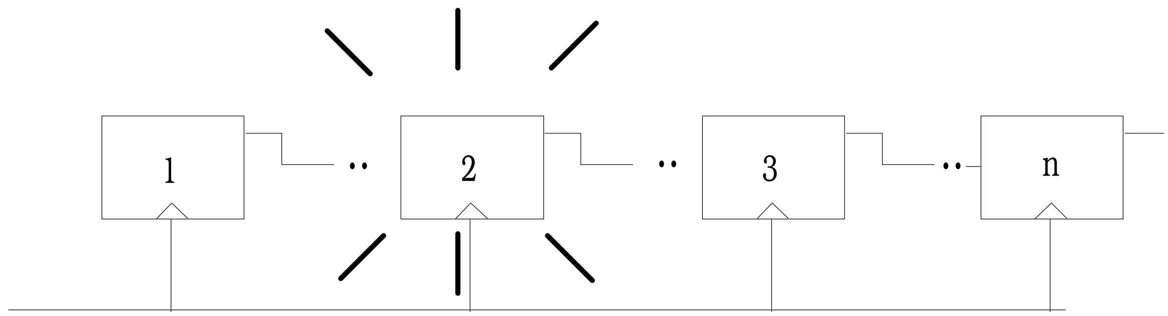
PART 6

Manual Controller Design

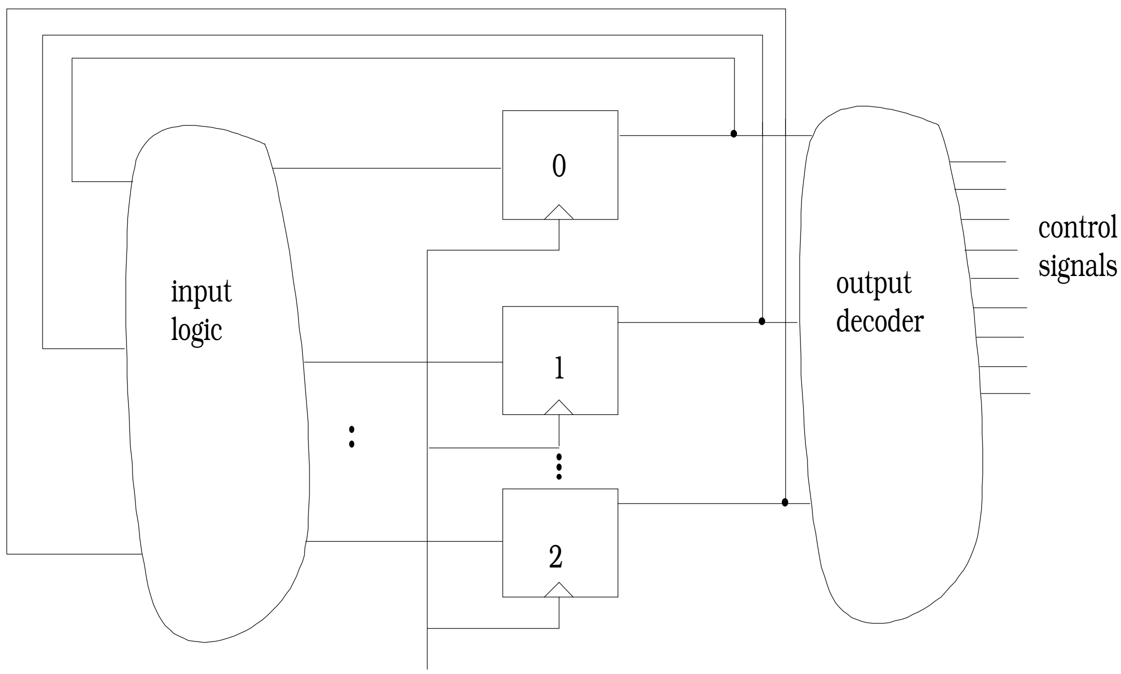
- Controller hardware
- VHDL style
- Signals and resolutions
- State descriptions
- Complete CPU
- Testing CPU

Controller.onehot



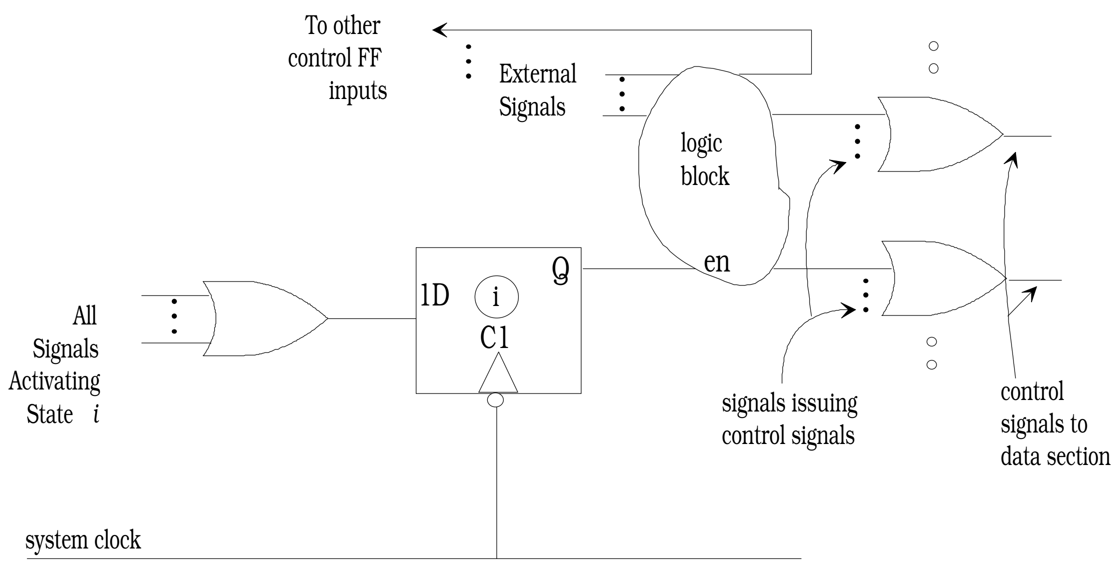
- Only one hot state at a given time
- Easier design,
- No need for encoding states
- No need for decoding control lines

Controller.hardware



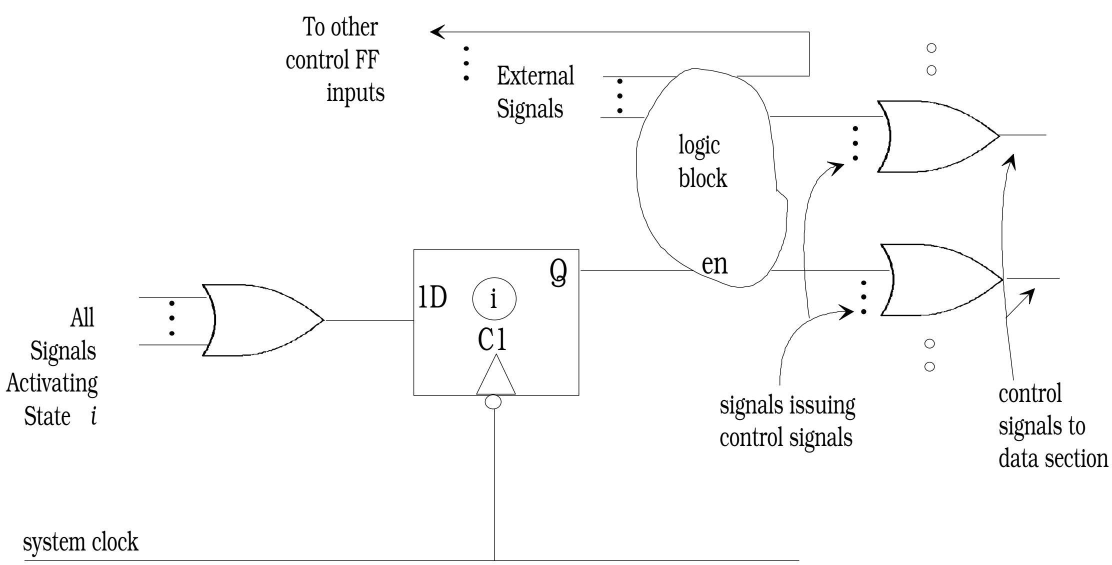
- Encoded state machine
- Requires more output and input logic
- Requires fewer flip-flops

Controller.hardware



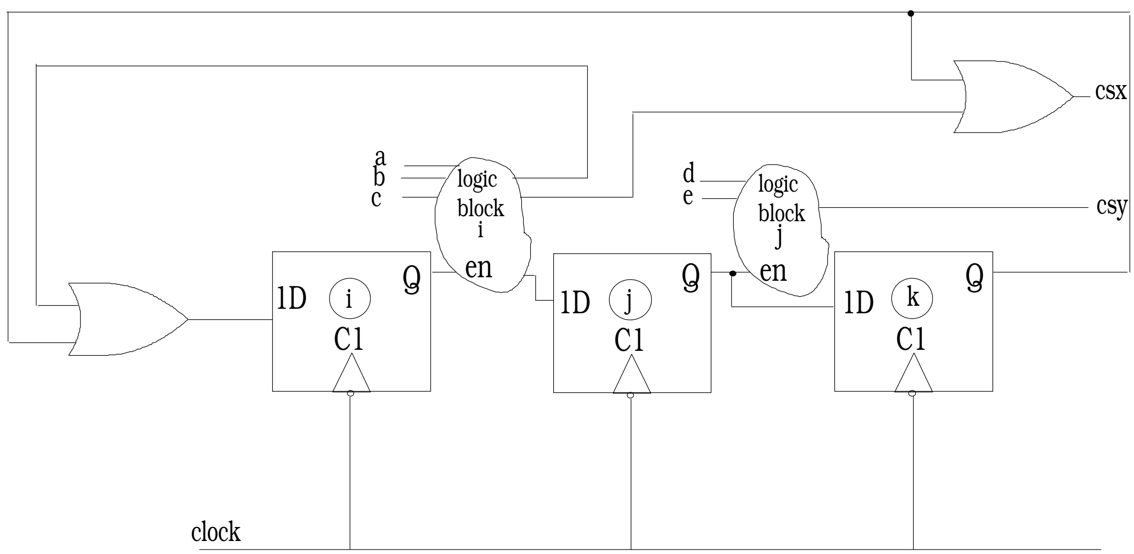
- We will use a one hot controller
- Hardware surrounding a control flip-flop
- A control flip-flop conditionally activates control signals

Controller.hardware



- Control signals are issued by various control flip-flops
- Use *OR* gate for oring same-name control signals
- All signals activating a control flip-flop are ored

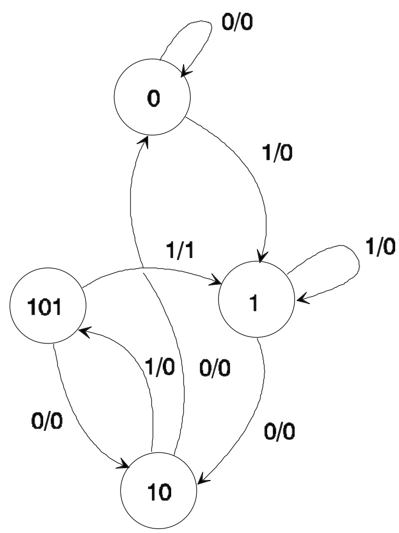
Controller.hardware



- Three control states and corresponding logic
- *csy* conditionally issued by *j*
- *csx* conditionally issued by *i* also unconditionally issued by *k*

Controller.VHDL_style

```
ARCHITECTURE multiple_state_machine OF detector IS
  SIGNAL s : ored_bit_vector (1 TO 4) REGISTER := "1000";
BEGIN
  clocking : BLOCK (clk = '1' AND NOT clk'STABLE)
  BEGIN
    s1: BLOCK (s(1) = '1' AND GUARD)
    BEGIN
      s(1) <= GUARDED '1' WHEN x = '0' ELSE '0';
      s(2) <= GUARDED '1' WHEN x = '1' ELSE '0';
    END BLOCK s1;
    s2: BLOCK (s(2) = '1' AND GUARD)
    BEGIN
      s(3) <= GUARDED '1' WHEN x = '0' ELSE '0';
      s(2) <= GUARDED '1' WHEN x = '1' ELSE '0';
    END BLOCK s2;
    s3: BLOCK (s(3) = '1' AND GUARD)
    BEGIN
      s(1) <= GUARDED '1' WHEN x = '0' ELSE '0';
      s(4) <= GUARDED '1' WHEN x = '1' ELSE '0';
    END BLOCK s3;
    s4: BLOCK (s(4) = '1' AND GUARD)
    BEGIN
      s(3) <= GUARDED '1' WHEN x = '0' ELSE '0';
      s(2) <= GUARDED '1' WHEN x = '1' ELSE '0';
      z <= '1' WHEN (s(4) = '1' AND x = '1') ELSE '0';
    END BLOCK s4;
    s <= GUARDED "0000";
  END BLOCK clocking;
END multiple_state_machine;
```



- State machine description style
- A guarded block for each state

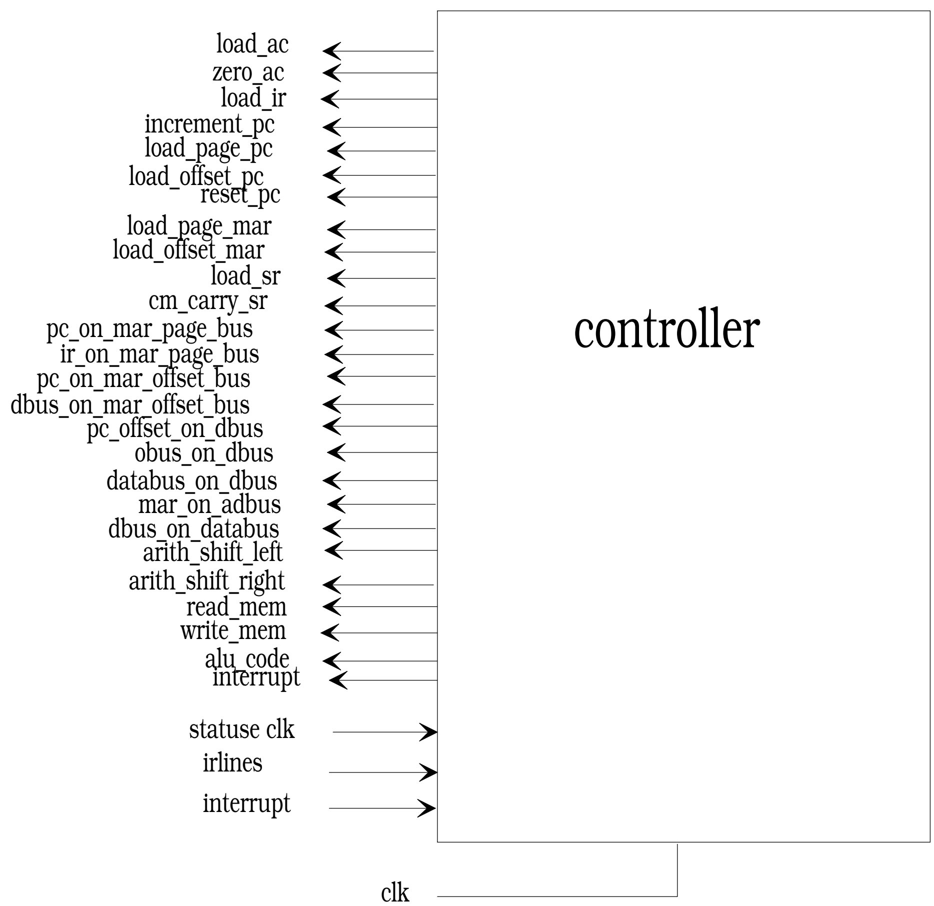
Controller.VHDL_style

```
sn: Block (guard on state n)
BEGIN
        issue control signals
         .
         .
         .
        ck: Block (guard on clock)
                issue next state;
        END Block ck;
END Block sn;
```

- Issue control signals in each state
- A guarded block guards next state assignment

© Zainalabedin Navabi, 1994

Controller.signals_and_resolutions



- Controller issues signals used by data
- Decisions made based on *ir_lines*

Controller.signals_and_resolutions

```
ARCHITECTURE dataflow OF par_control_unit IS
-- oring is implied in the following signals (oi)
SIGNAL load_ac_oi, zero_ac_oi,
       load_ir_oi,
       increment_pc_oi, load_page_pc_oi, load_offset_pc_oi, reset_pc_oi,
       load_page_mar_oi, load_offset_mar_oi,
       load_sr_oi, cm_carry_sr_oi,
       pc_on_mar_page_bus_oi, ir_on_mar_page_bus_oi,
       pc_on_mar_offset_bus_oi, dbus_on_mar_offset_bus_oi,
       pc_offset_on_dbus_oi, obus_on_dbus_oi, databus_on_dbus_oi,
       mar_on_adbus_oi,
       dbus_on_databus_oi,
       arith_shift_left_oi, arith_shift_right_oi,
       read_mem_oi, write_mem_oi : ored_qit BUS;
SIGNAL alu_code_oi : ored_qit_vector (2 DOWNTO 0) BUS;
SIGNAL s : ored_qit_vector (9 DOWNTO 1) REGISTER := "000000001";
```

- Use resolved signals to imply oring
- Multiple assignments to *oi* signals are ored
- Declare *s* as a register of 9 states

Controller.signals_and_resolutions

```
FUNCTION oring ( drivers : qit_vector) RETURN qit;
SUBTYPE ored_qit IS oring qit;
TYPE ored_qit_vector IS ARRAY (NATURAL RANGE <>) OF ored_qit;
```

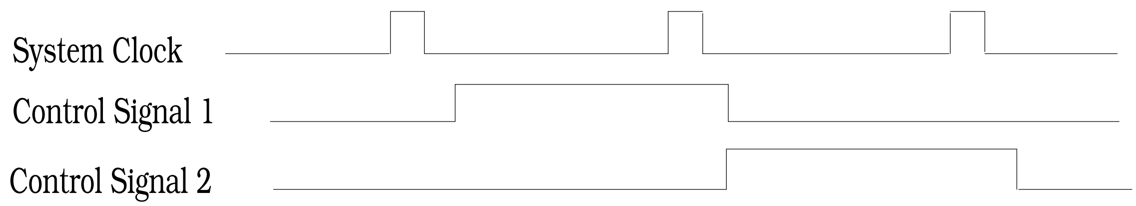
(a)

```
FUNCTION oring ( drivers : qit_vector) RETURN qit IS
  VARIABLE accumulate : qit := '0';
BEGIN
  FOR i IN drivers'RANGE LOOP
    accumulate := accumulate OR drivers(i);
  END LOOP;
  RETURN accumulate;
END oring;
```

(b)

- Use oring resolution function
- All assignments to same state are ored
- All assignments to same control lines are ored

Controller.signals_and_resolutions



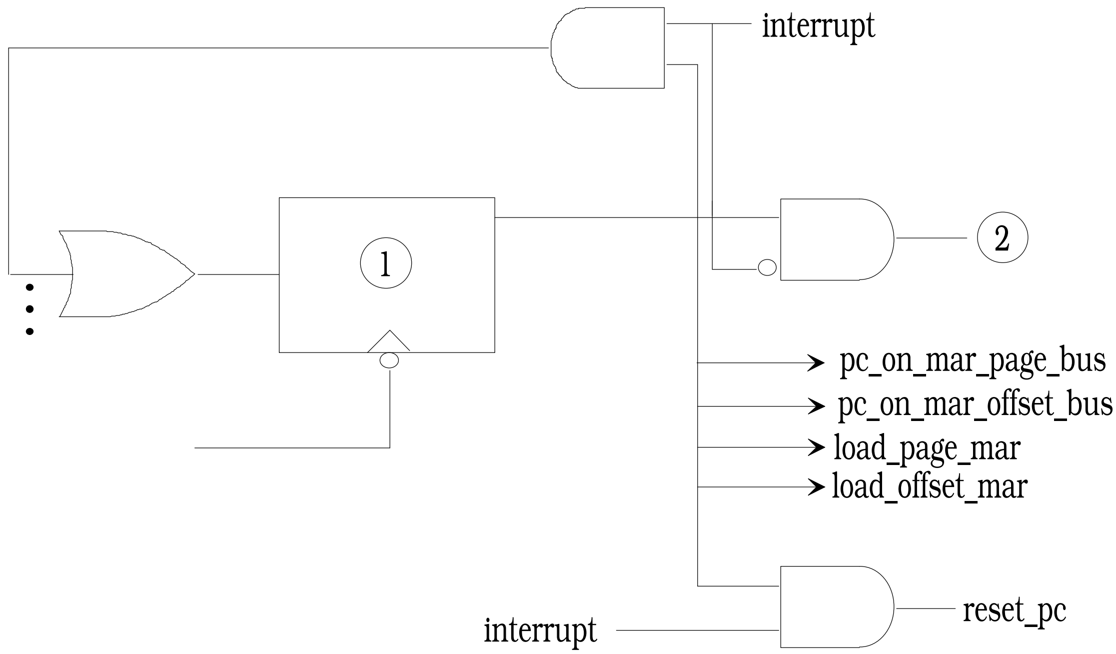
- Ored signals control flow of data in the data section
- Ored signals make appropriate state transitions
- Data and control signals follow the same timing
- Control signals are active from one falling edge to another
- One period allows propagation through logic and buses

Controller.signals_and_resolutions

```
BEGIN
  -- implied or assignments to output signals
  load_ac <= load_ac_oi;
  zero_ac <= zero_ac_oi;
  load_ir <= load_ir_oi;
  increment_pc <= increment_pc_oi;
  load_page_pc <= load_page_pc_oi;
  load_offset_pc <= load_offset_pc_oi;
  reset_pc <= reset_pc_oi;
  load_page_mar <= load_page_mar_oi;
  load_offset_mar <= load_offset_mar_oi;
  load_sr <= load_sr_oi;
  cm_carry_sr <= cm_carry_sr_oi;
  pc_on_mar_page_bus <= pc_on_mar_page_bus_oi;
  ir_on_mar_page_bus <= ir_on_mar_page_bus_oi;
  pc_on_mar_offset_bus <= pc_on_mar_offset_bus_oi;
  dbus_on_mar_offset_bus <= dbus_on_mar_offset_bus_oi;
  pc_offset_on_dbus <= pc_offset_on_dbus_oi;
  obus_on_dbus <= obus_on_dbus_oi;
  databus_on_dbus <= databus_on_dbus_oi;
  mar_on_adbus <= mar_on_adbus_oi;
  dbus_on_databus <= dbus_on_databus_oi;
  arith_shift_left <= arith_shift_left_oi;
  arith_shift_right <= arith_shift_right_oi;
  read_mem <= read_mem_oi;
  write_mem <= write_mem_oi;
  alu_code <= qit_vector (alu_code_oi);
```

- Make assignments to do type conversion

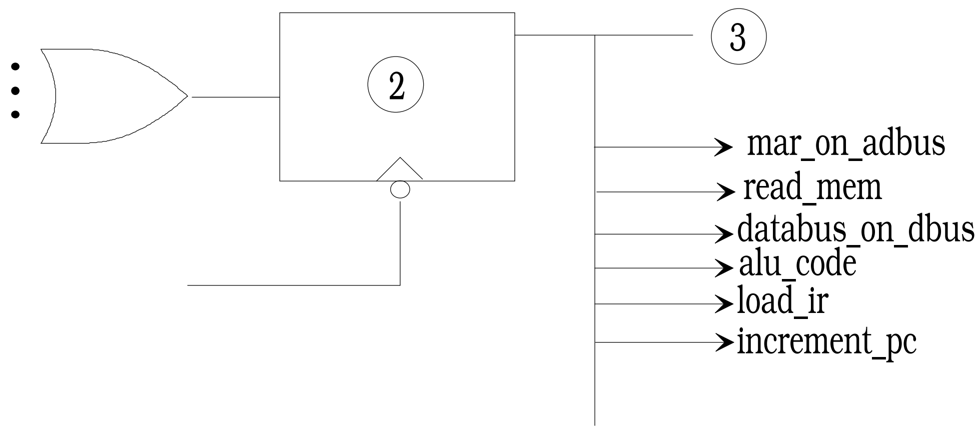
Controller.state_descriptions




```
s1: BLOCK (s(1) = '1')
BEGIN -- start of fetch
  -- pc to mar
  pc_on_mar_page_bus_oi <= GUARDED '1';
  pc_on_mar_offset_bus_oi <= GUARDED '1';
  load_page_mar_oi <= GUARDED '1';
  load_offset_mar_oi <= GUARDED '1';
  -- reset pc if interrupt
  reset_pc_oi <= GUARDED '1' WHEN interrupt = '1' ELSE '0';
  -- goto 2 if interrupt is off
  ck: BLOCK ( (clk = '0' AND NOT clk'STABLE) AND GUARD )
  BEGIN
    s(1) <= GUARDED '1' WHEN interrupt = '1' ELSE '0';
    s(2) <= GUARDED '1' WHEN interrupt /= '1' ELSE '0';
  END BLOCK ck;
END BLOCK s1;
```

- Start to fetch in state 1
- Transfer *pc* to *mar* via *mar_bus*
- Stay in state 1 if interrupted

Controller.state_descriptions




```
s2: BLOCK (s(2) = '1')
BEGIN -- fetching continues
  -- read memory into ir
  mar_on_adbus_oi <= GUARDED '1';
  read_mem_oi <= GUARDED '1' AFTER read_delay;
  databus_on_dbus_oi <= GUARDED '1';
  alu_code_oi <= GUARDED ored_qit_vector (a_input);
  load_ir_oi <= GUARDED '1';
  -- increment pc
  increment_pc_oi <= GUARDED '1';
  -- goto 3
  ck: BLOCK ( (clk = '0' AND NOT clk'STABLE) AND GUARD )
  BEGIN
    s(3) <= GUARDED '1';
  END BLOCK ck;
END BLOCK s2;
```

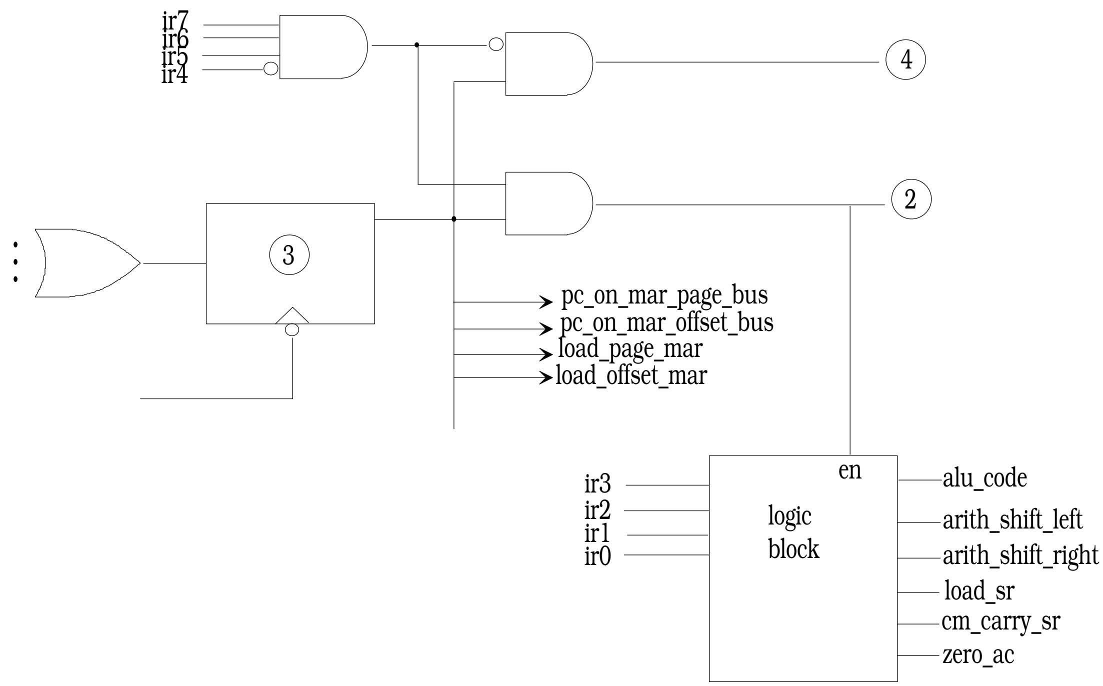
- Continue to fetch instruction
- Read memory into *ir*

Controller.state_descriptions

```
s3: BLOCK (s(3) = '1')
BEGIN
  -- pc to mar, for next read
  pc_on_mar_page_bus_oi <= GUARDED '1';
  pc_on_mar_offset_bus_oi <= GUARDED '1';
  load_page_mar_oi <= GUARDED '1';
  load_offset_mar_oi <= GUARDED '1';
  -- goto 4 if not single byte instruction
  ck: BLOCK ( (clk = '0' AND NOT clk'STABLE) AND GUARD )
  BEGIN
    s(4) <= GUARDED '1' WHEN ir_lines (7 DOWNTO 4) /= "1110" ELSE '0';
  END BLOCK ck;
  -- perform single byte instructions
  sb: BLOCK ( (ir_lines (7 DOWNTO 4) = "1110") AND GUARD)
  BEGIN
    alu_code_oi <= GUARDED
      ored_qit_vector (b_compl) WHEN ir_lines (1) = '1' ELSE
      ored_qit_vector (b_input);
    arith_shift_left_oi <= GUARDED
      '1' WHEN ir_lines (3 DOWNTO 0) = "1000" ELSE '0';
    arith_shift_right_oi <= GUARDED
      '1' WHEN ir_lines (3 DOWNTO 0) = "1001" ELSE '0';
    load_sr_oi <= GUARDED
      '1' WHEN ( ir_lines (3) = '1' OR ir_lines (1) = '1' ) ELSE '0';
    cm_carry_sr_oi <= GUARDED '1' WHEN ir_lines (2) = '1' ELSE '0';
    load_ac_oi <= GUARDED
      '1' WHEN ( ir_lines (3) = '1' OR ir_lines (1) = '1' ) ELSE '0';
    zero_ac_oi <= GUARDED
      '1' WHEN ( ir_lines (3) = '0' AND ir_lines (0) = '1' ) ELSE '0';
    ck: BLOCK ( (clk = '0' AND NOT clk'STABLE) AND GUARD )
    BEGIN
      s(2) <= GUARDED '1';
    END BLOCK ck;
  END BLOCK sb;
END BLOCK s3;
```

- For two byte instructions:
  Prepare for address fetch, goto state 4
- For single byte instructions:
  Prepare for next instruction fetch, complete current instruction

Controller.state_descriptions



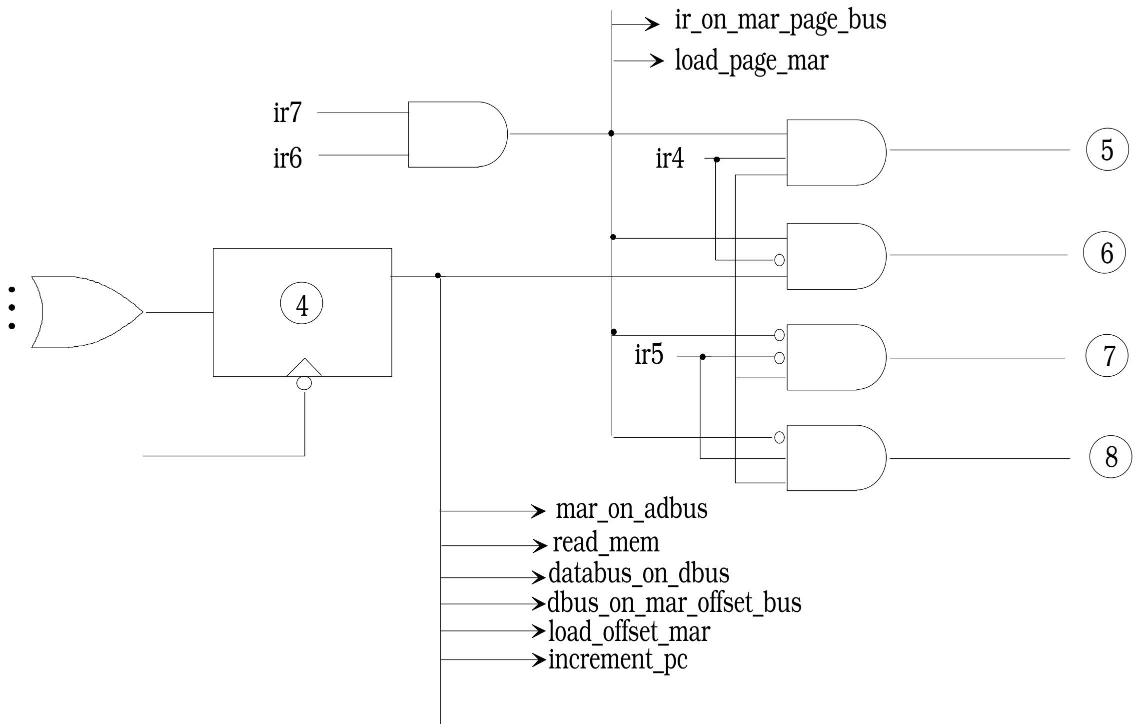
- Hardware for state3
- Logic block is enabled to do *single_byte* instructions
- *pc* to *mar* transfer takes place

Controller.state_descriptions

```
s4: BLOCK (s(4) = '1')
BEGIN -- page from ir, and offset from next memory makeup 12-bit address
  -- read memory into mar offset
  mar_on_adbus_oi <= GUARDED '1';
  read_mem_oi <= GUARDED '1' AFTER read_delay;
  databus_on_dbus_oi <= GUARDED '1';
  dbus_on_mar_offset_bus_oi <= GUARDED '1';
  load_offset_mar_oi <= GUARDED '1'; -- completed operand (dir/indir) address
  -- page from ir if not branch or jsr
  pg: BLOCK ( (ir_lines (7 DOWNTO 6) /= "11") AND GUARD)
  BEGIN
    ir_on_mar_page_bus_oi <= GUARDED '1';
    load_page_mar_oi <= GUARDED '1';
    -- goto 5 for indirect, 6 for direct
    ck: BLOCK ( (clk = '0' AND NOT clk'STABLE) AND GUARD )
    BEGIN
      s(5) <= GUARDED '1' WHEN ir_lines (4) = '1' ELSE '0';  -- indir
      s(6) <= GUARDED '1' WHEN ir_lines (4) = '0' ELSE '0';  -- direct
    END BLOCK ck;
  END BLOCK pg;
  -- keep page in mar_page if jms or bra (same-page instructions)
  sp: BLOCK ( (ir_lines (7 DOWNTO 6) = "11") AND GUARD)
  BEGIN
    -- fot 7 for jsr, 9 for bra
    ck: BLOCK ( (clk = '0' AND NOT clk'STABLE) AND GUARD )
    BEGIN
      s(7) <= GUARDED '1' WHEN ir_lines (5) = '0' ELSE '0';  -- jsr
      s(9) <= GUARDED '1' WHEN ir_lines (5) = '1' ELSE '0';  -- bra
    END BLOCK ck;
  END BLOCK sp;
  -- increment pc
  increment_pc_oi <= GUARDED '1';
END BLOCK s4;
```

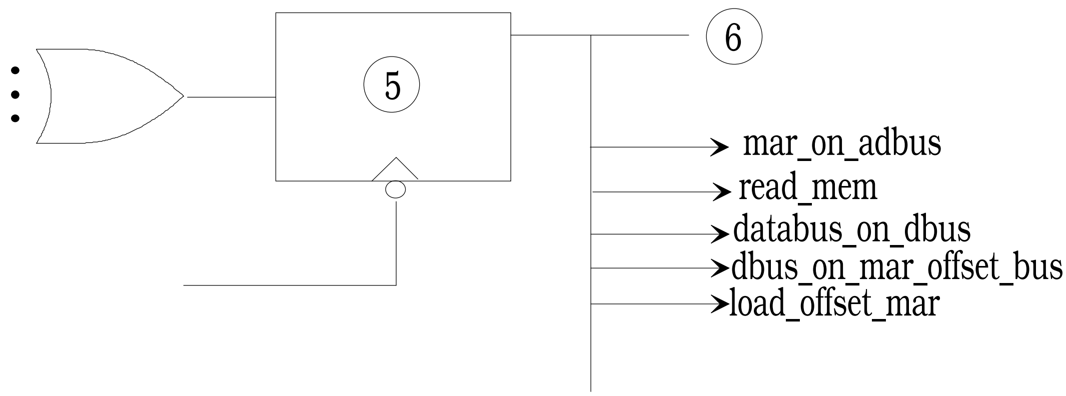
- Get operand for page and full instructions
- For full address : use page from *ir*, *offset* from *mar*
- For page address: keep *mar* page unchanged

Controller.state_descriptions



- Hardware for state 4
- *ir* to *mar* occurs only when *ir*(7:6)=11

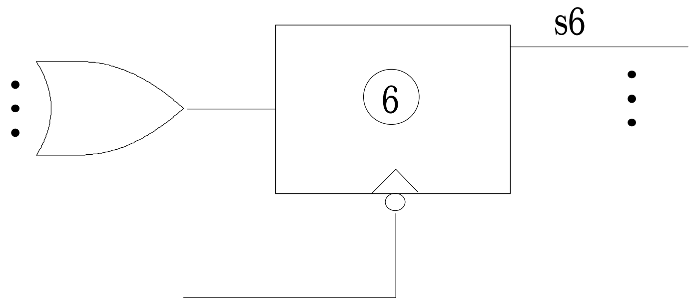
Controller.state_descriptions




```
s5: BLOCK (s(5) = '1')
BEGIN -- indirect addressing
  -- read actual operand from memory into mar offset
  mar_on_adbus_oi <= GUARDED '1';
  read_mem_oi <= GUARDED '1' AFTER read_delay;
  databus_on_dbus_oi <= GUARDED '1';
  dbus_on_mar_offset_bus_oi <= GUARDED '1';
  load_offset_mar_oi <= GUARDED '1';
  -- goto 6
  ck: BLOCK ( (clk = '0' AND NOT clk'STABLE) AND GUARD )
  BEGIN
    s(6) <= GUARDED '1';
  END BLOCK ck;
END BLOCK s5;
```

- Access memory with full address for indirect
- Load byte from memory to *mar* offset

Controller.state_descriptions



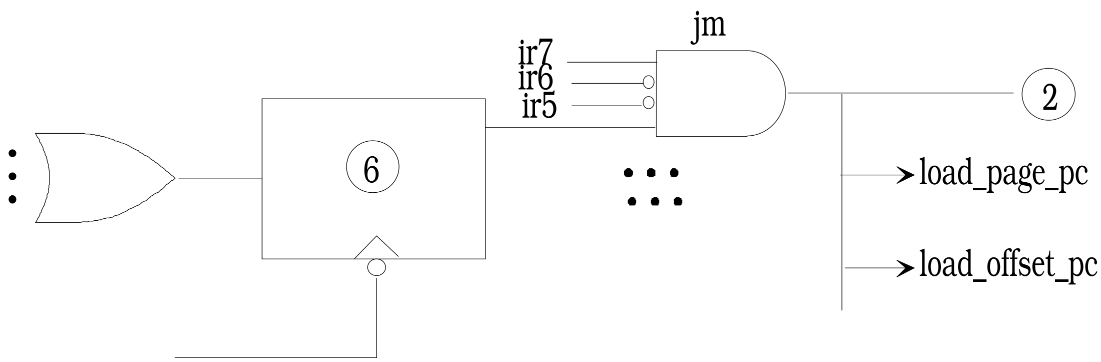

```
s6: BLOCK (s(6) = '1')
BEGIN
  jm : BLOCK ( (ir_lines (7 DOWNTO 5) = "100" ) AND GUARD
  BEGIN
  ... perform jmp
  END BLOCK jm;

  st: BLOCK ( (ir_lines (7 DOWNTO 5) = "101") AND GUARD)
  BEGIN
   ... perform sta
  END BLOCK st;

  rd: BLOCK ( (ir_lines (7) = '0') AND GUARD)
  BEGIN
   ... perform lda, and, add, sub
   END BLOCK rd;
END BLOCK s6;
```

- Full address instructions are handled here
- Three separate blocks for
    1) *jmp*
    2) *sta*
    3) *lda*, *and*, *add*, *sub*

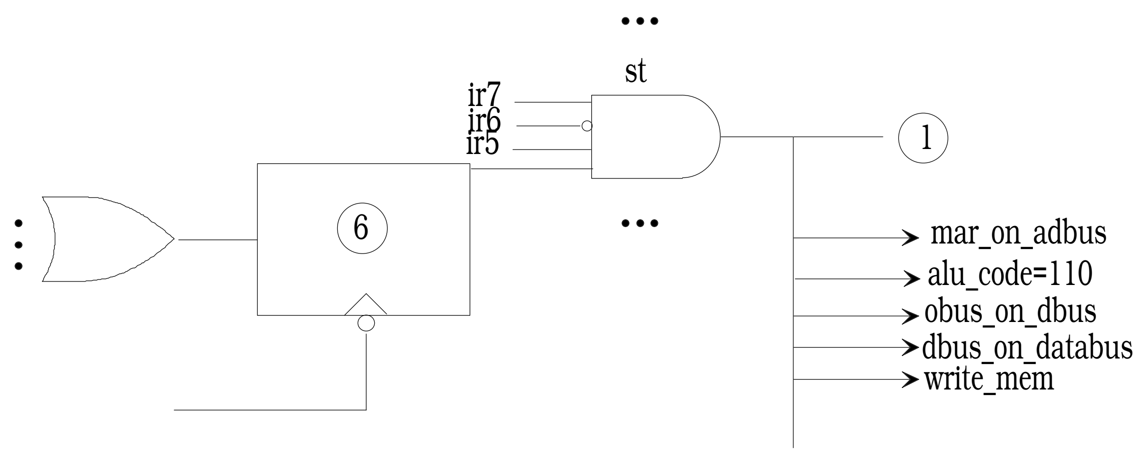
Controller.state_descriptions




```
s6: BLOCK (s(6) = '1')  BEGIN
  jm: BLOCK ( (ir_lines (7 DOWNTO 5) = "100") AND GUARD)
  BEGIN
    load_page_pc_oi <= GUARDED '1';
    load_offset_pc_oi <= GUARDED '1';
    -- goto 2
    ck: BLOCK ( (clk = '0' AND NOT clk'STABLE) AND GUARD )
    BEGIN
      s(2) <= GUARDED '1';
    END BLOCK ck;
  END BLOCK jm;
  . . .
  . . .
```

- Load *pc* with *mar* for *jmp*

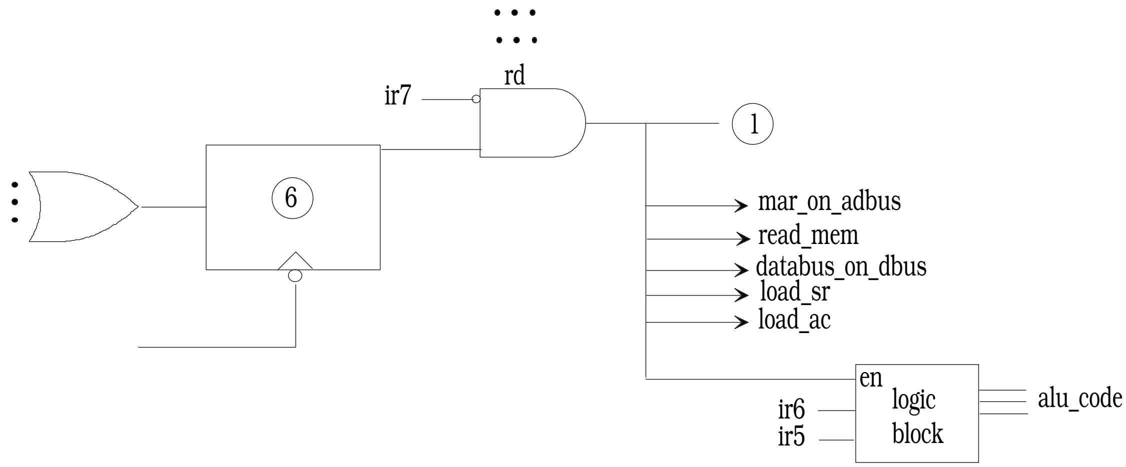
Controller.state_descriptions




```
s6: BLOCK (s(6) = '1')  BEGIN
 . . .
  st: BLOCK ( (ir_lines (7 DOWNTO 5) = "101") AND GUARD)
  BEGIN
    -- mar on adbus, ac on databus, write to memory
    mar_on_adbus_oi <= GUARDED '1';
    alu_code_oi <= GUARDED ored_qit_vector (b_input);
    obus_on_dbus_oi <= GUARDED '1';
    dbus_on_databus_oi <= GUARDED '1';
    write_mem_oi <= GUARDED '1' AFTER write_delay;
    -- goto 1
    ck: BLOCK ( (clk = '0' AND NOT clk'STABLE) AND GUARD )
    BEGIN
      s(1) <= GUARDED '1';
    END BLOCK ck;
  END BLOCK st;
  . . .
```

- For *sta*, write *ac* to memory via *alu*
- Assume writing is completed in one clock

Controller.state_descriptions

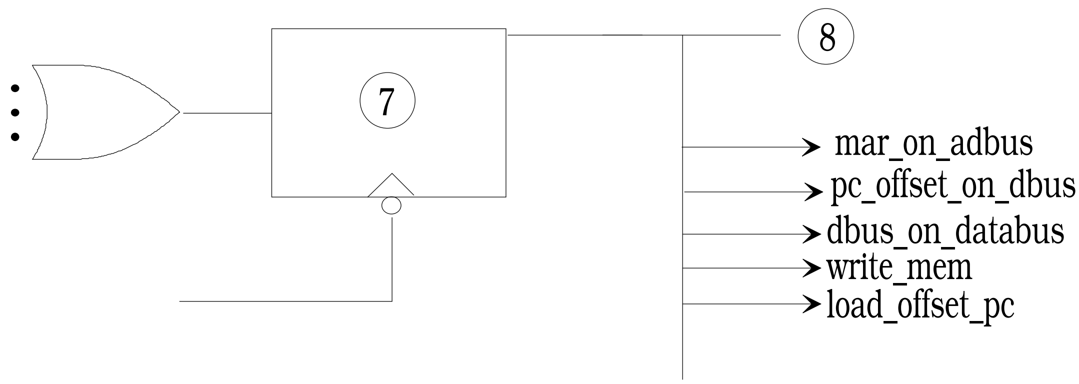


s6:

```
BLOCK (s(6) = '1')  BEGIN
   . . .
   . . .
   rd: BLOCK ( (ir_lines (7) = '0') AND GUARD)
   BEGIN
     -- mar on adbus, read memory for operand, perform operation
     mar_on_adbus_oi <= GUARDED '1';
     read_mem_oi <= GUARDED '1' AFTER read_delay;
     databus_on_dbus_oi <= GUARDED '1';
     WITH ir_lines (6 DOWNTO 5) SELECT
       alu_code_oi <= GUARDED
          ored_qit_vector (a_input) WHEN "00",
          ored_qit_vector (a_and_b) WHEN "01",
          ored_qit_vector (a_add_b) WHEN "10",
          ored_qit_vector (a_sub_b) WHEN "11",
          ored_qit_vector (b_input) WHEN OTHERS;
     load_sr_oi <= GUARDED '1';
     load_ac_oi <= GUARDED '1';
     -- goto 1
     ck: BLOCK ( (clk = '0' AND NOT clk'STABLE) AND GUARD )
     BEGIN  s(1) <= GUARDED '1';  END BLOCK ck;
   END BLOCK rd;
 END BLOCK s6;
```

- Handle *lda*, *and*, *add*, *sub* instructions
- Perform operation while operand is on *databus*

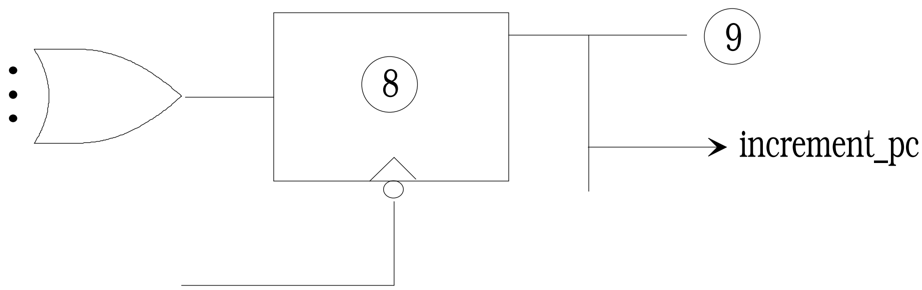
Controller.state_descriptions




```
s7: BLOCK (s(7) = '1')
BEGIN  -- jsr
  -- write pc offset to top of subroutine
  mar_on_adbus_oi <= GUARDED '1';
  pc_offset_on_dbus_oi <= GUARDED '1';
  dbus_on_databus_oi <= GUARDED '1';
  write_mem_oi <= GUARDED '1' AFTER write_delay;
  -- address of subroutine to pc
  load_offset_pc_oi <= GUARDED '1';
  -- goto 8
  ck: BLOCK ( (clk = '0' AND NOT clk'STABLE) AND GUARD )
  BEGIN
    s(8) <= GUARDED '1';
  END BLOCK ck;
END BLOCK s7;
```

- Handle *jsr* in this and next state
- Write *pc* to memory at present *mar* location
- Write *mar* to *pc*

Controller.state_descriptions

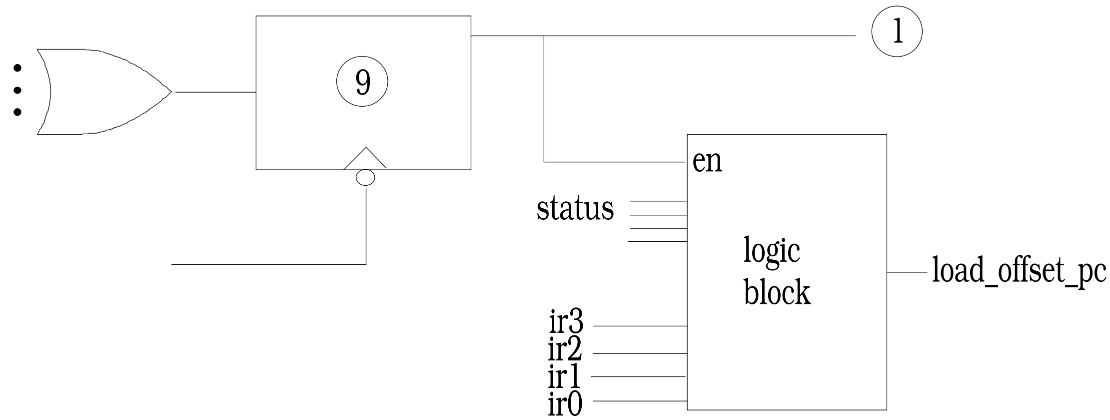



```
s8: BLOCK (s(8) = '1')
BEGIN
   -- increment pc
  increment_pc_oi <= GUARDED '1';
  -- goto 1
  ck: BLOCK ( (clk = '0' AND NOT clk'STABLE) AND GUARD )
  BEGIN
    s(1) <= GUARDED '1';
  END BLOCK ck;
END BLOCK s8;
```

- Complete *jsr*
- Top of subroutine contains last location fetched +1
- *pc* contains top of subroutine
- Increment *pc* to begin first instruction

© Zainalabedin Navabi, 1994

Controller.state_descriptions




```
s9: BLOCK (s(9) = '1')
BEGIN
  load_offset_pc_oi <= GUARDED
    '1' WHEN (status AND ir_lines (3 DOWNTO 0)) /= "0000" ELSE '0';
  -- goto 1
  ck: BLOCK ( (clk = '0' AND NOT clk'STABLE) AND GUARD )
  BEGIN
    s(1) <= GUARDED '1';
  END BLOCK ck;
END BLOCK s9;
```

- Logic block decides branch condition
- Load *pc* if branch condition is true
- Go to 1 for next fetch

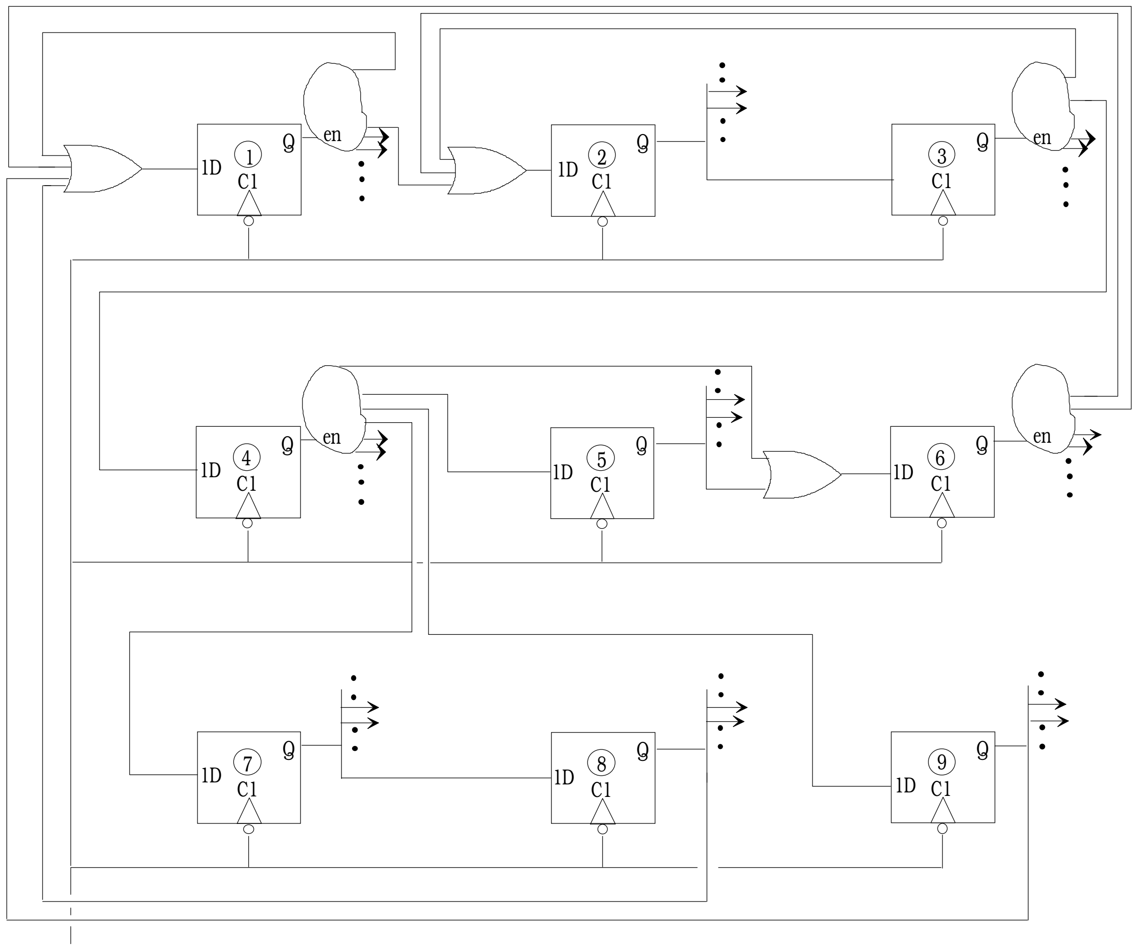
Controller.state_descriptions

```
  ck: BLOCK ( clk = '0' AND NOT clk'STABLE )
  BEGIN
    s (9 DOWNTO 1) <= GUARDED "000000000";
  END BLOCK ck;

END dataflow;
```

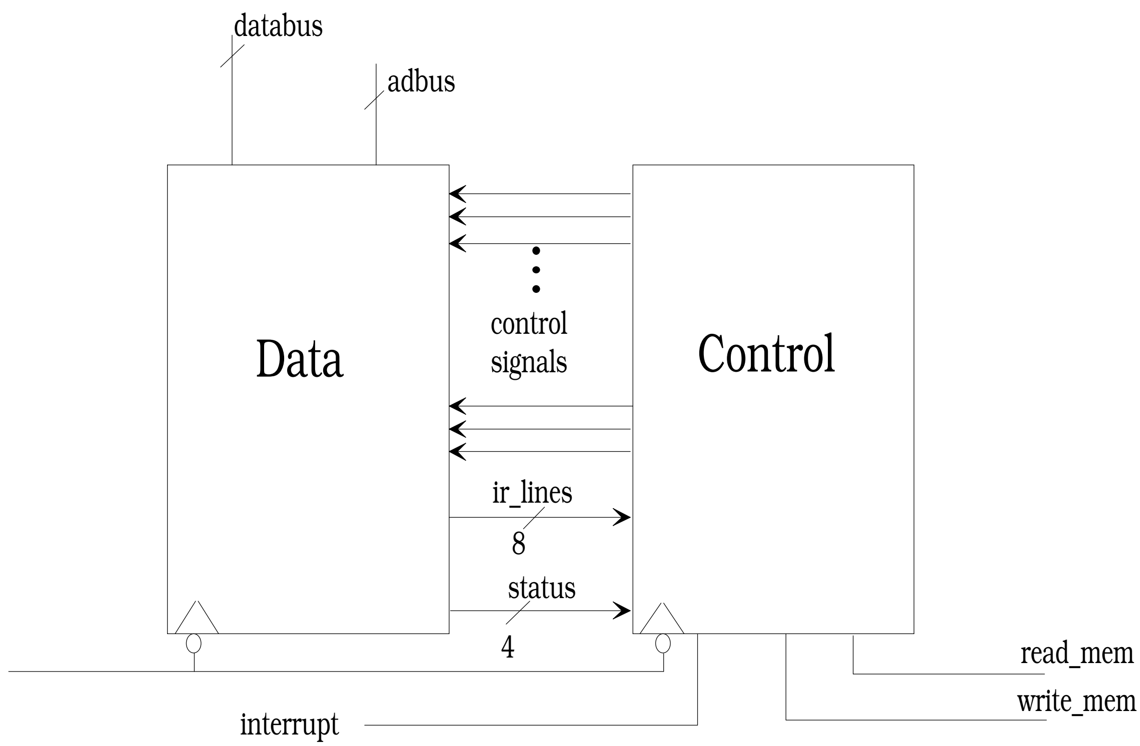
- Required resetting
- All state are set to 0 to activate resolution function
- REGISTER kind requires a driver for activating resolution function

Controller.state_descriptions



- Complete control unit
- Wire individual control flip-flops
- Oring is done at inputs of states when branching is done to them

Controller.state_descriptions




```
ENTITY par_central_processing_unit IS
  PORT (clk : IN qit;
        interrupt : IN qit;
        read_mem, write_mem : OUT qit;
        databus : INOUT wired_byte BUS := "ZZZZZZZZ"; adbus : OUT twelve
       );
END par_central_processing_unit;
```

- Complete CPU wires data and control

Controller.state_descriptions

```
ARCHITECTURE dataflow OF par_central_processing_unit IS
  COMPONENT par_data_path
  PORT (databus : INOUT wired_byte;  adbus : OUT twelve;
        clk : IN qit;
        load_ac, zero_ac,
        . . .
        alu_code : IN qit_vector (2 DOWNTO 0);
        ir_lines : OUT byte; status : OUT nibble
       );
  END COMPONENT;
  FOR data: par_data_path USE ENTITY WORK.par_data_path (structural);
  --
  COMPONENT par_control_unit
  PORT (clk : IN qit;
        load_ac, zero_ac,
        . . .
        alu_code : OUT qit_vector (2 DOWNTO 0);
        ir_lines : IN byte; status : IN nibble;
        read_mem, write_mem : OUT qit; interrupt : IN qit
       );
  END COMPONENT;
  FOR ctrl: par_control_unit USE ENTITY WORK.par_control_unit (dataflow);
  --
  SIGNAL load_ac, zero_ac,
              . . .
  SIGNAL alu_code : qit_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,
         . . .
         alu_code,
         ir_lines, status
        );
  ctrl: par_control_unit PORT MAP
        (clk,
         load_ac, zero_ac,
         . . .
         alu_code,
         ir_lines, status,
         read_mem, write_mem, interrupt
        );
END dataflow;
```

- Data and control declarations
- Data and control wiring

Controller.state_descriptions

```
ARCHITECTURE input_output OF parwan_tester IS
  COMPONENT parwan PORT (clk : IN qit;  interrupt : IN qit;
                        read_mem, write_mem : OUT qit;
                        databus : INOUT wired_byte; adbus : OUT twelve );
  END COMPONENT;
  SIGNAL clock, interrupt, read, write : qit;
  SIGNAL data : wired_byte := "ZZZZZZZZ";
SIGNAL adr : INTEGER := 0;
  SIGNAL address : twelve;
  TYPE byte_memory IS ARRAY ( INTEGER RANGE <> ) OF byte;
BEGIN
  int : interrupt <= '1', '0' AFTER 4500 NS;
  clk : clock <= NOT clock AFTER 1 US WHEN NOW <= 140 US ELSE clock;
  cpu : parwan PORT MAP (clock, interrupt, read, write, data, address);
  mem : PROCESS
    VARIABLE memory : byte_memory ( 0 TO 63 ) :=
     ("00000000", "00011000", "10100000", "00011001", --lda 24, sta 25
          VARIABLE ia : INTEGER;
  BEGIN
    WAIT ON read, write;
    qit2int (address, ia);
    IF read = '1' THEN
      IF ia >= 64 THEN
        data <= "ZZZZZZZZ";
      ELSE
        data <= wired_byte ( memory (ia) );
      END IF;
      WAIT UNTIL read = '0';
      data <= "ZZZZZZZZ";
    ELSIF write = '1' THEN
      IF ia < 64 THEN
        memory (ia) := byte ( data );
      END IF;
      WAIT UNTIL write = '0';
    END IF;
  END PROCESS mem;
END input_output;
```

- A simple testbench
- Include CPU instantiation, a short memory, and read/write handshaking

Controller.testing_Parwan

```
ENTITY parwan_tester IS
END parwan_tester;

ARCHITECTURE input_output OF parwan_tester IS
  COMPONENT parwan PORT (clk : IN qit;  interrupt : IN qit;
                        read_mem, write_mem : OUT qit;
                        databus : INOUT wired_byte; adbus : OUT twelve );
  END COMPONENT;
  SIGNAL clock, interrupt, read, write : qit;
  SIGNAL data : wired_byte := "ZZZZZZZZ";
  SIGNAL address : twelve;
  TYPE byte_memory IS ARRAY ( INTEGER RANGE <> ) OF byte;
  . . .
```

- Parwan tester applies data to Parwan buses
- Component is declared, binding will be done by configuration declaration
- Hold data normally at *z* (High Impedance)

Controller.testing_Parwan

```
ARCHITECTURE input_output OF parwan_tester IS

  . . .

BEGIN
 int : interrupt <= '1', '0' AFTER 4500 NS;
 clk : clock <= NOT clock AFTER 1 US WHEN NOW <= 140 US ELSE clock;
 cpu : parwan PORT MAP (clock, interrupt, read, write, data, address);
 mem : PROCESS
   VARIABLE memory : byte_memory ( 0 TO 63 ) :=
    ("00000000", "00011000", "10100000", "00011001", --lda 24, sta 25
     "00100000", "00011010", "01000000", "00011011", --and 26, add 27
     "11100010", "11101001", "01100000", "00011100", --cac, asr, sub 28
     "00010000", "00011101", "11000000", "00100100", --lda i 29, jsr 36
     "11101000", "11100000", "10000000", "00100000", --asl, nop, jmp 32
     "00000000", "00000000", "00000000", "00000000",
     "01011100", "00000000", "01110000", "00010010", --(24, 25, 26, 27)
     "00001100", "00011111", "00000000", "01011010", --(28, 29, 30, 31)
     "10000000", "00010010", "00000000", "00000000", --jmp 18
     "00000000", "11100010", "10010000", "00100100", --   , cma, jmp i 36
     "00000000", "00000000", "00000000", "00000000",
     "00000000", "00000000", "00000000", "00000000",
     "00000000", "00000000", "00000000", "00000000",
     "00000000", "00000000", "00000000", "00000000",
     "00000000", "00000000", "00000000", "00000000",
     "00000000", "00000000", "00000000", "00000000" );
  . . .
END input_output;
```

- Produce test waveforms on interrupt and clock signals
- Testing is done by modeling memory read and write operations
- A single process assigns values from memory to *databus*
- Same process handles memory write

Controller.testing_Parwan

```
ARCHITECTURE input_output OF parwan_tester IS
  . . .
BEGIN
  . . .
  mem : PROCESS
    VARIABLE memory : byte_memory ( 0 TO 63 ) :=
      ("00000000", "00011000", "10100000", "00011001", --lda 24, sta 25
       "00100000", "00011010", "01000000", "00011011", --and 26, add 27
       "11100010", "11101001", "01100000", "00011100", --cac, asr, sub 28
       "00010000", "00011101", "11000000", "00100100", --lda i 29, jsr 36
       "11101000", "11100000", "10000000", "00100000", --asl, nop, jmp 32
       "00000000", "00000000", "00000000", "00000000",
       "01011100", "00000000", "01110000", "00010010", --(24, 25, 26, 27)
       "00001100", "00011111", "00000000", "01011010", --(28, 29, 30, 31)
       "10000000", "00010010", "00000000", "00000000", --jmp 18
       "00000000", "11100010", "10010000", "00100100", --    , cma, jmp i 36
       "00000000", "00000000", "00000000", "00000000",
       "00000000", "00000000", "00000000", "00000000",
       "00000000", "00000000", "00000000", "00000000",
       "00000000", "00000000", "00000000", "00000000",
       "00000000", "00000000", "00000000", "00000000",
       "00000000", "00000000", "00000000", "00000000" );
  BEGIN
    WAIT ON read, write; qit2int (address, ia);
    IF read = '1' THEN
    IF ia >= 64 THEN       data <= "ZZZZZZZZ";  ELSE
        data <= wired_byte ( memory (ia) );
      END IF;
      WAIT UNTIL read = '0';  data <= "ZZZZZZZZ";
    ELSIF write = '1' THEN    IF ia < 64 THEN   memory (ia) := byte ( data );
      END IF;
      WAIT UNTIL write = '0';
    END IF;
  END PROCESS mem;
END input_output;
```

- Wait for falling edge of clock to start reading or writing
- A 20 NS wait makes sure data is stable
- When reading from memory, memory model holds data on *databus* til the next clock edge

Controller.testing_Parwan

```
CONFIGURATION behavior OF parwan_tester IS
  FOR input_output
    FOR cpu : parwan
      USE ENTITY behavioral.par_central_processing_unit(behavioral);
    END FOR;
  END FOR;
END behavior;
```

(a)

```
CONFIGURATION dataflow OF parwan_tester IS
  FOR input_output
    FOR cpu : parwan
      USE ENTITY par_dataflow.par_central_processing_unit(dataflow);
    END FOR;
  END FOR;
END dataflow;
```

(b)

- Configuration declarations configure
  Test_bench for testing
  Behavioral or dataflow Parwan
- Can modify testbench to read and write from text I/O files

Controller.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: undefined
OFFENDING COMMAND: n

STACK:

0
-1.0