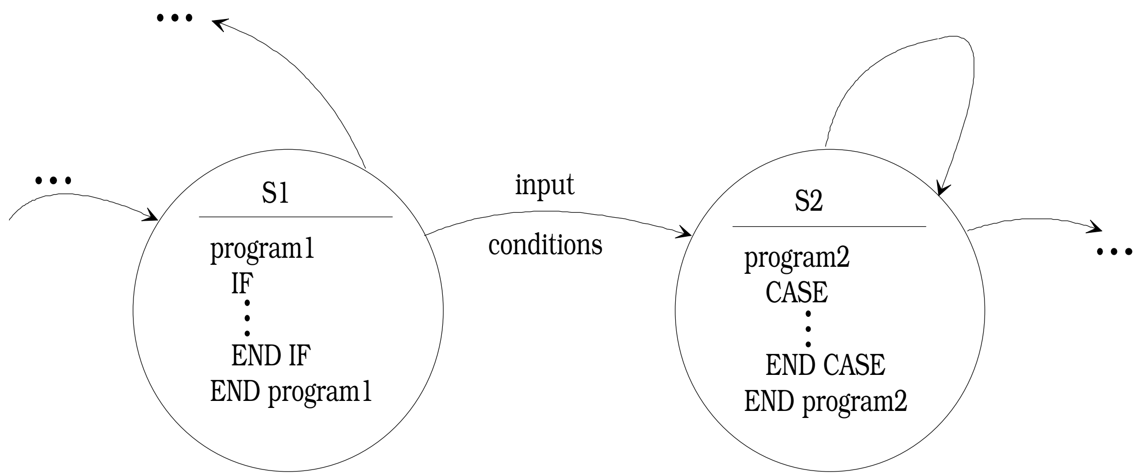
# PART 8

## Behavioral_Synthesis

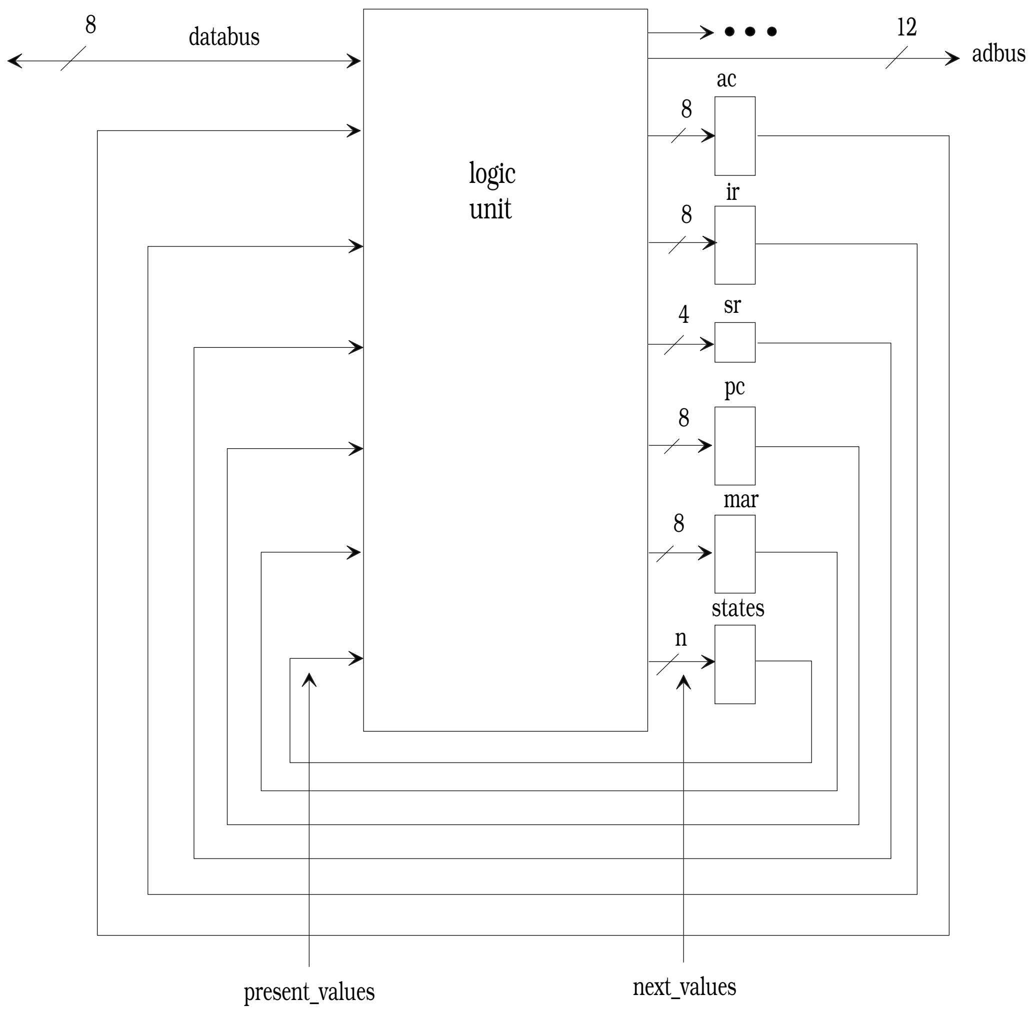
- **Synthesis style**
  - **- Description style**
  - **- Hardware style**
  - **- Coding style**
- **Necessary Package**
- **Interface**
- **General Layout**
- **Registers**
- **Clocking**
- **Sequencing**
- **Simulation and Synthesis**

# Behavioral_synthesis.style




- Description style:
  - State diagram style
  - Program for each state
- Programs specify register transfers and arithmetic
- Bussing will be decided by synthesis tool
- Each program is sequential
- Can use a state diagram template
- Customize each state program

# Behavioral_synthesis.style




- Generated hardware contains a logical unit and an array of registers
- State register is handled like any other register
- Clocking influences registers only
- Number of states depends on state assignment

# Behavioral_synthesis.style

```
ARCHITECTURE synthesizable ...
   state and register
   declarations
BEGIN
   clocking : PROCESS ( clk, ... )
   BEGIN
        clocked next to present transfers
        for registers and states
   END PROCESS, clocking;
   - -
   sequencing : PROCESS ( all present values and inputs )
   BEGIN
        assigning combinational values
        setting present values with next
        CASE present_state IS
           WHEN state_i =>
                sequential program i
                conditionally set next values
                use present values for conditions
                conditionally set next state
            WHEN state_j =>
                ...
        END CASE
   END PROCESS;
END synthesizable;
```

- In declarations, next and present signals are declared for all registers
- Clocking process uses clock to transfer next to present values
- Sequencing process contains sequential programs for each state

# Behavioral_synthesis.parwan.packages

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
--
PACKAGE synthesis_parameters IS
  CONSTANT single_byte_instructions : std_logic_vector (3 DOWNTO 0) := "1110";
  CONSTANT cla : std_logic_vector (3 DOWNTO 0) := "0001";
  CONSTANT cma : std_logic_vector (3 DOWNTO 0) := "0010";
  CONSTANT cmc : std_logic_vector (3 DOWNTO 0) := "0100";
  CONSTANT asl : std_logic_vector (3 DOWNTO 0) := "1000";
  CONSTANT asr : std_logic_vector (3 DOWNTO 0) := "1001";
  CONSTANT jsr : std_logic_vector (2 DOWNTO 0) := "110";
  CONSTANT bra : std_logic_vector (3 DOWNTO 0) := "1111";
  CONSTANT indirect : std_logic := '1';
  CONSTANT jmp : std_logic_vector (2 DOWNTO 0) := "100";
  CONSTANT sta : std_logic_vector (2 DOWNTO 0) := "101";
  CONSTANT lda : std_logic_vector (2 DOWNTO 0) := "000";
  CONSTANT ann : std_logic_vector (2 DOWNTO 0) := "001";
  CONSTANT add : std_logic_vector (2 DOWNTO 0) := "010";
  CONSTANT sbb : std_logic_vector (2 DOWNTO 0) := "011";
  CONSTANT jsr_or_bra : std_logic_vector (1 DOWNTO 0) := "11";
  --
  CONSTANT zero_12 : std_logic_vector (11 DOWNTO 0) := "000000000000";
  CONSTANT zero_8 : std_logic_vector (7 DOWNTO 0) := "00000000";
END synthesis_parameters;
```

- Parwan synthesizable description uses *synthesizable_package*
- Constants used for readability of code
- Use the *std_logic_1164 mvl9*

# Behavioral_synthesis.parwan.packages

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
--
PACKAGE synthesis_utilities IS
  FUNCTION all_or (v : std_logic_vector) RETURN std_logic;
  FUNCTION addsub_cv (a, b : std_logic_vector; cin : std_logic; add_sub :
std_logic)
  RETURN std_logic_vector;
  SUBTYPE nibble IS std_logic_vector (3 DOWNTO 0);
  SUBTYPE byte IS std_logic_vector (7 DOWNTO 0);
  SUBTYPE twelve IS std_logic_vector (11 DOWNTO 0);
END synthesis_utilities;
--
```

- Parwan *synthesizable_utilities* include functions and type declarations
- *all_or* and *addsub_cv* bodies are in the package body

# Behavioral_synthesis.parwan.packages

```
FUNCTION all_or (v : std_logic_vector) RETURN std_logic IS
   VARIABLE temp : std_logic;
 BEGIN
   temp := '0';
   FOR i IN v'RANGE LOOP
     temp := temp OR v(i);
   END LOOP;
   RETURN temp;
 END all_or;
```

- *all_or* ORes all its input bits
- Implies a *multi_input* OR gate
- Used for checking zero result

# Behavioral_synthesis.parwan.packages

```
FUNCTION addsub_cv
     (a, b : std_logic_vector; cin : std_logic; add_sub : std_logic)
RETURN std_logic_vector IS
   --for adding add_sub is 0, for subtracting add_sub is 1
   --left bits of a and b are sign bit
   VARIABLE r: std_logic_vector (a'LEFT + 2 DOWNTO 0); --temp  result
   -- Left two bits of r are for v and c respectively
   VARIABLE c: std_logic_vector (a'LEFT DOWNTO 0);  --intermediate carry
   VARIABLE a_sign, b_sign: std_logic;  --sign bits of a and b
   VARIABLE cin_signed : std_logic;  --complement cin for sub
   VARIABLE b_signed : std_logic_vector (b'RANGE);  --complement b for sub
  BEGIN
   cin_signed := cin XOR add_sub;
   FOR i IN b'RANGE LOOP
     b_signed (i) := b(i) XOR add_sub;
   END LOOP;
   a_sign := a(a'LEFT);     b_sign := b_signed(b'LEFT);
   r(0) := a(0) XOR b_signed(0) XOR cin_signed;
   c(0) := ((a(0)XOR b_signed(0))AND cin_signed)OR (a(0) AND b_signed(0));
   FOR i IN 1 TO (a'LEFT) LOOP
     r(i) := a(i) XOR b_signed(i) XOR c(i-1);
     c(i) := ((a(i) XOR b_signed(i)) AND c(i-1)) OR (a(i) AND b_signed(i));
   END LOOP;
   r(a'LEFT+1) := c(a'LEFT);
   IF a_sign = b_sign AND r(a'LEFT) /= a_sign
     THEN r(a'LEFT+2) := '1'; --overflow
   ELSE r(a'LEFT+2) := '0'; END IF;
   RETURN r;
  END addsub_cv;
END synthesis_utilities;
```

- *addsub_cv* adds or subtracts
- MSB bits are *v* and *c* bits
- Combined *add_sub* helps resource sharing

# Behavioral_synthesis.parwan.interface

```
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_central_processing_unit IS
  PORT (clk : IN std_logic;
        interrupt : IN std_logic;
        read_mem, write_mem : OUT std_logic;
        databus : INOUT std_logic_vector (7 DOWNTO 0);
        adbus : OUT std_logic_vector (11 DOWNTO 0)
       );
END par_central_processing_unit;
```

- Interface and libraries for synthesizable Parwan
- *std_logic* is resolved signal
- databus is INOUT to be driver by the CPU or the memory

# Behavioral_synthesis.parwan.general_layout

ARCHITECTURE behavioral_synthesizable

| |
|---|
| Declarations:<br>ac, ir, ..., states |

BEGIN

| |
|---|
| Clocking process |

```
Sequencing : PROCESS
   CASE present_state
       .
       .
       .
   END CASE
```

END behavioral_synthesizable;

- Declare registers and states
- Update registers in clocking PROCESS
- Condition on *present_state* perform register and arithmetic operations

# Behavioral_synthesis.parwan.registers

```
SIGNAL ac, next_ac, ir, next_ir : std_logic_vector (7 DOWNTO 0);
SIGNAL sr, next_sr : std_logic_vector (3 DOWNTO 0);
SIGNAL pc, next_pc, mar, next_mar : std_logic_vector (11 DOWNTO 0);
TYPE cpu_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;
```

- Declarations include present & next for all registers and states
- No buses are declared
- There are 9 CPU states
- Use *CPU_states* enumeration type for the states of Parwan

# Behavioral_synthesis.parwan.clocking

```
clocking : PROCESS (clk, interrupt)
  BEGIN
    IF (interrupt = '1') THEN
      present_state <= initial;
      ac <= zero_8;
      ir <= zero_8;
      sr <= "0000";
      pc <= zero_12;
      mar <= zero_12;
    ELSIF clk'EVENT AND clk = '0' THEN
      ac <= next_ac;
      ir <= next_ir;
      sr <= next_sr;
      pc <= next_pc;
      mar <= next_mar;
      present_state <= next_state;
    END IF;
  END PROCESS clocking;
```

- Register present states must receive values in all cases
- Clock condition must be the last condition
- Resetting is done synchronously
- On falling edge of clock all next values are transferred to present

# Behavioral_synthesis.parwan.sequencing

```
sequencing : PROCESS ( present_state, interrupt, databus, ac, ir, sr, pc, mar )
   VARIABLE ten_bit : std_logic_vector (9 DOWNTO 0);
 BEGIN
   next_pc <= pc;
   databus <= "ZZZZZZZZ";
   adbus <= "ZZZZZZZZZZZZ";
   read_mem <= '0';
   write_mem <= '0';
   next_ir <= ir;
   next_ac <= ac;
   next_mar <= mar;
   next_sr <= sr;
```

- Sequencing represents a combinational circuit
- All next values retain values in process
- Retaining implies latching
- To avoid latching, set initial values to all next states

# Behavioral_synthesis.parwan.sequencing

```
CASE present_state IS
   WHEN initial =>
      ...
   WHEN instr_fetch =>
      ...
   WHEN do_one_bytes =>
      ...
END CASE;
```

- The sequencing PROCESS contains a CASE statement
- Choices separate CPU states

# Behavioral_synthesis.parwan.sequencing

```
WHEN initial =>          --------------------------------------------1
      IF (interrupt = '1') THEN
        next_pc <= zero_12;
        next_state <= initial;
      ELSE
        next_mar <= pc;
        next_state <= instr_fetch;
      END IF;
    WHEN instr_fetch =>   ---------------------------------------2
      adbus <= mar;
      read_mem <= '1';
      next_ir <= databus;
      next_pc <= pc + "01";
      next_state <= do_one_bytes;
```

- Programs for initial and *instr_fetch*
- No internal bussing is specified

# Behavioral_synthesis.parwan.sequencing

```
WHEN do_one_bytes =>   --------------------------------------3
    next_mar <= pc;  -- prepare for next memory read
    IF (ir(7 DOWNTO 4) /= single_byte_instructions) THEN
      next_state <= opnd_fetch;
    ELSE
      CASE ir(3 DOWNTO 0) IS
        WHEN cla =>
          next_ac <= zero_8;
        WHEN cma =>
          next_sr(0) <= ac(7);                          --n flag
          next_sr(1) <= NOT all_or (NOT ac);            --z
          next_ac <= NOT ac;
        WHEN cmc =>
          next_sr(2) <= NOT sr(2);                      --c flag
        WHEN asl =>
          next_sr(0) <= ac(6);                          --n
          next_sr(1) <= NOT all_or (ac(6 DOWNTO 0)); --z
          next_sr(2) <= ac(7);                          --c
          next_sr(3) <= ac(6) XOR ac(7);                --v
          next_ac <= ac (6 DOWNTO 0) & '0';
        WHEN asr =>
          next_sr(0) <= ac(7);                          --n
          next_sr(1) <= NOT all_or (ac(7 DOWNTO 1)); --z
          next_ac <= ac (7) & ac (7 DOWNTO 1);
        WHEN OTHERS => NULL;
      END CASE;
      next_state <= instr_fetch;
    END IF;
```

- Program for *do_one_bytes*
- A sequential program contains behavioral constructs
- Each program sets next states

# Behavioral_synthesis.parwan.sequencing

```
WHEN opnd_fetch =>   ----------------------------------------4
        adbus <= mar;
        read_mem <= '1';
        next_mar(7 DOWNTO 0) <= databus;
        IF ( ir(7 DOWNTO 6) /= jsr_or_bra ) THEN
          next_mar(11 DOWNTO 8) <= ir(3 DOWNTO 0);
          IF ( ir(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(5) = '0' ) THEN  -- jsr
            next_state <= do_jsr;
          ELSE
            next_state <= do_branch;
          END IF;
        END IF;
        next_pc <= pc + "01";
```

- Assignment to adbus and registers are done in each program
- Use *pc* + "01", + is overloaded for *std_logic*

# Behavioral_synthesis.parwan.sequencing

```
WHEN do_indirect =>   ---------------------------------------5
     adbus <= mar;
     read_mem <= '1';
     next_mar(7 DOWNTO 0) <= databus;
     next_state <= do_two_bytes;
```

- Program to handle indirect
- Can easily be extended for indexed or other modes

# **Behavioral_synthesis.parwan.sequencing**

```
WHEN do_two_bytes =>   ---------------------------------------6
    IF ( ir(7 DOWNTO 5) = jmp ) THEN
      next_pc <= mar;
      next_state <= instr_fetch;
    ELSIF ( ir(7 DOWNTO 5) = sta ) THEN
      adbus <= mar;
      databus <= ac;
      write_mem <= '1';
      next_state <= initial;
    ELSIF ( ir(7) = '0' ) THEN              ------ lda, and, add, sub
      adbus <= mar;
      read_mem <= '1';
      IF ( ir(6) = '0' ) THEN                    ---- lda, and
        IF ( ir(5) = '0' ) THEN                    -- lda
          ten_bit := sr (3 DOWNTO 2) & databus;          --
        ELSE                                     -- and
          ten_bit := sr (3 DOWNTO 2) & ( ac AND databus ); --
        END IF;
      ELSE                                     ---- add, sub
        ten_bit := addsub_cv (ac, databus, sr(2), ir(5));
      END IF;
      next_sr(0) <= ten_bit (7);                  --n
      next_sr(1) <= NOT all_or ( ten_bit(7 DOWNTO 0) );  --z
      next_sr(2) <= ten_bit (8);                  --c
      next_sr(3) <= ten_bit (9);                  --v
      next_ac <= ten_bit (7 DOWNTO 0);
      next_state <= initial;
    ELSE
      next_state <= initial; --never happens
    END IF;
```

- Program for doing two byte instructions
- Think sequentially in each state
- Multiplex various operation results on *ten_bit* variable

# Behavioral_synthesis.Parwan.sequencing

```
WHEN do_jsr =>   -------------------------------------------7
        adbus <= mar;
        databus <= pc (7 DOWNTO 0);
        write_mem <= '1';
        next_pc <= mar;
        next_state <= continue_jsr;
      WHEN continue_jsr =>   -------------------------------------8
        next_pc <= pc + "01";
        next_state <= initial;
      WHEN do_branch =>   ----------------------------------------9
        IF ( all_or (sr AND ir(3 DOWNTO 0)) = '1') THEN
          next_pc <= mar;
        END IF;
        next_state <= initial;
    END CASE;
```

- Last three state programs for *do_jsr*, *continue_jsr* and *do_branch*
- This ends all state programs

# Behavioral_synthesis.Parwan.Parwan

```
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_central_processing_unit IS
  PORT (clk : IN std_logic;    interrupt : IN std_logic;
      read_mem, write_mem : OUT std_logic;
      databus : INOUT std_logic_vector (7 DOWNTO 0);
      adbus : OUT std_logic_vector (11 DOWNTO 0)   );
END par_central_processing_unit;
--
ARCHITECTURE behavioral_synthesizable OF par_central_processing_unit IS
  SIGNAL ac, next_ac, ir, next_ir : std_logic_vector (7 DOWNTO 0);
  SIGNAL sr, next_sr : std_logic_vector (3 DOWNTO 0);
  SIGNAL pc, next_pc, mar, next_mar : std_logic_vector (11 DOWNTO 0);
  TYPE cpu_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;      ac <= zero_8;      ir <= zero_8;      sr <= "0000";    pc <= zero_12;    mar <= zero_12;
    ELSIF clk'EVENT AND clk = '0' THEN
      ac <= next_ac;     ir <= next_ir;      sr <= next_sr;     pc <= next_pc;     mar <= next_mar;    present_state <= next_state;
    END IF;
  END PROCESS clocking;
  --
  sequencing : PROCESS ( present_state, interrupt, databus, ac, ir, sr, pc, mar )
    VARIABLE ten_bit : std_logic_vector (9 DOWNTO 0);
  BEGIN
    next_pc <= pc;
    databus <= "ZZZZZZZZ";    adbus <= "ZZZZZZZZZZZZ";
    read_mem <= '0';    write_mem <= '0';    next_ir <= ir;     next_ac <= ac;     next_mar <= mar;     next_sr <= sr;
    ten_bit := "0000000000";
    CASE present_state IS
      WHEN initial =>   --------------------------------------------1
        IF (interrupt = '1') THEN
          next_pc <= zero_12;         next_state <= initial;
        ELSE
          next_mar <= pc;
          next_state <= instr_fetch;
        END IF;
      WHEN instr_fetch =>   ----------------------------------------2
        adbus <= mar;      read_mem <= '1';      next_ir <= databus;      next_pc <= pc + "01";     next_state <= do_one_bytes;
      WHEN do_one_bytes =>   --------------------------------------3
        next_mar <= pc;  -- prepare for next memory read
        IF (ir(7 DOWNTO 4) /= single_byte_instructions) THEN
          next_state <= opnd_fetch;
        ELSE
          CASE ir(3 DOWNTO 0) IS
            WHEN cla =>
              next_ac <= zero_8;
            WHEN cma =>
              next_sr(0) <= ac(7);                   --n flag
              next_sr(1) <= NOT all_or (NOT ac);      --z
              next_ac <= NOT ac;
            WHEN cmc =>
              next_sr(2) <= NOT sr(2);  --c flag
            WHEN asl =>
              next_sr(0) <= ac(6);                       --n
              next_sr(1) <= NOT all_or (ac(6 DOWNTO 0));  --z
              next_sr(2) <= ac(7);                       --c
              next_sr(3) <= ac(6) XOR ac(7);             --v
              next_ac <= ac (6 DOWNTO 0) & '0';
            WHEN asr =>
              next_sr(0) <= ac(7);                       --n
              next_sr(1) <= NOT all_or (ac(7 DOWNTO 1));  --z
              next_ac <= ac (7) & ac (7 DOWNTO 1);
            WHEN OTHERS => NULL;
          END CASE;
          next_state <= instr_fetch;
        END IF;
```

- Complete Parwan description
- Part 1/2


© Zainalabedin Navabi, 1994

# Behavioral_synthesis.Parwan.Parwan

```
 WHEN opnd_fetch =>   ----------------------------------------4
       adbus <= mar;
       read_mem <= '1';
       next_mar(7 DOWNTO 0) <= databus;
       IF ( ir(7 DOWNTO 6) /= jsr_or_bra ) THEN
         next_mar(11 DOWNTO 8) <= ir(3 DOWNTO 0);
         IF ( ir(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(5) = '0' ) THEN  -- jsr
           next_state <= do_jsr;
         ELSE
           next_state <= do_branch;
         END IF;
       END IF;
       next_pc <= pc + "01";
     WHEN do_indirect =>   ---------------------------------------5
       adbus <= mar;
       read_mem <= '1';
       next_mar(7 DOWNTO 0) <= databus;
       next_state <= do_two_bytes;
     WHEN do_two_bytes =>   -------------------------------------6
       IF ( ir(7 DOWNTO 5) = jmp ) THEN
         next_pc <= mar;
         next_state <= instr_fetch;
       ELSIF ( ir(7 DOWNTO 5) = sta ) THEN
         adbus <= mar;
         databus <= ac;
         write_mem <= '1';
         next_state <= initial;
       ELSIF ( ir(7) = '0' ) THEN              ------ lda, and, add, sub
         adbus <= mar;
         read_mem <= '1';
         IF ( ir(6) = '0' ) THEN                          ---- lda, and
           IF ( ir(5) = '0' ) THEN                         -- lda
             ten_bit := sr (3 DOWNTO 2) & databus;           --
           ELSE                                            -- and
             ten_bit := sr (3 DOWNTO 2) & ( ac AND databus ); --
           END IF;
         ELSE                                            ---- add, sub
           ten_bit := addsub_cv (ac, databus, sr(2), ir(5));
         END IF;
         next_sr(0) <= ten_bit (7);                       --n
         next_sr(1) <= NOT all_or ( ten_bit(7 DOWNTO 0) );  --z
         next_sr(2) <= ten_bit (8);                       --c
         next_sr(3) <= ten_bit (9);                       --v
         next_ac <= ten_bit (7 DOWNTO 0);
         next_state <= initial;
       ELSE
         next_state <= initial; --never happens
       END IF;
     WHEN do_jsr =>   --------------------------------------------7
       adbus <= mar;
       databus <= pc (7 DOWNTO 0);
       write_mem <= '1';
       next_pc <= mar;
       next_state <= continue_jsr;
     WHEN continue_jsr =>   --------------------------------------8
       next_pc <= pc + "01";
       next_state <= initial;
     WHEN do_branch =>   -----------------------------------------9
       IF ( all_or (sr AND ir(3 DOWNTO 0)) = '1') THEN
         next_pc <= mar;
       END IF;
       next_state <= initial;
   END CASE;
  END PROCESS;
END behavioral_synthesizable;
```

- Complete Parwan description
- Part 2/2



© Zainalabedin Navabi, 1994

# Behavioral_synthesis.Parwan.simulation_&_synthesis

- Simulating Parwan in a simple testbench

# Behavioral_synthesis.Parwan.simulation_&_synthesis

```
entity PARWAN is
  port (
    CLK, INTERRUPT : in std_logic ;
    DATABUS_7, DATABUS_6, DATABUS_5, DATABUS_4, DATABUS_3, DATABUS_2,
    DATABUS_1, DATABUS_0 : inout std_logic ;
    READ_MEM, WRITE_MEM, ADBUS_11, ADBUS_10, ADBUS_9, ADBUS_8, ADBUS_7,
    ADBUS_6, ADBUS_5, ADBUS_4, ADBUS_3, ADBUS_2, ADBUS_1, ADBUS_0 : out
    std_logic) ;
end PARWAN ;

architecture exemplar of PARWAN is
  . . .
   component FD2
    port (CD, CP, D : in std_logic ;  Q : inout std_logic ;  QN : out std_logic) ;
  end component ;
  . . .
  g1342 : B4I port map ( Z=>vh_619, A=>vh_621);
  g1343 : B4I port map ( Z=>vh_620, A=>vh_621);
  g1344 : IVA port map ( Z=>vh_621, A=>vh_36);
  vh_622 : FD2 port map ( CD=>vh_619, CP=>vh_38, D=>NEXT_STATE_0,
                                        Q=>PRESENT_STATE_0, QN=>UPRESENT_STATE_0);
  vh_623 : FD2 port map ( CD=>vh_619, CP=>vh_38, D=>NEXT_STATE_1,
                                        Q=>PRESENT_STATE_1, QN=>UPRESENT_STATE_1);
  vh_624 : FD2 port map ( CD=>vh_619, CP=>vh_38, D=>NEXT_STATE_2,
                                        Q=>PRESENT_STATE_2, QN=>UPRESENT_STATE_2);
  vh_625 : FD2 port map ( CD=>vh_619, CP=>vh_38, D=>NEXT_STATE_3,
                                        Q=>PRESENT_STATE_3, QN=>OPEN);
  vh_626 : FD2 port map ( CD=>vh_619, CP=>vh_38, D=>NEXT_MAR_11, Q=>MAR_11, QN=>UMAR_11);
  vh_627 : FD2 port map ( CD=>vh_619, CP=>vh_38, D=>NEXT_MAR_10, Q=>MAR_10, QN=>UMAR_10);
  vh_628 : FD2 port map ( CD=>vh_619, CP=>vh_38, D=>NEXT_MAR_9, Q=>MAR_9, QN=>UMAR_9);
  vh_629 : FD2 port map ( CD=>vh_619, CP=>vh_38, D=>NEXT_MAR_8, Q=>MAR_8, QN=>UMAR_8);
  vh_630 : FD2 port map ( CD=>vh_619, CP=>vh_38, D=>NEXT_MAR_7, Q=>MAR_7, QN=>UMAR_7);
  vh_631 : FD2 port map ( CD=>vh_619, CP=>vh_38, D=>NEXT_MAR_6, Q=>MAR_6, QN=>UMAR_6);
  vh_632 : FD2 port map ( CD=>vh_619, CP=>vh_38, D=>NEXT_MAR_5, Q=>MAR_5, QN=>UMAR_5);
  vh_633 : FD2 port map ( CD=>vh_619, CP=>vh_38, D=>NEXT_MAR_4, Q=>MAR_4, QN=>UMAR_4);
  . . .
```

- Parwan uses 1309 LSI 10k library cells
- Cells are basic gates, flip-flop and mux
- VHDL was chosen for output format
- Can use VHDL simulator to verify synthesized circuit
- 9 states are mapped to 4 state variables

# Behavioral_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

© Zainalabedin Navabi, 1994

ERROR: syntaxerror
OFFENDING COMMAND: ----nostringval----

STACK: