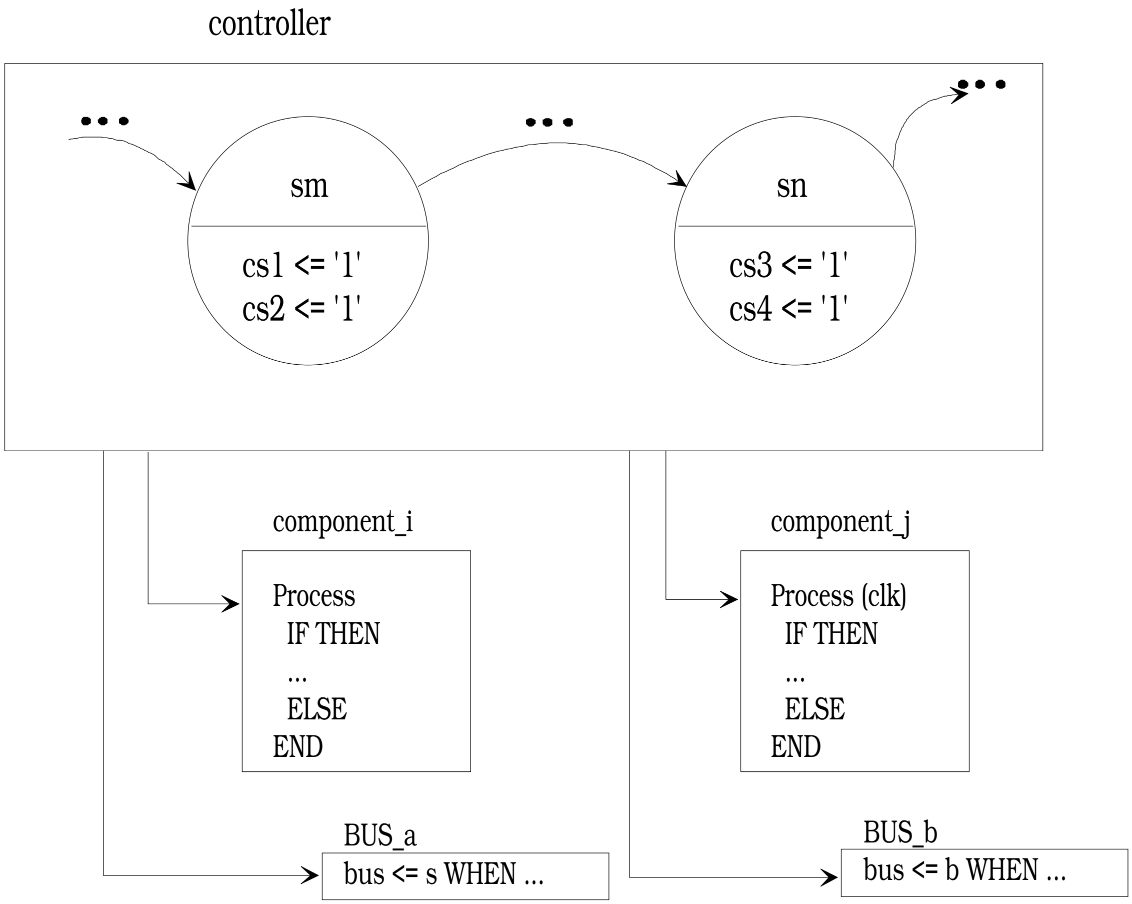
PART 9

Dataflow_Synthesis

- **Synthesis style**
  - **Description style**
  - **Hardware style**
  - **Coding style**
- **Controller**
  - **General Layout**
  - **Individual States**
- **Data components**
- **Data path**
- **Synthesized example**
- **Conclusions**

Dataflow_synthesis.style



- Controller is a state machine
- Control signals control component behavior
- Bus assignments are done by control signals

Dataflow_synthesis.style

Controller

In each state issue control signals. No complex state programs. Specify Bussing.

Data Components

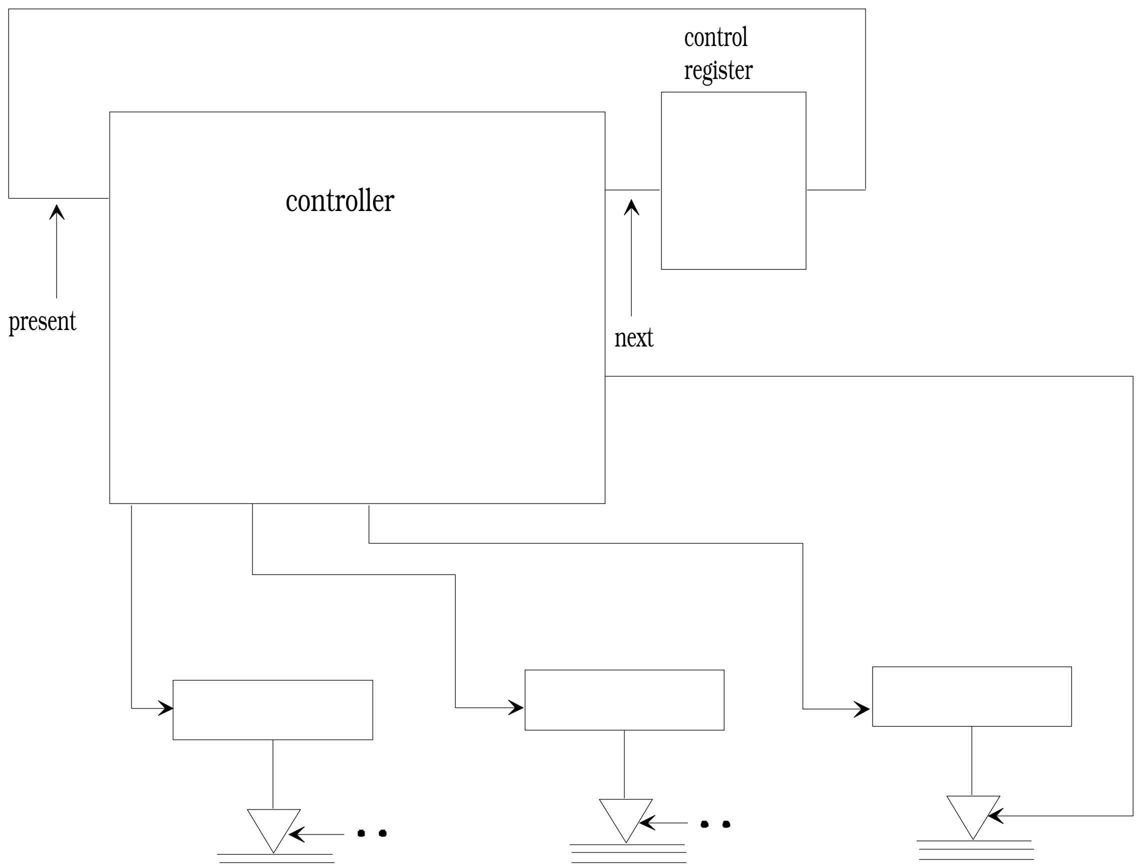
Behavioral Descriptions of components. Clocked and unclocked components.

Bussing

Use conditional signal assignments for bus assignment specification.

- Partition into Data and Control
- Describe Data components and Control unit separately

Dataflow_synthesis.style



- Controller style is one of general state machine
- Data components are described and synthesized separately

Dataflow_synthesis.style

```
ARCHITECTURE synthesizable
    state declarations
BEGIN
    clocking : PROCESS ( clk, ... )
    BEGIN
      next_state TO present_state transfer
    END PROCESS clocking;
    --
    sequencing : PROCESS ( state present values and inputs )
    BEGIN
      reset all control signals
      setting present values with next
      CASE present_state
         WHEN state_i =>
            control signal assignments
         WHEN state_j =>
            ...
      END CASE;
     END PROCESS;
END synthesizable;
```

- Declare next and present states
- Perform clocking to load present with next when clocked
- Reset all control signals in sequencing
- In a case statement, conditionally assign '1' to control signals

Dataflow_synthesis.style.data

Clocked components

```
PROCESS ( clk )
BEGIN
   IF ( edge detection)
         ...
   END IF;
END PROCESS
```

Combinational units

```
PROCESS ( all inputs )
BEGIN
   behavioral description
END PROCESS;
```

Bussing

```
bus2 <= bus1 WHEN bus1on2 ELSE "zzzz";
```

- Use clocked process style for registers including *pc*, *mar*, ...
- Use combinational process style for *alu* and *shu*
- Wire components using *three_state* buses

Dataflow_synthesis.controller

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
--
PACKAGE alu_operations IS
  CONSTANT a_and_b     : std_logic_vector (2 DOWNTO 0) := "000";
  CONSTANT b_compl     : std_logic_vector (2 DOWNTO 0) := "001";
  CONSTANT a_input     : std_logic_vector (2 DOWNTO 0) := "100";
  CONSTANT a_add_b     : std_logic_vector (2 DOWNTO 0) := "101";
  CONSTANT b_input     : std_logic_vector (2 DOWNTO 0) := "110";
  CONSTANT a_sub_b     : std_logic_vector (2 DOWNTO 0) := "111";
END alu_operations;
```

- Use *alu_operations* for readability of code
- Compile *alu_operations* in WORK library

Dataflow_synthesis.controller

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
--
LIBRARY EXEMPLAR;
USE EXEMPLAR.exemplar_1164.ALL;
--
USE WORK.synthesis_parameters.ALL;
USE WORK.synthesis_utilities.ALL;
USE WORK.alu_operations.ALL;
--
ENTITY par_control_unit IS
  PORT (clk : IN std_logic;
        -- register control signals:
        load_ac, zero_ac,
        load_ir,
        increment_pc, load_page_pc, load_offset_pc, reset_pc,
        load_page_mar, load_offset_mar,
        load_sr, cm_carry_sr,
        -- bus connection control signals:
        pc_on_mar_page_bus, ir_on_mar_page_bus,
        pc_on_mar_offset_bus, dbus_on_mar_offset_bus,
        pc_offset_on_dbus, obus_on_dbus, databus_on_dbus,
        mar_on_adbus,
        dbus_on_databus,
        -- logic unit function control outputs:
        arith_shift_left, arith_shift_right : OUT std_logic;
        alu_code : OUT std_logic_vector (2 DOWNTO 0);
        -- inputs from the data section:
        ir_lines : IN byte;
        status : IN nibble;
        -- memory control and other external signals:
        read_mem, write_mem : OUT std_logic; interrupt : IN std_logic
       );
END par_control_unit;
```

- Controller interface includes all signals for data control
- Type *std_logic* is used through out

Dataflow_synthesis.controller

```
ARCHITECTURE dataflow_synthesizable OF par_control_unit IS
  TYPE _states IS (initial, instr_fetch, do_one_bytes, opnd_fetch,
                   do_indirect, do_two_bytes, do_jsr, continue_jsr,
                   do_branch);
  SIGNAL present_state, next_state : cpu_states;
BEGIN
  clocking : PROCESS (clk, interrupt)
  BEGIN
    IF (interrupt = '1') THEN
      present_state <= initial;
    ELSIF clk'EVENT AND clk = '0' THEN
      present_state <= next_state;
    END IF;
  END PROCESS clocking;
  --
```

- Synthesizable dataflow architecture
- Declare nine states
- Clocking process assigns *present_state*

Dataflow_synthesis.controller

```
 sequencing : PROCESS ( present_state, ir_lines, status, interrupt )
  BEGIN
    load_ac <= '0'; zero_ac <= '0'; load_ir <= '0'; increment_pc <= '0';
    load_page_pc <= '0'; load_offset_pc <= '0'; reset_pc <= '0';
    load_page_mar <= '0'; load_offset_mar <= '0';
    load_sr <= '0'; cm_carry_sr <= '0';
    -- bus connection control signals:
    pc_on_mar_page_bus <= '0'; ir_on_mar_page_bus <= '0';
    pc_on_mar_offset_bus <= '0'; dbus_on_mar_offset_bus <= '0';
    pc_offset_on_dbus <= '0'; obus_on_dbus <= '0'; databus_on_dbus <= '0';
    mar_on_adbus <= '0'; dbus_on_databus <= '0';
    -- logic unit function control outputs:
    arith_shift_left <= '0'; arith_shift_right <= '0';
    alu_code  <= "000";
    -- memory control and other external signals:
    read_mem <= '0'; write_mem <= '0';

CASE present_state IS
    WHEN initial =>
         assign signals
     WHEN instr_fetch =>
         ...
   END CASE;
  END PROCESS;
END dataflow_synthesizable;
```

- All control signals are set to '0' at each state
- A case statement handles state transitions

Dataflow_synthesis.controller

```
 WHEN initial =>   -------------------------------------------1
       IF (interrupt = '1') THEN
         reset_pc <= '1';
         next_state <= initial;
       ELSE
         -- pc to mar
         pc_on_mar_page_bus <= '1';
         pc_on_mar_offset_bus <= '1';
         load_page_mar <= '1';
         load_offset_mar <= '1';
         next_state <= instr_fetch;
       END IF;
     WHEN instr_fetch =>   ---------------------------------------2
       -- read memory into ir
       mar_on_adbus <= '1';
       read_mem <= '1';
       databus_on_dbus <= '1';
       alu_code <= a_input;
       load_ir <= '1';
       -- increment pc
       increment_pc <= '1';
       next_state <= do_one_bytes;
```

- Signals issued for initial and *instr_fetch* states

Dataflow_synthesis.controller

```
WHEN do_one_bytes =>   ---------------------------------------3
        pc_on_mar_page_bus <= '1';
        pc_on_mar_offset_bus <= '1';
        load_page_mar <= '1';
        load_offset_mar <= '1';
        IF (ir_lines (7 DOWNTO 4) /= single_byte_instructions) THEN
          next_state <= opnd_fetch;
        ELSE
          CASE ir_lines (3 DOWNTO 0) IS
            WHEN cla =>
              zero_ac <= '1';
              load_ac <= '1';
            WHEN cma =>
              alu_code <= b_compl;
              load_sr <= '1';
              load_ac <= '1';
            WHEN cmc =>
              cm_carry_sr <= '1';
            WHEN asl =>
              alu_code <= b_input;
              arith_shift_left <= '1';
              load_sr <= '1';
              load_ac <= '1';
            WHEN asr =>
              alu_code <= b_input;
              arith_shift_right <= '1';
              load_sr <= '1';
              load_ac <= '1';
            WHEN OTHERS => NULL;
          END CASE;
          next_state <= instr_fetch;
        END IF;
```

- Control signals activated to execute *one_byte* instructions

Dataflow_synthesis.controller

```
 WHEN opnd_fetch =>   -------------------------------------4
       -- read memory into mar offset
       mar_on_adbus <= '1';
       read_mem <= '1';
       databus_on_dbus <= '1';
       dbus_on_mar_offset_bus <= '1';
       load_offset_mar <= '1';
       IF ( ir_lines (7 DOWNTO 6) /= jsr_or_bra ) THEN
         ir_on_mar_page_bus <= '1';
         load_page_mar <= '1';
         IF ( ir_lines (4) = indirect ) THEN
           next_state <= do_indirect;
         ELSE
           next_state <= do_two_bytes;
         END IF;
       ELSE --jsr or bra, do not alter mar page
         IF ( ir_lines (5) = '0' ) THEN  -- jsr
           next_state <= do_jsr;
         ELSE
           next_state <= do_branch;
         END IF;
       END IF;
       increment_pc <= '1';
     WHEN do_indirect =>   -------------------------------------5
       -- read actual operand from memory into mar offset
       mar_on_adbus <= '1';
       read_mem <= '1';
       databus_on_dbus <= '1';
       dbus_on_mar_offset_bus <= '1';
       load_offset_mar <= '1';
       next_state <= do_two_bytes;
```

- Activated control signals to perform operand fetch and indirecting

Dataflow_synthesis.controller

```
WHEN do_two_bytes =>   --------------------------------------6
       IF ( ir_lines (7 DOWNTO 5) = jmp ) THEN
         load_page_pc <= '1';
         load_offset_pc <= '1';
         next_state <= instr_fetch;
       ELSIF ( ir_lines (7 DOWNTO 5) = sta ) THEN
         -- mar on adbus, ac on databus, write to memory
         mar_on_adbus <= '1';
         alu_code<=  b_input;
         obus_on_dbus <= '1';
         dbus_on_databus <= '1';
         write_mem <= '1';
         next_state <= initial;
       ELSIF ( ir_lines (7) = '0' ) THEN              ------ lda, and, add, sub
         -- mar on adbus, read memory for operand, perform operation
         mar_on_adbus <= '1';
         read_mem <= '1';
         databus_on_dbus <= '1';
         IF ( ir_lines (6) = '0' ) THEN                  ---- lda, and
           IF ( ir_lines (5) = '0' ) THEN                -- lda
             alu_code<=  a_input;
           ELSE                                           -- and
             alu_code<=  a_and_b;
           END IF;
         ELSE                                             ---- add, sub
           IF ( ir_lines (5) = '0' ) THEN                -- add
             alu_code<=  a_add_b;
           ELSE                                           -- sub
             alu_code<=  a_sub_b;
           END IF;
         END IF;
         load_sr <= '1';     load_ac <= '1';    next_state <= initial;   ELSE
         next_state <= initial; --never happens
       END IF;
```

- Issue signals to perform two byte instructions

Dataflow_synthesis.controller

```
WHEN do_jsr =>   -------------------------------------------7
        -- write pc offset to top of subroutine
        mar_on_adbus <= '1';
        pc_offset_on_dbus <= '1';
        dbus_on_databus <= '1';
        write_mem <= '1';
        -- address of subroutine to pc
        load_offset_pc <= '1';
        next_state <= continue_jsr;
      WHEN continue_jsr =>   ------------------------------------8
        increment_pc <= '1';
        next_state <= initial;
      WHEN do_branch =>   -----------------------------------------9
        IF ( all_or (status AND ir_lines (3 DOWNTO 0)) = '1') THEN
          load_offset_pc <= '1';
        END IF;
        next_state <= initial;
    END CASE;
  END PROCESS;
END dataflow_synthesizable;
```

- Issue signals to perform *jsr* and *branch*
- Use *all_or* function to imply a logic unit

Dataflow_synthesis.controller

```
entity PAR_CTRL is
  port ( . . . );
end PAR_CTRL ;

architecture exemplar of PAR_CTRL is
  component FD2
    port ( CD, CP, D : in std_logic ;  Q : inout std_logic ;  QN : out std_logic) ;
  end component ;
begin
  g1087 : AN3 port map ( Z=>vh_91, A=>vh_3, B=>vh_51, C=>vh_25);
  g1088 : NR2 port map ( Z=>vh_92, A=>LOAD_IR, B=>vh_91);
  g1089 : ND2 port map ( Z=>NEXT_STATE_1, A=>vh_86, B=>vh_92);
  g1090 : NR2 port map ( Z=>vh_96, A=>vh_91, B=>PC_OFFSET_ON_DBUS);
  g1091 : ND2 port map ( Z=>vh_97, A=>IR_LINES_4, B=>IR_ON_MAR_PAGE_BUS);
  g1092 : ND3 port map ( Z=>NEXT_STATE_2, A=>vh_96, B=>vh_97, C=>vh_83);
  g1093 : AN3 port map ( Z=>NEXT_STATE_3, A=>vh_3, B=>IR_LINES_5, C=>vh_25);
  vh_102 : FD2 port map ( CD=>vh_40, CP=>vh_80, D=>NEXT_STATE_0,
Q=>PRESENT_STATE_0, QN=>UPRESENT_STATE_0);
  vh_103 : FD2 port map ( CD=>vh_40, CP=>vh_80, D=>NEXT_STATE_1,
Q=>PRESENT_STATE_1, QN=>UPRESENT_STATE_1);
  vh_104 : FD2 port map ( CD=>vh_40, CP=>vh_80, D=>NEXT_STATE_2,
Q=>PRESENT_STATE_2, QN=>UPRESENT_STATE_2);
  vh_105 : FD2 port map ( CD=>vh_40, CP=>vh_80, D=>NEXT_STATE_3,
Q=>PRESENT_STATE_3, QN=>OPEN);
end exemplar ;
```

- Partial VHDL netlist generated by synthesizer for controller
- Four state flip-flops implementing binary encoding are used

© Zainalabedin Navabi, 1994

Dataflow_synthesis.data_components

```
LIBRARY IEEE;  USE IEEE.std_logic_1164.ALL;
--
LIBRARY EXEMPLAR;  USE EXEMPLAR.exemplar_1164.ALL;
--
USE WORK.synthesis_parameters.ALL;
USE WORK.synthesis_utilities.ALL;
USE WORK.alu_operations.ALL;
--
ENTITY arithmetic_logic_unit IS
  PORT (a_side, b_side : IN byte; code : IN std_logic_vector (2 DOWNTO 0);
        in_flags : IN nibble; z_out : OUT byte;  out_flags : OUT nibble);
END arithmetic_logic_unit;
--
ARCHITECTURE synthesizable_behavioral OF arithmetic_logic_unit IS
BEGIN
  coding: PROCESS (a_side, b_side, code, in_flags)
    VARIABLE t : std_logic_vector (9 DOWNTO 0);
    VARIABLE v, c, z, n : std_logic; --in_flags 3210
  BEGIN
    CASE code IS
      WHEN a_add_b  |  a_sub_b =>
        t := addsub_cv (b_side, a_side, in_flags(2), code(1));
        c := t(8);  v := t(9);  -- other flags are set at the end
      WHEN a_and_b =>
        t := "00" & ( a_side AND b_side );  c := in_flags(2);  v := in_flags(3);
      WHEN a_input  =>
        t := "00" & a_side;    c := in_flags(2);  v := in_flags(3);
      WHEN b_input  =>
        t := "00" & b_side;     c := in_flags(2);  v := in_flags(3);
      WHEN b_compl =>      t := "00" & ( NOT b_side );
        c := in_flags(2);  v := in_flags(3);
      WHEN OTHERS =>   t := "0000000000"; c:= '0'; v := '0';
    END CASE;
    n := t(7);    z := NOT all_or (t);   z_out <= t (7 DOWNTO 0);
    out_flags <= v & c & z & n;
  END PROCESS coding;
END synthesizable_behavioral;
```

- *alu* synthesizable description
- Avoid latching by assigning values to variables
- An example of combinational circuit synthesis

Dataflow_synthesis.data_components

```
LIBRARY IEEE;                USE IEEE.std_logic_1164.ALL;
--
LIBRARY EXEMPLAR;      USE EXEMPLAR.exemplar_1164.ALL;
--
USE WORK.synthesis_parameters.ALL;
USE WORK.synthesis_utilities.ALL;
--
ENTITY shifter_unit IS
  PORT (alu_side : IN byte;  arith_shift_left, arith_shift_right : IN std_logic;
       in_flags : IN nibble; obus_side : OUT byte;  out_flags : OUT nibble);
END shifter_unit;
--
ARCHITECTURE synthesizable_behavioral OF shifter_unit IS
BEGIN
  coding: PROCESS (alu_side, arith_shift_left, arith_shift_right, in_flags)
    VARIABLE t : std_logic_vector (7 DOWNTO 0);
    VARIABLE v, c, z, n : std_logic; --in_flags 3210
  BEGIN
    IF arith_shift_left = '1' THEN
      t := alu_side (6 DOWNTO 0) & '0';    n := t (7);       z := NOT all_or (t);
      c := alu_side (7);        v := alu_side (6) XOR alu_side (7);
    ELSIF arith_shift_right = '1' THEN
      t := alu_side (7) & alu_side (7 DOWNTO 1);
      n := t (7);  z := NOT all_or (t);    c := in_flags(2);      v := in_flags(3);
    ELSE -- none are '1'
      t := alu_side (7 DOWNTO 0);
      n := in_flags(0);  z := in_flags(1);  c := in_flags(2); v := in_flags(3);
    END IF;
    obus_side <= t;      out_flags <= v & c & z & n;
  END PROCESS coding;
END synthesizable_behavioral;
```

- *shu* synthesizable description
- Single combinational process
- Avoid latching by assigning values to variable under all conditions

Dataflow_synthesis.data_components

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
--
LIBRARY EXEMPLAR;
USE EXEMPLAR.exemplar_1164.ALL;
--
USE WORK.synthesis_parameters.ALL;
USE WORK.synthesis_utilities.ALL;
--
ENTITY accumulator_unit IS
  PORT (i8 : IN byte; o8 : OUT byte; load, zero, ck : IN std_logic);
END accumulator_unit;
--
ARCHITECTURE synthesizable_behavioral OF accumulator_unit IS
BEGIN
  PROCESS (ck)
  BEGIN
    IF (ck'EVENT AND ck = '0') THEN
      IF load = '1' THEN
        IF zero = '1' THEN
          o8 <= "00000000";
        ELSE
          o8 <= i8;
        END IF;
      END IF;
    END IF;
  END PROCESS;
END synthesizable_behavioral;
```

- Synthesizable behavioral description for *ac*
- On clock load or zero accumulator
- *ck* in the sensitivity and (ck'EVENT AND ck='0') are required for synthesis

Dataflow_synthesis.data_components

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
--
LIBRARY EXEMPLAR;
USE EXEMPLAR.exemplar_1164.ALL;
--
USE WORK.synthesis_parameters.ALL;
USE WORK.synthesis_utilities.ALL;
--
ENTITY instruction_register_unit IS
  PORT (i8 : IN byte; o8 : OUT byte; load, ck : IN std_logic);
END instruction_register_unit;
--
ARCHITECTURE synthesizable_behavioral OF instruction_register_unit IS
BEGIN
  PROCESS (ck)
  BEGIN
    IF (ck'EVENT AND ck = '0') THEN
      IF load = '1' THEN
         o8 <= i8;
      END IF;
    END IF;
  END PROCESS;
END synthesizable_behavioral;
```

- Synthesizable ir uses same style as *ac*

Dataflow_synthesis.data_components

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
--
LIBRARY EXEMPLAR;
USE EXEMPLAR.exemplar_1164.ALL;
--
USE WORK.synthesis_parameters.ALL;
USE WORK.synthesis_utilities.ALL;
--
ENTITY status_register_unit IS
  PORT (in_flags : IN nibble; out_status : OUT nibble;
        load, cm_carry, ck : IN std_logic );
END status_register_unit;
--
ARCHITECTURE synthesizable_behavioral OF status_register_unit IS
  SIGNAL status : nibble := "0000";
BEGIN
  clocking: PROCESS (ck)
  BEGIN
    IF ( ck'EVENT AND ck = '0') THEN
      IF (load = '1') THEN              status <= in_flags;
      ELSIF (cm_carry = '1') THEN       status (2) <= NOT status (2); --c flag
      END IF;
    END IF;
  END PROCESS;
  assigning: PROCESS (status)
  BEGIN
    out_status <= status;
  END PROCESS;
END synthesizable_behavioral;
```

- Synthesizable *sr* uses two processes
- First process is clocked and generates status register
- Assigning process is combinational and assigns status to output

Dataflow_synthesis.data_components

```
LIBRARY IEEE;     USE IEEE.std_logic_1164.ALL;
--
LIBRARY EXEMPLAR;
USE EXEMPLAR.exemplar_1164.ALL;    USE EXEMPLAR.exemplar.ALL;
--
USE WORK.synthesis_parameters.ALL;   USE WORK.synthesis_utilities.ALL;
--
ENTITY program_counter_unit IS
  PORT (i12 : IN twelve; o12 : OUT twelve;
        increment, load_page, load_offset, reset, ck : IN std_logic);
END program_counter_unit;
--
ARCHITECTURE synthesizable_behavioral OF program_counter_unit IS
  SIGNAL count : twelve := zero_12;
BEGIN
  clocking : PROCESS (ck)
  BEGIN
    IF ( ck'EVENT AND ck = '0' ) THEN
  IF reset = '1' THEN      count <= zero_12;
    ELSIF increment = '1' THEN    count <= count + "01";
      ELSE     IF load_page = '1' THEN
          count (11 DOWNTO 8) <= i12 (11 DOWNTO 8);
        END IF;
        IF load_offset = '1' THEN     count (7 DOWNTO 0) <= i12 (7 DOWNTO 0);
        END IF;
      END IF;
    END IF;
  END PROCESS;
  outputting: PROCESS (count)
  BEGIN  --using o12 in clocking, would imply registered o12
    o12 <= count;
  END PROCESS;
END synthesizable_behavioral;
```

- Synthesizable *pc* uses the general clocked process
- The single clock checking IF statement resets, increments or loads *pc*
- Use the overloaded + for adding
- Outputting process is combinational

© Zainalabedin Navabi, 1994

Dataflow_synthesis.data_components

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
--
LIBRARY EXEMPLAR;
USE EXEMPLAR.exemplar_1164.ALL;
--
USE WORK.synthesis_parameters.ALL;
USE WORK.synthesis_utilities.ALL;
--
ENTITY memory_address_register_unit IS
  PORT (i12 : IN twelve; o12 : OUT twelve;
       load_page, load_offset, ck : IN std_logic);
END memory_address_register_unit;
--
ARCHITECTURE synthesizable_behavioral OF memory_address_register_unit IS
BEGIN
  PROCESS (ck)
  BEGIN
    IF ( ck'EVENT AND ck = '0' ) THEN
      IF load_page = '1' THEN
        o12 (11 DOWNTO 8) <= i12 (11 DOWNTO 8);
      END IF;
      IF load_offset = '1' THEN
        o12 (7 DOWNTO 0) <= i12 (7 DOWNTO 0);
      END IF;
    END IF;
  END PROCESS;
END synthesizable_behavioral;
```

- A simple clocked process is used for synthesizable *mar*
- Two separate load inputs are synthesized

Dataflow_synthesis.data_path

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
--
LIBRARY EXEMPLAR;
USE EXEMPLAR.exemplar_1164.ALL;
--
USE WORK.synthesis_parameters.ALL;
USE WORK.synthesis_utilities.ALL;
--
ENTITY par_data_path IS
  PORT (databus : INOUT byte := "ZZZZZZZZ";  adbus : OUT twelve;
        clk : IN std_logic;
        -- register controls:
        load_ac, zero_ac,
        load_ir,
        increment_pc, load_page_pc, load_offset_pc, reset_pc,
        load_page_mar, load_offset_mar,
        load_sr, cm_carry_sr,
        -- bus connections:
        pc_on_mar_page_bus, ir_on_mar_page_bus,
        pc_on_mar_offset_bus, dbus_on_mar_offset_bus,
        pc_offset_on_dbus, obus_on_dbus, databus_on_dbus,
        mar_on_adbus,
        dbus_on_databus,
        -- logic unit function control inputs:
        arith_shift_left, arith_shift_right : IN std_logic;
        alu_code : IN std_logic_vector (2 DOWNTO 0);
        -- outputs to the controller:
        ir_lines : OUT byte; status : OUT nibble
       );
END par_data_path;
--
```

- Parwan data path uses signals of control unit
- Use the *std_logic* logic value system

Dataflow_synthesis.data_path

```
ARCHITECTURE synthesizable_structural OF par_data_path IS
  --
  COMPONENT accumulator_unit
    PORT (i8: IN byte; o8: OUT byte; load, zero, ck: IN std_logic);
  END COMPONENT;
  --
  COMPONENT instruction_register_unit
    PORT (i8: IN byte; o8: OUT byte; load, . . .);
  END COMPONENT;
   --
  COMPONENT program_counter_unit
    PORT (i12 : IN twelve; o12 : OUT twelve; increment, load_page, . . .);
  END COMPONENT;
  --
  COMPONENT memory_address_register_unit
    PORT (i12 : IN twelve; o12 : OUT twelve;  load_page, . . .);
  END COMPONENT;
  --
  COMPONENT status_register_unit
    PORT (in_flags : IN nibble; out_status : OUT nibble;  load, cm_carry, . . .);
  END COMPONENT;
  --
  COMPONENT arithmetic_logic_unit
    PORT (a_side, b_side : IN byte; code : IN std_logic_vector; . . .);
  END COMPONENT;
  --
  COMPONENT shifter_unit
    PORT (alu_side : IN byte; arith_shift_left, . . .);
  END COMPONENT;
  --
  SIGNAL ac_out, ir_out, alu_out, obus : byte;
  SIGNAL pc_out, mar_out : twelve;                 SIGNAL dbus : byte;
  SIGNAL alu_flags, shu_flags, sr_out : nibble;  SIGNAL mar_bus : twelve;
```

- Continuing synthesizable data path
- Data path components are declared
- Configuration specification will be ignored by synthesizer

Dataflow_synthesis.data_path

```
BEGIN
  -- bus connections --
  --
  mar_bus (7 DOWNTO 0)
     <= dbus WHEN dbus_on_mar_offset_bus = '1' ELSE "ZZZZZZZZ";
  databus <= dbus WHEN dbus_on_databus = '1' ELSE "ZZZZZZZZ";
  --
  dbus <= obus WHEN obus_on_dbus = '1' ELSE "ZZZZZZZZ";
  --
  dbus <= databus WHEN databus_on_dbus = '1' ELSE "ZZZZZZZZ";
  --
  -- connection of logical and register structures --
  --
  l1: arithmetic_logic_unit PORT MAP (dbus, ac_out, alu_code, sr_out, alu_out,
alu_flags);
  l2: shifter_unit PORT MAP (alu_out, arith_shift_left, arith_shift_right, alu_flags,
obus, shu_flags);

END synthesizable_structural;
```

- Instantiate components
- Use conditional signal assignments for bussing
- Explicit bussing is specified

Dataflow_synthesis.data_path

```
BEGIN
-- register connections --
  --
  r1: accumulator_unit PORT MAP (obus, ac_out, load_ac, zero_ac, clk);
  --
  r2: instruction_register_unit PORT MAP (obus, ir_out, load_ir, clk);
  ir_lines <= ir_out;
  mar_bus (11 DOWNTO 8) <= ir_out (3 DOWNTO 0) WHEN ir_on_mar_page_bus =
'1' ELSE "ZZZZ";
  --
  r3: program_counter_unit PORT MAP (mar_out, pc_out, increment_pc,
load_page_pc, load_offset_pc, reset_pc, clk);
  mar_bus (11 DOWNTO 8) <= pc_out (11 DOWNTO 8) WHEN
pc_on_mar_page_bus = '1' ELSE "ZZZZ";
  mar_bus (7 DOWNTO 0) <= pc_out (7 DOWNTO 0) WHEN pc_on_mar_offset_bus
= '1' ELSE "ZZZZZZZZ";
  dbus <= pc_out (7 DOWNTO 0) WHEN pc_offset_on_dbus = '1' ELSE "ZZZZZZZZ";
  --
  r4: memory_address_register_unit PORT MAP
     (mar_bus, mar_out, load_page_mar, load_offset_mar, clk);
  adbus <= mar_out WHEN mar_on_adbus = '1' ELSE "ZZZZZZZZZZZZ";
  --
  r5: status_register_unit PORT MAP (shu_flags, sr_out, load_sr, cm_carry_sr, clk);
  status <= sr_out;
 END synthesizable_structural;
```

- Instantiate components
- Register outputs are assigned to buses

Dataflow_synthesis.example

```
library ieee ;
use ieee.std_logic_1164.all ;

entity AC_REG is
  port (
    I8_7, I8_6, I8_5, I8_4, I8_3, I8_2, I8_1, I8_0, LOAD, ZERO, CK : in
    std_logic ;
    O8_7, O8_6, O8_5, O8_4, O8_3, O8_2, O8_1, O8_0 : inout std_logic) ;
end AC_REG ;

architecture exemplar of AC_REG is
    ...
  component FD1
    port (
      CP, D : in std_logic ;
      Q : inout std_logic ;
      QN : out std_logic) ;
  end component ;

begin
  g1017 : IVA port map ( Z=>vh_46, A=>vh_45);
  g1018 : AO2 port map ( Z=>vh_49, A=>I8_0, B=>vh_19, C=>O8_0, D=>vh_20);
  g1019 : IVA port map ( Z=>vh_50, A=>vh_49);
  vh_52 : FD1 port map ( CP=>vh_16, D=>vh_22, Q=>O8_7, QN=>OPEN);
  vh_59 : FD1 port map ( CP=>vh_16, D=>vh_50, Q=>O8_0, QN=>OPEN);

end exemplar ;
```

- Synthesized accumulator
- Hardware includes loading logic and eight flip-flops

Dataflow_synthesis.example

```
--
ENTITY par_central_processing_unit IS
  PORT (clk : IN std_logic;
        interrupt : IN std_logic;
        read_mem, write_mem : OUT std_logic;
        databus : INOUT byte := "ZZZZZZZZ"; adbus : OUT twelve
       );
END par_central_processing_unit;
--
ARCHITECTURE dataflow_synthesizable OF par_central_processing_unit IS
  --
  COMPONENT par_data_path
  PORT (databus : INOUT byte;  adbus : OUT twelve;
        clk : IN std_logic;
              );
  END COMPONENT;
  FOR data: par_data_path USE ENTITY WORK.par_data_path (structural_synthesizable);
  --
  COMPONENT par_control_unit
  PORT (clk : IN std_logic;
        load_ac, zero_ac,
                  );
  END COMPONENT;
  FOR ctrl: par_control_unit USE ENTITY WORK.par_control_unit (dataflow_synthesizable);
  --
  SIGNAL load_ac, zero_ac,
              load_ir,
  SIGNAL alu_code : std_logic_vector (2 DOWNTO 0);
  SIGNAL ir_lines : byte;  SIGNAL status : nibble;
BEGIN
  data: par_data_path PORT MAP
        (databus, adbus,
         clk,
         load_ac, zero_ac,
               );
  ctrl: par_control_unit PORT MAP
        (clk,
         load_ac, zero_ac,
                );
END dataflow_synthesizable;
```

- Wire data and control for the complete systems
- Each part has been separately synthesized

Dataflow_synthesis.example

	gates	critical path
AC	86	5.5 ns
ALU	241	21 .1
IR	81	3.5
MAR	121	3.5
Control	187	12.7
Data	204	8.8
PC	199	11.8
SHU	69	5.6
SR	43	4.2
Total	1231	21.1+8.8+5.6=35.5

- This compares with 1309 and cp of 29.6 ns for behavioral synthesis
- Worst case of data does not neccesarily correspond to ALU path
- Designer has more control in dataflow synthesis

Dataflow_synthesis.conclusions

1. Outline: Introduction, Organization, Outline

2. Review: Levels of abstraction, Entity and Architecture, Signal assignments, Guarded signal assignments, Three state bussing, Process statements, Combinational processes, Sequential processes, Multiplexing, Package

3. MSI Based Design: Use MSI parts of Part 2, Sequential multiplication, Designing the multiplier, Control and data parts, Testing the multiplier

4. General CPU Description: Will present a high level VHDL description of a small CPU. The CPU, Memory organization, Instructions, Addressing, Utilities for VHDL description, Interface, Behavioral description, Coding individual instructions

5. Manual Data_path Design: Will present VHDL description for manual design of data_path. Data components, Bussing structure, Description of logic, Description of registers, Bus resolutions, Component wiring

6. Manual Controller Design: Will present VHDL description for manual design of controller. Controller hardware, VHDL style, Signals and resolutions, State descriptions, Complete CPU, Testing CPU

7. Synthesis: Main concepts, Structural synthesis, Combinational circuits, Functional registers, State machines

8. Behavioral_Synthesis: Will present a high level synthesizable CPU description. Synthesis style, Necessary Package, Interface, General Layout, Registers, Clocking, Sequencing, Simulation and Synthesis

9. Dataflow_Synthesis: Will partition the CPU and synthesize each part separately. Synthesis style, Controller, Data components, Data path, Synthesized example, Conclusions

ERROR: undefined
OFFENDING COMMAND: -

STACK:

1
0