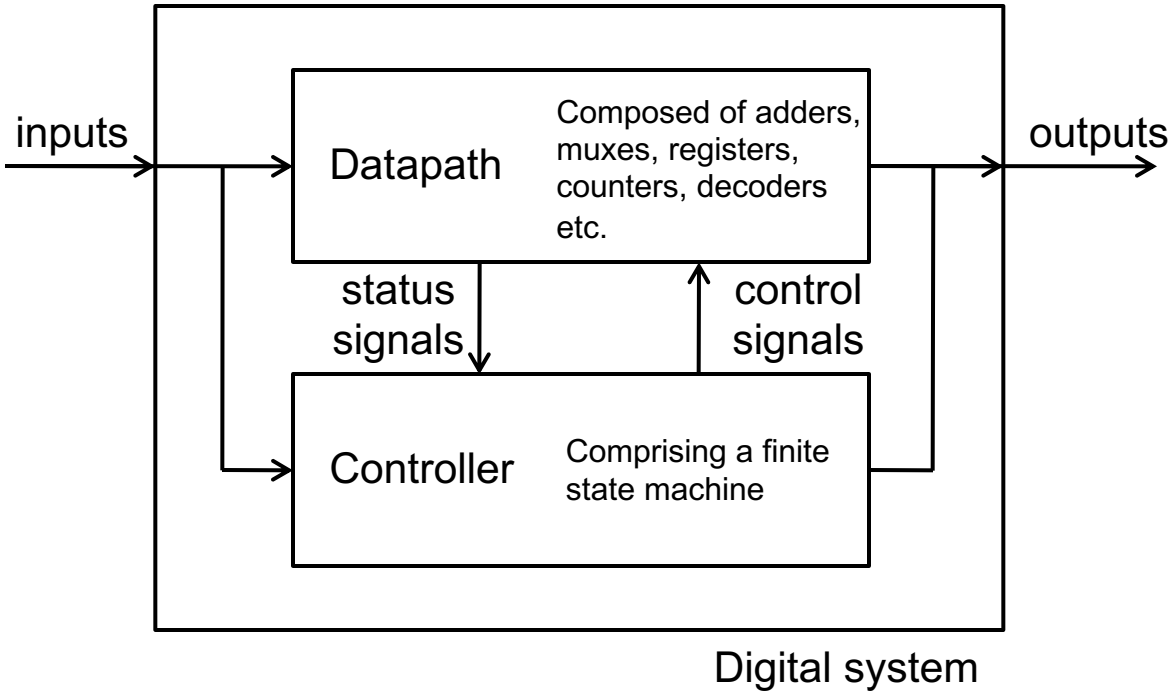
COMP3222/9222 Digital Circuits & Systems

8. Digital System Design

Objectives

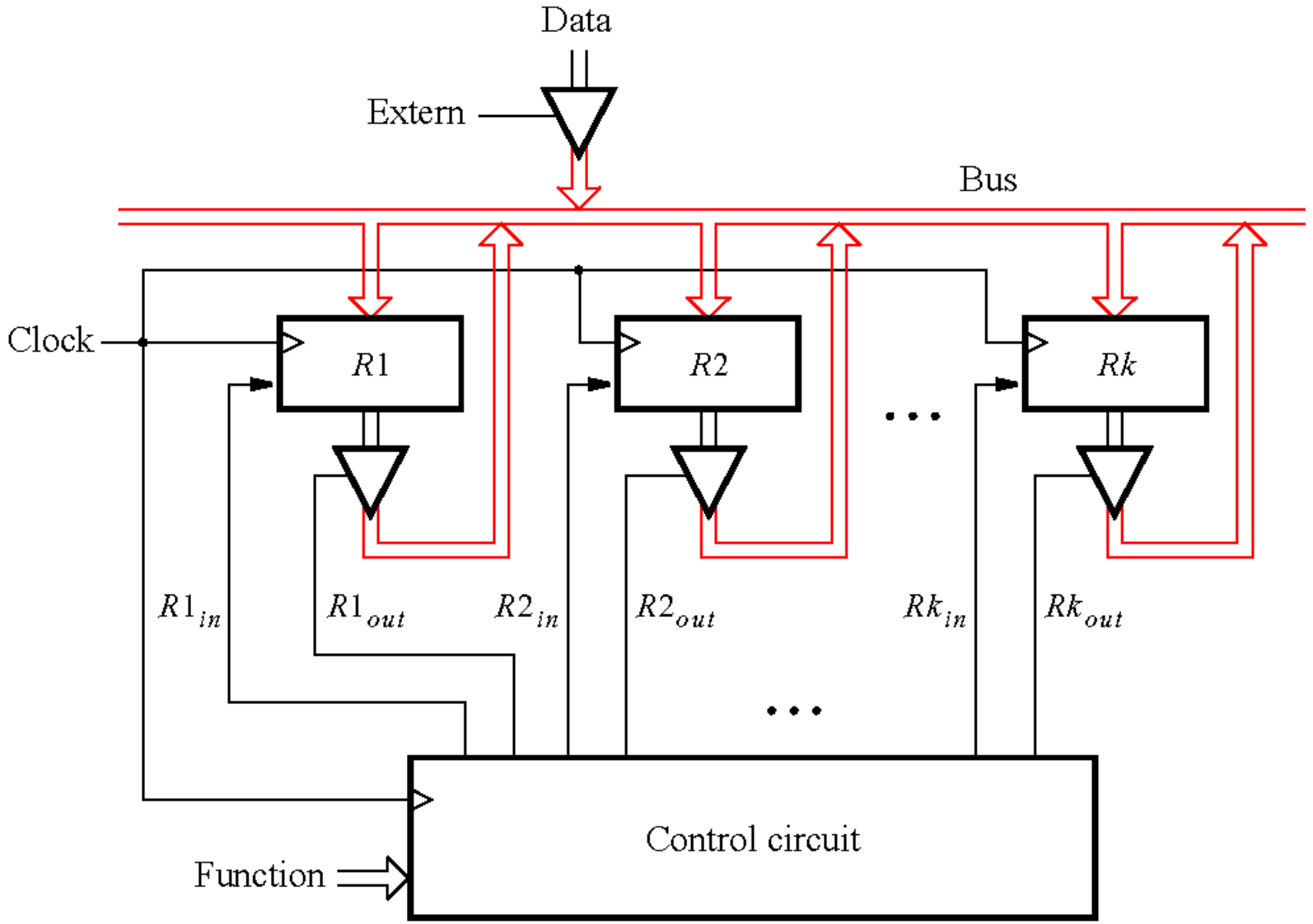
- Apply design techniques to comprehensive digital design problems
  - Consider the datapath components needed, the finite state machines required for their control and their description in VHDL
- Learn how digital systems comprising datapaths and control circuits can be derived from an ASM chart
- Look at a number of practical issues to do with real system inputs and outputs

Digital system



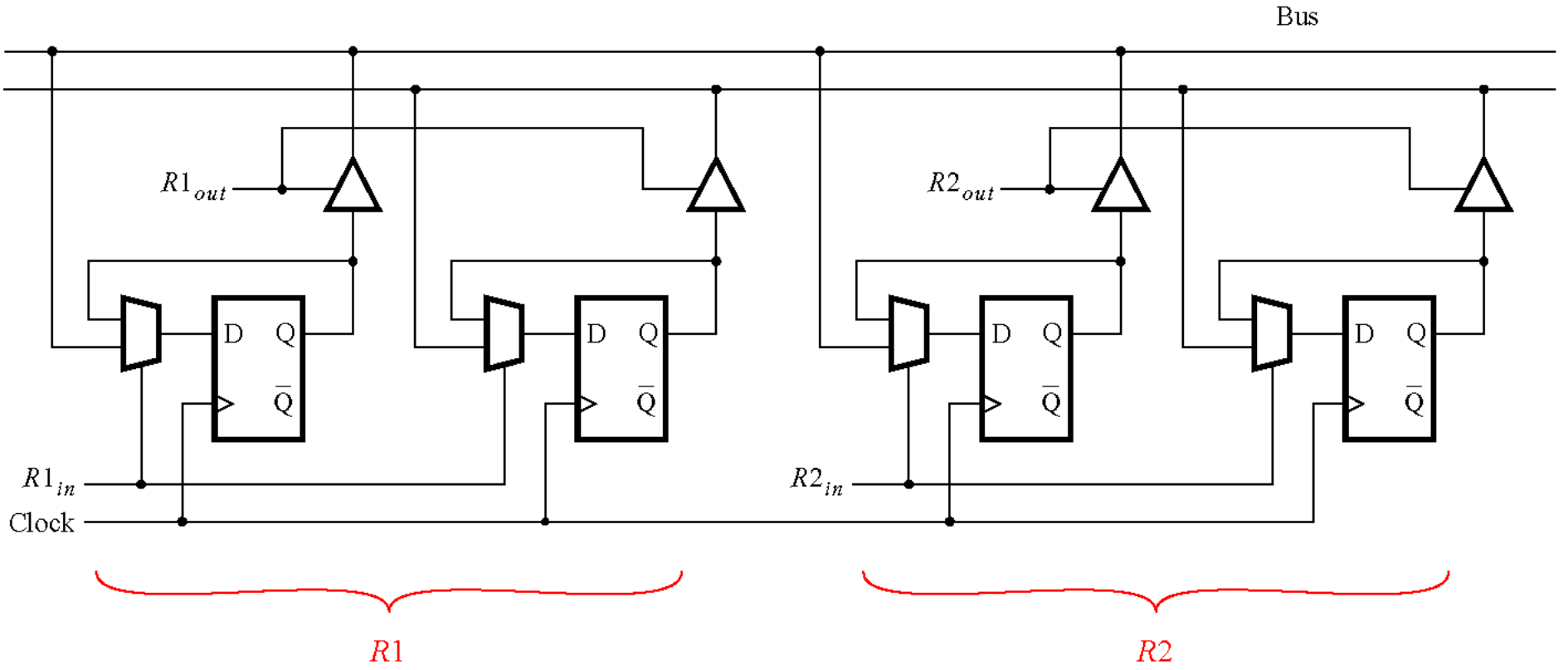
- A digital system comprises a *datapath*, which transforms the data as required by a specification, and a *controller* (control unit, control path), which supervises the operation of the datapath by monitoring its *status* and setting *control* signals
- The behaviour of both parts is conveniently modelled in an integrated manner using an Algorithmic State Machine (ASM) chart (see later)

Design Example 1 (pp 438-450): A digital system with *k* registers



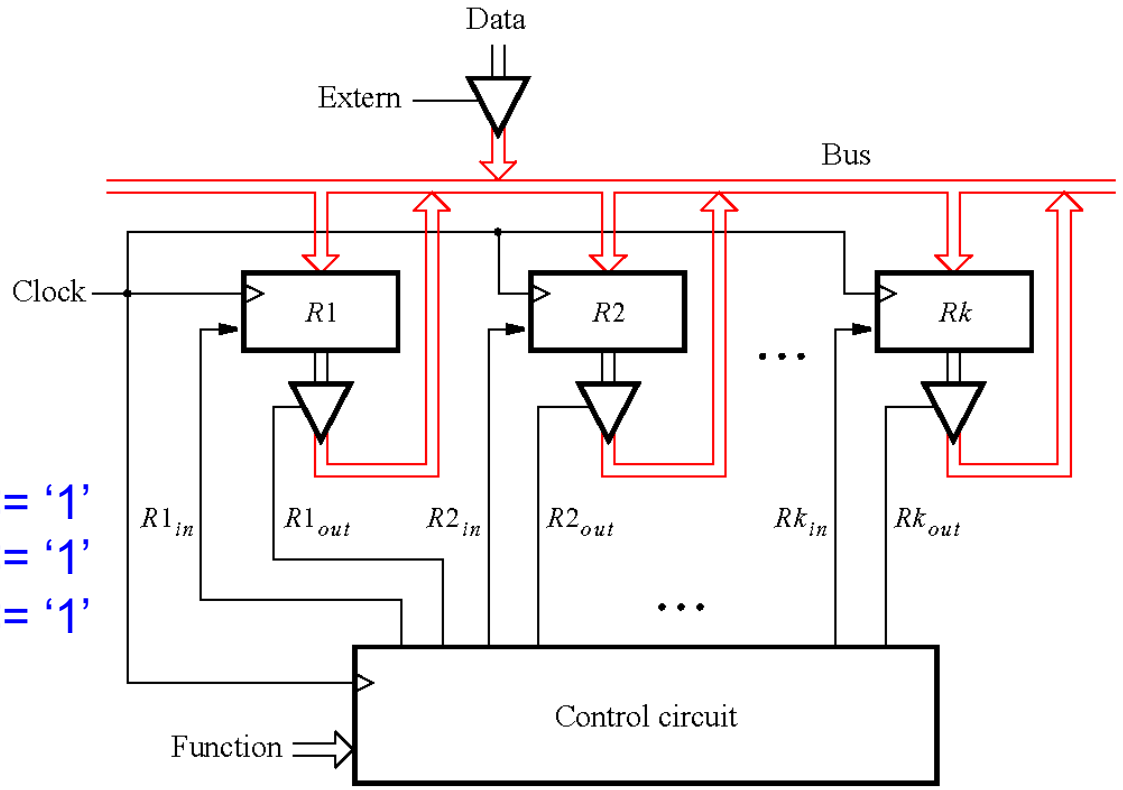
Recall details for connecting registers to a bus

- Consider two 2-bit registers
  - 3-state (tri-state) buffers used to avoid “tying” outputs together



Control circuit design

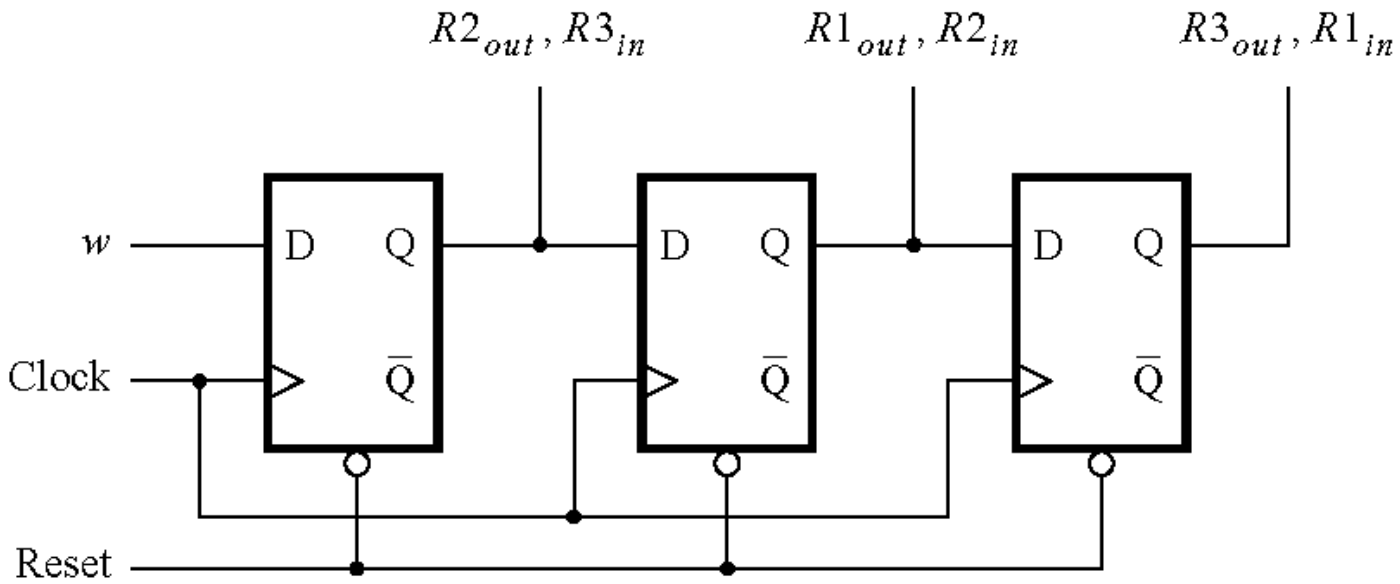
- Consider the control required to swap the contents of R1 and R2 using R3 for temporary storage
  - What are the individual *register transfers* required to effect the swap?
  - Which control signals need to be asserted for each transfer?
  - When & how should the control signals be sequenced?



In successive steps:
1. R3 <= R2: R3in <= ‘1’; R2out <= ‘1’
2. R2 <= R1: R2in <= ‘1’; R1out <= ‘1’
3. R1 <= R3: R1in <= ‘1’; R3out <= ‘1’

A shift-register based control circuit

- Swapping the contents of R1 and R2 using R3 for temporary storage
  - Could use *one-hot* control to enable 3-state buffers and loading of registers
  - Suffers delay of 1 cycle after input *w* asserted
  - Assumes *w* is deasserted for at least two clock cycles after the period



Code for an *n*-bit register with enable *Rin*

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regne IS
      GENERIC ( N : INTEGER := 8 ) ;
      PORT (  R              : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
              Rin, Clock     : IN     STD_LOGIC ;
              Q              : OUT    STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END regne ;

ARCHITECTURE Behavior OF regne IS
BEGIN
      PROCESS
      BEGIN
            WAIT UNTIL Clock'EVENT AND Clock = '1' ;
            IF Rin = '1' THEN
                  Q <= R ;
            END IF ;
      END PROCESS ;
END Behavior ;
```

Code for an *n*-bit 3-state buffer

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY trin IS
      GENERIC ( N : INTEGER := 8 ) ;
      PORT (  X    : IN       STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
              E    : IN       STD_LOGIC ;
              F    : OUT      STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END trin ;

ARCHITECTURE Behavior OF trin IS
BEGIN
      F <= (OTHERS => 'Z') WHEN E = '0' ELSE X ;
END Behavior ;
```

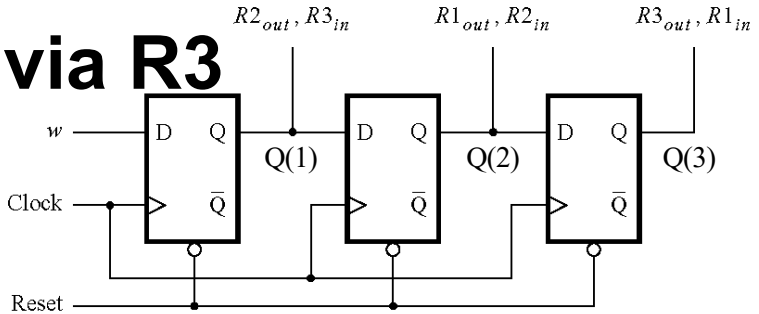
Code for L-R shift register with reset

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shiftr IS -- left-to-right shift register with async reset
        GENERIC ( K : INTEGER := 4 ) ;
        PORT ( Resetn, Clock, w : IN     STD_LOGIC ;
                Q               : BUFFER STD_LOGIC_VECTOR(1 TO K) ) ;
END shiftr ;

ARCHITECTURE Behavior OF shiftr IS
BEGIN
        PROCESS ( Resetn, Clock )
        BEGIN
                IF Resetn = '0' THEN
                        Q <= (OTHERS => '0') ;
                ELSIF Clock'EVENT AND Clock = '1' THEN
                    Genbits: FOR i IN K DOWNTO 2 LOOP
                      Q(i) <= Q(i-1) ;
                    END LOOP ;
                    Q(1) <= w ;
                END IF ;
        END PROCESS ;
END Behavior ;
```

Package and component declarations

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE components IS
    COMPONENT regne -- register
        GENERIC ( N : INTEGER := 8 ) ;
        PORT  ( R            : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
                Rin, Clock   : IN     STD_LOGIC ;
                Q            : OUT    STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
    END COMPONENT ;

    COMPONENT shiftr -- left-to-right shift register with async reset
        GENERIC ( K : INTEGER := 4 ) ;
        PORT  ( Resetn, Clock, w   : IN          STD_LOGIC ;
                Q                  : BUFFER      STD_LOGIC_VECTOR(1 TO K) ) ;
    END component ;

    COMPONENT trin -- 3-state buffers
        GENERIC ( N : INTEGER := 8 ) ;
        PORT  ( X  : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
                E  : IN     STD_LOGIC ;
                F  : OUT    STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
    END COMPONENT ;
END components ;
```

The digital system from L08/S6 used to swap R1 and R2 via R3



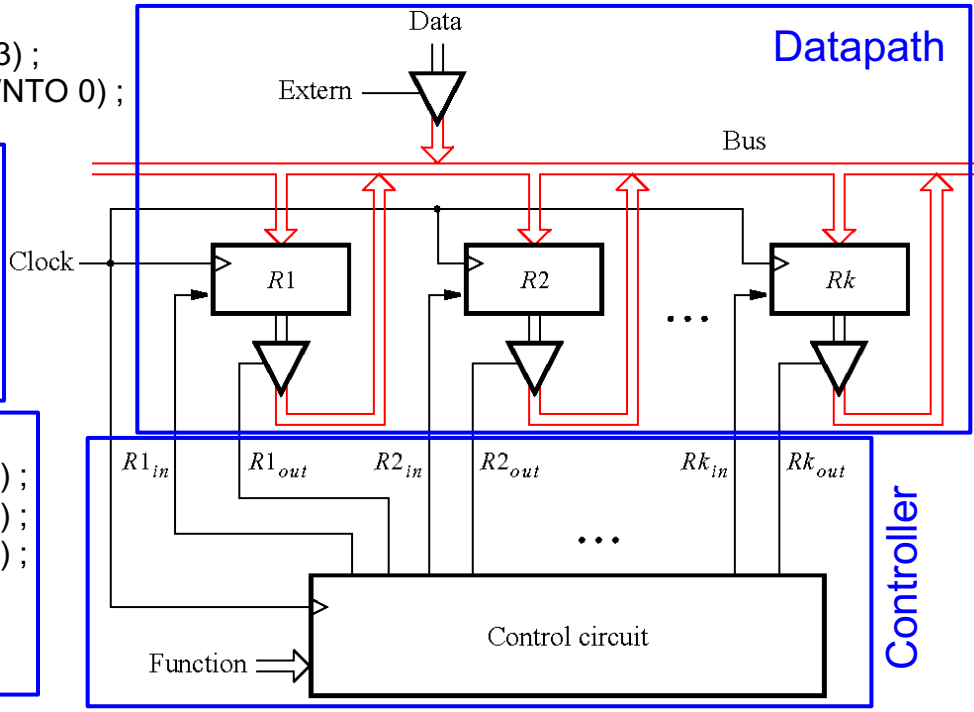

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.components.all ;

ENTITY swap IS
    PORT (  Data          : IN     STD_LOGIC_VECTOR(7 DOWNTO 0) ;
            Resetn, w     : IN     STD_LOGIC ;
            Clock, Extern : IN     STD_LOGIC ;
            RinExt        : IN     STD_LOGIC_VECTOR(1 TO 3) ; -- allows regs to be externally loaded
            BusWires      : BUFFER STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END swap ;

ARCHITECTURE Structure OF swap IS
    SIGNAL Rin, Rout, Q : STD_LOGIC_VECTOR(1 TO 3) ;
    SIGNAL R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
BEGIN
```

Controller

```
    control: shiftr GENERIC MAP ( K => 3 )
            PORT MAP ( Resetn, Clock, w, Q ) ;
    Rin(1) <= RinExt(1) OR Q(3) ;
    Rin(2) <= RinExt(2) OR Q(2) ;
    Rin(3) <= RinExt(3) OR Q(1) ;
    Rout(1) <= Q(2) ; Rout(2) <= Q(1) ; Rout(3) <= Q(3) ;
```

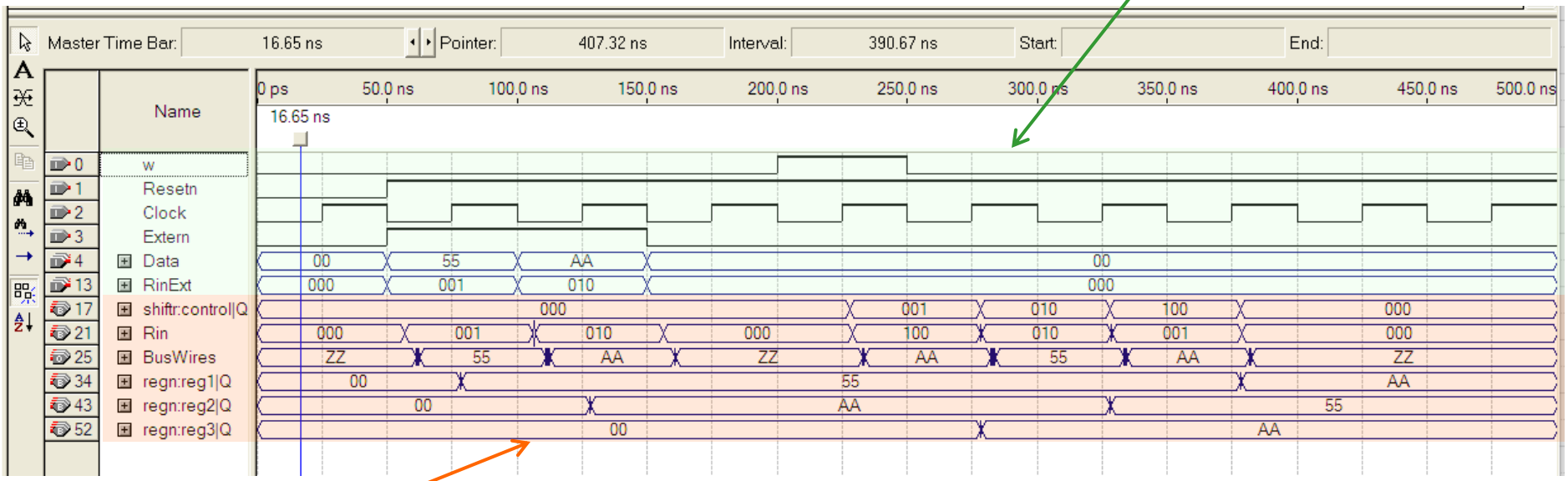
Datapath

```
    tri_ext: trin PORT MAP ( Data, Extern, BusWires ) ;
    reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
    reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
    reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
    tri1: trin PORT MAP ( R1, Rout(1), BusWires ) ;
    tri2: trin PORT MAP ( R2, Rout(2), BusWires ) ;
    tri3: trin PORT MAP ( R3, Rout(3), BusWires ) ;
END Structure ;
```



Timing simulation for the code of L08/S12

Stimuli (external signals)

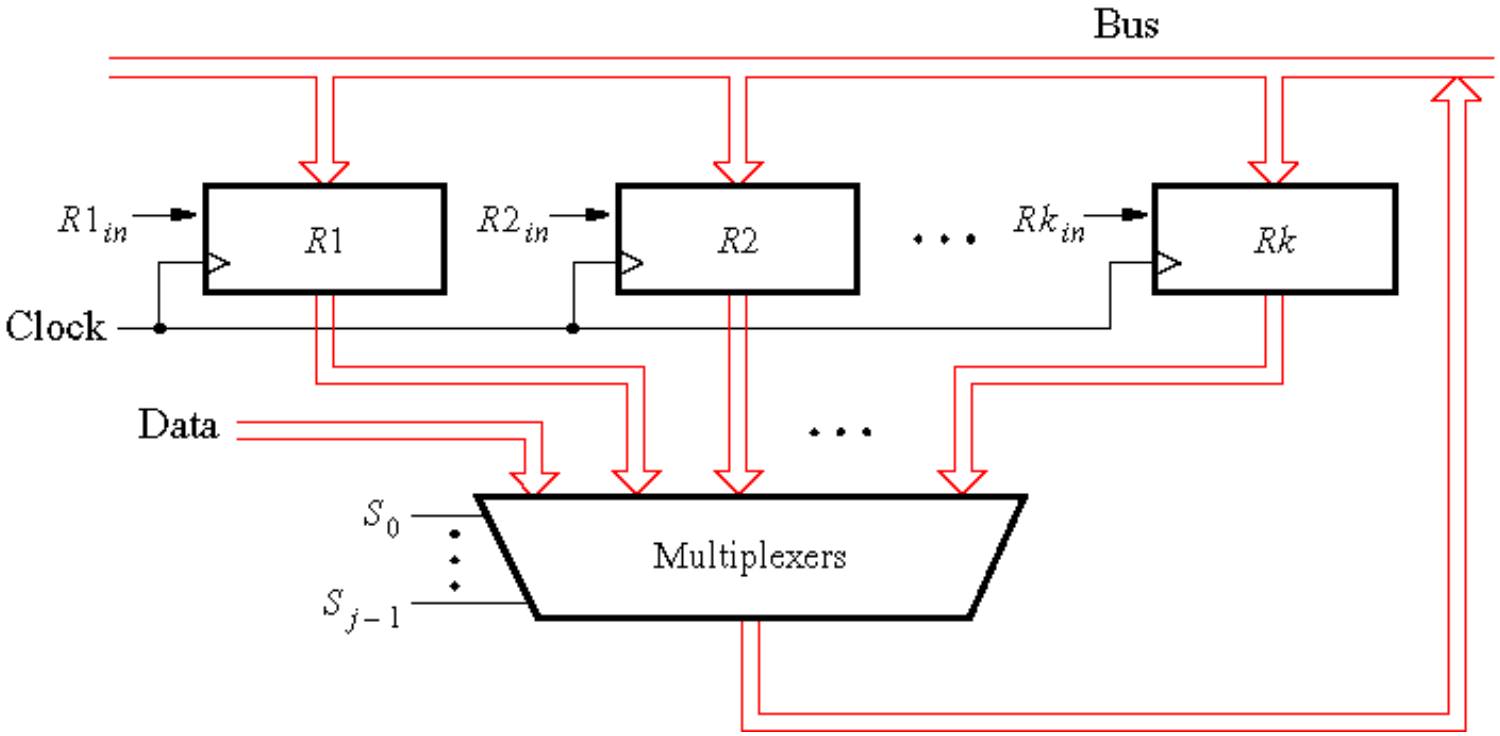


Response (internal signals)

Code and waveform inputs to generate this waveform are available from the course website

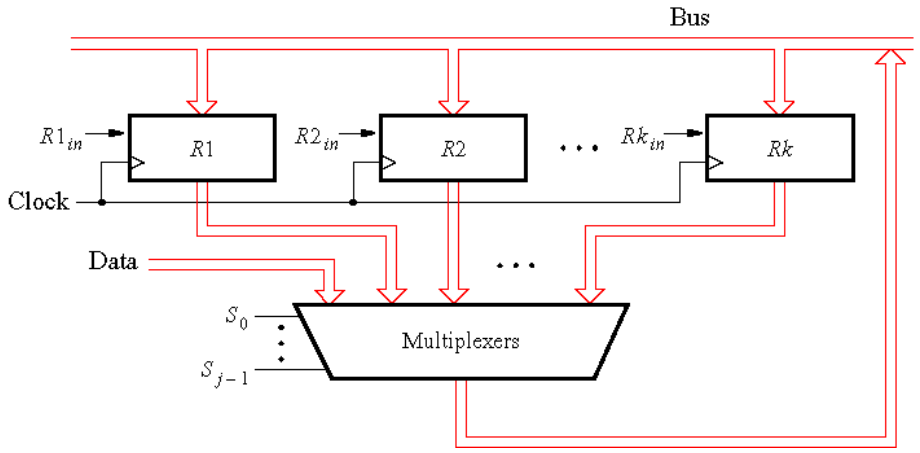
Using multiplexers to implement a bus

- More typical to use MUXes instead of 3-state buffers since programmable devices (such as FPGAs) don't usually have many 3-state resources
- Both MUX and 3-state approaches are equally valid

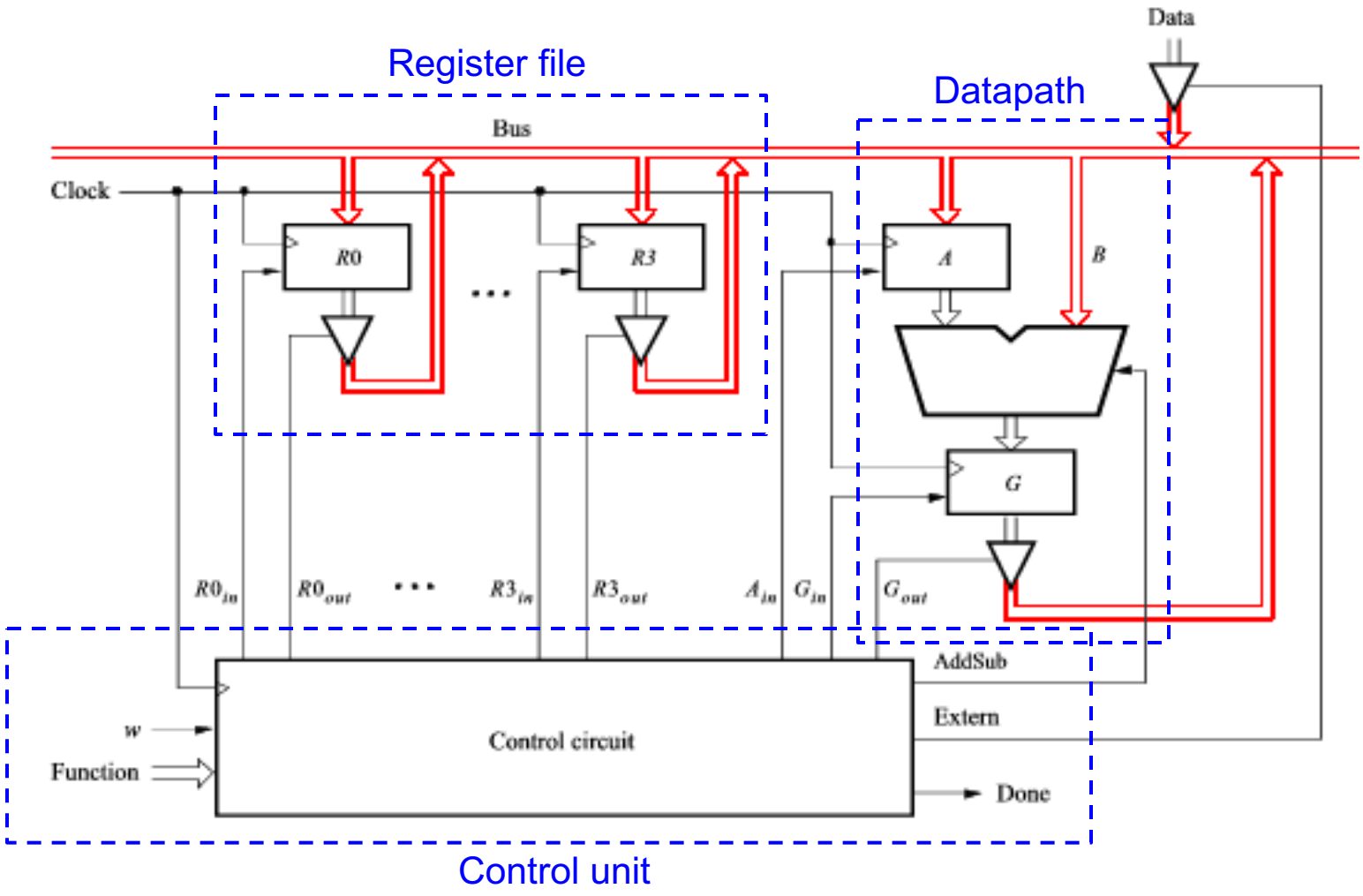


Using multiplexers for swap

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.components.all ;

ENTITY swapmux IS
        PORT (Data          : IN        STD_LOGIC_VECTOR(7 DOWNTO 0) ;
              Resetn, w     : IN        STD_LOGIC ;
              Clock         : IN        STD_LOGIC ;
              RinExt        : IN        STD_LOGIC_VECTOR(1 TO 3) ;
              BusWires      : BUFFER    STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END swapmux ;

ARCHITECTURE Mixed OF swapmux IS
        SIGNAL Rin, Q : STD_LOGIC_VECTOR(1 TO 3) ;
        SIGNAL R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
BEGIN
        control: shiftr GENERIC MAP ( K => 3 )
              PORT MAP ( Resetn, Clock, w, Q ) ;
        Rin(1) <= RinExt(1) OR Q(3) ;
        Rin(2) <= RinExt(2) OR Q(2) ;
        Rin(3) <= RinExt(3) OR Q(1) ;
        reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
        reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
        reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
        muxes: WITH Q SELECT
              BusWires <= Data WHEN "000",
                          R2 WHEN ”100",
                          R1 WHEN ”010",
                          R3 WHEN OTHERS ;
END Mixed ;
```

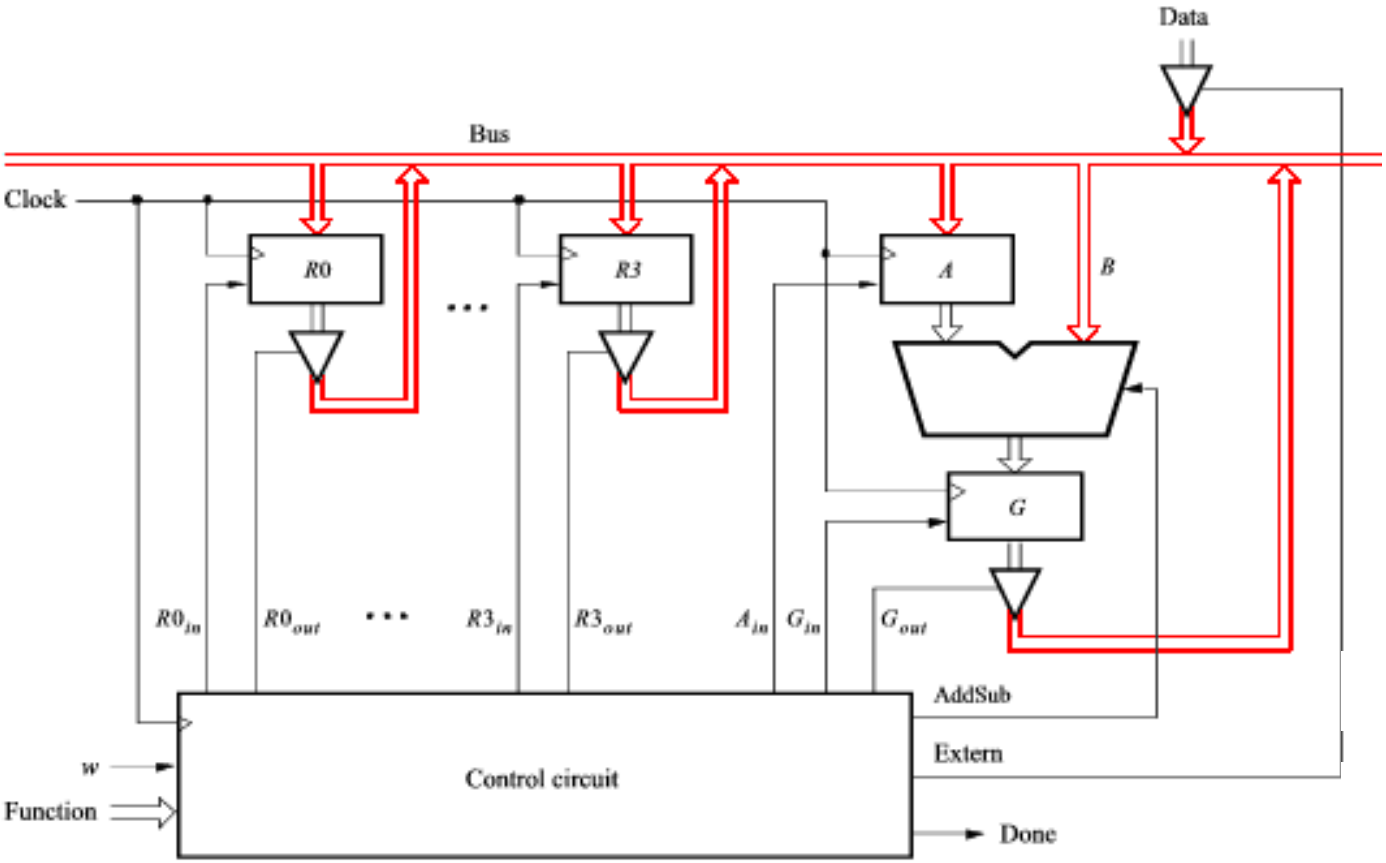


Design Example 2 (pp 450-462): A simple processor

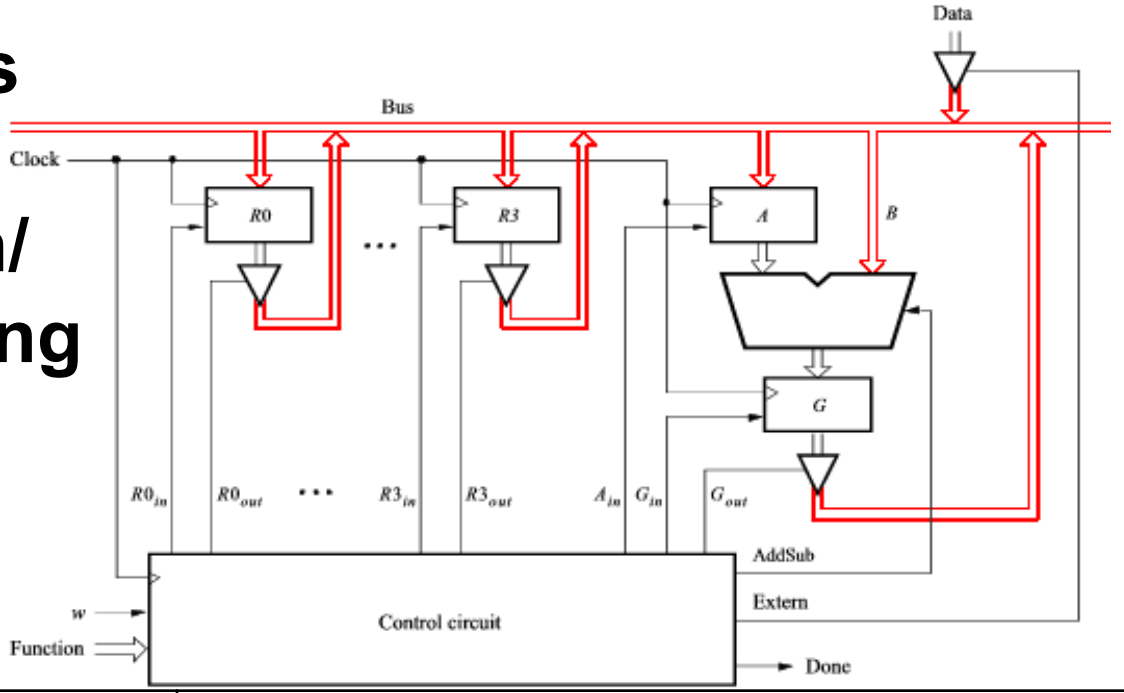


Operations performed by the processor

Operation	Function performed
Load Rx,$Data$	$Rx \leftarrow Data$
Move Rx, Ry	$Rx \leftarrow Ry$
Add Rx, Ry	$Rx \leftarrow Rx + Ry$
Sub Rx, Ry	$Rx \leftarrow Rx - Ry$



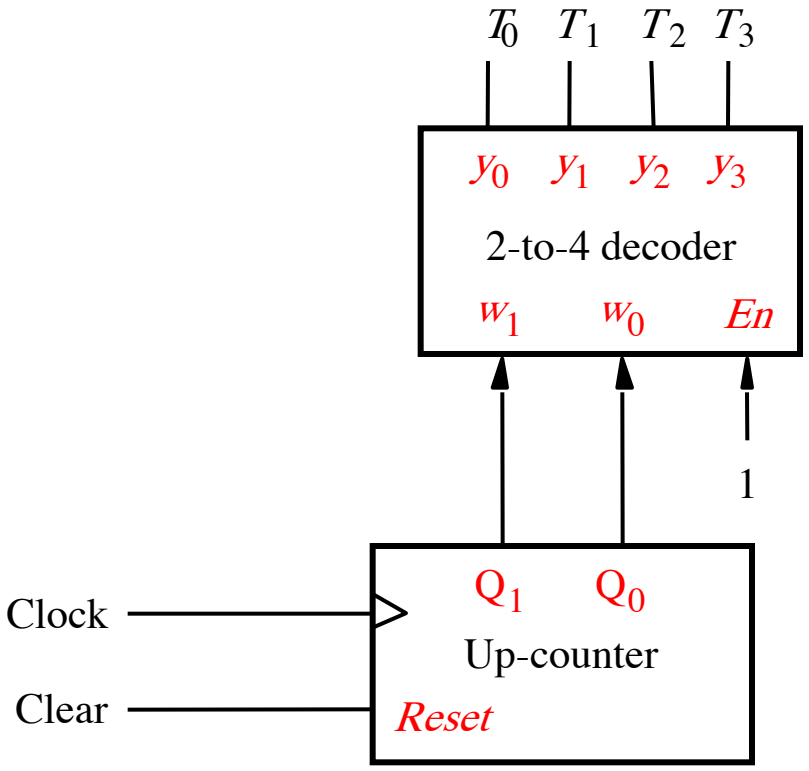
Control signals asserted in each operation/ time step (during each "micro-instruction")



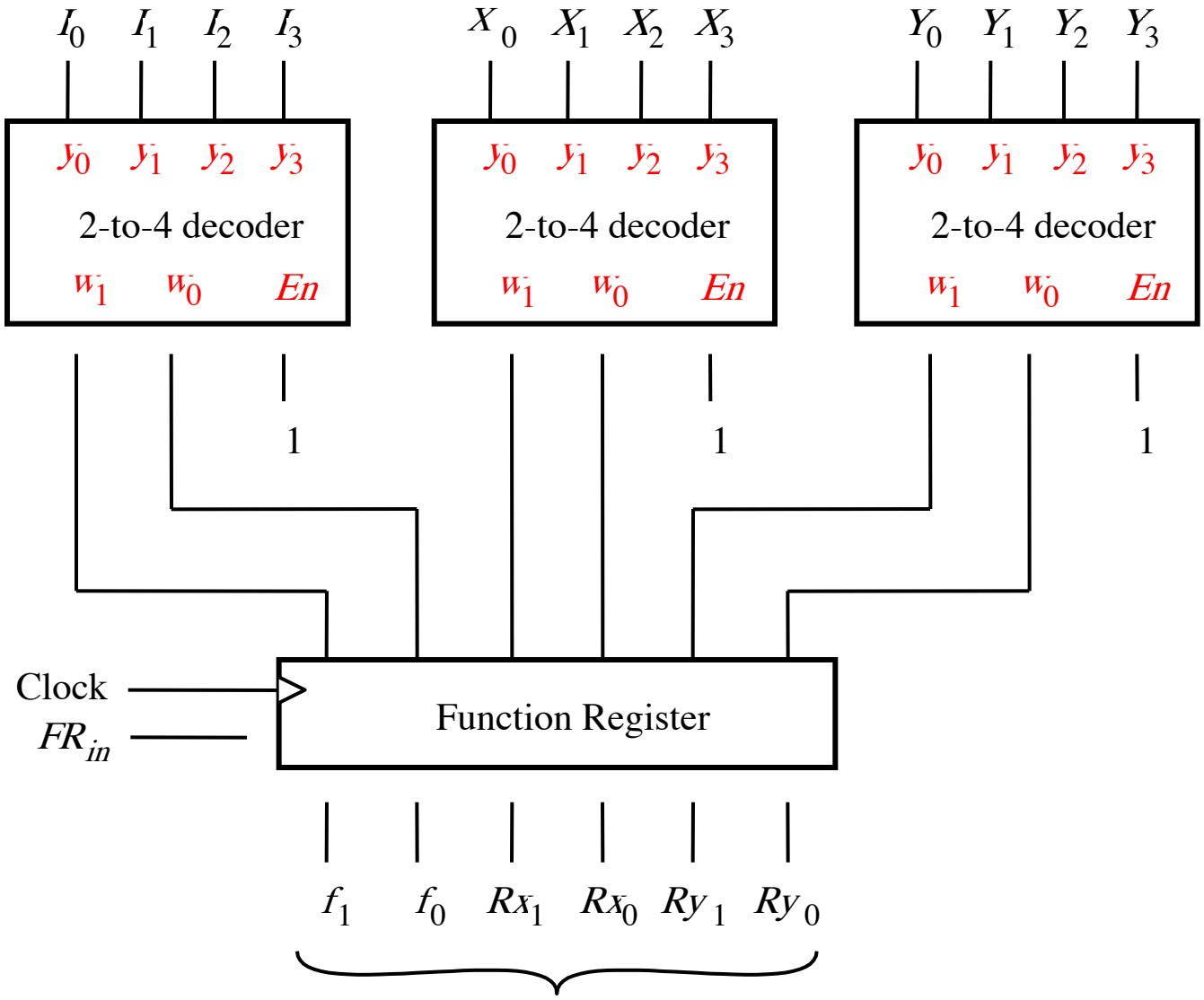
		Control Signals		
Operation	*Function*	*Time Step 1*	*Time Step 2*	*Time Step 3*
I_0: *Load* Rx, Data	Rx ← Data			
I_1: *Move* Rx, Ry	Rx ← Ry			
I_2: *Add* Rx, Ry	Rx ← Rx + Ry			
I_3: *Sub* Rx, Ry	Rx ← Rx – Ry			

Keeping track of the instruction step

- In Example 1 we used a shift register to keep track of which cycle we were in
- This time, let's use a counter, and decode it to obtain the instruction step
- $T_0 \Rightarrow$ not currently executing an instruction



In order to simplify derivation of control signals, save the function and decode its fields



Derivation of control equations (1)

a) Clear instruction counter when *Reset* or *Done* or when in T_0 and *w* not asserted: $Clear = \overline{w}T_0 + Done + Reset$

b) Load Function Register when *w* is asserted in T_0: $FR_{in} = wT_0$

c) All other control signals are derived from the table on slide L08/S18 depending upon the instruction being executed and the current time step e.g. Extern is only asserted in I_0 (*Load*) during T_1 i.e. $Extern = I_0T_1$

d) Done is asserted at the end of T_1 for *Load* & *Move* and at the end of T_3 for *Add* & *Sub*: $Done = (I_0 + I_1)T_1 + (I_2 + I_3)T_3$

		Control Signals		
Operation	*Function*	*Time Step 1*	*Time Step 2*	*Time Step 3*
I_0: *Load* Rx, Data	Rx ← Data	*Extern*, $R_{in} = X$, *Done*		
I_1: *Move* Rx, Ry	Rx ← Ry	$R_{in} = X$, $R_{out} = Y$, *Done*		
I_2: *Add* Rx, Ry	Rx ← Rx + Ry	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in}, *AddSub* = 0	G_{out}, $R_{in} = X$, *Done*
I_3: *Sub* Rx, Ry	Rx ← Rx – Ry	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in}, *AddSub* = 1	G_{out}, $R_{in} = X$, *Done*

Derivation of control equations (2)

e) $Ain = (I_2 + I_3)T_1$

$Gin = (I_2 + I_3)T_2$

$Gout = (I_2 + I_3)T_3$

$AddSub = I_3$

		Control Signals		
Operation	*Function*	*Time Step 1*	*Time Step 2*	*Time Step 3*
I_0: *Load* Rx, Data	Rx ← Data	*Extern,* $R_{in} = X$, *Done*		
I_1: *Move* Rx, Ry	Rx ← Ry	$R_{in} = X$, $R_{out} = Y$, *Done*		
I_2: *Add* Rx, Ry	Rx ← Rx + Ry	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in}, *AddSub = 0*	G_{out}, $R_{in} = X$, *Done*
I_3: *Sub* Rx, Ry	Rx ← Rx – Ry	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in}, *AddSub = 1*	G_{out}, $R_{in} = X$, *Done*

f) $R_0..R_3$ are determined from $X_0..X_3$ or $Y_0..Y_3$.

In the table from L08/S18, Rin = X means the register corresponding to the asserted X value should be loaded. Thus we can derive

$$R0_{in} = (I_0 + I_1)T_1X_0 + (I_2 + I_3)T_3X_0 \quad \text{and}$$

$$R0_{out} = I_1T_1Y_0 + (I_2 + I_3)(T_1X_0 + T_2Y_0)$$

Similar expressions can be derived for $R1_{in}..R3_{in}$ and $R1_{out}..R3_{out}$.

Code for a two-bit up-counter with synchronous reset

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY upcount IS
        PORT (   Clear, Clock   : IN            STD_LOGIC ;
                 Q              : BUFFER        STD_LOGIC_VECTOR(1 DOWNTO 0) ) ;
END upcount ;

ARCHITECTURE Behavior OF upcount IS
BEGIN
        upcount: PROCESS ( Clock )
        BEGIN
                IF (Clock'EVENT AND Clock = '1') THEN
                        IF Clear = '1' THEN
                                Q <= "00" ;
                        ELSE
                                Q <= Q + '1' ;
                        END IF ;
                END IF;
        END PROCESS;
END Behavior ;
```

Code for the processor (Part *a*)

- This code is for a 3-state, bus-based processor

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;
USE work.subccts.all ;

ENTITY proc IS
        PORT ( Data: IN STD_LOGIC_VECTOR(7 DOWNTO 0) ;
                Reset, w : IN STD_LOGIC ;
                Clock: IN STD_LOGIC ;
                F, Rx, Ry: IN STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                Done: BUFFER STD_LOGIC ;
                BusWires: BUFFER STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END proc ;

ARCHITECTURE Mixed OF proc IS
        SIGNAL Rin, Rout : STD_LOGIC_VECTOR(0 TO 3) ;
        SIGNAL Clear, High, AddSub : STD_LOGIC ;
        SIGNAL Extern, Ain, Gin, Gout, FRin : STD_LOGIC ;
        SIGNAL Count : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
        SIGNAL T, I, X, Y : STD_LOGIC_VECTOR(0 TO 3) ;
        SIGNAL R0, R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
        SIGNAL A, Sum, G : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
        SIGNAL Func, FuncReg : STD_LOGIC_VECTOR(1 TO 6) ;
```

```
BEGIN
        High <= '1' ;
        Clear <= Reset OR Done OR (NOT w AND T(0)) ;
        counter: upcount PORT MAP ( Clear, Clock, Count ) ;
        decT: dec2to4 PORT MAP ( Count, High, T );
        Func <= F & Rx & Ry ;
        FRin <= w AND T(0) ;
        functionreg: regn GENERIC MAP ( N => 6 )
                        PORT MAP ( Func, FRin, Clock, FuncReg ) ;
        decI: dec2to4 PORT MAP ( FuncReg(1 TO 2), High, I ) ;
        decX: dec2to4 PORT MAP ( FuncReg(3 TO 4), High, X ) ;
        decY: dec2to4 PORT MAP ( FuncReg(5 TO 6), High, Y ) ;

        Extern <= I(0) AND T(1) ;
        Done <= ((I(0) OR I(1)) AND T(1)) OR ((I(2) OR I(3)) AND T(3));
        Ain <= (I(2) OR I(3)) AND T(1) ;
        Gin <= (I(2) OR I(3)) AND T(2) ;
        Gout <= (I(2) OR I(3)) AND T(3) ;
        AddSub <= I(3) ;

        … continued in Part b.
```

Controller (continued on next page)

Code for the processor (Part *b*)

```
    RegCntl:                                                    Controller
    FOR k IN 0 TO 3 GENERATE
        Rin(k) <= ((I(0) OR I(1)) AND T(1) AND X(k)) OR
            ((I(2) OR I(3)) AND T(3) AND X(k)) ;
        Rout(k) <= (I(1) AND T(1) AND Y(k)) OR
            ((I(2) OR I(3)) AND ((T(1) AND X(k)) OR (T(2) AND Y(k)))) ;
    END GENERATE RegCntl ;
    tri_extern: trin PORT MAP ( Data, Extern, BusWires ) ;
    reg0: regn PORT MAP ( BusWires, Rin(0), Clock, R0 ) ;
    reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
    reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
    reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
    tri0: trin PORT MAP ( R0, Rout(0), BusWires ) ;
    tri1: trin PORT MAP ( R1, Rout(1), BusWires ) ;
    tri2: trin PORT MAP ( R2, Rout(2), BusWires ) ;
    tri3: trin PORT MAP ( R3, Rout(3), BusWires ) ;
    regA: regn PORT MAP ( BusWires, Ain, Clock, A ) ;
    alu:
    WITH AddSub SELECT
        Sum <= A + BusWires WHEN '0',
               A - BusWires WHEN OTHERS ;
    regG: regn PORT MAP ( Sum, Gin, Clock, G ) ;
    triG: trin PORT MAP ( G, Gout, BusWires ) ;                 Datapath
END Mixed ;
```

Alternative code for a MUX-based processor (Part *a*)

- Uses the same entity description as before
- Note that the table from slide L08/S18 is implemented directly

T, I not decoded to simplify CASE selection in controlsignals PROCESS

ALL outputs cleared by default at start of PROCESS to avoid risk of implying memory

```
ARCHITECTURE Mixed OF proc IS
    SIGNAL X, Y, Rin, Rout : STD_LOGIC_VECTOR(0 TO 3) ;
    SIGNAL Clear, High, AddSub : STD_LOGIC ;
    SIGNAL Extern, Ain, Gin, Gout, FRin : STD_LOGIC ;
    SIGNAL Count, T, I : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
    SIGNAL R0, R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0);
    SIGNAL A, Sum, G : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL Func, FuncReg, Sel : STD_LOGIC_VECTOR(1 TO 6);
BEGIN
    High <= '1' ;
    Clear <= Reset OR Done
             OR (NOT w AND NOT T(1) AND NOT T(0)) ;
    counter: upcount PORT MAP ( Clear, Clock, Count ) ;
    T <= Count ;
    Func <= F & Rx & Ry ;
    FRin <= w AND NOT T(1) AND NOT T(0) ;
    functionreg: regn GENERIC MAP ( N => 6 )
                      PORT MAP ( Func, FRin, Clock, FuncReg );
    I <= FuncReg(1 TO 2) ;
    decX: dec2to4 PORT MAP ( FuncReg(3 TO 4), High, X ) ;
    decY: dec2to4 PORT MAP ( FuncReg(5 TO 6), High, Y ) ;
```

```
controlsignals: PROCESS ( T, I, X, Y )
BEGIN
    Extern <= '0' ; Done <= '0' ; Ain <= '0' ; Gin <= '0' ;
    Gout <= '0' ; AddSub <= '0' ; Rin <= "0000" ; Rout <= "0000" ;
    CASE T IS WHEN "00" => -- no signals asserted in time step T0
    WHEN "01" => -- define signals asserted in time step T1
        CASE I IS
            WHEN "00" => -- Load
                Extern <= '1' ; Rin <= X ; Done <= '1' ;
            WHEN "01" => -- Move
                Rout <= Y ; Rin <= X ; Done <= '1' ;
            WHEN OTHERS => -- Add, Sub
                Rout <= X ; Ain <= '1' ;
        END CASE ;
```

… continued in Part *b*

Controller

Alternative code for a MUX-based processor (Part *b*)

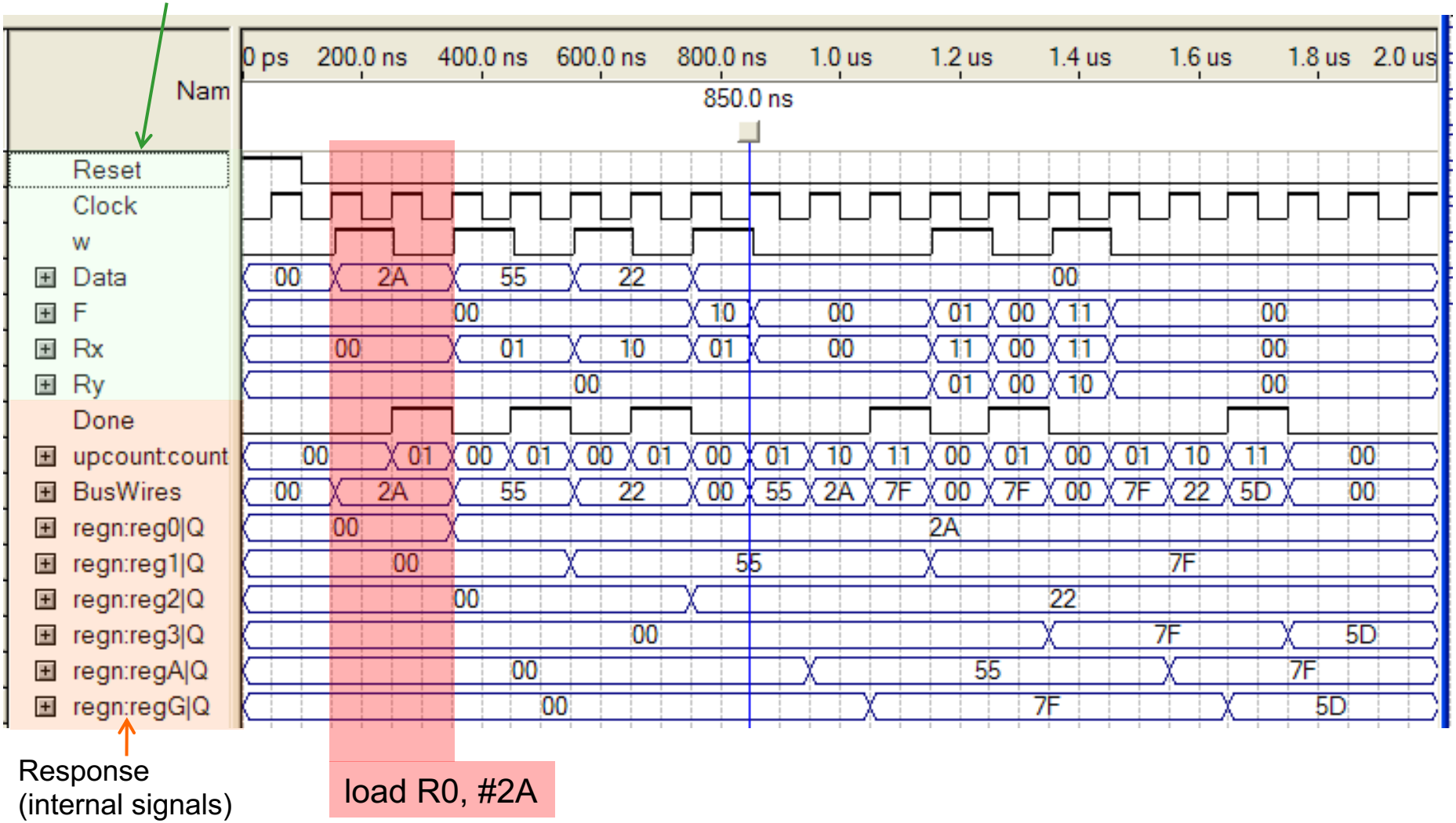
```
        WHEN "10" => -- define signals asserted in time step T2
                CASE I IS
                        WHEN "10" => -- Add
                                Rout <= Y ; Gin <= '1' ;
                        WHEN "11" => -- Sub
                                Rout <= Y ; AddSub <= '1' ; Gin <= '1' ;
                        WHEN OTHERS => -- Load, Move
                END CASE ;
        WHEN OTHERS => -- define signals asserted in time step T3
                CASE I IS
                        WHEN "00" => -- Load
                        WHEN "01" => -- Move
                        WHEN OTHERS => -- Add, Sub
                                Gout <= '1' ; Rin <= X ; Done <= '1' ;
                END CASE ;
        END CASE ;                                          Controller
END PROCESS ;
```

```
        reg0: regn PORT MAP ( BusWires, Rin(0), Clock, R0 ) ;
        reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
        reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
        reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
        regA: regn PORT MAP ( BusWires, Ain, Clock, A ) ;
        alu: WITH AddSub SELECT
                Sum <= A + BusWires WHEN '0',
                       A - BusWires WHEN OTHERS ;
        regG: regn PORT MAP ( Sum, Gin, Clock, G ) ;
        Sel <= Rout & Gout & Extern ;
        WITH Sel SELECT
                BusWires <=     R0 WHEN "100000",
                                R1 WHEN "010000",
                                R2 WHEN "001000",
                                R3 WHEN "000100",
                                G WHEN "000010",
    Datapath                    Data WHEN OTHERS ;
END Mixed ;
```

- Both versions have equivalent functionality
- However, the behavioural style used to capture the control signalling in the second version is less prone to error in deriving and coding the control equations

Functional simulation of the MUX-based processor

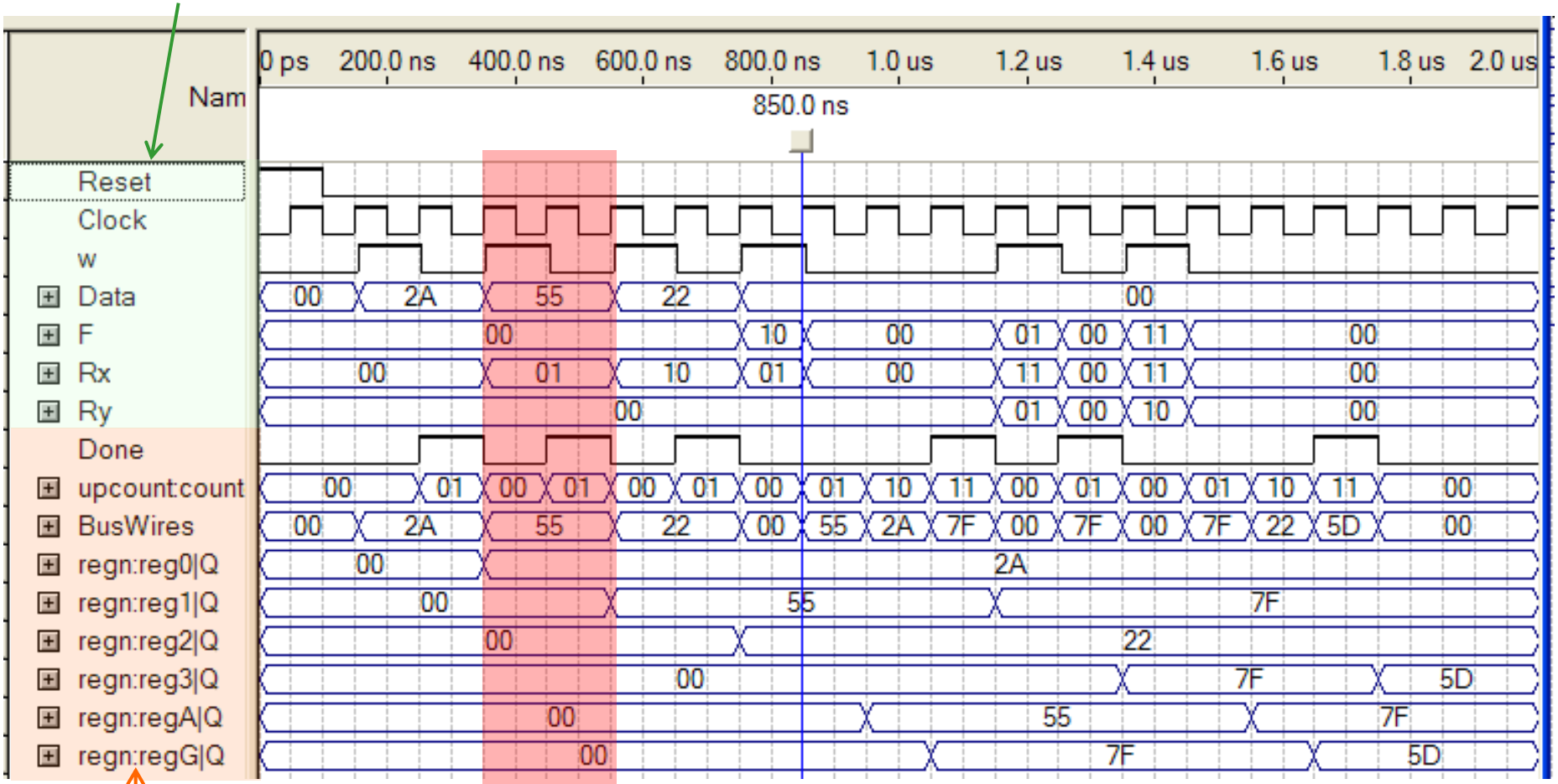
- Code and waveform inputs to generate this waveform are available from the course website



Functional simulation of the MUX-based processor

- Code and waveform inputs to generate this waveform are available from the course website

Stimuli (external signals)



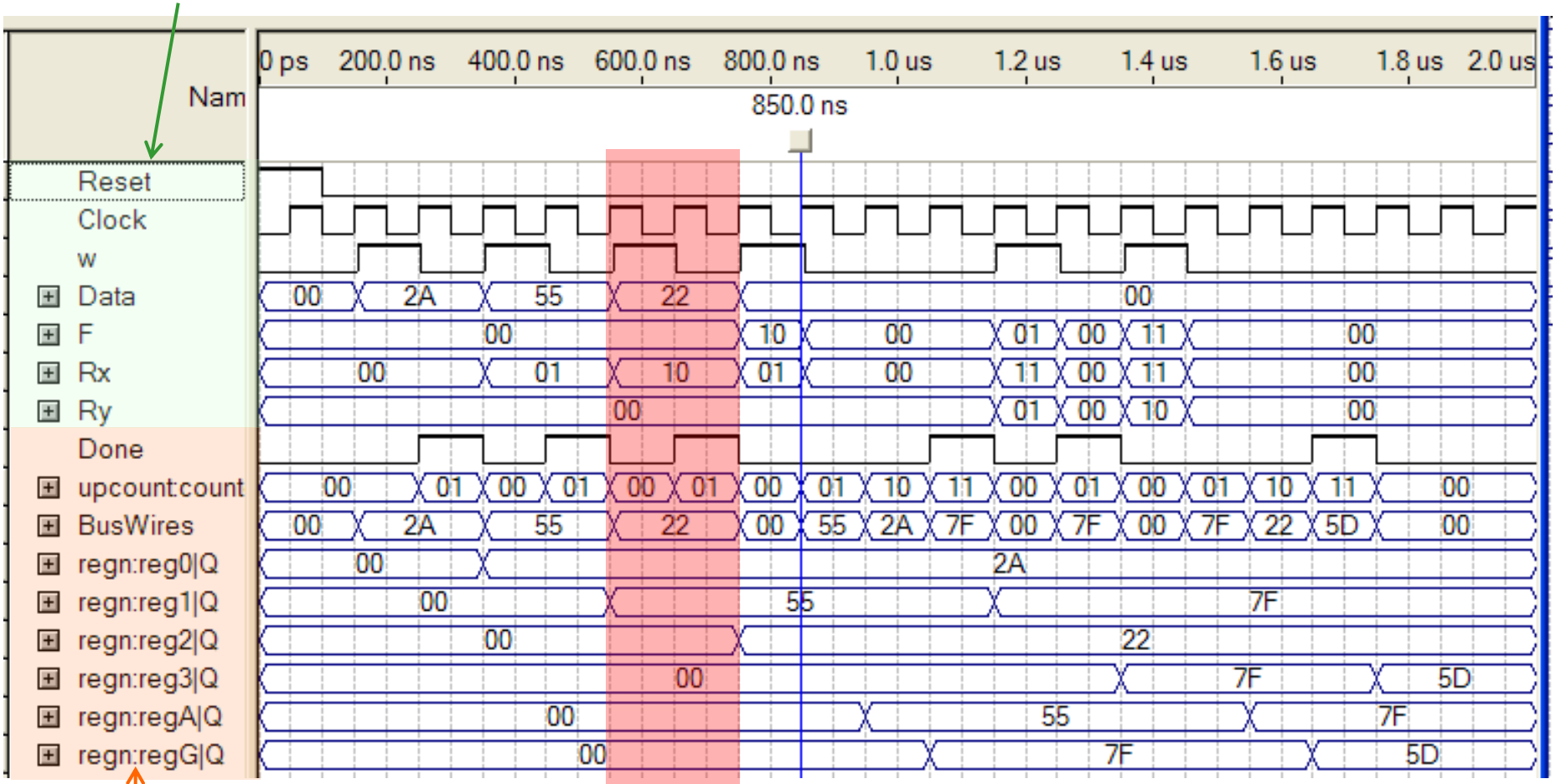
Response (internal signals)

load R1, #55

Functional simulation of the MUX-based processor

- Code and waveform inputs to generate this waveform are available from the course website

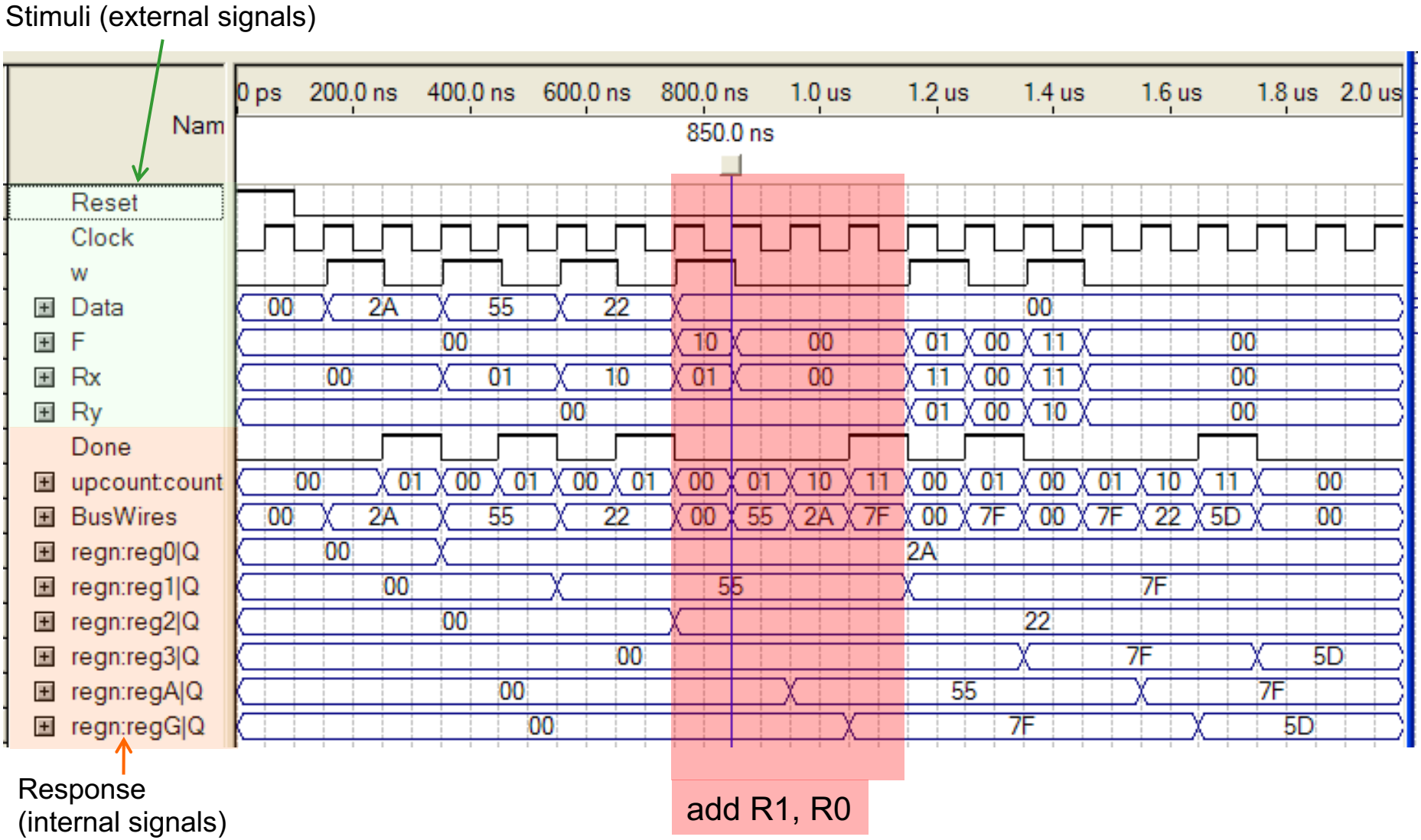
Stimuli (external signals)



Response
(internal signals)

Functional simulation of the MUX-based processor

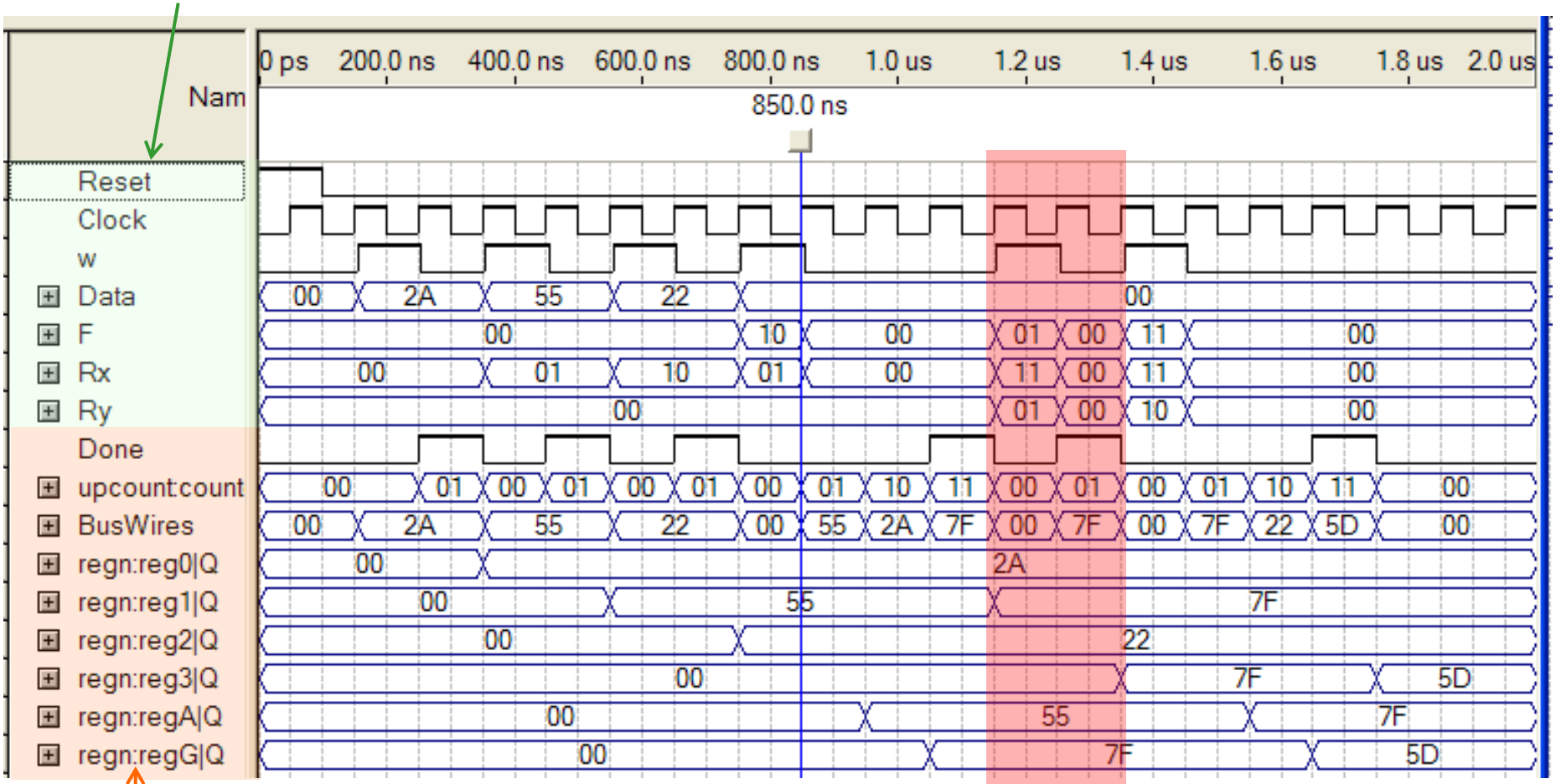
- Code and waveform inputs to generate this waveform are available from the course website



Functional simulation of the MUX-based processor

- Code and waveform inputs to generate this waveform are available from the course website

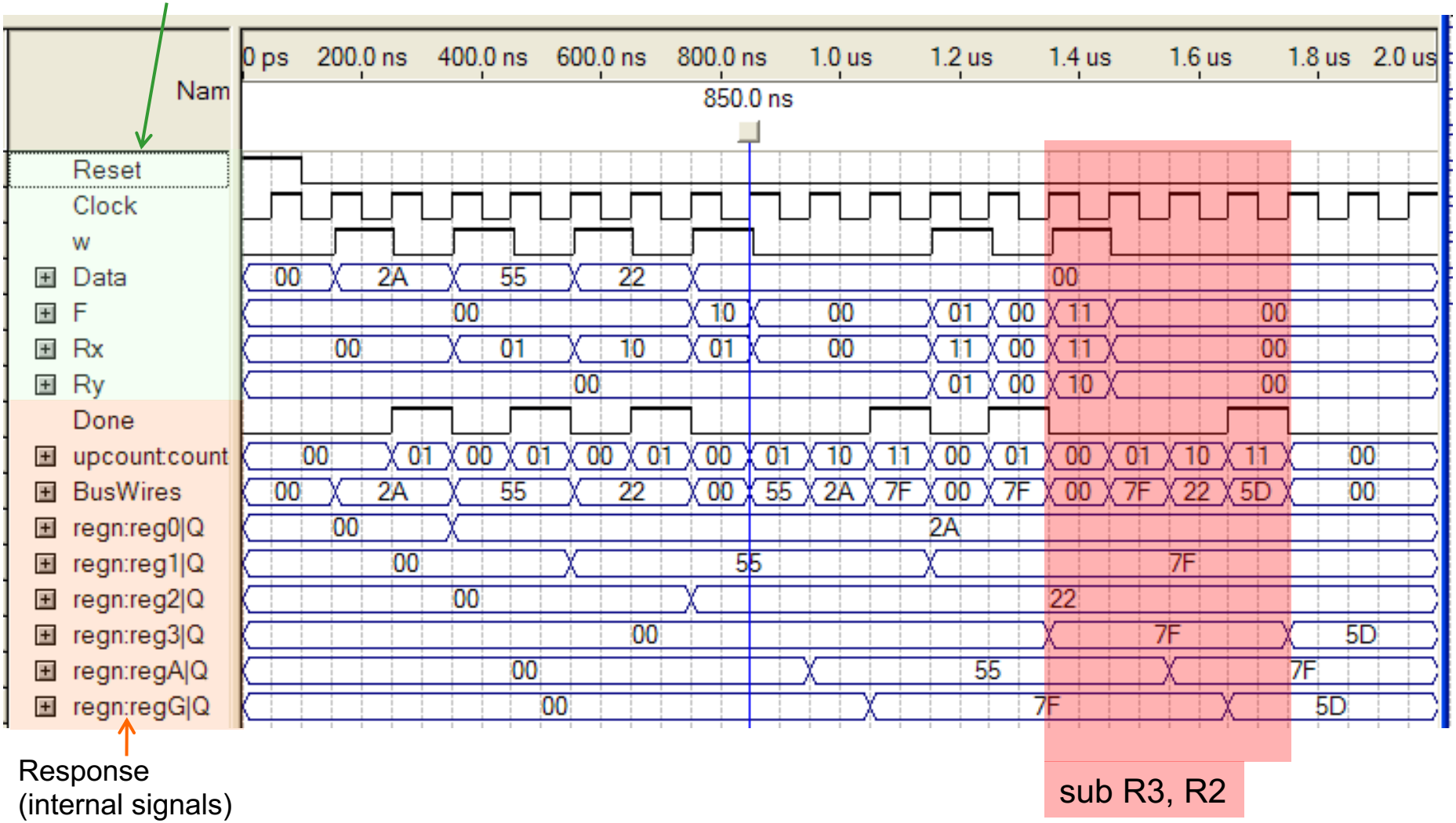
Stimuli (external signals)



Response
(internal signals)

Functional simulation of the MUX-based processor

- Code and waveform inputs to generate this waveform are available from the course website

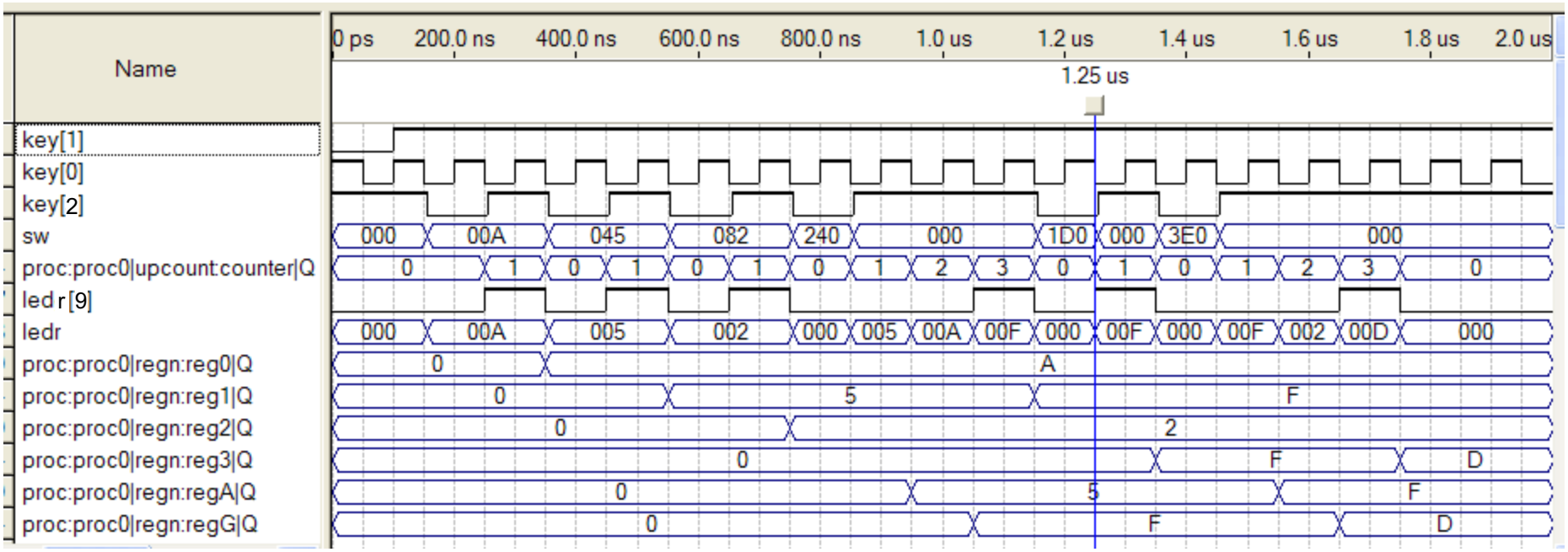


Mapping the processor to our board

```
library ieee;
use ieee.std_logic_1164.all;
use work.subccts.all;
use work.proc_pkg.all;

entity mapproc is
    port (sw: in std_logic_vector(9 downto 0);
        key: in std_logic_vector(2 downto 0);
        ledr: out std_logic_vector(9 downto 0); --DE1
        hex0: out std_logic_vector(0 to 6);
        hex1: out std_logic_vector(0 to 6);
        hex2: out std_logic_vector(0 to 6);
        hex3: out std_logic_vector(0 to 6));
end mapproc;

architecture structural of mapproc is
    signal done, reset, w, clock: std_logic;
    signal buswires: std_logic_vector(3 downto 0);
    signal r0,r1,r2,r3: std_logic_vector(3 downto 0);
```

```
begin
    clock <= not key(0);
    reset <= not key(1);
    w <= not key(2);
    ledr(9) <= done; --DE1
    ledr(3 downto 0) <= buswires; --DE1
    proc0: proc
        generic map (L => 4)
        port map(Data => sw(3 downto 0),
            Reset => reset, w => w,
            Clock => clock,
            F => sw(9 downto 8),
            RX => sw(7 downto 6),
            RY => sw(5 downto 4),
            Done => done, BusWires => buswires,
            R0 => r0, R1 => r1, R2 => r2,
            R3 => r3);
    dig0: seg7 PORT map (  r0, hex0);
    dig1: seg7 PORT map (  r1, hex1);
    dig2: seg7 PORT map (  r2, hex2);
    dig3: seg7 PORT map (  r3, hex3);
end structural;
```

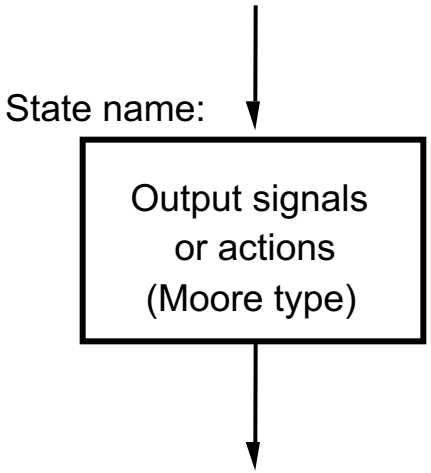
Simulating the mapping



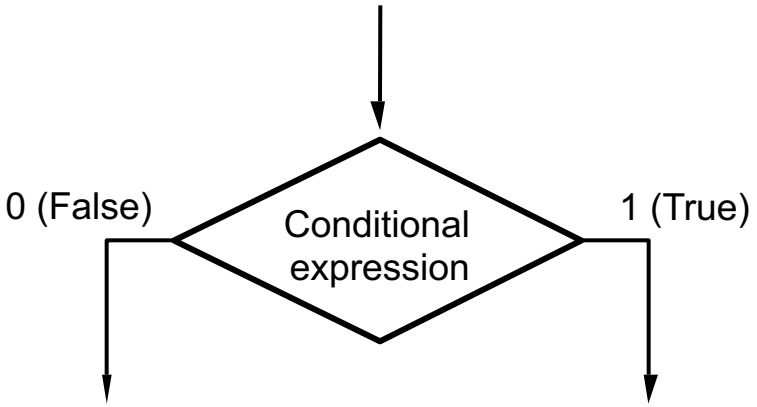
Algorithmic state machines

- Algorithmic state machines (ASMs) are a type of flowchart
  - They are used to represent more complex (larger) FSMs that are impractical to represent using state diagrams and state tables
  - They can be used to represent the state transitions and generated outputs for an FSM
- There are three types of elements in ASM charts

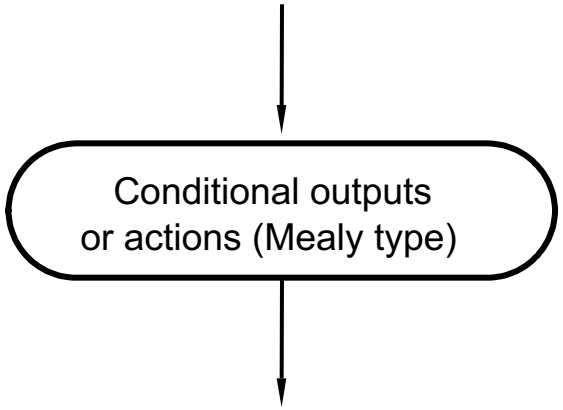
Elements used in ASM charts



(a) Named state box



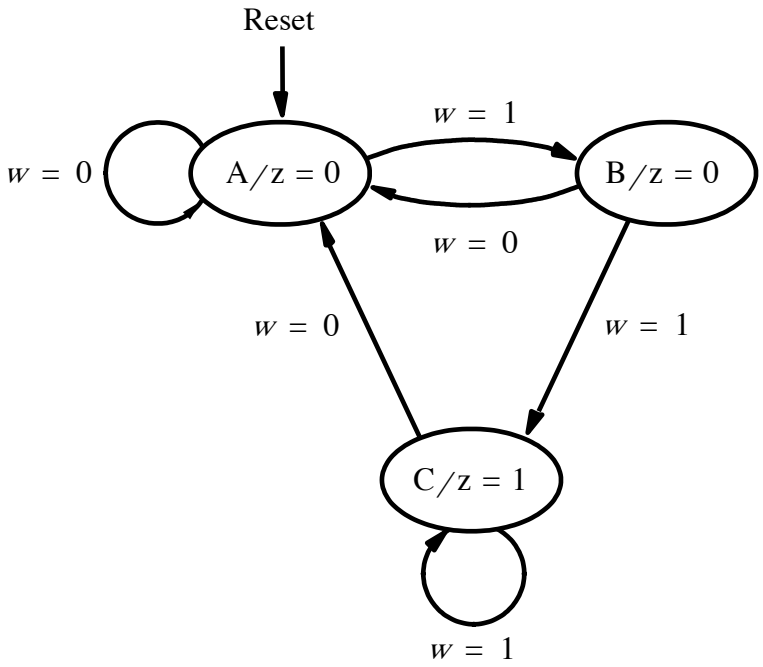
(b) Decision box

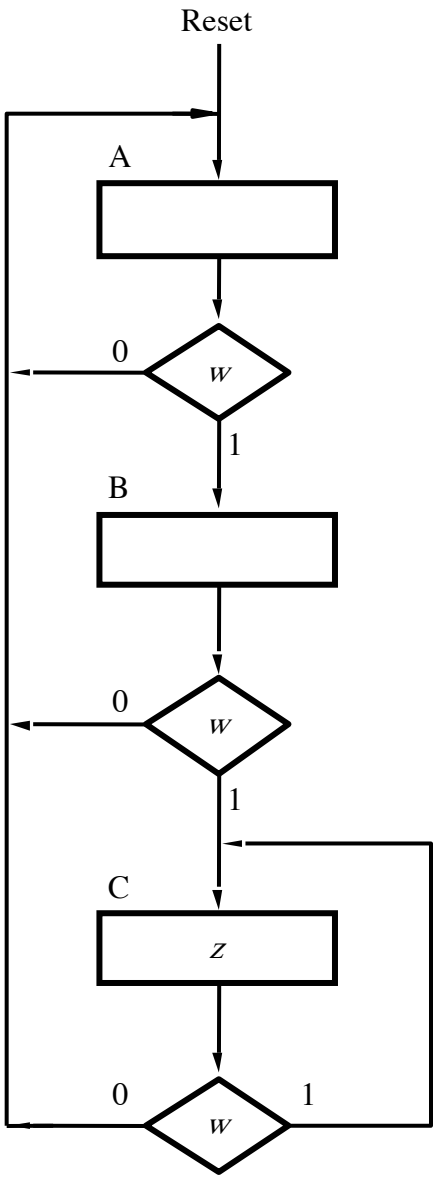


(c) Conditional output box

B&V3, Figure 8.86

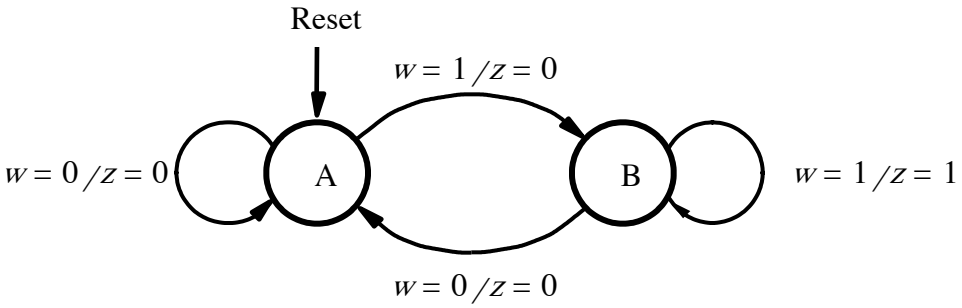
ASM chart for the FSM of L06/S10

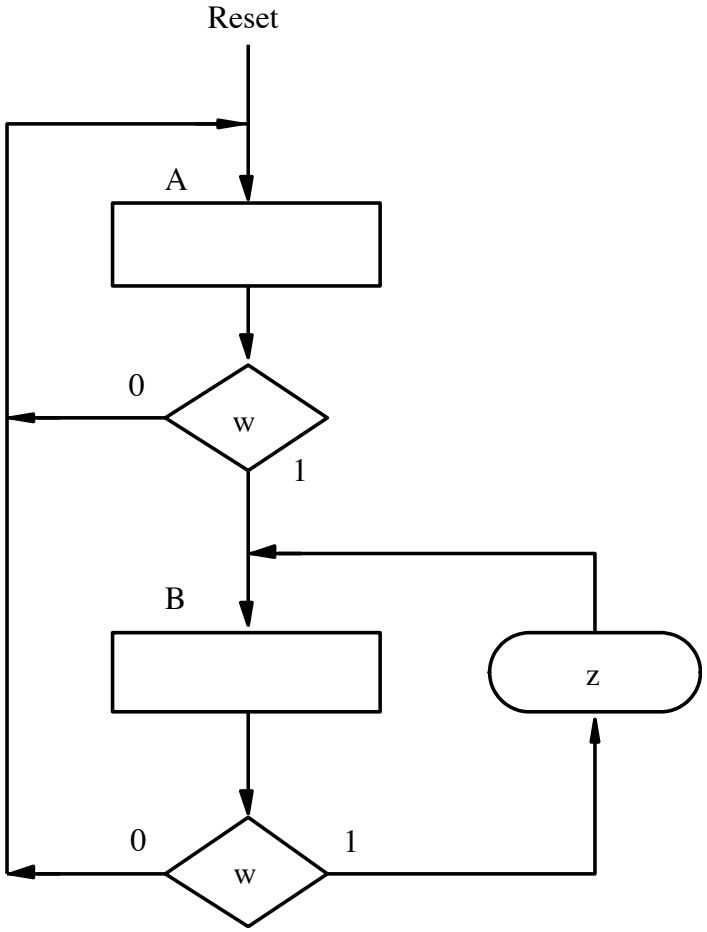




B&V3, Figure 8.87

ASM chart for the FSM of L06/S23





B&V3, Figure 8.88

Design Exercise 1:
pp 679 – 683, B&V3

- Devise a circuit that will count the number of ON bits in a data word

Pseudo-code for a bit counter (popcount)

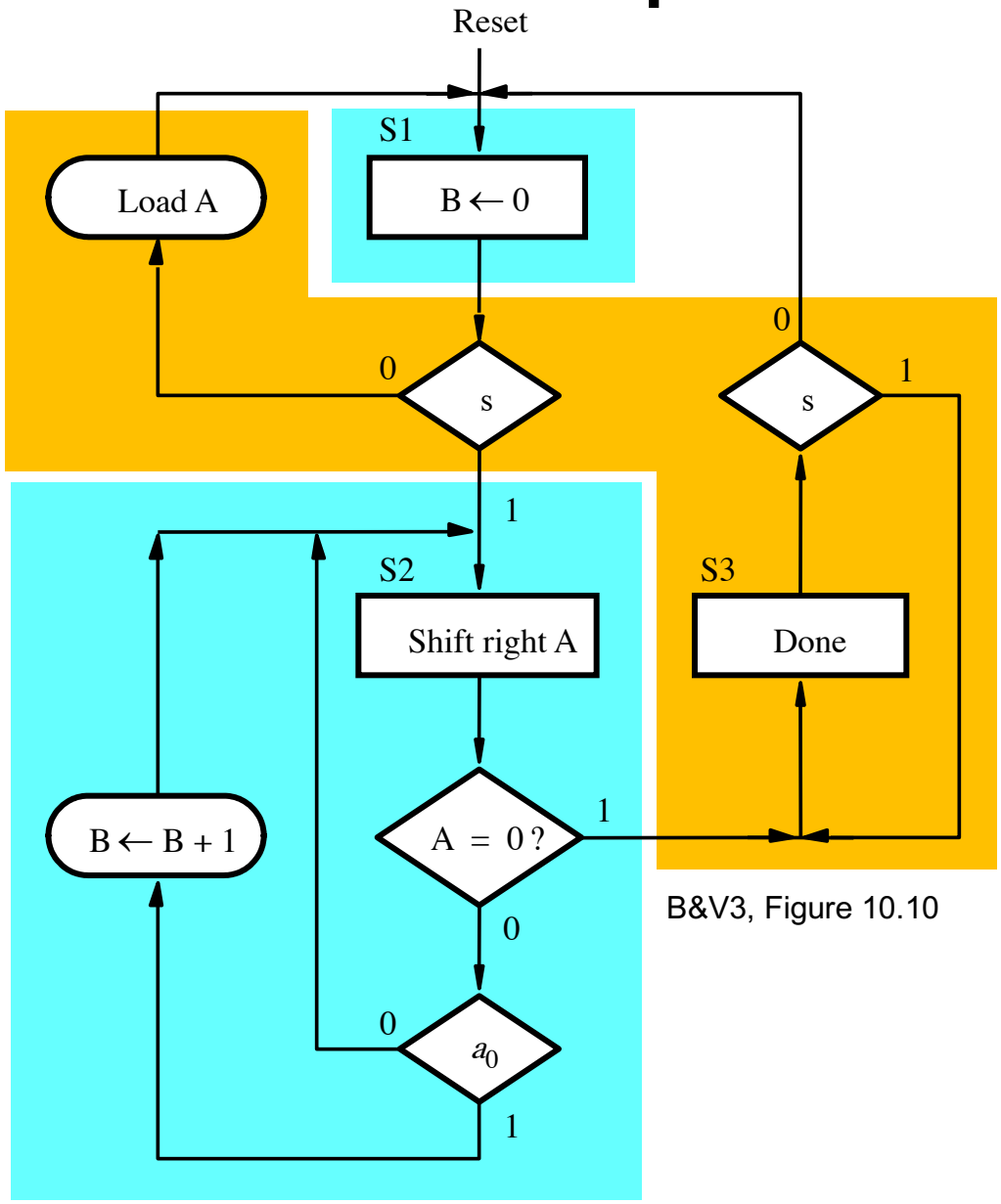
input: *A* -- the word whose ON bits are to be counted
output: *B* -- the count of the number of ON bits in A

```
B = 0
while A ≠ 0 do
    if a0 = 1 then
        B = B + 1
    end if
    right shift A
end while
```

1. What datapath components do we need to perform the computation?
2. How do we control the computation?
3. How do we transfer inputs/outputs?

B&V3, Figure 10.9

ASM chart for the pseudo-code



B&V3, Figure 10.10

```
B = 0
while A ≠ 0 do
    if a0 = 1 then
        B = B + 1
    end if
    right shift A
end while
```

Note that the ASM chart describes control and datapath aspects of the system in an integrated way

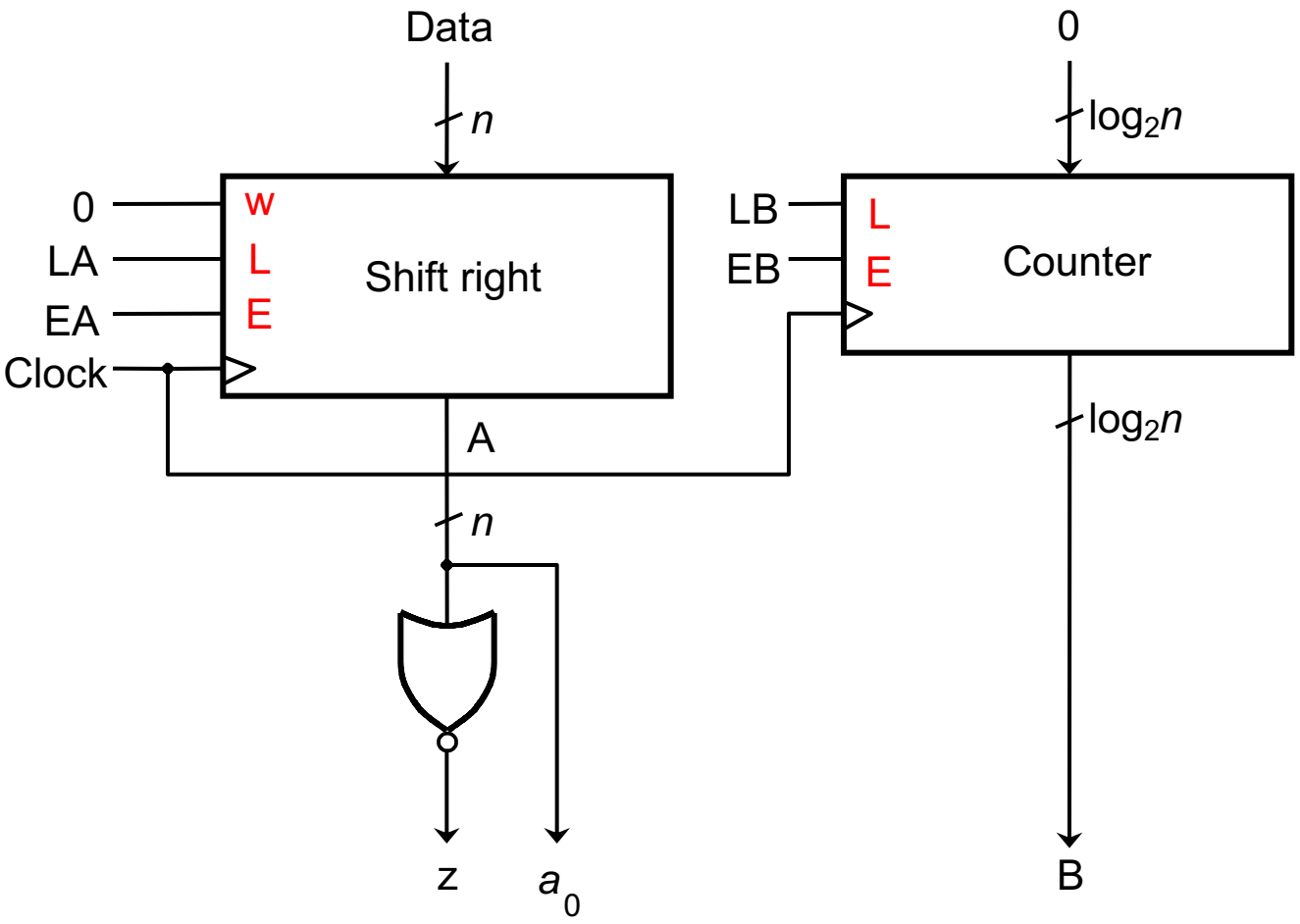
Note use of a "start" signal, s, to indicate when input is available, and a Done signal to indicate when computation has finished

⇒ handshake protocol used to communicate with the environment or "user" circuit

Take careful note of the state actions – particularly for S2

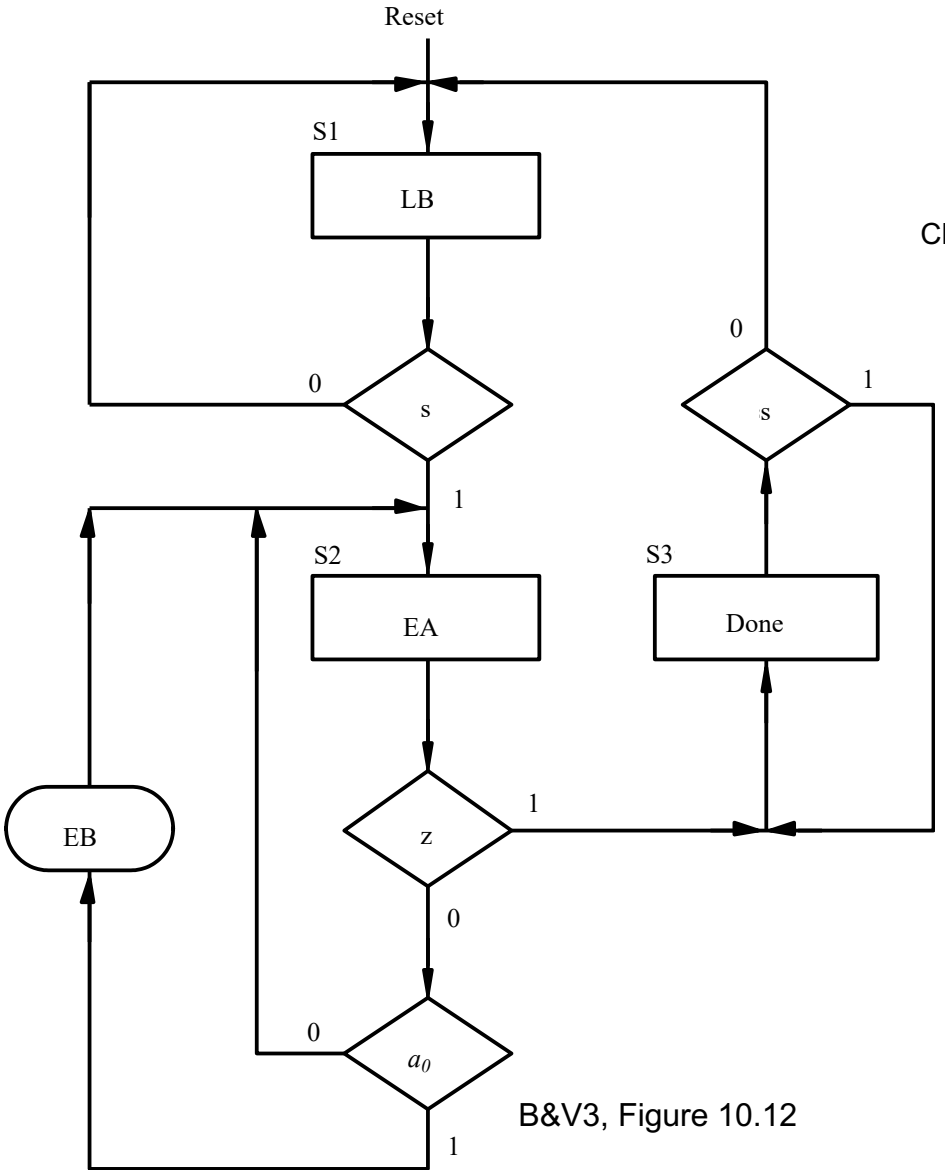
⇒ since the "Shift right" action is a Moore-like state output, it won't occur until the first active clock edge after S2 has been entered, even if that edge causes transition to S3

Datapath for the ASM chart

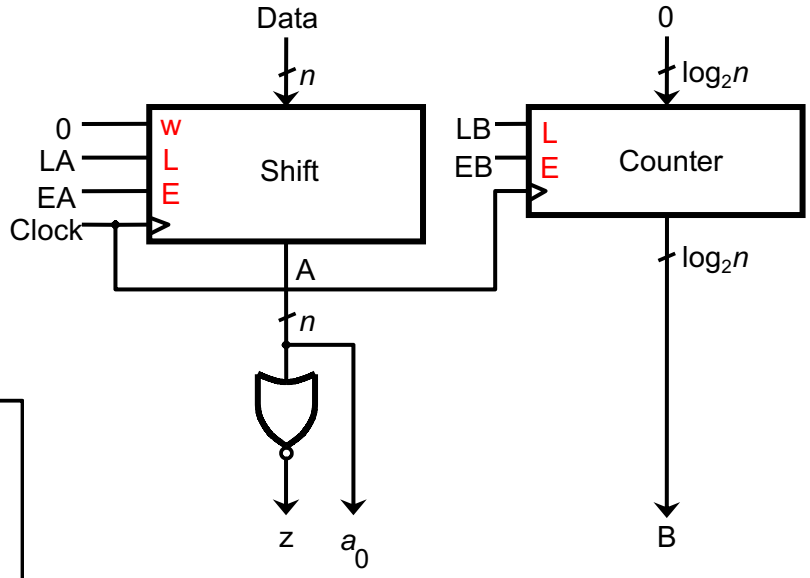


B&V3, Figure 10.11

ASM chart for the bit counter control circuit



B&V3, Figure 10.12

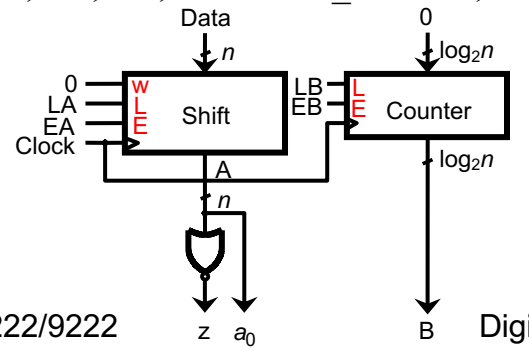


The ASM chart for the pseudocode is refined according to the FSM reqs.

Note: It is assumed that the circuit "user" loads A by asserting LA before asserting the start signal s, and has not completed reading the counter value (after Done is asserted) until after s is deasserted.

VHDL code for the bit-counting circuit (Part *a*)

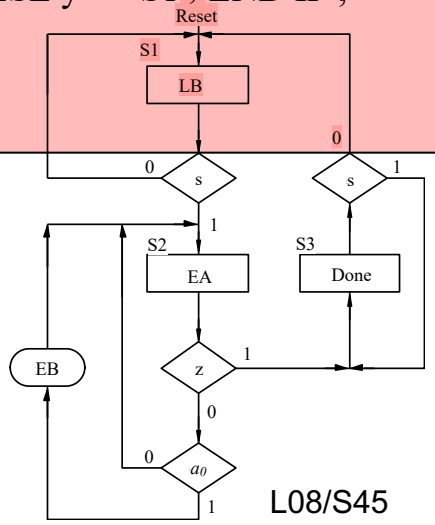
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;
USE work.components.shiftrne ;

ENTITY bitcount IS
    PORT( Clock, Resetn    : IN STD_LOGIC ;
          LA, s            : IN STD_LOGIC ;
          Data             : IN
                STD_LOGIC_VECTOR(7 DOWNTO 0) ;
          B                : BUFFER
                INTEGER RANGE 0 to 8 ;
          Done             : OUT STD_LOGIC ) ;
END bitcount ;

ARCHITECTURE Behavior OF bitcount IS
    TYPE State_type IS ( S1, S2, S3 ) ;
    SIGNAL y : State_type ;
    SIGNAL A : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL z, EA, LB, EB, low : STD_LOGIC ;
```




```
BEGIN
    FSM_transitions: PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= S1 ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN S1 =>
                    IF s = '0' THEN y <= S1 ;
                    ELSE y <= S2 ; END IF ;
                WHEN S2 =>
                    IF z = '0' THEN y <= S2 ;
                    ELSE y <= S3 ; END IF ;
                WHEN S3 =>
                    IF s = '1' THEN y <= S3 ;
                    ELSE y <= S1 ; END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;
```

… continued in Part *b*

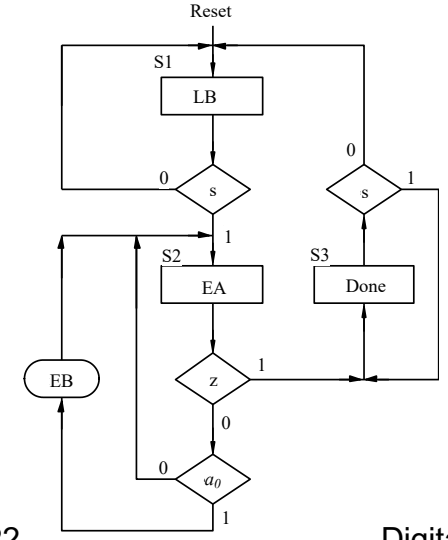
B&V3, Figure 10.13



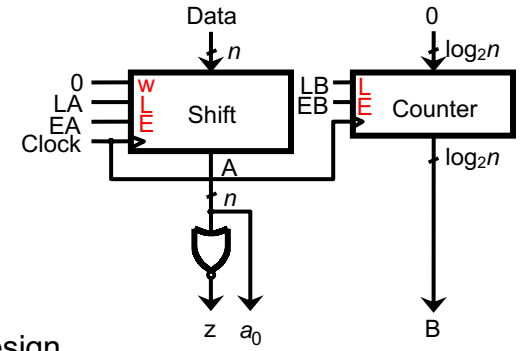
VHDL code for the bit-counting circuit (Part *b*)

```
FSM_outputs: PROCESS ( y, A(0) )
BEGIN
    EA <= '0' ; LB <= '0' ; EB <= '0' ; Done <= '0' ;
    CASE y IS
        WHEN S1 =>
            LB <= '1'
        WHEN S2 =>
            EA <= '1' ;
            IF A(0) = '1' THEN EB <= '1' ;
            END IF ;
        WHEN S3 =>
            Done <= '1' ;
    END CASE ;
END PROCESS ;
```

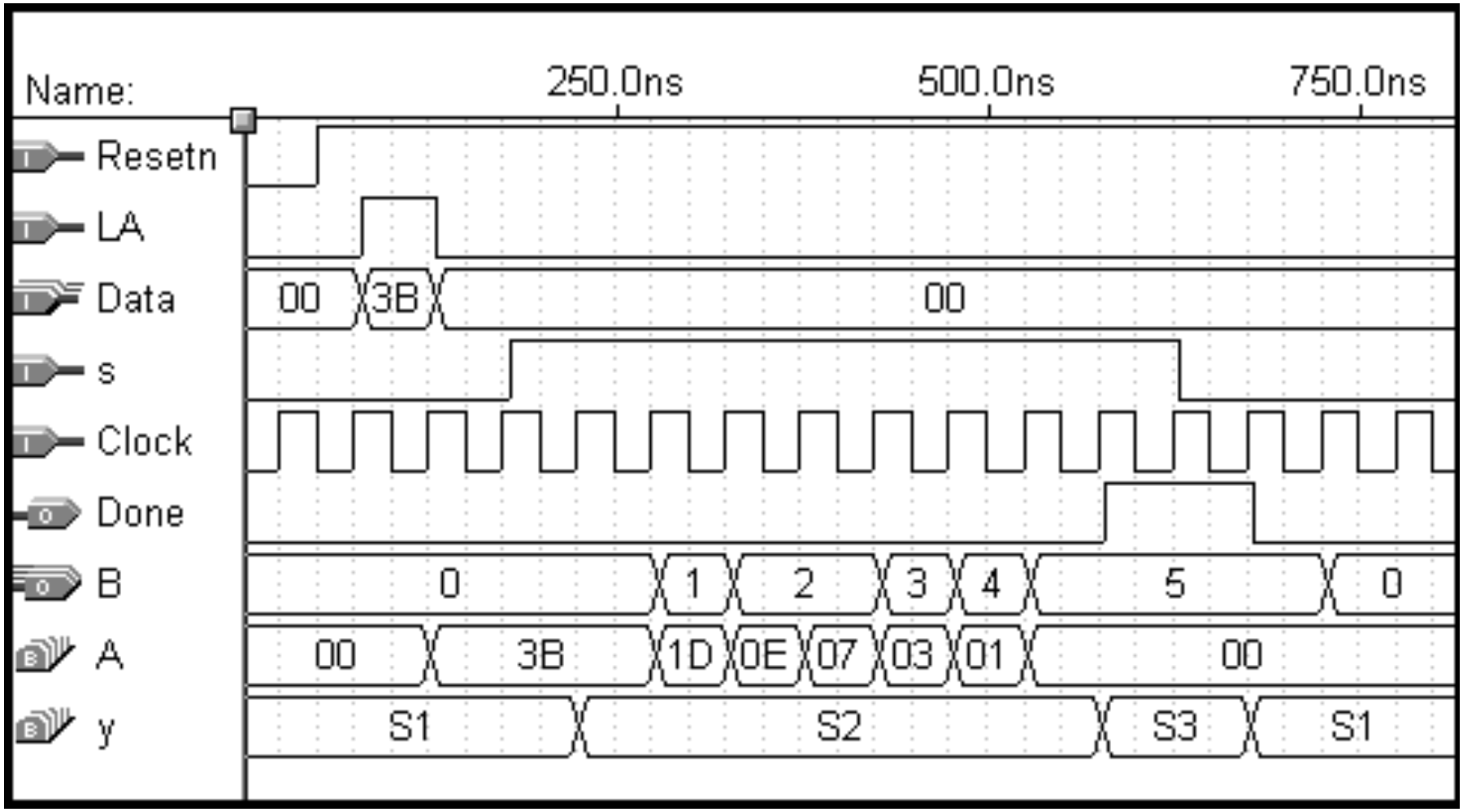
B&V3, Figure 10.13




```
-- The datapath circuit is described below
upcount: PROCESS ( Resetn, Clock )
BEGIN
    IF Resetn = '0' THEN
        B <= 0 ;
    ELSIF (Clock'EVENT AND Clock = '1') THEN
        IF LB = '1' THEN
            B <= 0 ;
        ELSIF EB = '1' THEN
            B <= B + 1 ;
        END IF ;
    END IF;
END PROCESS;

low <= '0' ;
ShiftA: shiftrne GENERIC MAP ( N => 8 )
    PORT MAP ( Data, LA, EA, low, Clock, A ) ;
z <= '1' WHEN A = "00000000" ELSE '0' ;
END Behavior ;
```

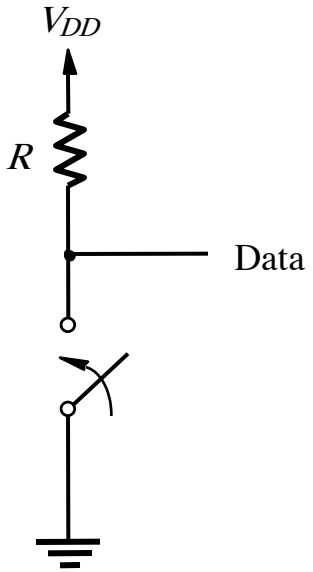


Simulation results for the bit-counting circuit



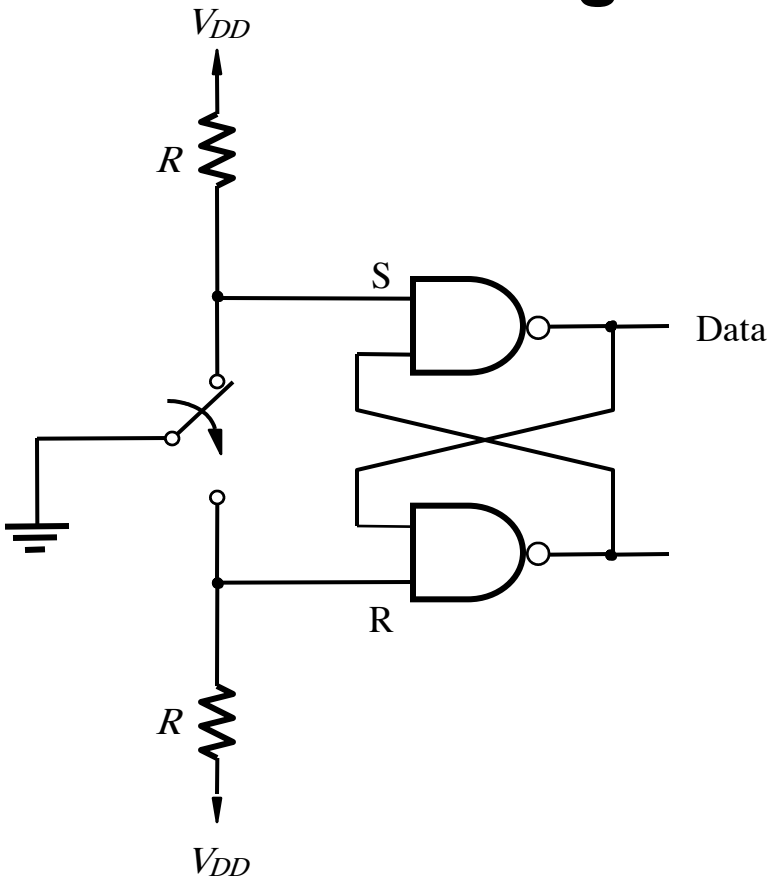
B&V3, Figure 10.14

Practical Issue: Input switch debouncing



(a) Single-pole single-throw switch

B&V3, Figure 10.48



(b) Single-pole double-throw switch with a basic SR latch

When an input switch is thrown, it can bounce for up to 10ms and thus give rise to an undesirable sequence of pulses on Data.

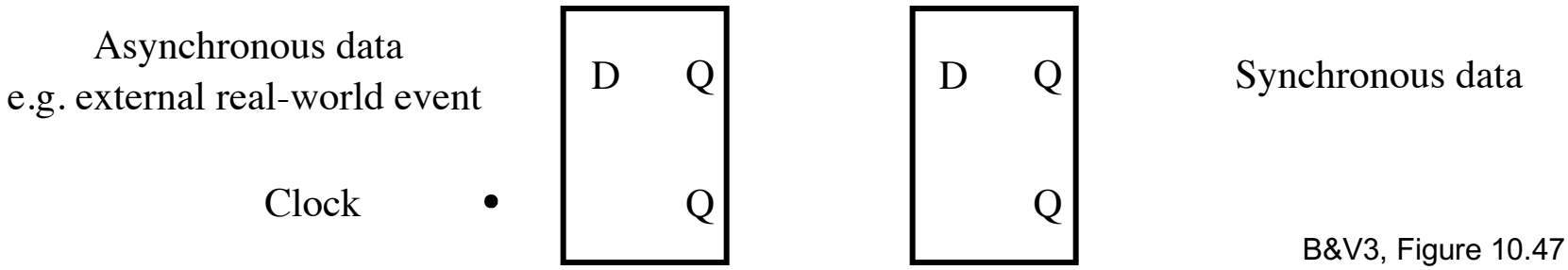
⇒ One approach to avoiding misreads is to use a latch to trap the switch value.

⇒ Another is to wait that period before sampling Data.

Debounce code

```
synchronise: process (clk)
begin
      wait until clk'event and clk = '1';          -- rising clock edge
      input_prev <= input_switch;                  -- save the current switch setting

-- The following counter counts time that the inputs have been steady
-- The input signal must be steady for approx. 10 milliseconds

      if input_switch /= input_prev then           -- if the switch has bounced
            sync_count <= (others => '0');         --      reset a counter
      elsif sync_count /= x”80000" then            -- otherwise, count ~10ms worth of clock
            sync_count <= sync_count + 1;          --      pulses at 50MHz (assumed clock freq.)
      end if;

-- If the full time is reached, update the input signals

      if sync_count = x”80000" then
            input_value <= input_switch;           -- switch has stopped bouncing
      end if;
end process synchronise;
```

Practical Issue: Asynchronous inputs



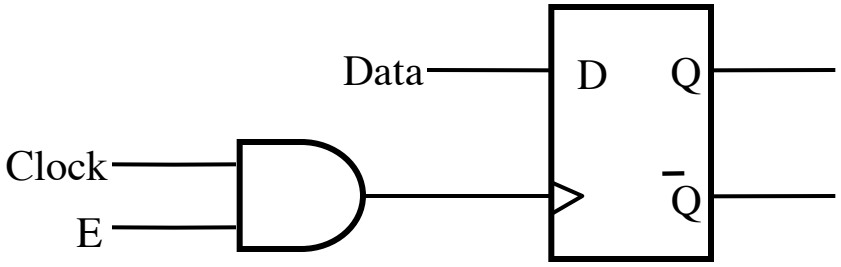
B&V3, Figure 10.47

When an asynchronous input fails to satisfy setup or hold times, the flip-flop can enter a metastable state (intermediate/indeterminate value) and not recover for an indefinite period of time.

Using a pair of flip-flops in series significantly reduces the likelihood that any synchronous system reading Data will observe such a metastable value

- COTS (Commercial, Off-The-Shelf) devices typically specify a maximum period of metastability
- as long as the clock period in the circuit above exceeds this value, the FF on the right will not also enter a metastable state even when the FF on the left does
- cost is one clock period of delay or latency in the arrival of the data input

Practical Issue: Clock enable circuit

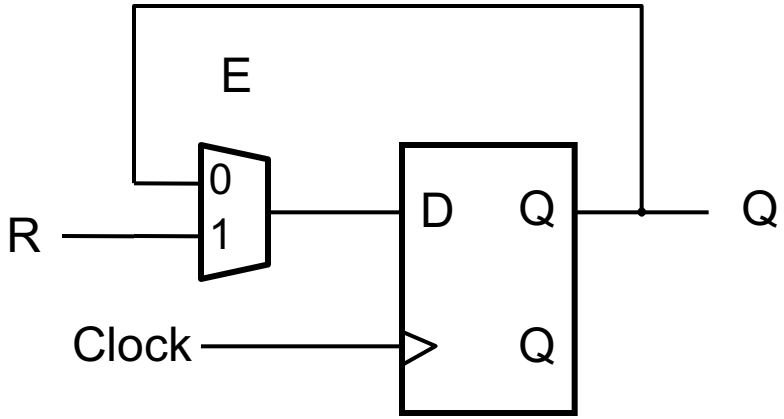


B&V3, Figure 10.43

While seemingly attractive, clock gating to enable a flip-flop is to be avoided as it contributes to clock skew.

Better solution: A flip-flop with an enable input

It is often desirable to control when a flip-flop or register is loaded with a new value




```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY rege IS
      PORT ( R, Resetn, E, Clock : IN          STD_LOGIC ;
                Q                : BUFFER    STD_LOGIC ) ;
END rege ;

ARCHITECTURE Behavior OF rege IS
BEGIN
      PROCESS ( Resetn, Clock )
      BEGIN
            IF Resetn = '0' THEN
                  Q <= '0' ;
            ELSIF Clock'EVENT AND Clock = '1' THEN
                  IF E = '1' THEN
                        Q <= R ;
                  ELSE
                        Q <= Q ;
                  END IF ;
            END IF ;
      END PROCESS ;
END Behavior ;
```

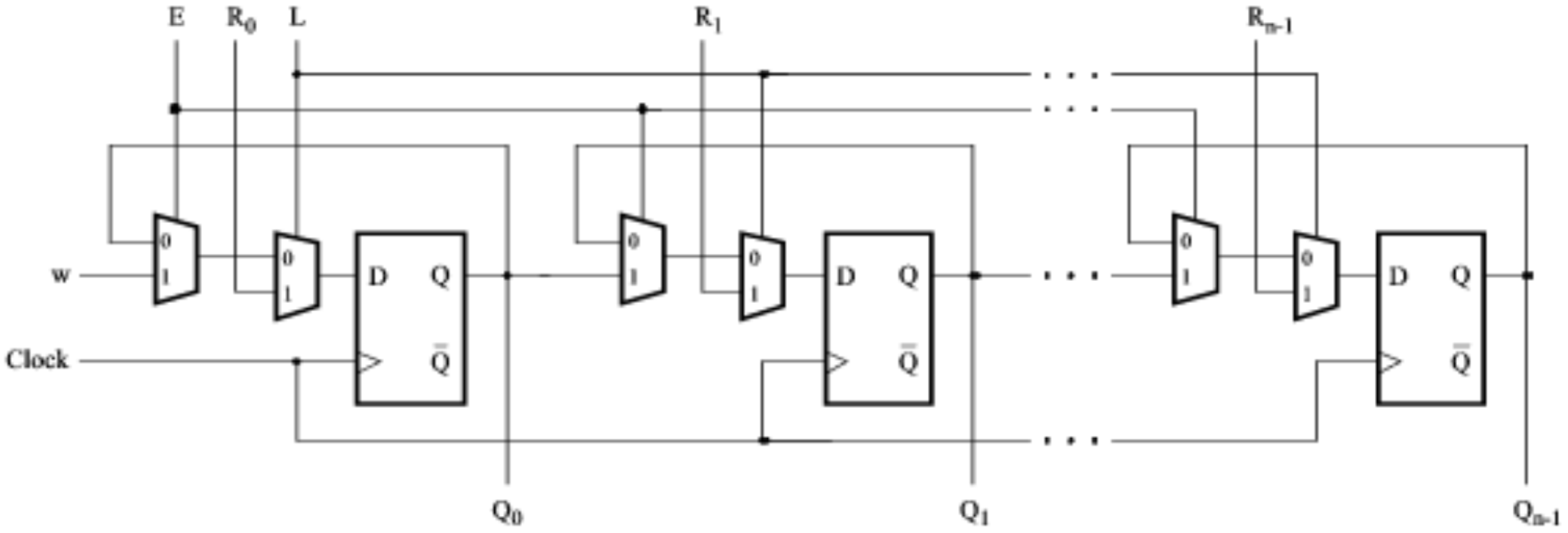
B&V3, Figure 10.1

VHDL code for a *n*-bit register with an enable input

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regne IS
        GENERIC ( N : INTEGER := 4 ) ;
        PORT (  R           : IN    STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
                Resetn      : IN    STD_LOGIC ;
                E, Clock    : IN    STD_LOGIC ;
                Q           : OUT   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END regne ;

ARCHITECTURE Behavior OF regne IS
BEGIN
        PROCESS ( Resetn, Clock )
        BEGIN
                IF Resetn = '0' THEN
                    Q <= (OTHERS => '0') ;
                ELSIF Clock'EVENT AND Clock = '1' THEN
                    IF E = '1' THEN
                        Q <= R ;
                    END IF ;
                END IF ;
        END PROCESS ;
END Behavior ;
```

B&V3, Figure 10.2

A shift register with parallel-load and enable control inputs



B&V3, Figure 10.3

Code for a right-to-left shift register with an enable input

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

-- right-to-left shift register with parallel load and enable
ENTITY shiftlne IS
      GENERIC ( N : INTEGER := 4 ) ;
      PORT(  R          : IN      STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
             L, E, w    : IN      STD_LOGIC ;
             Clock      : IN      STD_LOGIC ;
             Q          : BUFFER  STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END shiftlne ;

ARCHITECTURE Behavior OF shiftlne IS
BEGIN
      PROCESS
      BEGIN
            WAIT UNTIL Clock'EVENT AND Clock = '1' ;
            IF L = '1' THEN
                  Q <= R ;
            ELSIF E = '1' THEN
                  Q(0) <= w ;
                  Genbits: FOR i IN 1 TO N-1 LOOP
                        Q(i) <= Q(i-1) ;
                  END LOOP ;
            END IF ;
      END PROCESS ;
END Behavior ;
```

B&V3, Figure 10.4

Component declaration statements assumed for remaining design problems (Part *a*)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE components IS

    -- 2-to-1 multiplexer
    COMPONENT mux2to1
        PORT ( w0, w1     : IN     STD_LOGIC ;
               s          : IN     STD_LOGIC ;
               f          : OUT    STD_LOGIC ) ;
    END COMPONENT ;

    -- D flip-flop with 2-to-1 multiplexer connected to D
    COMPONENT muxdff
        PORT ( D0, D1, Sel, Clock  : IN     STD_LOGIC ;
               Q                   : OUT    STD_LOGIC ) ;
    END COMPONENT ;
```

```
    -- n-bit register with enable
    COMPONENT regne
        GENERIC ( N : INTEGER := 4 ) ;
        PORT ( R            : IN
                   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
               Resetn       : IN     STD_LOGIC ;
               E, Clock     : IN     STD_LOGIC ;
               Q            : OUT
                   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
    END COMPONENT ;

    -- n-bit right-to-left shift register with parallel load and enable
    COMPONENT shiftlne  -- shift left (towards msb)–mult by 2
        GENERIC ( N : INTEGER := 4 ) ;
        PORT ( R          : IN
                   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
               L, E, w    : IN          STD_LOGIC ;
               Clock      : IN          STD_LOGIC ;
               Q          : BUFFER
                   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
    END COMPONENT ;
```

.. continued in Part *b*

B&V3, Figure 10.5

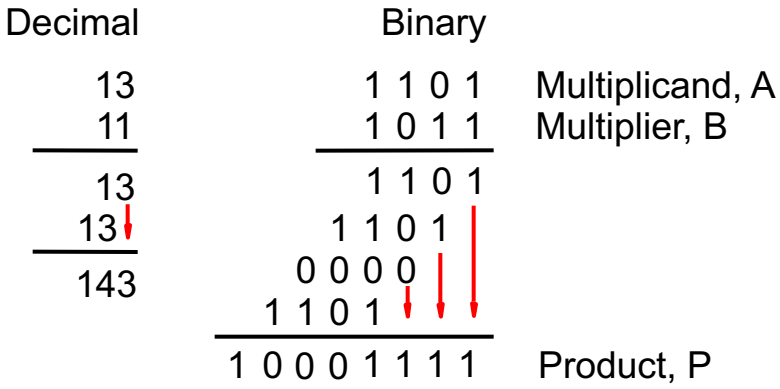
Component declaration statements for digital systems building blocks (Part *b*)

```
-- n-bit left-to-right shift register with parallel load and enable
COMPONENT shiftrne -- shift right (towards lsb)–div by 2
    GENERIC ( N : INTEGER := 4 ) ;
    PORT ( R          : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           L, E, w    : IN     STD_LOGIC ;
           Clock      : IN     STD_LOGIC ;
           Q          : BUFFER STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END COMPONENT ;

-- up-counter that counts up from initial value R to modulus-1
COMPONENT upcount
    GENERIC    ( modulus : INTEGER := 8 ) ;
    PORT ( Resetn       : IN     STD_LOGIC ;
           Clock, E, L  : IN     STD_LOGIC ;
           R            : IN     INTEGER RANGE 0 TO modulus-1 ;
           Q            : BUFFER INTEGER RANGE 0 TO modulus-1 ) ;
END COMPONENT ;

-- down-counter that counts from modulus-1 down to 0
COMPONENT downcnt
    GENERIC    ( modulus : INTEGER := 8 ) ;
    PORT ( Clock, E, L  : IN     STD_LOGIC ;
           Q            : BUFFER INTEGER RANGE 0 TO modulus-1 ) ;
END COMPONENT ;

END components ;
```

B&V3, Figure 10.5

Design Exercise 2: pp 683 – 692, B&V3

- Implement a binary multiplier circuit



(a) Manual method

```
P = 0
for i = 0 to n-1 do
    if b_i = 1 then
        P = P + A
    end if
    left shift A
end for
```

(b) Pseudo-code

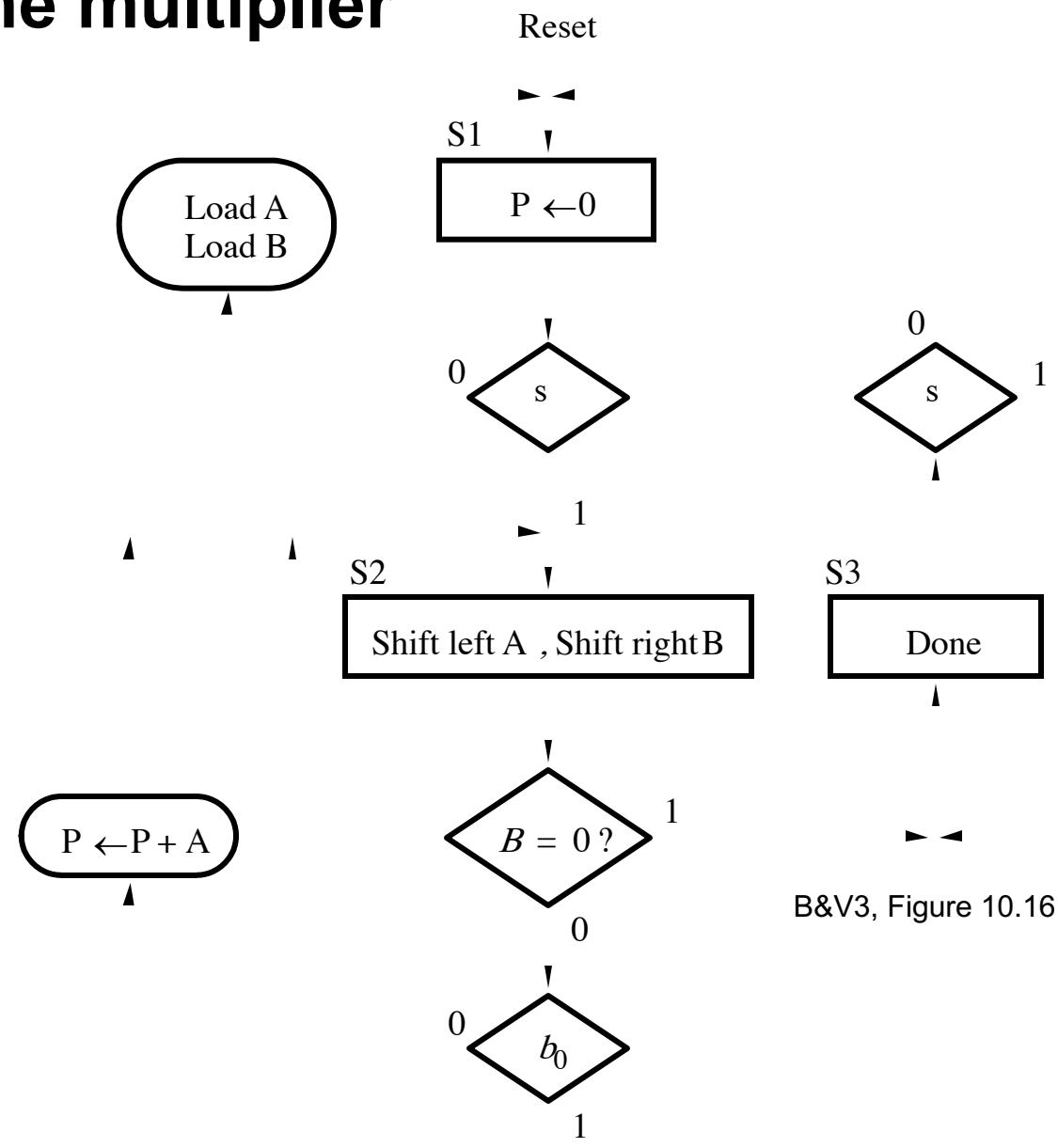
B&V3, Figure 10.15

1. How is the computation performed?
2. What datapath components are required?
3. How are they to be controlled?

ASM chart for the multiplier

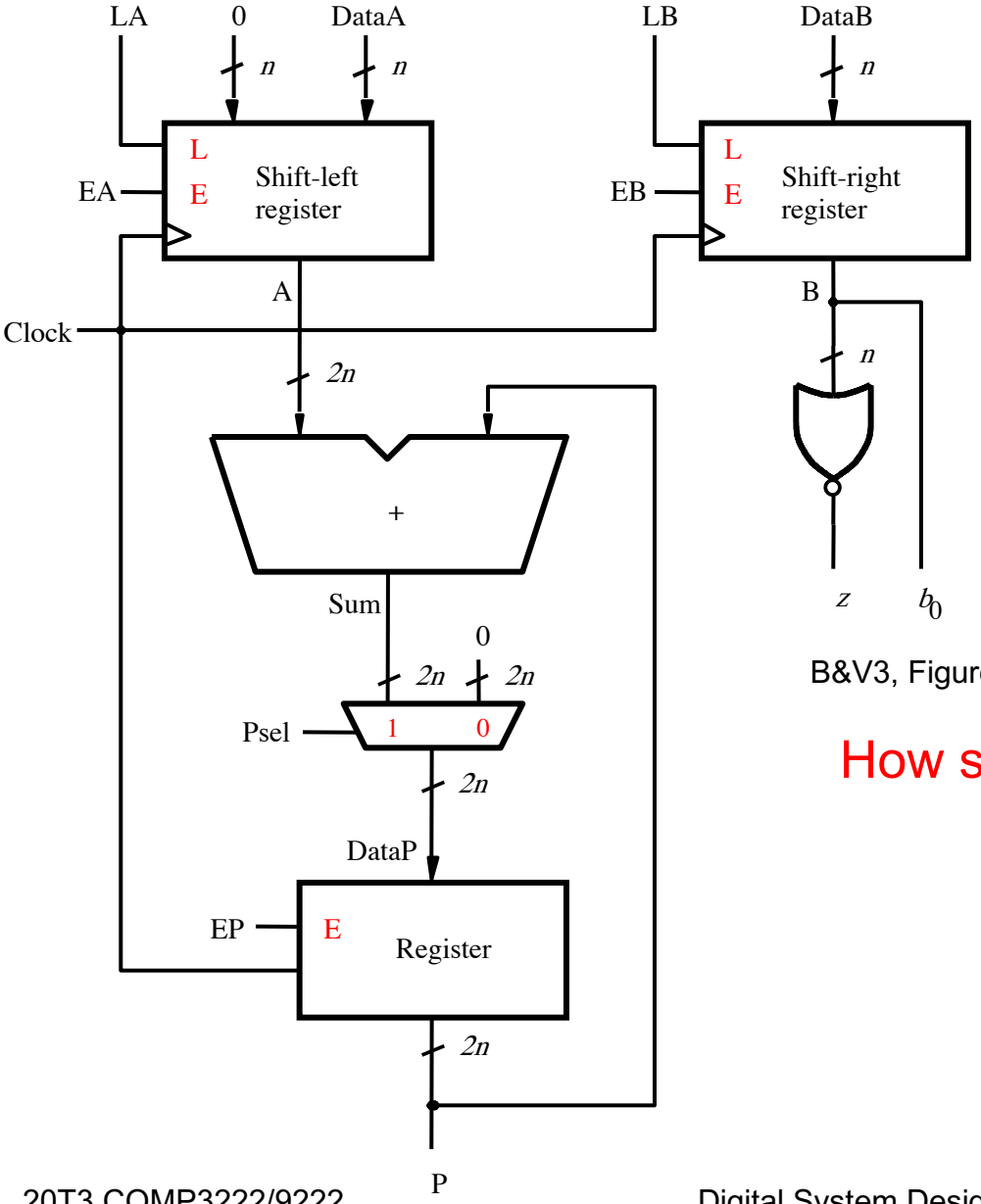
A: multiplicand
B: multiplier
P: product

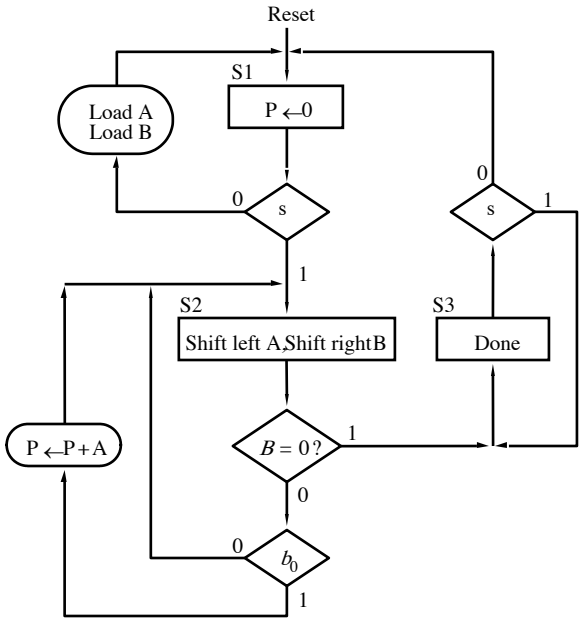
$P = 0$
for $i = 0$ to n-1 do
  if $b_i = 1$ then
    $P = P + A$
  end if
  left shift A
end for



B&V3, Figure 10.16

Datapath circuit for the multiplier

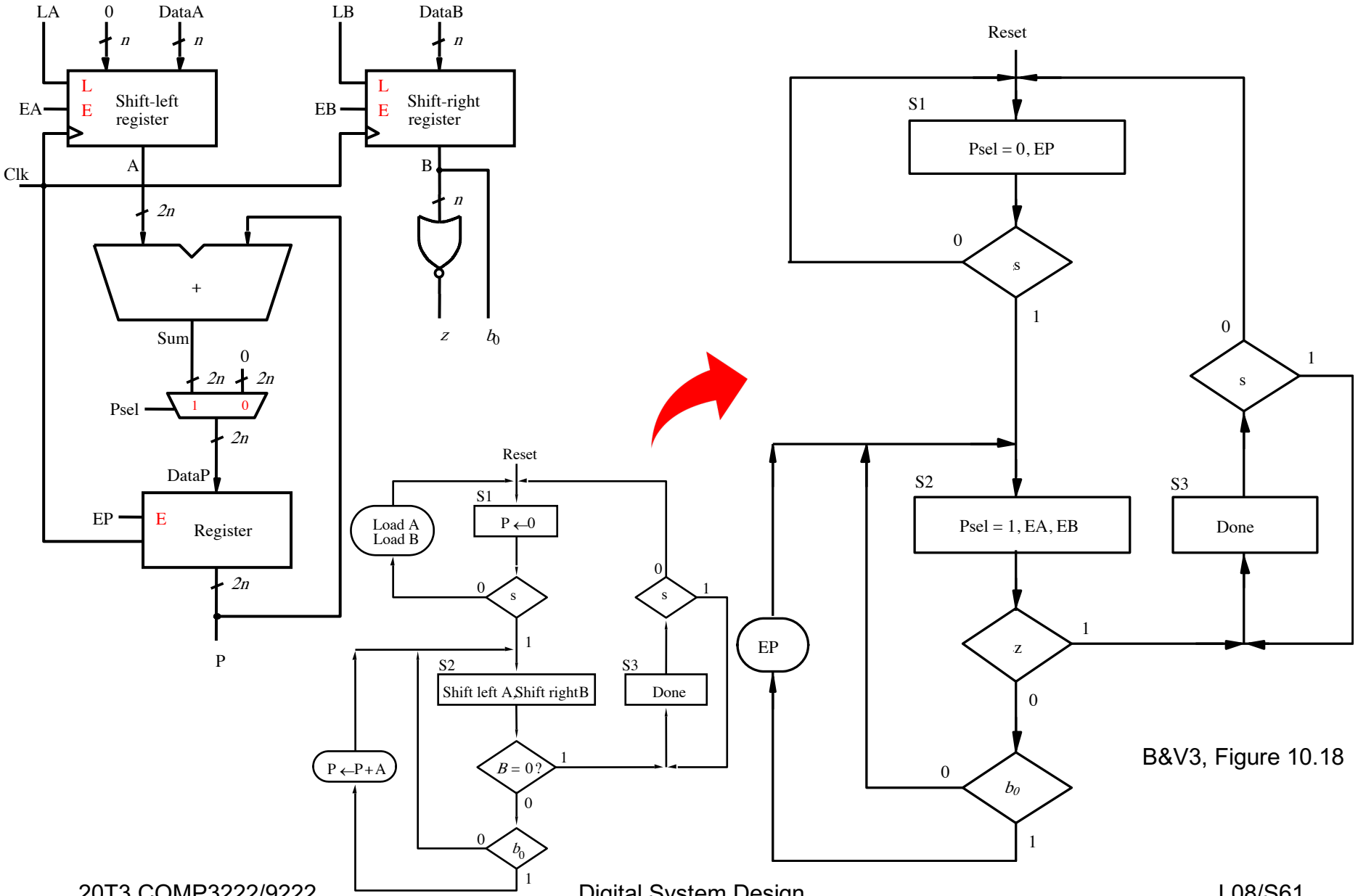




B&V3, Figure 10.17

How should the ASM chart be refined?

ASM chart for the multiplier control circuit



B&V3, Figure 10.18

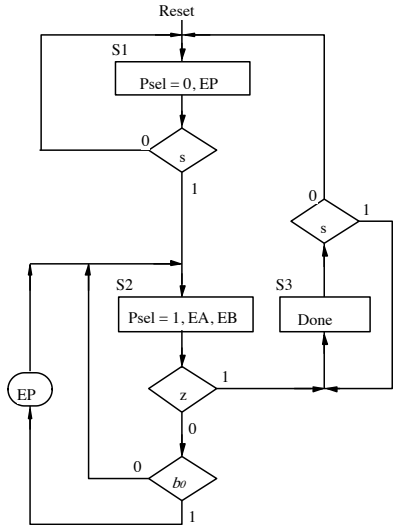
VHDL code for the multiplier circuit (Part *a*)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;
USE work.components.all ;

ENTITY multiply IS
    GENERIC ( N : INTEGER := 8; NN : INTEGER := 16 ) ;
    PORT ( Clock      : IN     STD_LOGIC ;
           Resetn     : IN     STD_LOGIC ;
           LA, LB, s  : IN     STD_LOGIC ;
           DataA      : IN     STD_LOGIC_VECTOR(N–1 DOWNTO 0) ;
           DataB      : IN     STD_LOGIC_VECTOR(N–1 DOWNTO 0) ;
           P          : BUFFER STD_LOGIC_VECTOR(NN–1 DOWNTO 0) ;
           Done       : OUT    STD_LOGIC ) ;
END multiply ;

ARCHITECTURE Behavior OF multiply IS
    TYPE State_type IS ( S1, S2, S3 ) ;
    SIGNAL y :  State_type          ;
    SIGNAL Psel, z, EA, EB, EP, Zero :  STD_LOGIC ;
    SIGNAL B, N_Zeros :  STD_LOGIC_VECTOR(N–1 DOWNTO 0) ;
    SIGNAL A, Ain, DataP, Sum :  STD_LOGIC_VECTOR(NN–1 DOWNTO 0) ;
BEGIN
```

B&V3, Figure 10.19

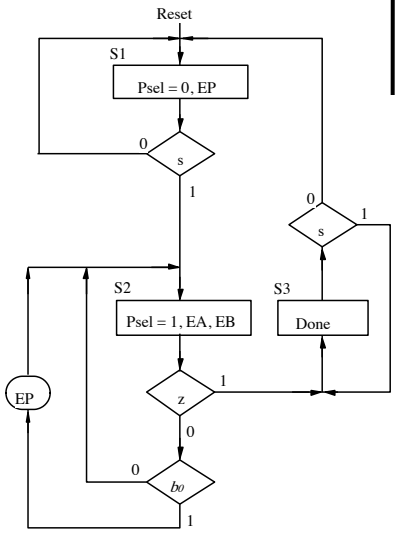
```
FSM_transitions:  PROCESS ( Resetn, Clock )
BEGIN
    IF Resetn = '0' THEN
        y <= S1 ;
    ELSIF (Clock'EVENT AND Clock = '1')
    THEN CASE y IS
        WHEN S1 =>
            IF s = '0' THEN y <= S1 ;
            ELSE y <= S2 ; END IF;
        WHEN S2 =>
            IF z = '0' THEN y <= S2 ;
            ELSE y <= S3 ; END IF;
        WHEN S3 =>
            IF s = '1' THEN y <= S3 ;
            ELSE y <= S1 ; END IF;
      END CASE ;
    END IF ;
END PROCESS ;
```

… continued in Part *b*



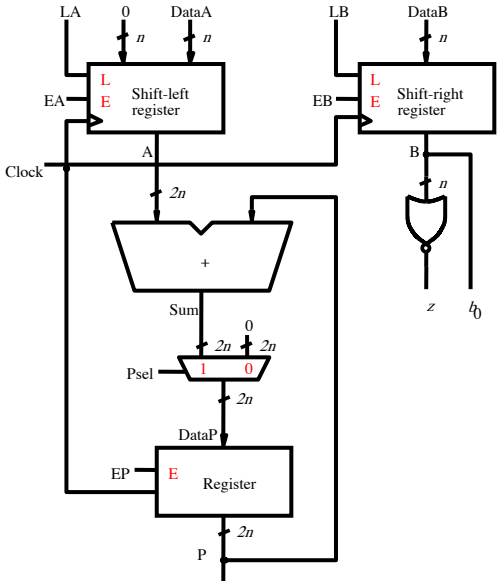
VHDL code for the multiplier circuit (Part *b*)

```
FSM_outputs:  PROCESS ( y, B(0) )
BEGIN
      EP <= '0' ; EA <= '0' ; EB <= '0' ;
      Done <= '0' ; Psel <= '0';
      CASE y IS
            WHEN S1 =>
                  EP <= '1' ;
            WHEN S2 =>
                  EA <= '1' ; EB <= '1' ; Psel <= '1' ;
                  IF B(0) = '1' THEN EP <= '1' ;
                  END IF ;
            WHEN S3 =>
                  Done <= '1' ;
      END CASE ;
END PROCESS ;
```

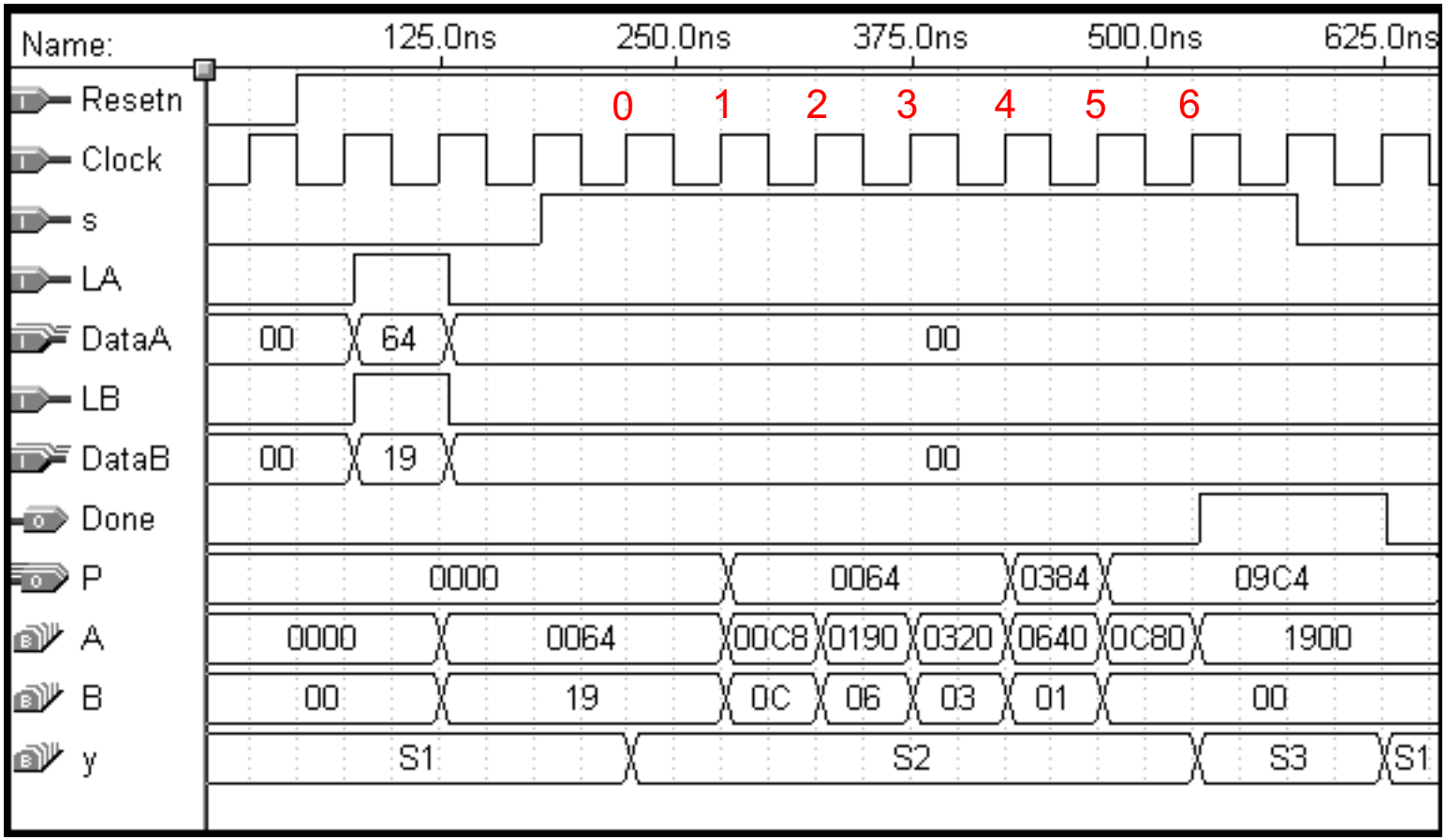



```
-- Define the datapath circuit
Zero <= '0' ;
N_Zeros <= (OTHERS =>  '0' ) ;
Ain <= N_Zeros & DataA ;
ShiftA:  shiftlne GENERIC MAP ( N =>  NN  )
      PORT MAP ( Ain, LA, EA, Zero, Clock, A ) ;
ShiftB: shiftrne GENERIC MAP ( N =>  N )
      PORT MAP ( DataB, LB, EB, Zero, Clock, B ) ;
z <= '1' WHEN B = N_Zeros ELSE '0' ;
Sum <= A + P ;
-- Define  the 2n  2-to-1 multiplexers for DataP
GenMUX: FOR i IN 0 TO NN–1 GENERATE
      Muxi: mux2to1 PORT MAP ( Zero, Sum(i), Psel, DataP(i) ) ;
END GENERATE;
RegP: regne GENERIC MAP ( N =>  NN  )
      PORT MAP ( DataP, Resetn, EP, Clock, P ) ;
END Behavior ;
```

B&V3, Figure 10.19



Simulation results for the multiplier circuit



B&V3, Figure 10.20

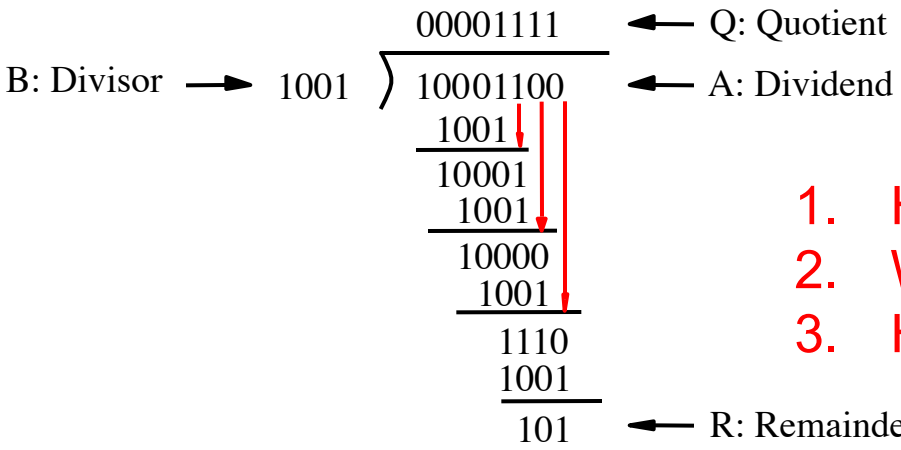
Design Exercise 3: Division
pp 692 – 702, B&V3

```
      15
   ------
9 ) 140
    9
   ---
    50
    45
   ---
     5
```

(a) An example using decimal numbers

```
R = 0 ;
for i = 0 to n – 1 do
    Left-shift R || A ;
    if R ≥ B then
        q_{n-1-i} = 1 ;
        R = R – B ;
    else
        q_{n-1-i} = 0 ;
    end if;
end for;
```

(c) Pseudo-code



(b) Using binary numbers

B&V3, Figure 10.21

1. How is the computation performed?
2. What datapath components are required?
3. How are they to be controlled?

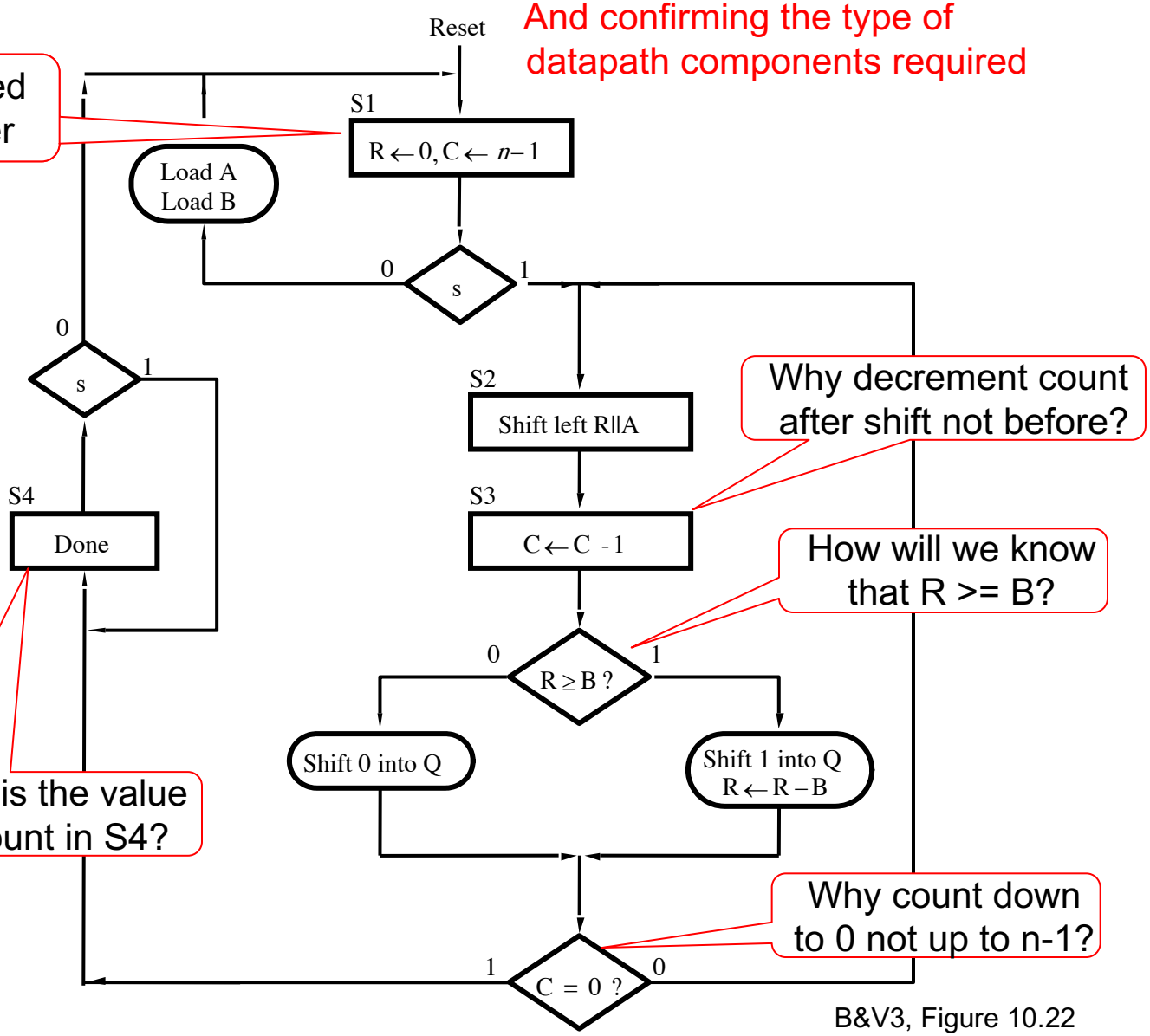
ASM chart for the divider

Answering the question:
How is the computation performed?

And confirming the type of datapath components required

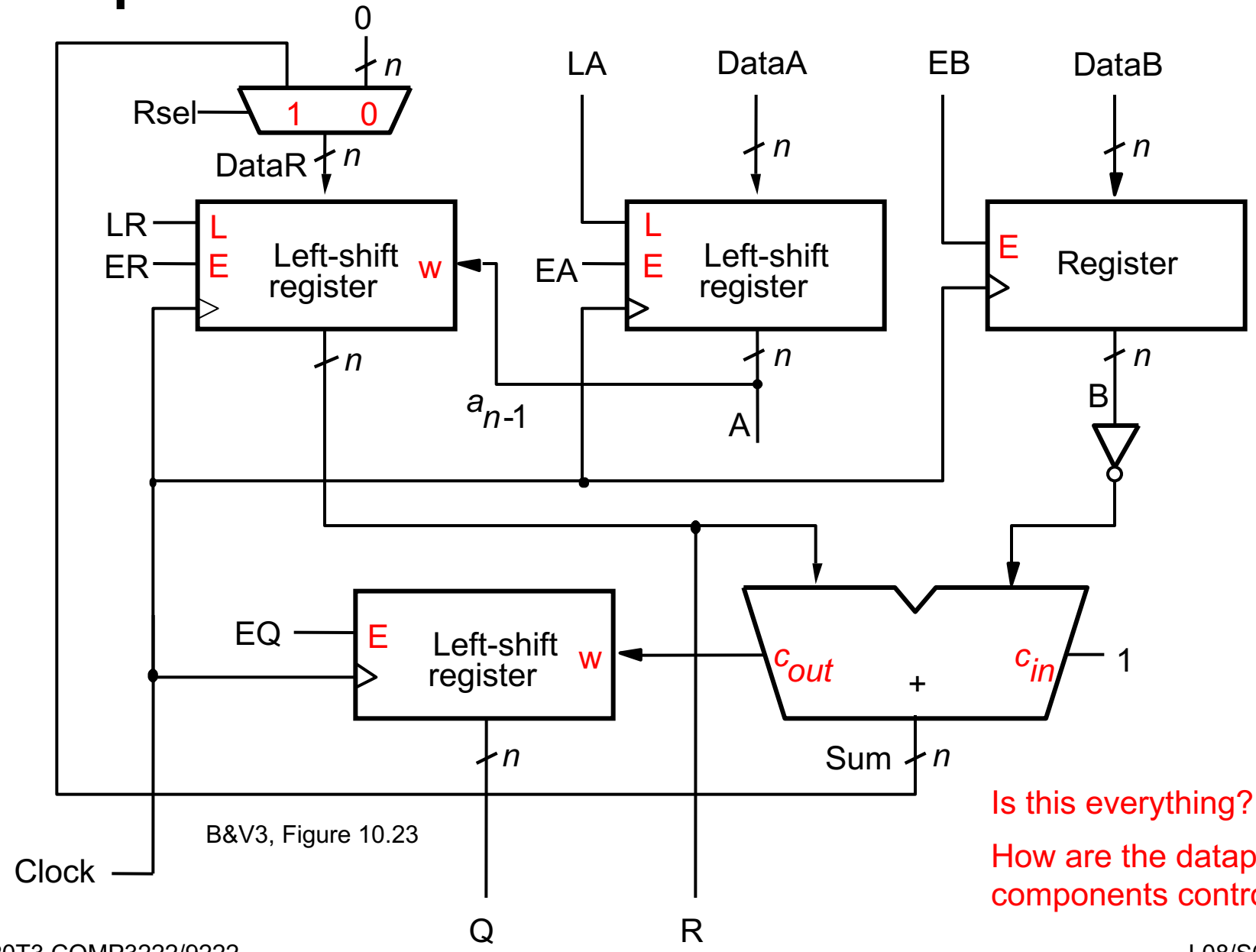
```
R = 0 ;
for i = 0 to n – 1 do
      Left-shift R || A ;
      if R ≥ B then
            q_{n-1-i} = 1 ;
            R = R – B ;
      else
            q_{n-1-i} = 0 ;
      end if;
end for;
```

A: Dividend
B: Divisor
Q: Quotient
R: Remainder



B&V3, Figure 10.22

Datapath circuit for the divider

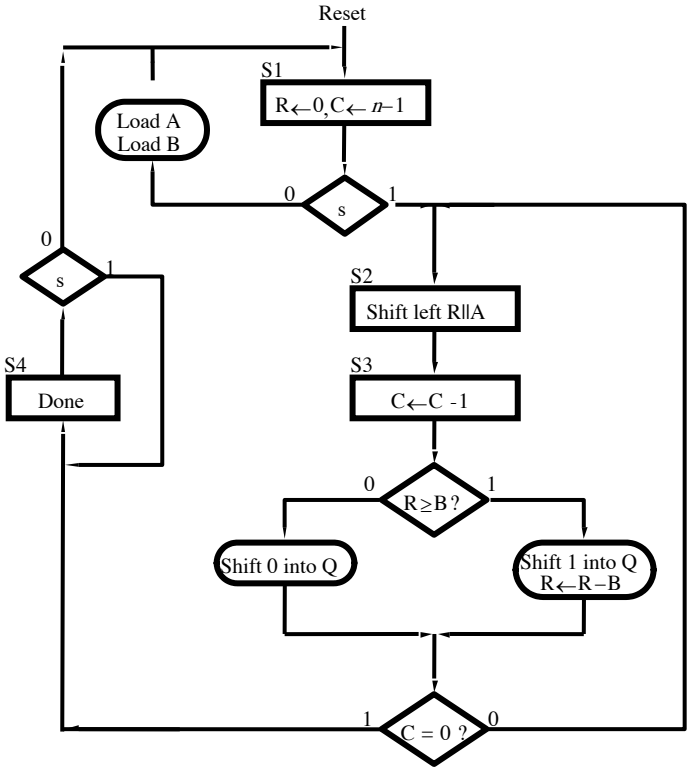


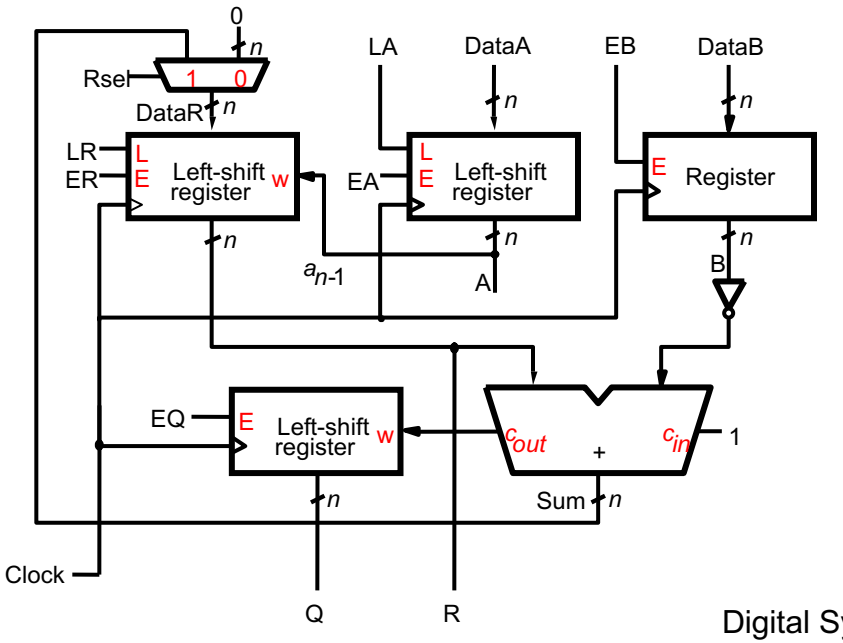
B&V3, Figure 10.23

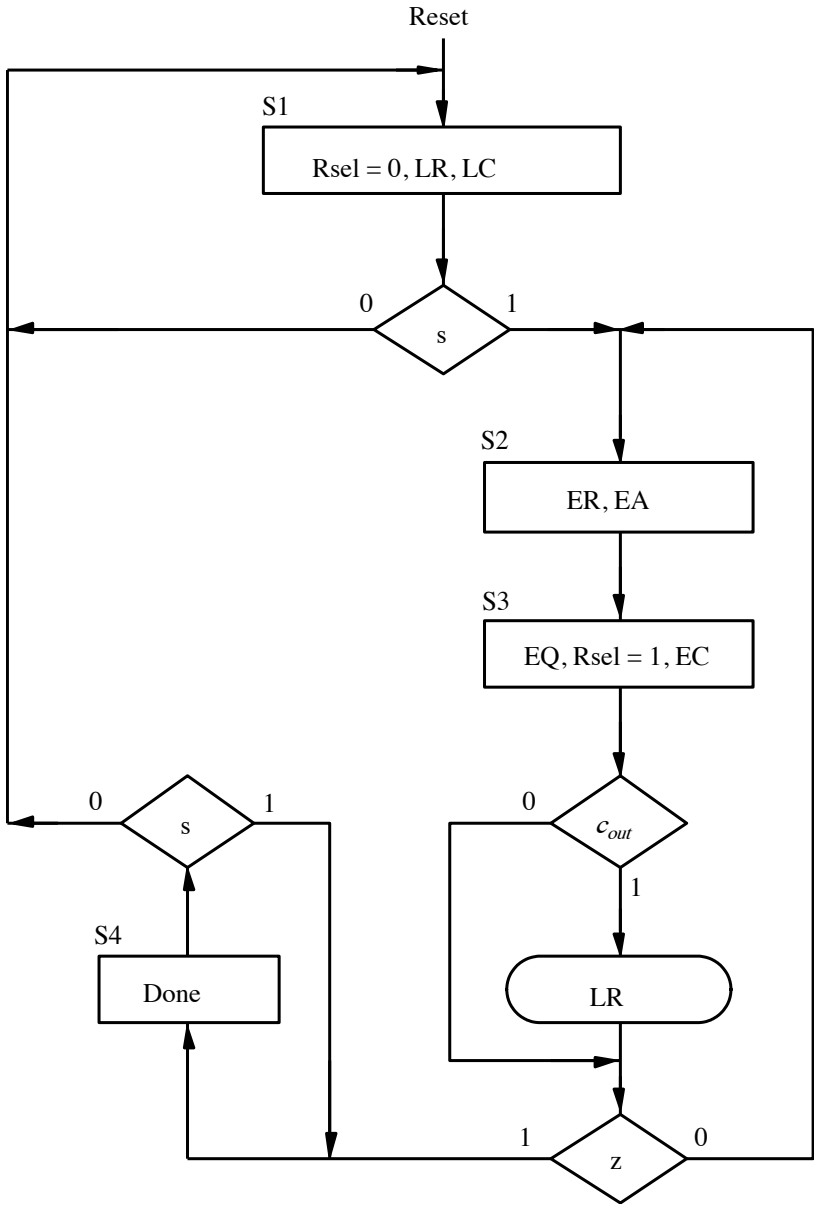
Is this everything?

How are the datapath components controlled?

ASM chart for the divider control circuit







An example of division using *n* = 8 clk cycles

One drawback of the divider we've designed is that it takes <u>two cycles per iteration</u> to (i) shift R||A, and (ii) update R←R - B when required.

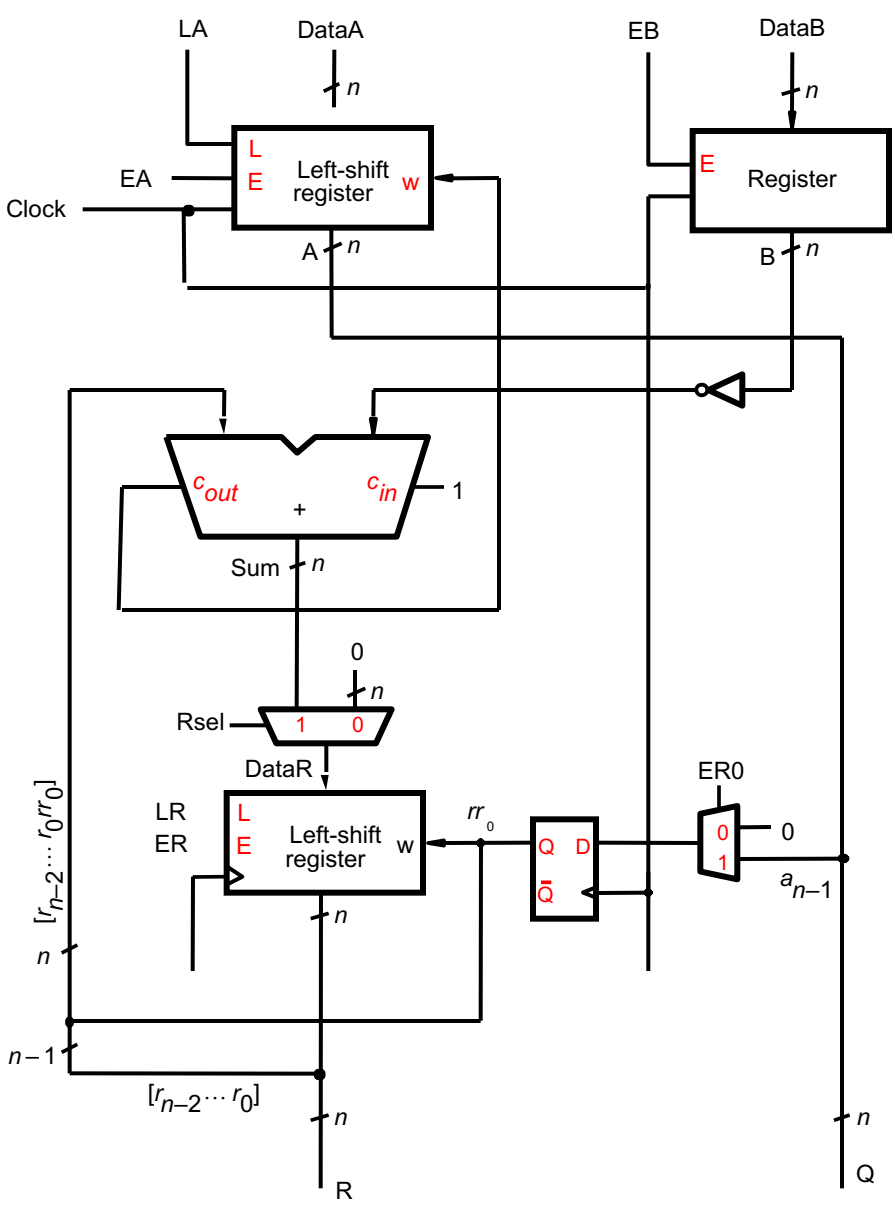
- A possible enhancement is to perform a shift and a subtraction in a single clock cycle. [write the results of the subtraction to $r_{n-1}..r_0$||write the new a_{n-1} to a flip-flop, rr_0]
- A second enhancement is to reuse the redundant bits of the shift register for A to store Q.

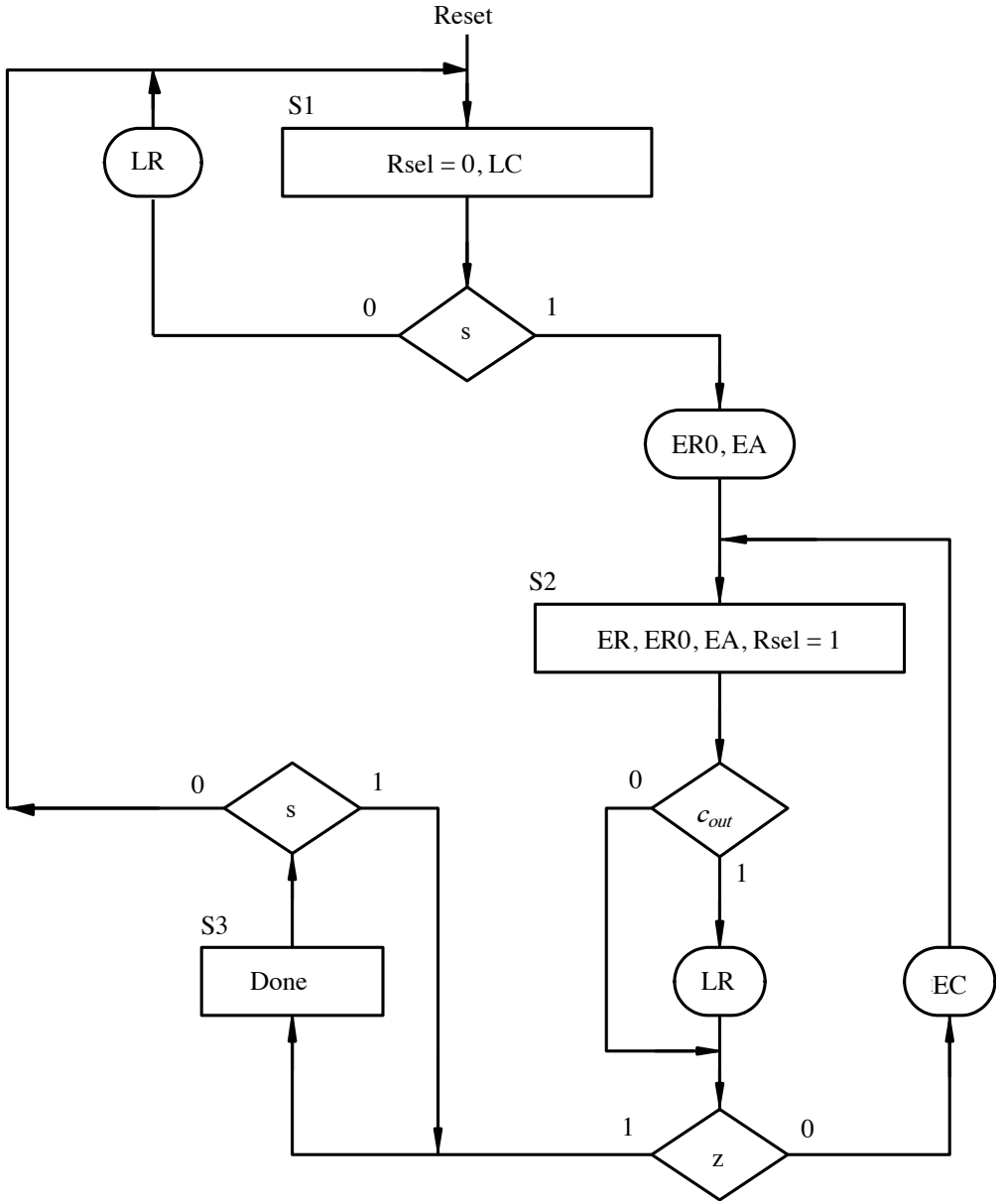
B → 1001 ⟩ 10001100 ← A

Clock cycle		R	rr_0	A/Q
	Load A, B	0 0 0 0 0 0 0 0	0	1 0 0 0 1 1 0 0
0	Shift left	0 0 0 0 0 0 0 0	1	0 0 0 1 1 0 0 0
1	Shift left, $Q_0 \leftarrow 0$	0 0 0 0 0 0 0 1	0	0 0 1 1 0 0 0 0
2	Shift left, $Q_0 \leftarrow 0$	0 0 0 0 0 0 1 0	0	0 1 1 0 0 0 0 0
3	Shift left, $Q_0 \leftarrow 0$	0 0 0 0 0 1 0 0	0	1 1 0 0 0 0 0 0
4	Shift left, $Q_0 \leftarrow 0$	0 0 0 0 1 0 0 0	1	1 0 0 0 0 0 0 0
5	Subtract, $Q_0 \leftarrow 1$	0 0 0 0 1 0 0 0	1	0 0 0 0 0 0 0 1
6	Subtract, $Q_0 \leftarrow 1$	0 0 0 0 1 0 0 0	0	0 0 0 0 0 0 1 1
7	Subtract, $Q_0 \leftarrow 1$	0 0 0 0 0 1 1 1	0	0 0 0 0 0 1 1 1
8	Subtract, $Q_0 \leftarrow 1$	0 0 0 0 0 1 0 1	0	0 0 0 0 1 1 1 1

B&V3, Figure 10.25

ASM chart and datapath circuit for the enhanced divider





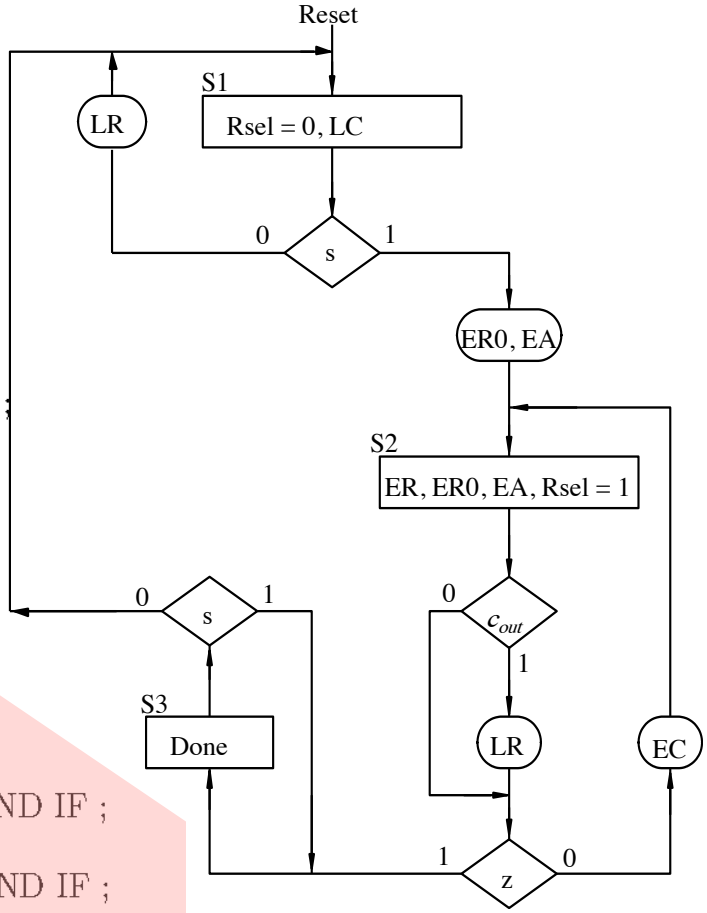
VHDL code for the enhanced divider circuit (Part *a*)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all ;
USE work.components.all ;

ENTITY divider IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT(  Clock       : IN      STD_LOGIC ;
           Resetn      : IN      STD_LOGIC ;
           s, LA, EB   : IN      STD_LOGIC ;
           DataA       : IN      STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           DataB       : IN      STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           R, Q        : BUFFER  STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           Done        : OUT     STD_LOGIC ) ;
END divider ;
```

B&V3, Figure 10.28

VHDL code for the enhanced divider circuit (Part *b*)

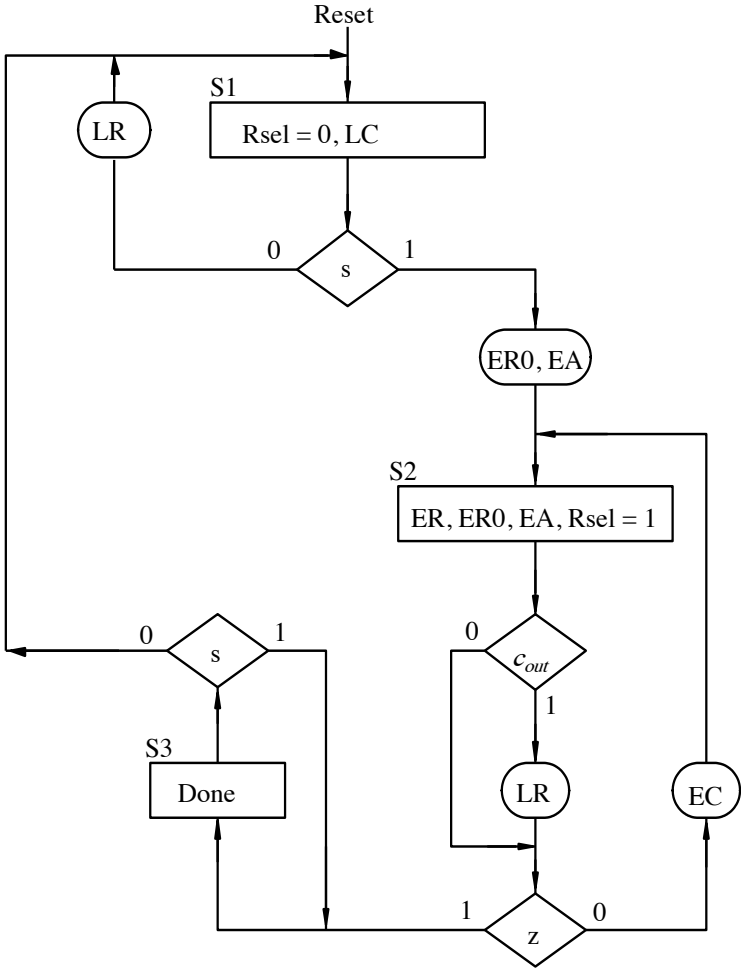
```
ARCHITECTURE Behavior OF divider IS
    TYPE State_type IS ( S1, S2, S3 ) ;
    SIGNAL y : State_type ;
    SIGNAL Zero, Cout, z : STD_LOGIC ;
    SIGNAL EA, Rsel, LR, ER, ER0, LC, EC, R0 : STD_LOGIC ;
    SIGNAL A, B, DataR : STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
    SIGNAL Sum : STD_LOGIC_VECTOR(N DOWNTO 0) ;
    SIGNAL Count : INTEGER RANGE 0 TO N-1 ;
BEGIN
    FSM_transitions: PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN y <= S1 ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN S1 =>
                    IF s = '0' THEN y <= S1 ; ELSE y <= S2 ; END IF ;
                WHEN S2 =>
                    IF z = '0' THEN y <= S2 ; ELSE y <= S3 ; END IF ;
                WHEN S3 =>
                    IF s = '1' THEN y <= S3 ; ELSE y <= S1 ; END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;
```



B&V3, Figure 10.28

VHDL code for the enhanced divider circuit (Part *c*)

```
FSM_outputs: PROCESS ( s, y, Cout, z )
BEGIN
    LR <= '0' ; ER <= '0' ; ER0 <= '0' ;
    LC <= '0' ; EC <= '0' ; EA <= '0' ; Done <= '0' ;
    Rsel <= '0' ;
    CASE y IS
        WHEN S1 =>
            LC <= '1' ;
            IF s = '0' THEN
                LR <= '1' ; EA <= '0'; ER0 <= '0' ;
            ELSE
                LR <= '0'; EA <= '1' ; ER0 <= '1' ;
            END IF ;
        WHEN S2 =>
            Rsel <= '1' ; ER <= '1' ; ER0 <= '1' ; EA <= '1' ;
            IF Cout = '1' THEN LR <= '1' ; ELSE LR <= '0' ; END IF ;
            IF z = '0' THEN EC <= '1' ; ELSE EC <= '0' ; END IF ;
        WHEN S3 =>
            Done <= '1' ;
    END CASE ;
END PROCESS ;
```

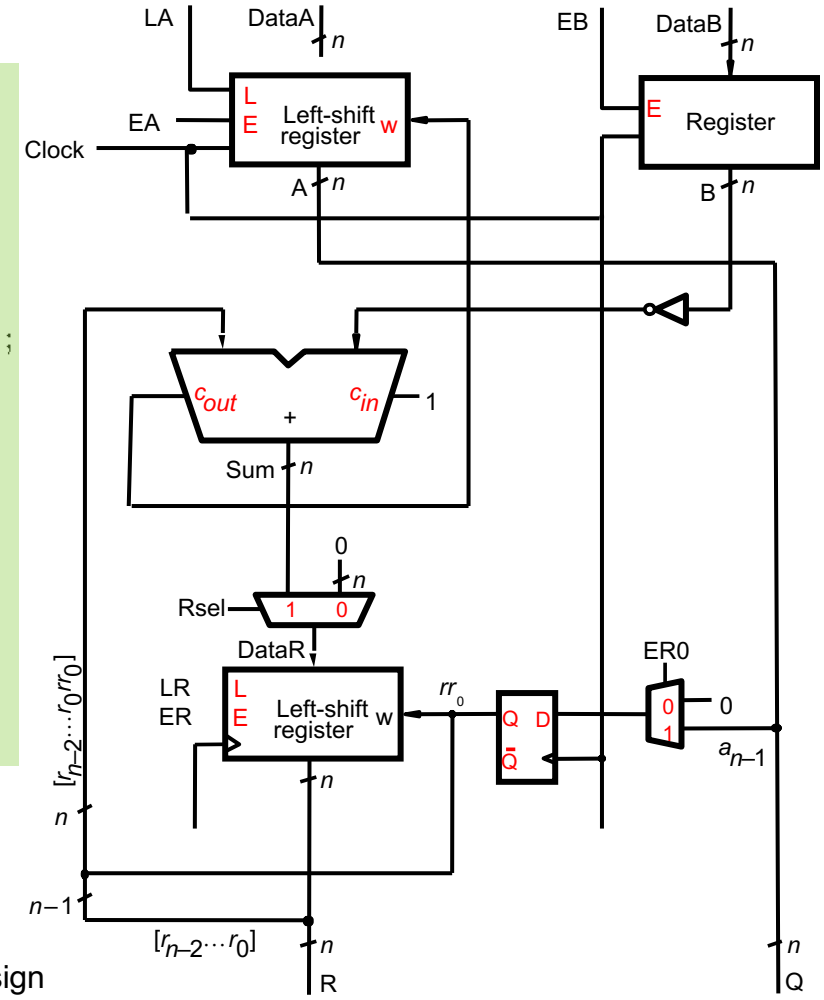


B&V3, Figure 10.28

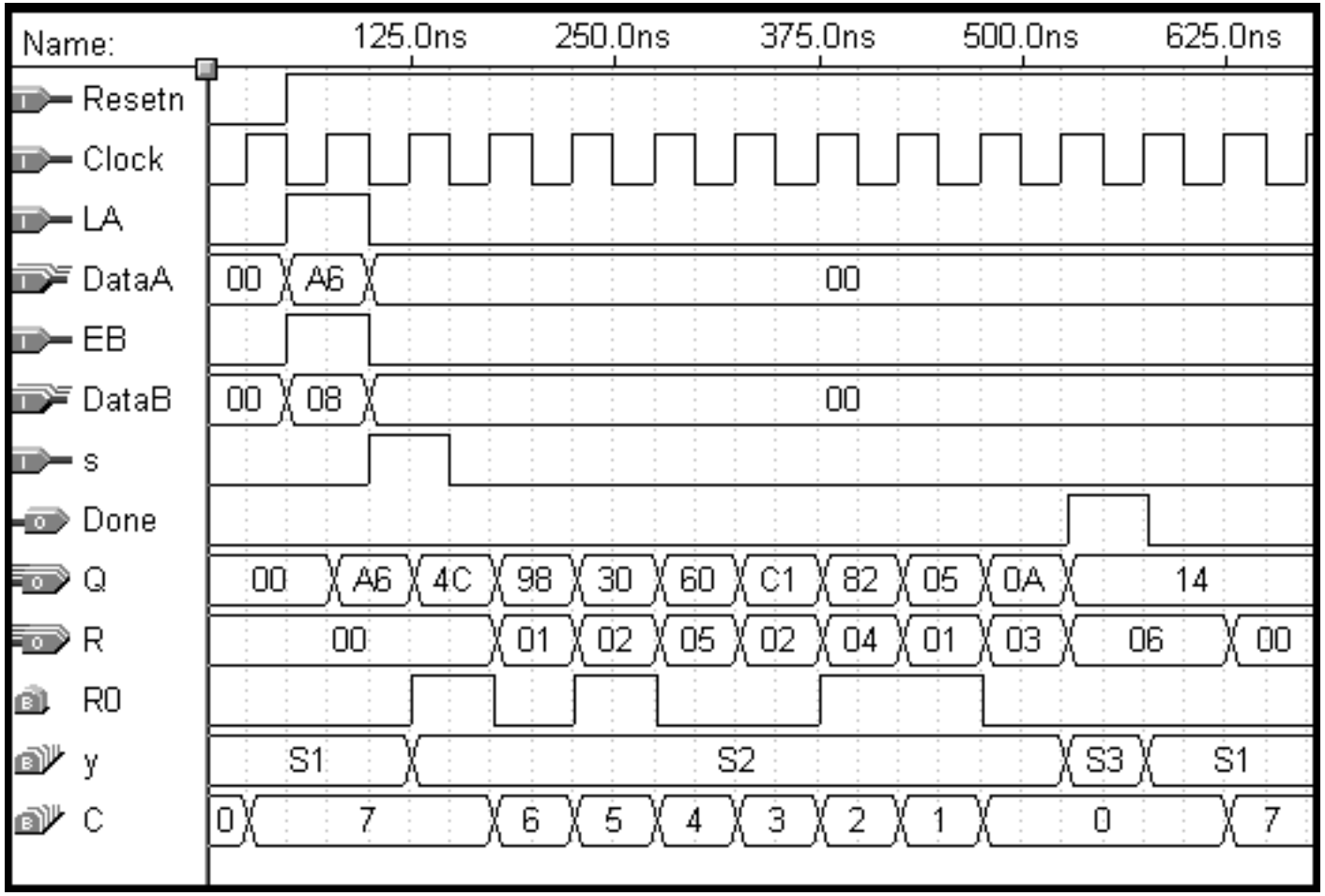
VHDL code for the enhanced divider circuit (Part *d*)

```
-- define the datapath circuit
Zero <= '0' ;
RegB: regne GENERIC MAP ( N => N )
    PORT MAP ( DataB, Resetn, EB, Clock, B ) ;
ShiftR: shiftlne GENERIC MAP ( N => N )
    PORT MAP ( DataR, LR, ER, R0, Clock, R ) ;
FF_R0: muxdff PORT MAP ( Zero, A(N-1), ER0, Clock, R0 ) ;
ShiftA: shiftlne GENERIC MAP ( N => N )
    PORT MAP ( DataA, LA, EA, Cout, Clock, A ) ;
Q <= A ;
Counter: downcnt GENERIC MAP ( modulus => N )
    PORT MAP ( Clock, EC, LC, Count ) ;
z <= '1' WHEN Count = 0 ELSE '0' ;

Sum <= R & R0 + (NOT B +1) ;
Cout <= Sum(N) ;
DataR <= (OTHERS => '0') WHEN Rsel = '0' ELSE Sum ;
END Behavior ;
```

B&V3, Figure 10.28



Simulation results for the enhanced divider circuit

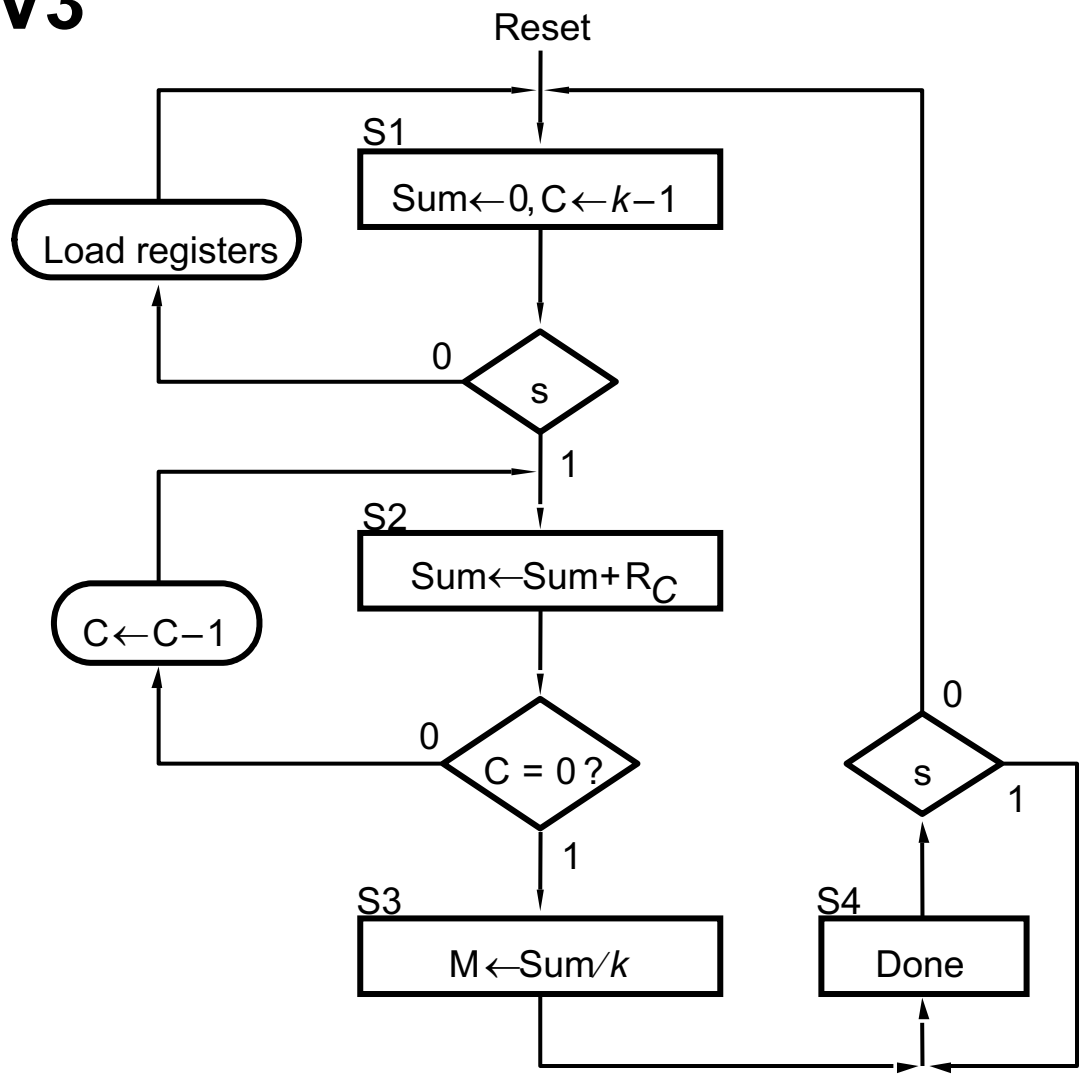


B&V3, Figure 10.29

Design Ex 4: Finding the mean of *k* numbers pp 702 – 708, B&V3

Sum = 0 ;
for *i* = *k* − 1 downto 0 do
    Sum = *Sum* + R_i ;
end for;
M = *Sum* ÷ *k* ;

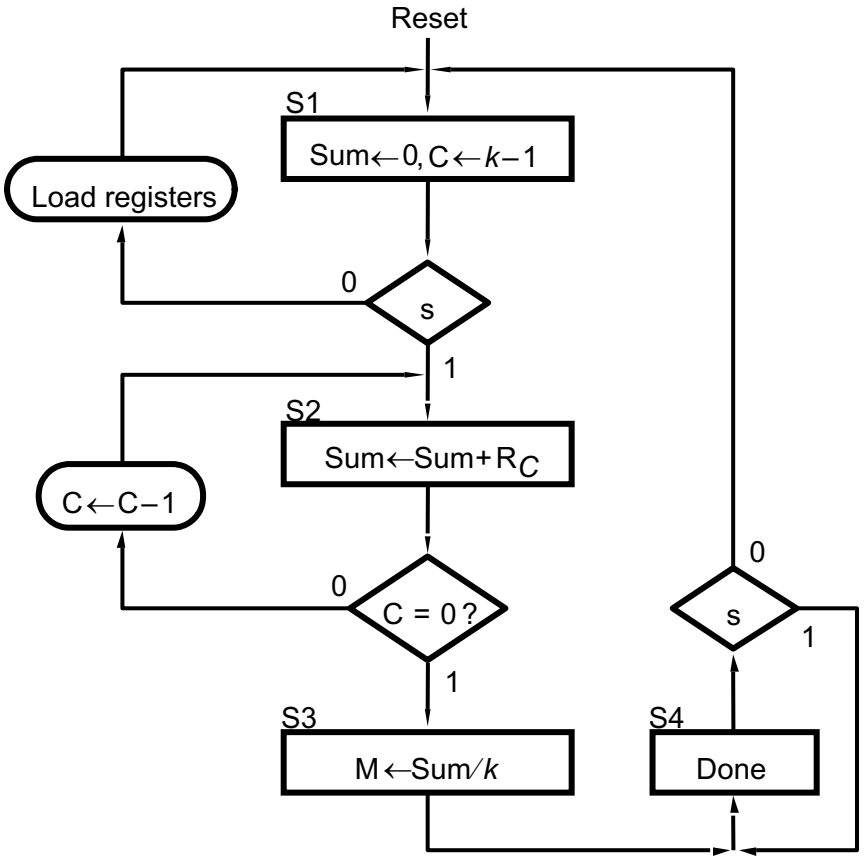
(a) Pseudo-code



B&V3, Figure 10.30

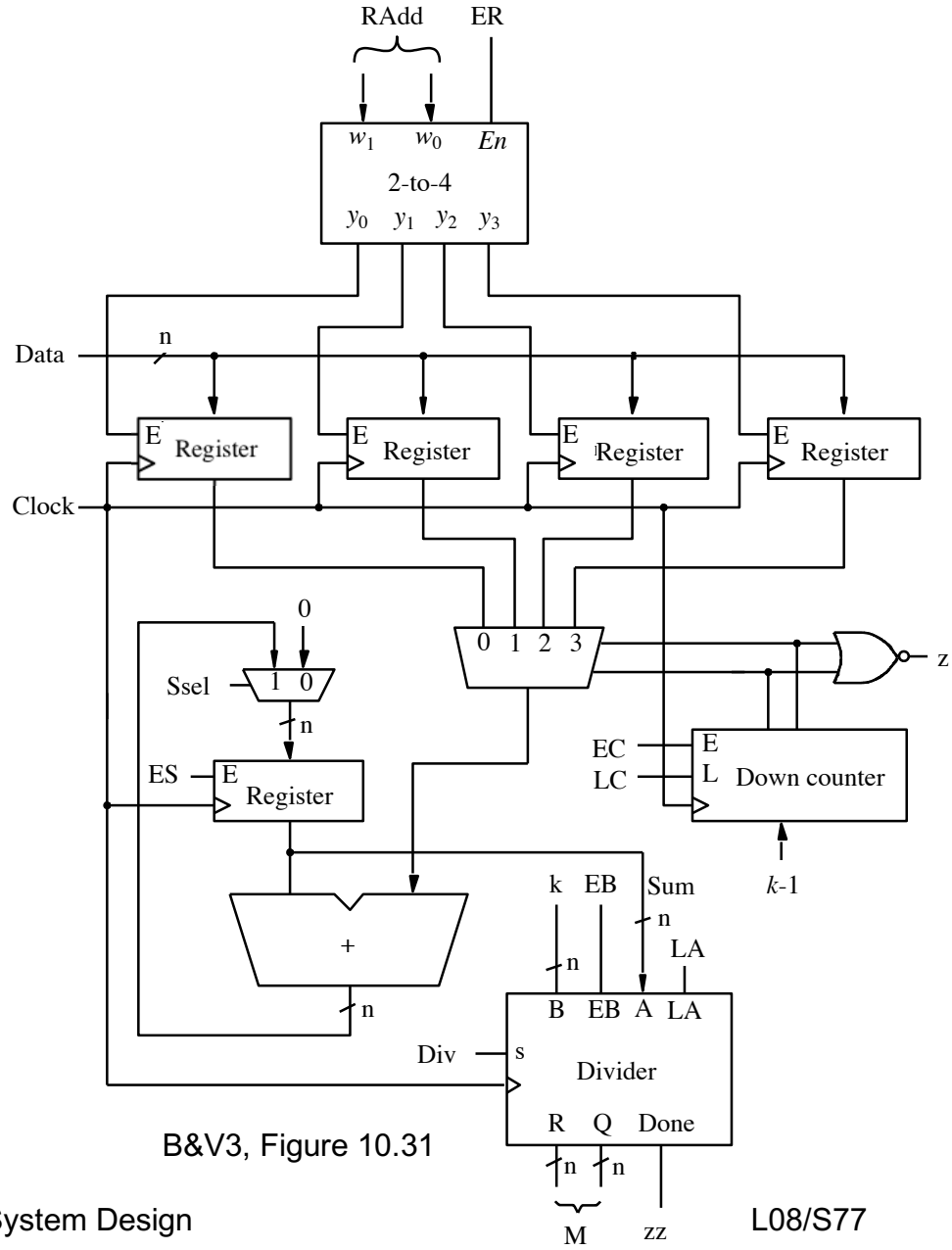
(b) ASM chart

Finding the mean of *k* numbers – datapath



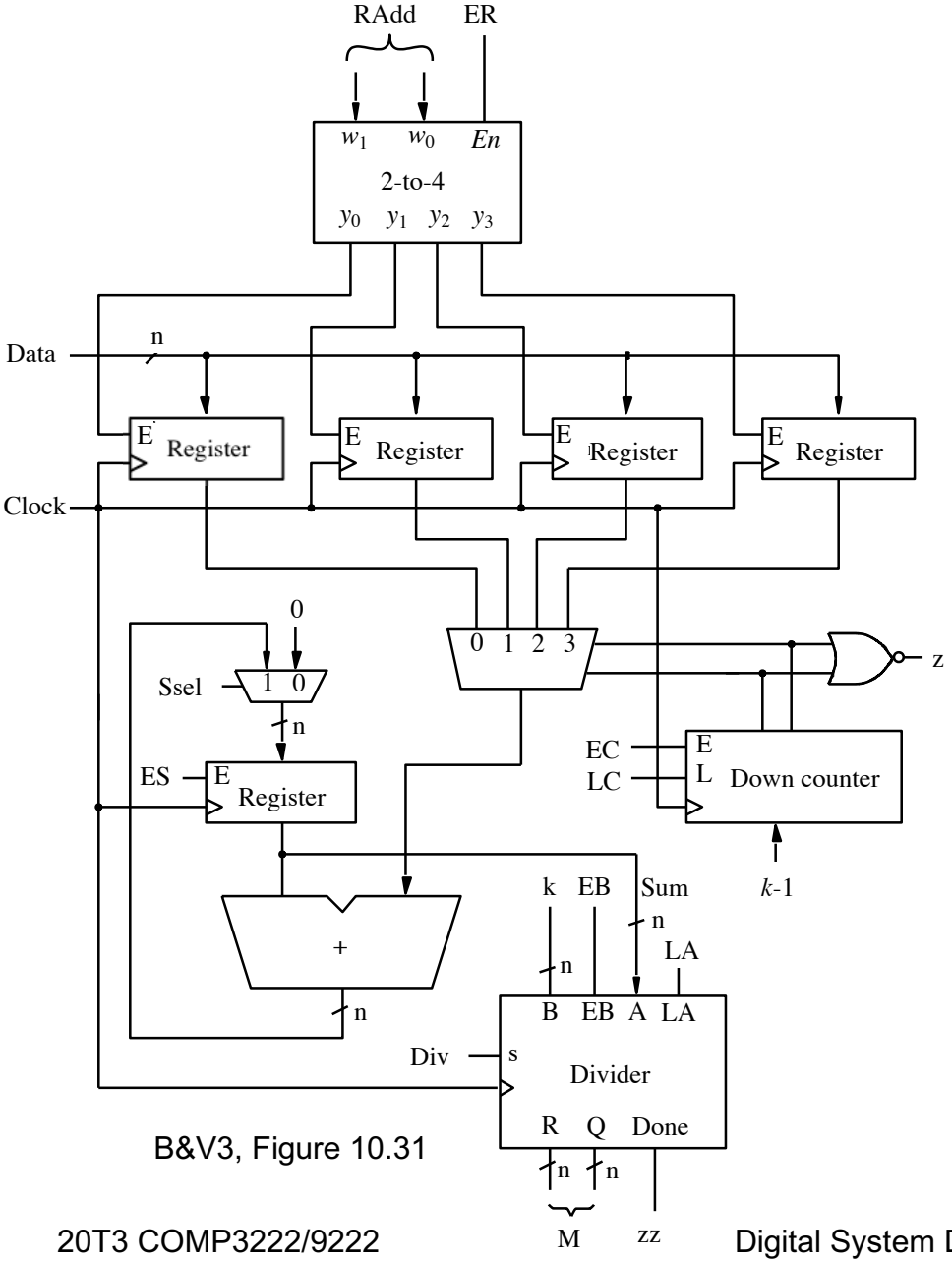
(b) ASM chart

B&V3, Figure 10.30

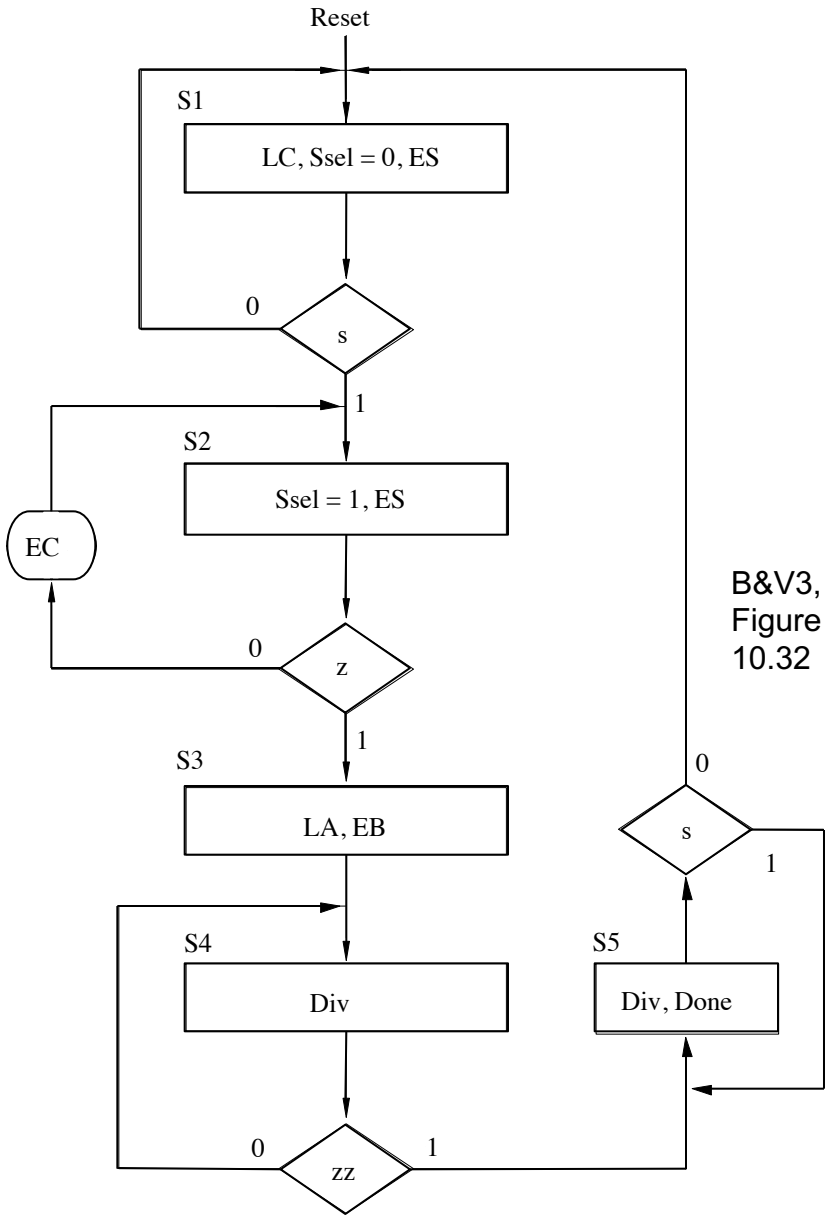


B&V3, Figure 10.31

Datapath & controlpath ASM for the mean operation

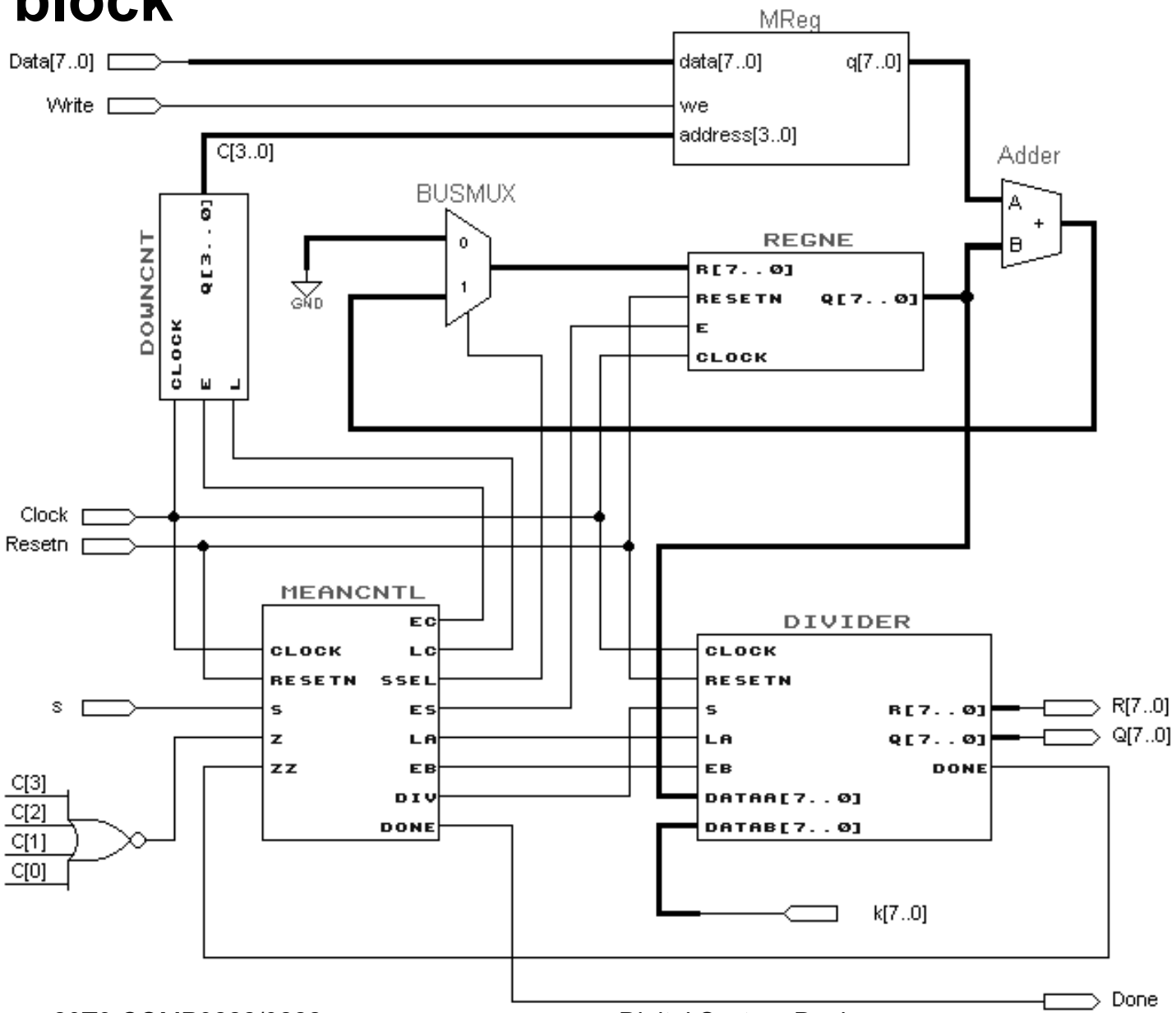


B&V3, Figure 10.31



B&V3, Figure 10.32

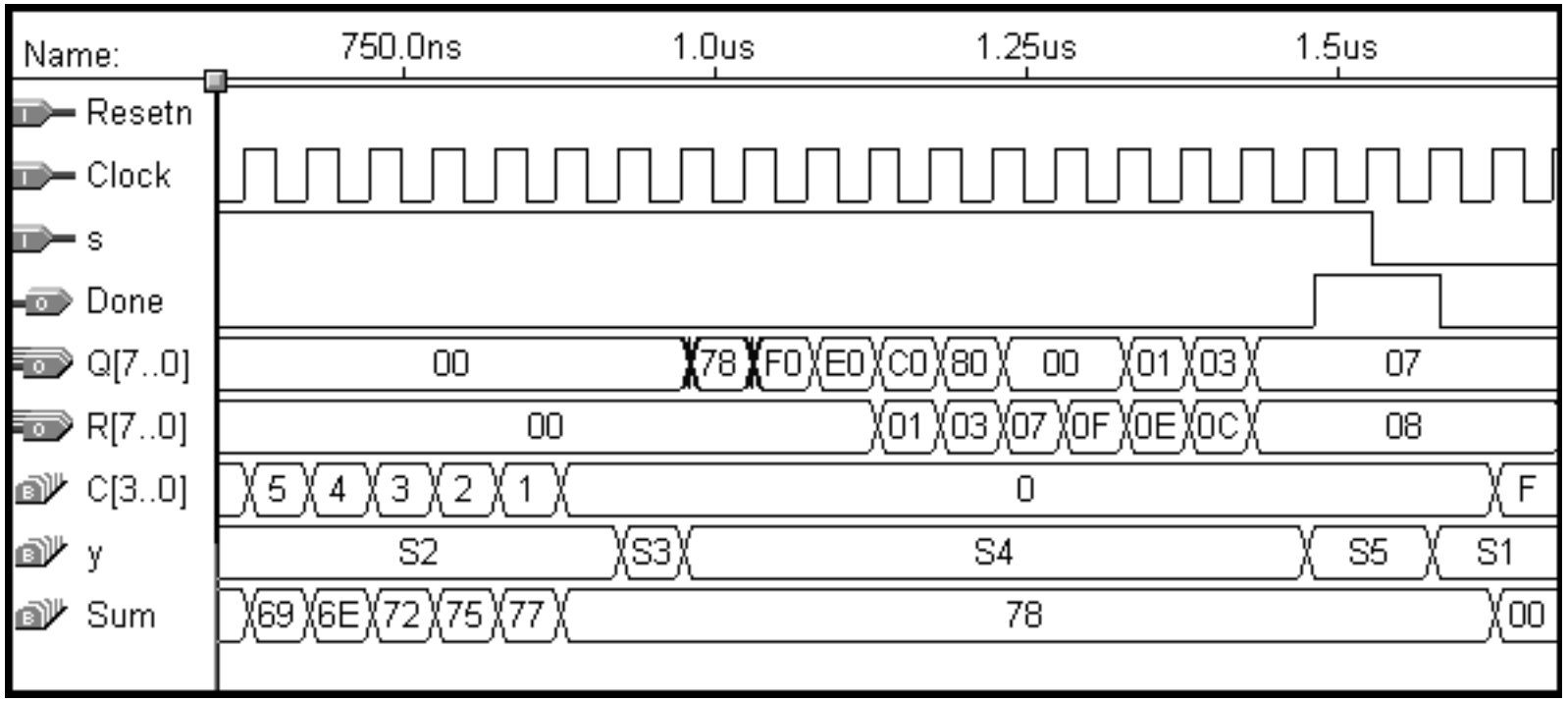
Schematic of the mean circuit with an SRAM block



B&V3, Figure 10.33

Simulation results for the mean circuit using SRAM

Results are for finding the mean of numbers 0..15



B&V3, Figure 10.34

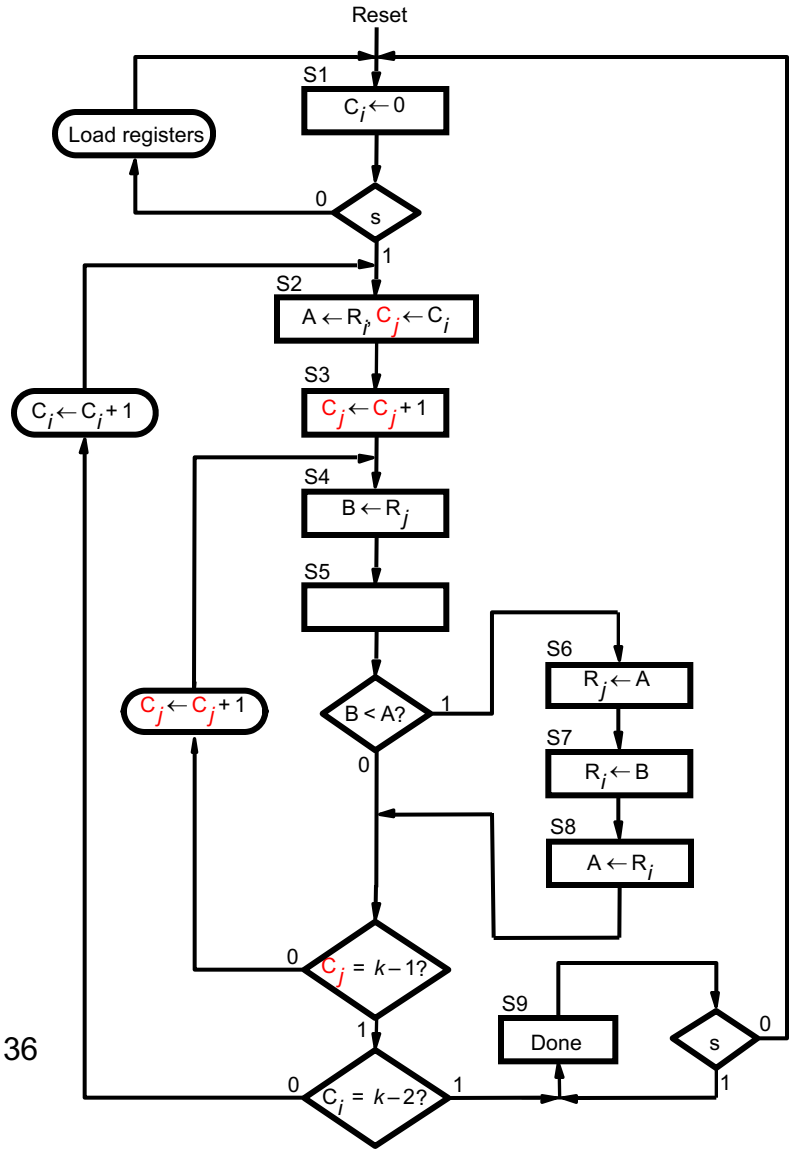
Design Ex 5: Sort *k* words
pp 708 – 718, B&V3

```
// bubblesort
for i = 0 to k-2 do
    A = R_i;
    for j = i+1 to k-1 do
        B = R_j;
        if B < A then
            R_j = A;
            R_i = B; // place smallest of [i+1, k-1] in R_i
            A = R_i;
        end if;
    end for;
end for;
```

B&V3, Figure 10.35

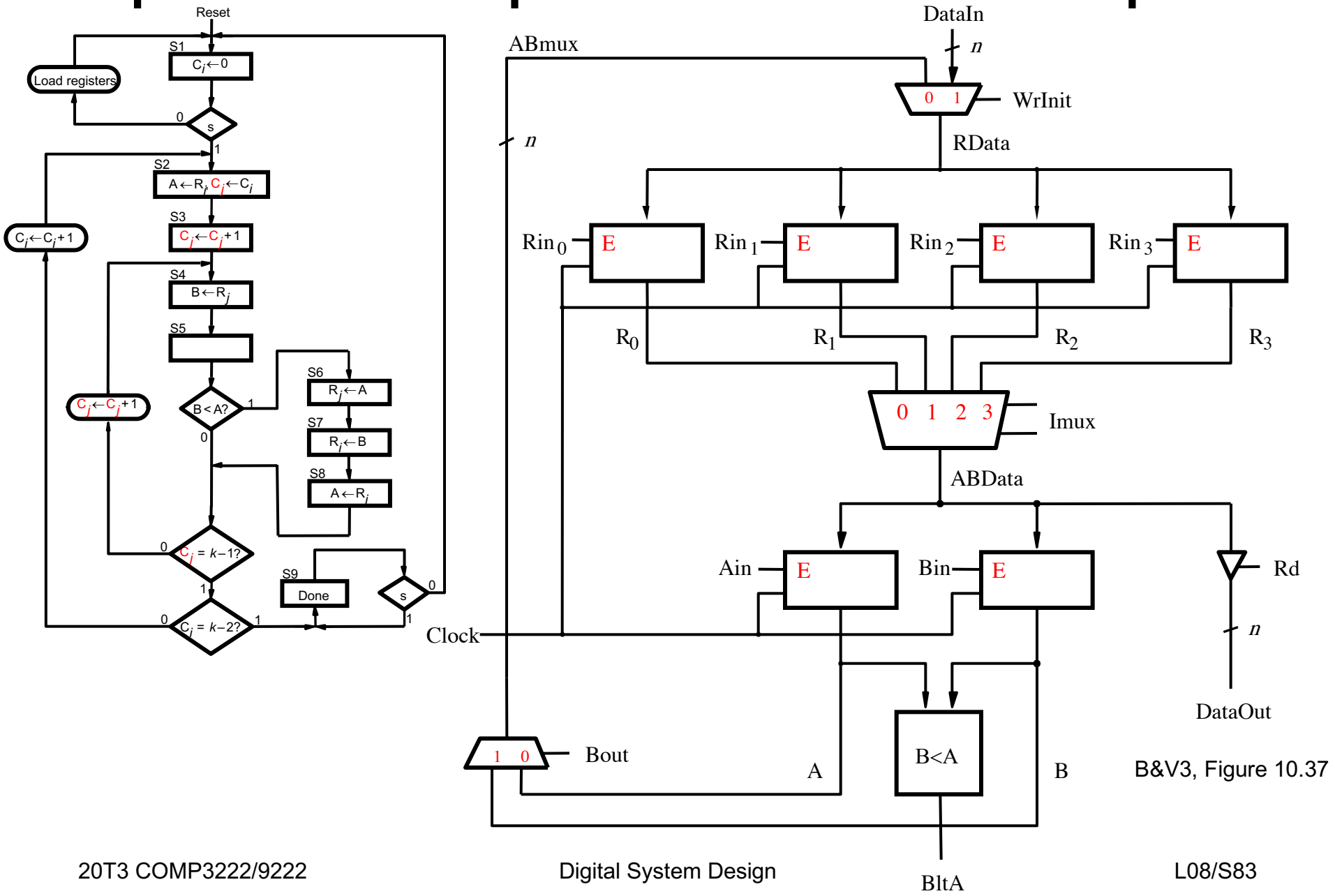
ASM chart for the sort operation

```
for i = 0 to k-2 do
    A = Ri;
    for j = i+1 to k-1 do
        B = Rj;
        if B < A then
            Rj = A;
            Ri = B;
            A = Ri;
        end if;
    end for;
end for;
```



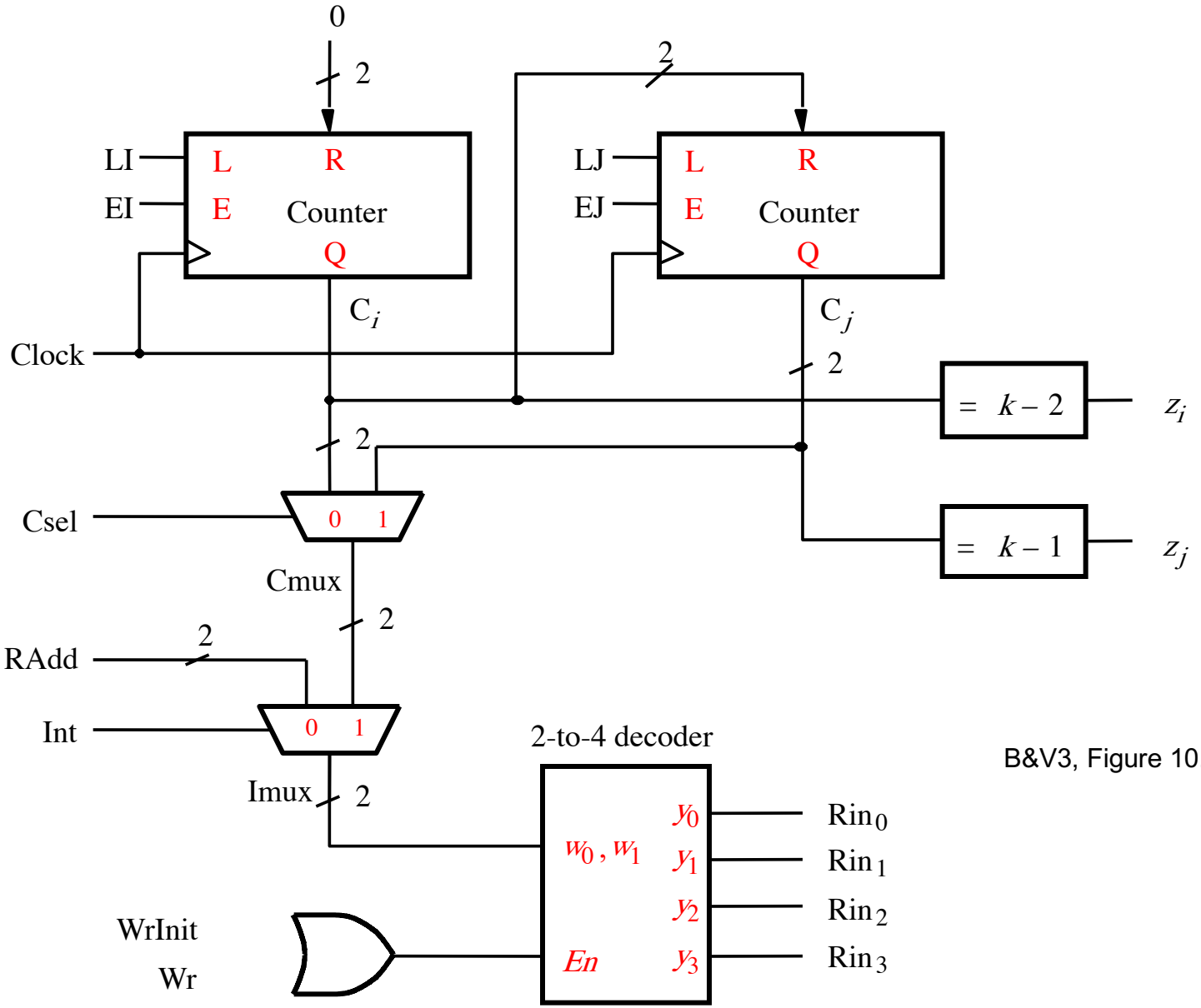
B&V3, Figure 10.36

A part of the datapath circuit for the sort op



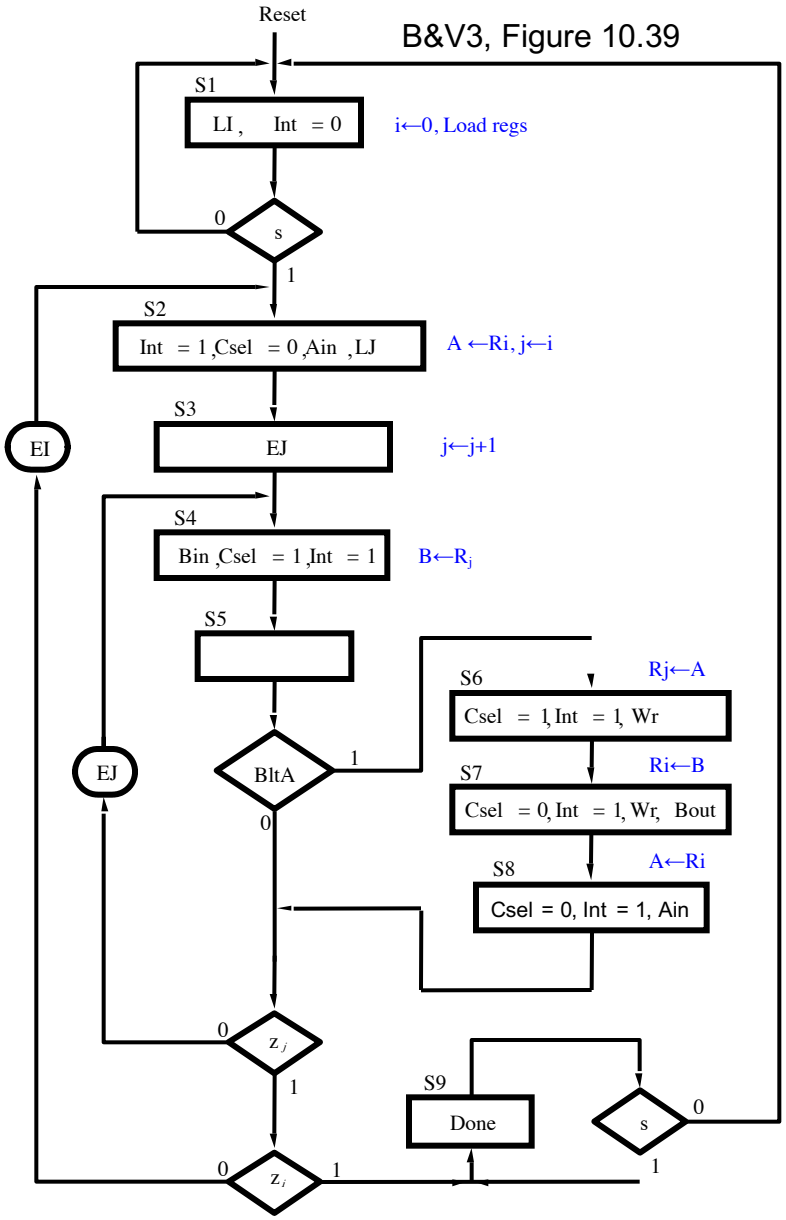
B&V3, Figure 10.37

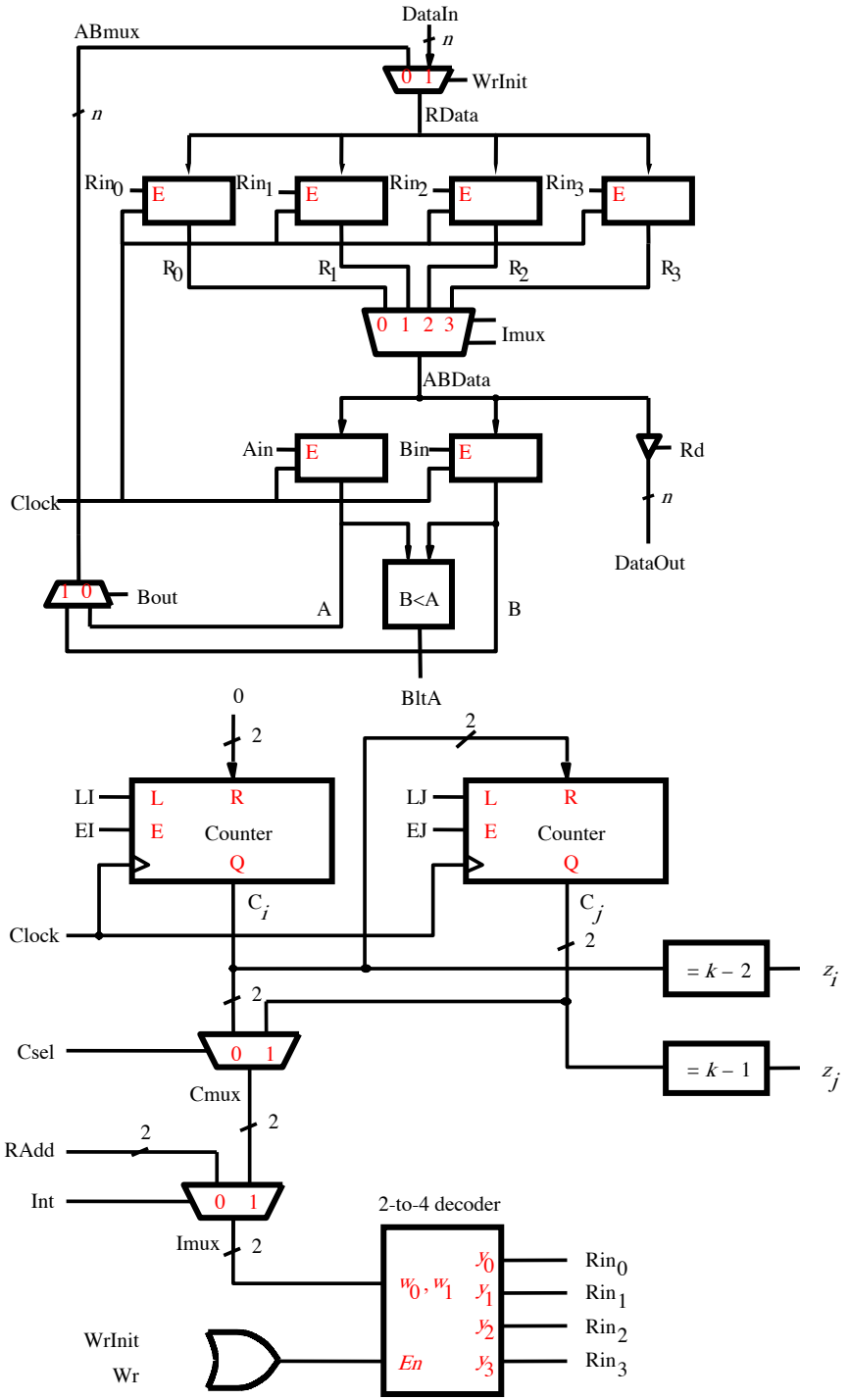
A part of the datapath circuit for the sort op



B&V3, Figure 10.38

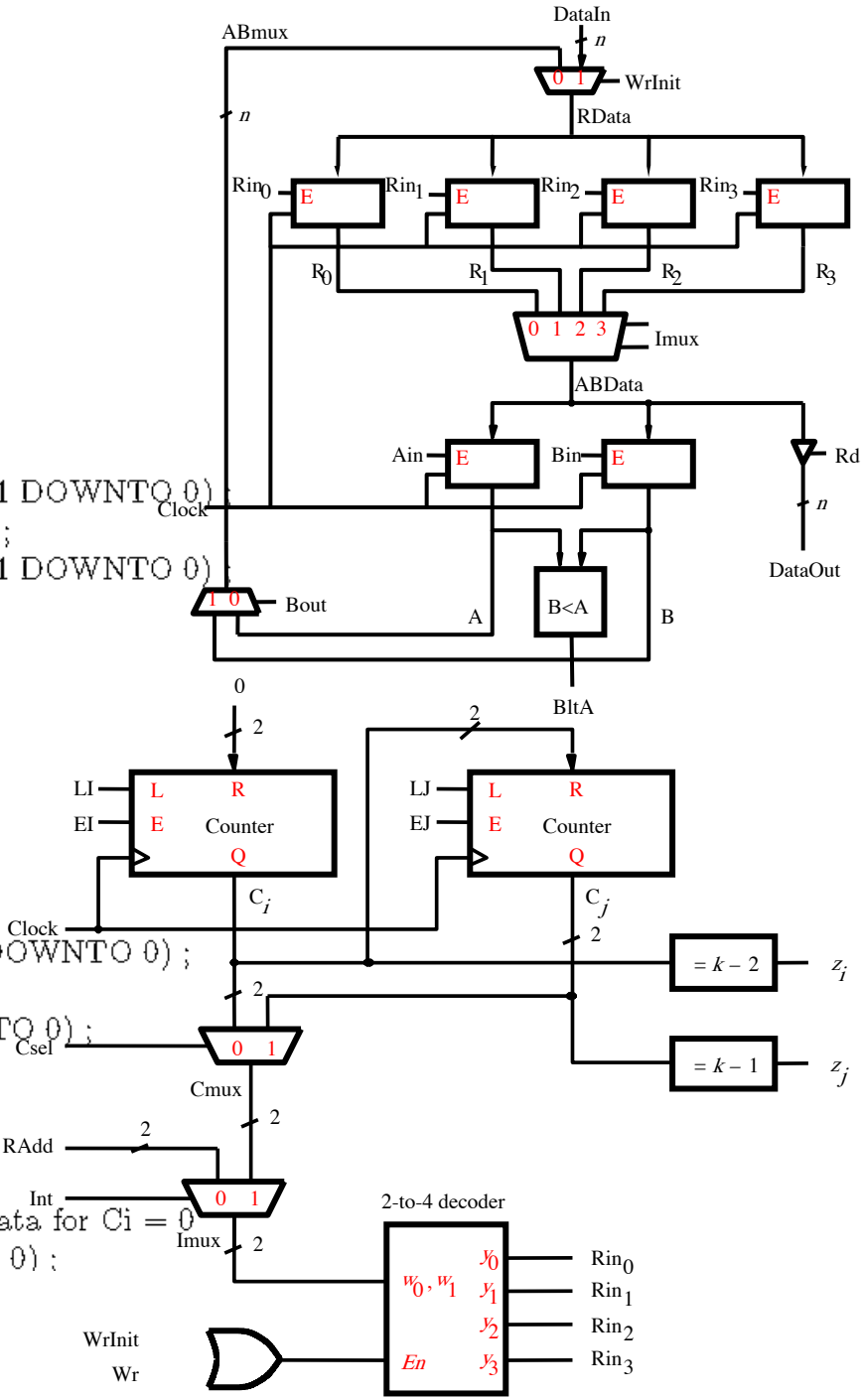
ASM chart for the control circuit





VHDL code for the sort operation (Part *a*)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.components.all ;

ENTITY sort IS
    GENERIC ( N : INTEGER := 4 ) ;
    PORT ( Clock, Resetn   : IN     STD_LOGIC ;
           s, WrInit, Rd   : IN     STD_LOGIC ;
           DataIn          : IN     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           RAdd            : IN     INTEGER RANGE 0 TO 3 ;
           DataOut         : BUFFER STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           Done            : BUFFER STD_LOGIC ) ;
END sort ;

ARCHITECTURE Behavior OF sort IS
    TYPE State_type IS ( S1, S2, S3, S4, S5, S6, S7, S8, S9 ) ;
    SIGNAL y : State_type ;
    SIGNAL Ci, Cj : INTEGER RANGE 0 TO 3 ;
    SIGNAL Rin : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
    TYPE RegArray IS
        ARRAY(3 DOWNTO 0) OF STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
    SIGNAL R : RegArray ;
    SIGNAL RData, ABMux : STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
    SIGNAL Int, Csel, Wr, BltA : STD_LOGIC ;
    SIGNAL CMux, IMux : INTEGER RANGE 0 TO 3 ;
    SIGNAL Ain, Bin,        Bout : STD_LOGIC ;
    SIGNAL LI, LJ, EI, EJ, zi, zj : STD_LOGIC ;
    SIGNAL Zero : INTEGER RANGE 3 DOWNTO 0 ; -- parallel data for Ci = 0
    SIGNAL A, B, ABData : STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
```

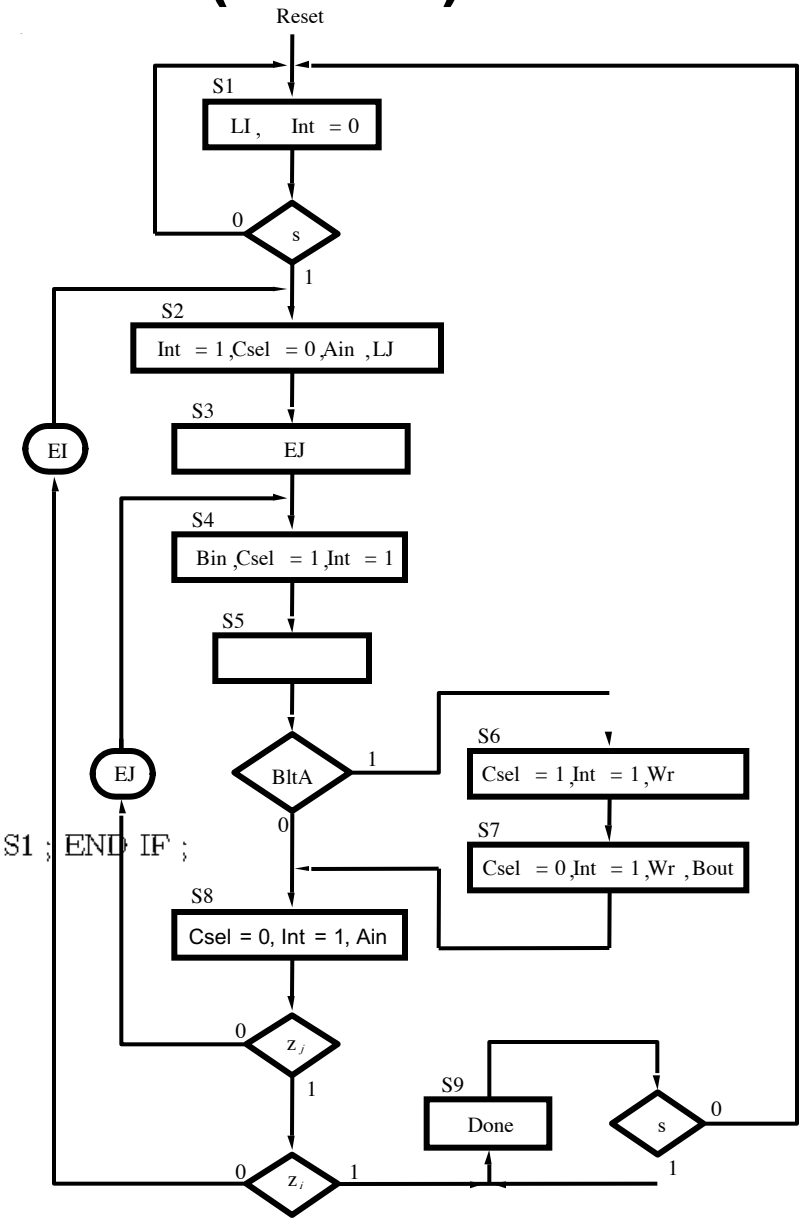


B&V3, Figure 10.40

VHDL code for the sort operation (Part *b*)

```
BEGIN
    FSM_transitions: PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= S1 ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN S1 => IF S = '0' THEN y <= S1 ;
                    ELSE y <= S2 ; END IF ;
                WHEN S2 => y <= S3 ;
                WHEN S3 => y <= S4 ;
                WHEN S4 => y <= S5 ;
                WHEN S5 => IF BltA = '1' THEN y <= S6 ;
                    ELSE y <= S8 ; END IF ;
                WHEN S6 => y <= S7 ;
                WHEN S7 => y <= S8 ;
                WHEN S8 =>
                    IF zj = '0' THEN y <= S4 ;
                    ELSIF zi = '0' THEN y <= S2 ;
                    ELSE y <= S9 ;
                    END IF ;
                WHEN S9 => IF s = '1' THEN y <= S9 ; ELSE y <= S1 ; END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;
```

B&V3, Figure 10.40

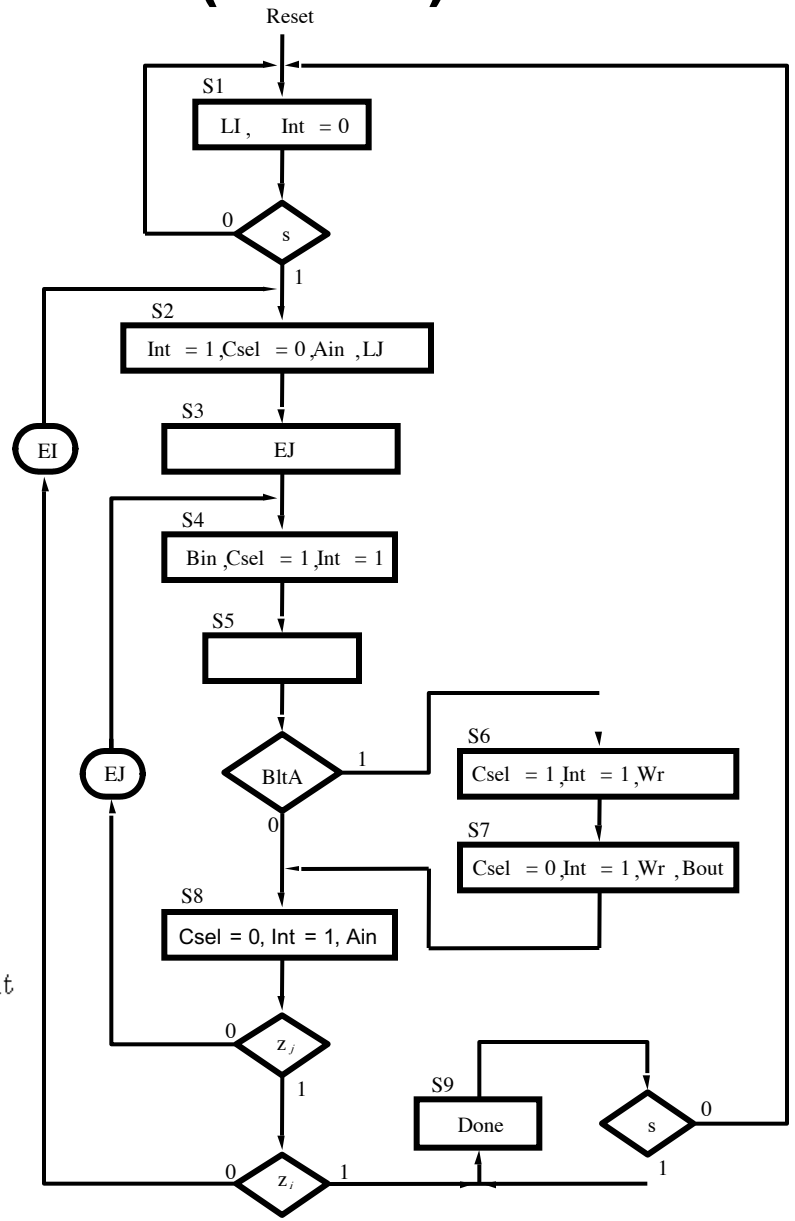


Note that the text executes S8 irrespective of need in order to simplify the control.
What are the costs/benefits of this approach?

VHDL code for the sort operation (Part *c*)

```
-- define the outputs generated by the FSM
Int <= '0' WHEN y = S1 ELSE '1' ;
Done <= '1' WHEN y = S9 ELSE '0' ;
FSM_outputs: PROCESS ( y, zi, zj )
BEGIN
     LI <= '0' ; LJ <= '0' ; EI <= '0' ; EJ <= '0' ; Csel <= '0' ;
     Wr <= '0'; Ain <= '0' ; Bin <= '0' ;                Bout <= '0' ;
     CASE y IS
          WHEN S1 => LI <= '1' ;
          WHEN S2 => Ain <= '1' ; LJ <= '1' ;
          WHEN S3 => EJ <= '1' ;
          WHEN S4 => Bin <= '1' ; Csel <= '1' ;
          WHEN S5 => -- no outputs asserted in this state
          WHEN S6 => Csel <= '1' ; Wr <= '1' ;
          WHEN S7 => Wr <= '1' ; Bout <= '1' ;
          WHEN S8 => Ain <= '1' ;
               IF zj = '0' THEN
                    EJ <= '1' ;
               ELSE
                    EJ <= '0' ;
                    IF zi = '0' THEN
                         EI <= '1' ;
                    ELSE
                         EI <= '0' ;
                    END IF;
               END IF ;
          WHEN S9 => -- Done is assigned 1 by conditional signal assignment
     END CASE ;
END PROCESS ;
```

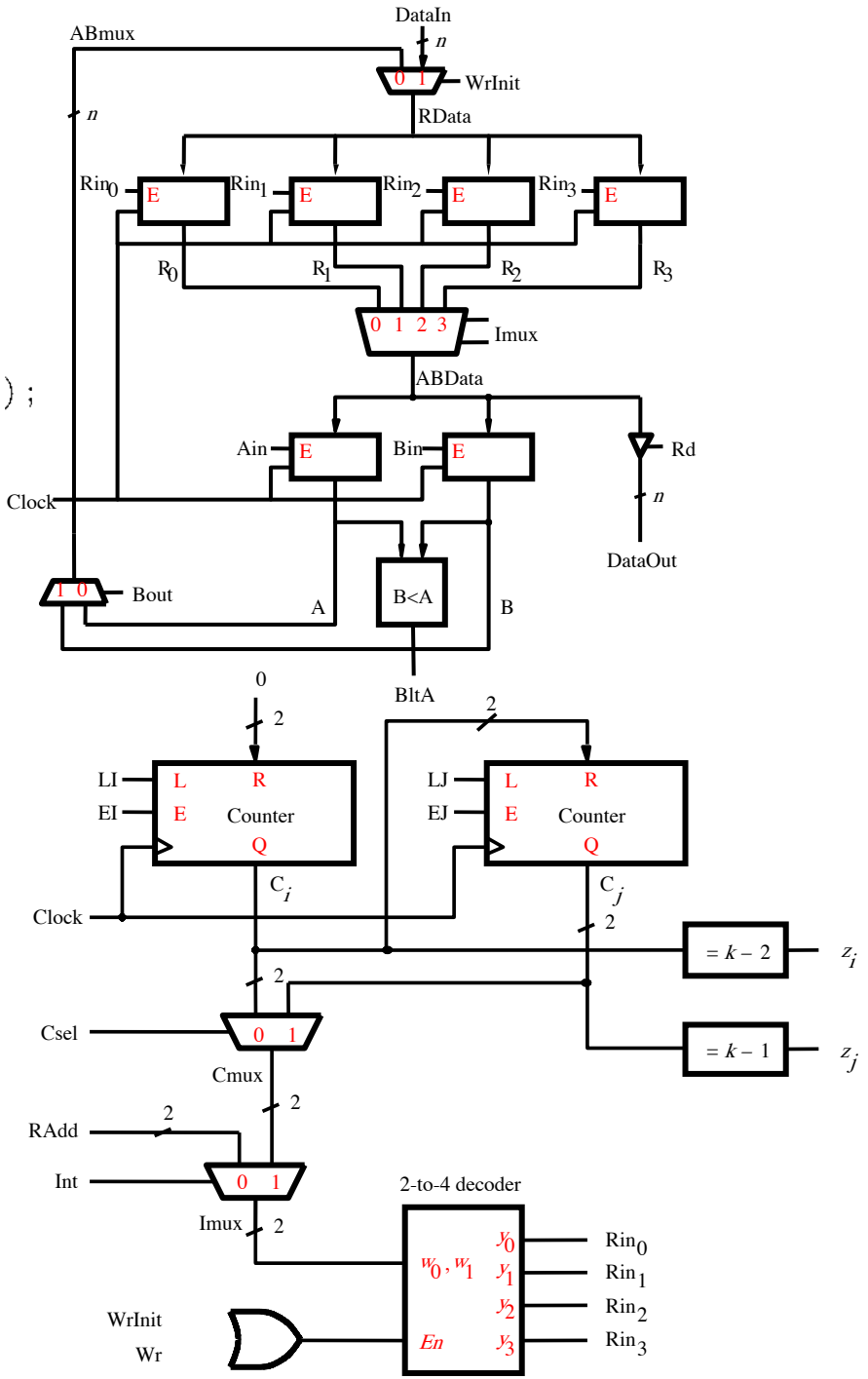
B&V3, Figure 10.40



VHDL code for the sort operation (Part *d*)

```
-- define the datapath circuit
Zero <= 0 ;
GenReg: FOR i IN 0 TO 3 GENERATE
      Reg: regne GENERIC MAP ( N => N )
           PORT MAP ( RData, Resetn, Rin(i), Clock, R(i) ) ;
END GENERATE ;
RegA: regne GENERIC MAP ( N => N )
      PORT MAP ( ABData, Resetn, Ain, Clock, A ) ;
RegB: regne GENERIC MAP ( N => N )
      PORT MAP ( ABData, Resetn, Bin, Clock, B ) ;
BltA <= '1' WHEN B < A ELSE '0' ;
ABMux <= A WHEN Bout = '0' ELSE B ;
RData <= ABMux WHEN WrInit = '0' ELSE DataIn ;
OuterLoop: upcount GENERIC MAP ( modulus => 4 )
      PORT MAP ( Resetn, Clock, EI, LI, Zero, Ci ) ;
InnerLoop: upcount GENERIC MAP ( modulus => 4 )
      PORT MAP ( Resetn, Clock, EJ, LJ, Ci, Cj ) ;
CMux <= Ci WHEN Csel = '0' ELSE Cj ;
IMux <= Cmux WHEN Int = '1' ELSE Radd ;
WITH IMux Select
      ABData <= R(0) WHEN 0,
                R(1) WHEN 1,
                R(2) WHEN 2,
                R(3) WHEN OTHERS ;
```

B&V3, Figure 10.40



VHDL code for the sort operation (Part e)

```
RinDec: PROCESS ( WrInit, Wr, IMux )
BEGIN
    IF (WrInit OR Wr) = '1' THEN
        CASE IMux IS
            WHEN 0 => Rin <= "0001" ;
            WHEN 1 => Rin <= "0010" ;
            WHEN 2 => Rin <= "0100" ;
            WHEN OTHERS => Rin <= "1000" ;
        END CASE ;
    ELSE Rin <= "0000" ;
    END IF ;
END PROCESS ;
Zi <= '1' WHEN Ci = 2 ELSE '0' ;
Zj <= '1' WHEN Cj = 3 ELSE '0' ;
DataOut <= (OTHERS => 'Z') WHEN Rd = '0' ELSE ABData ;
END Behavior ;
```

B&V3, Figure 10.40