Basic Logic Gates

Buffer

Input	Output
0	0
1	1

Inverter

Input	Output
0	1
1	0

AND

A	B	Output
0	0	0
1	0	0
0	1	0
1	1	1

A•B=Y

NAND

A	B	Output
0	0	1
1	0	1
0	1	1
1	1	0

OR

A	B	Output
0	0	0
1	0	1
0	1	1
1	1	1

NOR

A	B	Output
0	0	1
1	0	0
0	1	0
1	1	0

XOR

A	B	Output
0	0	0
1	0	1
0	1	1
1	1	0

XNOR

A	B	Output
0	0	1
1	0	0
0	1	0
1	1	1

SOPs and POSs

We can create boolean algebra expressions for truth tables.

Minterm: Corresponds to each row of truth table, i.e. $m_3 = \overline{x_2}x_1x_0$ such that when $3 = 0b011$ is substituted in, $m_3 = 1$ and $m_3 = 0$ otherwise.

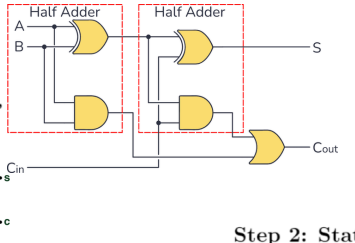
Maxterm: They give $M_i = 0$ if and only if the input is i. For example, $M_3 = x_2 + \overline{x_1} + \overline{x_0}$.

SOP and POS: Truth tables can be represented as a sum of minterms, or product of maxterms.

- Use minterms when you have to use NAND gates and maxterms when you have to use NOR gates.
- When converting expressions to its dual, it's often helpful to negate expressions twice, or draw out the logic circuit.

Half Adder

```
module HA(x, y, s, c);
    input x, y; output s,
    assign s = x^y;
    assign c = x&y;
endmodule
```



Full Adder

```
module FA(a, b, c_in, s_out, c_out);
    input a, b, c_in; output s_out, c_out;
    wire w1, w2, w3;
    HA u0(.x(a), .y(b), .s(w1), .c(w2));
    HA u1(.x(c_in), .y(w1), .s(s_out), .c(w3));
    assign c_out = w2|w3;
endmodule
```

ModelSim and Other Lab Things

- FGPA: Field Programmable Gate Array
- To repeat signals, use this syntax:

  `force {MuxSelect[2]} 0 0ns, 1 {4ns} -r 8ns`

  which starts at 0 at 0ns, 1 at 4ns, and repeats every 8 ns.
- On the DE1-SoC board, hex thing is red if 0 and white if 1.

Number Conversion

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^9 = 512$
$2^{10} = 1024$
$2^{11} = 2048$
$2^{12} = 4096$
$2^{13} = 8192$
$2^{14} = 16384$
$2^{15} = 32768$
$2^{16} = 65536$

Octal Base (Base 8)

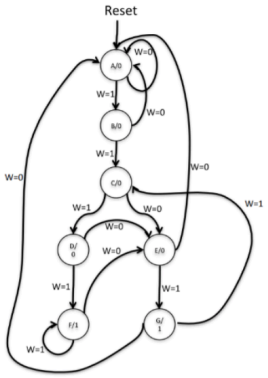
$8^1 = 8$
$8^2 = 64$
$8^3 = 512$
$8^4 = 4096$
$8^5 = 32768$
$8^6 = 262144$
$8^7 = 2097152$
$8^8 = 16777216$
$8^9 = 134217728$
$8^{10} = 1073741824$

Finite State Machines

Steps

1. State Diagram
2. State Table
3. State Assignment
4. State-Assigned Table
5. Synthesize Circuit
6. Celebrate!

Step 1: State Diagram Example



Step 2: State Table Example

Present State	Next State		Output (z)
A	A	B	0
B	A	C	0
⋮	⋮	⋮	⋮
G	A	C	1

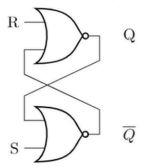
Step 3: State Assignment Example

- Using **one-hot encoding:** Choose number of flip flops: 7 (since 7 states)
- Choose state codes:
  - A = 0000001, B=0000010, …, G=1000000

Alternatively use 3 flip flops to represent state codes as 000, 001, 010, etc.

RS Latch

Sequential circuits depend on sequence of inputs. A **SR Latch** are cross-coupled NOR gates.



S	R	Q	$\overline{Q}$
0	0	0/1	1/0
0	1	0	1
1	0	1	0
1	1	0	0

When $S = R = 0$, it stores the last Q value. In practice, we should not have $S = R = 1$.

Operator	Operation
a&b	Bitwise AND
a|b	Bitwise OR
a^b	Bitwise XOR
~(a&b)	Bitwise NAND
~(a|b)	Bitwise NOR
~(a^b)	Bitwise XNOR

Step 4: State-Assigned Table Example

By convention, use y for input and Y for output.

$y_3y_2y_1$	$Y_3Y_2Y_1$ $(W = 0)$	$Y_3Y_2Y_1$ $(W = 1)$	z
000	000	001	0
001	000	010	0
⋮	⋮	⋮	⋮
110	000	010	1

Step 5: Synthesize Example

We first write boolean algebra expressions for the outputs $Y_n = f_n(y_1, y_2, y_3, W)$ and $z = g(y_1, y_2, y_3)$. For each flip flop i, the input is Y_i and the output is y_i. The output then branches off into two paths:

- The first path goes into the function $g(y_1, y_2, y_3)$ and leads to output z
- The second path goes into the function $f_nm(y_1, y_2, y_3, W)$ and **loops back** to Y_n.

The D flip flops are connected to same clock and reset signal.

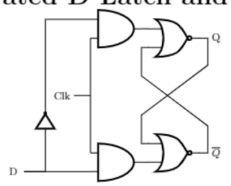
ModelSim Do Files

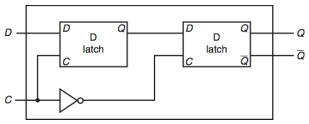
```
# set working dir, where compiled verilog goes
vlib work
# compile all verilog modules in mux.v to working
# dir could also have multiple verilog files
vlog mux.v
#load simulation using mux as the
# top level simulation module
vsim mux
#log signals and add signals to waveform window
log {/*}
# add wave {/*} would add all items in
# top level simulation module
add wave {/*}
# set input values using the force command
# signal names need to be in {} brackets
force {SW[0]} 0
force {SW[1]} 0
run 10ns
```

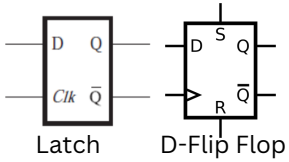
Resets

- Active High/Low: Resets when Signal is 1/0
- Synchronous High/Low: Resets during positive/negative edge

Gated D Latch and







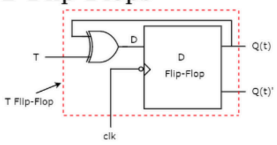
Latch D-Flip Flop

AB / CD

CD \ AB	00	01	11	10
00	m0	m4	m12	m8
01	m1	m5	m13	m9
11	m3	m7	m15	m11
10	m2	m6	m14	m10

x_3 \ x_1x_2	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

T Flip Flops



clk	$Q(t+1)$
↑	T^Q(t)

Verilog for D_latch

```
module D_latch(input logic D,
               input logic Clk,
               output logic Q);
    always_latch
        if (Clk == 1)
            Q <= D;
endmodule
```

Verilog for D-Flip Flop

```
module D_ff(input logic D,
            input logic Clk,
            output logic Q);
    always_ff @(posedge clk)
        Q <= D;
endmodule
```

Verilog for FF (Stores on both edges)

```
module DDR(input logic D,
           input logic Clk,
           output logic Q);
    logic p, n;
    always_ff @(posedge Clk)
        p <= D;
    always_ff @(negedge Clk)
        n <= D;
    assign Q <= Clk ? p : n;
endmodule
```

Verilog for T- Flip Flop

```
module T_flip_flop(t, clk, reset, q);
    input wire t;        // T input
    input wire clk;      // Clock input
    input wire reset;    // Reset input
    output reg q;        // Output Q
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            q <= 1'b0;  // Reset the flip-flop to 0
        end else begin
            if (t) begin
                q <= ~q;  // Toggle the output when T is 1
            end
        end
    end
endmodule
```

```
module part1(
    input logic Clock,
    input logic Reset,
    input logic w,
    output logic z,
    output logic [3:0] CurState);
    // Enum allows you to use names for numerical values, such as state encoding
    // With enums, you will see the names (e.g., A) in the waveform, instead of
    // the state encoding (e.g., 0000), which makes debug easier.
    typedef enum logic [3:0] {A = 4'd0,
                              B = 4'd1,
                              C = 4'd2,
                              D = 4'd3,
                              E = 4'd4,
                              F = 4'd5,
                              G = 4'd6} statetype;
    statetype y_Q, Y_D; // y_Q represents current state, Y_D represents next sta
    //State table
    //The state table should only contain the logic for state transitions
    //Do not mix in any output logic. The output logic should be handled separate
    //This will make it easier to read, modify and debug the code.
    always_comb begin
        case (y_Q)
            A: begin
                if (!w) Y_D = A;
                else Y_D = B;
            end
            B: begin
                if (!w) Y_D = A;
                else Y_D = C;
            end
            C: begin
                if (!w) Y_D = E;
                else Y_D = D;
            end
            D: begin
                if (!w) Y_D = E;
                else Y_D = F;
            end
            E: begin
                if (!w) Y_D = A;
                else Y_D = G;
            end
            F: begin
                if (!w) Y_D = E;
                else Y_D = F;
            end
            G:begin
                if (!w) Y_D = A;
                else Y_D = C;
            end
        endcase
        default: Y_D <= statetype'('dx);
    endcase
end // state_table

// State Registers
always_ff @(posedge Clock) begin
    if(Reset == 1'b1)
        y_Q <= A;
    else
        y_Q <= Y_D;
end // state_FFS

// Output logic
// Set out_light to 1 to turn on LED when in relevant states
assign z = ((y_Q == F) | (y_Q == G));

assign CurState = y_Q
endmodule
```

	Reset	1	2	3	4	5	6	7	
go		1	0	1	0	0	0	0	
data_in		5	5	3	3	X	X	X	
RA		0	5	5	5	5	25	25	
RB		0	0	0	3	3	3	3	
RR		0	0	0	0	0	0	28	
ld_a		1	0	0	0	1	0	0	
ld_b		0	0	1	0	0	0	0	
ld_alu_out		0	0	0	0	1	0	0	= select alu output
alu_select_a		0	0	0	0	0	0	0	0 = select A
alu_select_b		0	0	0	0	0	1	0	1 = select B
alu_op		0	0	0	0	1	0	0	0 = add, 1 = multiply
ld_r		0	0	0	0	0	1	0	
result_valid		0	0	0	0	0	0	1	

Step	Data changes	Control changes to setup for next cycle
1		Select data_in to RA ld_a=1, ld_alu_out=0
2	RA gets data_in RA=5	ld_a=0
3		Select data_in to RB ld_b=1, ld_alu_out=0
4	RB gets data_in RB=3	ld_b=0
5		Select RA for both ALU inputs: alu_select_a=0 alu_select_b=0, alu_op=1 ld_a=1, ld_alu_out=1
6	RA gets A×A = 25	Select RA & RB as inputs alu_select_a=0, alu_select_b=1, alu_op=0, ld_r=1
7	RR gets result = 28	result_valid=1

Load and Store

- LDR Ra, [Rb], #offset: value at [address] found in Rb is loaded into register Ra. Then the [address] is incremented by offset.
- STR Ra, [Rb], #offset: value found in register Ra is stored to [address] found in Rb. Then the [address] is incremented by offset.
- LDR Ra, =LIST Makes Ra contain the address to the first element of the input variable.

Byte Address				Word Address	Data				Word Nur
3	2	1	0	0x00000000	01	23	45	67	word 0
7	6	5	4	0x00000004	FF	EE	DD	CC	word 1
B	A	9	8	0x00000008	AA	BB	CC	DD	word 2
F	E	D	C	0x0000000C	~~FF~~	~~EE~~	~~FF~~	~~FF~~	〃 3
3	2	1	0	0x00000010	00	00	00	01	〃 4

what's actually in memory

Load word instruction reads a data word from memory into a register

```
a = mem[2];   #s7 = a
              lw s7, 8(zero)   # s7 = 0xAABBCCDD
              word 2
```



```
.global _start
.text
_start:
  la s2, LIST
  addi s10, zero, 0  # Sum
  addi s11, zero, 0  # Counter

loop:
  lw s3, 0(s2)
  bge x0, s3, END
  add s10, s10, s3
  addi s2, s2, 4
  addi s11, s11, 1
  j loop

END:
  ebreak

.global LIST
.data
LIST:
  .word 1, 2, 3, 5, 0xA, -1
```

Registers

```
module reg8(D, clk, Q);
    input clock;
    input [7:0] D;
    output reg[7:0] Q;
    always@(posedge clock)
        Q <= D;
    endmodule
```

Common Logic Gates

- Mux 2→1: mux2to1$(s, x_0, x_1) = \bar{s}x_0 + sx_1$
- Not: not(x)=nand(x,x)=nor(x,x)
- XOR acts as modular arithmetic.
- Multiplexers are functionally complete. AND $= mux(x, y, 1)$, OR $= mux(x, 0, y)$.

```
module mux2to1(x, y, s, m);
    input logic x; //select 0
    input logic y; //select 1
    input logic s; //select signal
    output logic m; //output
    assign m = s & y | ~s & x;  // OR
    assign m = s ? y : x;
endmodule
```

```
module ripple_carry_adder_4_Bit(
    input logic [3:0] a, b,
    input logic c_in,
    output logic [3:0] s,
    output logic [3:0] c_out);
full_adder fa0 (a[0], b[0], c_in, s[0], c_out[0]);
full_adder fa1 (a[1], b[1], fa0.cout, s[1], c_out[1]);
full_adder fa2 (a[2], b[2], fa1.cout, s[2], c_out[2]);
full_adder fa3 (a[3], b[3], fa2.cout, s[3], c_out[3]);
endmodule

module full_adder (
    input logic a,
    input logic b,
    input logic cin,
    output logic s,
    output logic cout);
    // Logic
    assign s = a ^ b ^ cin;
    assign cout = (a & b) | (a & cin) | (b & cin);
endmodule
```

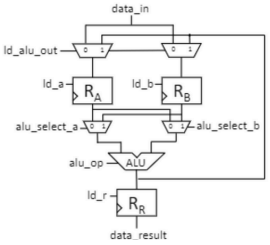


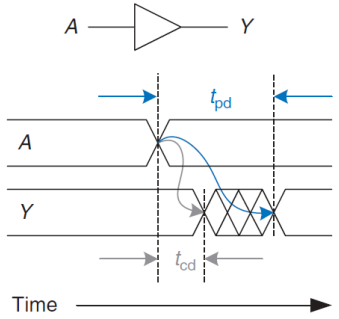
Table 2: Register contents and control signals for computing $A^2 + B$

	Reset	1	2	3	4	5	6	7	
Go		1	0	1	0	0	0	0	
data_in		5	5	4	4	-	-	-	- = don't care.
State		0	2	3	4	5	6	1	
R_A		0	5	5	5	5	25	25	
R_B		0	0	0	4	4	4	4	
R_R		0	0	0	0	0	0	29	
ld_a		1	0	0	0	1	0	1	
ld_b		0	0	1	0	0	0	0	
ld_r		0	0	0	0	0	1	0	
ld_alu_out		0	0	0	0	1	0	0	1 = select alu output
alu_select_a		0	0	0	0	0	0	0	0 = select A
alu_select_b		0	0	0	0	0	1	0	1 = select B
alu_op		0	0	0	0	1	0	0	0 = add, 1 = multiply
result_valid		0	0	0	0	0	0	1	

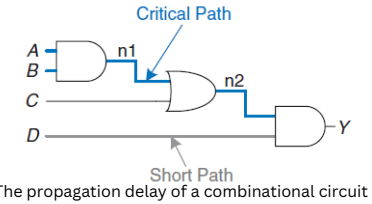
```
module ALU(
  input logic [3:0] A,          // 4-bit input A
  input logic [3:0] B,          // 4-bit input B
  input logic [1:0] Function,   // 2-bit function select
  output logic [7:0] ALUout  ); // 8-bit output
  logic [3:0] and_result;
  logic [3:0] carry_out;  // Capture carry-out from the addition
  // Part 1: Ripple Carry Adder
  part1 u0 (
    .a(A),
    .b(B),
    .c_in(0),
    .s(add_result),
    .c_out(carry_out));// Corrected connection
  // Part 2: Multiplexer to select the output based on Function
  always_comb
  begin
    case (Function)
      0: ALUout = {3'b0, carry_out[3], add_result};
      1: ALUout = |(A | B) ; // Part 3: OR Operation
      2: ALUout = &(A & B); // Part 4: AND Operation
      3: ALUout = {A, B}; // Concatenate A and B
      default: ALUout = 8'b0; // Assuming default behavior is to output 0
    endcase
  end
endmodule
```

Timing Analysis

Combinational logic is characterized by its propagation delay and contamination delay. The propagation delay tpd is the maximum time from when any input changes until the output or outputs reach their final value. The contamination delay tcd is the minimum time from when any input changes until any output starts to change its value.



The critical path, shown in blue, is the path from input A or B to output Y. It is the longest—and, therefore, the slowest—path because the input travels. through three gates to the output. This path is critical because it limits the speed at which the circuit operates. The short path through the circuit, shown in gray, is from input D to output Y.



The propagation delay of a combinational circuit is the sum of the propagation delays through each element on the critical path. The contamination delay is the sum of the contamination delays through each element on the short path.
T_cq -> "Clock to Q" Propagation Delay in Flip Flop
T_su -> Set-up time - The data D must be stable during the set-up time
T_h -> Hold time - The data D must remain stable during the hold time

To calculate "the clock: Maximum clock frequency results from a minimum time delay. Look for the longest path between 2 Flip Flops & includes the Set-up Time
T_min = T_cq + T_logic (gates) + T_su

Hold Time Violation - Shortest path between 2 Flip flops, Data racing through the clock (set up time not used) To check for hold time violation use t_h + Delta < = T_cq + T_logic_min (short Path) if inequality balances out - no hold time violation if doens't balance...there is hold time violation

Setup Time violation - Logic too slow for correct value to arrive at input to FF on right (D2) by T_su before clock edge.