Register Transfer Methodology II

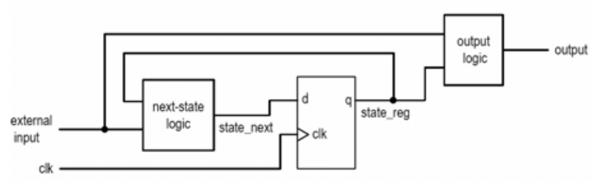
Outline

1. Design example: One–shot pulse generator
2. Design Example: GCD
3. Design Example: UART
4. Design Example: SRAM Interface Controller
5. Square root approximation circuit

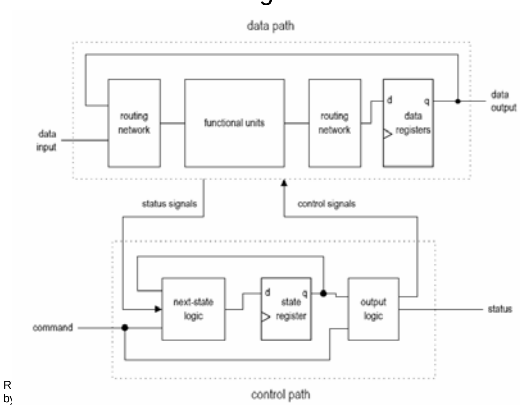
1. One–shot pulse generator

- Sequential circuit divided into
  - Regular sequential circuit: w/ regular next-state logic
  - FSM: w/ random next-state logic
  - FSMD: w/ both
- Division for code development; no formal definition;
- Some design can be coded in different types
- FSMD is most flexible
- One–shot pulse generator as an example

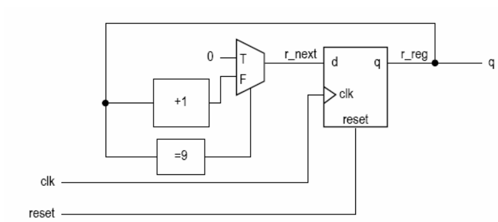
- Basic block diagram



- Refined block diagram of FSMD

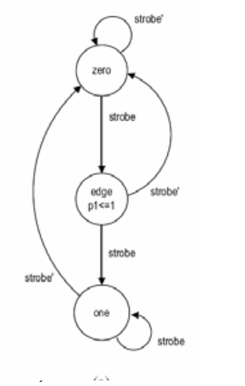


- Regular sequential circuit. E.g., mod-10 counter



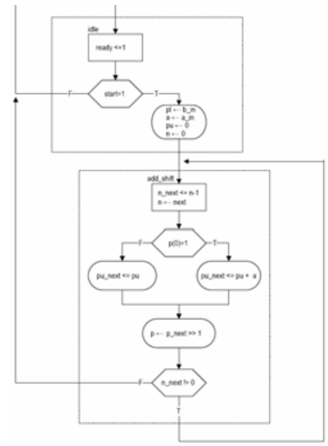

```
-- next-state logic
r_next <= (others=>'0') when r_reg=(TEN-1) else
          r_reg + 1;
```

- FSM. E.g., edge-detection circuit



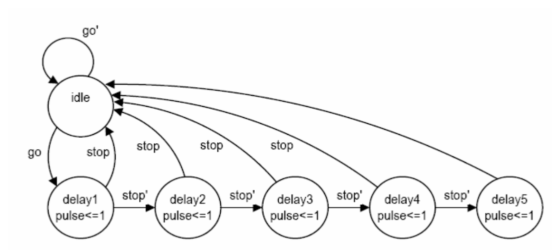

```
-- next-state logic
process(state_reg, strobe)
begin
   case state_reg is
      when zero=>
         if strobe= '1' then
            state_next <= edge;
         else
            state_next <= zero;
         end if;
      when edge =>
         if strobe= '1' then
            state_next <= one;
         else
            state_next <= zero;
         end if;
      when one =>
         if strobe= '1' then
            state_next <= one;
         else
            state_next <= zero;
         end if;
   end case;
end process;
```


- FSMD.
  E.g., multiplier



- One-shot pulse generator
  - I/O: Input: go, `stop;` Output: `pulse`
  - go is the trigger signal, usually asserted for only one clock cycle
  - During normal operation, assertion of go activates `pulse` for 5 clock cycles
  - If go is asserted again during this interval, it will be ignored
  - If `stop` is asserted during this interval, pulse will be cut short and return to 0

- FSM implementation




```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pulse_5clk is
   port(
      clk, reset: in std_logic;
      go, stop: in std_logic;
      pulse: out std_logic
   );
end pulse_5clk;

architecture fsm_arch of pulse_5clk is
   type fsm_state_type is
      (idle, delay1, delay2, delay3, delay4, delay5);
   signal state_reg, state_next: fsm_state_type;
begin
   -- 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;
```



```
-- next-state logic & output logic
process(state_reg,go,stop)
begin
   pulse <= '0';
   case state_reg is
      when idle =>
         if go='1' then
            state_next <= delay1;
         else
            state_next <= idle;
         end if;
      when delay1 =>
         if stop='1' then
            state_next <=idle;
         else
            state_next <=delay2;
         end if;
         pulse <= '1';
      when delay2 =>
         if stop='1' then
            state_next <=idle;
         else
```

```
            state_next <=delay3;
         end if;
         pulse <= '1';
      when delay3 =>
         if stop='1' then
            state_next <=idle;
         else
            state_next <=delay4;
         end if;
         pulse <= '1';
      when delay4 =>
         if stop='1' then
            state_next <=idle;
         else
            state_next <=delay5;
         end if;
         pulse <= '1';
      when delay5 =>
         state_next <=idle;
         pulse <= '1';
   end case;
 end process;
end fsm_arch;
```

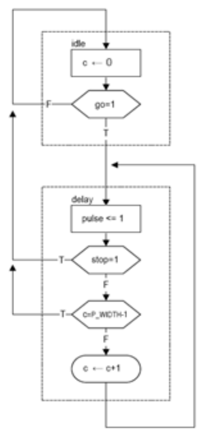

- Regular sequential circuit implementation
  - Based on a mod-5 counter
  - Use a flag FF to indicate whether counter should be active
  - Code difficult to comprehend

```
architecture regular_seq_arch of pulse_5clk is
   constant P_WIDTH: natural:= 5;
   signal c_reg, c_next: unsigned(3 downto 0);
   signal flag_reg, flag_next: std_logic;
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         c_reg <= (others=>'0');
         flag_reg <= '0';
      elsif (clk'event and clk='1') then
         c_reg <= c_next;
         flag_reg <= flag_next;
      end if;
   end process;
```



```
   -- next-state logic
   process(c_reg,flag_reg,go,stop)
   begin
      c_next <= c_reg;
      flag_next <= flag_reg;
      if (flag_reg='0') and (go='1') then
         flag_next <= '1';
         c_next <= (others=>'0');
      elsif (flag_reg='1') and
           ((c_reg=P_WIDTH-1) or (stop='1')) then
         flag_next <= '0';
      elsif (flag_reg='1') then
         c_next <= c_reg + 1;
      end if ;
   end process;
   -- output logic
   pulse <= '1' when flag_reg='1' else '0';
end regular_seq_arch;
```


- FSMD Implementation




```
architecture fsmd_arch of pulse_5clk is
   constant P_WIDTH: natural:= 5;
   type fsmd_state_type is (idle, delay);
   signal state_reg, state_next: fsmd_state_type;
   signal c_reg, c_next: unsigned(3 downto 0);
begin
   -- state and data registers
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
         c_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         c_reg <= c_next;
      end if;
```

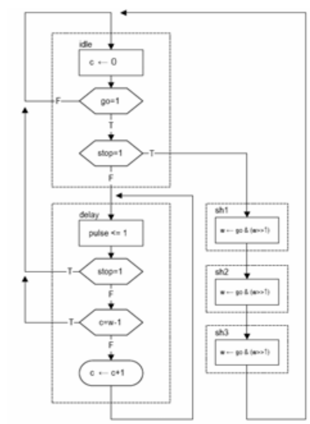


```
   -- next-state logic & data path functional units/routing
   process(state_reg,go,stop,c_reg)
   begin
      pulse <= '0';
      c_next <= c_reg;
      case state_reg is
         when idle =>
            if go='1' then
               state_next <= delay;
            else
               state_next <= idle;
            end if;
            c_next <= (others=>'0');
         when delay =>
            if stop='1' then
               state_next <=idle;
            else
               if (c_reg=P_WIDTH-1) then
                  state_next <=idle;
               else
                  state_next <=delay;
                  c_next <= c_reg + 1;
               end if;
            end if;
            pulse <= '1';
      end case;
   end process;
end fsmd_arch;
```

- Comparison:
  - – FSMD is most flexible and easy to comprehend
- What happens to the following modifications
  - – The delay extend from 5 cycles to 100 ccyles
  - – The `stop` signal is only effective for the first 2 delay cycles and will be ignored otherwise

- “Programmable” one-shot generator
  - – The desired width can be programmed.
  - – The circuit enters the programming mode when both `go` and `stop` are asserted
  - – The desired width shifted in via `go` in the next three clock cycles

- Can be easily extended in ASMD chart
- How about FSM and regular sequential circuit?



2. GCD circuit

- GCD: Greatest Common Divisor
  - – E.g, gcd(1, 10)=1, gcd(12,9)=3
- GCD without division:

$$\gcd(a,b) = \begin{cases} a & \text{if } a = b \\ \gcd(a-b, b) & \text{if } a > b \\ \gcd(a, b-a) & \text{if } a < b \end{cases}$$

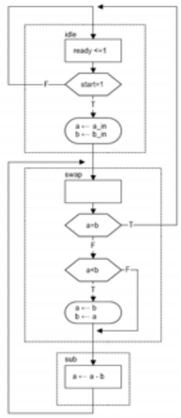
- Pseudo algorithm

```
a = a_in;
b = b_in;
while (a /= b) {
    if (b > a) then
       a = a - b;
    else
       b = b - a;
    end if
}
r = a:
```


- Modified pseudo algorithm w/o while loop

```
       a = a_in;
       b = b_in;
swap:  if (a = b) then
          goto stop;
       else
          if (b > a) then -- swap a and b
             a = b;
             b = a;
          end if;
          a = a - b;
          goto swap;
       end if;
stop:  r = a;
```

- ASMD chart



- VHDL code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity gcd 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(7 downto 0)
   );
end gcd ;

architecture slow_arch of gcd is
   type state_type is (idle, swap, sub);
   signal state_reg, state_next: state_type;
   signal a_reg, a_next, b_reg, b_next: unsigned(7 downto 0);
```



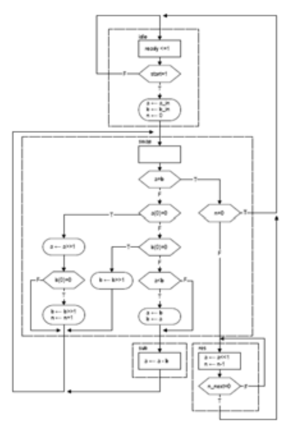
```
-- state & data registers
process(clk,reset)
begin
   if reset='1' then
      state_reg <= idle;
      a_reg <= (others=>'0');
      b_reg <= (others=>'0');
   elsif (clk'event and clk='1') then
      state_reg <= state_next;
      a_reg <= a_next;
      b_reg <= b_next;
   end if;
end process;
```



```
process(state_reg,a_reg,b_reg,start,a_in,b_in)
begin
   a_next <= a_reg;
   b_next <= b_reg;
   case state_reg is
      when idle =>
         if start='1' then
            a_next <= unsigned(a_in);
            b_next <= unsigned(b_in);
            state_next <= swap;
         else
            state_next <= idle;
         end if;
      when swap =>
         if (a_reg=b_reg) then
            state_next <= idle;
         else
            if (a_reg < b_reg) then
               a_next <= b_reg;
               b_next <= a_reg;
            end if;
            state_next <= sub;
         end if;
      when sub =>
         a_next <= a_reg - b_reg;
         state_next <= swap;
   end case;
end process;
```


- What is the problem of this code?
- Another observation

$$\gcd(a,b) = \begin{cases} a & \text{if } a = b \\ 2\gcd(\frac{a}{2}, \frac{b}{2}) & \text{if } a \neq b \text{ and } a, b \text{ even} \\ \gcd(a, \frac{b}{2}) & \text{if } a \neq b \text{ and } a \text{ odd, } b \text{ even} \\ \gcd(\frac{a}{2}, b) & \text{if } a \neq b \text{ and } a \text{ even, } b \text{ odd} \\ \gcd(a-b, b) & \text{if } a > b \text{ and } a, b \text{ odd} \\ \gcd(a, b-a) & \text{if } a < b \text{ and } a, b \text{ odd} \end{cases}$$



- What is the performance now?
- Can we do better with more hardware resources

Square root approximation circuit

- A example of data-oriented (computation-intensive) application
- Equation:

$$\sqrt{a^2 + b^2} \approx \max(((x - 0.125x) + 0.5y), x)$$
$$\text{where } x = \max(|a|, |b|) \text{ and } y = \min(|a|, |b|)$$

- 0.125x and 0.5y corresponds to shift right 3 bits and 1 bit

- Pseudo code:

```
a = a_in;
b = b_in;
t1 = abs(a);
t2 = abs(b);
x = max(t1, t2);
y = min(t1, t2);
t3 = x*0.125;
t4 = y*0.5;
t5 = x - t3;
t6 = t4 + t5;
t7 = max(t6, x)
r = t7;
```


- Direct “data-flow” implementation

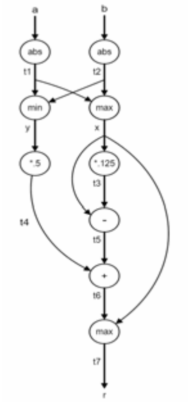
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity sqrt is
   port(
      a_in, b_in: in std_logic_vector(7 downto 0);
      r: out std_logic_vector(8 downto 0)
   );
end sqrt;

architecture comb_arch of sqrt is
   constant WIDTH: natural:=8;
   signal a, b, x, y: signed(WIDTH downto 0);
   signal t1, t2, t3, t4, t5, t6, t7: signed(WIDTH downto 0);
```



```
begin
   a <= signed(a_in(WIDTH-1) & a_in); -- signed extension
   b <= signed(b_in(WIDTH-1) & b_in);
   t1 <= a when a > 0 else
         0 - a;
   t2 <= b when b > 0 else
         0 - b;
   x <= t1 when t1 - t2 > 0 else
        t2;
   y <= t2 when t1 - t2 > 0 else
        t1;
   t3 <= "000" & x(WIDTH downto 3);
   t4 <= "0" & y(WIDTH downto 1);
   t5 <= x - t3;
   t6 <= t4 + t5;
   t7 <= t6 when t6 - x > 0 else
         x;
   r <= std_logic_vector(t7);
end comb_arch;
```

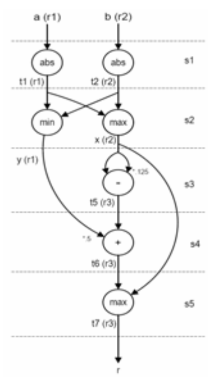

- Requires one adder and six subtractors
- Code contains only concurrent signal assignment statements
- The order is not important.
- Sequence of execution is embedded in the flow of data

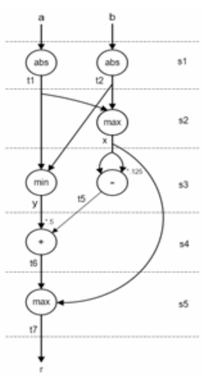
- Data flow graph
  - Shows data dependency
  - Node (circle): an operation
  - Arches: input and output variables
- Note that there is limited degree of parallelism
  - At most two operations can be perform simultaneously



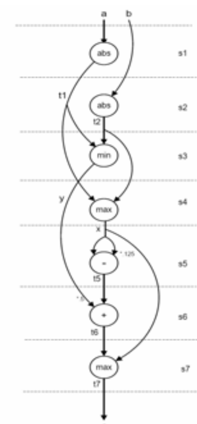
- RT methodology can be used to share the operator
- Tasks in converting a dataflow graph to an ASMD chart
  - Scheduling: *when* a function (circle) can start execution
  - Binding: *which* functional unit is assigned to perform the operation
- In square root algorithm,
  - all operations can be performed by a modified addition unit
  - No function unit is needed for shifting

- Scheduling with two functional units

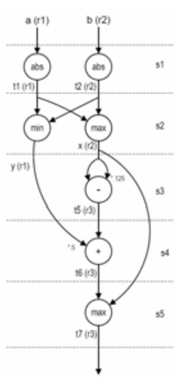


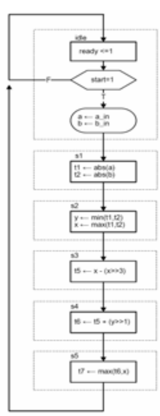


- Scheduling with one functional unit

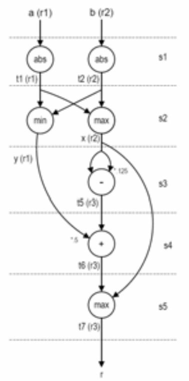


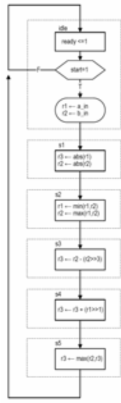
- ASMD chart





- Registers can be shared as well
  - reduce the number of unique variables
  - A variable can be reused if its value is no longer needed
- E.g.,
  - Use `r1` to replace `a`, `t1` and `y`.
  - Use `r2` to replace `b`, `t2` and `x`.
  - Use `r3` to replace `t5`, `t6` and `t7`.





- VHDL code
  - Needs to manually code the data path two insure functional units sharing
  - One unit for abs and min
  - One unit for abs, min, - and +
  - Can be implemented by using an adder/subtractor with special input and output routing circuits


```
-- state & data registers
process(clk,reset)
begin
   if reset='1' then
      state_reg <= idle;
      r1_reg <= (others=>'0');
      r2_reg <= (others=>'0');
      r3_reg <= (others=>'0');
   elsif (clk'event and clk='1') then
      state_reg <= state_next;
      r1_reg <= r1_next;
      r2_reg <= r2_next;
      r3_reg <= r3_next;
   end if;
end process;
```



```
case state_reg is
   when idle =>
      if start='1' then
         r1_next <= signed(a_in(WIDTH-1) & a_i
         r2_next <= signed(b_in(WIDTH-1) & b_i
         state_next <= s1;
       else
         state_next <= idle;
      end if;
      ready <='1';
   when s1 =>
      r1_next <= au1_out; -- t1 =|a|
      r2_next <= au2_out; -- t2 =|b|
      state_next <= s2;
   when s2 =>
      r1_next <= au1_out; -- y=min(t1 , t2)
      r2_next <= au2_out; -- x=max(t1 , t2)
      state_next <= s3;
   when s3 =>
      r3_next <= au2_out; -- t5=x-0.125x
      state_next <= s4;
   when s4 =>
      r3_next <= au2_out; -- t6 =0.5y+t5
      state_next <= s5;
   when s5 =>
      r3_next <= au2_out; -- t7=max(t6 , x)
      state_next <= idle;
end case;
```



```
-- arithmetic unit 1
-- subtractor
diff <= sub_op0 - sub_op1;
-- input routing
process(state_reg,r1_reg,r2_reg)
begin
   case state_reg is
      when s1 => -- 0-a
           sub_op0 <= (others=>'0');
           sub_op1 <= r1_reg; -- a
       when others => -- s2: t2-t1
           sub_op0 <= r2_reg; -- t2
           sub_op1 <= r1_reg; -- t1
    end case;
 end process;
```



```
-- output routing
process(state_reg,r1_reg,r2_reg,diff)
begin
   case state_reg is
      when s1 => --|a|
         if diff(WIDTH)='0' then  -- (0-a)>0
            au1_out <= diff; -- -a
         else
            au1_out <= r1_reg; -- a
         end if;
      when others => -- s2: min(a,b)
         if diff(WIDTH)='0' then --(t2-t1)>0
            au1_out <= r1_reg; -- t1
         else
            au1_out <= r2_reg; -- t2
         end if;
   end case;
end process;
```

High-level synthesis

- Convert a “dataflow code” into ASMD based code (RTL code).
  - RTL code can be optimized for performance (min # clock cycles), area (min # functional units) etc.
  - Perform scheduling, binding
  - Minimize # registers and muxes
- Mainly for computation intensive applications (e.g., DSP)