VLSI DESIGN

Figure 11.10 System-level hierarchy

TOP-DOWN DESIGN

- "SYSTEM-ON-A-CHIP" DESIGNS ARE TYPICALLY DONE FROM THE TOP DOWN TO THE CIRCUIT LEVEL.
- VERILOG OR VHDL CODE IS WRITTEN, FOLLOWED BY SYNTHESIS TO GENERATE A GATE-LEVEL NETLIST, FOLLOWED BY PLACE & ROUTE OF THE LAYOUT.
- TIMING IS CHECKED BEFORE P&R, AND THEN AFTER P&R INCLUDING PARASITIC CAPS EXTRACTED FROM THE LAYOUT.
  - PASS → Go to NEXT STEP
  - FAIL → ITERATE!
- "ECO" ROUTES (ENGINEERING CHANGE ORDER) CAN BE RUN ON EXISTING LAYOUTS TO MAKE MINOR CHANGES WITHOUT HAVING TO START FROM SCRATCH.

Figure 10.1 Example of a VLSI design flow

- CUSTOM CELLS ARE ONLY DESIGNED IF NEEDED (E.G., TO PUSH THE SPEED LIMITS OF THE PROCESS).
- STANDARD CELL LIBRARIES ARE USED WHENEVER POSSIBLE.
  → FASTER TO DESIGN BIGGER BLOCKS WHEN THE SMALLER BUILDING BLOCKS ALREADY EXIST (E.G., NO NEW LAYOUTS).
  → ALLOWS LIBRARY CELLS TO ALREADY BE FULLY CHARACTERIZED AND (OFTEN) TESTED IN SILICON TO BE GOOD.
 USING NAND/NORS TO GATE SIGNALS

NAND:

\[
\begin{align*}
A & \quad \text{y} \\
\text{ENABLE} & \quad \text{ENABLE}
\end{align*}
\]

- ENABLE = ACTIVE HIGH
- OUTPUT = HIGH WHEN GATE IS DISABLED

\[
\begin{align*}
\text{IF: ENABLE} & = 1 \\
\text{THEN: } y & = \overline{A} \quad \text{(LOOKS LIKE AN INVERTER!)} \\
\text{ELSE: } y & = 1
\end{align*}
\]

NOR:

\[
\begin{align*}
A & \quad \text{y} \\
\text{ENABLE} & \quad \text{ENABLE}
\end{align*}
\]

- ENABLE = ACTIVE LOW
- OUTPUT = LOW WHEN GATE IS DISABLED

\[
\begin{align*}
\text{IF: ENABLE} & = \emptyset \\
\text{THEN: } y & = \overline{A} \quad \text{(LOOKS LIKE AN INVERTER!)} \\
\text{ELSE: } y & = \emptyset
\end{align*}
\]

CAN EXTEND TO MORE INPUTS:

\[
\begin{align*}
\text{ENABLE} & \quad \text{y} \\
A & \quad B \quad C \quad \ldots
\end{align*}
\]

\[
\begin{align*}
\text{ENABLE} & \quad \text{y} \\
A & \quad B \quad C \quad \ldots
\end{align*}
\]

★ KEY DIFFERENCES:

1) NAND ENABLE IS ACTIVE HIGH
   NOR ENABLE IS ACTIVE LOW
2) NAND OUTPUT = 1 WHEN DISABLED
   NOR OUTPUT = \emptyset WHEN DISABLED
2-INPUT MUX

(a) TG circuit          (b) Pass transistors

Figure 11.2 Multiplexer using switch logic

Figure 11.1 Gate-level NAND 2:1 multiplexer

module simple_mux (mux_out, p0, p1, select);
input p0, p1 ;
input select;
output mux_out ;
always @ (select )
case ( select )
1'0 : mux_out = p0 ;
1'1 : mux_out = p1 ;
endcase
endmodule

- T-GATE MUX OFTEN USED IN ANALOG, Seldom used in digital VLSI DUE TO TOOL ISSUES
- CAN ALSO BUILD OUT OF NOR GATES, OR A COMPLEX CMOS GATE

- NOTE THAT THE VERILOG CODE HERE SHOWS ONLY THE FUNCTION, NOT HOW THE MUX IS BUILT
Figure 11.3 A 4:1 MUX using instanced 2:1 devices

```verilog
class bigger_mux (out_4, p0, p1, p2, p3, s0, s1);
input p0, p1, p2, p3;
input s0, s1;
output out_4;
assign out_4 = s1 ? (s0 ? p3 : p2) : (s0 ? p1 : p0);
endmodule
```

logic; this is equivalent to the SOP expression

\[ f = p_0 \cdot \overline{s_1} \cdot \overline{s_3} + p_1 \cdot \overline{s_1} \cdot s_0 + p_2 \cdot s_1 \cdot \overline{s_0} + p_3 \cdot s_1 \cdot s_0 \]  \hspace{1cm} (11.5)

obtained from applying basic logic.

---

Figure 11.4 Gate-level 4:1 MUX

```verilog
class gate_mux_4 (out_gate, p0, p1, p2, p3, s0, s1);
input p0, p1, p2, p3;
input s0, s1;
wire w1, w2, w3, w4;
output out_gate;
nand (w1, p_0, \overline{s_1}, \overline{s_0})
(w2, p_1, \overline{s_1}, s_0)
(w3, p_2, s_1, \overline{s_0})
(w4, p_3, s_1, s_0)
(out_gate, w1, w2, w3, w4);
endmodule
```

→ THIS APPROACH USES THE SELECT INPUTS AS ENABLES TO MULTIPLE NAND GATES

→ FAST, BUT SELECT INPUTS NEED BUFFERS TO DRIVE THE INPUTS OF ALL 4 NANDS

→ NOTE THAT THIS VERILOG CODE SHOWS EXACTLY HOW THE DESIGNER WANTS THIS BLOCK IMPLEMENTED
*Verilog here is very explicit! (unusual!)*

- This approach is useful for full-custom designs (e.g., to increase speed) but would not be part of a typical digital cell library.

- Full T-gate version is big and hard to route all the signals!
**Multiple Bit (Buss) MUXes**

Architectural specifications treat n-bit words in about the same manner as single-bit entities. Suppose that we have two 8-bit words

\[
\begin{align*}
    a &= a_7a_6a_5a_4a_3a_2a_1a_0 \\
    b &= b_7b_6b_5b_4b_3b_2b_1b_0
\end{align*}
\]  

(11.6)

that we want to use as inputs to a 2:1 MUX. The output

\[
f = f_7f_6f_5f_4f_3f_2f_1f_0
\]  

(11.7)

is determined by the select bit \( s \) such that

\[
f_i = a_i \bar{s} + b_i s
\]  

(11.8)

for \( i = 0, ..., 7 \). This of course implies that we should use 8 identical 2:1 MUXes that are all controlled by the same select bit \( s \).

---

![Symbol and Bit-level Realization](image)

**Figure 11.8** A vector 2:1 MUX

---

```
module mux_2-1_8b (f, a, b, s);
    input [7:0] a, b;
    input s;
    output [7:0] f;
    assign f = s ? b : a;
endmodule
```

- Verilog is very high-level here; no details
- *Note Buss notation used in Fig. 11.8 (a)

---

![Typical Layout Floorplan](image)

**Figure 11.9** Single-bit cell tiling for an 8-bit 2:1 MUX
**Binary Decoders** *(Active High)*

- Used, for example, to select which multiplexer to turn on

\[
\begin{align*}
    d_0 &= \overline{s_1} \cdot \overline{s_0} = s_1 + s_0 \\
    d_1 &= \overline{s_1} \cdot s_0 = s_1 + s_0 \\
    d_2 &= s_1 \cdot \overline{s_0} = s_1 + s_0 \\
    d_3 &= s_1 \cdot s_0 = s_1 + s_0
\end{align*}
\]

\{ Logic equations for each output bit \}

(11.8)

A straightforward NOR-gate implementation is shown in Figure 11.11(b). This gives the basis for the structural description:

```verilog
module decode_4 (d0, d1, d2, d3, s0, s1);
    input s0, s1;
    output d0, d1, d2, d3;
    nor (d3, ~s0, ~s1),
        (d2, ~s0, s1),
        (d1, s0, ~s1),
        (d0, s0, s1);
endmodule
```

where we have absorbed the NOT drivers into the notation using the `nor` operator.

An equivalent architectural description using `case` keywords can be written as:

```verilog
module dec_4 (d0, d1, d2, d3, sel);
    input [1:0] sel;
    output d0, d1, d2, d3;
    case (sel)
        0 : d0 = 1, d1 = 0, d2 = 0, d3 = 0;
        1 : d0 = 0, d1 = 1, d2 = 0, d3 = 0;
        2 : d0 = 0, d1 = 0, d2 = 1, d3 = 0;
        3 : d0 = 0, d1 = 0, d2 = 0, d3 = 1;
    endcase
endmodule
```

- **Verilog code here shows how to implement function with NOR gates.**

- Much more general, this leaves the details up to the synthesis tool and faster to design, but may be bigger, slower, etc., etc.

- **This approach uses the select inputs as enables to multiple NOR gates.**

![Figure 11.11 An active-high 2/4 decoder](image)

(a) Symbol and table

(b) NOR2 implementation
**Binary Decoders (cont.)**

*(Active Low)*

```verilog
module dec_lo (d0, d1, d2, d3, s0, s1);
  input s0, s1;
  output d0, d1, d2, d3;
  nand (d0, ~s0, ~s1),
       (d1, ~s0, s1),
       (d2, s0, ~s1),
       (d3, s0, s1);
endmodule
```

← Code explicitly says to use NAND gates

![Symbol and table](a) Symbol and table

![NAND2 implementation](b) NAND2 implementation

**Figure 11.12** Active low 2/4 decoder

- **NOR gate approach gives mostly 1s as outputs, with the selected output pulled low.**
  → **Exact opposite of NAND approach!**

- **NANDs tend to be smaller & faster than NORs (series NMOS vs series PMOS)**
**EQUALITY DETECTORS**

Figure 11.13 A 4-bit equality detector

This uses the equality (XNOR) relation:

\[ a_i \oplus b_i = 1 \]  (11.10)

iff \( a_i = b_i \) as a means to compare the inputs. If every XNOR produces a 1, then the output AND gate gives \( \text{Equal} = 1 \); otherwise, \( \text{Equal} = 0 \).

A Verilog listing for the operation is:

```
module equality (Equal, a, b);
  input [3:0] a, b;
  output Equal;
  always @ (a or b)
  begin
    if (a == b)
      Equal = 1;
    else
      Equal = 0;
  end
endmodule
```

*RECALL THAT!*

\[ A \oplus B = 1 \text{ only if } A = B \]

Very general architectural level code (no details)

Figure 11.14 8-bit equality detector

Can also extend to more bits
### Magnitude Comparators

<table>
<thead>
<tr>
<th>Condition</th>
<th>GT</th>
<th>LT</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &gt; b</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a = b</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 11.16** Comparator output summary

**Figure 11.15** 4-bit magnitude comparator logic

```verbatim
class module comp_4 (GT, LT, a, b);
input [3:0] a, b;
output GT, LT;
always @(a or b)
begin
  if (a > b)
    GT = 1, LT = 0;
  else if (a < b)
    GT = 0, LT = 1;
  else
    GT = 0, LT = 0;
end
endmodule
```

The high-level description masks the internal structure completely, making it appropriate for architectural simulations. However, the logic and circuit implementations can be quite complicated.

**Figure 11.17** Additional logic for A_EQ_B and Enable features

Gate-level design is probably synthesized rather than hand design.

Very general code allows the synthesis tool to decide what logic gates to use and how.

Simplifies design, saves time, but will the tool make good choices? (Not always!)
MAGNITUDE COMPARATORS – EXTENDING TO MORE BITS

- Good example of re-use of blocks to save design time!
- Create a new function out of existing blocks, plus a little logic.
- When you build a new block, ask yourself if this is worth adding to your library?

Figure 11.18 8-bit comparator system

- Note that, A_GT_B goes high if M_GT is high, regardless of other inputs (except enable).

Figure 11.19 Comp 8 logic diagram

- Need to be careful when re-using existing circuit blocks!
  - Be sure you understand how a block works, or you may be in for a surprise!
**Priority Encoder**

<table>
<thead>
<tr>
<th>$d_7$</th>
<th>$d_6$</th>
<th>$d_5$</th>
<th>$d_4$</th>
<th>$d_3$</th>
<th>$d_2$</th>
<th>$d_1$</th>
<th>$d_0$</th>
<th>$Q_3$</th>
<th>$Q_2$</th>
<th>$Q_1$</th>
<th>$Q_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$d_7$ has highest priority  
$d_0$ has lowest priority  
$Q_3 = 1$ when $d_i = 1$ for any $t = 0, ..., 7$.

* Q3 tells you an input is high.  
* Q2–0 tell you, of the high inputs which has highest priority.

**Figure 11.20** Function table for an 8-bit priority encoder

![Priority Encoder Symbol](image_url)

**Figure 11.21** Symbol for priority encoder

* An example of where a priority encoder is used would be interrupts to a CPU (e.g., do I handle an interrupt request from my keyboard or printer first?)
The logic for the network is drawn in two parts. The first section in Figure 11.22 shows the input buffers and complement generators for each bit. The output logic for the \( Q_2 \) and \( Q_3 \) is simple and is given by the expressions:

\[
\begin{align*}
Q_2 &= (d_0 + d_1 + d_2 + d_3) \cdot (d_4 + d_5 + d_6 + d_7) \\
Q_3 &= (d_0 + d_1 + d_2 + d_3) \cdot (d_4 + d_5 + d_6 + d_7)
\end{align*}
\]

as can be verified from the schematic. The \( Q_0 \) and \( Q_1 \) encoders use the buffered and complemented inputs as shown in the circuits of Figure 11.23. The logic equation for the \( Q_0 \) circuit is:

\[
\begin{align*}
Q_0 &= \overline{d_7} \cdot (d_0 + \overline{d_3} \cdot (d_4 + \overline{d_5} \cdot [d_6 + \overline{d_1} \cdot \overline{d_0}])) \\
Q_1 &= \overline{d_7} \cdot \overline{d_6} \cdot [d_5 + \overline{d_4} + \overline{d_3} \cdot \overline{d_2} \cdot (d_1 + d_0)]
\end{align*}
\]

gives the \( Q_1 \) bit.

Even though the internal details of the circuit are complicated, the behavioral description is concerned only with the overall functional behavior. One implementation for the module is:

```verilog
module priority_8 (Q, Q3, d);
input [7:0] d;
output Q3;
output [2:0] Q;
always @ (d)
begin
  Q3 = 1;
  if (A[7]) Q = 7;
  else if (A[6]) Q = 6;
  else if (A[5]) Q = 5;
  else if (A[4]) Q = 4;
  else if (A[3]) Q = 3;
  else if (A[2]) Q = 2;
  else if (A[1]) Q = 1;
  else Q = 0;
  begin
    Q3 = 0;
    Q = 3'b000;
  end
endmodule
```

\{ THE DESIGNER MAY NOT EVER WRITE THESE EQUATIONS! \}

\{ VERY GENERAL, NO GATE LEVEL DETAILS \}
Figure 11.22 Logic diagram for the priority encoder

Figure 11.23 Q0 and Q1 circuits for the 8-bit priority encoder

- Complex CMOS gates this big are rarely used!
  - Only in full custom design, when speed is not an issue
  - Can be built, layout much faster using standard cells!
right movement. For example, a 1-bit left rotation yields an output of:

$$f_3 f_2 f_1 f_0 = a_2 a_1 a_0 a_3$$  \hspace{1cm} (11.16)

while a 1-bit right rotation gives

$$f_3 f_2 f_1 f_0 = a_3 a_2 a_1 a_0$$  \hspace{1cm} (11.17)

A rotation exhibits wrap-around behavior where a bit that is pushed out of the word is added to the other side. A shift operation forces a 0 into the empty space. If we modify the unit to give a 1-bit shift left operation, then an input of $a_0 a_2 a_1 a_0$ produces an output of

$$f_3 f_2 f_1 f_0 = a_2 a_1 a_0 0$$  \hspace{1cm} (11.18)

with a similar behavior for a shift right operation.

Verilog provides bit-wise shift operators of

- $\ll$ // This is a shift left operation
- $\gg$ // This is a shift right operation

that can be used to specify vector shifts; both fill slots with 0s. These are shown in the example code

```verilog
reg [7:0] a;
reg [7:0] new_1;
reg [3:0] new_2;
reg [3:0] b;
new_1 = a $\ll$ b; // This shifts the 7-bit word a by b-bits to the right
new_2 = a $\gg$ b; // This shifts a by b-bits to the left
```

A rotation can be specified in a number of different ways. The simplest is a bit-by-bit assignment as in the clocked-behavior unit that is described by the listing

```verilog
reg [3:0];
always @ (posedge clk)
begin // This is a bit-by-bit rotate left
a[0] <= a[3];
a[1] <= a[0];
a[2] <= a[1];
a[3] <= a[2];
end
```

---

**OPERATES WHEN CLOCK GOES FROM 0 TO 1 (+EDGE)**
**ROTATE CIRCUITS**

*Pass gate implementations use MOS switches to connect inputs to desired outputs.*

→ NMOS passes a "weak 1" → needs an output buffer

---

**Figure 11.25** A 4-bit rotate-right network

**Figure 11.26** Left-rotate switching array

*Can be very small

A good layout can allow this sort of cell to be easily programmed for different functions!

(E.g., one layout for both cells above, just change vias!)
**Barrel Shifters**

**Figure 11.27** An $8 \times 4$ barrel shifter

- Out of $m$ inputs, select which $n$ to send to the output.
- Can also lay this out using a MOS array, with vias to program which MOSFET connects to which input & output.

**Figure 11.28** FET-array barrel shifter
**R/S LATCH**

**NANDS**:

\[ \overline{S} \rightarrow Q \]

\[ \overline{R} \rightarrow \overline{Q} \]

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>\bar{Q}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<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>
</tr>
</tbody>
</table>

- \( S \) = "SET" (ACTIVE LOW)
- \( \overline{R} \) = "RESET" (ACTIVE LOW)

**NORs**:

\[ R \rightarrow Q \]

\[ S \rightarrow \overline{Q} \]

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>\overline{Q}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Hold</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- \( S \) = "SET*" (ACTIVE HIGH)
- \( R \) = "RESET" (ACTIVE HIGH)

- LATCHES ARE USED TO STORE DATA
- R/S LATCHES ARE ASYNCHRONOUS \( \Rightarrow \) NOT CLOCKED,
  Output Changes as Soon as Input Does (After Delay of Gates)

\( \Rightarrow "TRANSPARENT" \)
**D-LATCH**

- Helps input state to form of static memory
- Based on R/S latch NOR version, NAND version also available
- Single "D" input replaces R/S inputs, thus preventing disallowed states (e.g., S=R=1 for NOR SR)
- "Transparent" → Q, Q̅ change when D changes, after gate delays
  → Asynchronous (not clocked)

![D-Latch Symbol and Logic Diagram](image)

**Figure 11.29** D-latch

**Figure 11.30** CMOS circuit for a D-latch

- Also referred to as a "cross-coupled" latch, due to connection type
- If D = 1, latch is "set" ⇒ Q = 1, Q̅ = 0
  - If D = 0, latch is "reset" ⇒ Q = 0, Q̅ = 1

* In this form, this circuit is not terribly useful
  → Output changes as soon as the input does, so where's the memory?
**D-LATCH WITH ENABLE**

(a) Symbol

(b) Logic diagram

Figure 11.31 Gated D-latch with Enable control

Figure 11.32 AOI CMOS gate for D-latch with Enable

- **Addition of "Enable" input makes this circuit much more useful!**
  - Output only changes when you want it to, regardless of value of "D" input, because the "and" gates the input (En ≈ a clock)

- **If En = 0** ⇒ Output of both AND gates = 0 ⇒ Latch holds its previous value

- **If En = 1** ⇒ Output of AND gates depends on "D", setting or resetting the latch as desired

⇒ A form of static memory!
**BISTABLE CIRCUITS & RING OSCILLATORS**

(a) Bistable circuit  
(b) Ring oscillator

**Figure 11.33** Closed-loop inverter configurations.

(a) Stable states  
(b) CMOS circuit

**Figure 11.34** Operation of a bistable circuit

- **BISTABLE CIRCUITS ARE THE BASIC BUILDING BLOCKS USED IN MOST LATCHES AND FLIP-FLOPS**
  - Cross-coupled inverters (or NANDs, NORs, etc.)
  - Each inverter output "re-enforces" the value stored
  - Will hold the stored value as long as power is applied
  - Uses positive feedback!

- **RING OSCILLATORS** are built using an odd number of gates
  - No stable state exists! Therefore the logic level at each point in the ring oscillates back and forth between 1 and 0
    \[ 1 \rightarrow 0 \rightarrow 1 \ldots \]
  - Often used to build a simple oscillator
  - Often used to characterize the speed of a process

**Note:** For \( n \) gates, \( \text{freq} = \frac{1}{2nT_d} \)

Where: \( T_d = \text{gate delay} \)
**Bistable Circuits as Latches**

![Diagrams of bistable circuits](image)

(a) Receiver circuit  
(b) Controlled loading

**Figure 11.35** Adding an input node to the bistable circuit

- **D INPUT MUST BE STRONG ENOUGH TO OVERCOME THE DRIVE FROM INVERTER 1 IN ORDER TO FORCE A GIVEN STATE INV 2 SHOULD BE "WEAK"**
- **ADDING A T-GATE INPUT ALLOWS THE OUTPUT TO ONLY CHANGE AT DEFINED TIMES, SIMILAR TO ENABLE IN PREVIOUS CIRCUIT**

![Diagrams of bistable circuits](image)

(a) CMOS TG version  
(b) nFET pass gates

**Figure 11.36** D-latch using oppositely phased switches

- **C = 1**
- **C = 0**

![Diagrams of bistable circuits](image)

(a) Load with C = 1  
(b) Hold with C = 0

**Figure 11.37** Operation of the D-latch

- **THE INPUT MUST "FIGHT" WITH INV 2 & OUTPUT IN THE CIRCUIT OF FIG. 11.35 = SLOWS DOWN THE LATCH!**

**SOLUTION:** BREAK THE FEEDBACK WHILE THE INPUT IS APPLIED!
(a) C²MOS static latch
(b) Dynamic latch

Figure 11.38 C²MOS-based D-latch circuits

(2) Above uses "clocked inverters" to gate the input and feedback paths instead of T-gates
   → When active, a clocked inverter acts just like a normal inverter (just a little slower, due to extra series FETs)
   → When off, a clocked inverter presents a high impedance at its output ("high-z state")

(4) Above is a "dynamic latch", which stores the input as charge on a capacitor
   → Works, but easy to mess up!
      (e.g., what happens if another signal couples onto the charge storage node? The data can be changed!) risky!
   → Will work at higher frequencies than static latches (but beware of low freq clocks!)

*Tip: Only use dynamic circuits when you have no other choice! (e.g., static circuits are too slow)
D Flip-Flops

(a) Positive edge-triggered DFF
(b) Negative edge-triggered DFF

Figure 11.40 Edge-triggered DFF symbols

Figure 11.39 Master-slave D-type flip-flop

A Verilog behavioral description of a positive edge-triggered DFF can be written in the following manner:

```
module positive_diff (q, q_bar, d, clk);
input d, clk;
output q, q_bar;
reg q, q_bar;
always @ (posedge clk)
begin
    q = d;
    q_bar = ~ d;
end
endmodule
```

In a realistic application, a set of delay times would be needed. A negative edge-triggered module is obtained by modifying the `always` statement to

```
always @ (negedge clk)
```

- The "master" latch latches the input on either the rising or falling edge of clock (depends on phasing)
- While the master is acquiring the new data, the "slave" latch holds the data from last time
- When the master latches the input, the data is transferred to the slave → NOT TRANSPARENT, "CLOCK-TO-Q" DELAY

![Very high level, no circuit details]

Figure 11.41 Alternate circuitry for the master-slave DFF

- Still works, but slower!
  (look @ # of gate delays in signal path)
**TOGGLE FLIP FLOP**

![Diagram of a toggle flip flop]

*Figure 11.44 DFF modified to a TFF circuit using feedback*

- "TOGGLE" FLIP FLOPS JUST SWITCH BACK AND FORTH BETWEEN $Q = 1$ AND $Q = 0$ ON EACH CLOCK
- REALLY JUST A $D$ FLIP FLOP WITH $D = \overline{Q}$ HARD WIRED IN CELL
- VERY USEFUL TO DIVIDE A CLOCK DOWN TO $\frac{1}{2}$ IT'S FREQUENCY!

\[ \text{Input} \quad \overline{Q} \]
\[ \overline{\overline{Q}} \quad \overline{Q} \quad Q \]


**D-Flip-Flops with Set and Clear**

(a) DFF with direct Clear

(b) DFF with direct Clear and Set

---

**Figure 11.49** DFF circuits with assert-low Clear and Clear/Set controls

- **By changing some of the inverters to NAND/NOR, can add inputs which force the outputs to either** $Q = 1$ **(set) or** $Q = 0$ **(reset) (or clear)**

- Very useful when resetting a state machine to a known state, and on power up of chip

**Note:** Without this, the state of the flip-flop on power up is unknown!

Do the other circuits on your chip allow either state on power up? Sometimes, but often no!
D Flip Flops with Load Control

- Only latches new input when "load" is active
- The circuit in Fig. 11.45 is better than the one in Fig. 11.46, because the data is gated by load instead of the clock

*Tip! Avoid gating clocks whenever possible!

Figure 11.45 D-type flip-flop with Load control

Figure 11.46 CMOS master-slave FF with Load control

(a) Load with $\phi = 0$, Load = 1

(b) Hold with $\phi = 1$, Load = 0

"Quasi-dynamic" in this state, since the master's output depends on charge stored on $\overline{CS}$

Risky!

Figure 11.47 Operation of the CMOS DFF with load control
A "register" is an array of flip flops intended to handle n bits at a time (e.g., 16 bits).

Figure 11.48 Construction of an n-bit register

Figure 11.49 One-bit static multiport register circuit

"Multiport" outputs allow the register outputs to be sent to several different outputs (used less often).

Figure 11.50 An n-bit static multiport register