An example of RTL

Madhav P. Desai
EE Department, IITB
email: madhav@ee.iitb.ac.in

January 8, 2016

Consider the following specification of a digital system. Read 128 2-bit numbers and find out the most commonly occuring number among them.

To implement this, we could use the following algorithm:

```
// algorithm
compute_max()
{
   int max_count = 0;
   int max_count_index = 0;
   int num_count[4];
   for (i=0; i < 4; i++)
   {
      num_count[i] = 0;
   }

   for (i=0; i < 128; i++)
   {
      int j = read(); // from the external world.
      num_count[j] += 1;
      if(num_count[j] > max_count)
      {
         max_count = num_count[j];
         max_count_index = j;
      }
   }

   write(max_count_index);   // to the external world
}
```

Suppose we wish to convert this algorithm into a digital circuit (that is, the digital circuit implements the algorithm). The following steps are clearly necessary:

- We must define the protocol by which the system will interact with the environment.

- The algorithm must be expressed using sequences of operations that can be implemented in the digital system. These operations are basically of the form $a = f(b, c, ..)$, where a, b, c are values stored in the system and f is some function computed on these values. The assignment $a = f(b, c, ..)$ can be viewed as a register transfer. In the synchronous paradigm, this is equivalent to

  ```
  a(k+1) = f(b(k),c(k), ...)
  ```

- From the expression of the algorithm it should be possible to infer **a** gate-level description of the system in a routine manner[1].

1 The interface between the system and the environment

Let us assume that the system will have the following ports.

- A clock input: all transfers occur on the rising edge of clock.
- A reset input: when true, initialize the system.
- data_in : 2 bit input data.
- data_out : 2 bit output data.
- erdy : environment ready control input.
- srdy : system ready status output.
- sdone : system done output status.
- estart : environment control input, asserted to request the system to start.

The system will follow a ready-ready protocol to read data from the environment If the environment has data to send, it asserts its *erdy* signal. If the receiver accepts the data then it asserts its *srdy* signal. Thus the typical handshake will be of the form

$$erdy \uparrow \rightarrow srdy \uparrow \rightarrow erdy \downarrow \rightarrow srdy \downarrow$$

All transfers occur on postive edges of the clock. Burst behaviour is supported in this protocol. As long as *erdy* is held high, the system can continue to accept data at every clock edge by keeping *srdy* asserted.

[1]There could be several gate-level descriptions which are equivalent to the algorithm description.

2 An algorithmic description of the system

In order to implement the algorithm, we will use storage elements for the array *num_count*, a temporary register *inreg* for the input data and a temporary register *outreg* for the output data. The register-transfer-level description of the algorithm can then be written as follows (all transfers on the positive edge of clock, clock and reset inputs not explicitly used in this description).

```
declare: registers nc1(7:0), nc2(7:0), nc3(7:0), nc4(7:0),
                   inreg(1:0), outreg(1:0), treg(7:0),
                   count(7:0), mc(7:0)
reset:  if estart then
            goto start
        else
            goto reset
        end if
start:   {nc0 = 0 , nc1 = 0 , nc2 = 0 ,
            mc = 0 , nc3 = 0 , count = 0 , treg = 0}
read:    {
         if erdy then
           {inreg = data_in , count = count + 1 , goto compare}
         else
            goto read
         end if
         , emit srdy }
compare: if inreg = "00" then
           nc0, treg = nc0 + 1
         else
           if inreg = "01" then
             nc1, treg = nc1 + 1
           else
              if inreg = "10" then
                 nc2, treg = nc2 + 1
              else
                 nc3, treg = nc3 + 1
              end if
           end if
         end if
update:  if treg > mc  then
           {mc = treg , outreg = inreg}
         end if
loop:    if count = 128 then
           go to done
         else
           go to read
         end if
```

```
done:    {
         if estart then
           go to start
         else
           go to done
         end if
          , emit sdone }
```

3 Towards a digital system: the control-data decomposition

If we examine the RTL description outlined above, we observe that in the description, the actual value stored in the registers involved in transfers is not immediately relevant in the sense that the next step does not depend on the actual value. However, at many points, *predicates* (predicates are functions which evaluate to true or false) which are dependent on register values are immediately relevant.

By default, we define all values stored in registers as **data**, and all predicates used in the RTL description as **control predicates**. The **data-path** derived from the RTL description is then the set of registers and function units together with the communication resources that help them interact. The **control-path** derived from the RTL description is a *sequencer* which is a finite-state machine which uses control predicates to determine the sequence of operations that the system executes.

3.1 Inferring a data-path from the RTL description

The data-path is an interconnection of registers and functional units. Assume that each register has a single data input, a single data output and an enable input. In this example, we will use a single adder, because there is at most one addition in any step[2]. A comparator is also used.

To construct the data-path, there is a simple procedure:

- Finalize the actual registers and the function units, and the allocation of individual transfers in the RTL to these actual resources.
- For each register, look at all its possible sources and introduce a multiplexor which selects one of the potential sources. The select signals for the multiplexor will be provided by the control path.
- For each function unit, look at the possible sources for each of its inputs. For each input, introduce a multiplexor as we did for the register in the step above.

[2]For reasons related to interconnection lengths, two adders could also be used even if they are under-utilized.

- If relevant, introduce decoders at register outputs in order to generate appropriate status signals which act as inputs to the control path.

The resulting data-path together with the pathways and other components is shown in Figure 1.

3.2 Inferring a control-path from the RTL description

The control-path is a Mealy FSM which communicates with the data-path and the environment. In this case, since all transfers take place in one cycle, the number of states is the number of statements in the RTL description. The control FSM effects transfers in the data-path. To effect a particular transfer, the target register has to be enabled, and the select inputs to all multiplexors in the transfer path are driven to appropriate values. The resulting state transition graph of the control path is indicated in Figure 2. Only asserted outputs are shown on the relevant state or transition in the state transition graph.

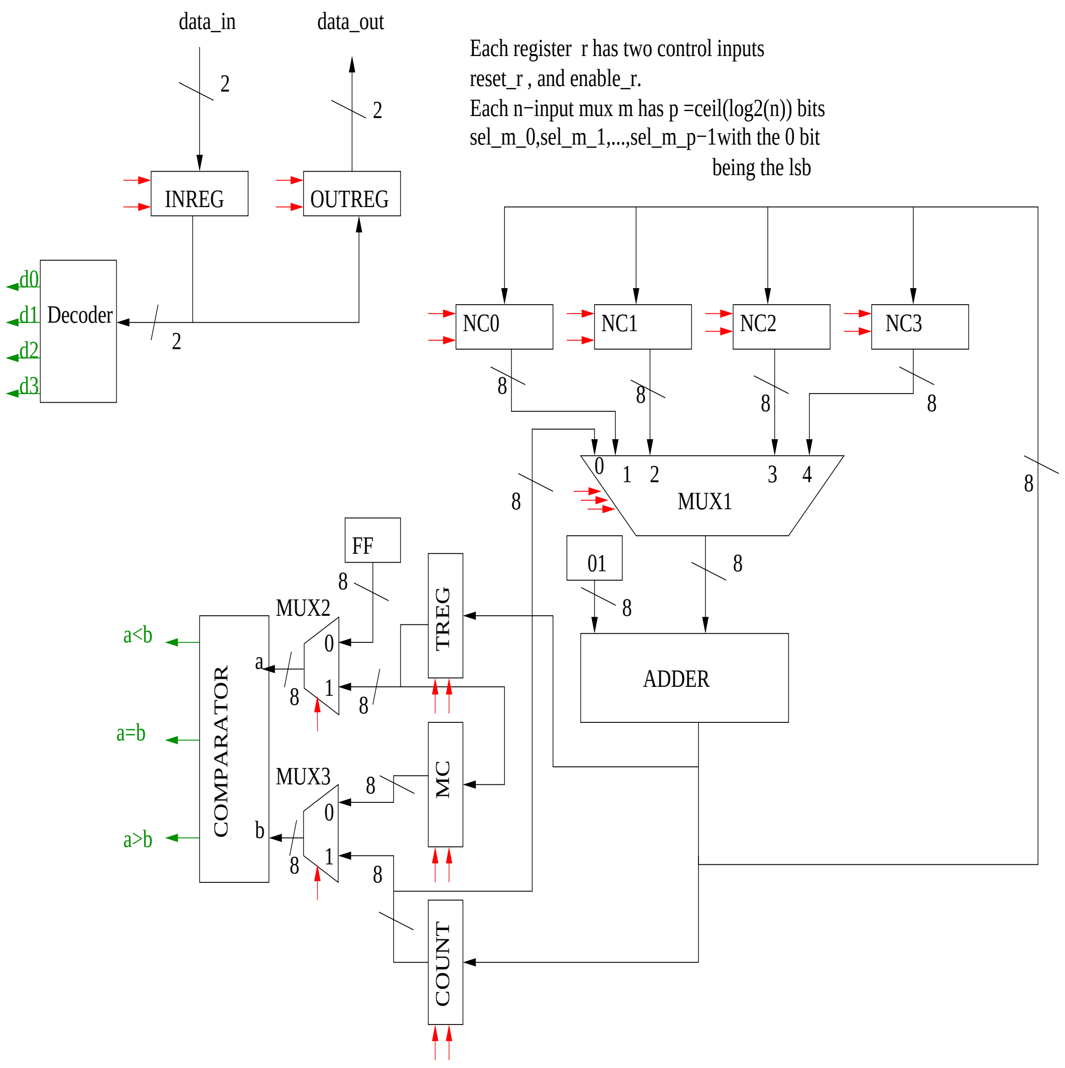

Figure 1: Data-path of the proposed system

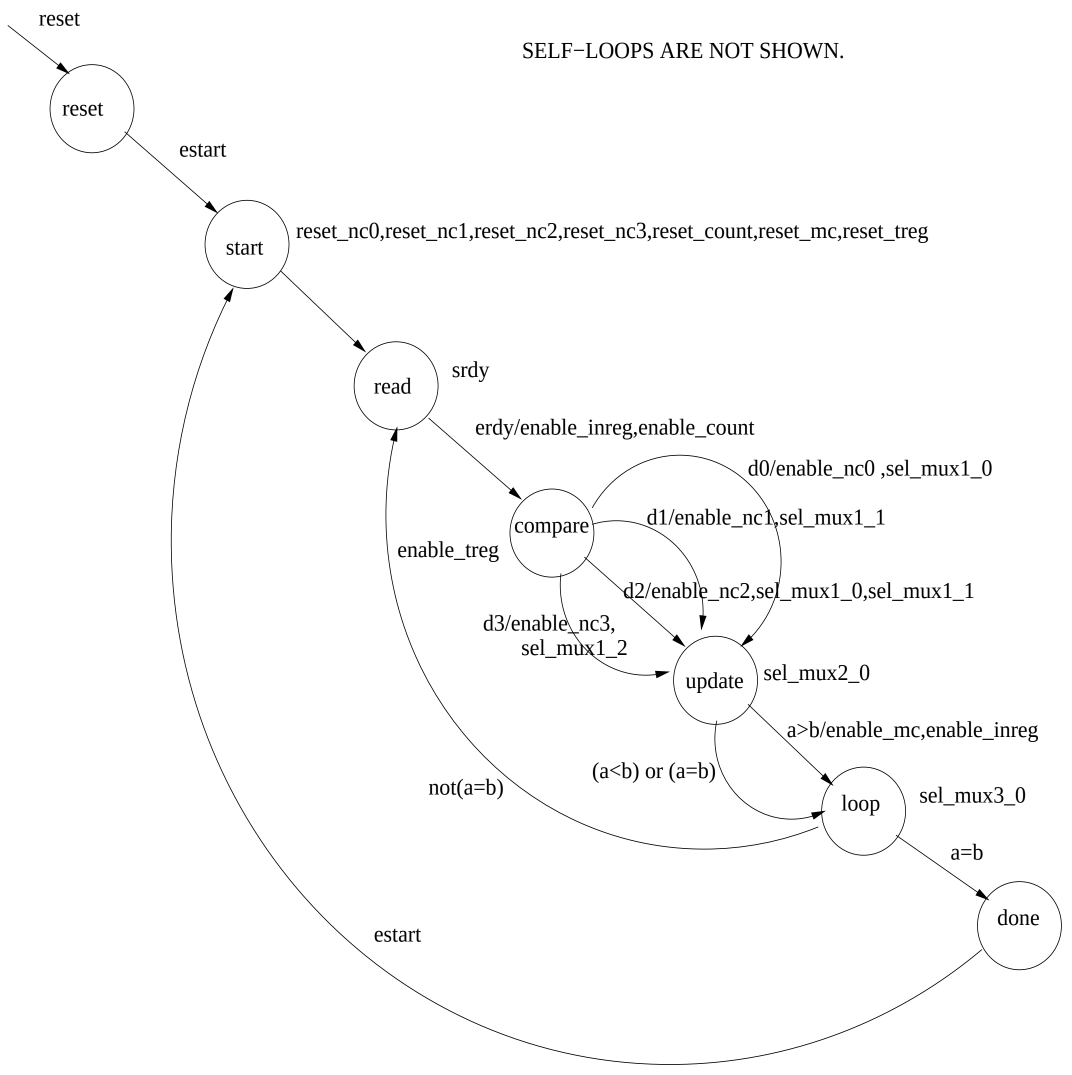
reset
SELF−LOOPS ARE NOT SHOWN.
reset
estart
start
reset_nc0,reset_nc1,reset_nc2,reset_nc3,reset_count,reset_mc,reset_treg
read
srdy
erdy/enable_inreg,enable_count
d0/enable_nc0 ,sel_mux1_0
compare
d1/enable_nc1,sel_mux1_1
enable_treg
d2/enable_nc2,sel_mux1_0,sel_mux1_1
d3/enable_nc3,
sel_mux1_2
update
sel_mux2_0
a>b/enable_mc,enable_inreg
(a<b) or (a=b)
not(a=b)
loop
sel_mux3_0
a=b
done
estart

Figure 2: Control-path of the proposed system