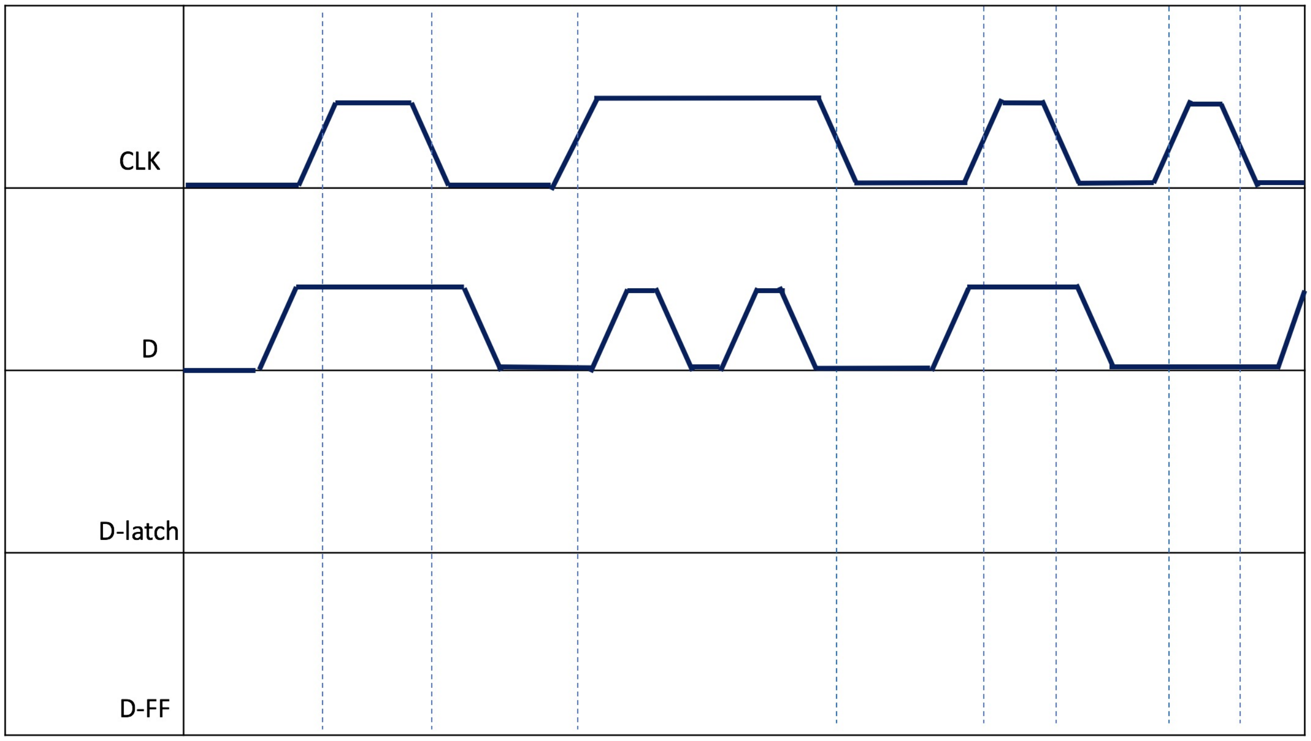
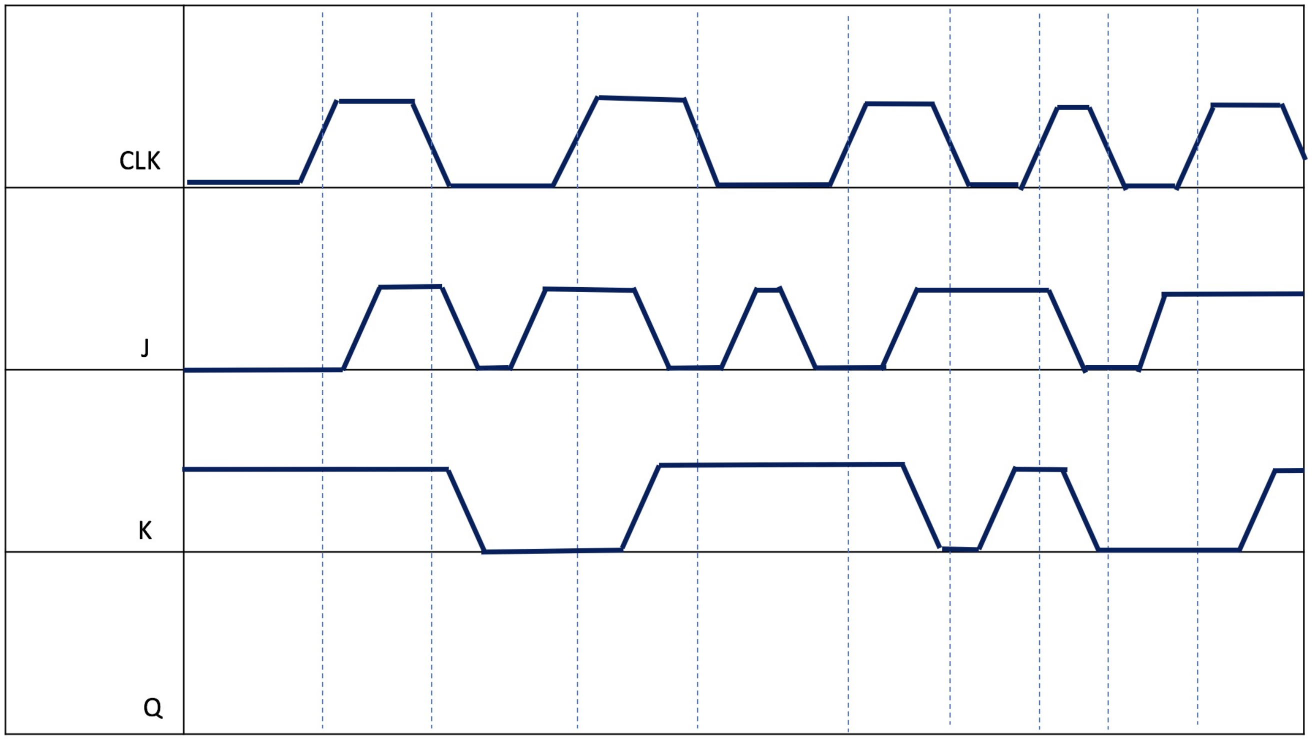
CSE140 - HW #4

Due Monday May 22, 11:59PM

In this homework, we practice the analysis and synthesis of sequential networks. For the first problem, we draw the timing diagram of the latch and flip-flop. For the second problem, we translate a pattern recognizer into state diagrams and state tables. For the third problem, we analyze the behavior of a sequential circuit. For the fourth problem, we analyze a specific use of a SR latch. For the fifth problem, we design the circuit from a given description. For the last problem, we use BSV to implement a branch predictor commonly used in microprocessors.

1. Timing Diagram of Latch and Flip-Flop: Given the input waveforms shown below, sketch the outputs.





2. A sequential network has one binary input $x(t)$ and one binary output $y(t)$. The network produces $y = 1$, whenever input pattern $x(t-2, t)$= 101 or 010. Otherwise, the output $y = 0$.

2.1 Design the system as a Mealy machine with a minimal number of states.

(i) Draw the state diagram.

(ii) Write the state table.

2.2 Design the system as a Moore machine with a minimal number of states.

(i) Draw the state diagram.

(ii) Write the state table.

3. Given the state table as described in zyBook 3.8.2, implement the machine with two JK flip-flops

(i) Write the excitation table.

(ii) Draw the state diagram.

4. For the logic diagram of SR latch in subsection 3.1.2 zyBook, we replace the two NOR gates with two NAND gates. Explain the relation between inputs and outputs. How would you make use of this circuit?

5. A state machine has one input $x(t)$ and two-bit state $(Q_1(t), Q_0(t))$. The machine is described by the following state equations.

$$Q_1(t+1) \;=\; Q_0(t)' + \; x(t),$$

$Q_0(t+1) \;=\; Q_1(t)x'(t).$
$y(t) = Q_1(t)Q_0(t)$

Use two JK flip-flops and a minimal two-level NAND network to implement the machine.

(i). Write the state excitation table and draw the state diagram.

(ii). Show your derivation (K maps) and draw the logic diagram.

6. Branch predictor.

Below is a BSV module which represent a simple state transfer diagram.

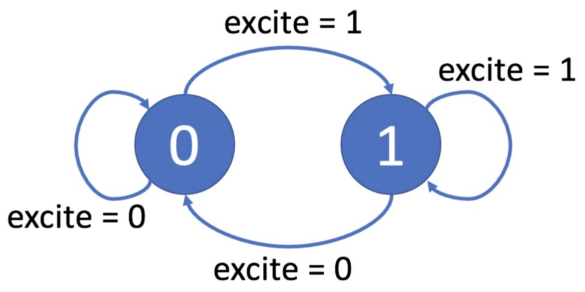
```
module Fsm;
        Reg#(Bit#(1)) state <- mkReg(0);
        Reg#(Bit#(1)) excite <- mkReg(0);

        rule rule1 (state == 0 );
                if (excite == 0)
                        state <= 0;
                else
                        state <= 1;
        endrule

        rule rule2 (state == 1 );
                if (excite == 1)
                        state <= 1;
                else
                        state <= 0;
        endrule

        method ActionValue#(Bit#(1)) get_state( Bit#(1) ext);
                excite <= ext;
                return state;
        endmethod
endmodule
```

The respective FSM is drawn as follows.



Pipelining is a fundamental technology in modern microprocessors. Pipeline increases the throughput of instructions. However, in case of a branch instruction, pipeline loses its power since you do not know which instructions to fetch.

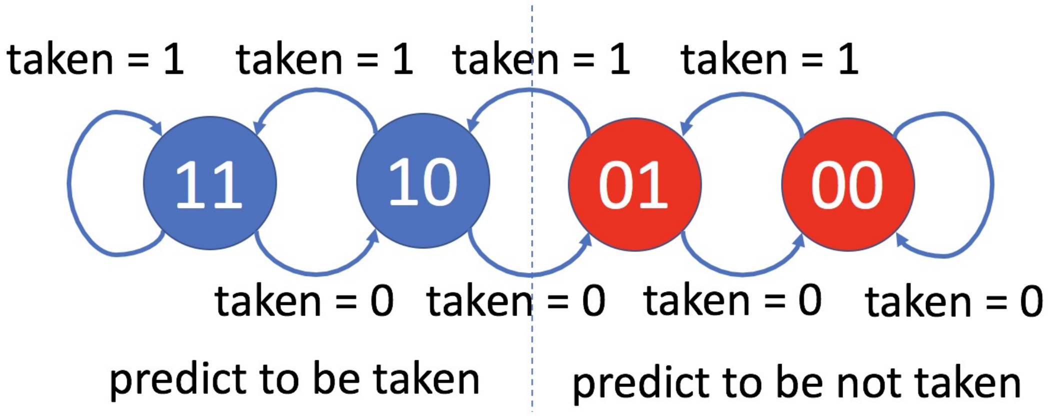
A good way to fix this issue is to use a branch predictor. Branch predictor returns single bit indicating whether the branch is taken nor not taken. After the instruction is executed, the actual result (taken/not taken) is sent to the branch predictor to change its state for future prediction. Below is the interface of a common branch predictor.

```
interface BranchPredictor;
    method Bit#(1) get_prediction;
    method Action set_result(Bit#(1) result);
endinterface
```

`get_prediction` returns one bit prediction indicating the branch is taken (`return 1`) or not taken (`return 0`).

After the branch instruction is executed, we know actually the branch is taken or not taken, and we get back this result to the branch predictor by calling `set_result`, which passes one bit value indicating whether the branch is actually taken (1) or not taken (0). With that information the branch predictor may change its internal state so that it gets better accuracy in the following predictions.

A common implementation of a branch predictor is to use a 2-bit FSM. Below is the state transfer diagram of the branch predictor to be implemented.



When `get_prediction` is called and the state is `11` or `10`, the branch predictor will predict the branch to be taken (`1`), otherwise it will predict the branch to be not taken (`0`). After the branch predictor gets back the results when `set_result` is called, a state transfer will occur based on whether the branch is taken or not taken.

- Complete the following code so that the branch predictor works.

```
module mkBp(BranchPredictor);
    Reg#(Bit#(2)) state <- mkReg(0);
```

```
Reg#(Bit#(1)) taken <- mkReg(0);
Reg#(Bit#(1)) transfer <- mkReg(0);

rule rule1 (transfer == 1 && state == 'b11);
    if ( taken == 1)
        state <= 'b11;
    else
        _______________________;
    transfer <= 0;
endrule

rule rule2 (transfer == 1 && state == 'b10);
    if ( taken == 1)
        _______________________;
    else
        _______________________;
    transfer <= 0;
endrule

rule rule3 (transfer == 1 && state == 'b01);
    if (taken ==1)
        _______________________;
    else
        _______________________;
    transfer <= 0;
endrule

rule rule4 (transfer == 1 && state == 'b00);
    if (taken ==1)
        _______________________;
    else
        _______________________;
    transfer <= 0;
endrule

method Action set_result(Bit#(1) result) if (transfer == 0);
    transfer <= 1;
    taken <= result;
endmethod

method Bit#(1) get_prediction;
    return __________________________;
endmethod
```

```
endmodule
```

- What is the prediction accuracy (the percentage of the correction predictions in all the predictions) of a branch predictor that always predict the branch to be not taken (`N`) with the following actual sequence `NNNTTTTNTTNTNTNTNNNTN`? (N = not taken, T = taken). Please explain how you get the number.
- What is the prediction accuracy for the branch predictor implemented in this question in BSV with the sequence

  `NNNTTTTNTTNTNTNTNNNTN`?

(The codes are available on ieng6-24x (x=0-8) server with
in the path `/home/linux/ieng6/cs140s/public/hw4`.)