# ECE253 Midterm Cheatsheet

## Boolean Algebra

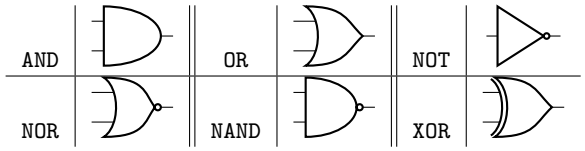
**De Morgan's Theorem** tells us

$$\overline{x \cdot y} = \overline{x} + \overline{y}, \qquad \overline{x + y} = \overline{x} \cdot \overline{y} \tag{1}$$

Inverting the inputs to an `or` gate is the same as inverting the outputs to an `and` gate, and the other way around. We also have:

- $(x + y)(y + z)(\overline{x} + z) = (x + y)(\overline{x} + z)$
- $x + yz = (x + y)(x + z)$
- $x + xy = x$ (Absorption)
- $xy + x\overline{y} = x$ (Combining)
- $(x + y)(x + \overline{y}) = x$
- $x + \overline{x}y = x + y$
- $x(\overline{x} + y) = xy$
- $xy + yz + z\overline{x} = xy + z\overline{x}$ (Consensus)

## Gates




## 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.

## Cost

The cost of a logic circuit is given by

$$\text{cost} = \text{gates} + \text{inputs} \tag{2}$$

If an inversion (`NOT`) is performed on the primary inputs, then it is not included. If it is needed inside the circuit, then the `NOT` gate is included in the cost.

## Karnaugh Map

Method of finding a minimum cost expression: We can map out truth table on a grid for easier pattern recognition. Example of a four variable map is shown below:

| $x_4x_3$ \ $x_2x_1$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 1 | 1 | 0 |
| 01 | 1 | 1 | 1 | 0 |
| 11 | 0 | 0 | 1 | 1 |
| 10 | 0 | 0 | 1 | 1 |

and the representation is $\overline{x_2} \cdot \overline{x_4} + x_2 \cdot x_1 + \overline{x_4} \cdot x_2$ when using *minterms*. To use *maxterms*, we take the intersection of sets that don't include blocks of 0s. For example, $(\overline{x_2} \cdot \overline{x_1})(\overline{x_2} + x_1 + x_4)$. Some *rules*:

- Side lengths should be powers of 2 and be as large as possible.
- Use **graycoding**: adjacent rows/columns should share one bit.

### Minimization Procedure

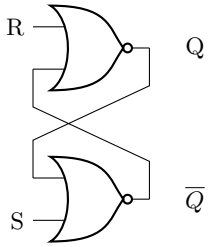
1. Generate all prime implicants for given function $f$
2. Find the set of essential prime implicants
3. Determine the nonessential prime implicants that should be added.

## Common Logic Gates

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

## RS Latch

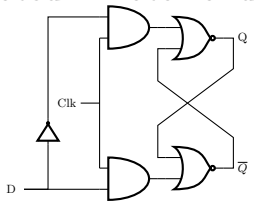
Sequential circuits depend on sequence of inputs. A **SR Latch** are cross-coupled `NOR` gates.




| 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

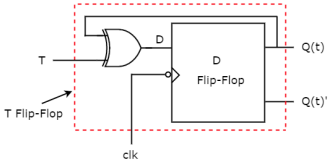



| Clk | $Q(t+1)$ |
|---|---|
| 0 | Q(t) |
| 1 | D |

## 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)` |

## Verilog

### 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.

### Case Statements

```
module mux(MuxSelect, Input, Out);
    input [4:0] Input; input [2:0] MuxSelect;
    output Out;
    reg Out; // declare output for always block
    always @(*) // declare always block
     begin
     case (MuxSelect[2:0]) // start case statement
     3'b000: Out = Input[0]; // case 0
     // ...
     3'b100: Out = Input[4]; // case 4
     default: Out = 1'bx; // default case
     endcase end endmodule
```

### Half Adder

```
module HA(x, y, s, c);
    input x, y; output s, c;
    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
```

### D Flip Flop

```
module D-ff(D, clk, Q);
    input D, clk;
    output reg Q;
    always@(posedge clk) Q <= D; // use <= operator
endmodule
```

### Flip Flop (stores on both edges)

```
module DDR (input c, input D, output Q) ;
    reg p, n;
    always @ (posedge c) p <= D;
    always @ (negedge c) n <= D;
    assign Q <= c ? p : n;
endmodule
```

## Registers

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

### 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.

## 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

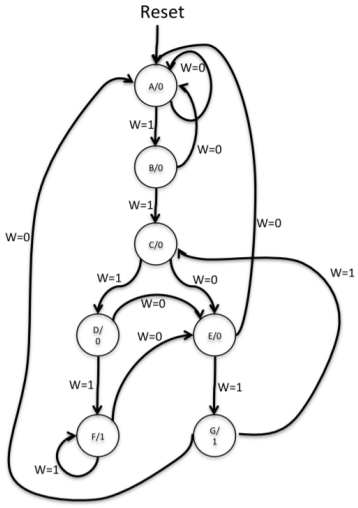
- Active High/Low: Resets when Signal is 1/0
- Synchronous High/Low: Resets during positive/negative edge

# Finite State Machines

## Steps

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

### 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.

### Execution in Verilog

```
module FSM(input Clock, input Resetn, input w,
           output z, output [3:0] CurState);

reg [3:0] y_Q, Y_D;
localparam A=4'b0000, B=4'b0001, ... , G=4'b0110

always@(*) // State Table
begin: state_table
    case: (Y_Q)
        A: begin
                if (!w) Y_D = A; else Y_D = B;
           end
        // ...
        G: begin
                if (!w) Y_D = A; else Y_D = C;
        default: Y_D = A;
    endcase
end

always @(posedge Clock) // State Registers
begin: state_FFs
    if(Resetn == 1'b0) y_Q <= A;
    else y_Q <= Y_D;
end

// Output Logic
assign z = ((y_Q == F) | (y_Q == G));
assign CurState = y_Q;
endmodule
```

## ARM Assembly

### Registers

- R0-R3: Scratch Registers: will be overwritten by subroutines.
- R4-R12: Preserved Registers: stack before using, restore before returning
- R13 (SP): Stack Pointer: points to top of stack
- R14 (LR): Link Register (Points to return address when BL is executed
- R15 (PC): Program Counter: Holds address of next instruction to execute

### Instructions

Let `r0=1`, `r1=2`, `r2=#0b1010`.

| Instruction | Example | Result |
|---|---|---|
| `MOV` | `mov r3, #3` | `r3 = 3` |
| `ADD` | `add r3, r0, r0` | `r3 = 1 + 1` |
| `SUB` | `sub r3, r0, r0` | `r3 = 1 - 1` |
| `MUL` | `mul r3, r0, r0` | `r3 = 1 * 1` |
| `LSL` | `lsl r3, r2, #1` | `r3 = #0b0100` |
| `LSR` | `lsr r3, r2, #1` | `r3 = #0b0101` |
| `ASR` | `asr r3, r2, #1` | `r3 = #0b1101` |
| `AND` | `and r3, r1, r0` | `r3 = (1 and 2) = 0` |

### Memory Stuff

- `BL:Branch Link`: Goes to a branch but updates `LR`
- Stacks: `PUSH {R0, R1}` pushes `R0,R1` to a stack where `R0` is at the top. Last in First Out.
- Each instruction is a place in memory, with addresses going up by 4 bytes.
- Addresses of inputs are stored in the memory immediately after the instructions.

### 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.

**Fancy Stuff:**

- `LDR Ra, [Rb, #offset]` is pre-indexed (doesn't change Rb)
- `LDRB` - load byte (8 bits, which is 2 hex letters)
- `LDRSB` - signed load byte (LDRB but gives sign extension to result to make it 4 bytes while retaining the sign)
- `LDRH` - load halfword (for `[R3, #n]`, n must be even)
- `LDRSH` - signed load halfword

The same applies for STR, but no signed versions.

### Flags

| N | C | V | Z |
|---|---|---|---|
| negative | carry | overflow | zero |

### Conditionals

`CMP R0, R1` computes `R0-R1` and updates flag. We can append conditionals after instructions to act as if-then statements:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| EQ | == | NE | ≠ | GT | ‖ | ¿ | |
| GE | ≥ | LE | | ≤ | ‖ | | |

### Interrupts

1. Provide Exception Vector Table (When an exception occurs, the processor must execute handler code that corresponds to the exception. The location in memory where the handler is stored is called the exception vector.)
2. Initialize SP for IRQ mode, then initialize SP for SVC mode.
3. Configure GIC to enable interrupts (code given)
4. Enable interrupt generation in I/O device.
5. Set I=0 in CPSR.
6. Provide IRQ Handler code which queries GIC to determine source of interrupt.
7. Provide interrupt service routines (ISRs), i.e. `KEY_ISR`
8. The interrupt handler must clear interrupt from GIC.

# ARM Assembly Example Code

## Enabling Interrupts

```
// SP for IRQ
MOV R0, #0b11010010
MSR CPSR, R0 // Now in IRQ mode
LDR SP, =0x20000 // IRQ SP

// SP for SCV
MOV R0, #0b11010011
MSR CPSR, R0 // Now in SVC mode
LDR SP, =0x3FFFFFFC

// Skey enable
LDR R1, =0xFF20058 // mask keys
MOV R2, #0b1001 // enable for key3, key0
STR R2, [R1] // store to enable interrupts

// Enable Interrupts
MOV R1, #0b01010011 // enable in SVC mode
MSR CPSR, R1
```

## Check Cause of Interrupt

```
SERVICE_IQR:
PUSH {R0-R5, LR}
LDR R4, =MPCORE_GIC CPUIF
LDR R5, [R4, #ICCIAR] // makes R5 interrupt ID

CMP R5, #73 // check if key has ID 73
BNE ERROR
BL KEY_ISR // must be a key IRQ, BL to subroutine
B EXIT_IRQ // exit after handling IRQ

ERROR:
B ERROR // unknown IRQ

EXIT_IRQ:
STR R5, [R4, #ICCEOIR]
POP {R0-R5, LR}
SUBS PC, LR, #4
```

## Subroutine to Deal with Interrupts

```
KEY_ISR:
PUSH {R1-R5}
LDR R5, =CURR_VALUE // address of curr_value
LDR R4, [R5] // current value to R4
LDR R2, =0xFC20005C // edge capture address
LDR R3, [R2] // edge capture value
CMP R3, #0b1000 // check if key 3
BNE KEY0 // if not key3, must be key0
CMP R4, #0 // check if curr_value 0
BEQ ENDISR // branch if 0
SUB R4, R4, #1 // decrease value
STR R4, [R5] // store
B ENDISR

KEY0:
// code for key 0

ENDISR:
POP {R1-R5}
MOV PC, LR
```

## Polled IO with Timer

```
.text
.global_start

_start:
LDR R0,=0xFFFEC600 //load base address of timer
LDR R1,=200000000 //200 million -> starting time
STR R1,[R0] //load time into base address of timer
MOV R1,#0b111
STR R1,[R0,#8] //A=E=1
MOV R4,#0 //seconds
MOV R5,#0 //minutes
MOV R6,#0 //hours

POLL:

LDR R1,[R0,#12] // Load F bit. It's offset
                   from base address by #12
CMP R1,#0
BEQ POLL //If the F bit is 0, then continue polling.
STR R1,[R0,#12] // Plug 1->F bit to reset F bit.
                   R0 has base address
ADD R4,R4,#1 //Increment seconds
CMP R4,#60 //Have we hit 60s?
BNE POLL //If we haven't continue polling.
MOV R4,#0 //If we have hit 60, move 0 into seconds
ADD R5,R5,#1 //And then increment minutes
CMP R5,#60 //Then we check if minutes have hit 60
BNE POLL //If they haven't, continue polling
MOV R5,#0 //If they have, set the minutes to 0
ADD R6,R6,#1 //And then increment hours
CMP R6,#24 //Then we check if the hours have hit 24
BNE POLL //If they haven't, continue polling
MOV R6,#0 //If they have, set the hours to 0
B POLL //Continue polling
```

## Exception Vector Table

```
0x0 Reset exception (branch to _start)
0x18 B to IRQ_HANDLER
```

## Find Sum with Recursion

```
.global _start
_start:
    LDR SP, =0x20000
    LDR R4, =N
    LDR R0, [R4]
    MOV R1, #0
    BL FINDSUM
    ADD R1, R1, R0
END: B END
FINDSUM:
    PUSH {R0, LR}
    CMP R0, #2
    BLT RETURN
    SUB R0, R0, #1
    BL FINDSUM
    ADD R1, R1, R0
RETURN:
    POP {R0, PC}
.data
N: .word 5
```

## Fibonacci with Recursion

```
.data
    N: .word 10
.text
.global _start
_start:
    LDR SP, =0x20000
    LDR R4, =N
    LDR R0, [R4]
    MOV R1, #0
    MOV R2, #0
    BL FIB
END: B END
FIB:
    PUSH {R0, R2, LR}
    CMP R0, #2
    BGE RECUR
    MOV R1, R0
    POP {R0, R2, PC}
RECUR:
    SUB R0, #1
    BL FIB
    MOV R2, R1
    SUB R0, #1
    BL FIB
    ADD R1, R2
    POP {R0, R2, PC}
```