Modeling Sequential Elements with Verilog

Prof. Chien-Nan Liu
TEL: 03-4227151 ext:34534
Email: jimmy@ee.ncu.edu.tw

Sequential Circuit

- Outputs are functions of inputs and present states of storage elements
- Two types of sequential circuits
  - Synchronous (preferred !!)
  - Asynchronous

Fig. 5-1 Block Diagram of Sequential Circuit
Synchronous vs. Asynchronous

- Avoid asynchronous and multi-cycle paths
- Register-based synchronous design is preferred
  - Accelerate synthesis and simulation
  - Ease static timing analysis
  - Use single (positive) edge triggered flip-flop
- Avoid to use latch as possible

Memory Elements

- Allow sequential logic design
- Latch — a level-sensitive memory element
  - SR latches
  - D latches
- Flip-Flop — an edge-triggered memory element
  - Master-slave flip-flop
  - Edge-triggered flip-flop
- RAM and ROM — a mass memory element
SR Latch

**Logic diagram**

- **S (Set)**
- **R (Reset)**

**Function table**

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q'</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 4-4. SR Latch with NOR Gates

Should be very careful for this case

**Logic Simulation of SR Latch Behavior**

---

Other SR Latches

**Logic diagram**

- **S and R are active-low**

**Function table**

<table>
<thead>
<tr>
<th>C</th>
<th>S</th>
<th>R</th>
<th>Q'</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Logic diagram**

- **C=0: disable all actions**

**Function table**

<table>
<thead>
<tr>
<th>C</th>
<th>S</th>
<th>R</th>
<th>Next state of Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>No change</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>No change</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Q = 0, Reset state</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Q = 1, set state</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Indeterminate</td>
</tr>
</tbody>
</table>

Fig. 5-4. SR Latch with NAND Gates

Fig. 5-5. SR Latch with Control Input
D Latch with Control

Function Table

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>Next state of Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>No change</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Q = 0; Reset state</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Q = 1; Set state</td>
</tr>
</tbody>
</table>

(a) Logic diagram

Implemented with Transmission Gates

Symbols for Latches

D with 1 Control

D with 1 Control

4-7

4-8
Latch Inference

- Incompletely specified wire in the synchronous section
- **D latch**
  
  ```verilog
  always @(enable or data)
  if (enable)
    q = data;
  ```

- **D latch with gated asynchronous data**
  
  ```verilog
  always @(enable or data or gate)
  if (enable)
    q = data & gate;
  ```

More Latches

- **D latch with gated “enable”**
  
  ```verilog
  always @(enable or d or gate)
  if (enable & gate)
    q = d;
  ```

- **D latch with asynchronous reset**
  
  ```verilog
  always @(reset or data or gate)
  if (reset)
    q = 1'b0;
  else if (enable)
    q = data;
  ```
Latch with Preset/Clear (1/2)

module LATCH_ASYNC_P_C (En1, Clear1, X1, En2, Preset2, X2, En3, Preset3, Clear3, X3, Y1, Y2, Y3);
input  En1, Clear1, X1, En2, Preset2, X2, En3, Preset3, Clear3, X3;
output  Y1, Y2, Y3;
reg  Y1, Y2, Y3;
always @(En1 or Clear1 or X1 or En2 or Preset2 or X2 or En3 or Preset3 or Clear3 or X3)
begin
  if (! Clear1)
    Y1 = 0;
  else if (En1)
    Y1 = X1;
end
endmodule

Latch with Preset/Clear (2/2)

if (Preset2)
  Y2 = 1;
else if (En2)
  Y2 = X2;
if (Clear3)
  Y3 = 0;
else if (Preset3)
  Y3 = 1;
else if (En3)
  Y3 = X3;
end
endmodule
Latch vs. Flip-Flop

- **Latch**:
  - Change stored value under specific status of the control signals
  - Transparent for input signals when control signal is “on”
  - May cause combinational feedback loop and extra changes at the output

- **Flip-Flop**
  - Can only change stored value by a momentary switch in value of the control signals
  - Cannot “see” the change of its output in the same clock pulse
  - Encounter fewer problems than using latches

---

Master-Slave SR Flip-Flop

- **C**: clock signal (positive pulse)
- **SR latch with control**

![Logic Simulation of a Master-Slave Flip-Flop](image)
Master-Slave J K Flip-Flop

Solve the undefined problem in SR FF

(b)

Symbols for Master-Slave F/F
Problems of Master-Slave F/F

- Master-slave flip-flops are also referred as **pulse-triggered** flip-flops
- May cause errors when the delay of combinational feedback path is too long

To solve:
- Ensure the delay of combinational block is short enough
- Use **edge-triggered** flip-flops instead

Positive-Edge-Triggered D F/F

- The master latch is a D latch
  - Transparent to input during C=0 without other control signals
  - The input value of the slave latch is decided just at the end of the period "C=0"
  - The delayed changes can still be stored if they can arrive before C has a 0->1 transition
- Solve the problems of master-slave flip-flops
Positive-Edge-Triggered D F/F

- If only SR latches are available, three latches are required
- Two latches are used for locking the two inputs (CLK & D)
- The final latch provides the output of the flip-flop

Positive-Edge-Triggered JK F/F

\[ D = JQ' + K'Q \]

(a) Circuit diagram
(b) Graphic symbol

Fig. 5-10 D-Type Positive-Edge-Triggered Flip-Flop

Fig. 5-12 JK Flip-Flop
Symbols for Edge-Triggered FF

Positive-Edge-Triggered T F/F

\[ D = T \oplus Q = TQ' + T'Q \]
Characteristic Tables

Define the logical properties in tabular form

<table>
<thead>
<tr>
<th>TABLE 4-1</th>
<th>Flip-Flop Characteristic Tables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(a) JK Flip-Flop</strong></td>
<td><strong>(b) SR Flip-Flop</strong></td>
</tr>
<tr>
<td>J</td>
<td>K</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>(c) D Flip-Flop</strong></th>
<th><strong>(d) T Flip-Flop</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Q (t + 1)</td>
</tr>
<tr>
<td>---</td>
<td>-----------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Flip-Flop Inference

Wire (port) assigned in the synchronous section

module FF_PN (Clock, X1, X2, Y1, Y2);
    input Clock;
    input X1, X2;
    output Y1, Y2;
    reg Y1, Y2;
    always @(posedge Clock)
        Y1 = X1;
    always @(negedge Clock)
        Y2 = X2;
endmodule

4-23

4-24
Direct Inputs

- Directly set or reset a flip-flop immediately
  - Independent of clock signal
- Often called *direct set (preset)* and *directly reset (clear)*
- Also called *asynchronous set/reset*

![Graphic symbols and function table](image)

Fig. 4-16 JK Flip-Flop with Direct Set and Reset

---

D F/F with Asynchronous Reset

![Circuit diagram and graphic symbol](image)

![Function table](image)

Fig. 4-26 D Flip-Flop with Asynchronous Reset
Reset (1/2)

- Synchronous reset
  - Easy to synthesize
  - Requires a free-running clock, especially at power-up phase
  - Hard to deal with tri-state bus initialization at power on

Reset (2/2)

- Asynchronous reset
  - Do not require a free-running clock
  - Hard to implement the huge reset network
    - Clock tree synthesis (CTS) technique may also required for the reset signal
  - Synchronous de-assertion problem
  - Make STA and cycle-based simulation more difficult
- Asynchronous reset is more popular so far
Modeling Reset

Synchronous reset

always@(posedge clk)
if (!reset) q=1'b0;
else q=data;

Asynchronous reset

always@(posedge clk or negedge reset)
if (!reset) q=1'b0;
else q=data;

Flip-Flop with Set/Reset (1/3)

- Asynchronous / synchronous reset / enable

module FFS (Clock, SReset, ASReset, En, Data1, Data2, Y1, Y2, Y3, Y4, Y5, Y6);
  input  Clock, SReset, ASReset, En, Data1, Data2;
  output  Y1, Y2, Y3, Y4, Y5, Y6;
  reg  Y1, Y2, Y3, Y4, Y5, Y6;
  always @(posedge Clock)
    begin
      if (! SReset) // Synchronous reset
        Y1 = 0;
      else
        Y1 = Data1 | Data2;
    end
Flip-Flop with Set/Reset (2/3)

// Negative active asynchronous reset
always @(posedge Clock or negedge ASReset)
  if (! ASReset)
    Y2 = 0;
  else
    Y2 = Data1 & Data2;

// One synchronous & one asynchronous reset
always @(posedge Clock or negedge ASReset)
  if (! ASReset)
    Y3 = 0;
  else if (SReset)
    Y3 = 0;
  else
    Y3 = Data1 | Data2;

endmodule

Flip-Flop with Set/Reset (3/3)

// Single enable
always @(posedge Clock)
  if (En)
    Y4 = Data1 & Data2;

// Synchronous reset and enable
always @(posedge Clock)
  if (SReset)
    Y5 = 0;
  else if (En)
    Y5 = Data1 | Data2;
endmodule
Registers and Counters

- **Register:**
  - A set of flip-flops, possibly with added combinational gates, that perform data-processing tasks
  - Store and manipulate information in a digital system

- **Counter:**
  - A register that goes through a predetermined sequence of states
    - A special type of register
  - Employed in circuits to sequence and control operations

---

The Simplest Register

- Consist of only flip-flops
- Triggered by common clock input
- The Clear input goes to the R (reset) input of all flip-flops
  - Clear = 0 → all flip-flops are reset *asynchronously*
- The Clear input is useful for cleaning the registers to all 0’s prior to its clocked operation
  - Must maintain at logic 1 during normal operations
Register with Parallel Load

- When Load = 1, all the bits of data inputs are transferred into the registers
  - Parallel loaded
- When Load = 0, the outputs of the flip-flops are connected to their respective inputs
  - Keep no change

control data loading at the input side

Fig. 6-2. 4-Bit Register with Parallel Load

HDL Modeling for Registers

```verilog
module Reg4 (CLK, CLR_N, Load, D, Q);
    input CLK, CLR_N, Load;
    input [3:0] D;
    output [3:0] Q;
    reg [3:0] Q;
    always @(posedge CLK) begin
        if (CLR_N == 0)
            Q <= 0;
        else if (Load == 1)
            Q <= D;
        end
    endmodule
```

4-35 4-36
The Simplest Shift Register

HDL Modeling for Shift Reg.

- **Blocking assignment**
  ```verilog
  assign SO = D;
  always @(posedge CLK) begin
    if (reset) begin
      A = 0;  B = 0;  C = 0;  D = 0;
      end
    else if (shift) begin
      D = C;
      C = B;
      B = A;
      A = SI;
    end
  end
  ```

- **Non-blocking assignment**
  ```verilog
  assign SO = D;
  always @(posedge CLK) begin
    if (reset) begin
      A <= 0;  B <= 0;  C <= 0;  D <= 0;
    end
    else if (shift) begin
      A <= SI;
      C <= B;
      D <= C;
      B <= A;
    end
  end
  ```

**Fig. 6-3** 4-Bit Shift Register

**SRG 4**

Order dependent: can be any order
Serial Addition

- The bits of two binary numbers are added one pair at a time through a single full adder (FA) circuit.
- Initialization:
  - A = augend; B = addend
  - Carry flip-flop is cleared to 0
- For each clock pulse:
  - A new sum bit is transferred to A
  - A new carry is transferred to Q
  - Both registers are shifted right
- To add three or more numbers:
  - Shift in the next number from the serial input while B is shifted to the FA
  - A will accumulate their sum

Serial v.s. Parallel

- Serial adders:
  - Use shift registers
  - A sequential circuit
  - Require only one FA and a carry flip-flop
  - Slower but require less equipment
- Parallel adders:
  - Use registers with parallel load for sum
  - Basically a pure combinational circuit
  - $n$ FAs are required
  - Faster
Shift Register with Parallel Load

always @(posedge CLK) begin
    if (Reset)
        Q <= 4'b0;
    else if (Shift)
        for (i=0; i<4; i=i+1)
            Q[i+1] <= Q[i];
    else if (Load)
        Q <= D;
end

should be careful about the priority

Modeling Shift Reg. with Load

TABLE 5-2
Function Table for the Register of Figure 5-6

<table>
<thead>
<tr>
<th>Shift</th>
<th>Load</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No change</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Load parallel data</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>Shift down from Q0 to Q3</td>
</tr>
</tbody>
</table>

Table 5-2: Function Table for the Register of Figure 5-6
Universal Shift Register

clear all registers to zero

<table>
<thead>
<tr>
<th>Mode Control</th>
<th>Register Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1  S0</td>
<td></td>
</tr>
<tr>
<td>0  0</td>
<td>No change</td>
</tr>
<tr>
<td>0  1</td>
<td>Shift right</td>
</tr>
<tr>
<td>1  0</td>
<td>Shift left</td>
</tr>
<tr>
<td>1  1</td>
<td>Parallel load</td>
</tr>
</tbody>
</table>

Counters

- A register that goes through a prescribed sequence of states
- Ripple counter
  - The flip-flop output transition serves as a source for triggering other flip-flops
  - Hardware is much simpler
  - Unreliable and delay dependent due to the asynchronous behaviors
- Synchronous counter
  - The common clock pulse triggers all flip-flops simultaneously
  - Hardware may be more complex but more reliable
Binary Ripple Counter

Binary Count Sequence

A3  A2  A1  A0

| 0  | 0  | 0  | 0 |
| 0  | 0  | 1  | 1 |
| 0  | 0  | 1  | 0 |
| 1  | 0  | 1  | 1 |
| 0  | 1  | 1  | 0 |
| 1  | 1  | 0  | 0 |

the output transition triggers the next flip-flop

Binary Synchronous Counter

\[ J=0, K=0: \] no change
\[ J=1, K=1: \] complement

complemented when all the lower bits are "1" (checked by a AND chain)

triggered by positive clock edge

0 \rightarrow 1 \rightarrow \ldots \rightarrow 15
\rightarrow 0 \rightarrow 1 \rightarrow \ldots
### Up-Down Binary Counter

**Up:** 0000 → 0001 → 0010 → …  
**Down:** 1111 → 1110 → 1101 → …

> take the complemented values for count-down calculation

### Binary Counter with Parallel Load

<table>
<thead>
<tr>
<th>Clear</th>
<th>CLK</th>
<th>Load</th>
<th>Count</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Clear to 0</td>
</tr>
<tr>
<td>1</td>
<td>↑</td>
<td>1</td>
<td>X</td>
<td>Load inputs</td>
</tr>
<tr>
<td>1</td>
<td>↑</td>
<td>0</td>
<td>1</td>
<td>Count next</td>
</tr>
<tr>
<td>1</td>
<td>↑</td>
<td>0</td>
<td>0</td>
<td>No change</td>
</tr>
</tbody>
</table>

- **Counted by:** AND gates  
- **Merged by:** OR gates  
- **Asynchronous input:** Set when count = 1111
Modeling Sync. Counters

module counter (clk, reset, load, in, count) ;  
input           clk, reset, load ;  
input   [7:0] in ;  
output [7:0] count ;  
reg    [7:0] count ;  
always @(posedge clk) begin  
    if (reset) count = 0 ;  
    else if (load) count = in ;  
    else if (count == 255) count = 0 ;  
    else count = count + 1 ;  
end  
endmodule

BCD Counter

- Divide-by-N counter:  
  - Go through a repeated sequence of N states  
- BCD counter: only 0 to 9

Fig. 6-9 State Diagram of a Decimal BCD-Counter
Modeling BCD Counters

module BCDcounter (clk, reset, count);
input clk, reset;
output [3:0] count;
reg [3:0] count;

always @(posedge clk) begin
  if (reset) count = 0;
  else if (count == 9) count = 0;
  else count = count + 1;
end
endmodule

Arbitrary Count Sequence

Design it as a FSM

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>(A_1)</th>
<th>(A_0)</th>
<th>(B_1)</th>
<th>(B_0)</th>
<th>(C_1)</th>
<th>(C_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>0 1 1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0 1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1 1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Mass Memory Elements

- Memory is a collection of binary cells together with associated circuits needed to transfer information to or from any desired location.

- Two primary categories of memory:
  - Random access memory (RAM)
  - Read only memory (ROM)

RAM Cell Array
Write and Read Operations

- **Write to RAM**
  - Apply the binary address of the desired word to the **address lines**
  - Apply the data bits that must be stored in memory to the **data input lines**
  - Activate the **write control**

- **Read from RAM**
  - Apply the binary address of the desired word to the **address lines**
  - Activate the **read control**

Timing Waveforms

- **CPU clock** = 50 MHz
  - cycle time = 20 ns
- **Memory access time** = 50 ns
  - The time required to complete a read or write operation
- The control signals must stay active for at least 50 ns
  - 3 CPU cycles are required
Coincident Decoding

- Address decoders are often divided into two parts
  - A two-dimensional scheme
- The total number of gates in decoders can be reduced
- Can arrange the memory cells to a square shape
- EX: 10-bit address
  404 = 0110010100
  X = 01100
  Y = 10100

Address Multiplexing

- Memory address lines often occupy too much I/O pads
  - 64K = 16 lines
  - 256M = 28 lines
- Share the address lines of X and Y domains
  - Reduce the number of lines to a half
  - An extra register is required for both domain to store the address
SRAM vs. DRAM

- Static RAM:
  - Use internal latch to store the binary info.
  - Stored information remains valid as long as power is on.
  - Shorter read and write cycles
  - Larger cell area and power consumption

- Dynamic RAM:
  - Use a capacitor to store the binary info.
  - Need periodically refreshing to hold the stored info.
  - Longer read and write cycles
  - Smaller cell area and power consumption

![Static RAM Cell](image1.png)  ![Dynamic RAM Cell](image2.png)

Read Only Memory (1/2)

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A₀ A₁ A₂ A₃ A₄ A₅ A₆ A₇</td>
</tr>
<tr>
<td>l₀ l₁ l₂ l₃</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0</td>
<td>1 0 1 1 0 1 1 0</td>
</tr>
<tr>
<td>0 0 0 0 1 0</td>
<td>0 0 0 1 1 1 0 1</td>
</tr>
<tr>
<td>0 0 0 1 1 1</td>
<td>1 1 0 0 0 1 0 1</td>
</tr>
<tr>
<td>0 0 1 1 1 1</td>
<td>1 0 1 1 0 0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0 0 0</td>
<td>0 0 0 0 1 0 0 1</td>
</tr>
<tr>
<td>1 1 1 0 1 0</td>
<td>1 1 1 0 0 0 0 1</td>
</tr>
<tr>
<td>1 1 1 1 0 0</td>
<td>0 1 0 0 1 0 1 0</td>
</tr>
<tr>
<td>1 1 1 1 1 1</td>
<td>0 0 1 1 0 0 1 1</td>
</tr>
</tbody>
</table>

Table 7-3
Types of ROMs

- Mask programming
  - Program the ROM in the semiconductor factory
  - Economic for large quantity of the same ROM
- Programmable ROM (PROM)
  - Contain all fuses at the factory
  - Program the ROM by burning out the undesired fuses (irreversible process)
- Erasable PROM (EPROM)
  - Can be restructured to the initial state under a special ultraviolet light for a given period of time
- Electrically erasable PROM (EEPROM or E²PROM)
  - Like the EPROM except being erased with electrical signals
Programmable Logic Devices

- ROM provides full decoding of variables
  - Waste hardware if the functions are given
- For known combinational functions, Programmable Logic Devices (PLD) are often used
  - Programmable read-only memory (PROM)
  - Programmable array logic (PAL)
  - Programmable logic array (PLA)
- For sequential functions, we can use
  - Sequential programmable logic device (SPLD)
  - Complex programmable logic device (CPLD)
  - Field programmable gate array (FPGA) **most popular**

Configurations of Three PLDs

(a) Programmable read-only memory (PROM)

(b) Programmable array logic (PAL)

(c) Programmable logic array (PLA)

Fig. 7.13 Basic Configuration of Three PLDs
PLD Examples

Modeling ROM and combinational PLDs
- Similar to modeling a combinational code converter

Modeling RAM
- Use memory array declaration in Verilog
  
  \texttt{ex: reg [3:0] \texttt{MY\_MEM} [0:63]; // 64 4-bit registers}
  
  \texttt{MY\_MEM[0] \leftarrow 4-bit variable}

- Can load memory by using a system task
  
  \texttt{ex: \$readmemb(“mem\_content”, MY\_MEM, 0, 63);}

- If synthesized, only SRAM (array of registers) will be generated
  - Use memory compiler or pre-designed layout instead