ENR-325/325L Principles of Digital Electronics and Laboratory

Xiang Li

Fall 2025

Configuration on the go: PLD and FPGA

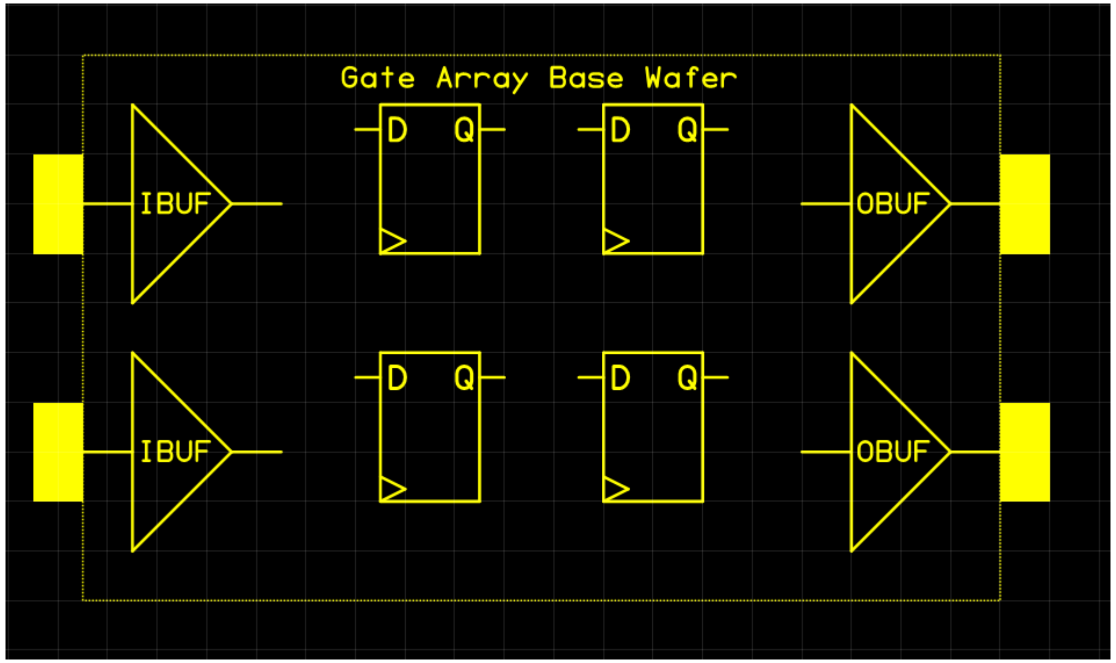
- PLD: Programmable Logic Device
  - PROM: Programmable Read-Only Memory (1960s)
  - PLA: Programmable Logic Array (1970s)
  - PAL: Programmable Array Logic (1970s)
- FPGA: Field-Programmable Gate Array (1990s)

What is the meaning of hardware coding?

As a small dev:

There’s NO IC design. This is the only standard CMOS layout I can afford.

(2 input, 2 output, four D flip-flops).

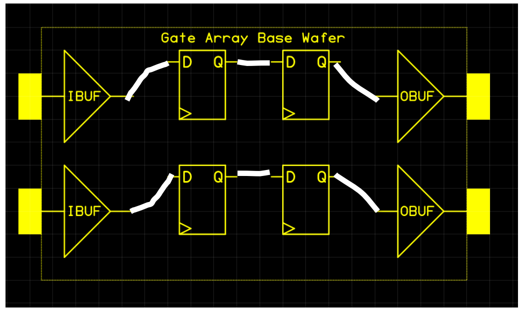


What is the meaning of hardware coding?

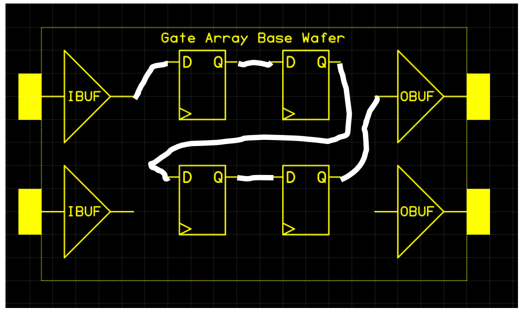
But the top wiring (aka additional routing layer) is another story.

This is GA (Gate-Arrays).

The cost is greatly reduced since I only need to pay for the customized top layer.



This is a two 2-bit data I/O.

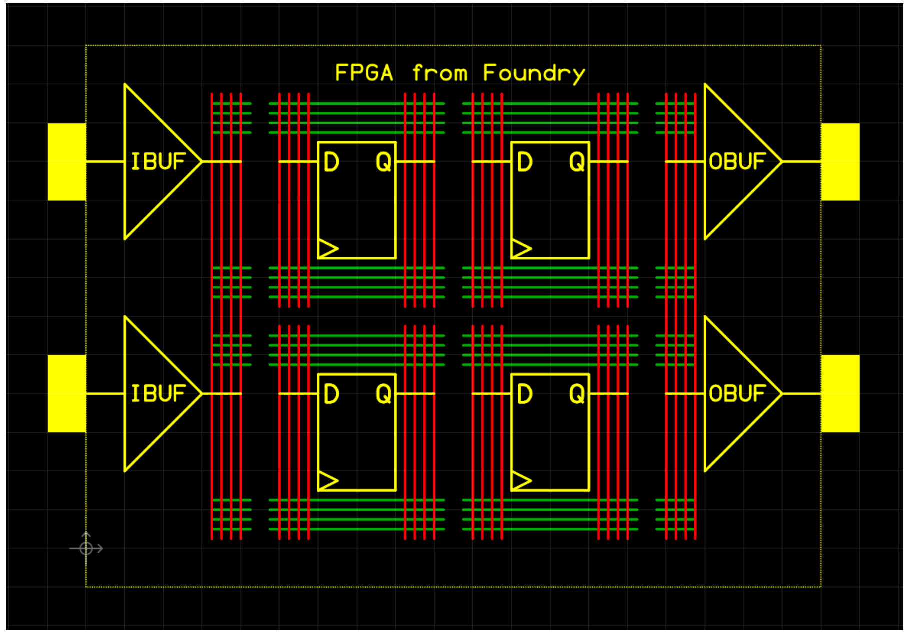


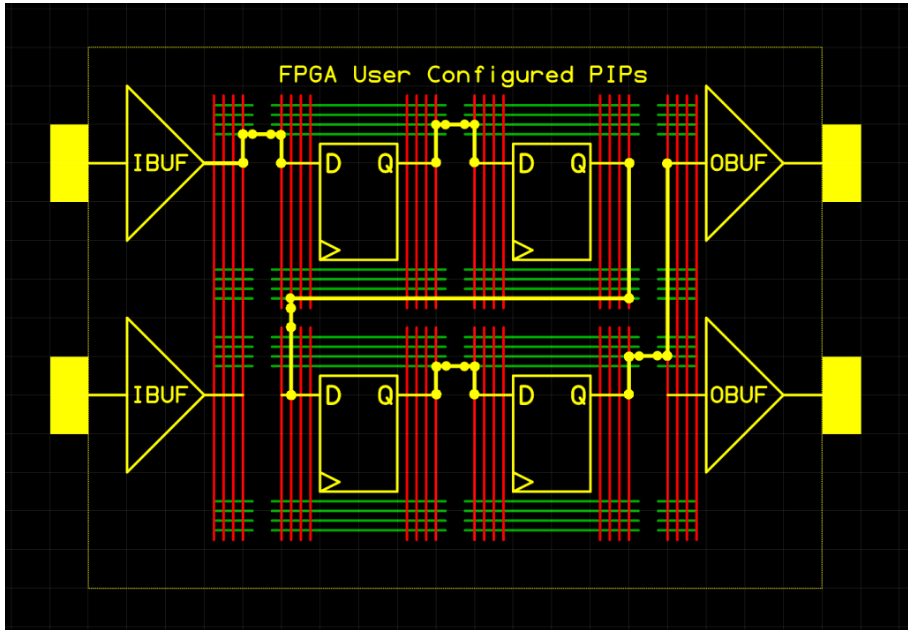
This is a one 4-bit data I/O.

What is the meaning of hardware coding?

And if there are built-in wiring on the IC level, we can greatly increase the speed of the chip.

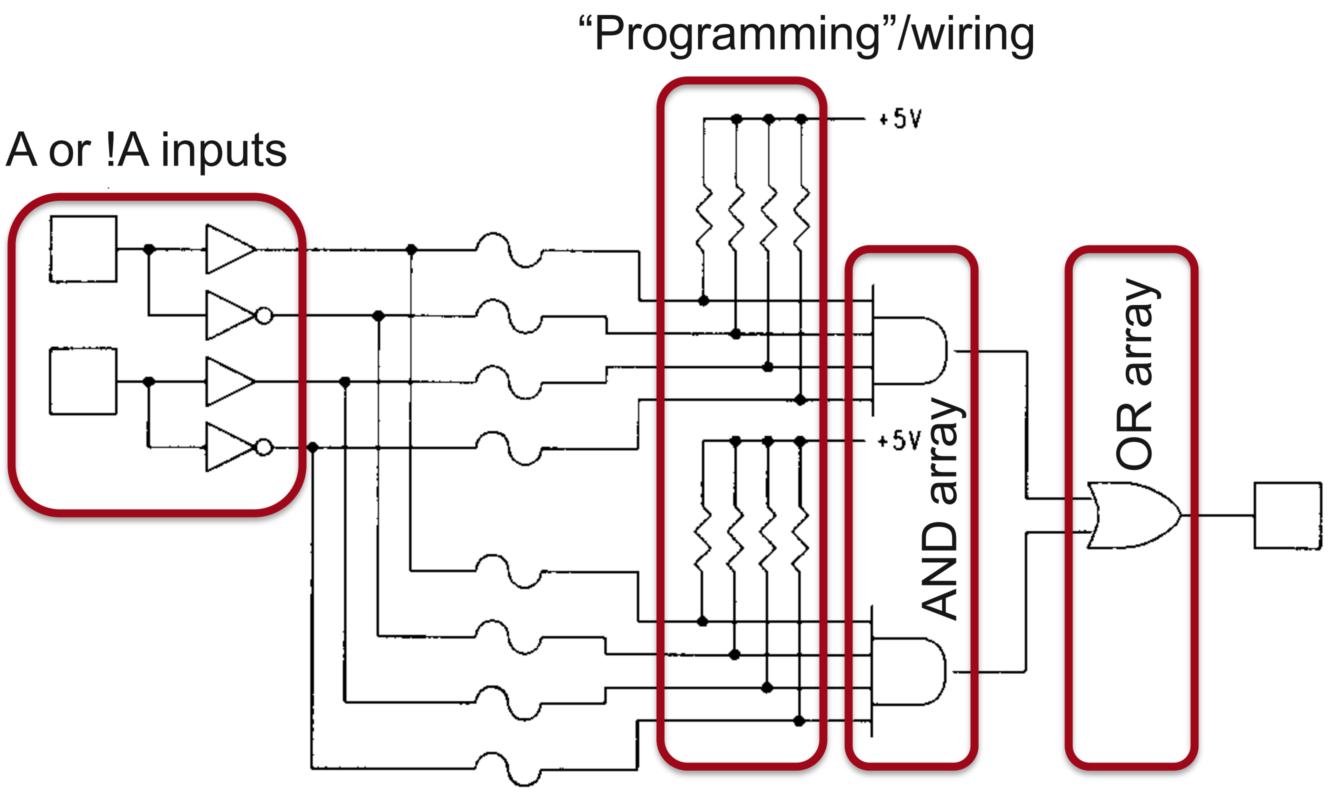
The build-in wiring is called: Programmable Interconnect Points (PIPs).



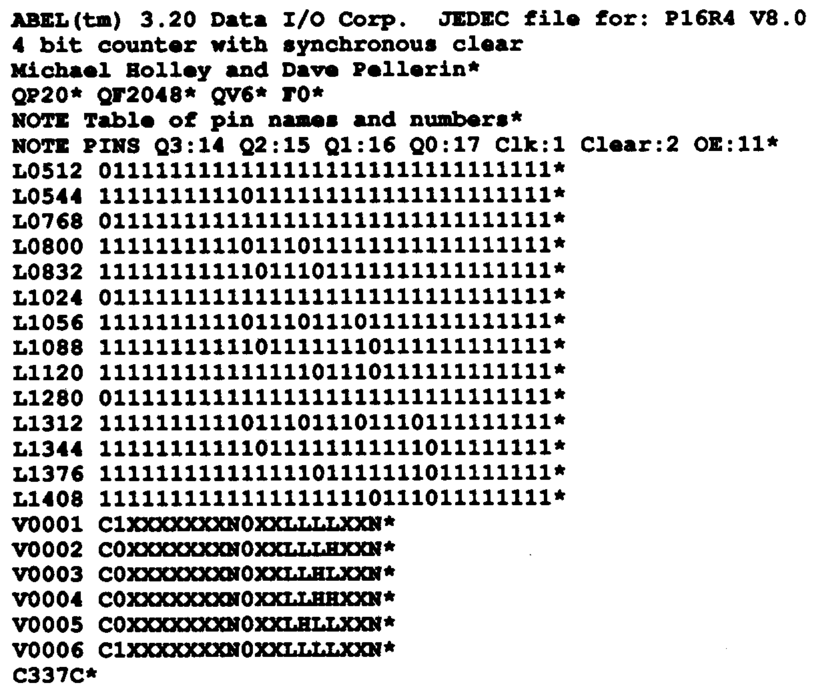


How PIPs are done in the early days:

Do less by doing more:



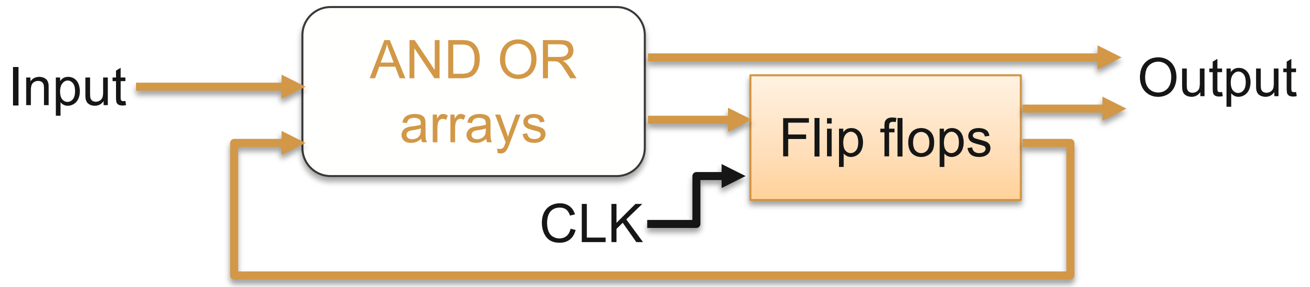
Example of codes (1989) that do the wiring:



```
ABEL(tm) 3.20 Data I/O Corp.  JEDEC file for: P16R4 V8.0
4 bit counter with synchronous clear
Michael Holley and Dave Pellerin*
QP20* QF2048* QV6* F0*
NOTE Table of pin names and numbers*
NOTE PINS Q3:14 Q2:15 Q1:16 Q0:17 Clk:1 Clear:2 OE:11*
L0512 01111111111111111111111111111111*
L0544 11111111101111111111111111111111*
L0768 01111111111111111111111111111111*
L0800 11111111101110111111111111111111*
L0832 11111111110111011111111111111111*
L1024 01111111111111111111111111111111*
L1056 11111111101110111011111111111111*
L1088 11111111110111111101111111111111*
L1120 11111111111111101110111111111111*
L1280 01111111111111111111111111111111*
L1312 11111111101110111011101111111111*
L1344 11111111110111111111111101111111*
L1376 11111111111111101111111101111111*
L1408 11111111111111111111101110111111*
V0001 C1XXXXXXXN0XXLLLLXXN*
V0002 C0XXXXXXXN0XXLLLHXXN*
V0003 C0XXXXXXXN0XXLLHLXXN*
V0004 C0XXXXXXXN0XXLLHHXXN*
V0005 C0XXXXXXXN0XXLHLLXXN*
V0006 C1XXXXXXXN0XXLLLLXXN*
C337C*
```

http://www.pldworld.com/html/technote/Tour_of_PLDs.htm

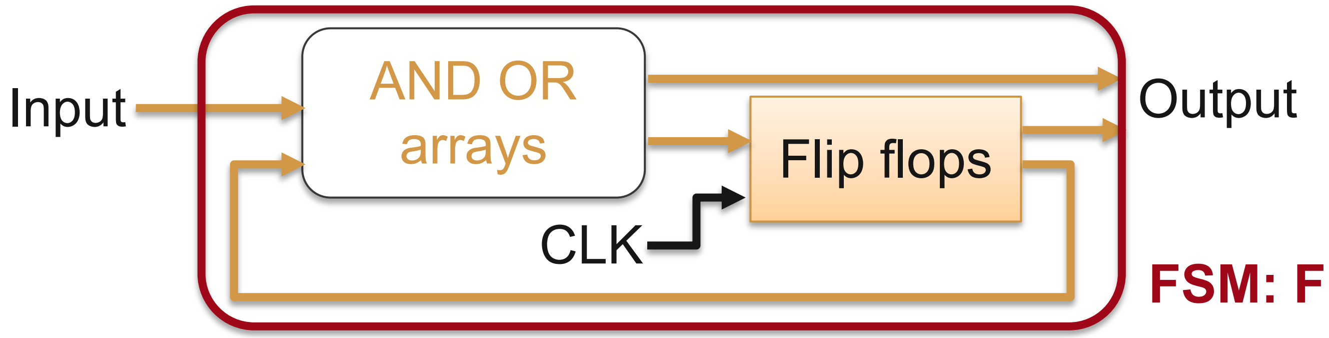
Let's do higher level of abstraction:



With the help of flip flops, now the system can store **state** variables.
The system can then output:

- Function of states (Moore machine)
- Function of sates and inputs (Mealy machine)

Let’s do higher level of abstraction:

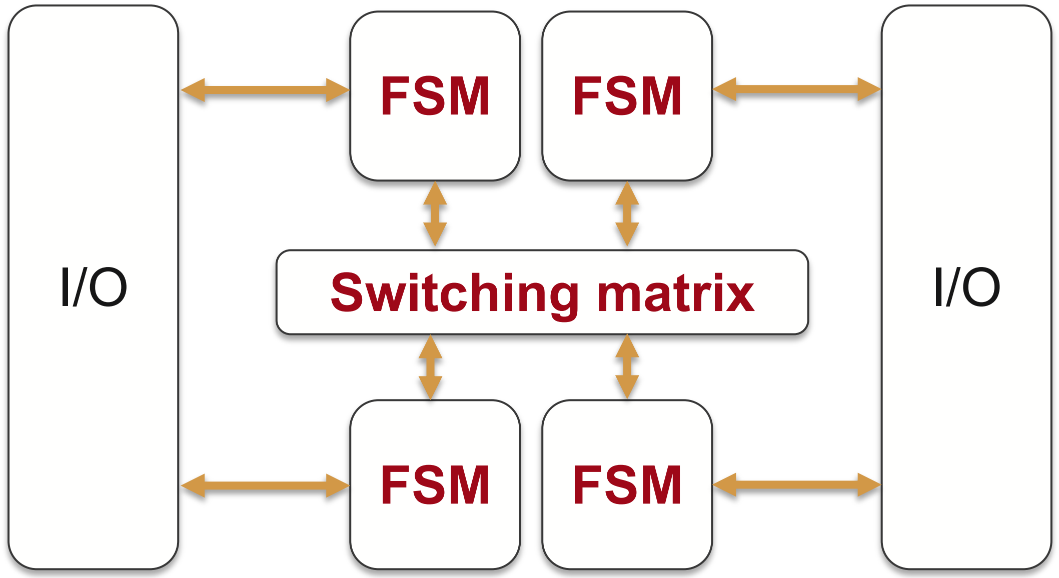


FSM: Finite State Machine

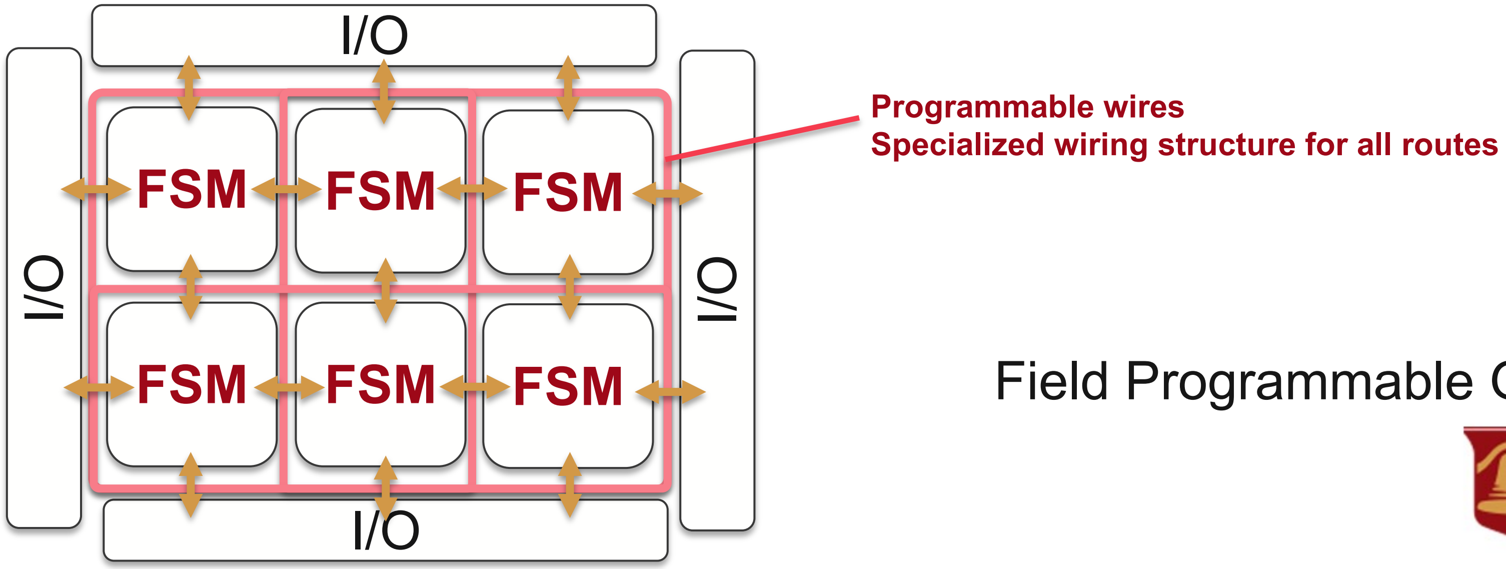
With the help of flip flops, now the system can store **state** variables.
The system can then output:

- Function of states (Moore machine)
- Function of sates and inputs (Mealy machine)

Let’s do even a higher level of abstraction:



Complex Programmable Logic Device - CPLD



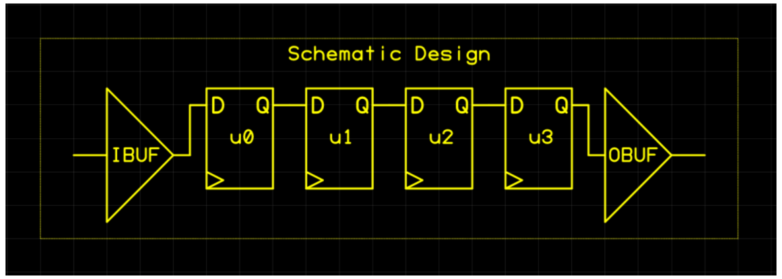
Field Programmable Gate Array - FPGA

Coe College

How programming has changed overtime

In early days (10-32 flip-flops):

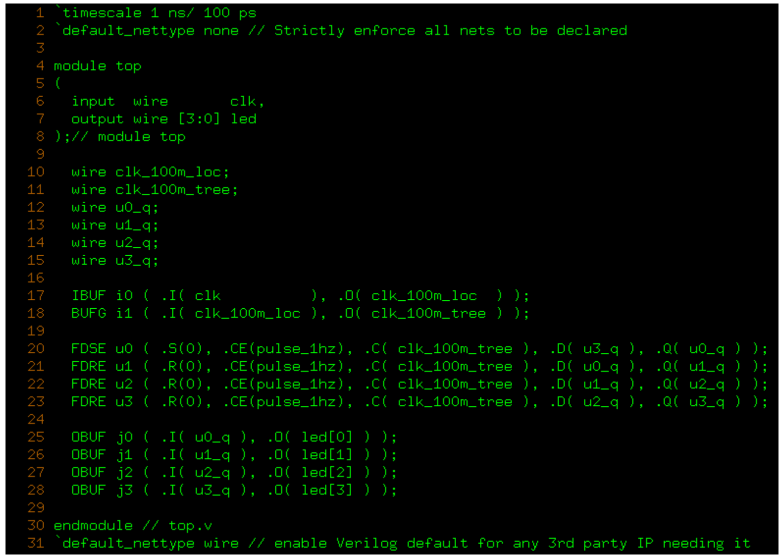
I draw this:



Software (EDA) will do this (a netlist):

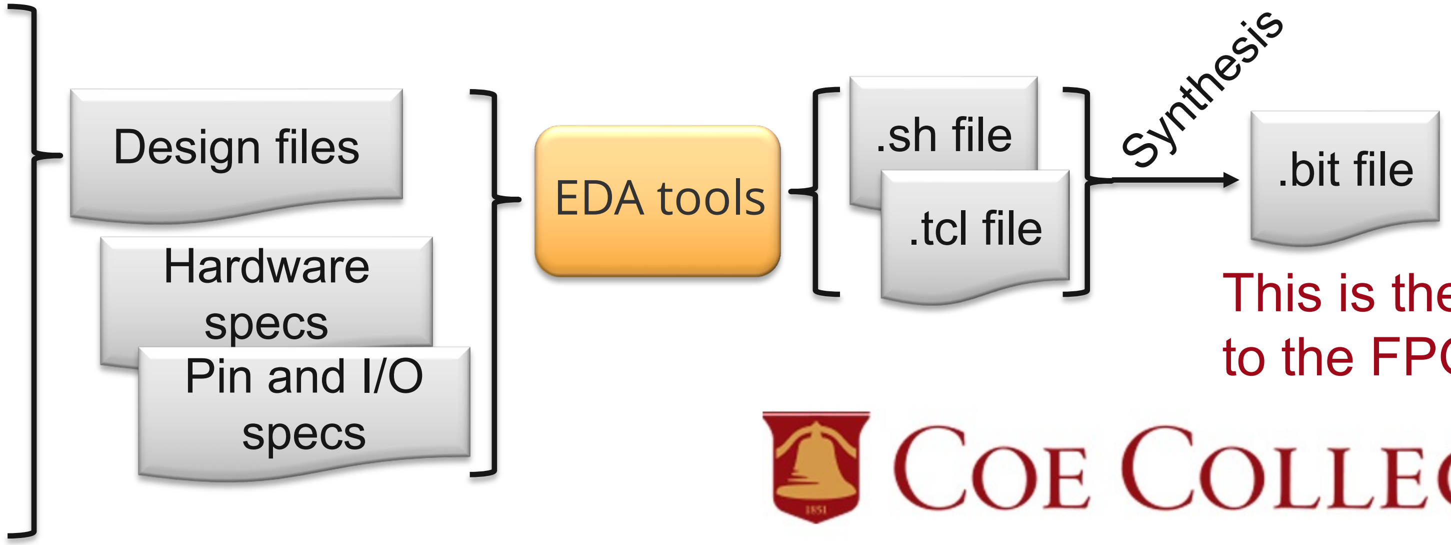
```
…
(cell top_shift_register (cellType GENERIC)
   (interface
   (port (identifier SI) (direction INPUT))
   (port (identifier CLK) (direction INPUT))
   (port (identifier RST) (direction INPUT))
   (port (identifier Q0) (direction OUTPUT))
   (port (identifier Q1) (direction OUTPUT))
   (port (identifier Q2) (direction OUTPUT))
   (port (identifier Q3) (direction OUTPUT))
  )
  (contents
   (instance i_dff0 (view netlist) (cell DFF) (library generic_gates))
   (instance i_dff1 (view netlist) (cell DFF) (library generic_gates))
   (instance i_dff2 (view netlist) (cell DFF) (library generic_gates))
   (instance i_dff3 (view netlist) (cell DFF) (library generic_gates))

…
```

Later I use Verilog:



```
`timescale 1 ns/ 100 ps
`default_nettype none // Strictly enforce all nets to be declared

module top
(
  input  wire       clk,
  output wire [3:0] led
);// module top

  wire clk_100m_loc;
  wire clk_100m_tree;
  wire u0_q;
  wire u1_q;
  wire u2_q;
  wire u3_q;

  IBUF i0 ( .I( clk          ), .O( clk_100m_loc  ) );
  BUFG i1 ( .I( clk_100m_loc ), .O( clk_100m_tree ) );

  FDSE u0 ( .S(0), .CE(pulse_1hz), .C( clk_100m_tree ), .D( u3_q ), .Q( u0_q ) );
  FDRE u1 ( .R(0), .CE(pulse_1hz), .C( clk_100m_tree ), .D( u0_q ), .Q( u1_q ) );
  FDRE u2 ( .R(0), .CE(pulse_1hz), .C( clk_100m_tree ), .D( u1_q ), .Q( u2_q ) );
  FDRE u3 ( .R(0), .CE(pulse_1hz), .C( clk_100m_tree ), .D( u2_q ), .Q( u3_q ) );

  OBUF j0 ( .I( u0_q ), .O( led[0] ) );
  OBUF j1 ( .I( u1_q ), .O( led[1] ) );
  OBUF j2 ( .I( u2_q ), .O( led[2] ) );
  OBUF j3 ( .I( u3_q ), .O( led[3] ) );

endmodule // top.v
`default_nettype wire // enable Verilog default for any 3rd party IP needing it
```

Then I use this design codes to “build”:

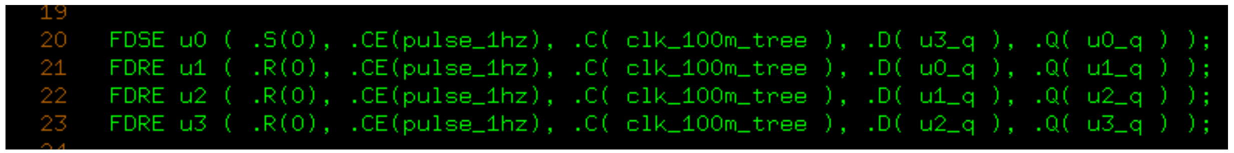


This is the file to the FPGA

COE COLLEGE

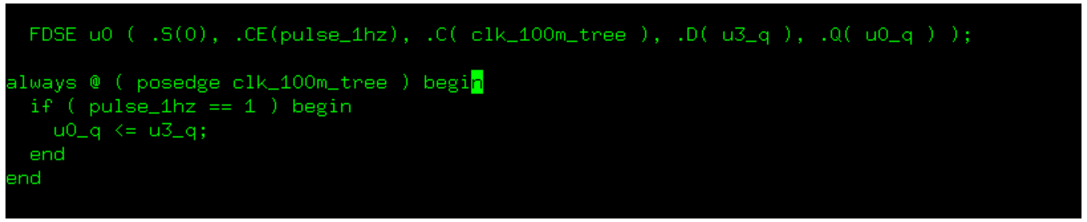
How programming has changed overtime

Verilog



```
FDSE u0 ( .S(0), .CE(pulse_1hz), .C( clk_100m_tree ), .D( u3_q ), .Q( u0_q ) );
FDRE u1 ( .R(0), .CE(pulse_1hz), .C( clk_100m_tree ), .D( u0_q ), .Q( u1_q ) );
FDRE u2 ( .R(0), .CE(pulse_1hz), .C( clk_100m_tree ), .D( u1_q ), .Q( u2_q ) );
FDRE u3 ( .R(0), .CE(pulse_1hz), .C( clk_100m_tree ), .D( u2_q ), .Q( u3_q ) );
```

With higher level language, the codes are now more readable:



```
 FDSE u0 ( .S(0), .CE(pulse_1hz), .C( clk_100m_tree ), .D( u3_q ), .Q( u0_q ) );

always @ ( posedge clk_100m_tree ) begin
  if ( pulse_1hz == 1 ) begin
    u0_q <= u3_q;
  end
end
```

Higher language options:

System Verilog (RTL Verilog)

VHDL

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  assign y = ~a;
endmodule
```

a[3:0] represents a 4-bit bus. The bits, from most significant to least significant, are a[3], a[2], a[1], and a[0]. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have named the bus a[4:1], in which case a[4] would have been the most significant. Or we could have used a[0:3], in which case the bits, from most significant to least significant, would be a[0], a[1], a[2], and a[3]. This is called *big-endian* order.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of inv is
begin
  y <= not a;
end;
```

VHDL uses STD_LOGIC_VECTOR to indicate busses of STD_LOGIC. STD_LOGIC_VECTOR(3 downto 0) represents a 4-bit bus. The bits, from most significant to least significant, are a(3), a(2), a(1), and a(0). This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have declared the bus to be STD_LOGIC_VECTOR(4 downto 1), in which case bit 4 would have been the most significant. Or we could have written STD_LOGIC_VECTOR(0 to 3), in which case the bits, from most significant to least significant, would be a(0), a(1), a(2), and a(3). This is called *big-endian* order.

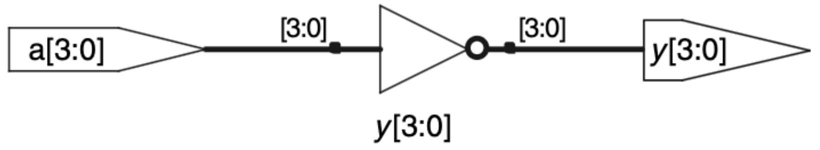
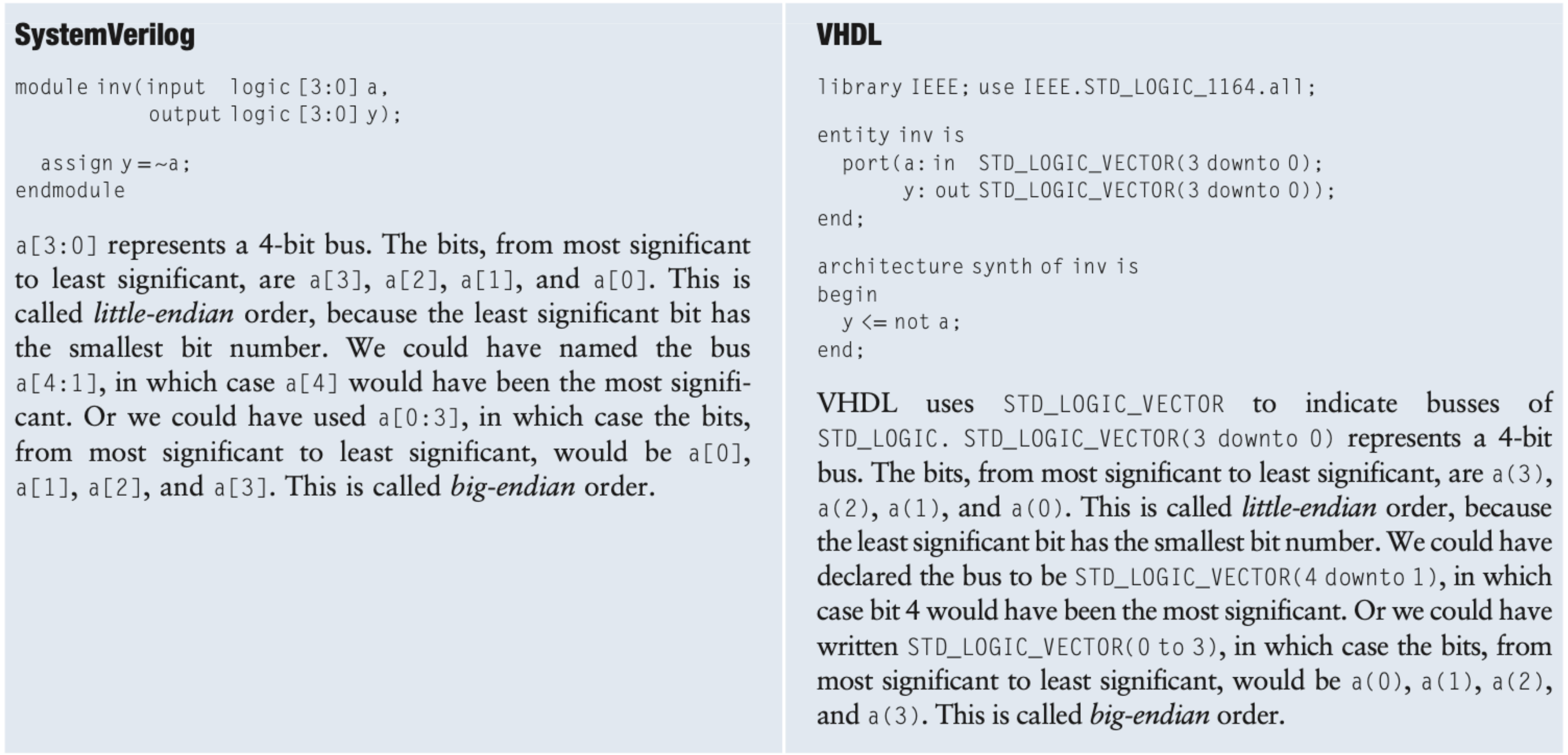


Figure 4.3 inv **synthesized circuit**

Higher language options:

System Verilog (RTL Verilog) VHDL



SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  assign y = ~a;
endmodule
```

a[3:0] represents a 4-bit bus. The bits, from most significant to least significant, are a[3], a[2], a[1], and a[0]. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have named the bus a[4:1], in which case a[4] would have been the most significant. Or we could have used a[0:3], in which case the bits, from most significant to least significant, would be a[0], a[1], a[2], and a[3]. This is called *big-endian* order.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of inv is
begin
  y <= not a;
end;
```

VHDL uses STD_LOGIC_VECTOR to indicate busses of STD_LOGIC. STD_LOGIC_VECTOR(3 downto 0) represents a 4-bit bus. The bits, from most significant to least significant, are a(3), a(2), a(1), and a(0). This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have declared the bus to be STD_LOGIC_VECTOR(4 downto 1), in which case bit 4 would have been the most significant. Or we could have written STD_LOGIC_VECTOR(0 to 3), in which case the bits, from most significant to least significant, would be a(0), a(1), a(2), and a(3). This is called *big-endian* order.

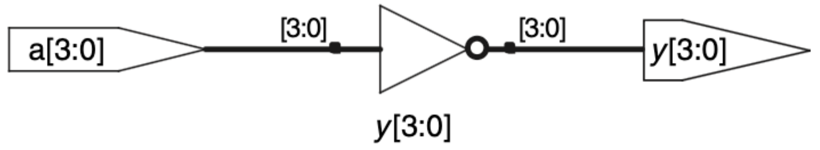


Figure 4.3 inv **synthesized circuit**

More HDL examples:

HDL Example 4.4 EIGHT-INPUT AND

SystemVerilog

```
module and8(input  logic [7:0] a,
            output logic       y);

  assign y = &a;

  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and8 is
  port(a: in  STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of and8 is
begin
  y <= and a;
  -- and a is much easier to write than
  -- y <= a(7) and a(6) and a(5) and a(4) and
  --      a(3) and a(2) and a(1) and a(0);
end;
```

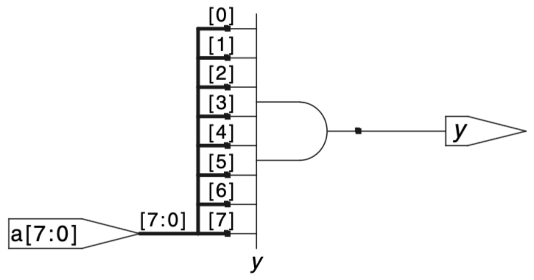
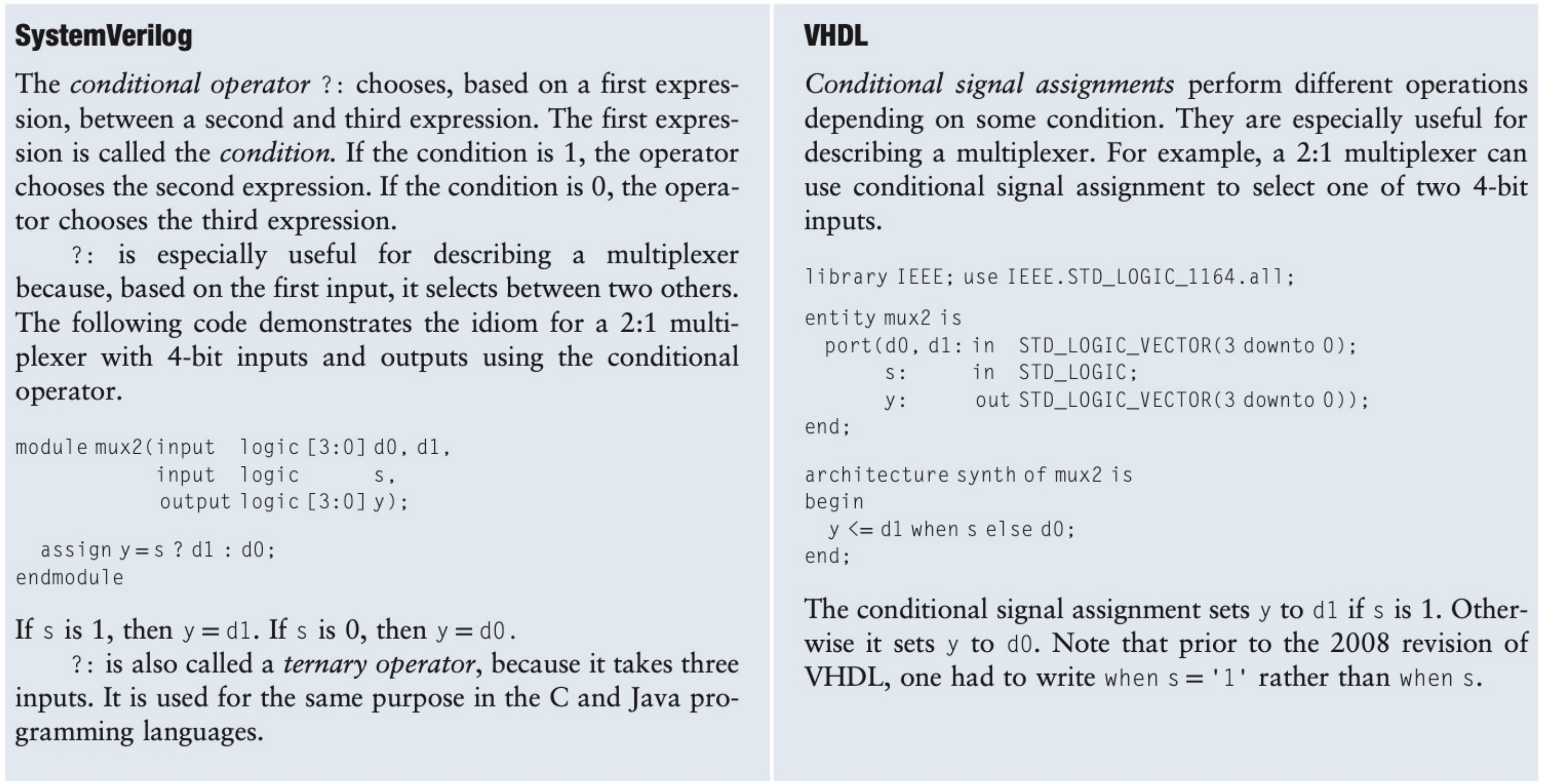


Figure 4.5 and8 **synthesized circuit**

More HDL examples:

HDL Example 4.5 2:1 MULTIPLEXER



SystemVerilog

The *conditional operator* ?: chooses, based on a first expression, between a second and third expression. The first expression is called the *condition*. If the condition is 1, the operator chooses the second expression. If the condition is 0, the operator chooses the third expression.

?: is especially useful for describing a multiplexer because, based on the first input, it selects between two others. The following code demonstrates the idiom for a 2:1 multiplexer with 4-bit inputs and outputs using the conditional operator.

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);

  assign y = s ? d1 : d0;
endmodule
```

If s is 1, then y = d1. If s is 0, then y = d0.

?: is also called a *ternary operator*, because it takes three inputs. It is used for the same purpose in the C and Java programming languages.

VHDL

Conditional signal assignments perform different operations depending on some condition. They are especially useful for describing a multiplexer. For example, a 2:1 multiplexer can use conditional signal assignment to select one of two 4-bit inputs.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d1 when s else d0;
end;
```

The conditional signal assignment sets y to d1 if s is 1. Otherwise it sets y to d0. Note that prior to the 2008 revision of VHDL, one had to write when s = '1' rather than when s.

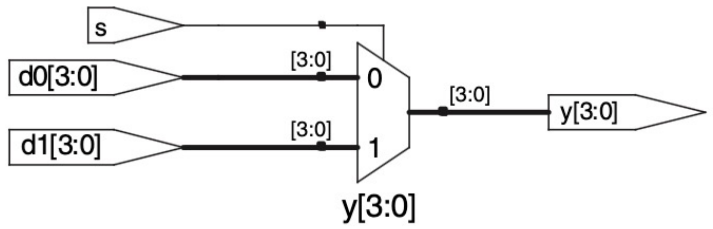
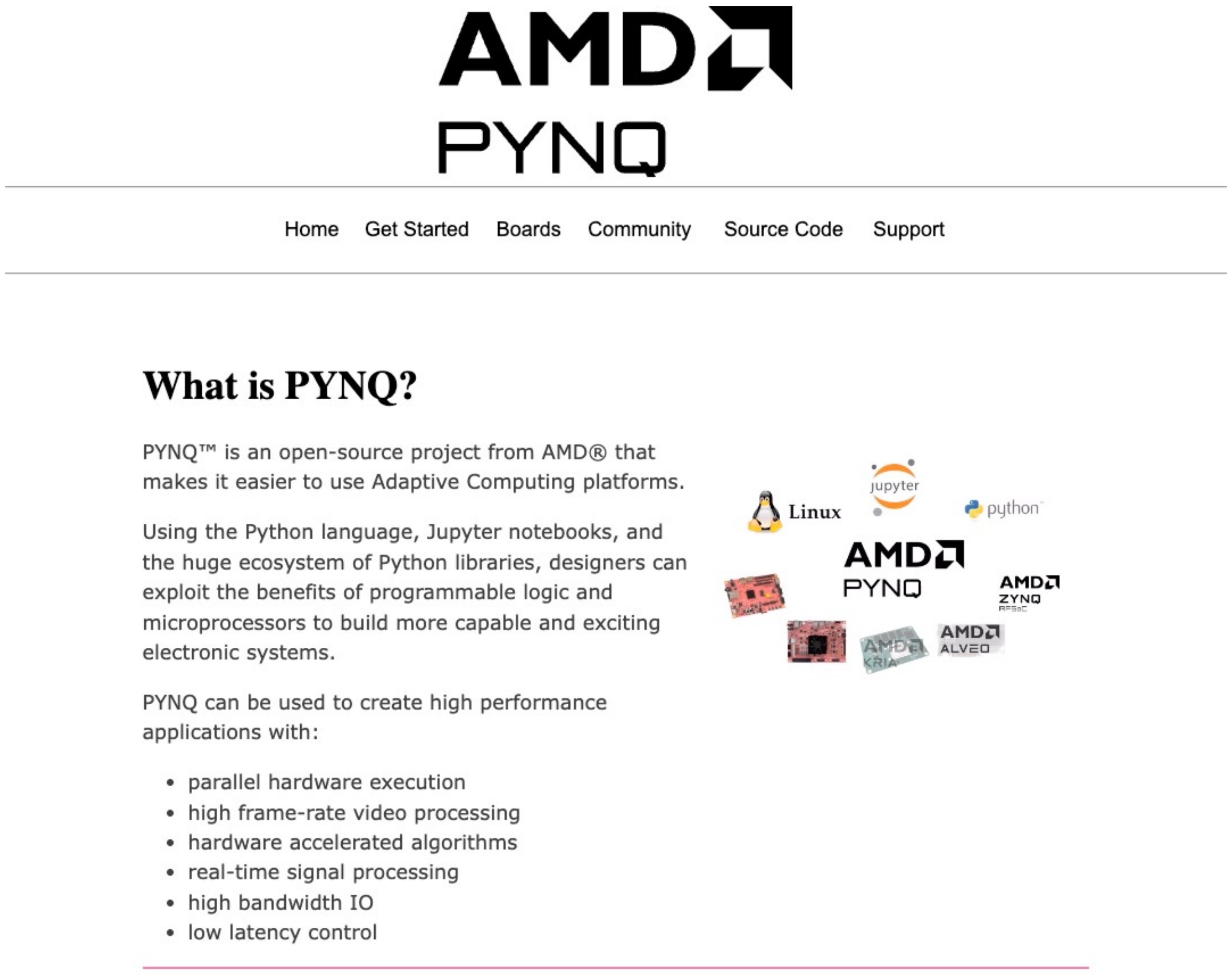


Figure 4.6 mux2 **synthesized circuit**

More modern FPGA dev:

https://www.pynq.io/



Home Get Started Boards Community Source Code Support

What is PYNQ?

PYNQ™ is an open-source project from AMD® that makes it easier to use Adaptive Computing platforms.

Using the Python language, Jupyter notebooks, and the huge ecosystem of Python libraries, designers can exploit the benefits of programmable logic and microprocessors to build more capable and exciting electronic systems.

PYNQ can be used to create high performance applications with:

- parallel hardware execution
- high frame-rate video processing
- hardware accelerated algorithms
- real-time signal processing
- high bandwidth IO
- low latency control

https://github.com/Xilinx/PYNQ_Workshop