4.1 Design of a Serial Adder with Accumulator - Schematic

UAH

Chapter 4

CPE/EE 422/522

4.1 Design of Networks for Arithmetic Operations

Dr. Rhonda Kay Gaede

UAH

Electrical and Computer Engineering
4.1 Design of a Serial Adder with Accumulator - Operation

<table>
<thead>
<tr>
<th>t</th>
<th>X</th>
<th>Y</th>
<th>C_i</th>
<th>s</th>
<th>C_{i+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0101</td>
<td>0111</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>1011</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0001</td>
<td>1101</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1000</td>
<td>1110</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1100</td>
<td>0111</td>
<td>0</td>
<td>(1)</td>
<td>(0)</td>
</tr>
</tbody>
</table>

- Use variable names instead of 0s and 1s
  - E.g., X_i X_j / Z_p Z_q
  - If X_i and X_j inputs are 1, the outputs Z_p and Z_q are 1
    (all other outputs are 0s)
- E.g., X = X_1 X_2 X_3 X_4, Z = Z_1 Z_2 Z_3 Z_4
  - X_1 X_4 / Z_2 Z_3 == 1 - - 0 / 0 1 1 0

4.2 State Graphs for Control Networks

- Use variable names instead of 0s and 1s
  - E.g., X_i X_j / Z_p Z_q
  - If X_i and X_j inputs are 1, the outputs Z_p and Z_q are 1
    (all other outputs are 0s)
4.2 State Graphs for Control Networks
- Constraints on Input Labels

¥ Assume: \( I \) — input expression =>
we traverse the arc when \( I=1 \)

1. If \( I_i \) and \( I_j \) are any pair of input labels on arcs exiting state \( S_k \), then \( I_i I_j = 0 \) if \( i \neq j \).
   Assures that at most one input label can be 1 at any given time

2. If \( n \) arcs exit state \( S_k \) and the \( n \) arcs have input labels \( I_1, I_2, ..., I_n \), respectively, then \( I_1 + I_2 + ... + I_n = 1 \).
   Assures that at least one input label will be 1 at any given time

1 + 2: Exactly one label will be 1 =>
the next state will be uniquely defined for every input combination

4.2 State Graphs for Control Networks
- Constraints on Input Labels (cont'd)

Inputs are \( X_1 X_2 X_3 \)
\((X_1 = X_2 = 1 \text{ not allowed})\)
4.3 Design of a Binary Multiplier - Terms

Note: we use unsigned binary numbers

4.3 Design of a Binary Multiplier - Block Diagram

Ad — add signal // adder outputs are stored into the ACC
Sh — shift signal // shift all 9 bits to right
Ld — load signal // load multiplier into the 4 lower bits of the ACC and clear the upper 5 bits
4.3 Design of a Binary Multiplier -
Multiplication Example

initial contents of product register:
(0 0 0 0 0 1 0 1 1) — M (11)
(add multiplicand since M=1)
after addition:
(1 1 0 1 1 0 1 1)
after shift:
(0 0 1 1 0 1 1 0 — M)
(add multiplicand since M=1)
after addition:
(1 0 0 1 1 1 1 0 — M)
after shift:
(0 1 0 0 1 1 1 1 — M)
(skip addition since M=0)
after shift:
(0 0 1 0 0 1 1 1 — M)
(add multiplicand since M=1)
after addition:
(1 1 0 1)
after addition (final answer):
(0 1 0 0 0 1 1 1 — M)

dividing line between product and multiplier

4.3 Design of a Binary Multiplier -
State Graph

Electrical and Computer Engineering

Spring 2004 Slide #9

Electrical and Computer Engineering

Spring 2004 Slide #10
library BITLIB;
use BITLIB.bit_pack.all;

entity mult4X4 is
  port (Clk, St: in bit;
        Mplier,Mcand : in bit_vector(3 downto 0);
        Done: out bit);
end mult4X4;

architecture behav of mult4X4 is

  signal State: integer range 0 to 9;
  signal ACC: bit_vector(8 downto 0); -- accumulator

  alias M: bit is ACC(0); -- M is bit 0 of ACC

begin
  process
    begin
      wait until Clk = '1'; -- executes on rising edge of clock
      case State is
        when 0 => -- initial State
          if St='1' then
            ACC(8 downto 4) <= "00000"; -- Begin cycle
            ACC(3 downto 0) <= Mplier; -- load the multiplier
            State <= 1;
          end if;
        when 1 | 3 | 5 | 7 => -- 'add/shift' State
          if M = '1' then
            ACC(8 downto 4) <= ACC(7 downto 4),Mcand,'0'); -- Add multiplicand
            State <= State + 1;
          else
            ACC <= '0' & ACC(8 downto 1); --Shift accumulator right
            State <= State + 2;
          end if;
        when 2 | 4 | 6 | 8 => -- 'shift' State
          ACC <= '0' & ACC(8 downto 1); -- Right shift
          State <= State + 1;
        when 9 => -- End of cycle
          State <= 0;
      end case;
    end process;
end behav;
4.3 Design of a Binary Multiplier - Multiplier Control with Counter

- Current design: control part generates the control signals (shift/add) and counts the number of steps.
- If the number of bits is large (e.g., 64), the control network can be divided into a counter and a shift/add control.

Add-shifts control: tests St and M and generates the proper sequence of add and shift signals.
Counter control: counter generates a completion signal K that stops the multiplier after the proper number of shifts have been completed.
4.3 Design of a Binary Multiplier - Multiplier Control with Counter

- Increment counter each time a shift signal is generated
- Generate K after n-1 shifts occurred

**Operation Using a Counter**

<table>
<thead>
<tr>
<th>Time</th>
<th>State</th>
<th>Counter</th>
<th>Product Register</th>
<th>St</th>
<th>M</th>
<th>K</th>
<th>Load</th>
<th>Ad</th>
<th>Sh</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>S0</td>
<td>0</td>
<td>0000000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>t1</td>
<td>S0</td>
<td>0</td>
<td>0000000000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>t2</td>
<td>S1</td>
<td>0</td>
<td>0000001011</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>t3</td>
<td>S2</td>
<td>0</td>
<td>0110110111</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t4</td>
<td>S1</td>
<td>01</td>
<td>0011011011</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>t5</td>
<td>S2</td>
<td>01</td>
<td>1001111111</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t6</td>
<td>S1</td>
<td>10</td>
<td>0100111101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t7</td>
<td>S1</td>
<td>11</td>
<td>0010011111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>t8</td>
<td>S2</td>
<td>11</td>
<td>1000111111</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>t9</td>
<td>S3</td>
<td>00</td>
<td>0100011111</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
4.3 Design of a Binary Multiplier -
Array Multiplier

What do we need to realize
AND gates = ?
Array Multiplier? FA = ?
HA = ?

Electrical and Computer Engineering
4.3 Design of a Binary Multiplier - Array Multiplier (cont’d)

- Complexity of the N-bit array multiplier
  - number of AND gates = ?
  - number of HA = ?
  - number of FA = ?

- Delay
  - $t_g$ — longest AND gate delay
  - $t_{ad}$ — longest possible delay through an adder

4.4 Multiplication of Signed Binary Numbers

- How to multiply signed binary numbers?

- Procedure
  - Complement the multiplier if negative
  - Complement the multiplicand if negative
  - Multiply two positive binary numbers
  - Complement the product if it should be negative

- Simple but requires more hardware and time than other available methods
4.4 Multiplication of Signed Binary Numbers

Four cases

- Multiplicand is positive, multiplier is positive
- Multiplicand is negative, multiplier is positive
- Multiplicand is positive, multiplier is negative
- Multiplier is negative, multiplicand is negative

Examples

- 0111 x 0101 = ?  
  (Preserve the sign of the partial product at each step)
- 1101 x 0101 = ?  
  (If multiplier is negative, complement the multiplicand before adding it in at the last step)
- 0101 x 1101 = ?
- 1011 x 1101 = ?
4.4 Multiplication of Signed Binary Numbers - State Graph

- Move wires from the adder outputs one position to the right => add and shift can occur at the same clock cycle

4.4 Multiplication of Signed Binary Numbers - Faster Multiplier

- Move wires from the adder outputs one position to the right => add and shift can occur at the same clock cycle
4.4 Multiplication of Signed Binary Numbers - New State Graph

4.4 Multiplication of Signed Binary Numbers - New Behavioral Model

```vhdl
library BITLIB;
use BITLIB.bit_pack.all;

entity multi2C is
    port (CLX, S: in bit;
          Mplier,Mcand : in bit_vector(3 downto 0);
          Product: out bit_vector(6 downto 0);
          Done: out bit);
end multi2C;

architecture behavel of multi2C is
    signal State : integer range 0 to 5;
    signal A, B: bit_vector(3 downto 0);
    alias M: bit is B(0);
begin
    process
        variable addout: bit_vector(4 downto 0);
        begin
            wait until CLK = '1';
            case State is
            when 0 =>
                if S='1' then
                    A <= "0000";
                    B <= Mplier;
                    State <= 1;
                end if;
            end case;
        end process;
end behavel;
```

---

**4.4 Multiplication of Signed Binary Numbers - New State Graph**

- States: S0, S1, S2, S3, S4, S5
- Transitions:
  - S0 -> S'0
  - S0 -> S/Ld
  - S1 -> M/AdSh, M'/Sh
  - S2 -> M/AdSh, M'/Sh
  - S3 -> M/AdSh, M'/Sh
  - S4 -> M/AdSh, M'/Sh
  - S5 -> M/Cm, AdSh

---

**4.4 Multiplication of Signed Binary Numbers - New Behavioral Model**

```vhdl```

---

Spring 2004 Slide 25

---

Electrical and Computer Engineering
4.4 Multiplication of Signed Binary Numbers - New Behavioral Model

```plaintext
when l2 | l3 #=> -- *add/shift* State
  if M = '1' then
    addout := add(A, M and '0'); -- Add multiplicand to A and shift
    A <= A(3) & A(3 downto 1); -- Arithmetic right shift
    B <= A(0) & B(3 downto 1);
  else
    A <= A(3) & A(3 downto 1); -- Arithmetic right shift
    B <= A(0) & B(3 downto 1);
  end if;
  State <= State + 1;
when 4 #=> -- add complement if sign bit
  if M = '1' then
    addout := add(A, not M and '1');
    A <= not M and A(3) & addout(3 downto 1);
    B <= addout(0) & B(3 downto 1);
  else
    A <= A(3) & A(3 downto 1); -- Arithmetic right shift
    B <= A(0) & B(3 downto 1);
  end if;
  State <= State + 1;
when 5 #=> -- output product
  State <= 0;
  Done <= '0';
end case;
end process;
```

4.4 Multiplication of Signed Binary Numbers - Simulation

```plaintext
-- command file to test signed multiplier
list CLK St State A B Done Product
force st 1 2 0 22
force clk 1 0 0 10 - repeat 20
-- (5/8 * -3/8)
force M and 0101
define multiplier 1101
run l20
```

<table>
<thead>
<tr>
<th>ns</th>
<th>delta</th>
<th>CLK</th>
<th>St</th>
<th>State</th>
<th>A</th>
<th>B</th>
<th>Done</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>+1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0000</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>10</td>
<td>+0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0000</td>
<td>0000</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>20</td>
<td>+1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0000</td>
<td>1101</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>30</td>
<td>+0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0000</td>
<td>1101</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>40</td>
<td>+1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0010</td>
<td>1110</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>50</td>
<td>+0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0010</td>
<td>1110</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>60</td>
<td>+1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0011</td>
<td>0111</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>70</td>
<td>+0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0011</td>
<td>0111</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>80</td>
<td>+1</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0011</td>
<td>0011</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>90</td>
<td>+0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0011</td>
<td>0011</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>100</td>
<td>+2</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>1111</td>
<td>0001</td>
<td>1</td>
<td>1110001</td>
</tr>
<tr>
<td>110</td>
<td>+0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>1111</td>
<td>0001</td>
<td>1</td>
<td>1110001</td>
</tr>
<tr>
<td>120</td>
<td>+1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1111</td>
<td>0001</td>
<td>0</td>
<td>1110001</td>
</tr>
</tbody>
</table>
| ```

Electrical and Computer Engineering
library BITLIB;

use BITLIB.bit_pack_all;

entity testmult is
  end entity;

architecture testt of testmult is
  component mult2c
    port(CLK, St: in bit;
         Mplier, Mcand: in bit_vector(3 downto 0);
         Product: out bit_vector(6 downto 0);
         Done: out bit);
  end component;

constant H: integer := 11; type arr is array(1 to H) of bit_vector(3 downto 0);
constant Mcheckarr: arr := ('0111', '1011', '0101', '1101', '0011', '0100', '0000',
                           '1111', '1011', '0101');
constant Mplierarr: arr := ('0101', '0101', '1101', '1101', '0111', '0111', '1000',
                           '1000', '1111', '1011', '0100');

signal CLK, St, Done: bit; signal Mplier, Mcand: bit_vector(3 downto 0);
signal Product: bit_vector(6 downto 0);

begin
  CLK <= not CLK after 10 ns;

  process
    begin
      for i in 1 to H loop
        Mplier <= Mplierarr(i); Mcand <= Mcheckarr(i); St <= '1';
        wait until rising_edge(CLK); St <= '0'; wait until falling_edge(Done);
      end loop;

  end process;

  mult: mult2c port map(CLk, St, Mplier, Mcand, Product, Done);

end testt;