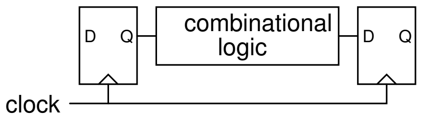
RTL Design

This lecture describes the most common hardware design method: register transfer level (RTL) design.
After this lecture you should be able to: compute setup and hold times from clock period and min/max propagation delays; determine the level of a circuit design; select an appropriate design level; convert an algorithm into an RTL design described in synthesizable Verilog.

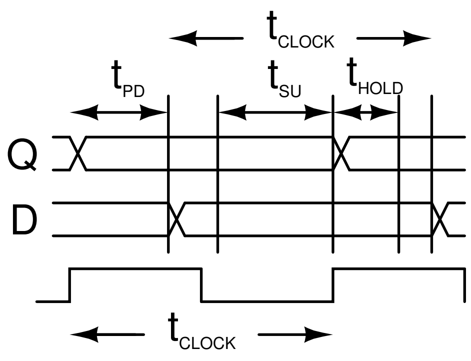
Synchronous Design

A synchronous logic circuit is one where all flip-flops and registers use the same clock signal. The clock does not pass through gates (e.g. it's not disabled or "gated") and is not generated by other logic circuits (e.g. as in a ripple counter).

Practically all digital design today is synchronous. The use of a common clock means that all paths between flip-flops have the following structure:



This makes it practical for EDA (Electronic Design Automation) tools to determine the maximum clock rate. For the setup and hold time requirements to be met the clock period must be greater than the maximum propagation delay through the combinational logic plus the required flip-flop setup time:



The amount by which this requirement is exceeded is known as the "slack" time.

Exercise 1: Mark the slack time in the diagram above.

Exercise 2: If the clock period (t_{CLOCK}) is known, how are the minimum setup (t_{SU}) and hold (t_{HOLD}) times related to the minimum and maximum propagation delays (t_{PD})?

Levels of Design

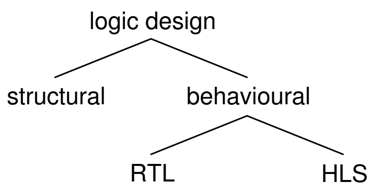
Digital logic circuits can be designed at various levels of abstraction.

At the lowest level, *structural*, we specify circuit elements such as registers, adders, etc. and how they are to be interconnected. This is often done with schematic capture but can also be done with an HDL. For example, using Verilog we can instantiate modules that represent components from a library and connect their ports using named signals.

At a higher level of abstraction, *behavioral*, we specify the desired behaviour in the form of variables, expressions and procedures and allow the logic synthesizer to select the required circuit elements and how they are to be connected.

A behavioral design can include a clock and specify the operations performed on each clock cycle. This is called Register Transfer Level (RTL) design. This is the most common approach used today and the one we will cover in this course.

We can also specify a behavioral model without a clock. The synthesizer must then decide how to schedule the steps of the algorithm. For example whether the operations in a particular loop should be done sequentially or if the loop should be "unrolled" and its operations done in parallel. The designer may provide "hints" to the synthesizer to help achieve the required design objectives. This level of design is called High Level Synthesis (HLS) and the language used is often C or C++. The following diagram summarizes the different levels at which we commonly design digital circuits:

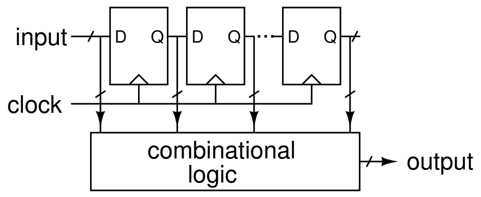


Exercise 3: Which of these requires the most time and effort? Least? Which gives the designer most control over the cost and performance of the design? Least? Which produce(s) designs that are portable to different implementation technologies (FPGAs, ASICs)? Which allow the same design to meet a variety of speed/area targets?

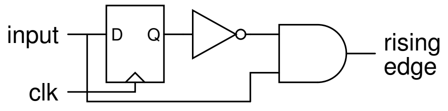
Sequential Logic

Sequential logic circuits have outputs that are a function of previous inputs as well as current inputs. They are thus said to have "memory."

In principle, any sequential logic circuit can be built from a shift register that stores previous inputs and a lookup table (combinational logic) that computes the output based on these previous inputs:



In some cases, such as a sequence detector, this direct form is useful. For example, the following circuit detects a rising edge on the input signal:



Exercise 4: How would you design a falling-edge detector? For how many clock cycles is the detector output true?

However, in most cases we can greatly simplify the implementation by storing only a summary of previous inputs. For example, we could store a count of the number of times a particular input condition occurred rather than the previous inputs themselves.

Algorithmic State Machines

Certain sequential logic circuits are most conveniently described as finite state machines. At any time a state machine (SM) can be in only one state. Inputs can cause the SM to move from one state to another depending on the current state and the state transition rules.

In principle any sequential logic circuit could be described as a single state machine, with its state being the contents of all of its registers.

In practice, the design of complex digital circuits is partitioned into relatively simple state machines that control the operations of other logic circuits, including registers that are considered to store data rather than "state." These types of state machines are sometimes called "Algorithmic" State Machines (ASMs).

State Machine Design

Since we are designing state machines to implement algorithms, our starting point is the algorithm. These can be described as executable code (e.g. a C program), pseudo-code or a flowchart.

Here's the pseudo-code for an algorithm to find the smallest value in an array of 100 3-digit values:

```
min=999
for ( i=0 ; i<100 ; i++ ) begin
   if ( x[i] < min ) min = x[i]
end
stop
```

One difference between an algorithmic description for a computer program and that for a hardware implementation is that hardware can operate in parallel. All operations that do not depend on the result of a previous operation can be carried out simultaneously.

In this example, the two initializations (`min=999` and `i=0`) can be done in parallel as can the possible assignment to `min` and incrementing of `i` (`min=x[i]` and `i++`).

Another difference is that branching (state transitions) can be carried out at the same time as other computations and do not require a separate state.

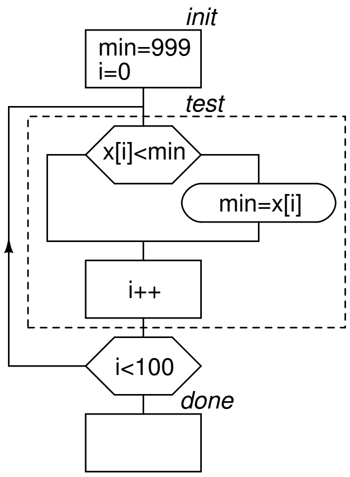
To design the state machine we define one state for each group of operations that need be carried out. The state transition conditions are defined by the sequence of operations required to implement the algorithm.

In this case we have three states (e.g. `init`, `test` and `done`). The transition from `init` to `test` is unconditional while the transition from `test` to `done` is defined by the condition `!(i<100)`.

The condition `x[i]<min` does not control a state transition but instead controls the operation of the datapath that computes the value of `min` which can

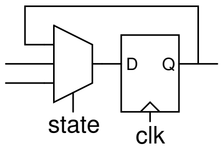
be loaded with 999, `min` or `x[i]`. This is a "Mealy" output of the state machine because its value depends on an input (the comparison result) and not solely on the state.

A diagram that includes the state transitions as well as the operations of the datapath is called an ASM chart. For this algorithm it might look as follows:[1]:



Datapath Design

The datapath consists of one register for each variable required by the algorithm. The combinational logic driving each of these registers is the same: a multiplexer that selects one of various combinational logic functions depending on the outputs of the state machine.



The algorithm above requires two registers, `min` and `i`. The multiplexer at the input of the `min` register would select the constant 999, `min` or `x[i]`. The multiplexer for `i` would select 0, `i+1` or `i` (in the done state).

Verilog Implementation

The ASM can be described in Verilog using `always_ff` block(s) for the state and datapath registers. An enumerated variable allows the use of symbolic names for the states.

The combinational logic that defines the next state and the next register values is defined in `always_comb` block(s).

The code (for a 4-element array) is shown below. Simulation results are shown in Figure 1 and the synthesized circuit in Figure 2.

```
module ex10 ( output logic [9:0] min,
              input logic reset, clk ) ;

   logic [9:0] min_next ;
   logic [7:0] i, i_next ;

   enum logic [1:0] { init, test, done }
        state, state_next ;
   logic [9:0] x[4] = '{ 700, 800, 30, 900 } ;

   // controller state
   always_ff@(posedge clk)
      if ( reset ) state <= init ;
      else         state <= state_next ;

   // datapath registers
   always_ff@(posedge clk) begin
      i <= i_next ;
      min <= min_next ;
   end

   // datapath and next-state logic
   always_comb begin
      min_next = min ;          // defaults
      i_next = i ;
      state_next = state ;
      case(state)
        init: begin
           min_next = 999 ;
           i_next = 0 ;
           state_next = test ;
        end
        test: begin
           if ( x[i] < min ) min_next = x[i] ;
           i_next = i+1 ;
           if ( i_next >= 4 ) state_next = done;
        end
        default: ;
      endcase
   end
endmodule
```

Note the following:

- signals are defined for both the "`_next`" next-state and the current-state values. This gives

[1]Don't worry about the details, we won't use them in this course.

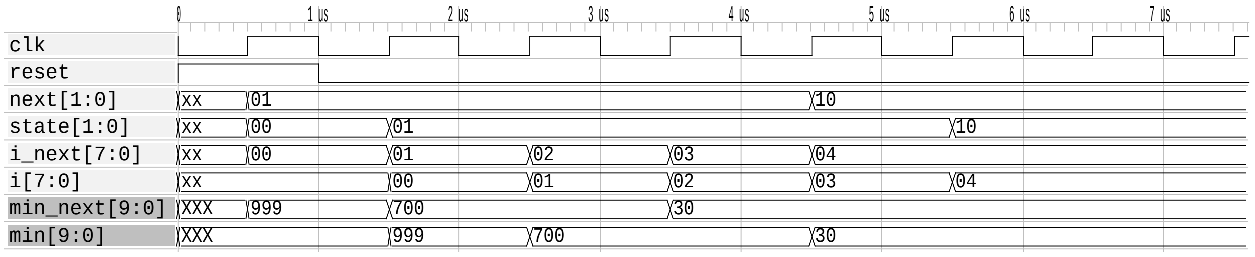


Figure 1: Simulation results.

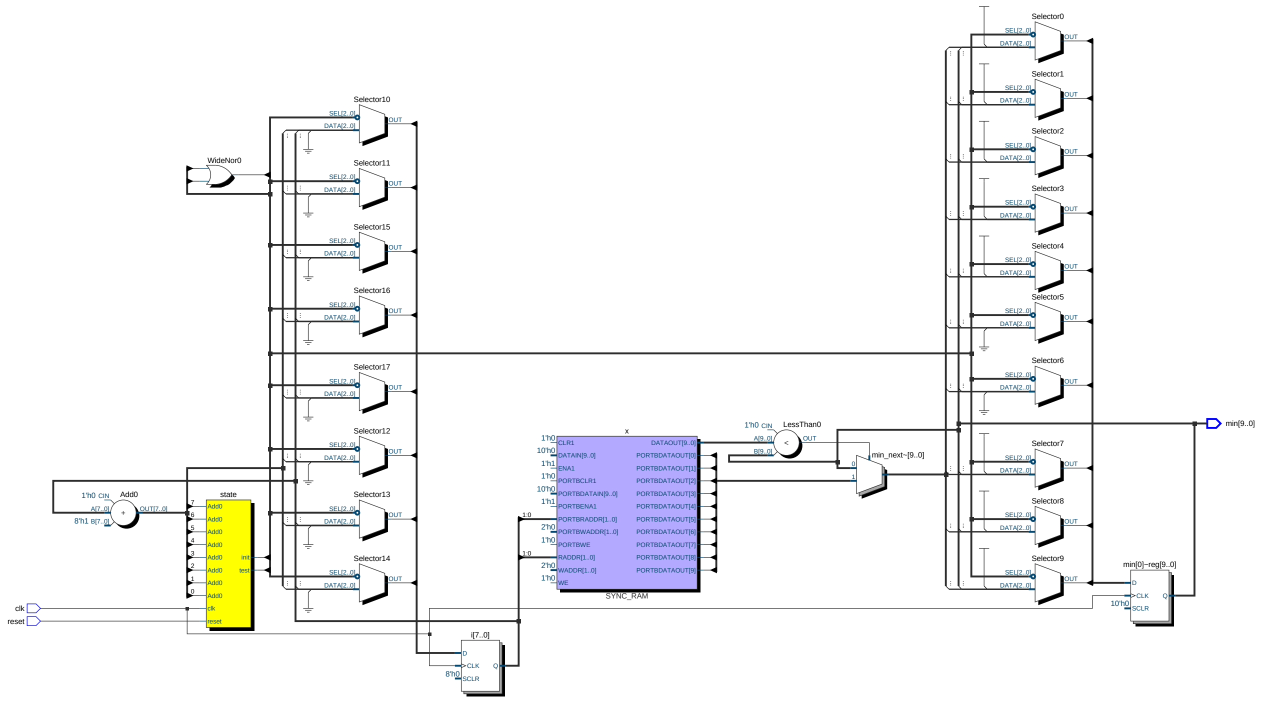


Figure 2: Schematic.

access to both the input (combinational or "Mealy") and output (registered or "Moore") values of each register. This is a common idiom in HDLs.

- default values are assigned at the top of the `always_comb` block. This ensures that each signal is always assigned and that this block generates combinational logic. In many cases this also simplifies the code.

Exercise 5: Find the following blocks in the schematic: the `min` register, the `x[]` memory block, the `i` register, the `state` state machine.

Exercise 6: Each number in the Fibonacci sequence is the sum of the previous two. Write an algorithm to compute the values of the sequence that are less than 100 and stop. Ignore the first two (both 1). What registers are required? What states? Draw the state transition diagram for the controller. Write the Verilog code.