

Register Transfer Methodology I

RTL Hardware Design
by P. Chu

Chapter 11

1

Outline

1. Introduction
2. Overview of FSM
3. FSM design of a repetitive-addition multiplier
4. Alternative design of a repetitive-addition multiplier
5. Timing and performance analysis of FSM
6. Sequential add-and-shift multiplier

RTL Hardware Design
by P. Chu

Chapter 11

2

1. Introduction

- How to realize an algorithm in hardware?
- Two characteristics of an algorithm:
 - Use of variables (symbolic memory location)
e.g., $n = n + 1$ in C
 - Sequential execution
(execution order is important)

RTL Hardware Design
by P. Chu

Chapter 11

3

- E.g., an algorithm:
 - Summate 4 number
 - Divide the result by 8
 - Round the result
- Pseudocode

```
size = 4
sum = 0;
for i in (0 to size-1) do {
    sum = sum + a(i);}
q = sum / 8;
r = sum rem 8;
if (r > 3) {
    q = q+1;}
outp = q;
```

RTL Hardware Design
by P. Chu

Chapter 11

4

- “Dataflow” implementation in VHDL
 - Convert the algorithm in to combinational circuit
 - No memory elements
 - The sequence is embedded into the “flow of data”

RTL Hardware Design
by P. Chu

Chapter 11

5

- VHDL code

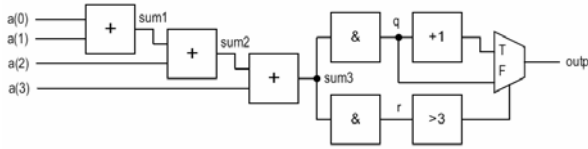
```
sum <= 0;
sum0 <= a(0);
sum1 <= sum0 + a(1);
sum2 <= sum1 + a(2);
sum3 <= sum2 + a(3);
q <= "000" & sum3(8 downto 3);
r <= "00000" & sum3(2 downto 0);
outp <= q + 1 when (r > 3) else
    q;
```

RTL Hardware Design
by P. Chu

Chapter 11

6

- Block diagram



- Problems with dataflow implementation:

- Can only be applied to trivial algorithm
- Not flexible
 - Can we just share one adder in a time-multiplexing fashion to save hardware resources
 - What happen if input size is not fixed (i.e., size is determined by an external input)

Register Transfer Methodology

- Realized algorithm in hardware
- Use register to store intermediate data and imitate variable
- Use a datapath to realize all register operations
- Use a control path (FSM) to specify the order of register operation

- The system is specified as sequence of data manipulation/transfer among registers
- Realized by FSM with a datapath (FSMD)

2. Overview of FSMD

Basic RT operation

- Basic form:

$$r_{\text{dest}} \leftarrow f(r_{\text{src1}}, r_{\text{src2}}, \dots, r_{\text{srcn}})$$

- Interpretation:

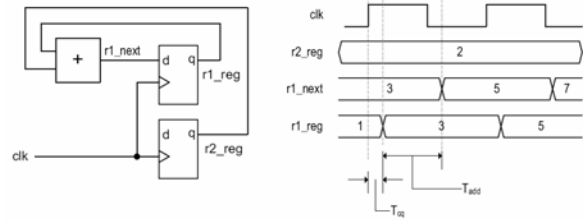
- At the rising edge of the clock, the output of registers r_{src1} , r_{src2} , etc are available
- The output are passed to a combinational circuit that performs $f()$
- At the next rising edge of the clock, the result is stored into r_{dest}

- E.g.,

$$\begin{aligned}
 r &\leftarrow 1 \\
 r &\leftarrow r \\
 r0 &\leftarrow r1 \\
 n &\leftarrow n - 1 \\
 y &\leftarrow a \oplus b \oplus c \oplus d \\
 s &\leftarrow a^2 + b^2
 \end{aligned}$$

• Implementation example

$$r1 \leftarrow r1+r2$$

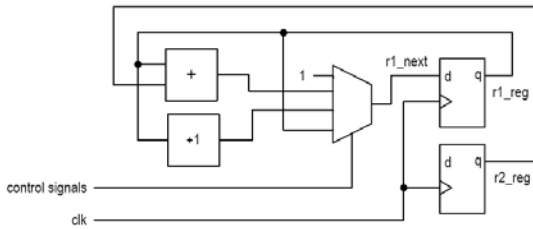


• Multiple RT operations

```

r1 ← 1;
r1 ← r1 + r2;
r1 ← r1 + 1;
r1 ← r1;

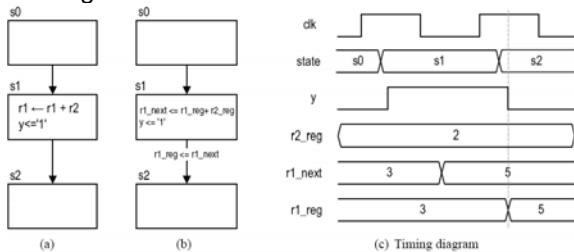
```



FSM as control path

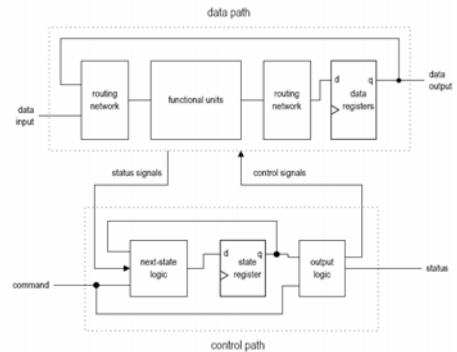
- FSM is a good to control RT operation
 - State transition is on clock-by-clock basis
 - FSM can enforce order of execution
 - FSM allows branches on execution sequence
- Normally represented in an extended ASM chart known as ASMD (ASM with datapath) chart

• E.g.



- Note: new value of r1 is only available when the FSM exits s1state

Basic Block Diagram of FSMD



3. FSM design example: Repetitive addition multiplier

- Basic algorithm: $7*5 = 7+7+7+7+7$
- Pseudo code

```

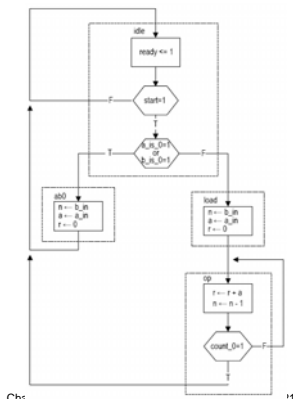
if (a_in=0 or b_in=0) then {
    r = 0;}
else{
    a = a_in;
    n = b_in;
    r = 0;
    while (n != 0){
        r = r + a;
        n = n-1;}
}
return (r)
    
```

- ASMD-friendly code

```

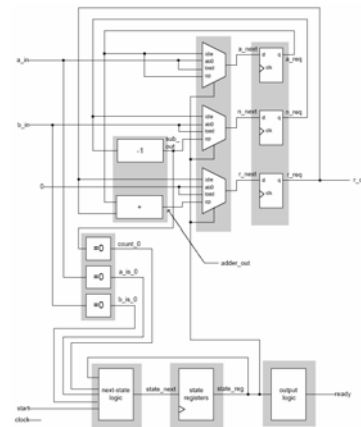
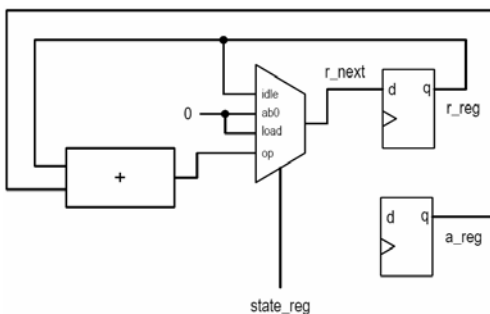
if (a_in=0 or b_in=0) then {
    r = 0;}
else{
    a = a_in;
    n = b_in;
    r = 0;
op:   r = r + a;
      n = n-1;
      if (n = 0) then{
          goto stop;}
      else{
          goto op;}
}
stop: return (r);
    
```

- Input:
 - a_in, b_in: 8-bit unsigned
 - clk, reset
 - start: command
- Output:
 - r: 16-bit unsigned
 - ready: status
- ASMD chart
 - Default RT operation: keep the previous value
 - Note the parallel execution in op state



- Construction of the data path
 - List all RT operations
 - Group RT operation according to the destination register
 - Add combinational circuit/mux
 - Add status circuits
- E.g.
 - RT operations with the r register:
 - $r \leftarrow r$ (in the idle state)
 - $r \leftarrow 0$ (in the load and op states)
 - $r \leftarrow r + b$ (in the op state)
 - RT operations with the n register:
 - $n \leftarrow n$ (in the idle state)
 - $n \leftarrow a_{in}$ (in the load and ab0 states)
 - $n \leftarrow n - 1$ (in the op state)
 - RT operations with the b register:
 - $b \leftarrow b$ (in the idle and op states)
 - $b \leftarrow b_{in}$ (in the load and ab0 states)

- E.g., Circuit associated with r register



- VHDL code: follow the block diagram

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity seq_mult is
  port(
    clk, reset: in std_logic;
    start: in std_logic;
    a_in, b_in: in std_logic_vector(7 downto 0);
    ready: out std_logic;
    r: out std_logic_vector(15 downto 0)
  );
end seq_mult;

```

RTL Hardware Design
by P. Chu

Chapter 11

25

```

-- control path: state register
process(clk,reset)
begin
  if reset='1' then
    state_reg <= idle;
  elsif (clk'event and clk='1') then
    state_reg <= state_next;
  end if;
end process;

```

RTL Hardware Design
by P. Chu

Chapter 11

26

```

-- control path: next-state/output logic
process(state_reg,start,a_is_0,b_is_0,count_0)
begin
  case state_reg is
    when idle =>
      if start='1' then
        if (a_is_0='1' or b_is_0='1') then
          state_next <= ab0;
        else
          state_next <= load;
        end if;
      else
        state_next <= idle;
      end if;
    when ab0 =>
      state_next <= idle;
    when load =>
      state_next <= op;
    when op =>
      if count_0='1' then
        state_next <= idle;
      else
        state_next <= op;
      end if;
  end case;
end process;

```

RTL Hardw
by P. Chu

27

```

-- control path: output logic
ready <= '1' when state_reg=idle else '0';
-- data path: data register
process(clk,reset)
begin
  if reset='1' then
    a_reg <= (others=>'0');
    n_reg <= (others=>'0');
    r_reg <= (others=>'0');
  elsif (clk'event and clk='1') then
    a_reg <= a_next;
    n_reg <= n_next;
    r_reg <= r_next;
  end if;
end process;

```

RTL Hardware Design
by P. Chu

Chapter 11

28

```

-- data path: routing multiplexer
process(state_reg,a_reg,n_reg,r_reg,
  a_in,b_in,adder_out,sub_out)
begin
  case state_reg is
    when idle =>
      a_next <= a_reg;
      n_next <= n_reg;
      r_next <= r_reg;
    when ab0 =>
      a_next <= unsigned(a_in);
      n_next <= unsigned(b_in);
      r_next <= (others=>'0');
    when load =>
      a_next <= unsigned(a_in);
      n_next <= unsigned(b_in);
      r_next <= (others=>'0');
    when op =>
      a_next <= a_reg;
      n_next <= sub_out;
      r_next <= adder_out;
  end case;
end process;

```

RTL Hardware D
by P. Chu

29

```

-- data path: functional units
adder_out <= ("00000000" & a_reg) + r_reg;
sub_out <= n_reg - 1;
-- data path: status
a_is_0 <= '1' when a_in="00000000" else '0';
b_is_0 <= '1' when b_in="00000000" else '0';
count_0 <= '1' when n_next="00000000" else '0';
-- data path: output
r <= std_logic_vector(r_reg);

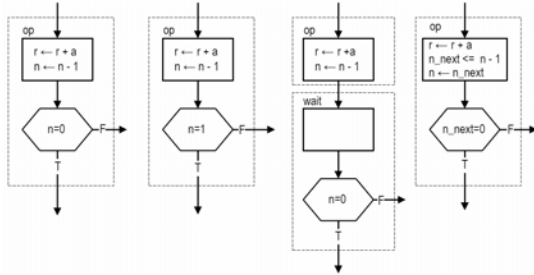
```

RTL Hardware Design
by P. Chu

Chapter 11

30

- Use of register in decision box
 - Register is updated when the FSM exits current state
 - How to represent count_0='1' using register?



- Other VHDL coding styles:
 - Various code segments can be combined
 - Should always separate registers from combinational logic
 - May be a good idea to isolate the main functional units

- E.g., 2-segment code

```

-- state and data register
process (clk, reset)
begin
  if reset='1' then
    state_reg <= idle;
    a_reg <= (others=>'0');
    n_reg <= (others=>'0');
    r_reg <= (others=>'0');
  elsif (clk'event and clk='1') then
    state_reg <= state_next;
    a_reg <= a_next;
    n_reg <= n_next;
    r_reg <= r_next;
  end if;
end process;

```

```

-- combinational circuit
process (start, state_reg, a_reg, n_reg, r_reg,
        a_in, b_in, n_next)
begin
  -- default value
  a_next <= a_reg;
  n_next <= n_reg;
  r_next <= r_reg;
  ready <='0';
  case state_reg is
    when idle =>
      if start='1' then
        if (a_in="00000000" or b_in="00000000") then
          state_next <= ab0;
        else
          state_next <= load;
        end if;
      else
        state_next <= idle;
      end if;
      ready <='1';
    ...
  end case;
end process;

```

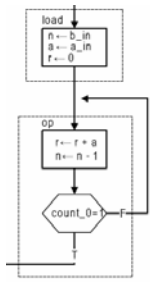
```

when ab0 =>
  a_next <= unsigned(a_in);
  n_next <= unsigned(b_in);
  r_next <= (others=>'0');
  state_next <= idle;
when load =>
  a_next <= unsigned(a_in);
  n_next <= unsigned(b_in);
  r_next <= (others=>'0');
  state_next <= op;
  ready <='0';
when op =>
  n_next <= n_reg - 1;
  r_next <= ("00000000" & a_reg) + r_reg;
  if (n_next="00000000") then
    state_next <= idle;
  else
    state_next <= op;
  end if;
  ready <='0';
end case;
end process;
r_<=_std_logic_vector(r_reg);

```

4. Alternative design of a repetitive-addition multiplier

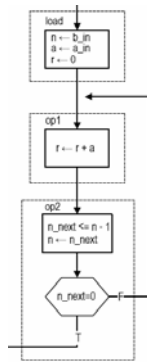
- Resource sharing
 - Hardware can be shared in a time-multiplexing fashion
 - Assign the operation in different states
 - Most complex circuits in the FSM design is normally the functional units of the datapath
- Sharing in repetitive addition multiplier
 - Addition and decrementing
 - The same adder can be used in 2 states



RTL Hardware Design
by P. Chu

Chapter 11

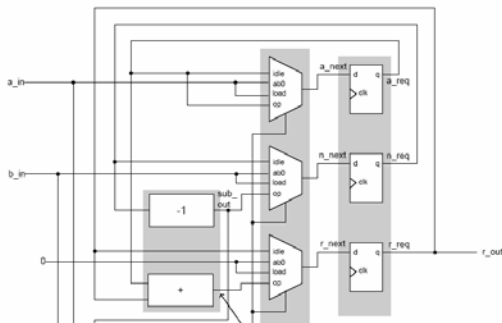
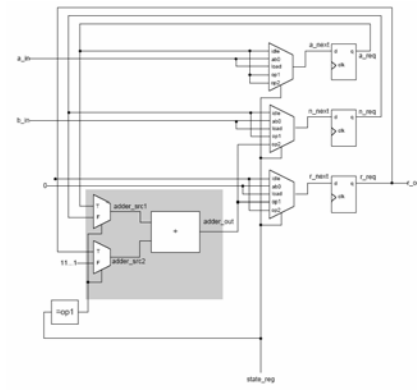
37



RTL Hardware Design
by P. Chu

Chapter 11

38



RTL Hardware Design
by P. Chu

Chapter 11

39

```

when op1 =>
  r_next <= adder_out;
  state_next <= op2;
when op2 =>
  n_next <= adder_out(WIDTH-1 downto 0);
  if (n_next="00000000") then
    state_next <= idle;
  else
    state_next <= op1;
  end if;

```

RTL Hardware Design
by P. Chu

Chapter 11

40

```

-- data path input routing and junctional units
process(state_reg,r_reg, a_reg, n_reg)
begin
  if (state_reg=op1) then
    adder_src1 <= r_reg;
    adder_src2 <= "00000000" & a_reg;
  else -- for op2 state
    adder_src1 <= "00000000" & n_reg;
    adder_src2 <= (others=>'1');
  end if;
end process;
adder_out <= adder_src1 + adder_src2;

```

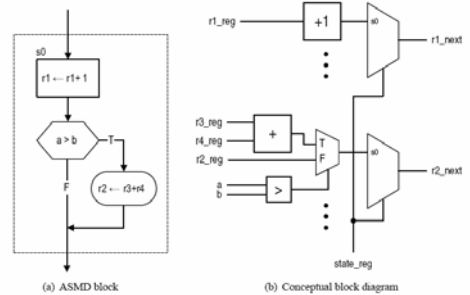
RTL Hardware Design
by P. Chu

Chapter 11

41

• Mealy-controlled operation

- Control signals is edge-sensitive
- Mealy output is faster and requires fewer states
- E.g.:



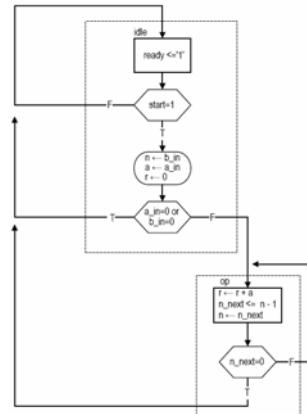
RTL Hardware Design
by P. Chu

(a) ASMD block

(b) Conceptual block diagram

- Mealy control signal for multiplier
 - load and ab0 states perform no computation
 - Mealy control can be used to eliminate ab0 and load states

- r, n, b register loaded using Mealy signal



```

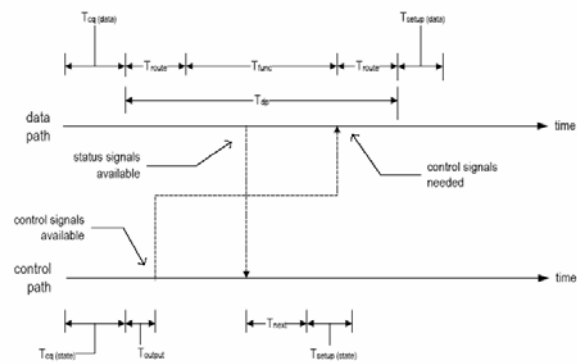
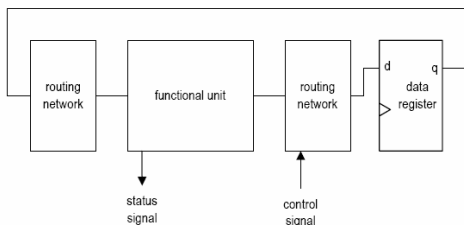
case state_reg is
when idle =>
  if start='1' then
    a_next <= unsigned(a_in);
    n_next <= unsigned(b_in);
    r_next <= (others=>'0');
    if a_in="00000000" or b_in="00000000" then
      state_next <= idle;
    else
      state_next <= op;
    end if;
  else
    state_next <= idle;
  end if;
  ready <='1';
when op =>
  n_next <= n_reg - 1;
  r_next <= ("00000000" & a_reg) + r_reg;
  if (n_next="00000000") then
    state_next <= idle;
  else
    state_next <= op;
  end if;
end if;

```

5. Clock rate and Performance of FSM

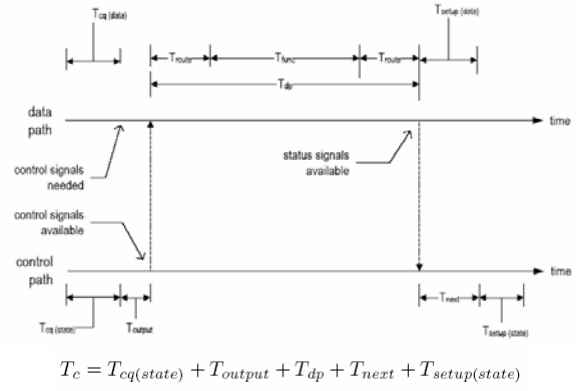
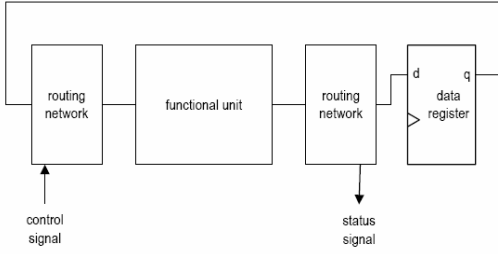
- Maximal clock rate
 - More difficult to analyze because of two interactive loops
 - The boundary of the clock rate can be found

- Best-case scenario:
 - Control signals needed at late stage
 - Status signal available at early stage



$$T_c = T_{cq}(data) + T_{dp} + T_{setup}(data)$$

- Best-case scenario:
 - Control signals needed at early stage
 - Status signal available at late stage

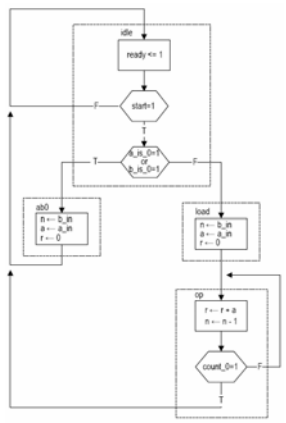


$$T_{cq} + T_{dp} + T_{setup} \leq T_c \leq T_{cq} + T_{output} + T_{dp} + T_{next} + T_{setup}$$

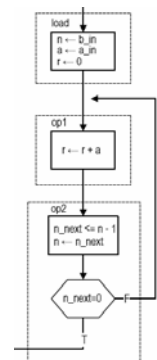
$$\frac{1}{T_{cq} + T_{output} + T_{dp} + T_{next} + T_{setup}} \leq f \leq \frac{1}{T_{cq} + T_{dp} + T_{setup}}$$

- Performance of FSMD
 - Tc: Clock period
 - K: # clock cycles to complete the computation
 - Total time = K * Tc
- K determined by algorithm, input patterns etc.

- 8-bit input
 - Best: b=0, K=2
 - Worst: b=255, K=257
- N-bit input:
 - Worst: K = 2+(2ⁿ-1)



- 8-bit input
 - Best: b=0, K=2
 - Worst: b=255, K=2 + 255*2
- N-bit input:
 - Worst: K=2+2*(2ⁿ-1)



6. Sequential add-and-shift multiplier

×		a_3	a_2	a_1	a_0	multiplicand				
		b_3	b_2	b_1	b_0	multiplier				
			a_3b_0	a_2b_0	a_1b_0	a_0b_0				
		a_3b_1	a_2b_1	a_1b_1	a_0b_1					
		a_3b_2	a_2b_2	a_1b_2	a_0b_2					
+	a_3b_3	a_2b_3	a_1b_3	a_0b_3						
	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0	product	

RTL Hardware Design
by P. Chu

Chapter 11

55

```

a = a_in;
b = b_in;
n = 8;
p = 0;
while (n!=0) {
  if (b(0)=1) then{
    p = p + a;}
  a = a << 1;
  b = b >> 1;
  n = n-1;}
return(p);

```

RTL Hardware Design
by P. Chu

Chapter 11

57

```

architecture shift_add_raw_arch of seq_mult is
  constant WIDTH: integer:=8;
  constant C_WIDTH: integer:=4; -- width of the counter
  constant C_INIT: unsigned(C_WIDTH-1 downto 0):="1000";
  type state_type is (idle, add, shift);
  signal state_reg, state_next: state_type;
  signal b_reg, b_next: unsigned(WIDTH-1 downto 0);
  signal a_reg, a_next: unsigned(2*WIDTH-1 downto 0);
  signal n_reg, n_next: unsigned(C_WIDTH-1 downto 0);
  signal p_reg, p_next: unsigned(2*WIDTH-1 downto 0);

```

RTL Hardware Design
by P. Chu

Chapter 11

59

1. Multiply the digits of the multiplier (b_3, b_2, b_1 and b_0) by the multiplicand (A) one at a time to obtain $b_3 * A, b_2 * A, b_1 * A$ and $b_0 * A$. The $b_i * A$ operation is bitwise and operations of b_i and the digits of A :

$$b_i * A = (a_3 \cdot b_i, a_2 \cdot b_i, a_1 \cdot b_i, a_0 \cdot b_i)$$

2. Shift $b_i * A$ to the left by i positions according to the position of digits b_i .
3. Add the shifted $b_i * A$ to obtain the final product.

```

n = 0;
p = 0;
while (n!=8) {
  if (b_in(n)=1) then{
    p = p + (a_in << n);}
  n = n+1;
}
return(p);

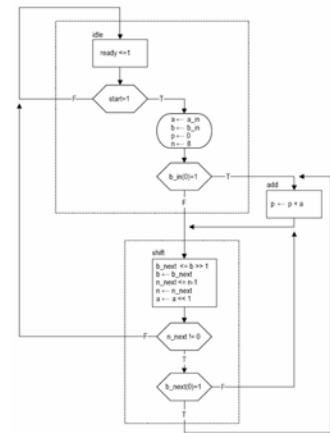
```

RTL Hardware Design
by P. Chu

Chapter 11

56

- Note the use of b_next and n_next
- $a \ll 1$ and $b \gg 1$ require no logic
- 8-bit input
 - Best: $b=0, K=1+8$
 - Worst: $b=255, K=1+8*2$
- N-bit input:
 - Worst: $K=2+2*n$



RTL Hardware Design
by P. Chu

```

-- combinational circuit
process(start, state_reg, b_reg, a_reg,
  n_reg, p_reg, b_in, a_in, n_next)
begin
  b_next <= b_reg;
  a_next <= a_reg;
  n_next <= n_reg;
  p_next <= p_reg;
  ready <='0';
  case state_reg is
    when idle =>
      if start='1' then
        b_next <= unsigned(b_in);
        a_next <= "00000000" & unsigned(a_in);
        n_next <= C_INIT;
        p_next <= (others=>'0');
        if b_in(0)='1' then
          state_next <= add;
        else
          state_next <= shift;
        end if;
      else
        state_next <= idle;
      end if;
      ready <='1';
    when add =>
      p_next <= p_reg + a_reg;
      state_next <= shift;

```

RTL Hardware
by P. Chu

60

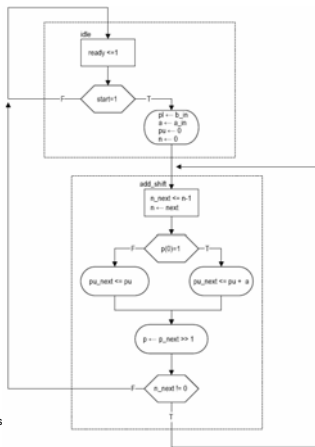
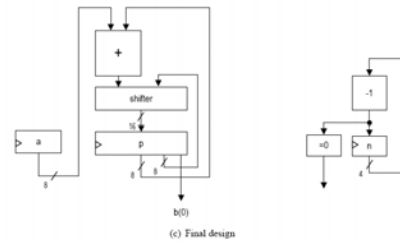
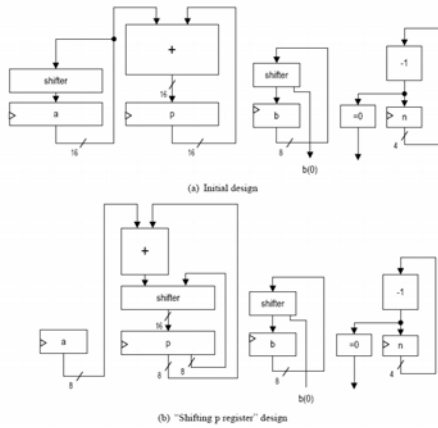
```

when shift =>
  n_next <= n_reg - 1;
  b_next <= '0' & b_reg (WIDTH-1 downto 1);
  a_next <= a_reg(2*WIDTH-2 downto 0) & '0';
  if (n_next /= "0000") then
    if a_next(0)='1' then
      state_next <= add;
    else
      state_next <= shift;
    end if;
  else
    state_next <= idle;
  end if;
end case;
end process;

```

• Refinement

- No major computation done in the shift state: the add and shift states can be merged
- Data path can be simplified:
 - Replace 2n-bit adder with (n+1)-bit adder
 - Reduce the a register from 2n bits to n bits
 - Use the lower part of the p register to store B and eliminate the b register



Design method	# Clock cycles	Size of functional units	# Register bits
Repetitive-addition	2 to 2 ⁿ + 1	2n-bit adder, n-bit decrementor	4n
Add-and-shift (original)	n + 1 to 2n + 1	2n-bit adder, [log ₂ (n + 1)]-bit dec	5n + [log ₂ (n + 1)]
Add-and-shift (refined)	n + 1	n-bit adder, [log ₂ (n + 1)]-bit dec	3n + [log ₂ (n + 1)] + 1