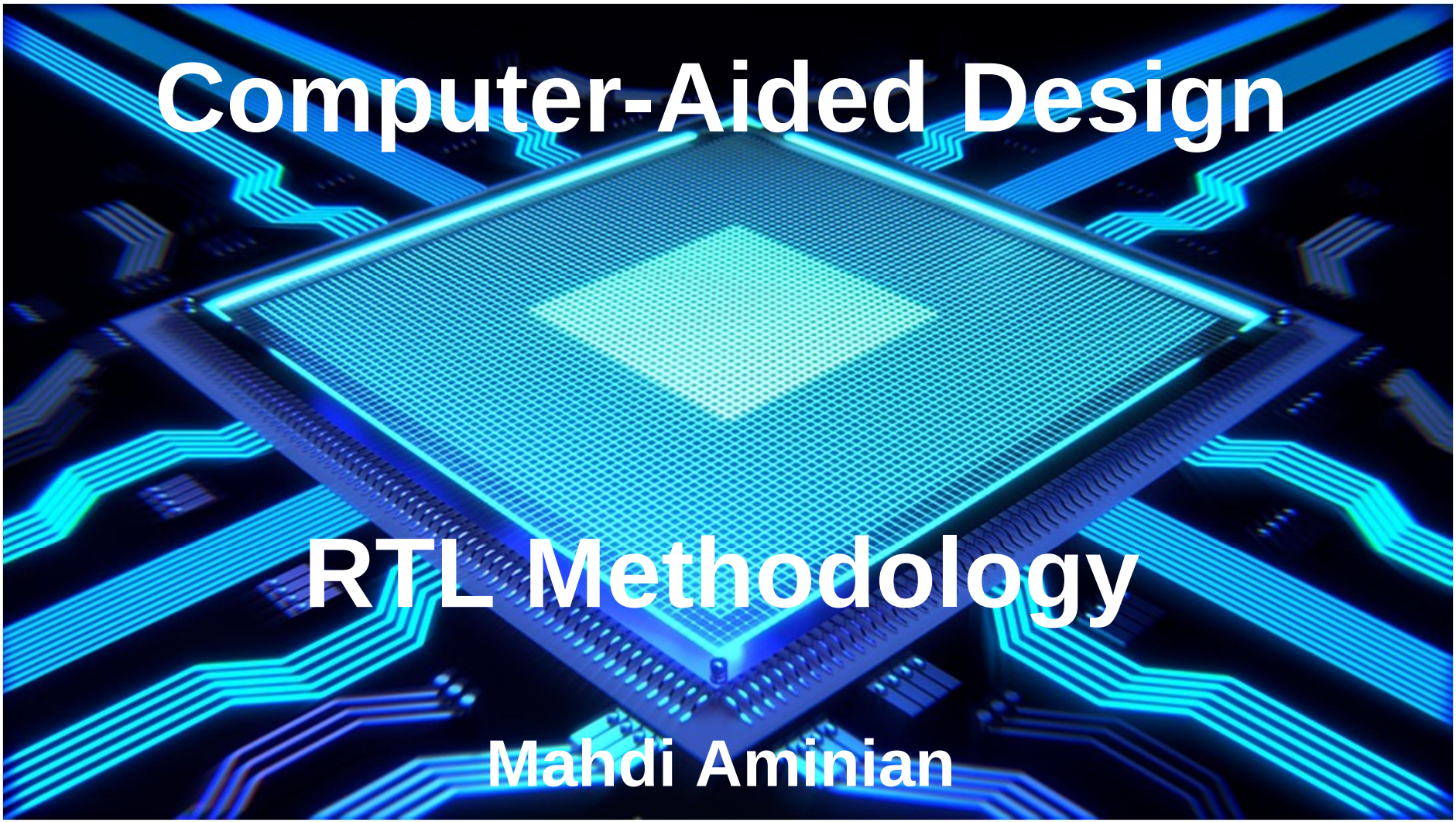


Computer-Aided Design

RTL Methodology

Mahdi Aminian

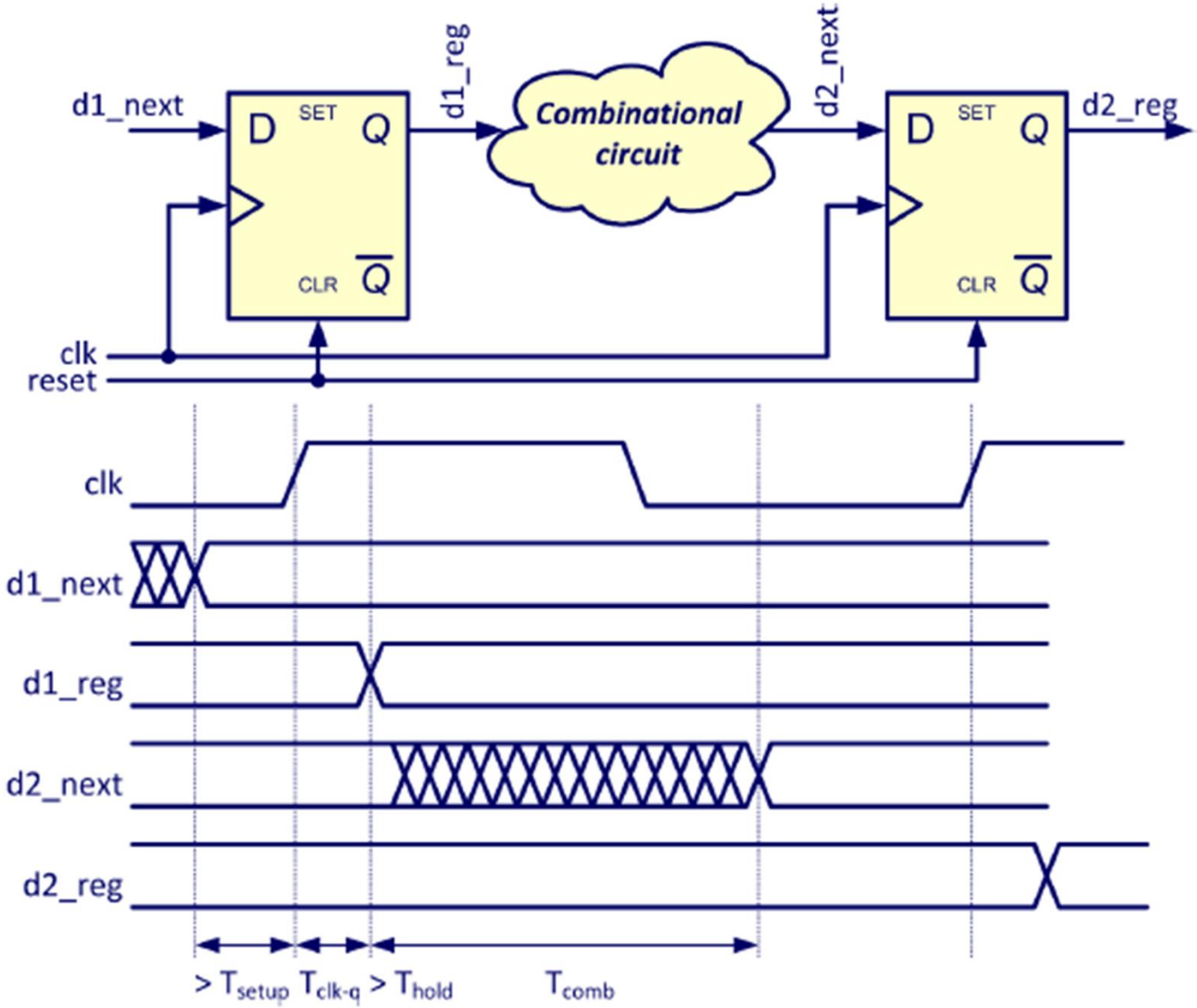
The University Of Guilan

Rasht - Iran

Some slides courtesy of:

- “Hardware Systems Modeling”, A. Vachoux, EPFL
- CAD slides from Dr. Saheb Zamani

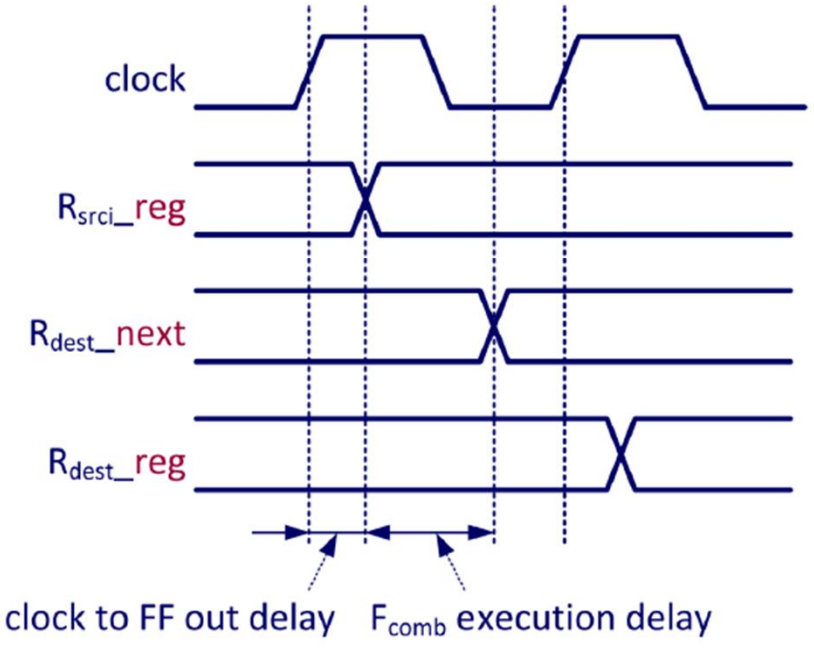
Synchronous Design



Register Transfer Methodology

- **Key elements:**
  - Registers: store data and represent variables of an algorithm
  - Control unit: controls register operations
  - Datapath unit: realizes register operations
- **Basic RT operation:**

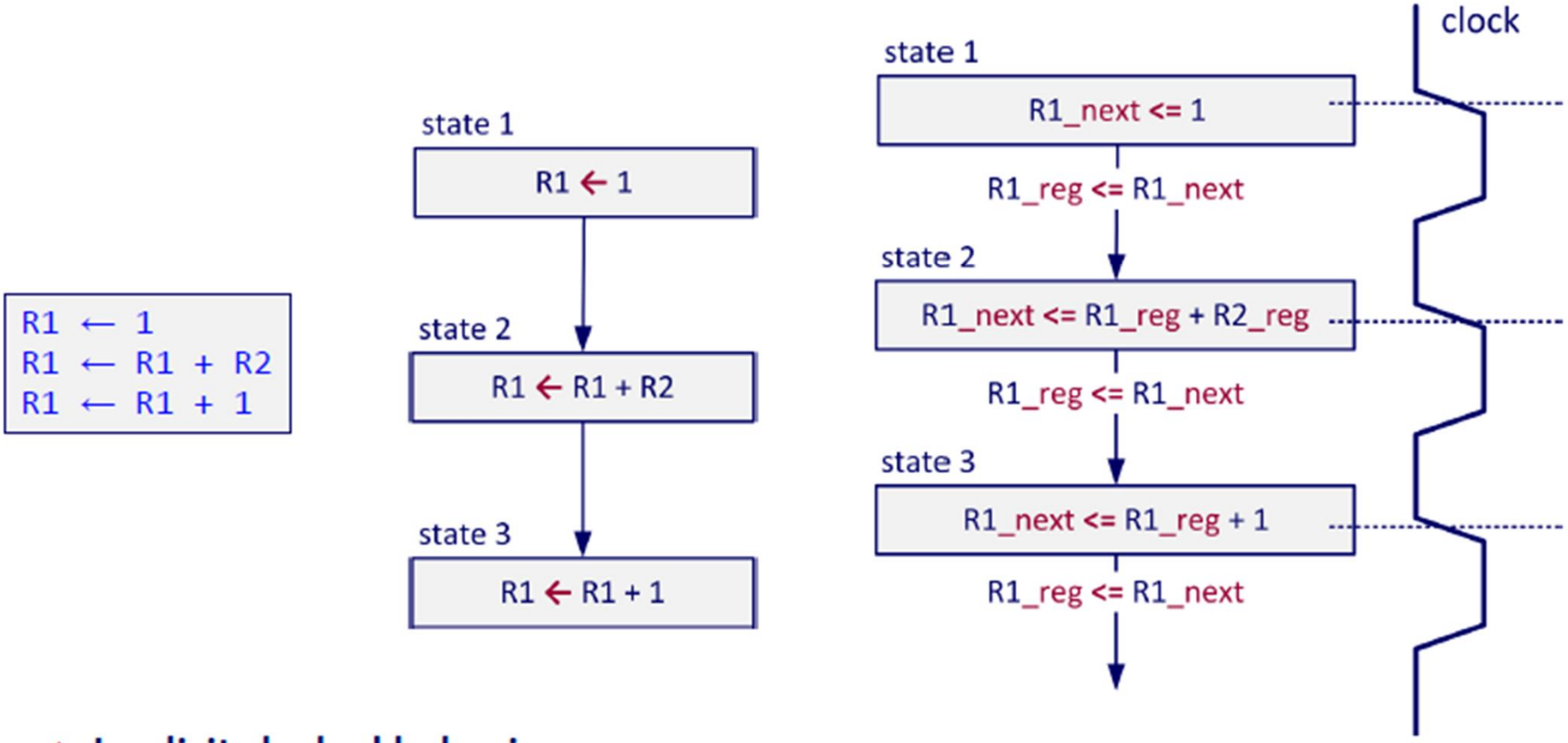
$R_{dest} \leftarrow F_{comb}(R_{src1}, R_{src2}, ..., R_{srcn})$



R_{dest} : destination register
R_{srci} : source registers
F_{comb}: operation to be performed
could be arbitrarily complex, but
must be realized as a combinational circuit

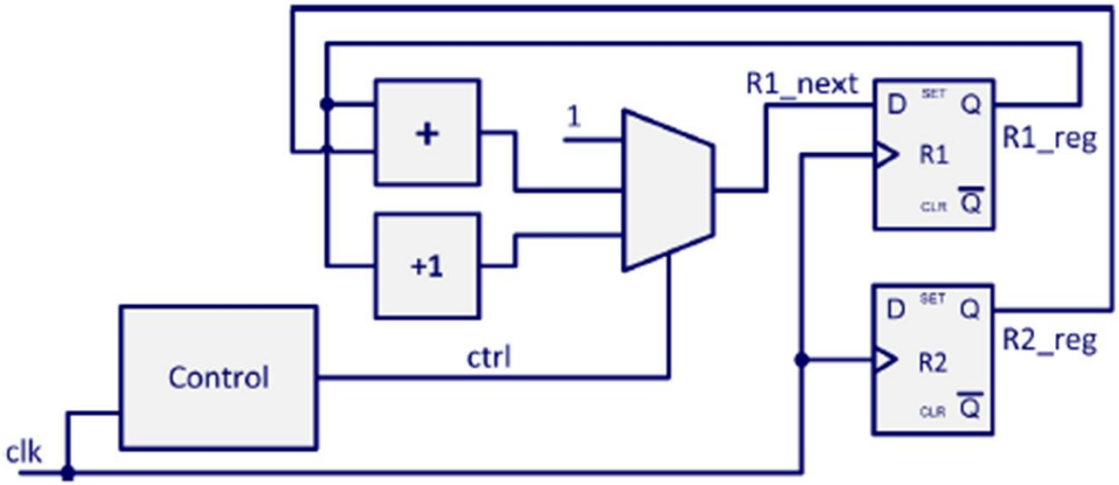
- The ← notation means that the assignment is actually done at the next clock cycle
- The _next signals are FF data inputs
- The _reg signals are FF data outputs

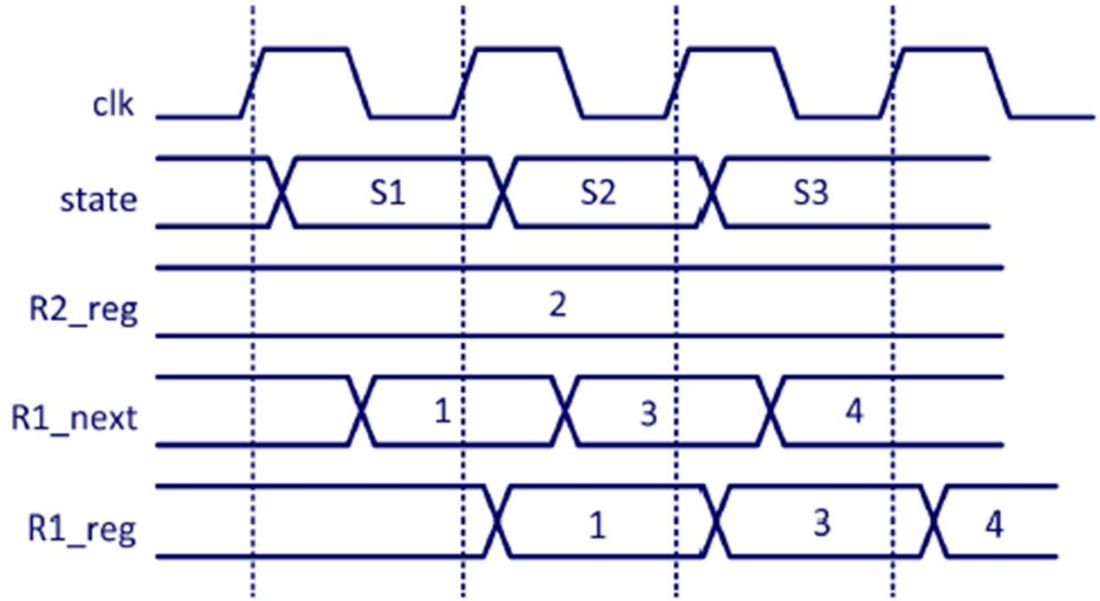
Abstract RT Graph



- **Implicit clocked behavior**

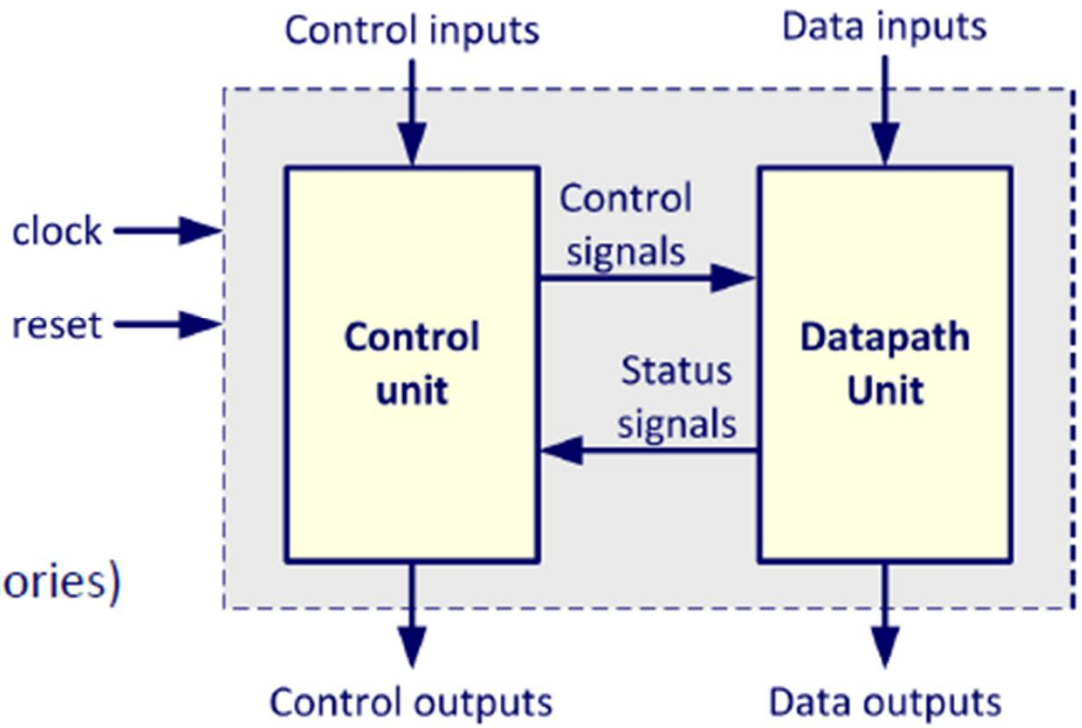
RT Block & Timing Diagram



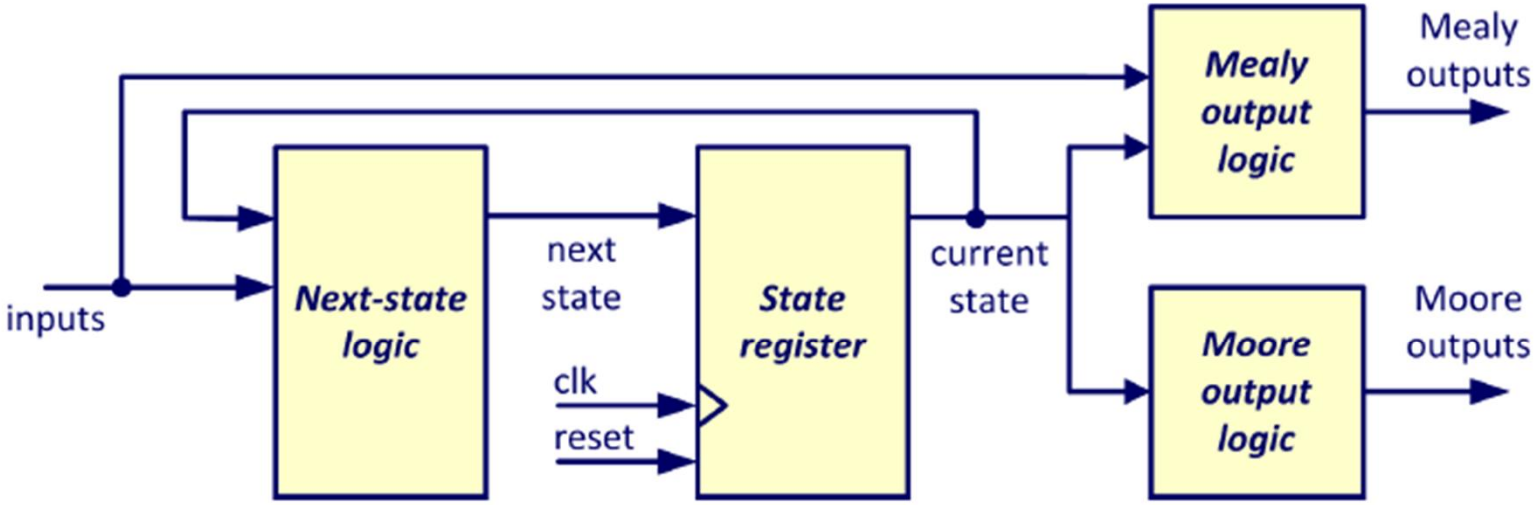


Generic RTL Architecture

- **RTL = Register Transfer Level**
- **Control unit**
  - Finite State Machine (FSM)
  - Sequencer
- **Datapath unit**
  - Operative unit
  - Storage components (registers, register files, memories)
  - Combinational components (ALUs, multipliers, shifters, comparators, ...)
- **Synchronous behavior**
  - Actions triggered at rising or falling clock edge

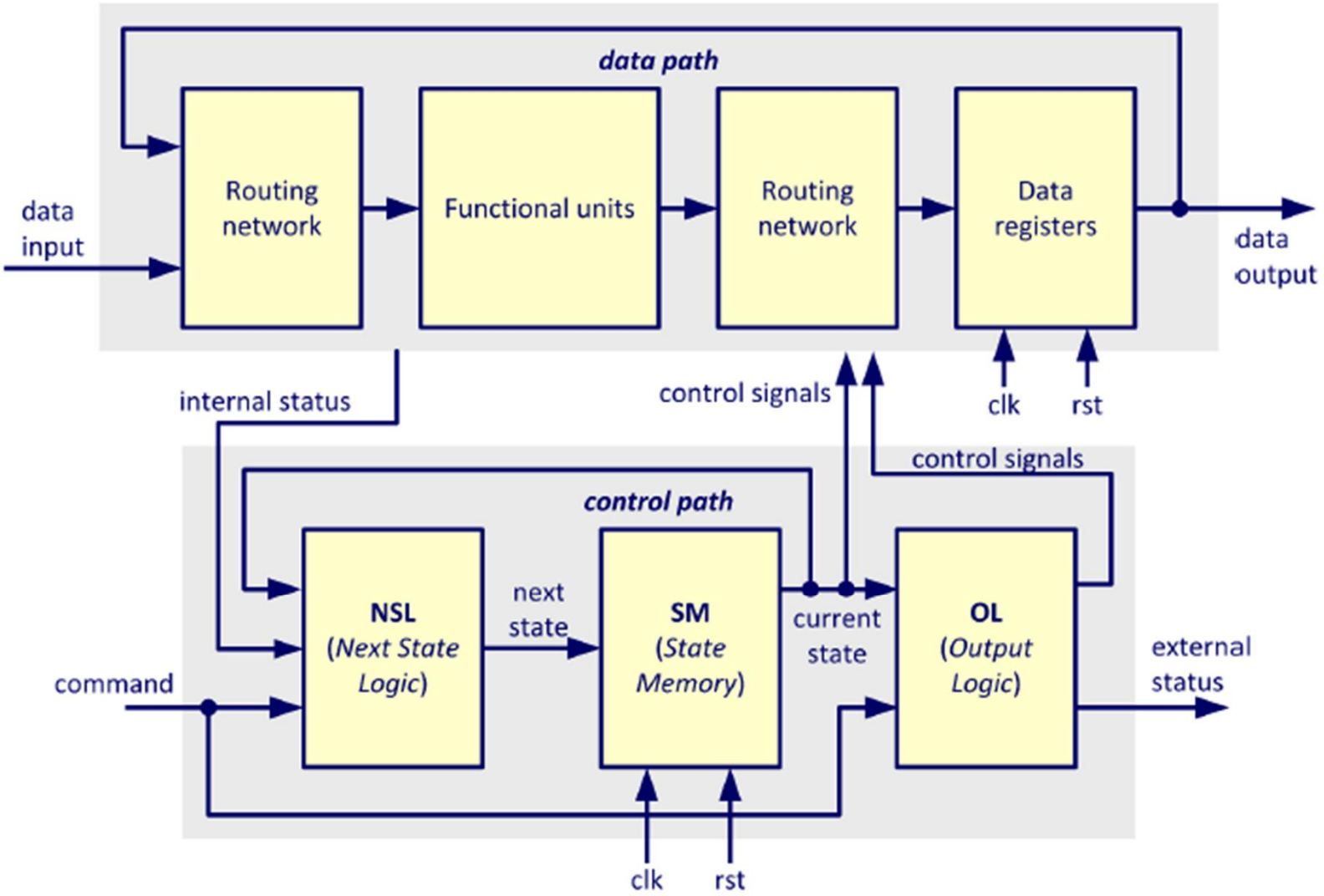


Finite State Machine



- **Next-state logic can be arbitrarily complex random logic**
- **Different kinds of primary outputs**
  - **Moore** outputs are always synchronous to the clock
  - **Mealy** outputs depend asynchronously from primary inputs
  - In general, FSMs may have both Moore and Mealy outputs
- **Somewhat reduced version of the full generic RTL model**

RTL Architecture
FSM + Datapath (FSMD)

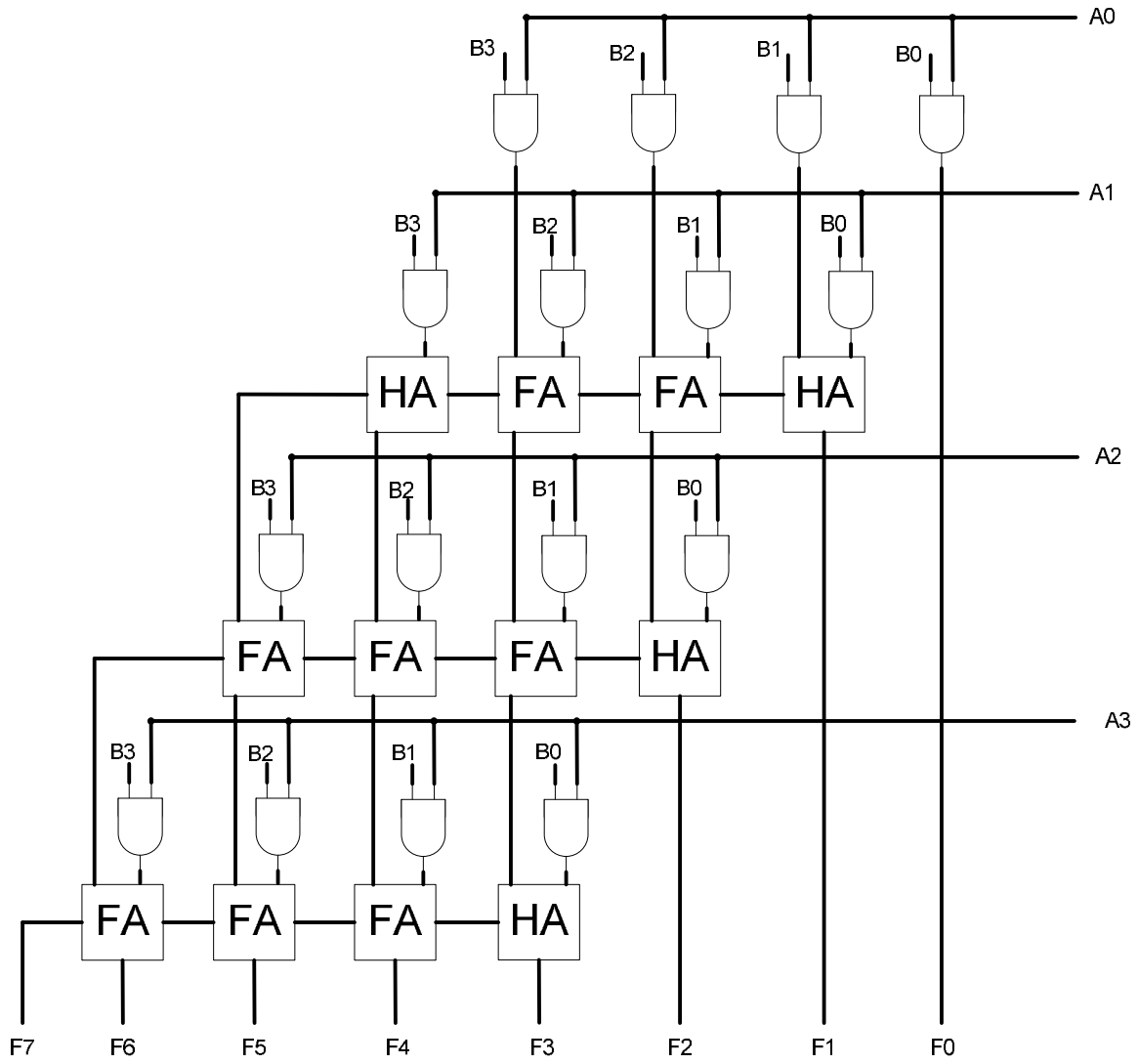


Example: 8-bit Repetitive-Addition Multiplier

- **Standard algorithm for 4-bit operands**

				a_3	a_2	a_1	a_0		multiplicand
			x	b_3	b_2	b_1	b_0		multiplier
				a_3b_0	a_2b_0	a_1b_0	a_0b_0		
			$pp_{0,4}$	$pp_{0,3}$	$pp_{0,2}$	$pp_{0,1}$	$pp_{0,0}$		partial product pp_0
		+	a_3b_1	a_2b_1	a_1b_1	a_0b_1			
		$pp_{1,4}$	$pp_{1,3}$	$pp_{1,2}$	$pp_{1,1}$	$pp_{1,0}$			partial product pp_1
	+	a_3b_2	a_2b_2	a_1b_2	a_0b_2				
	$pp_{2,4}$	$pp_{2,3}$	$pp_{2,2}$	$pp_{2,1}$	$pp_{2,0}$				partial product pp_2
+	a_3b_3	a_2b_3	a_1b_3	a_0b_3					
$pp_{3,4}$	$pp_{3,3}$	$pp_{3,2}$	$pp_{3,1}$	$pp_{3,0}$					partial product pp_3
$pp_{3,4}$	$pp_{3,3}$	$pp_{3,2}$	$pp_{3,1}$	$pp_{3,0}$	$pp_{2,0}$	$pp_{1,0}$	$pp_{0,0}$		product

Example: 8-bit Repetitive-Addition Multiplier



Combination Implementation

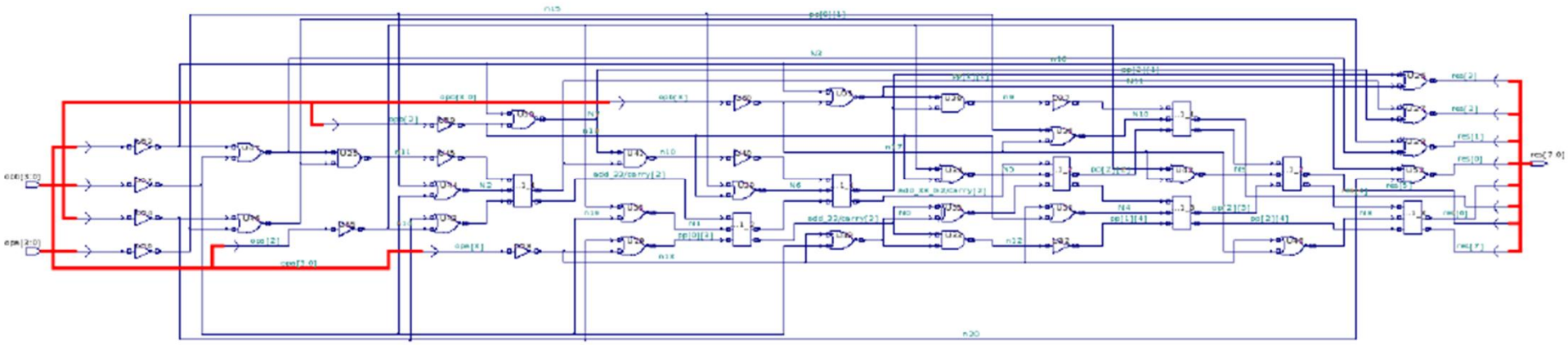
```
entity multn is
   generic (NBITS : natural := 8);
   port (
      signal opa, opb : in  std_logic_vector(NBITS-1 downto 0);
      signal prod     : out std_logic_vector(2*NBITS-1 downto 0));
end entity multn;
```

```
architecture comb2 of multn is
   use ieee.numeric_std.all;

   type opb_vector is array (0 to NBITS-1) of unsigned(NBITS-1 downto 0);
   type pp_vector is array (0 to NBITS-1) of unsigned(NBITS downto 0);

   signal opau  : unsigned(NBITS-1 downto 0);    -- unsigned version of opa
   signal produ : unsigned(2*NBITS-1 downto 0);  -- unsigned version of product

   signal bv : opb_vector;  -- bv(i) = {opb(i), opb(i), ..., opb(i)}
   signal pp : pp_vector;   -- partial products
   ...
```

Combination Implementation

```
begin
   opau <= unsigned(opa);

   -- build vectors bv(i) to facilitate later bitwise operations
   BV_GEN : for i in 0 to NBITS-1 generate
      bv(i) <= (others => opb(i));
   end generate BV_GEN;

   -- compute partial products (NBITS-1 (N+1)-bit additions)
   pp(0) <= "0" & (bv(0) and opau);
   PP_GEN : for i in 1 to NBITS-1 generate
       pp(i) <= ("0" & pp(i-1)(NBITS downto 1)) + ("0" & (bv(i) and opau));
   end generate PP_GEN;

   -- assemble product bits
   produ(2*NBITS-1 downto NBITS-1) <= pp(NBITS-1);  -- NBITS+1 most significant bits
   PROD_GEN : for i in NBITS-2 downto 0 generate    -- other bits
      produ(i) <= pp(i)(0);
   end generate PROD_GEN;

   prod <= std_logic_vector(produ);
end architecture comb2;
```

Comb. 4-bit Multiplier



FSMD Implementation

- **Use only one adder and perform additions sequentially**
- **Algorithm**

```
-- pseudo-code for z_out = a_in*b_in
if (a_in = 0 or b_in = 0) z_out = 0;
else {
   a = a_in; b = b_in; z = 0;
   while (b != 0) {
      z = z + a; b = b - 1;
   }
}
z_out = z;
```

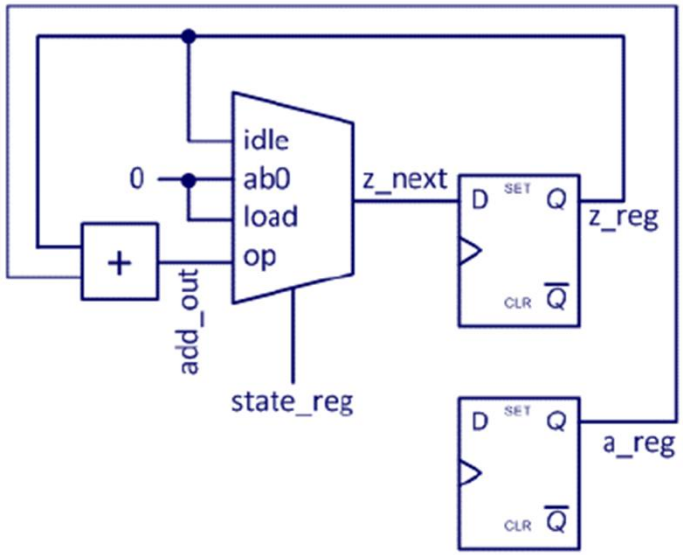
- **Define external signals (names and types)**
  - Asserted when '1' ('0' for signal names with _b suffix)
  - Data inputs: **a_in**, **b_in** operands [unsigned NBITS integers]
  - Control inputs: **clk**, **rst_b**, **start** clock, reset, start command [1-bit logic]
  - Data output: **z_out** product [unsigned 2*NBITS integer]
  - Control output: **ready** ready to accept new inputs/ previous operation has completed [1-bit logic]

FSMD Implementation

- **Define data registers**
  - **a_reg, b_reg**
    8-bit unsigned integers
  - **z_reg (acc_reg?)**
    16-bit unsigned integer

↓States Data regs →	a	b	z
idle	a ← a	b ← b	z ← z
ab0	(a ← a_in)	(b ← b_in)	z ← 0
load	a ← a_in	b ← b_in	z ← 0
op	a ← a	b ← b - 1	z ← z + a

- **RT operations grouped in the same state as long as there is no data dependency and enough HW resources**
- **Define data path associated to register a**
  - No specific functional unit needed
- **Define data path associated to register z**
  - Functional unit: adder
  - Output: **add_out** 16-bit unsigned integer
- **Define data path associated to register b**
  - Functional unit: decrementor
  - Output: **sub_out** 8-bit unsigned integer



FSMD Implementation

```
entity multn is
  generic (NBITS : natural := 8);
  port (
    signal clk, rst_b : in  std_logic;  -- clock, reset (asserted low)
    signal start      : in  std_logic;  -- ext. input command
    signal a_in, b_in : in  std_logic_vector(NBITS-1 downto 0);  -- ext. input data
    signal ready      : out std_logic;  -- ext. output status
    signal z_out      : out std_logic_vector(2*NBITS-1 downto 0));  -- ext. output data
end entity multn;
```

```
architecture fsmd of multn is
  constant ZERO : unsigned(NBITS-1 downto 0) := to_unsigned(0, NBITS);

  type state_type is (ST_IDLE, ST_AB0, ST_LOAD, ST_OP);
  constant RST_STATE : state_type := ST_IDLE;
  signal state_reg, state_next : state_type;

  -- data registers
  signal a_reg, a_next : unsigned(NBITS-1 downto 0);
  signal b_reg, b_next : unsigned(NBITS-1 downto 0);
  signal z_reg, z_next : unsigned(2*NBITS downto 0);

  -- internal status signals
  signal opa0, opb0, breg0 : std_logic;

  -- functional unit outputs
  signal add_out : unsigned(2*NBITS-1 downto 0);
  signal sub_out : unsigned(NBITS-1 downto 0);
  …
```

- **Define internal control signals**
  - **state_reg** 2-bit std_logic_vector
- **Define internal status signals**
  - **opa0** 1-bit logic vector, '1' if a_in = 0, '0' otherwise
  - **opb0** 1-bit logic vector, '1' if b_in = 0, '0' otherwise
  - **breg0** 1-bit logic vector, '1' if b_reg = 0, '0' otherwise

FSMD Implementation

```
begin -- architecture fsmd

  -- control part: state register
  CP_SR : process (clk, rst_b)
  begin
    if rst_b = '0' then
      state_reg <= RST_STATE;
    elsif rising_edge(clk) then
      state_reg <= state_next;
    end if;
  end process CP_SR;
  …
```

- **Control part (FSM)**

```
  …
  -- control part: next-state logic
  CP_NSL : process (state_reg, start, opa0, opb0, breg0)
  begin
    state_next <= state_reg;  -- avoid latches
    case state_reg is
      when ST_IDLE => if start = '1' then
                        if opa0 = '1' or opb0 = '1' then
                          state_next <= ST_AB0;
                        else
                          state_next <= ST_LOAD;
                        end if;
                      end if;
      when ST_AB0  => state_next <= ST_IDLE;
      when ST_LOAD => state_next <= ST_OP;
      when ST_OP   => if breg0 = '1' then
                        state_next <= ST_IDLE;
                      end if;
    end case;
  end process CP_NSL;

  -- control part: output logic
  CP_OL : ready <= '1' when state_reg = ST_IDLE else '0';
  …
```

FSMD Implementation

```
…
-- datapath: data registers
DP_DR : process (clk, rst_b)
begin
  if rst_b = '0' then
    a_reg <= (others => '0');
    b_reg <= (others => '0');
    z_reg <= (others => '0');
  elsif rising_edge(clk) then
    a_reg <= a_next;
    b_reg <= b_next;
    z_reg <= z_next;
  end if;
end process DP_DR;

-- datapath: status signals
opa0 <= '1' when a_in = ZERO else
        '0';
opb0 <= '1' when b_in = ZERO else
        '0';
breg0 <= '1' when b_next = ZERO else
         '0';
…
```

```
  …
  -- datapath: routing mux
  DP_RMUX : process (state_reg, a_in, b_in, a_reg,
                     b_reg, z_reg, add_out, sub_out)
  begin
    a_next <= a_reg;  -- avoid latches
    b_next <= b_reg;
    z_next <= z_reg;
    case state_reg is
      when ST_IDLE => null;
      when ST_AB0  => z_next <= (others => '0');
      when ST_LOAD => a_next <= unsigned(a_in);
                      b_next <= unsigned(b_in);
                      z_next <= (others => '0');
      when ST_OP   => z_next <= add_out;
                      b_next <= sub_out;
    end case;
  end process DP_RMUX;

  -- datapath: functional units
  add_out <= (ZERO & a_reg) + z_reg;
  sub_out <= b_reg - 1;

  -- datapath: data output
  z_out <= std_logic_vector(z_reg);
end architecture rtl_fsmd;
```

- **Datapath**