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)

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

Outline

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

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

Pseudocode

```vhdl
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;
}
```

VHDL code

```vhdl
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;
```
• Block diagram

<table>
<thead>
<tr>
<th>2. Overview of FSMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic RT operation</td>
</tr>
</tbody>
</table>
• Basic form:  
  \[ r_{\text{dest}} \leftarrow f(r_{\text{src1}}, r_{\text{src2}}, \ldots, r_{\text{srcN}}) \]
• Interpretation:  
  – At the rising edge of the clock, the output of registers \( r_{\text{src1}}, r_{\text{src2}}, \ldots \) 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 \( r_{\text{dest}} \)

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

• 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)
• E.g.,
  \[ r \leftarrow 1 \]
  \[ r \leftarrow r \]
  \[ r_0 \leftarrow r_1 \]
  \[ n \leftarrow n - 1 \]
  \[ y \leftarrow a \oplus b \oplus c \oplus d \]
  \[ s \leftarrow a^2 + b^2 \]

• Implementation example
  \[ r_1 \leftarrow r_1 + r_2 \]

• Multiple RT operations
  \[ r_1 \leftarrow 1; \]
  \[ r_1 \leftarrow r_1 + r_2; \]
  \[ r_1 \leftarrow r_1 + 1; \]
  \[ r_1 \leftarrow r_1. \]

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

• Note: new value of \( r_1 \) is only available when the FSM exits \( s_1 \) state
3. FSMD design example: Repetitive addition multiplier

- Basic algorithm: \(7 \times 5 = 7 + 7 + 7 + 7 + 7\)

- Pseudo code

```plaintext
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

```plaintext
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 add states)
    - \(n \leftarrow n - 1\) (in the op state)
  - RT operations with the `b` register:
    - \(b \leftarrow b\) (in the idle state and op states)
    - \(b \leftarrow b_in\) (in the load and add 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;

begin
  state_reg <= a_in_0 & b_in_0 & count_1;
  state_reg <= next_state;
end case;

--- 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;

--- control path: output logic
process(state_reg, start, a_in_0, b_in_0, ready, r)
begin
  if start='1' then
    when idle =>
      if a_in_0='1' then
        state_next <= abo;
      else
        state_next <= idle;
      end if;
    when abo =>
      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 if;
end case;

--- control path: next-state output logic
process(state_reg, start, a_in_0, b_in_0, count_1)
begin
  case state_reg is
    when idle =>
      if start='1' then
        if a_in_0='1' or b_in_0='1' then
          state_next <= abo;
        else
          state_next <= idle;
        end if;
      else
        state_next <= idle;
      end if;
    when abo =>
      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 if;
end case;

--- 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 abo =>
      a_next <= unsigned(a_in);
      n_next <= unsigned(n_reg);
      r_next <= unsigned(r_reg);
    when load =>
      a_next <= unsigned(a_in);
      n_next <= unsigned(b_in);
      r_next <= (others=>'0');
    when op =>
      a_next <= unsigned(a_reg);
      n_next <= sub_out;
      r_next <= adder_out;
  end case;
end process;

--- data path: functional unit
adder_out <= "00000000" & a_reg * r_reg;
sub_out <= n_reg - 1;

--- data path: status
a_in_0 <= '1' when a_in="00000000" else '0';
b_in_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);
• 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

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 FSMD 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
• Mealy-controlled operation
  – Control signals are edge-sensitive
  – Mealy output is faster and requires fewer states
  – E.g.

```vhdl
-- data path input routing and functional units
process(state_reg, r_reg, a_reg, n_reg)
begin
  if (state_reg=op1) then
    adder_erc1 <= r_reg;
    adder_erc2 <= "00000000" & a_reg;
  else -- for op2 state
    adder_erc1 <= "00000000" & n_reg;
    adder_erc2 <= (others=>'1');
  end if;
end process;
adder_out <= adder_erc1 + adder_erc2;
```

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;
```
• 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

5. Clock rate and Performance of FSMD
  • Maximal clock rate
    – More difficult to analyze because of two interactive loops
    – The boundary of the clock rate can be found

```
case state_reg is
  when idle =>
    if state='1' then
      a_next <= unsigned(a_im);
      m_next <= unsigned(b_im);
      r_next <= (others=>'0');
      if a_im='00000000' or b_im='00000000' then
        state_next <= idle;
      else
        state_next <= op;
        end if;
      else
        state_next <= idle;
        end if;
    end if;
  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 case;
```
• Best-case scenario:
  – Control signals needed at early stage
  – Status signal available at late stage

\[ T_{eq} + T_{dp} + T_{setup} \leq T_c \leq T_{eq} + T_{output} + T_{dp} + T_{next} + T_{setup} \]

\[ \frac{1}{T_{eq} + T_{output} + T_{dp} + T_{next} + T_{setup}} \leq f \leq \frac{1}{T_{eq} + T_{dp} + T_{setup}} \]

• Performance of FSMD
  – \( T_c \): Clock period
  – \( K \): # clock cycles to compete the computation
  – Total time = \( K \times T_c \)
  – \( K \) determined by algorithm, input patterns etc.

• 8-bit input
  – Best: \( b=0, K=2 \)
  – Worst: \( b=255, K=257 \)

• N-bit input:
  – Worst:
    \( K = 2^{(2^n-1)} \)

• 8-bit input
  – Best: \( b=0, K=2 \)
  – Worst: \( b=255, K=2 + 255 \times 2 \)

• N-bit input:
  – Worst:
    \( K=2+2^{(2^n-1)} \)
6. Sequential add-and-shift multiplier

| \(a_3\) | \(a_2\) | \(a_1\) | \(a_0\) | \(b_3\) | \(b_2\) | \(b_1\) | \(b_0\) | \(a_{3\prime}\) | \(a_{2\prime}\) | \(a_{1\prime}\) | \(a_{0\prime}\) | \(b_{3\prime}\) | \(b_{2\prime}\) | \(b_{1\prime}\) | \(b_{0\prime}\) | \(y_7\) | \(y_6\) | \(y_5\) | \(y_4\) | \(y_3\) | \(y_2\) | \(y_1\) | \(y_0\) |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
|       |       |       |       |       | \(a_{3\prime}\) | \(a_{2\prime}\) | \(a_{1\prime}\) | \(a_{0\prime}\) | \(b_{3\prime}\) | \(b_{2\prime}\) | \(b_{1\prime}\) | \(b_{0\prime}\) |
| \(+=\) |       |       |       |       | \(a_{3\prime}b_0\) | \(a_{2\prime}b_0\) | \(a_{1\prime}b_0\) | \(a_{0\prime}b_0\) | \(a_{3\prime}b_1\) | \(a_{2\prime}b_1\) | \(a_{1\prime}b_1\) | \(a_{0\prime}b_1\) |

1. Multiply the digits of the multiplicand \((b_3, b_2, b_1, \text{and } b_0)\) by the multiplicand \((A)\) one at a time to obtain \(b_3 + A, b_2 + A, b_1 + A\) and \(b_0 + A\). The \(b_i + A\) operations is bitwise and operations of \(b_i\) and the digits of \(A\):

\[
 b_i + A = (a_3, a_2, a_1, a_0, a_{-1}, a_{-2})
\]

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:

\[
 \begin{align*}
 n &= 0; \\
 p &= 0; \\
 \text{while } & (n! = 0) \\
 \text{if } (b_i(n) = 1) & \text{ then} \\
 p &= p + (a_i << n); \\
 \text{return } (p); \\
\end{align*}
\]

\[
\begin{align*}
 a &= a_i; \\
 b &= b_i; \\
 n &= 8; \\
 p &= 0; \\
 \text{while } & (n! = 0) \\
 \text{if } (b_i = 1) & \text{ then} \\
 p &= p + a; \\
 a &= a_i; \\
 b &= b_i; \\
 n &= n_i; \\
 \text{return } (p); \\
\end{align*}
\]

\[
\begin{align*}
 a &= a_i; \\
 b &= b_i; \\
 n &= 8; \\
 p &= 0; \\
 \text{while } & (n! = 0) \\
 \text{if } (b_i = 1) & \text{ then} \\
 p &= p + a; \\
 a &= a_i; \\
 b &= b_i; \\
 n &= n_i; \\
 \text{return } (p); \\
\end{align*}
\]

- Note the use of \(b_{\text{next}}\) and \(n_{\text{next}}\)
- \(a<<1\) and \(b>>1\) require no logic
- 8-bit input
  - Best: \(b_0=0\)
  - \(K = 1 + 8\)
  - Worst: \(b_0=255\)
  - \(K = 1 + 8\times2\)
- \(N\)-bit input:
  - Worst:
    - \(K=2+2^n\)

```vhdl
architecture shift_add_rav_arch of seq_mult is
  constant WIDTH: integer := 8;
  constant C_WIDTH: integer := 8;  — 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^C_WIDTH-1 downto 0);
  signal p_reg, p_next: unsigned(2^C_WIDTH-1 downto 0);
begin
  process(state, b_reg, b.next, a_reg, a.next, p_reg, p.next)
  begin
    case state_reg is
      when idle =>
        if state = '1' then
          b.next <= unsigned(b.lsb);
          a.next <= "00000000" & unsigned(a.lsb);
          p.next <= (others => '0');
          if b.lsb='1' then
            state.next <= add;
          else
            state.next <= shift;
          end if;
        else
          state.next <= idle;
        end if;
      when '1' =>
        when add =>
          p.next <= p.reg * a.reg;
        when shift =>
          state.next <= shift;
    end case;
  end process;
end shift_add_rav_arch;
```
• 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