg[23:0] cnt;
always @(posedge clk) begin cnt<= cnt + 1'b1; led<=cnt[23];
end
endmodule
```

Fine Tuning with PWM



```verilog
module LED_PWM(clk, PWM_input, LED);
input clk;
input [3:0] PWM_input;     // 16 intensity levels
output LED;

reg [4:0] PWM;
always @(posedge clk) PWM <= PWM[3:0] + PWM_input;

assign LED = PWM[4];
endmodule
```

```verilog
module LEDglow(clk, LED);
input clk;
output LED;

reg [23:0] cnt;
always @(posedge clk) cnt <= cnt+1;

reg [4:0] PWM;
wire [3:0] intensity = cnt[23] ? cnt[22:19] : ~cnt[22:19];
// ramp the intensity up and down
always @(posedge clk) PWM <= PWM[3:0] + intensity;

assign LED = PWM[4];
endmodule
```

Verilog Code for LED Blinking

```
module counter(clk, out);
 input clk;       // Clock input
 output out;      // Output signal
 reg out;         // Register for the output
 reg [23:0] count; // 24-bit register for the counter
 // Always block triggered on the positive edge of the clock
 always @(posedge clk) begin
   count <= count + 1'b1; // Increment the counter by 1
   out <= count[23];      // Assign the 24th bit of the counter to the output
 end
endmodule
```

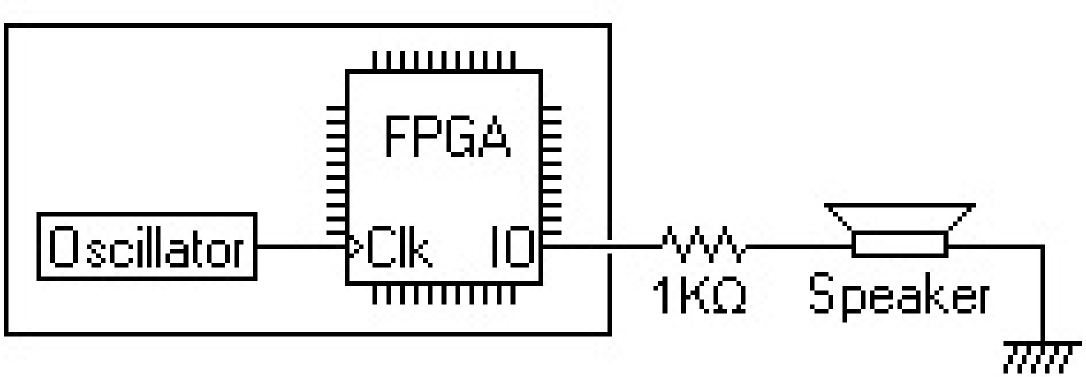
VHDL Code for LED Blinking

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity counter is
    Port (
        clk : in STD_LOGIC;  -- Clock input
        out : out STD_LOGIC  -- Output signal
    );
end counter;

architecture Behavioral of counter is
    signal count : STD_LOGIC_VECTOR(23 downto 0) := (others => '0'); -- 24-bit counter signal
begin
    process(clk)
    begin
        if rising_edge(clk) then
            count <= count + 1;        -- Increment the counter
            out <= count(23);          -- Assign the 24th bit of the counter to the output
        end if;
    end process;
end Behavioral;
```

Music Generation

Music

- Simple beeps
- Sirens
- Notes
- Tune



Music: Simple Beep (Verilog)

```
module music(clk, speaker);
input clk;
output speaker;

// first create a 16bit binary counter
reg [15:0] counter;
always @(posedge clk) counter <= counter+1;

// and use the most significant bit (MSB) of the counter to drive
the speaker
assign speaker = counter[15];
endmodule
```

Music: Simple Beep (VHDL)

```
entity music is
    Port (
        clk     : in  STD_LOGIC;  -- Clock input
        speaker : out STD_LOGIC    -- Speaker output
    );
end music;

architecture Behavioral of music is
signal counter : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
begin
process(clk)
    begin
        if rising_edge(clk) then
            counter <= counter + 1;
        end if;
    end process;

speaker <= counter(15);
end Behavioral;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

Thank you !

Happy Learning