ECE253 Final Cheatsheet

Author: your mother

Boolean Algebra

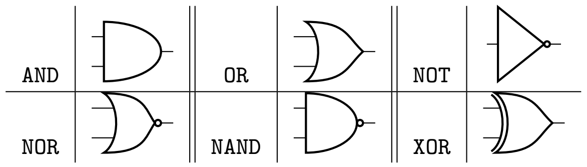
De Morgan's Theorem tells us

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

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

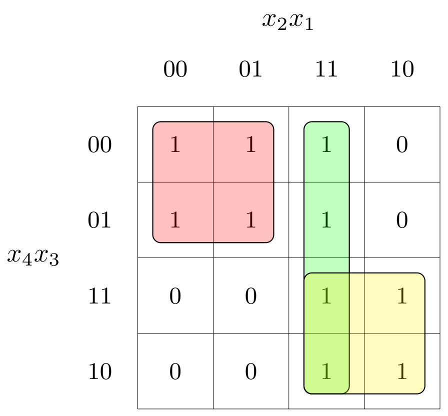
The cost of a logic circuit is given by

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

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

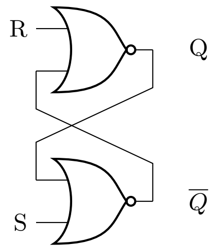
1. Generate all prime implicants for given function f
2. Find the set of essential prime implicants
3. Determine the nonessential prime implicants that should be added.

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

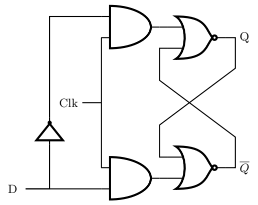
Sequential circuits depend on sequence of inputs. A **SR Latch** are cross-coupled `NOR` gates.



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

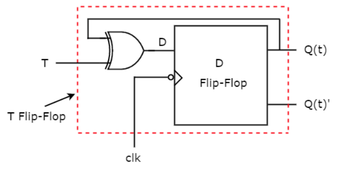


Clk	$Q(t+1)$
0	Q(t)
1	D

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)`

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 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

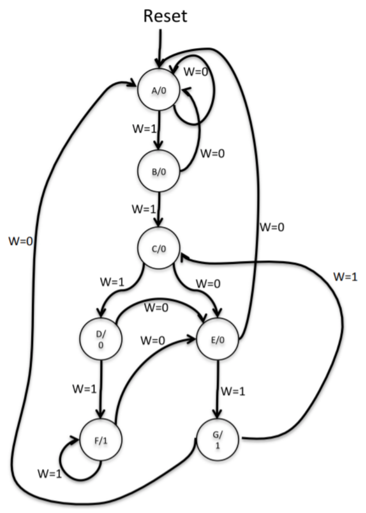
- Active High/Low: Resets when Signal is 1/0
- Synchronous High/Low: Resets during positive/negative edge

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
$\vdots$	$\vdots$	$\vdots$	$\vdots$
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 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

Registers

- x0 (zero): Hardwired zero, x1 (ra): Return address, x2 (sp): Stack pointer
- x5-x7, x28-x31 (t0-t6): Temporary (caller-saved)
- x8-x9, x18-x27 (s0-s11): Saved (callee-saved)
- x10-x17 (a0-a7): Function args/return values
- x3 (gp): Global pointer, x4 (tp): Thread pointer

Instructions

Let `a0=1`, `a1=2`, `a2=0b1010`.

Instruction	Example	Result
ADDI	`addi a3, zero, 3`	`a3 = 3`
ADD	`add a3, a0, a0`	`a3 = 1 + 1`
SUB	`sub a3, a0, a0`	`a3 = 1 - 1`
MUL	`mul a3, a0, a0`	`a3 = 1 * 1`
SLLI	`slli a3, a2, 1`	`a3 = 0b10100`
SRLI	`srli a3, a2, 1`	`a3 = 0b0101`
SRAI	`srai a3, a2, 1`	`a3 = 0b1101`
AND	`and a3, a1, a0`	`a3 = (1 and 2) = 0`

Memory Stuff

- `JAL`: Jump and Link - stores return address in `ra` (x1)
- `JALR`: Jump and Link Register - indirect jump
- Stacks: Manual push/pop: `addi sp, sp, -8` then `sw a0, 0(sp)`
- Each instruction is 4 bytes (32-bit) in RV32.

Load and Store

- `lw a0, offset(a1)`: Load word; `sw a0, offset(a1)`: Store word
- `la a0, label`: Load address (pseudo-instruction)
- `lb/lbu` - load byte (signed/unsigned), `lh/lhu` - load halfword
- `sb` - store byte, `sh` - store halfword

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
```