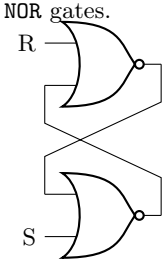
ECE253 Final Cheatsheet

Author: your mother

RS Latch

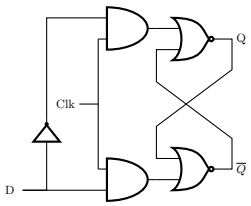
Sequential circuits depend on sequence of inputs. A **SR Latch** are cross-coupled `NOR` gates.



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 $S = R = 1$.

Gated D Latch and Clock Signal

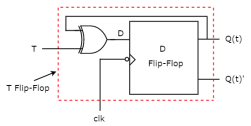


Clk	$Q(t+1)$
0	Q(t)
1	D

D Flip Flops

Consists of two gated D latches, connected in series and both connected to the same clock. However, clock input for the first D latch is inverted. When the clock rises up, Q stores value of D.

T Flip Flops



clk	$Q(t+1)$
↑	`T^Q(t)`

SystemVerilog

Logic Operators

bitwise AND	&	bitwise OR	|
bitwise NAND	~&	bitwise NOR	~|
bitwise XOR	^	bitwise XNOR	~^
logical negation	!	bitwise negation	~
concatenation	{}	replication	{{}}

- `reduction` operators are put at the start and output a scalar.
- `bitwise` operators
- `blocking assignment =`: executed in the order they are specified.
- `Nonblock assignments <=` executed in parallel.
- Use `logic` instead of `reg/wire` (4-state type)
- Use `always_comb` for combinational, `always_ff` for sequential logic

Case Statements

```
module mux(
    input logic [2:0] MuxSelect,
    input logic [4:0] Input,
    output logic Out
);
    always_comb begin
        case (MuxSelect)
            3'b000: Out = Input[0];
            // ...
            3'b100: Out = Input[4];
            default: Out = 1'bx;
        endcase
    end
endmodule
```

Half Adder

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

Full Adder

```
module FA(
    input logic a, b, c_in,
    output logic s_out, c_out
);
    logic 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
```

D Flip Flop

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

T Flip Flops

```
module t_ff(
    input logic Clock, Clear_b, T,
    output logic Q
);
    always_ff @(posedge Clock, negedge Clear_b) begin
        if (Clear_b == 1'b0)
            Q <= 1'b0;
        else
            Q <= T ^ Q;
    end
endmodule
```

Registers

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

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 win
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
```

ModelSim and Other Lab Things

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

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

  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.

Frequency Dividers

- To half the frequency, connect $\overline{Q}$ to D on the same gated D latch.
- To quarter the frequency, connect $\overline{Q}$ to the clock of the next gated D latch (which is set up the same as the half frequency case).
- To reduce frequency by $2k$, connect k D latches connected in series (D to Q) and to the same clock. First D is connected to last $\bar{Q}$. The last Q will have a reduced frequency of $2k$.

Resets

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

Finite State Machines

Steps: (1) State Diagram (2) State Table (3) State Assignment (4) State-Assigned Table (5) Synthesize Circuit

State Encoding: *One-hot:* n states = n flip flops (A=0000001, B=0000010, ...). *Binary:* n states = $\lceil \log_2 n \rceil$ flip flops (000, 001, 010, ...).

Synthesis: Write $Y_n = f_n(y_1, \ldots, W)$, $z = g(y_1, \ldots)$. For each FF i: input=Y_i, output=y_i. Output branches to (1) g for output z, (2) f_n loops back to Y_n. FFs share clock/reset.

FSM in SystemVerilog

```
module FSM(input logic Clock, Resetn, w,
    output logic z, output logic [3:0] CurState);
logic [3:0] y_Q, Y_D;
typedef enum logic [3:0] {A=4'b0000, B=4'b0001,
    /*...*/ G=4'b0110} state_t;
always_comb begin
    case (y_Q)
        A: Y_D = w ? B : A;
        /*...*/
        G: Y_D = w ? C : A;
        default: Y_D = A;
    endcase
end
always_ff @(posedge Clock) begin
    if (!Resetn) y_Q <= A;
    else y_Q <= Y_D;
end
assign z = (y_Q == F) | (y_Q == G);
assign CurState = y_Q;
endmodule
```

RISC-V Assembly

Conditionals

RISC-V has no flags. Branch instructions: `beq` (equal), `bne` (not equal), `blt/bge` (less/greater signed), `bltu/bgeu` (unsigned). Set: `slt`, `slti`, `sltu`, `sltiu`.

Interrupts

1. Set `mtvec` CSR (trap vector)
2. Enable interrupts in `mstatus` (MIE bit)
3. Enable sources in `mie` CSR
4. Configure PLIC/CLIC
5. Save context on trap entry
6. Read `mcause` CSR for cause
7. Handle in ISR, clear PLIC pending bit
8. Restore context, return with `mret`

RISC-V Assembly Example Code

Enabling Interrupts

```
li sp, 0x10000        # Initialize stack
la t0, trap_handler
csrw mtvec, t0        # Set trap vector
li t0, 0x8
csrs mstatus, t0      # Enable interrupts (MIE)
li t0, 0x800
csrs mie, t0          # Enable external int (MEIE)
li t0, 0xFF20058
li t1, 0b1001
sw t1, 0(t0)          # Enable key3, key0
```

Check Cause of Interrupt

```
trap_handler:
    addi sp, sp, -32
    sw t0, 0(sp); sw t1, 4(sp); sw a0, 8(sp)
    sw a1, 12(sp); sw ra, 16(sp)
    csrr t0, mcause       # Read cause
    li t1, 0x8000000B
    bne t0, t1, error_trap
    jal ra, key_isr
    j exit_trap
error_trap:
    j error_trap
exit_trap:
    lw t0, 0(sp); lw t1, 4(sp); lw a0, 8(sp)
    lw a1, 12(sp); lw ra, 16(sp)
    addi sp, sp, 32
    mret
```

ISR Subroutine

```
key_isr:
    addi sp, sp, -20
    sw t2, 0(sp); sw t3, 4(sp)
    sw t4, 8(sp); sw t5, 12(sp)
    la t5, CURR_VALUE
    lw t4, 0(t5)
    li t2, 0xFC20005C
    lw t3, 0(t2)
    li t0, 0b1000
    bne t3, t0, key0
    beq t4, zero, endisr
    addi t4, t4, -1
    sw t4, 0(t5)
    j endisr
key0:
    # code for key 0
endisr:
    lw t2, 0(sp); lw t3, 4(sp)
    lw t4, 8(sp); lw t5, 12(sp)
    addi sp, sp, 20
    ret
```

Polled IO with Timer

```
.text
.globl _start
_start:
    li a0, 0xFFFEC600
    li a1, 200000000
    sw a1, 0(a0)
    li a1, 0b111
    sw a1, 8(a0)
    li s0, 0; li s1, 0; li s2, 0  # sec,min,hr
poll:
    lw a1, 12(a0)
    beq a1, zero, poll
    sw a1, 12(a0)
    addi s0, s0, 1
    li t0, 60
    bne s0, t0, poll
    li s0, 0
    addi s1, s1, 1
    bne s1, t0, poll
    li s1, 0
    addi s2, s2, 1
    li t0, 24
    bne s2, t0, poll
    li s2, 0
    j poll
```

Exception Vector Table

```
.align 2
trap_vector:
    j trap_handler  # Direct: all to one handler
# Vectored: mtvec.MODE=1, base + 4*cause
# 0x00:Exception 0x04:Supervisor SW int
# 0x0C:Machine SW int 0x14:Supervisor timer
```

Find Sum with Recursion

```
.globl _start
_start:
    li sp, 0x20000
    la s0, N
    lw a0, 0(s0)
    li a1, 0
    jal ra, findsum
    add a1, a1, a0
end: j end
findsum:
    addi sp, sp, -8
    sw a0, 0(sp); sw ra, 4(sp)
    li t0, 2
    blt a0, t0, return
    addi a0, a0, -1
    jal ra, findsum
    add a1, a1, a0
return:
    lw a0, 0(sp); lw ra, 4(sp)
    addi sp, sp, 8
    ret
.data
N: .word 5
```

Fibonacci with Recursion

```
.data
N: .word 10
.text
.globl _start
_start:
    li sp, 0x20000
    la s0, N; lw a0, 0(s0)
    li a1, 0; li a2, 0
    jal ra, fib
end: j end
fib:
    addi sp, sp, -12
    sw a0, 0(sp); sw a2, 4(sp);
    sw ra, 8(sp)
    li t0, 2
    bge a0, t0, recur
    mv a1, a0
    lw a0, 0(sp); lw a2, 4(sp
    lw ra, 8(sp)
    addi sp, sp, 12
    ret
recur:
    addi a0, a0, -1
    jal ra, fib
    mv a2, a1
    addi a0, a0, -1
    jal ra, fib
    add a1, a1, a2
    lw a0, 0(sp); lw a2, 4(sp
    lw ra, 8(sp)
    addi sp, sp, 12
    ret
```

Timing Analysis

Minimum Clock Period

$$T_{\min} = T_{su} + T_{CQ} + T_{logic}$$

$$F_{\max} = \frac{1}{T_{\min}}$$

Hold Time Violation

For no hold time violation:

$$t_{hold} + t_{skew} \leq t_{CQ} + t_{logic}$$

where t_{skew} is the minimum clock skew.