EE214: Register-Transfer-Level Machines

Madhav Desai

March 28, 2019

A count-down timer

- Inputs: $Start$, $count\_in[31:0]$.
- Outputs: $Done$.
- Behaviour:
  - Wait until $Start = 1$. When $Start$ becomes 1, sample $count\_in$ as M. Wait until M clock cycles have elapsed and make $Done = 1$. After this wait for $Start = 1$ and so on.

Building this as a Mealy machine

How many states will you need?

Introduce registers

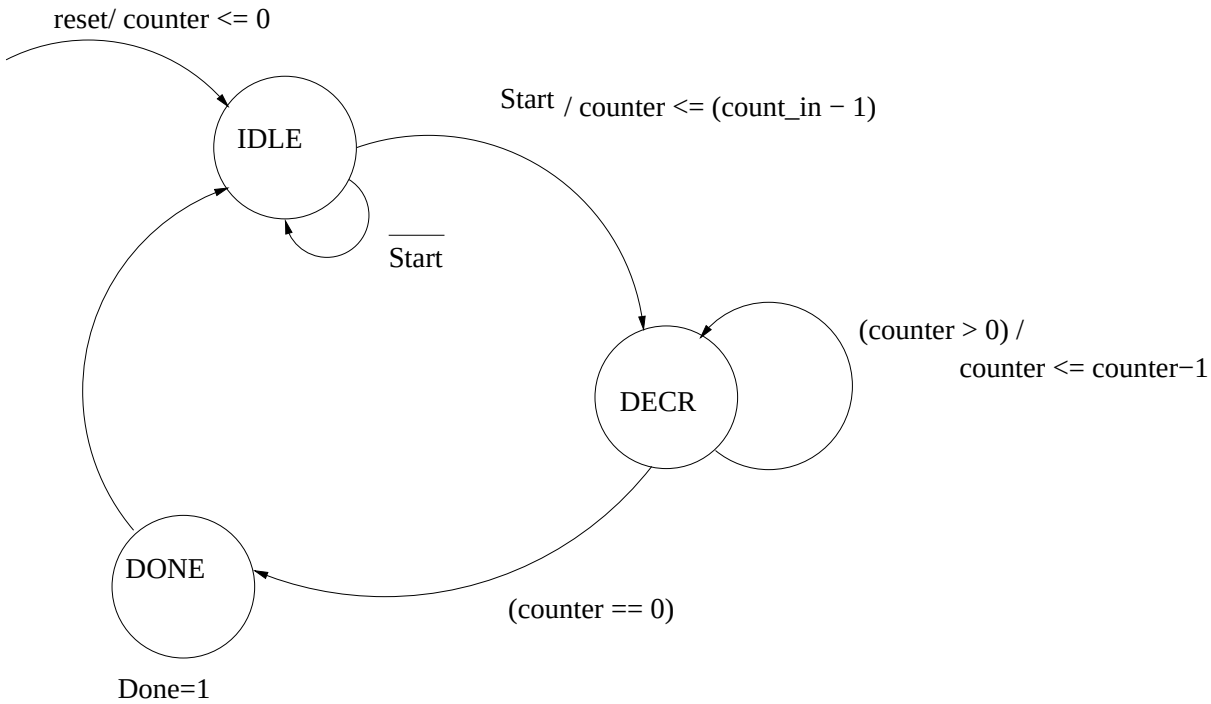


Figure: Count-down timer RTL machine

Recurrence relation

RTL-machine $= (\Sigma, \Lambda, Q, R, \delta, \mu, \lambda)$.

$$\begin{aligned} q(k+1) &= \delta(x(k), q(k), r(k)) \\ r(k+1) &= \mu(x(k), q(k), r(k)) \\ y(k) &= \lambda(x(k), q(k), r(k) \end{aligned}$$

Note that this is still a Mealy FSM, but the state is represented by $(q(k), r(k))$.

Register transfer: implementation

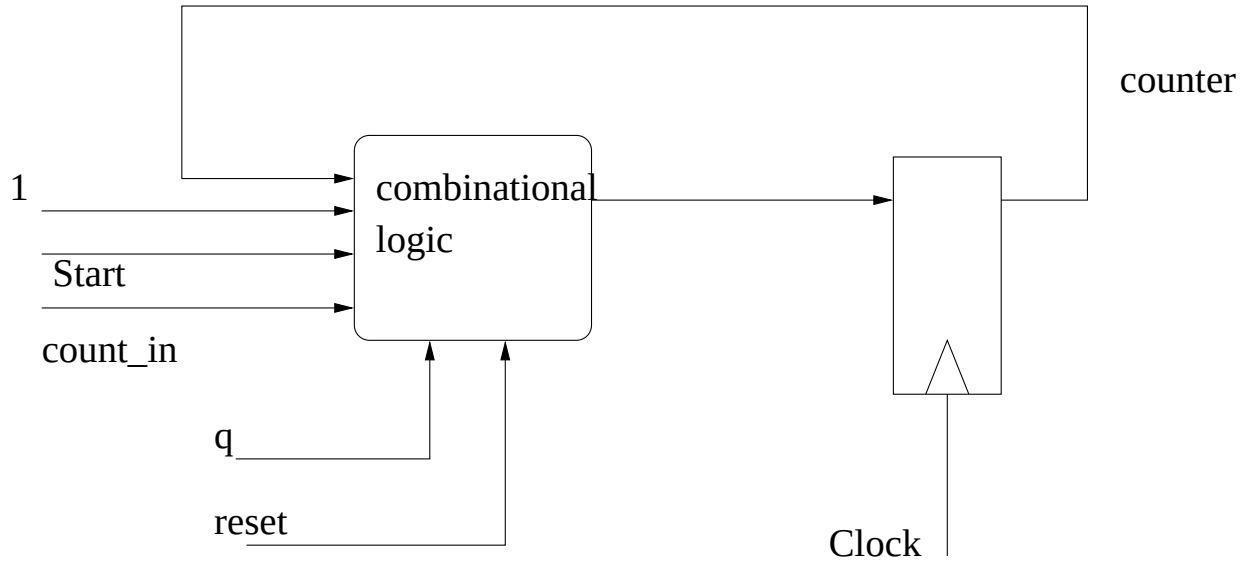
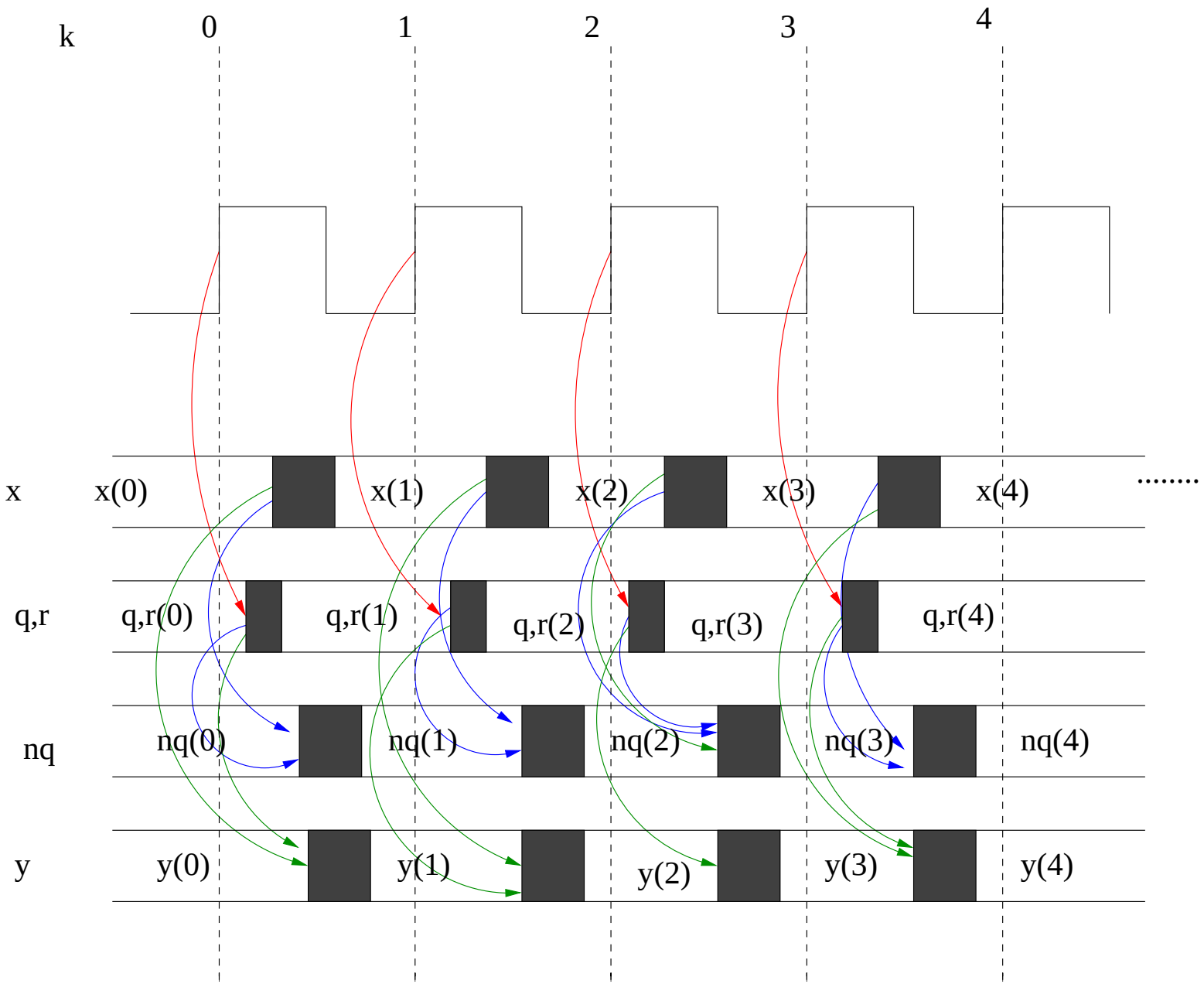


Figure: Register transfer implementation

Timing Diagram



Algorithm: RTL pseudo-code

```
input count_in[31:0], Start
output Done
register counter[31:0]
IDLE: if (Start) then
        goto DECR, counter <= (count_in - 1)
      else goto IDLE end if
DECR: if (counter > 0) then
        counter <= (counter - 1), goto DECR
      else goto DONES end if
DONES: Done = 1, goto IDLE
```

Algorithm: RTL pseudo-code

```
a = b
```

means $a(k) = b(k)$

```
p <= f(u,v)
```

means $p(k+1) = f(u(k), v(k))$.

RTL Algorithm: VHDL behavioural code

See

- See CountDownTimer.vhdl.
- See EfficientCountDownTimer.vhdl.

Trace file construction

inputs	outputs
clock=0 x(0)	y(0)
clock=1 x(0)	ignore
clock=0 x(1)	y(1)
clock=1 x(1)	ignore
clock=0 x(2)	y(2)
clock=1 x(2)	ignore
etc...	

RTL Subroutines

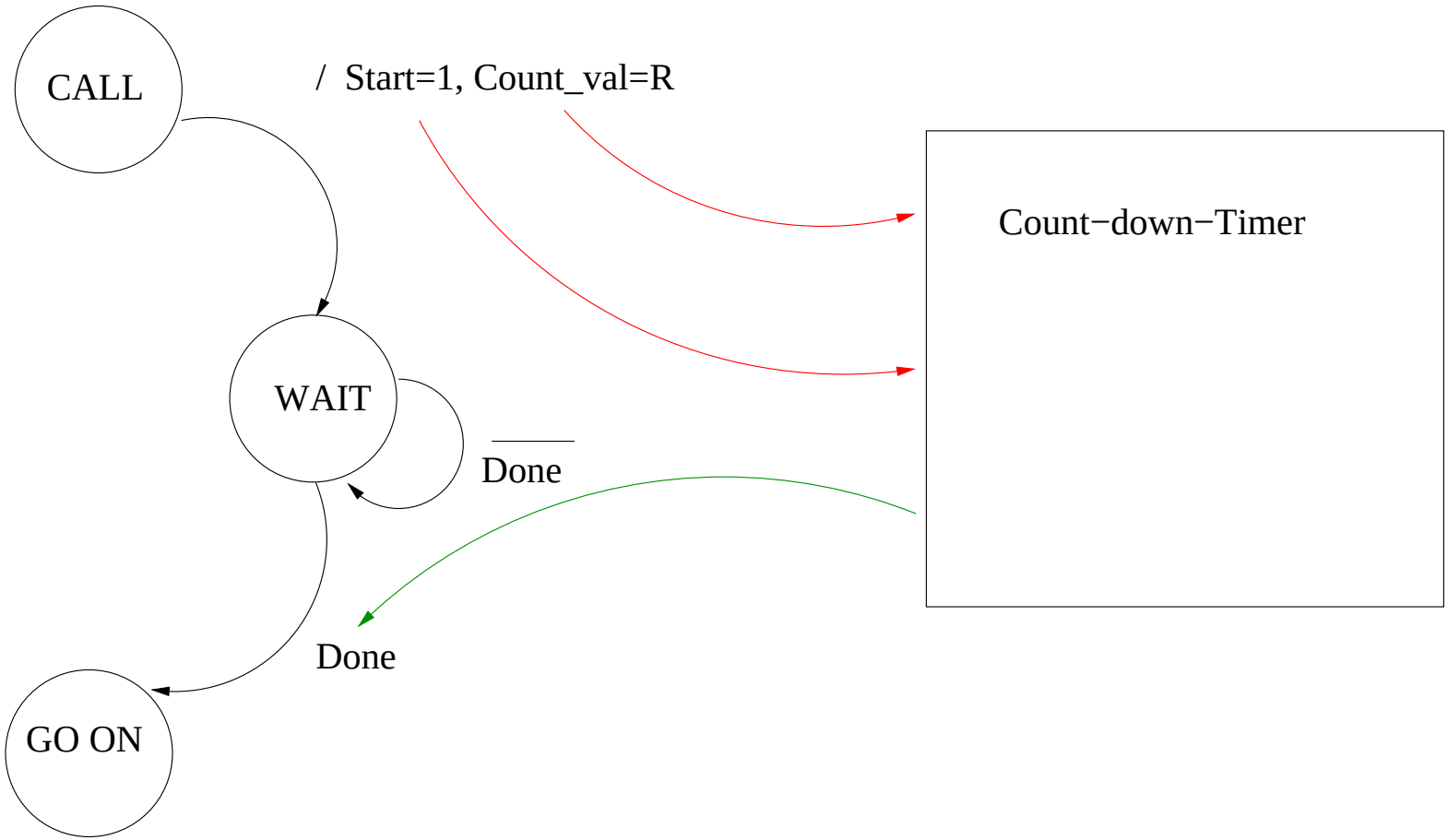


Figure: RTL Subroutine: be careful about combinational cycles

Lets solve a problem together

Implement a 4-bit multiplier, using a shift and add algorithm. The inputs are two 4-bit numbers and a Start signal. The output is an 8-bit product and a Done signal. When Start=1, the multiplier starts. When it finishes, it indicates Done=1 (output will be valid when Done=1) for a single clock cycle and then waits for Start all over again.

Algorithm: shift-and-add multiplier

$$p = \sum_{k=0}^{3} 2^k \times b_k \times a$$

This can be achieved as follows:

$$\begin{aligned}
S0[7:0] &= (0 + (2^4 \times b_0 \times a)) \times 2^{-1} \\
S1[7:0] &= (S0 + (2^4 \times b_1 \times a)) \times 2^{-1} \\
S2[7:0] &= (S1 + (2^4 \times b_2 \times a)) \times 2^{-1} \\
S3[7:0] &= (S2 + (2^4 \times b_3 \times a)) \times 2^{-1}
\end{aligned}$$

Algorithm: shift-and-add multiplier RTL pseudo-code

```
Input a[3:0], b[3:0], Start
Output p[7:0]
Register R[7:0],T[2:0],Count[1:0]
IDLE: if Start then
          goto WORKS,
            // add and shift..
            R[7] <= 0
             R[6:3] <= (b[0] ? a : 0),
              R[2:0] <= 0,
               T <= b[3:1], Count <= 0
      else goto IDLE
      end if
```

Algorithm: shift-and-add multiplier

```
WORKS: if (Count < 3) then
          goto WORKS,
            // add with carry...
            //  and shift by 1.
            R[7:3] <= (R[7:4] +
                              (T[0] ? a : 0)),
            R[2:0] <= R[3:1],
            T <= (T >> 1),
            Count <= (Count + 1)
       else
          goto DONES,
            p <= R[7:0]
       end if
DONES: Done=1, goto IDLE
```

Implement the shift-and-add multiplier

- Describe in VHDL.
- Simulate the VHDL and check if it works correctly (trace-file!).
- Can you design a shift-and-subtract divider?

Summary

- By introducing registers, we can describe algorithms.
  - Total state now has a register value component and FSM state component.
  - The exact values of registers are usually important.
  - An algorithm can be expressed as a state machine (sequence) of register transfers.
  - Implementation in logic is similar to FSM implementation.
- Concept of subroutines allows you to reuse stuff.
- Identify states, registers, express the algorithm, describe it in VHDL, simulate, synthesize, map to hardware, test.