Register-Transfer-Level (RTL) descriptions

Madhav Desai

March 23, 2018

An FSM can be viewed as an algorithm

```
// inputs {a,b}, outputs {P,Q}, initial state phi
phi: if(input == a)  then
       // Q(k) = 1, q(k+1) = s_a
        Q = 1, $state := s_a
     else
        Q = 1, $state := phi end if
s_a: if (input == a) then
       $state := s_aa
     else if(input == b)
       $state := s_ab end if
s_aa:...  etc ....
s_aab: if (input == b) then
         P = 1, $state := phi
       else if(input == a)
         Q = 1, $state := s_aba
       else
         Q = 1, $state := s_phi end if
...
```

An FSM for a counter down timer

```
// by default, Y = 0
rst: if(start)   then
        $state := s1
     else
        $state := s0
     end if
     ready := 1
s1:  $state := s2
s2:  $state := s3
s1023: Y = 1, $state := s0
```

for every state some variables are required and it gets multiplied

Too many states!

Introduce register (storage) variables and register transfers

```
// count is a register with 8 bits.
register count[9:0];
thread DCount {
 rst: if (start)  then
        // x := y means x(k+1) = y(k)
        count := 1023,
        // goto decr means
        //   q(k+1) = decr
        $state := decr
     else goto rst end if
 decr: if(count == 0) then
        // done = 1 means done(k) = 1
        done = 1, $state := rst
      else
        count := count - 1, $state := decr
      end if
}
```

The control-path and the data-path in the RTL

A couple of points:

- Identify the transfers used in the RTL using output variables (or symbols).
- Identify the predicates used in the RTL using output variables (or symbols).
- The RTL algorithm can be thought of as two parts:
  - A pure Mealy machine (the control path).
  - The implementation of the transfers (the data path)

The data-path for the counter

- Introduce signals $T0$, $T1$ to indicate the transfers $count := 1023$ and $count := count - 1$.
- the data-path implements the two transfers as

  $$count(k{+}1) <= (\,T1(k)?\; count(k){-}1 \;:\; (\,T0(k)?\; 1023 \;:\; count(k)))$$

  Text

  and produces the status signal

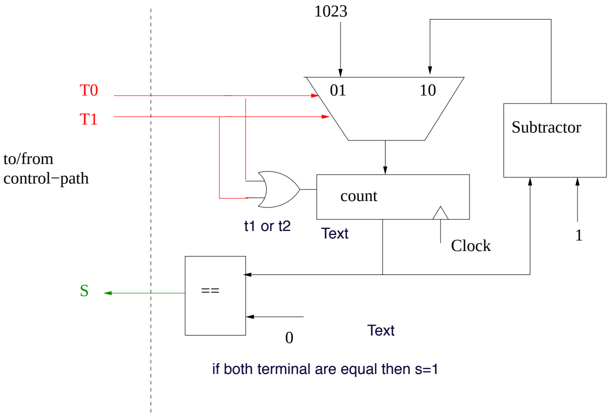
  $$S(k) = (count(k) == 0)$$

t1 =1 count-1

t1 = 0 , t0=1. 1023

t1=0 countK

The data-path: implement using muxes, combinational circuits, registers



t1 =1 count-1

t1 = 0 , t0=1. 1023

t0 = t1=0 countK

Figure : Datapath for the counter

Control path FSM

```
// by default
//    T0=T1=done=0 unless specified.
// by default, unless specified,
//   $state := next-statement-label
fsm DCount_control {
 rst: if (start)  then
        T0 = 1, $state := decr
     else $state := rst end if
 decr: if(S) then
        done=1, $state := rst
      else
        T1 = 1, $state := decr
      end if
}
```

t1 =1 count-1

t0 = t1=0 countK

t1 = 0 , t0=1. 1023

Control path FSM

state	reset	start	S	next-state	T0	T1	done
_	1	_	_	rst	_	_	_
rst	0	0	_	rst	0	0	0
rst	0	1	_	decr	1	0	0
decr	0	_	0	decr	0	1	0
decr	0	_	1	rst	0	0	1

Advantages of separating Data/Control paths

- State complexity is clearly demarcated.
- Datapath resources are clearly defined.
- Verification problem is simpler.
- It is easy to estimate resources needed.

RTL Pseudo-code Convention

```
input start, incount[31:0]
output done
register count[31:0]
thread Main() {
   // default reset state.
   rst: if (start) then
           $state := work, done = ’1’,
              count := incount
        else
           $state := rst
        end if
   decr: if (count > 0) then
            count := count-1,
            $state := decr
         else
            done = 1, $state := rst
         end if
}
```

RTL Pseudo-code Convention

```
.. declare ins, outs, registers.
.. define threads
thread <thread-1> {
   ..statements..
}
thread <thread-2> {
   ..statements..
}
etc..
```

Going further: collections of RTL threads

- The concept of subroutines can be extended to RTL.
- Fork-join structures are a natural extension of subroutines, but which provide parallelism.
- Synchronized threads can be used to create deadlock free implementations of useful things (such as pipelines).

Subroutines

```
thread master {
A: if (...)  then
    call=1, $state := W
   else
    $state := A
   end if
W: if (ret)  then
    $state := R
   else
    $state :=  W
   end if
R: ....
}
```

```
thread slave {
  Idle:
   if (call) then
     $state := W
   else goto Idle end if
  W:
   if (finished)   then
    ret = 1, $state := Idle
   else
    goto W
   end if
}
```

Fork-Join Structures

```
thread Master {
A: if (...) then
     call=1, $state := W
   else goto A end if
W: if (retA and retB) then
      $state := R
   else $state := W end if
R: ....
}
```

both the D filp flop will be used simultaniously

Text

```
thread Slave_1 {
Idle: retA = 1
  if (call)  then
   $state := W
  else $state := Idle end if
W: if (finished)
     $state := Idle
   else $state := W end if }
thread Slave_2 {
Idle: retB = 1
 if (call) then $state := W
 else goto Idle end if
W: if (finished) then
    $state := Idle
   else $state := W end if }
```

Synchronized Threads

```
thread Peer_1 {
Working:
  if (done-working) then
    $state := Synch
  else
    $state := Working
  end if
Synch:
  SynchA = 1
  if (SynchB) then
    $state := Working
  else
    $state := Synch
  end if
}
```

```
thread Peer 2 {
Working:
  if (done-working)
     $state := Synch
  else
     $state := Working
  end if
Synch:
  SynchB = 1
  if (SynchA)  then
    $state := Working
  else
    $state := Synch
  end if
}
```