RTL Design

This lecture describes register transfer level (RTL) design, the most commonly-used digital hardware design method for complex digital circuits.
After this lecture you should be able to: determine the circuit design method being used, select an appropriate design method, and use RTL design to convert an algorithm into synthesizable Verilog.

Levels of Design

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

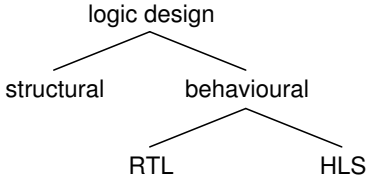
At the lowest level, *structural*, we specify circuit elements such as registers, adders, etc. and how they are to be interconnected. This is often done with schematic capture but can also be done with an HDL. For example, using Verilog we can instantiate modules that represent components from a library 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 sequential processes 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 can 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 primarily use 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 input to the synthesizer is often in C instead of an HDL.

The following diagram summarizes the different levels at which we commonly design digital circuits:



Exercise 1: 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?

Algorithmic State Machines

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 thought of as storing data rather than "state." Such a state machine is sometimes called an Algorithmic State Machine (ASM) or ASMD ("ASM with Data").

ASM Design

Since we are designing state machines to implement algorithms, our starting point is the algorithm. Algorithms can be described using flowcharts, pseudocode or executable code such as a C program. The executable description has a number of advantages: it can be written by subject-matter experts who may not be familiar with HDLs or hardware design, it will run faster than a simulation of an HDL description, and the software can be used as an independent reference against which the hardware can be validated.

As example, we'll design hardware to find the minimum value stored in a memory. Here's the C code for an algorithm to find the smallest value in an array of four 3-digit values:

```
short imin ( short x[] )
{
   short min=999 ;
```

```
   for ( int i=0 ; i<4 ; i++ )
      if ( x[i] < min ) min = x[i] ;
   return min ;
}

#include <stdio.h>
void main () {
   short x[]={700,800,30,900} ;
   printf ( "%d\n", imin(x) ) ;
}
```

Registers. The first step is to identify the variables required by the algorithm. Each variable will become a register in the RTL description.

Unlike a C program where variables are limited to certain sizes (e.g. `char`, `short`, `int`, or `long`), we can use any number of bits for each register. Some object-oriented languages have libraries that allow writing bit-exact, arbitrary-precision descriptions of algorithms that are executable.

Exercise 2: List the registers required to implement the minimum-finding algorithm above.

Computations. The second step is to identify the computations that are required by the algorithm. These will typically correspond to assignments to variables.

Exercise 3: List the computations required to implement the minimum-finding algorithm above.

Sequencing. We must then define a state machine to perform the computations in the correct sequence. Each computation, or group of computations, is assigned to a different state. The state transition conditions are defined by the looping and if/else constructs in the algorithm.

Unlike a computer program, the hardware implementation of an algorithm can schedule multiple computations simultaneously. However, this may require more hardware resources or longer propagation delays before the result of the operation is available.

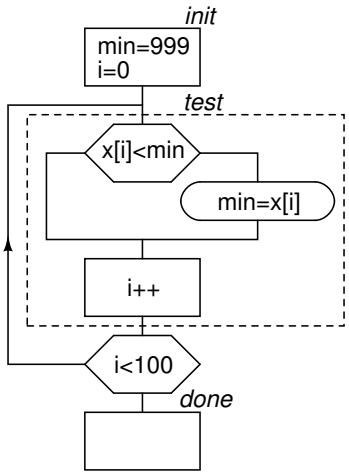
For example, the two initializations (`min=999` and `i=0`) could be done in parallel as could the conditional assignment to `min` and the incrementing of `i` (`min=x[i]` and `i++`).

It's also possible that computations required for branching (`i<100` and `x[i]<min`) could be done in parallel with other computations – again, assuming sufficient hardware resources.

In this case we can define three states which we'll call `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 computation of `min` which can be loaded with `999`, `min` or `x[i]`.

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]:



We can also describe the ASM in the form of a chart with one row for each register and one columns for each state. For the example above it might look like:

register	init	test	done
i			
min			

Exercise 4: Fill in the table.

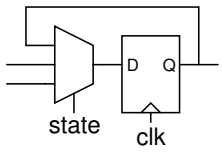
Interfaces. The C function has arguments and return values which seem to correspond to the inputs and outputs of an ASM. However, unlike a C function, hardware input and output signals can change during the execution of the algorithm. In addition, there could be multiple ASMs executing concurrently and there must be ways to synchronize their operation. We will study these later. Finally, the ASM needs clock and reset signals like any other logic circuit.

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

ASM Implementation

In RTL terminology, the hardware that implements computations is called the "datapath" while the hardware that controls the sequence of computations is called the "controller."

The datapath consists of one register for each variable. The input of each register is a multiplexer. The inputs to the multiplexer are each of the possible expressions that can be assigned to that variable. The multiplexer is controlled by the controller – the controller selects the value to be assigned to each variable in each step of the algorithm (i.e. in each controller state):



Exercise 5: Draw the datapath for the variables `i` and `min`.

ASM in Verilog

The ASM can be described in System Verilog using an `always_ff` block to implement a state register and each datapath register. An enumerated variable allows the use of symbolic names for the states.

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

The minimum-finding 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 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 6: Find the following blocks in the schematic: the `min` register, the `x[]` memory block, the `i` register, the `state` state machine.

Exercise 7: 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 computations? Draw the state transition diagram for the controller. Write the System Verilog code.

Converting a C program to Verilog would appear to be a simple mechanical operation. However, in more complex designs it's possible to trade speed (the

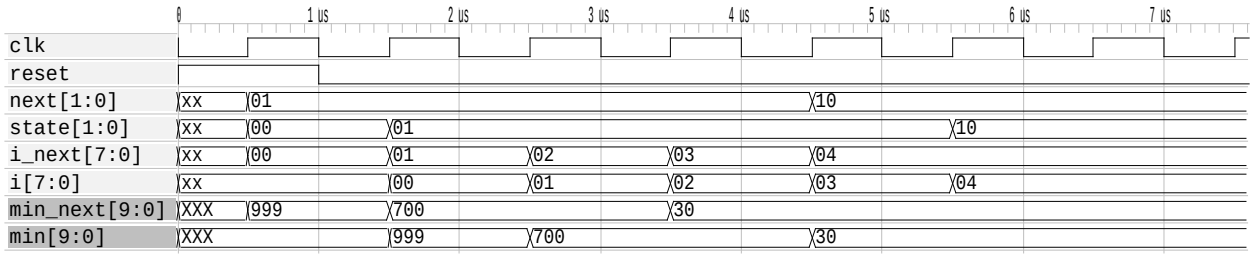


Figure 1: Simulation results.

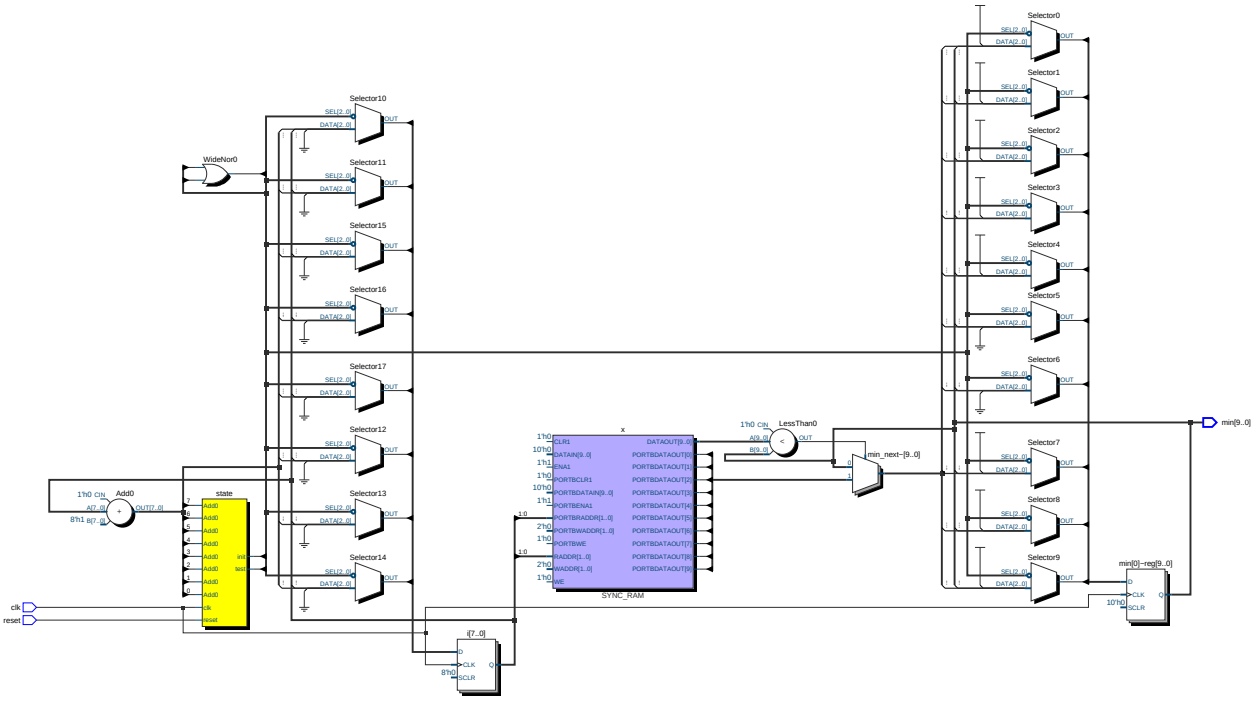


Figure 2: Schematic.

maximum possible frequency of the clock) and area (the amount of logic required) when implementing an algorithm.

The RTL design method requires the designer to explicitly make speed/area tradeoffs by allocating more or less parallel logic and scheduling the computations into more or fewer controller states.

At one extreme would be a design using a programmable processor that has a minimum of combinational logic (e.g. an ALU) but each computation or state change (branch) typically takes multiple clock cycles. At the other extreme are large combinational logic circuits that can implement an algorithm (e.g. multiplication) in one clock cycle.