ECE253 Midterm Cheatsheet

Boolean Algebra

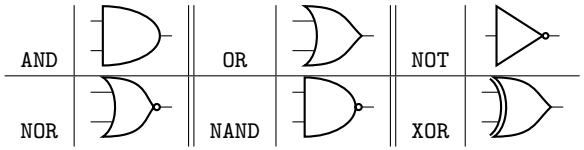
De Morgan's Theorem tells us

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

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.

Some *definitions:*

- **Literal:** variables in a product term: $x_1\overline{x_2}x_3$ has three literals.
- **Implicant:** a product term that indicates the input valuation(s) for which a given function is equal to 1.
- **Prime Implicant:** an implicant that cannot be combined into another implicant with fewer literals. *They are as big as possible.*
- **Cover:** A collection of implicants that account for all valuations for which function equals 1.
- **Essential Prime Implicant:** A prime implicant that includes a minterm not included in any other prime implicant. *They contain at least one minterm not covered by another prime implicant.*

In the above example, $\overline{x_2} \cdot \overline{x_4} + x_2x_1 + \overline{x_4}x_2$ are prime implicants.

Minimization Procedure

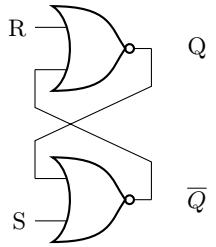
1. Generate all prime implicants for given function f
2. Find the set of essential prime implicants
3. Determine the nonessential prime implicants that should be added.

Common Logic Gates

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

RS Latch

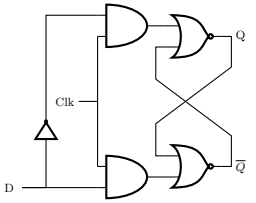
Sequential circuits depend on sequence of inputs. A **SR Latch** are cross-coupled `NOR` gates.



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

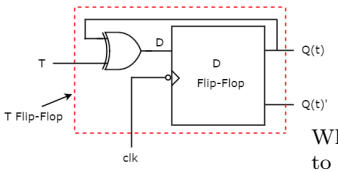
Where the Clk = 1 cases refer to `retain`, `reset`, `set`, and last one is not used.

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



T	Q	$Q(t+1)$
0	0	0
1	0	1
0	1	0
1	1	0

Where the Clk = 1 cases refer to `retain`, `reset`, `set`, and last one is not used.

Registers: Multiple flip flops connected together.

Verilog + Advanced Sequential Circuits

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.
- `nonblocking assignments <=` executed in parallel.

```
a <= b;
c <= a;
```

Here, `c` would take on the old value of `a`, but if blocking was used, it would take on the value of `b`.

Minimal Example

```
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'b001: Out = Input[1]; // case 1
     3'b010: Out = Input[2]; // case 2
     3'b011: Out = Input[3]; // case 3
     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 all signals and add some signals to
# waveform window
log {/*}
# add wave {/*} would add all items in
# top level simulation module
add wave {/*}
# first test case
#set input values using the force command
# signal names need to be in {} brackets
force {SW[0]} 0
force {SW[1]} 0
force {SW[9]} 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.

Resets

- Active High: Resets when signal is 1.
- Active Low: Resets when signal is 0.
- Synchronous: Resets when signal changes from 0 to 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 $2k$.

Finite State Machines

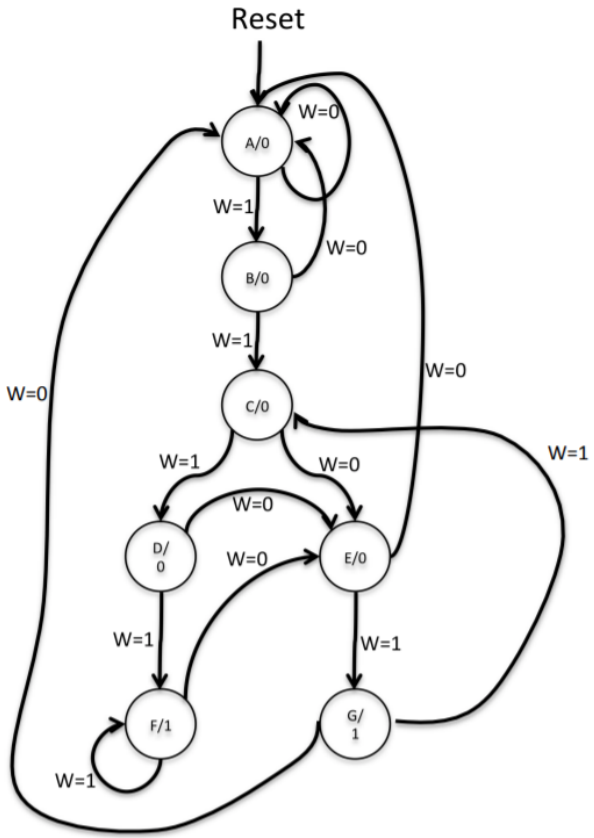
Finite state machines are similar to user interfaces.

Steps

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

Step 1: State Diagram Example

Source: Lab 6. The notation `C/0` means that at state `C` the output is 0. The user input is W



Step 2: State Table Example

Present State	Next State $W=0$	Next State $W=1$	Output (z)
A	A	B	0
B	A	C	0
C	E	D	0
D	E	F	0
E	A	G	0
F	E	F	1
G	A	C	1

Step 3: State Assignment Example

Using **one-hot encoding:**

- Choose the number of flip flops: 7 (since 7 states)
- Choose state codes:
  - `A` = 0000001, `B` = 0000010, ..., `G` = 1000000

Alternatively, we can use 3 flip flops and represent the state codes as $000, 001, 010, 011, 100, 101, 110$. One-hot encoding makes designing the circuit simpler and easier to debug, but this is more compact (so we will use this for the cheat sheet).

Step 4: State-Assigned Table Example

By convention, we use lowercase y for the input, and uppercase Y for the 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
010	100	011	0
011	100	101	0
100	000	110	0
101	100	101	1
110	000	010	1

Step 5: Synthesize Example

We first write boolean algebra expressions for the outputs $Y_n = f(y_1, y_2, y_3, W)$ 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 $f(y_1, y_2, y_3, W)$ and **loops back** to Y_n.

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

Execution in Verilog

The following makes use of one-hot encoding.

```
module part1(Clock, Resetn, w, z, CurState);
   input Clock, Resetn, w;
   output z;
   output [3:0] CurState;

   reg [3:0] y_Q, Y_D;

   localparam A = 4'b0000, B = 4'b0001, C = 4'b0010,
     D = 4'b0011, E = 4'b0100, F = 4'b0101, G = 4'b0110;

    //State table
    always@(*)
    begin: state_table
        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
            // ...
            G: begin
                   if(!w) Y_D = A;
                   else Y_D = C;
               end

            default: Y_D = A;
        endcase
    end // state_table

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

    // 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-R7: 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): Holds address of next instruction to execute

Instructions

Let $r0 = 1, r1 = 2, r2 = 3$.

Instruction	Examples	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, #0b1010`	r3 = #0b0100
LSR	`lsr r3, #0b1010`	r3 = #0b0101
ASR	`lsr r3, #0b1010`	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. **L**ast **I**n **F**irst **O**ut.
- Each line 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`: `Ra` contains the address to the first element of the input variable.

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`	$\neq$	`GT`	$>$
`GE`	$\geq$	`LE`	$\leq$		

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.
3. Initialize SP for SVC mode.
4. Configure GIC to enable interrupts (code provided)
5. Enable Interrupt Generation in I/O Device.
6. Set I=0 in CPSR.
7. PRovide IRQ Handler code which queries GIC to determine source of interrupt.
8. Provide interrupt service routines (ISRs), i.e. `KEY_ISR`
9. The interrupt handler must clear interrupt from GIC.

Steps 1-5

```
.global _start
_start:
    // 11 tells us to disable interrupts (IRQ mode)
    mov R0, #0b11010010

    MSR CPSR, R0 // Now in IRQ mode
    LDR SP, = 0x20000 // IRQ SP

    // disables interrupts (SVC mode)
    MOV R0, #0b11010011
    MSR CPSR, R0 // Back in SVC mode

    // SVC SP, different stack pointer than before
    LDR SP, =0x3FFFFFFC
    BL CONFIG_GIC // Given to you
    BL CONFIG_KEYS // enable keys to interrupt
    // Add additional subroutines to config others

    // enable interrupts
    MOV R0, #0b01010011

    MSR CPSR, R0
    // Your Program
```

Steps 6,8

```
.global IRQ_HANDLER
IRQ_HANDLER:
    /* Save R0-R3 b/c subroutines called from
    here may modify w/o saving or restoring */
    /* Save R4,R5 b/c we modify here */
    PUSH {R0-R5, LR}
    /* Read the ICCIAR from the CPU interface */
    LDR R4, =0xFFFEC100 // GIC Address
    LDR R5, [R4, #0x0C] // Read the Interrupt ID
PRIV_TIMER_CHEK:
    CMP R5, #29 // check for timer interrupt
    BNE KEYS_CHECK

    BL PRIV_TIMER_ISR
    B EXIT_ISR
KEYS_CHECK:
    CMP R5, #73 // check for keys
UNEXPECTED:
    BNE UNEXPECTED // if not recognized, stop here
    BL KEY_ISR
EXIT_IRQ: // return from KEY_ISR
    // write to end of interrupt register (ICCEOIR)

    STR R5, [R4, #0x10] // Clear interrupt from GIC
    POP {R0-R5, LR} // restore registers

    // return to where I was in main program
    SUBS PC, LR, #4
```

Step 7 Example

```
.global KEY_ISR
KEY_ISR:
    LDR R0, =0xFF200050 // base address of keys
    LDR R1, [R0, #0xC] // edge capture reg
```