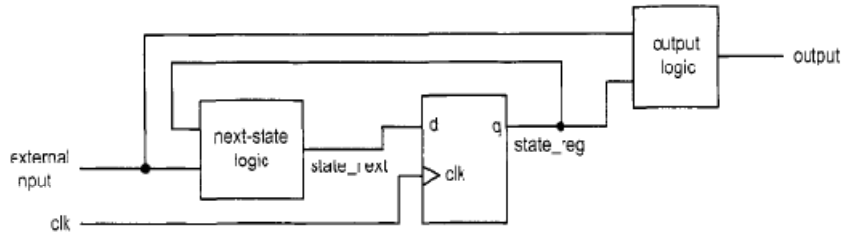
Embedded Systems

DEPARTMENT OF ELECTRICAL ENGINEERING

IIT ROORKEE, ROORKEE

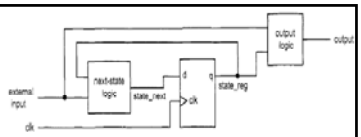
REGULAR SEQUENTIAL CIRCUIT:
Synchronous Systems

- The *synchronous design methodology* is the most commonly used practice in designing a sequential circuit. In this methodology, all storage elements are controlled (i.e., synchronized) by a global clock signal and the data is sampled and stored at the rising or falling edge of the clock signal.
- It allows to separate the storage components from the circuit and greatly simplifies the development process.



Block diagram of a synchronous system.

REGULAR SEQUENTIAL CIRCUIT:
Synchronous Systems



- *State register:* A collection of D FFs controlled by the same clock signal
- *Next-state logic:* Combinational logic that uses the external input and internal state (o/p of register) to determine the new value of the register
- *Output logic:* Combinational logic that generates the output signal

Max. Operational operating frequency

- The timing of a sequential circuit is characterized by fmax ,which specifies how fast the circuit can operate.
- To ensure correct operation, the next value must be generated and stabilized within Tclock.

3

REGULAR SEQUENTIAL CIRCUIT:
Synchronous Systems

- The Minimal Clock Period:

$$Tclock = Tcq + Tcomb + Tsetup$$

Tcq= clock-to-q delay, *Tsetup* = setup time,

Tcomb = the maximal propagation delay of next-state logic

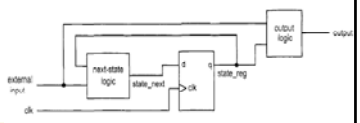
Xilinx Specific: During synthesis, Xilinx ISC software analyze the synthesized circuit and show *fmax* in a report. To specify desired operating frequency as a synthesis constraint, the following lines can be added to the constraint file:

Eg. For 50-MHz clock prototyping board

```
NET "clk" TNM-NET = "clk";
TIMESPEC "TS-clk" = PERIOD "clk" 20 ns HIGH 50 % ;
```

4

REGULAR SEQUENTIAL CIRCUIT: Synchronous Systems



- Code Development: follows the basic block diagram. First separate the memory component (i.e., the register) from the system. Once the register is isolated, the remaining portion is a pure combinational circuit, and the coding and analysis schemes apply as discussed earlier.

- Since our development process separates the register and the combinational circuit, following components are sufficient for most of the designs

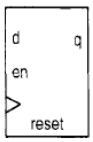
  - DFF
    - D FF without asynchronous reset
    - D FF with asynchronous reset
    - D FF with synchronous enable
  - Register
  - Register file

5

REGULAR SEQUENTIAL CIRCUIT: Components

- *DFF with synchronous enable:*

  en signal is examined only at the rising edge of the clock and thus is synchronous. If *en* not asserted, the FF keeps its previous value.

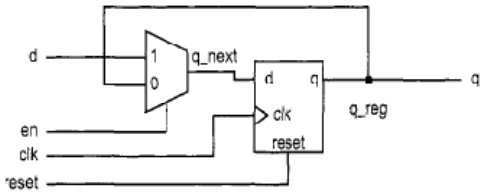


reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↓	0	q
0	↓	1	d

```
library ieee;
use ieee.std_logic_1164.all;
entity d_ff_en is
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end d_ff_en;

architecture arch of d_ff_en is
begin
   process(clk,reset)
   begin
      if (reset='1') then
         q <='0';
      elsif (clk'event and clk='1') then
         if (en='1') then
            q <= d;
         end if;
      end if;
   end process;
end arch;
```

REGULAR SEQUENTIAL CIRCUIT: Components

- Since the enable signal is synchronous, this circuit can be constructed by a regular D FF and simple next-state logic.




```
architecture two_seg_arch of d_ff_en is
   signal r_reg, r_next: std_logic;
begin
   -- D FF
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <='0';
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   r_next <= d when en ='1' else
             r_reg;
   -- output logic
   q <= r_reg;
end two_seg_arch;
```

D FF with synchronous enable: Two-segment coding style

7

REGULAR SEQUENTIAL CIRCUIT: Components

- Register is a collection of D FFs that are controlled by the same *clock* and *reset* signals.
- Code is identical to a DFF except that the array data type, *std_logic_vector*, is needed for the relevant input and output signals.

```
library ieee;
use ieee.std_logic_1164.all;
entity reg_reset is
   port(
      clk, reset: in std_logic;
      d: in std_logic_vector(7 downto 0);
      q: out std_logic_vector(7 downto 0)
   );
end reg_reset;
architecture arch of reg_reset is
begin
   process(clk,reset)
   begin
      if (reset='1') then
         q <=(others=>'0');
      elsif (clk'event and clk='1') then
         q <= d;
      end if;
   end process;
end arch;
```

8-bit register with asynchronous reset

8

REGULAR SEQUENTIAL CIRCUIT: Components

- Register file: A collection of registers with one input port and one or more output ports; generally used as fast, temporary storage.
  - w-addr : specifies where to store data,
  - r-addr : specifies where to retrieve data
  - Parameter e.g.. : 2^W-by-B register file (In coding defines as generic)
    - Generic W : specifies the number of address bits
    - Generic B specifies the number of bits in a word

Xilinx Specific:

- In a Spartan-3 device, each logic cell contains a D FF with asynchronous reset and synchronous enable. These D FFs basically constitute the register.
- The Spartan-3 device also has block RAM modules, and they can be used for larger storage requirements. These modules can be configured for synchronous operation, and their characteristics are somewhat like a restricted version of the register file.

9

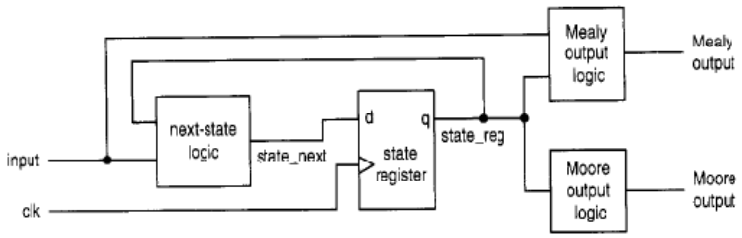
SEQUENTIAL CIRCUIT: Synchronous Systems

- Division of sequential circuits based on the characteristics of the next-state logic,
  - *Regular sequential circuit.* The state transitions exhibit a "regular" pattern, as in a counter or shift register. The next-state logic is constructed primarily by a pre-designed, "regular" component, viz. incrementor or shifter etc.
  - *FSM.* The state transitions do not exhibit a simple, repetitive pattern. The next-state logic is constructed by "random logic" and synthesized from scratch.
  - *FSMD.* It consists of a regular sequential circuit and an FSM; known as a *data path* and a *control path.* It is used to implement an algorithm represented by *register-transfer* (RT) methodology, which describes system operation by a sequence of data transfers and manipulations among registers.

10

Finite State Machine (FSM):

- An FSM is used to model a system that transits among a finite number of internal states. The transitions depend on the current state and external input.
- The main application of an FSM is to act as the controller of a large digital system, which examines the external commands and status and activates proper control signals to control operation of a *data path,* which is usually composed of regular sequential components. (FSMD)
- Both types of output, Moore output and Mealy output, may exist in a complex FSM.



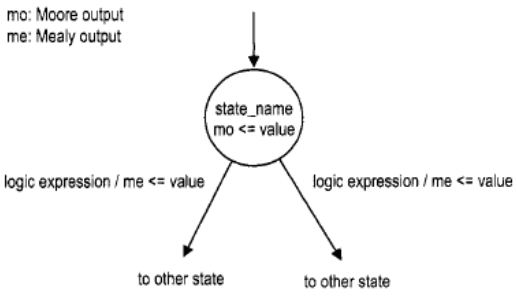
Block diagram of a synchronous FSM

Finite State Machine (FSM):

- FSM Representation
  - *State diagram*
  - *ASM chart* (algorithmic state machine chart),

Both capturing the FSM's input, output, states, and transitions in a graphical representation.

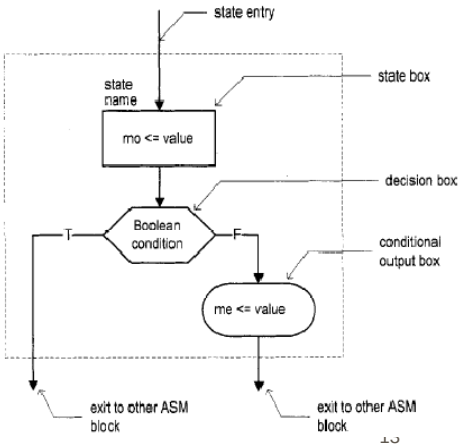
A state diagram is composed of *nodes,* which represent states and are drawn as circles, and annotated *transitional arcs.*



Finite State Machine (FSM):

- An ASM chart is composed of a network of **ASM blocks**. An *ASM block* consists of one *state box* and an optional network of *decision boxes* and *conditional output boxes.*

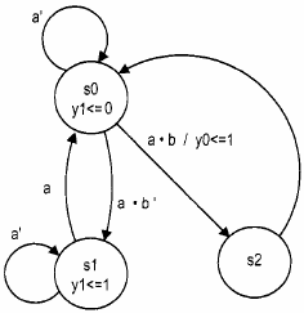
- State box represents a state in an FSM, and the asserted Moore output values are listed inside the box. It has only one exit path.
- Decision box tests the input condition and determines which exit path to take. It has two exit paths, which correspond to the **True** and **False** values of the condition.
- Conditional output box lists asserted Mealy output values and is usually placed after a decision box.



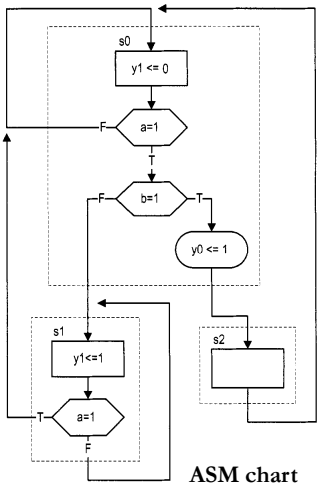
ASM Block

Finite State Machine (FSM):

- A state diagram can be converted to an ASM chart, and vice versa.



State diagram



ASM chart

Finite State Machine (FSM):

- FSM CODE DEVELOPMENT

  First separate the state register and then derive the code for the combinational next-state logic and output logic. The code for the next-state logic follows the flow of a state diagram or ASM chart.
- For clarity, use user's defined *enumerated data type* to represent the FSM's states. e.g.

  type eg-state-type **is** (s0 , s1, s2) ;

  signal state-reg, state-next : eg-state-type;

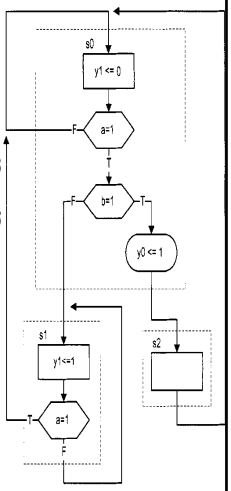
During synthesis, software automatically maps the values in an enumerated data type to binary representations, a process known as *state assignment.*

15

Code of the FSM consists of segments for the state register, next-state logic, Moore output logic, and Mealy output logic

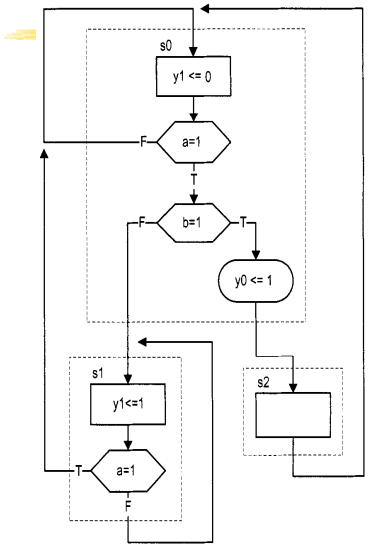
```
library ieee;
use ieee.std_logic_1164.all;
entity fsm_eg is
   port(
      clk, reset: in std_logic;
      a, b: in std_logic;
      y0, y1: out std_logic
   );
end fsm_eg;

architecture mult_seg_arch of fsm_eg is
   type eg_state_type is (s0, s1, s2);
   signal state_reg, state_next: eg_state_type;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= s0;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
```

```
-- next-state logic
process(state_reg,a,b)
begin
   case state_reg is
      when s0 =>
         if a='1' then
            if b='1' then
               state_next <= s2;
            else
               state_next <= s1;
            end if;
         else
            state_next <= s0;
         end if;
      when s1 =>
         if (a='1') then
            state_next <= s0;
         else
            state_next <= s1;
         end if;
      when s2 =>
         state_next <= s0;
   end case;
end process;
```



16

Code of the FSM: Moore output logic, and Mealy output logic




```
-- Moore output logic
process(state_reg)
begin
   case state_reg is
      when s0|s2 =>
         y1 <= '0';
      when s1 =>
         y1 <= '1';
   end case;
end process;
-- Mealy output logic
process(state_reg,a,b)
begin
   case state_reg is
      when s0 =>
         if (a='1') and (b='1') then
           y0 <= '1';
         else
           y0 <= '0';
         end if;
      when s1 | s2 =>
         y0 <= '0';
   end case;
end process;
end mult_seg_arch;
```

Finite State Machine:

- **Xilinx Specific:** Xilinx ISE includes a utility program called *StateCAD,* which allows a user to draw a state diagram in graphical format. The program then converts the state diagram to HDL code.

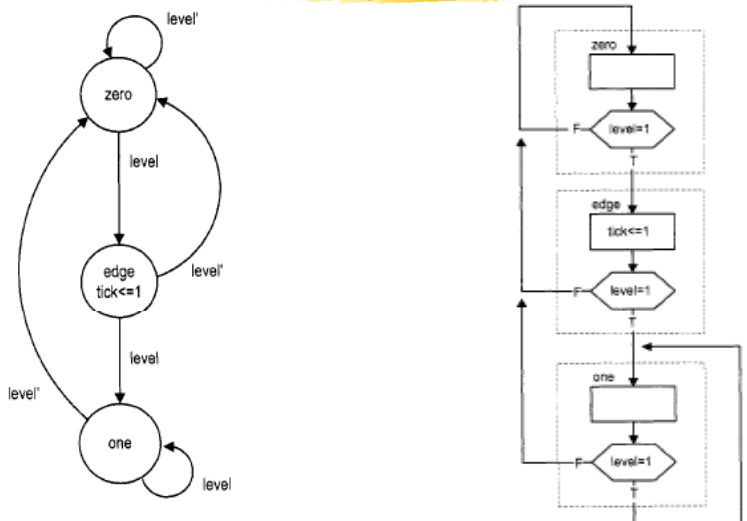
Finite State Machine :

Problem: Design a ‘rising-edge detector’ using Moore machine with following description. Draw the state diagram and ASM chart. The rising-edge detector is a circuit that generates a short, one-clock-cycle pulse (call it a ***tick)*** when the input signal changes from ’0’ to ’1 ’. It is usually used to indicate the onset of a slow time-varying input signal.

Moore-based design: The zero and one states indicate that the input signal has been ’0’ and ’1’ for awhile. The rising edge occurs when the input changes to ’1’ in the zero state. The FSM moves to the edge state and the output, tick , is asserted in this state.

19

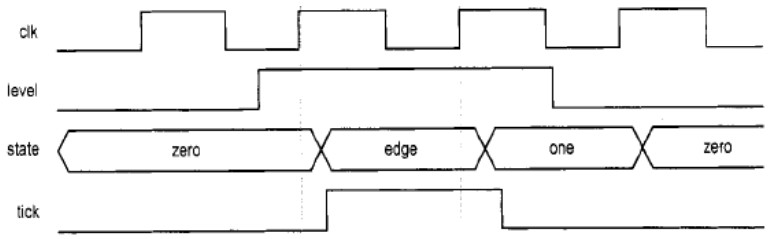
Finite State Machine :



Edge detector based on a Moore machine.

20

Finite State Machine :

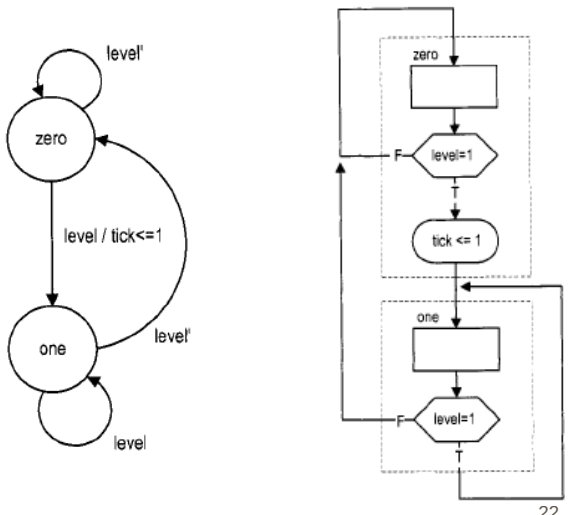


Timing diagram of two edge detectors Moore-based design.

21

Finite State Machine:

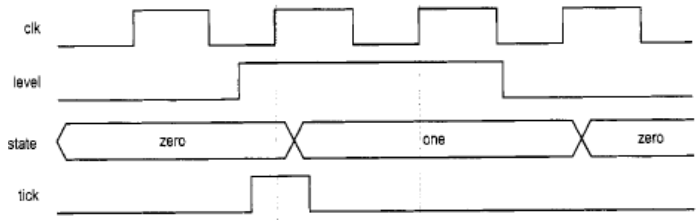
The zero and one states have similar meaning. When the FSM is in the zero state and the input changes to '1', the output is asserted immediately. The FSM moves to the one state at the rising edge of the next clock and the output is deasserted.



Edge detector based on a Mealy machine.

22

Finite State Machine :

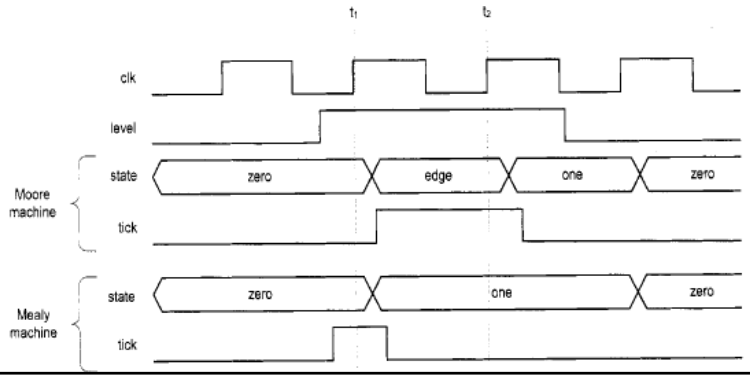


Timing diagram of two edge detectors Mealey-based design

Finite State Machine :

Comparison

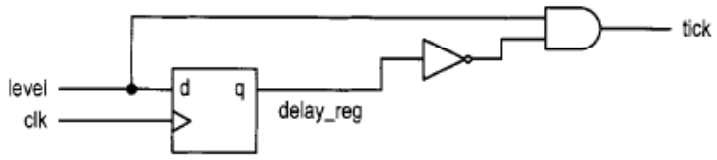
- The Mealy machine-based design requires fewer states and responds faster, but the width of its output may vary and input glitches may be passed to the output.
- The choice between the two designs depends on the subsystem that uses the output signal.
- The Mealy output signal is available for sampling at *t1* , which is one clock cycle faster than the Moore output, which is available at *t2*.



Finite State Machine :

Direct implementation:

The edge detector circuit can be implemented without using an FSM. It can be interpreted that the output is asserted only when the current input is '1' and the previous input, which is stored in the register, is '0'.



25

Finite State Machine with Data path (FSMD):

An FSMD combines an FSM and regular sequential circuits.

- The FSM, which is sometimes known as a *control path,* examines the external commands and status and generates control signals to specify operation of the regular sequential circuits, which are known collectively as a *data path.*
- The FSMD is used to implement systems described by *RT-methodology,* in which the operations are specified as data manipulation and transfer among a collection of registers.

Single RT operation

An RT operation specifies data manipulation and transfer for a single destination register.

rdest ← *f* **(rsrcl, rsrc2, . . . , rsrcn)**

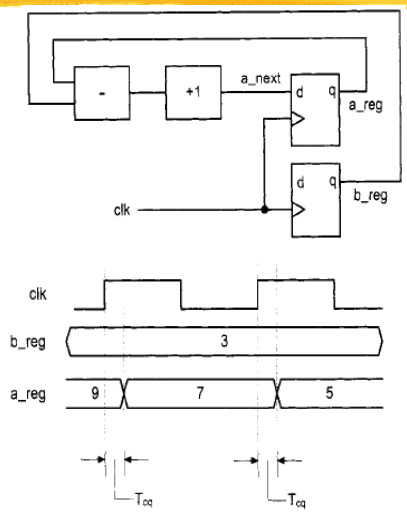
Notation: The contents of the source registers are fed to the *f(.)* function, which is realized by a combinational circuit, and the result is passed to the input of the destination register and stored in the destination register at the next rising edge of the clock.

e.g.: r2 ← r2 >> 3. The r2 register is shifted right three positions and then written back to it self

26

Finite State Machine with Data path (FSMD):

- Implementation of single RT operation:
  - Implemented by constructing a combinational circuit for the *f*(.) function and connecting the input and output of the registers.
  - Example,
  - Operation: a ← a-b+1.
  - function involves: a subtractor and an incrementor.
    - _reg and _next suffixes to represent the input and output of a register.
    - RT operation is synchronized by an embedded clock. The result is not stored to the destination register until the next rising edge of the clock.



Block and timing diagrams of an RT operatio

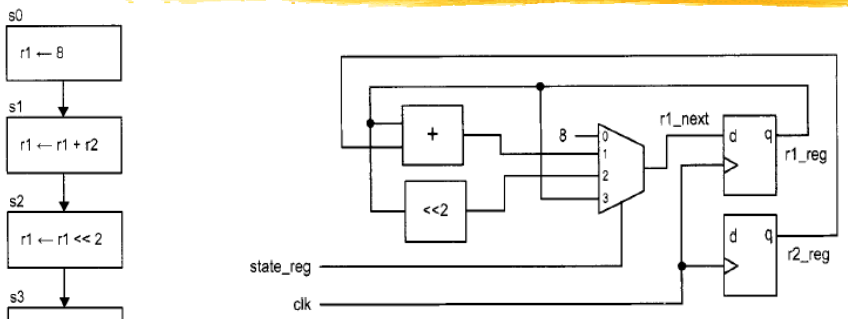
27

Finite State Machine with Data path (FSMD):

- **ASMD chart:**
  - A circuit based on the RT methodology specifies which RT operations should be executed in each step.
  - Since an RT operation is done in a clock-by-clock basis, its timing is similar to a state transition of an FSM. Thus, an FSM is a natural choice to specify the sequencing of an RT algorithm.
  - Extend the ASM chart to incorporate RT operations and call it an *ASMD* (ASM with data path) chart.

28

Finite State Machine with Data path (FSMD):



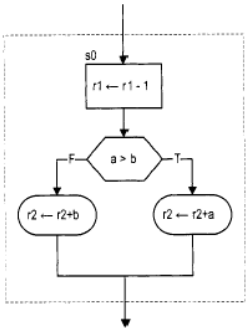
Realization of an ASMD segment.

- Implementing the RT operations of an ASMD chart involves a multiplexing circuit to route the desired next value to the destination register, the above segment is can be implemented by a 4-to-1 multiplexer,.
- The current state (i.e., the output of the state register) of the FSM controls the selection signal of the multiplexer and thus chooses the result of the desired RT operation.

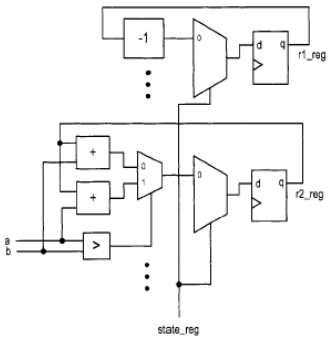
29

Finite State Machine with Data path (FSMD):

- An RT operation specified in a conditional output box,



ASM block



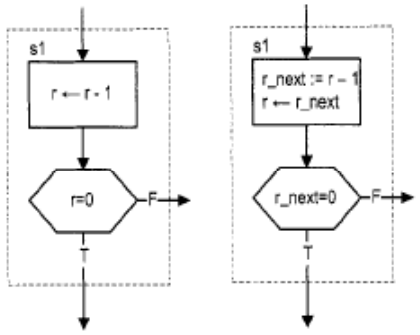
Realization of an RT operation in a conditional output box.

Note that all operations are done in parallel inside an ASMD block.

30

Finite State Machine with Data path (FSMD):

- Decision box with a register: The main difference between flow chart and ASMD is that the RT operation in an ASMD chart is controlled by an embedded clock signal and the destination register is updated *when the FSMD exits the current ASMD block*, but not within the block.



(a) Use old value of r (b) Use new value of r

31

Finite State Machine with Data path (FSMD):

- Block diagram of an FSMD

The conceptual block diagram of an FSMD is divided into a data path and a control path. The data path performs the required RT operations. It consists of

- *Data registers:* store the intermediate computation results
- *Functional units:* perform the functions specified by the RT operations
- *Routing network:* routes data between the storage registers and the functional units

The data path follows the control signal to perform the desired RT operations and generates the internal status signal.

32

Finite State Machine with Data path (FSMD):

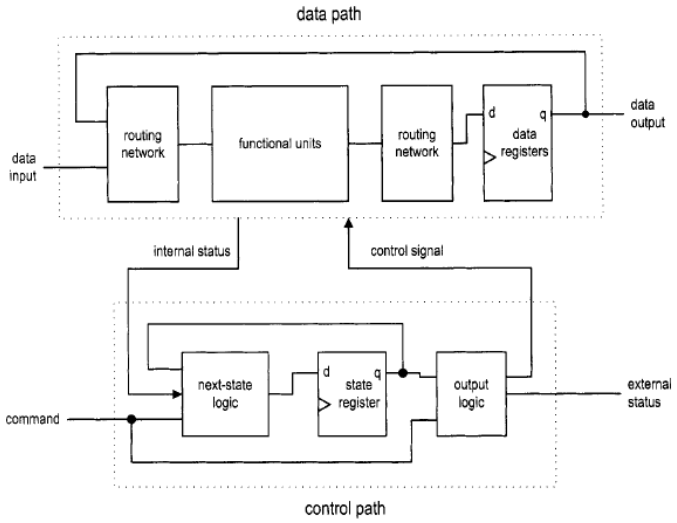
- The control path is an FSM.
  - It contains a state register, next-state logic, and output logic.
  - It uses the external command signal and the data path's status signal as the input and generates the control signal to control the data path operation.
  - The FSM also generates the external status signal to indicate the status of the FSMD operation.

  An FSMD consists of two types of sequential circuits, both circuits are controlled by the same clock, and thus the FSMD is still a synchronous system.
- Code development of an FSMD:

Demonstration example: The slide and pushbutton switches on the prototyping board are mechanical devices. When pressed, the switch may bounce back and forth a few times before settling down. The bounces lead to glitches in the signal. The bounces usually settle within 20 ms. The purpose of a debouncing circuit is to filter out the glitches associated with switch transitions.

33

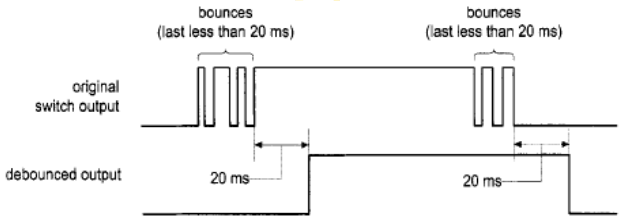
Finite State Machine with Data path (FSMD):



Block diagram of an FSMD.

34

FSMD



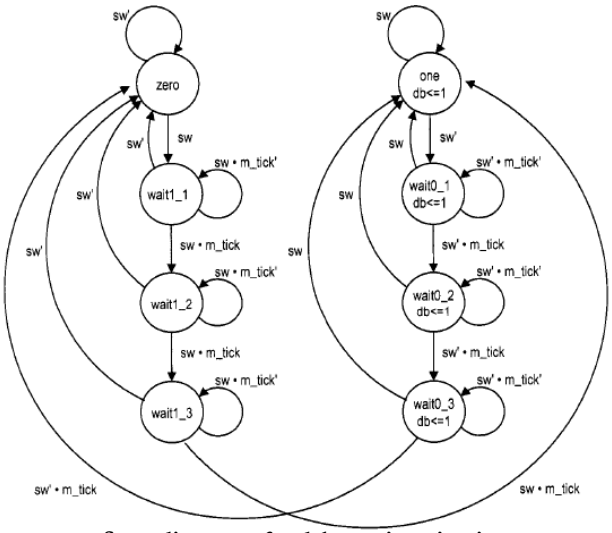
Original and debounced waveforms.

FSM-based Design: It uses a free-running 10-ms timer and an FSM. The timer generates a one-clock-cycle enable tick (the m-tick signal) every 10 ms and the FSM uses this information to keep track of whether the input value is stabilized. FSM ignores the short bounces and changes the value of the debounced output only after the input is stabilized for 20 ms. The zero and one states indicate that the switch input signal, **sw,** has been stabilized with '0' and ' 1 ' values.

35

FSMD

- 10-ms timer is free-running and the m_tick tick can be asserted at any time,
- FSM checks the assertion three times to ensure that the sw signal is stabilized for at least 20 ms (Actually between 20 and 30 ms).



State diagram of a debouncing circuit.

FSMD

```
entity db_fsm is
   port(
      clk, reset: in std_logic;
      sw: in std_logic;
      db: out std_logic
   );
end db_fsm;
```

```
architecture arch of db_fsm is
   constant N: integer:=19;  -- 2^N * 20ns = 10ms
   signal q_reg, q_next: unsigned(N-1 downto 0);
   signal m_tick: std_logic;
   type eg_state_type is (zero,wait1_1,wait1_2,wait1_3,
                          one,wait0_1,wait0_2,wait0_3);
   signal state_reg, state_next: eg_state_type;
```

```
-- counter to generate 10 ms tick  ( 2^N * 20ns )
process(clk,reset)
begin
   if (clk'event and clk='1') then
      q_reg <= q_next;
   end if;
end process;
-- next-state logic
q_next <= q_reg + 1;
--output tick
m_tick <= '1' when q_reg=0 else
          '0';
```

```
   -- debouncing
   FSM
process(clk,reset)
begin
   if (reset='1') then
      state_reg <= zero;
   elsif (clk'event and clk='1') then
      state_reg <= state_next;
   end if;
end process;
```


FSMD

-- *next-state/ out put logic*

```
process(state_reg,sw,m_tick)
begin
   state_next <= state_reg; --default: back to same state
   db <= '0';   -- default 0
   case state_reg is
      when zero =>
         if sw='1' then
            state_next <= wait1_1;
         end if;
      when wait1_1 =>
         if sw='0' then
            state_next <= zero;
         else
            if m_tick='1' then
               state_next <= wait1_2;
            end if;
         end if;
      when wait1_2 =>
         if sw='0' then
            state_next <= zero;
         else
            if m_tick='1' then
               state_next <= wait1_3;
            end if;
         end if;
      when wait1_3 =>
         if sw='0' then
            state_next <= zero;
         else
            if m_tick='1' then
               state_next <= one;
            end if;
         end if;
```

```
      when one =>
         db <='1';
         if sw='0' then
            state_next <= wait0_1;
         end if;
      when wait0_1 =>
         db <='1';
         if sw='1' then
            state_next <= one;
         else
            if m_tick='1' then
               state_next <= wait0_2;
            end if;
         end if;
      when wait0_2 =>
         db <='1';
         if sw='1' then
            state_next <= one;
         else
            if m_tick='1' then
               state_next <= wait0_3;
            end if;
         end if;
      when wait0_3 =>
         db <='1';
         if sw='1' then
            state_next <= one;
         else
            if m_tick='1' then
               state_next <= zero;
            end if;
         end if;
   end case;
end process;
```

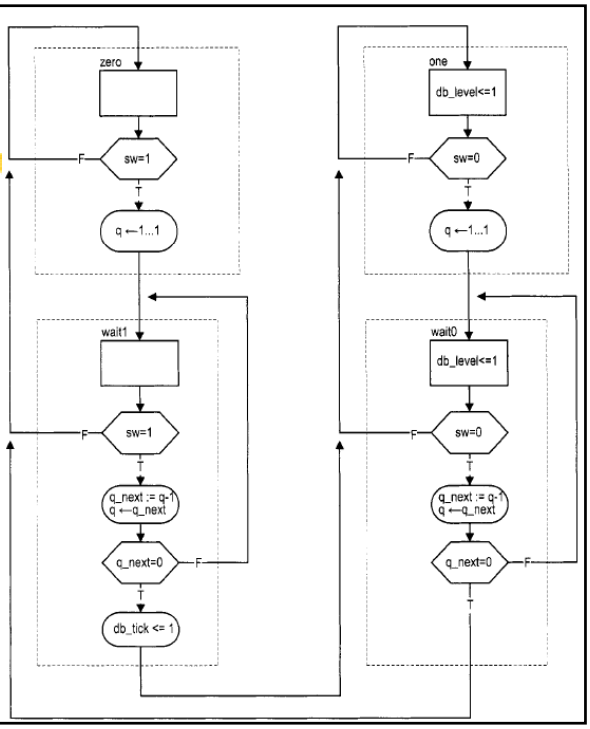
FSMD

We used an FSM and a timer (which is a regular sequential circuit), it is not based on the RT methodology because the two units are running independently and the FSM has no control over the timer. The waiting period in this design is between 20 and 30 ms but is not an exact interval. This deficiency can be overcome by applying the RT methodology.

- The circuit include two output signals: db-level, which is the debounced output, and db-tick, which is a one-clock-cycle enable pulse asserted at the zero-to-one transition.
- The zero and one states mean that the sw input has been stabilized for '0' and 'l' , respectively.
- The wait_l and wait_0 states are used to filter out short glitches.
- The data path contains one register, q, which is 21 bits wide.
- When the sw input signal becomes '1', the FSMD moves to the wait_1 state and initializes q to "1 . . . ". In the wait_l state, the q decrements in each clock cycle. If sw remains as '1', the FSMD returns to this state repeatedly until q reaches "0 . . . 0 " and then moves to the one state.

39

FSMD



ASMD chart of a debouncing circuit.

FSMD

- **Code with explicit data path components**

The key data path component custom 21-bit decrement counter that can:

- Be initialized with a specific value
- Count downward or pause
- Assert a status signal when the counter reaches 0

q-load : signal to load the initial value in binary counter
q-dec : signal to enable the counting.
q-zero : status signal, which is asserted when the counter reaches zero.

- The complete **data path** is composed of the q register and the next-state logic of the custom decrement counter. A comparison circuit is included to generate the q-zero status signal.
- The **control path** consists of an FSM, which takes the sw input and the q-zero status and asserts the control signals, q-load and q-dec, according to the desired action in the ASMD chart.

41

De-bouncing circuit with an explicit data path component

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity debounce is
  port(
    clk, reset: in std_logic;
    sw: in std_logic;
    db_level, db_tick: out std_logic
  );
end debounce ;

architecture exp_fsmd_arch of debounce is
  constant N: integer:=21;  -- filter of 40ms
  type state_type is (zero, wait0, one, wait1);
  signal state_reg, state_next: state_type;
  signal q_reg, q_next: unsigned(N-1 downto 0);
  signal q_load, q_dec, q_zero: std_logic;
begin
```

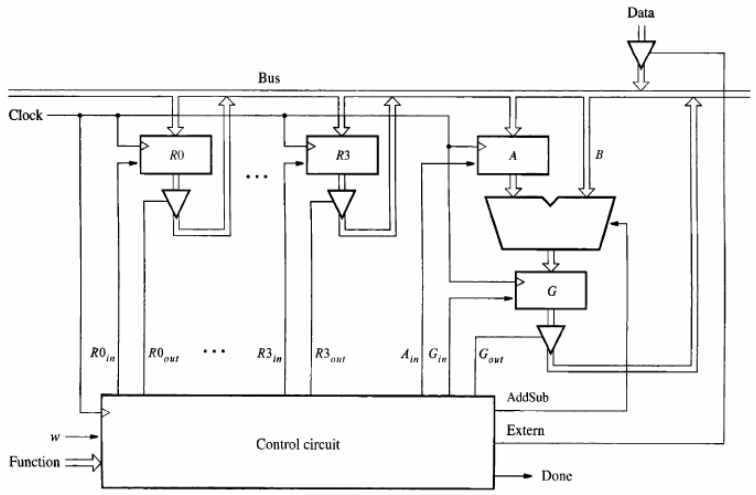
```
-- FSMD state & data registers
 process(clk, reset)
 begin
   if reset='1' then
     state_reg <= zero;
     q_reg <= (others=>'0');
   elsif (clk'event and clk='1') then
     state_reg <= state_next;
     q_reg <= q_next;
   end if;
 end process;

 -- FSMD data path (counter) next-state logic
 q_next <= (others=>'1') when q_load='1' else
         q_reg - 1 when q_dec='1' else
         q_reg;
 q_zero <= '1' when q_next=0 else '0';
```

42

```
-- FSMD control path next-state logic
  process(state_reg, sw, q_zero)
  begin
    q_load <= '0';
    q_dec <= '0';
    db_tick <= '0';
    state_next <= state_reg;
    case state_reg is
      when zero =>
        db_level <= '0';
        if (sw='1') then
          state_next <= wait1;
          q_load <= '1';
        end if;
      when wait1=>
        db_level <= '0';
        if (sw='1') then
          q_dec <= '1';
          if (q_zero='1') then
            state_next <= one;
            db_tick <= '1';
          end if;
        else -- sw='0'
          state_next <= zero;
        end if;
      when one =>
          db_level <= '1';
          if (sw='0') then
            state_next <= wait0;
            q_load <= '1';
          end if;
        when wait0=>
          db_level <= '1';
          if (sw='0') then
            q_dec <= '1';
            if (q_zero='1') then
              state_next <= zero;
            end if;
          else -- sw='1'
            state_next <= one;
          end if;
      end case;
    end process;
  end exp_fsmd_arch;
```

43

Simple Processor: Design



44

Simple Processor

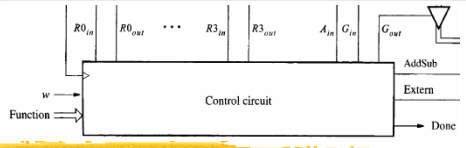
- R0, R1, R2, R3: n-bit registers
- Extern: control signal to enable n-bit external data input
- AddSub: 0 Sum A+B
  1 difference A-B [2's compliment of B]
- G: store output of add/sub with *Gin*, *Gout* control signals
- Operations performed in processor

Operation	Function Performed
Load *Rx*, *Data*	$Rx \leftarrow Data$
Move *Rx*, *Ry*	$Rx \leftarrow [Ry]$
Add *Rx*, *Ry*	$Rx \leftarrow [Rx] + [Ry]$
Sub *Rx*, *Ry*	$Rx \leftarrow [Rx] - [Ry]$

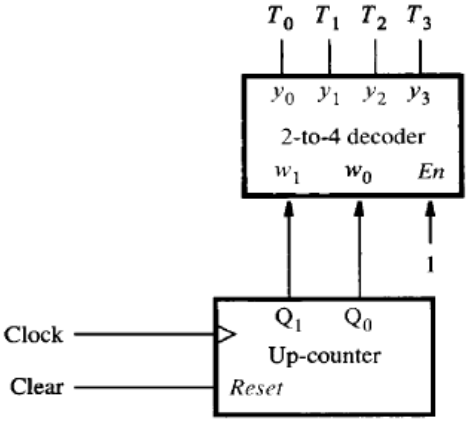
- *Load* and *Move* operations require only one step (clock-cycle, one transfer across the bus)
- Add and Sub operations require three steps

45

Simple Processor



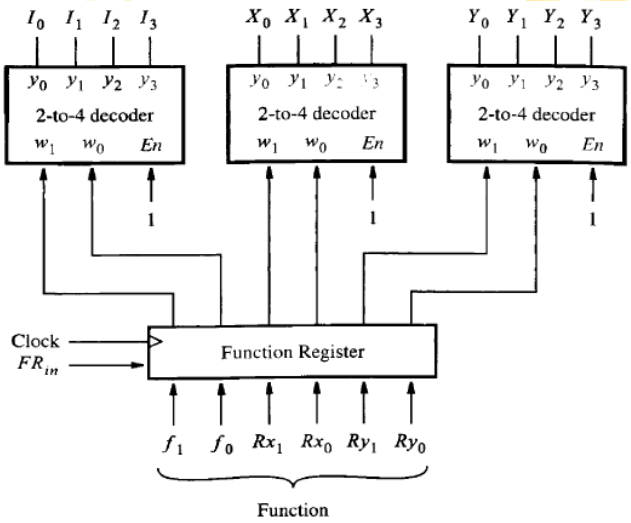
- *Function*: control signal input to perform specific operation initiated by setting *w*=1
- *Done*: output, when operation is completed
- Control circuit is based on a two bit counter since longest operation needs three steps
- Each of the decoder outputs represents a step in an operation.
- T_0, T_1, T_2, T_3: no-operation, step_1 to step_3



46

Simple Processor

- *Function* input: to specify the operation with six bits
- f_1f_0: opcode
  - 00-Load,
  - 01-Move,
  - 10-Add,
  - 11-Sub
- Rx_1Rx_0:operand Rx
- Ry_1Ry_0:operand Ry
- Function Register: store function inputs when FR_{in} is asserted



Simple Processor

Clear and FR_{in} for all the operation

- *Clear* :
  - ensure 00 count value as long as w=0 and no-operation is being executed;
  - clear the count value at the end of each operation

$$Clear = \overline{w}T_0 + Done$$

- FR_{in} :
  - load the values on the Function inputs into the Function Register when w changes to 1

$$FR_{in} = wT_0$$

48

Simple Processor

- *Rest outputs from control circuit depend on the specific step being performed in each operation*

	T_1	T_2	T_3
(Load): I_0	$Extern, R_{in} = X,$ $Done$		
(Move): I_1	$R_{in} = X, R_{out} = Y,$ $Done$		
(Add): I_2	$R_{out} = X, A_{in}$	$R_{out} = Y, G_{in},$ $AddSub = 0$	$G_{out}, R_{in} = X,$ $Done$
(Sub): I_3	$R_{out} = X, A_{in}$	$R_{out} = Y, G_{in},$ $AddSub = 1$	$G_{out}, R_{in} = X,$ $Done$

Control signals asserted in each operation/time step

49

Simple Processor

	T_1	T_2	T_3
(Load): I_0	$Extern, R_{in} = X,$ $Done$		
(Move): I_1	$R_{in} = X, R_{out} = Y,$ $Done$		
(Add): I_2	$R_{out} = X, A_{in}$	$R_{out} = Y, G_{in},$ $AddSub = 0$	$G_{out}, R_{in} = X,$ $Done$
(Sub): I_3	$R_{out} = X, A_{in}$	$R_{out} = Y, G_{in},$ $AddSub = 1$	$G_{out}, R_{in} = X,$ $Done$

Logic expressions for outputs of control circuit

Signals asserted in Add and Sub operations

$$Extern = I_0 T_1$$

$$Done = (I_0 + I_1)T_1 + (I_2 + I_3)T_3$$

$$A_{in} = (I_2 + I_3)T_1$$

$$G_{in} = (I_2 + I_3)T_2$$

$$G_{out} = (I_2 + I_3)T_3$$

$$AddSub = I_3$$

$$R0_{in} = (I_0 + I_1)T_1X_0 + (I_2 + I_3)T_3X_0$$

Values of RX_{0in} : $R0_{in}$ … $R3_{in}$
$RX0_{out}$: $R0_{out}$ … $R3_{out}$

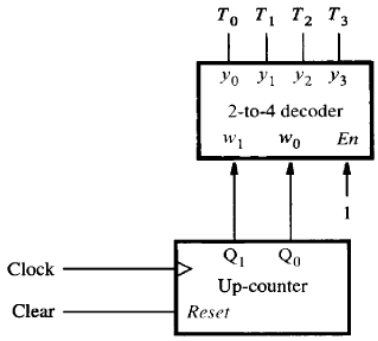
$$R0_{in} = (I_0 + I_1)T_1X_0 + (I_2 + I_3)T_3X_0$$

$$R0_{out} = I_1T_1Y_0 + (I_2 + I_3)(T_1X_0 + T_2Y_0)$$

50

Simple Processor: VHDL Code

- Components:
  - upcount
  - regn
  - trin
  - dec2to4



counter and decT instantiate the circuit

51

Simple Processor: Components

N-bit tri-state buffer

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY trin IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT ( X  : IN   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           E  : IN   STD_LOGIC ;
           F  : OUT  STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END trin ;

ARCHITECTURE Behavior OF trin IS
BEGIN
    F <= (OTHERS => 'Z') WHEN E = '0' ELSE X ;
END Behavior ;
```

52

Simple Processor: Components

N-bit register

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regn IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT ( R          : IN   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
           Rin, Clock : IN   STD_LOGIC ;
           Q          : OUT  STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF Rin = '1' THEN
            Q <= R ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Simple Processor: Components

2-bit up counter in synchronous reset

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY upcount IS
    PORT ( Clear, Clock : IN     STD_LOGIC ;
           Q            : BUFFER STD_LOGIC_VECTOR(1 DOWNTO 0) ) ;
END upcount ;

ARCHITECTURE Behavior OF upcount IS
BEGIN
    upcount: PROCESS ( Clock )
    BEGIN
        IF (Clock'EVENT AND Clock = '1') THEN
            IF Clear = '1' THEN
                Q <= "00" ;
            ELSE
                Q <= Q + '1' ;
            END IF ;
        END IF;
    END PROCESS;
END Behavior ;
```

Simple Processor: Components

COMPONENT : a piece of conventional code (LIBRARY declarations +ENTITY+ ARCHITECTURE). By declaring such code as being a COMPONENT, it can then be used within another circuit.

COMPONENT declaration:

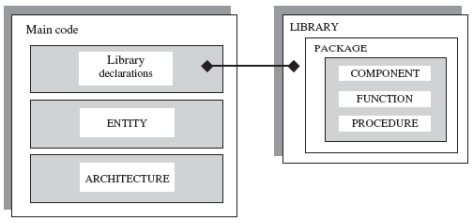
```
COMPONENT component_name IS
    PORT (
      port_name : signal_mode signal_type;
      port_name : signal_mode signal_type;
      ...);
END COMPONENT;
```

COMPONENT instantiation:

```
label: component_name PORT MAP (port_list);
```

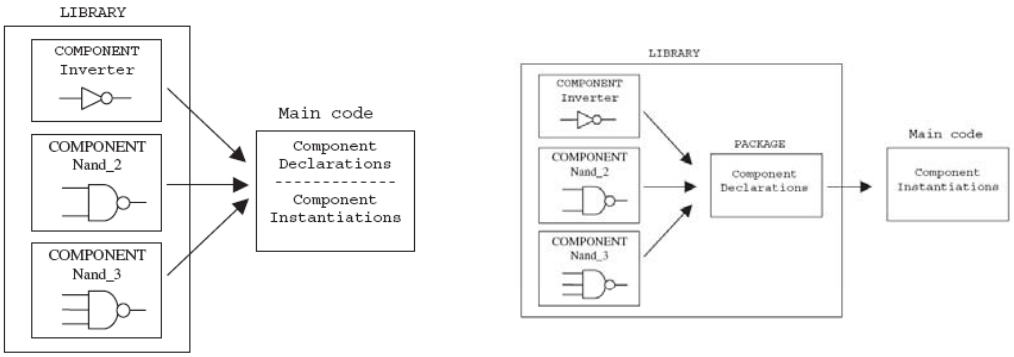

Simple Processor: Package

Frequently used pieces of VHDL code such as COMPONENTS, FUNCTIONS, or PROCEDURES are placed inside a PACKAGE and compiled into the destination LIBRARY. it allows code sharing and code reuse. Package can also contain TYPE and CONSTANT definitions.




```
PACKAGE package_name IS
    (declarations)
END package_name;

[PACKAGE BODY package_name IS
    (FUNCTION and PROCEDURE descriptions)
END package_name;]
```

Simple Processor: Package



Ways of declaring COMPONENTS: (a) declarations in the main code itself, (b) declarations in a PACKAGE.

57

Simple Processor: Package

```
----- File project.vhd: -----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_components.all;
```

```
------ File inverter.vhd: ----
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY inverter IS
   PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
END inverter;
-----------------------------------
ARCHITECTURE inverter OF inverter IS
BEGIN
   b <= NOT a;
END inverter;
```

```
------ File nand_2.vhd: -------
```

```
 ----- File nand_3.vhd: ----
```

```
----- File my_components.vhd: ---------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
------------------------
PACKAGE my_components IS
   ------ inverter: -------
   COMPONENT inverter IS
      PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
   END COMPONENT;
   ------ 2-input nand: ---
   COMPONENT nand_2 IS
      PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
   END COMPONENT;
   ------ 3-input nand: ---
   COMPONENT nand_3 IS
      PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
   END COMPONENT;
   ------------------------
END my_components;
```

58

Simple Processor: VHDL Code

- Reset: to initiate the counter to 00
- Func: six-bit signal (F & Rx & Ry), input to Function register

59

Simple Processor: VHDL Code

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;
USE work.subccts.all ;

ENTITY proc IS
    PORT ( Data      : IN     STD_LOGIC_VECTOR(7 DOWNTO 0) ;
           Reset, w  : IN     STD_LOGIC ;
           Clock     : IN     STD_LOGIC ;
           F, Rx, Ry : IN     STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           Done      : BUFFER STD_LOGIC ;
           BusWires  : INOUT  STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END proc ;
```

60

Simple Processor: VHDL Code

```
ARCHITECTURE Behavior OF proc IS
    SIGNAL Rin, Rout : STD_LOGIC_VECTOR(0 TO 3) ;
    SIGNAL Clear, High, AddSub : STD_LOGIC ;
    SIGNAL Extern, Ain, Gin, Gout, FRin : STD_LOGIC ;
    SIGNAL Count, Zero : STD_LOGIC_VECTOR(1 DOWNTO 0) ;
    SIGNAL T, I, X, Y : STD_LOGIC_VECTOR(0 TO 3) ;
    SIGNAL R0, R1, R2, R3 : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL A, Sum, G : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
    SIGNAL Func, FuncReg : STD_LOGIC_VECTOR(1 TO 6) ;
```

61

Simple Processor: VHDL Code

```
BEGIN
    Zero <= "00" ; High <= '1' ;
    Clear <= Reset OR Done OR (NOT w AND T(0)) ;
    counter: upcount PORT MAP ( Clear, Clock, Count ) ;
    decT: dec2to4 PORT MAP ( Count, High, T );
    Func <= F & Rx & Ry ;
    FRin <= w AND T(0) ;
    functionreg: regn GENERIC MAP ( N => 6 )
                 PORT MAP ( Func, FRin, Clock, FuncReg ) ;
    decI: dec2to4 PORT MAP ( FuncReg(1 TO 2), High, I ) ;
    decX: dec2to4 PORT MAP ( FuncReg(3 TO 4), High, X ) ;
    decY: dec2to4 PORT MAP ( FuncReg(5 TO 6), High, Y ) ;
    Extern <= I(0) AND T(1) ;
    Done <= ((I(0) OR I(1)) AND T(1)) OR ((I(2) OR I(3)) AND T(3)) ;
    Ain <= (I(2) OR I(3)) AND T(1) ;
    Gin <= (I(2) OR I(3)) AND T(2) ;
    Gout <= (I(2) OR I(3)) AND T(3) ;
    AddSub <= I(3) ;
```

Instantiation of Function Register with data inputs Func and the outputs FuncReg

Logic expressions for the output of control circuit

62

Simple Processor: VHDL Code

FOR / GENERATE:

```
label: FOR identifier IN range GENERATE
  (concurrent assignments)
END GENERATE;
```

$$R0_{in} = (I_0 + I_1)T_1X_0 + (I_2 + I_3)T_3X_0$$

$$R0_{out} = I_1T_1Y_0 + (I_2 + I_3)(T_1X_0 + T_2Y_0)$$

```
RegCntl:
FOR k IN 0 TO 3 GENERATE
      Rin(k) <= ((I(0) OR I(1)) AND T(1) AND X(k)) OR
            ((I(2) OR I(3)) AND T(3) AND X(k)) ;
      Rout(k) <= (I(1) AND T(1) AND Y(k)) OR
            ((I(2) OR I(3)) AND ((T(1) AND X(k)) OR (T(2) AND Y(k)))) ;
END GENERATE RegCntl ;
tri_extern: trin PORT MAP ( Data, Extern, BusWires ) ;
reg0: regn PORT MAP ( BusWires, Rin(0), Clock, R0 ) ;
reg1: regn PORT MAP ( BusWires, Rin(1), Clock, R1 ) ;
reg2: regn PORT MAP ( BusWires, Rin(2), Clock, R2 ) ;
reg3: regn PORT MAP ( BusWires, Rin(3), Clock, R3 ) ;
tri0: trin PORT MAP ( R0, Rout(0), BusWires ) ;
tri1: trin PORT MAP ( R1, Rout(1), BusWires ) ;
tri2: trin PORT MAP ( R2, Rout(2), BusWires ) ;
tri3: trin PORT MAP ( R3, Rout(3), BusWires ) ;
regA: regn PORT MAP ( BusWires, Ain, Clock, A ) ;
```

Instantiation of tri-state buffers and registers

63

Simple Processor: VHDL Code

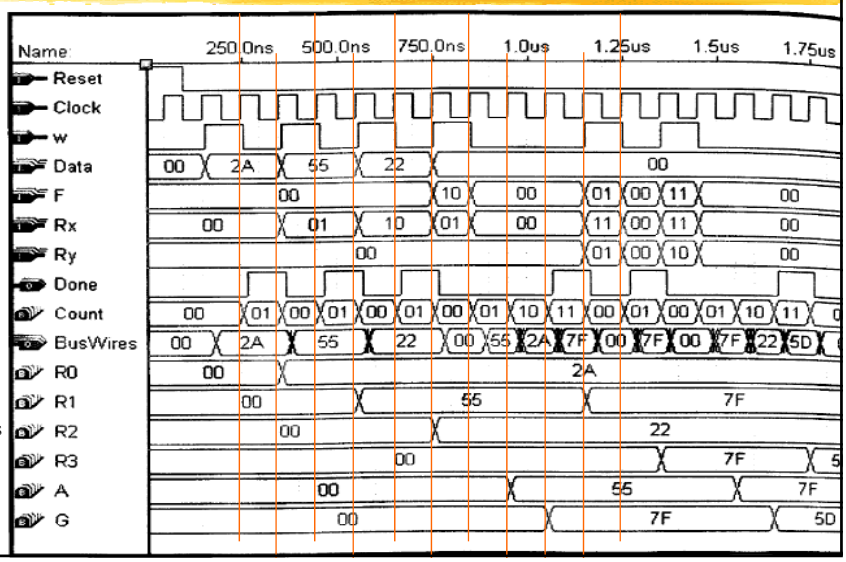
```
    alu:
    WITH AddSub SELECT
        Sum <= A + BusWires WHEN '0',
               A – BusWires WHEN OTHERS ;
    regG: regn PORT MAP ( Sum, Gin, Clock, G ) ;
    triG: trin PORT MAP ( G, Gout, BusWires ) ;
END Behavior ;
```

Description of adder/subtraction module

64

Simple Processor: VHDL Code: example results

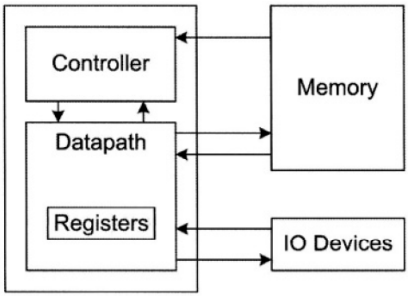
- Start of an operation at each ↑ and w=1
- At 250ns: Load R0, data
- Next Load R1, data
- Next Load R2, data
- At 850ns Add R1,R0
- At next ↑ R1(55) appear on bus
- At 950 ns 55 is loaded into A and R0 (2A) on bus Adder generate sum 7F,
- At 1050ns Loaded into G After this ↑, G (7F) is placed on bus
- At 1150 ns 7F is loaded into R1
- Move R3,R1
- Sub R3,R2



Embedded System: Computer System

A computer is an electronic machine which performs some computations. To have this machine perform a task, the task must be broken into small instructions, and the computer will be able to perform the complete task by executing each of its comprising instructions.

Each computer should have a CPU to execute instructions, a memory to store instructions and data, and an IO device to transfer information between the computer and the outside word.

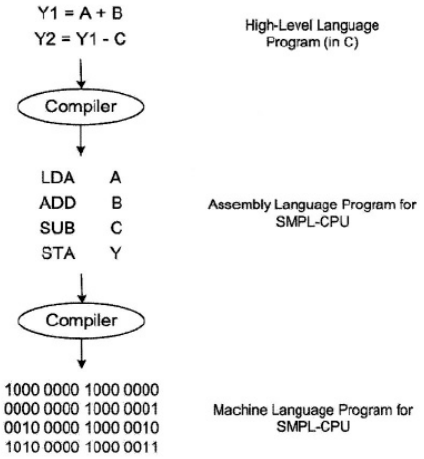


Von-Neumann Machine

66

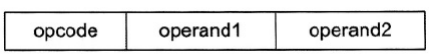
Computer System: Software

- The part of a computer system that contains instructions for the machine to perform is called its software.
- Ways to describe a computer software
  - Machine Language
  - Assembly Language
  - High-Level Language

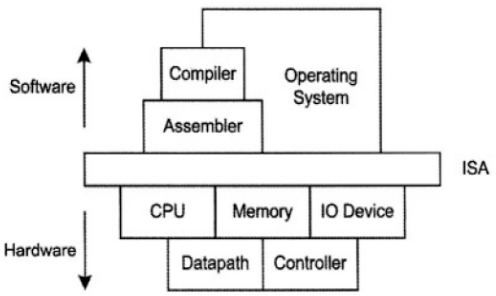


Computer System: ISA

- **Instruction Set Architecture:** specifies the interface between hardware and software of a processing unit. The ISA provides the details of the instructions that a computer should be able to understand and execute.



Instruction Format



Computer System Components

Computer System:
Simple CPU Design: Single-Cycle Implementation

CPU Specification:

CPU External Busses- 16-bit external data bus and a 13- bit address bus.

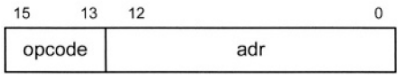
General Purpose Registers- a 16-bit register: *accumulator (acc).*

Memory Organization- Capable of addressing 8192 words of memory; each word has a 16 bit width.

Instruction Format-Each instruction is a 16-bit word and occupies a memory word. Operands: An *Explicit operand* (the memory location whose address is specified in the instruction), and an *implicit operand (acc).*

The CPU has a total of 8 instructions, divided into three classes:

- Arithmetic-logical instructions *(ADD, SUB, AND,* and *NOT)*
- Data-transfer instructions *(LDA, STA)*
- Control-flow instructions (*JMP, JZ*)



Computer System: Simple CPU Design

- Addressing Mode: Direct Addressing
- Instruction Set:

Opcode	Instruction	Instruction Class	Description
000	ADD *adr*	Arithmetic-Logical	*acc* ← *acc* + Mem[*adr*]
001	SUB *adr*	Arithmetic-Logical	*acc* ← *acc* - Mem[*adr*]
010	AND *adr*	Arithmetic-Logical	*acc* ← *acc* & Mem[*adr*]
011	NOT *adr*	Arithmetic-Logical	*acc* ← NOT (Mem[*adr*])
100	LDA *adr*	Data-Transfer	*acc* ← Mem[*adr*]
101	STA *adr*	Data-Transfer	Mem[*adr*] ← *acc*
110	JMP *adr*	Control-Flow	Unconditional jump to *adr*
111	JZ *adr*	Control-Flow	Conditional jump to *adr*

70

CPU Design:
Single-Cycle Implementation: datapath design

- Datapath design is an incremental process, at each increment, consider a class of instructions and build up a portion of the datapath.
- Then, in steps, combine these partial datapaths to generate the complete datapath.
- In the steps, decide on the control signals that control events in the datapath.
- In the design of the datapath, concern with how control signals affect flow of data and function of data units in the data path, and not how control signals are generated.

71

Simple CPU Design

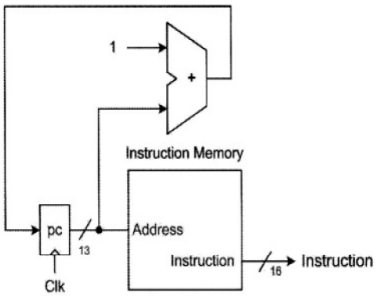
Step 1: Program Sequencing

•Instruction Fetch (IF): *Reading* an instruction from the memory

•Instruction memory: stores the instructions.

•Program Counter: A 13-bit register.

The register to hold the address of the current instruction to be read from the instruction memory. After the completion of the current instruction, the PC should be incremented by one to point to the next instruction in the instruction memory.



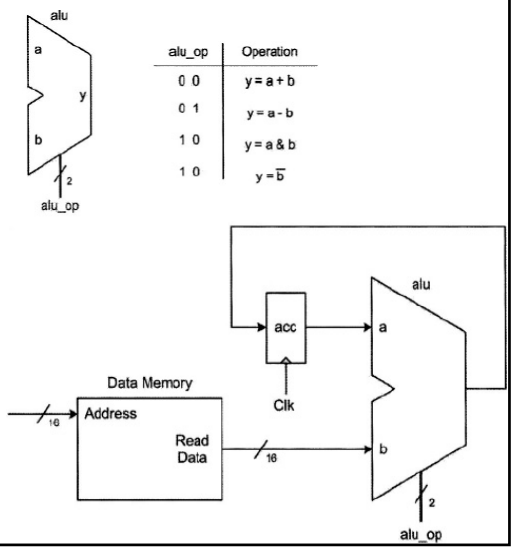
Program Sequencing Datapath

72

Simple CPU Design

Step 2: Arithmetic-Logical Instruction Data-Path.

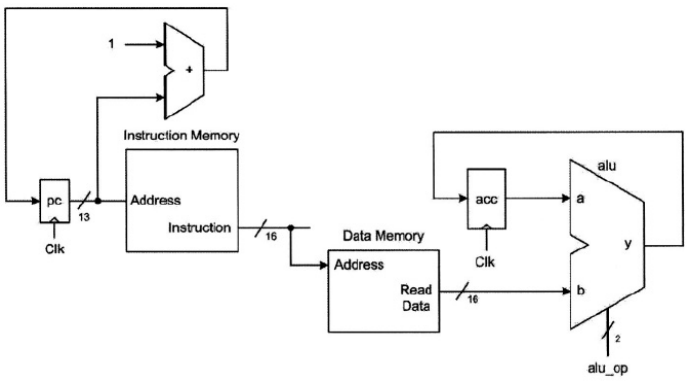
- First operand is *acc* . Second operand read from the *data memory*
- *adr* field of the instruction points to the memory location that contains the second operand (bits 12 to 0)
- Result of the operation is stored in *acc*
- *arithmetic-logical unit (alu)* performs the operation of arithmetic and logical instructions
- *alu* operation is controlled by a 2-bit input, *alu_op*
- *alu* is designed as combinational circuit



Simple CPU Design

Step 3: Combining the Two Previous Datapaths

- Connect the address input of the *data memory* to the *adr* field (bits 12 to 0) of the instruction which is read from the *instruction memory*.
- Combined datapath is able to sequence the program and execute arithmetic or logical instructions

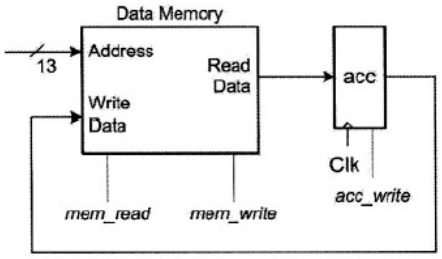


Simple CPU Design

Step 4: Data-Transfer Instruction

Datapath: *LDA* and *STA*

- *LDA* uses *adr* field of the instruction to read a 16-bit data from data *memory* and store it in *acc* register
- *STA* instruction writes the content of *acc* into a *data memory* location pointed by the *adr* field
- *data memory* have two control signals, *mem_read* and *mem_write* for control of reading from it or writing into it.
- In order to control accumulator writing, the *acc_write* (write-control, or clock enable) signal is needed

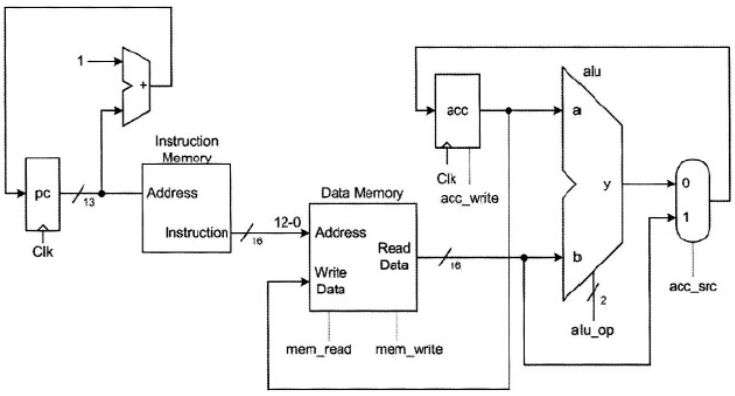


75

Simple CPU Design

Step 5: Combining the Two Previous Datapaths

- Combining the two datapaths, may result in multiple connections to the input of an element
- To have both connections, a multiplexer (or a bus) is used to select one of the sources.

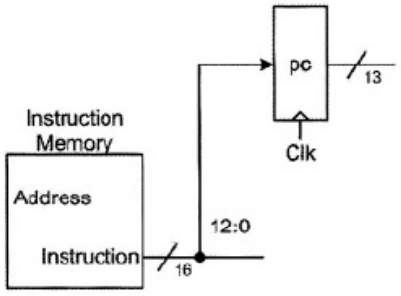


76

Simple CPU Design

Step 6: Control-Flow Instruction Datapath.

- *JMP*: an unconditional jump; writes the *adr* field (bits 12 to 0) of an instruction into *pc*.
- *JZ*: a conditional jump; writes the *adr* field into *pc* if *acc* is zero.
- Checking for the zero value of *acc*, a *NOR* gate on the output of this register generates the proper signal that detects this condition for execution of the *JZ* instruction.

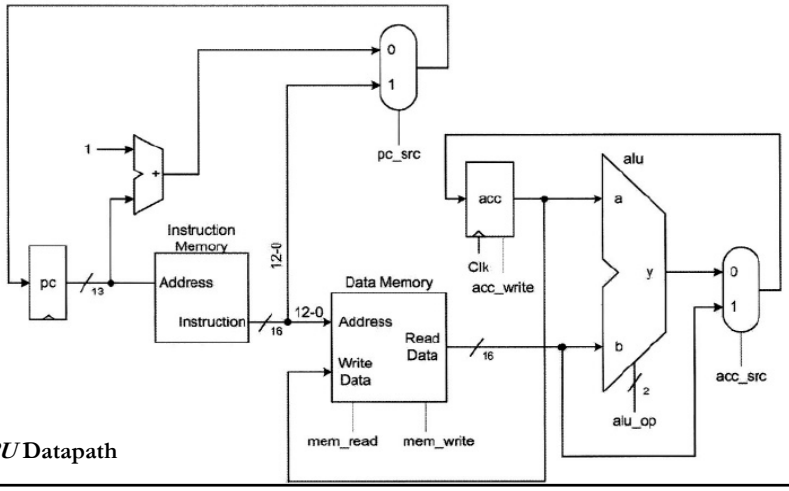


Control-Flow Instructions Datapath

Simple CPU Design

Step 7: Combining the Two Previous Datapaths

- To select the appropriate source for *pc* input; multiplexer select input is *pc_src*.



***Simple CPU* Datapath**

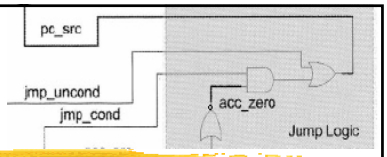
Simple CPU Design

Instruction Execution : Example instruction- *ADD 100*

- On the rising edge of the clock, a new value is written into *pc,* and *pc* points the *instruction memory* to read the instruction *ADD 100*.
- Memory read operation is complete and the controller starts to decode the instruction.
- Controller issue the appropriate control signals to control the flow of data in the datapath.
- On the next rising edge of the clock, the *alu* output is written into *acc* to complete the execution of the current instruction and the *pc+1* is written into *pc*. This new value of *pc* points to the next instruction.
- Since the execution of the instruction is completed in one clock cycle, the implementation is called *single-cycle* implementation.

79

Simple CPU Design

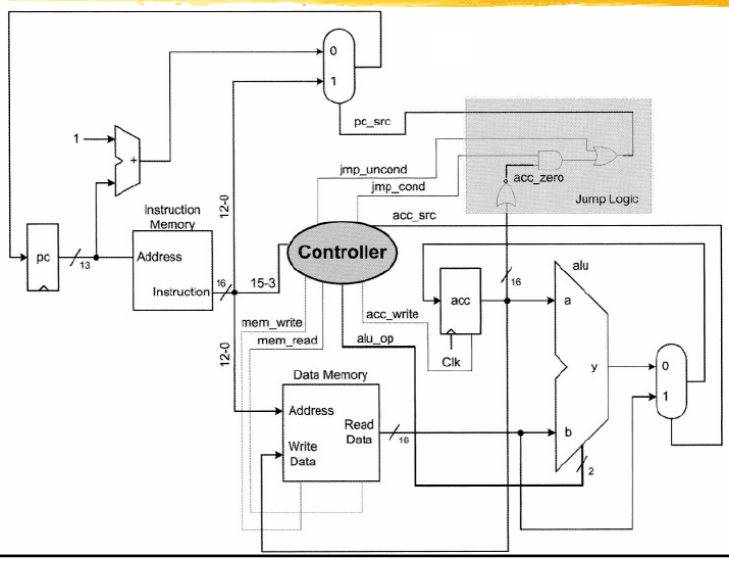


Single-Cycle Implementation – Controller Design

- Controller issues the control signals based on the *opcode* field of the instruction.
- *opcode* field will not change while the instruction is being executed
- Control signals will have fixed values during the execution of an instruction
- Controller is implemented as a combinational circuit
  - Controller issues all control signals directly, except *pc_src,* which is issued using a simple logic circuit.
  - For all instructions, except *JMP* and *JZ*, both *jmp_uncond* and *jmp_cond* are **0.** Hence the Jump Logic block produces a **0** on *pc_src* that causes *pc* to increment.
  - For the *JMP* instruction, the *jmp_uncond* becomes **1**, and this puts a **1** on the *pc_src* and directs the *adr* field of the instruction into the *pc* input.
  - For the *JZ* instruction, the *jmp_cond* is asserted and if the *acc_zero* is **1** (when all bits of *acc* are **0**, the *acc_zero* becomes **1)**, the address field of the instruction from the *instruction memory* is put into the *pc* register. Else if *acc* is not **0**, the *pc+1* source of *pc* is selected.

80

Simple CPU Design



Simple CPU Design

Status of control signals for controlling flow of data in datapath.

Arithmetic-Logical Class:

- *mem_read=1* to read an operand from *data memory*
- *acc_write=1* to store the *alu* result in *acc*
- *alu_op* becomes *00*, *01*, *10,* or *11* for *ADD*, *SUB*, *AND*, and *NEG*
- *acc_src=0*, to direct *alu* output to the *acc* input
- *jmp_cond,* and *jmp_uncond* are **0** to direct *pc+1* to the *pc* input

Data-Transfer Class:

LDA Instruction:

- *mem_read=1* to read an operand from the *data memory*
- *acc_write=1,* to store the *data memory* output in *acc*
- *alu_op* is *XX,* because *alu* has no role in the execution of *LDA*
- *acc_src=1*, to direct *data memory* output to the *acc* input
- *jmp_cond,* and *jmp_uncond* are **0** to direct *pc+1* to the *pc* input

Simple CPU Design

STA Instruction:

- *mem_write=1* to write *acc* into the *data memory*
- *acc_write=0,* so that the value of *acc* remains unchanged
- *alu_op* is *XX* because *alu* has no role in the execution of *STA*
- *acc_src=X*, because *acc* writing is disabled and its source is not important
- *jmp_cond,* and *jmp_uncond* are **0** to direct *pc+1* to the *pc* input.

Control-Flow Class:

JMP Instruction:

- *mem_read* & *mem_write* are **0**, because *JMP* does not read from or write into *data memory*
- *acc_write=0,* because *acc* does not change during *JMP*
- *alu_op* is *XX* because *alu* has no role in execution of *JMP*
- *acc_src=X,* because *acc* writing is disabled and its source is not important
- *jmp_cond=0, jmp_uncond=1*, puts **1** on *pc_src* and directs jump address(12- 0) to *pc*

83

Simple CPU Design

JZ Instruction:

- *mem_read* and *mem_write* are **0**, *JZ* does not read from or write into the *data memory*
- *acc_write=0*, *acc* does not change during *JZ*
- *alu_op* equals to *XX,* because *alu* has no role in execution of *JZ*
- *acc_src=X,* because *acc* writing is disabled and its source is not important
- *jmp_cond=1, jmp_uncond=0,* if *acc_zero* is **1**, puts **1** on *pc_src* and directs jump address (bits12 to 0) to *pc* input. Else value of *pc_src* is **0** and *pc+1* is directed to *pc* input.

84

Simple CPU Design

Controller Truth Table

Instruction Class	Inst	opcode	mem read	mem write	acc write	acc src	alu op	jmp uncond	jmp cond
Arithmetic Logical	ADD	000	1	0	1	0	00	0	0
	SUB	001	1	0	1	0	01	0	0
	AND	010	1	0	1	0	10	0	0
	NOT	011	1	0	1	0	11	0	0
Data Transfer	LDA	100	1	0	1	1	XX	0	0
	STA	101	0	1	0	X	XX	0	0
Control Flow	JMP	110	0	0	0	X	XX	1	0
	JZ	111	0	0	0	X	XX	0	1

The controller (single-cycle implementation) of the system can be designed as a combinational circuit using a truth table.

85

CPU Design: Multi-Cycle Implementation

Single-cycle implementation: used

- Two memory units
- Two functional units (the *alu* and the adder)

Multi-cycle implementation: reduces the required hardware

- Hardware can be shared within the execution steps of an instruction
- Each instruction is executed in a series of steps, each takes one clock cycle to execute

86

CPU Design:
Multi-Cycle Implementation: datapath design

- Use a single memory unit which stores both instructions and data,
- Use a single *alu* which plays the role of both *alu* and the adder.

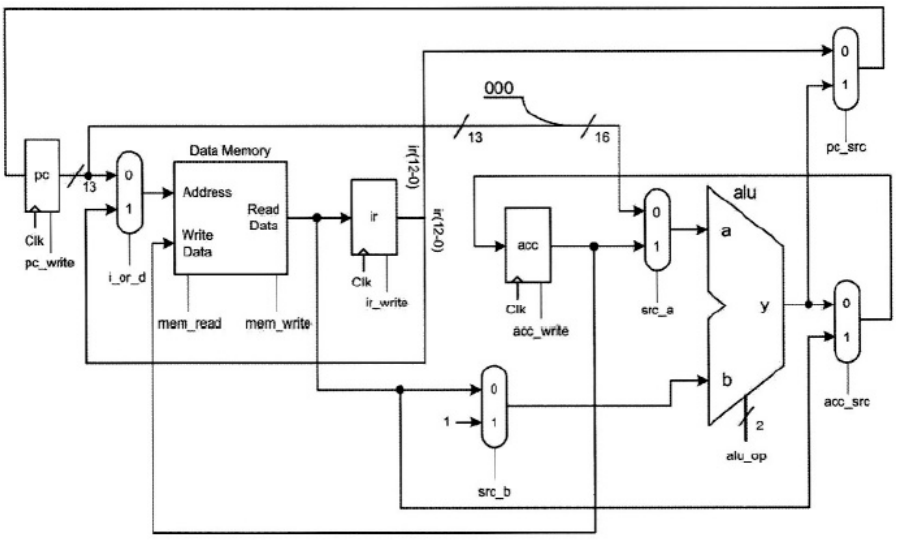
Sharing hardware adds one or more registers and multiplexers

- one multiplexer: To choose between the address of the memory unit from the *pc* output (to address instructions) and bits 12 to 0 of the instruction (to address data)
- two multiplexers: at the *alu* inputs;
  - first *alu* input, chooses between *pc* and *acc*
  - second *alu* input, chooses between memory output and a constant value 1

  alu inputs are 16 bits wide, so append 3 zeros on the left of *pc* to make 16-bits
- *instruction register (ir):* To store the instruction read from the memory

87

Multi-Cycle Implementation: Datapath design



88

Multi-Cycle Implementation:

Steps for execution of the instructions:

Instruction Fetch (IF): Read the instruction from memory and increment *pc*.

memory-read operation - *pc* addresses the memory for memory-read operation and write the instruction into *ir*.

pc increment - apply *pc* to the first *alu* input, and the constant value 1 to the second *alu* input, perform an addition, and store the *alu* output in *pc*

Instruction Decode (ID): Controller decodes the instruction (stored in *ir*) to issue the appropriate control signals.

Execution (EX): Datapath operation in this step is determined by the instruction class:

Arithmetic-Logical Class: Apply bits 12 to 0 of *ir* to the memory, and perform a memory read operation. Apply *acc* to the first *alu* input, and the memory output to the second *alu* input, perform an *alu* operation (addition, subtraction, logical and, and negation), and finally store the *alu* result into *acc*.

89

Multi-Cycle Implementation

Data-Transfer Class: Apply bits 12 to 0 of *ir* to the memory
LDA- perform a memory read operation, and write the data into *acc*.
STA- instruction: perform a memory write operation to write *acc* into the memory.

Control-Flow Class:

JMP- write bits 12 to 0 of *ir* to *pc*.

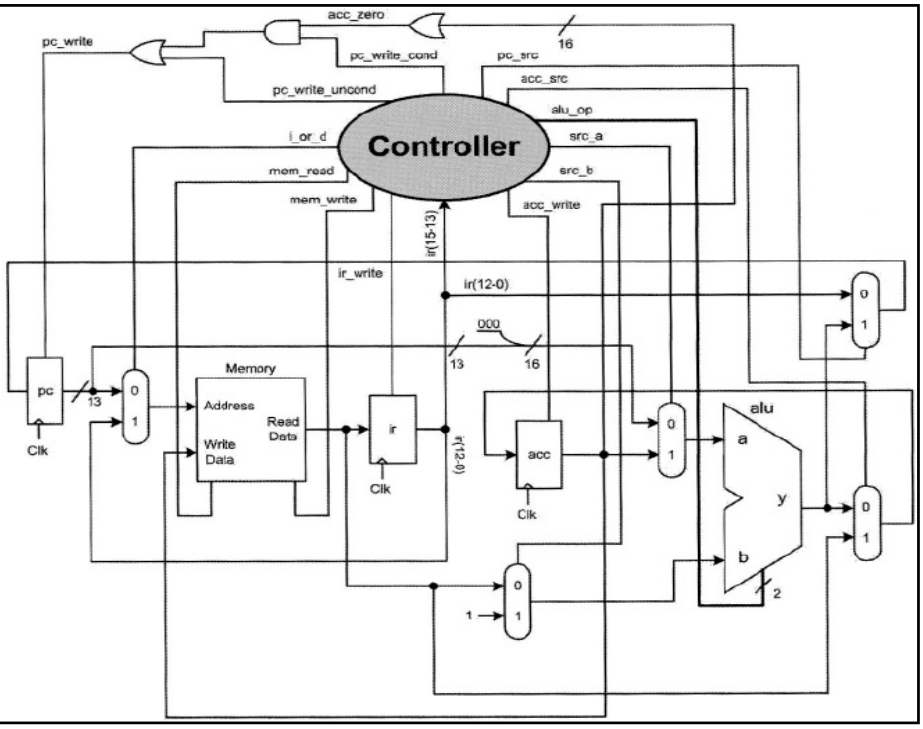
JZ - write bits 12 to 0 of *ir* to *pc* , if the content of *acc* is zero.

Controller Design:

- Controller specify the appropriate control signals for each step
- Controller of a multi-cycle datapath is designed as a sequential circuit

90

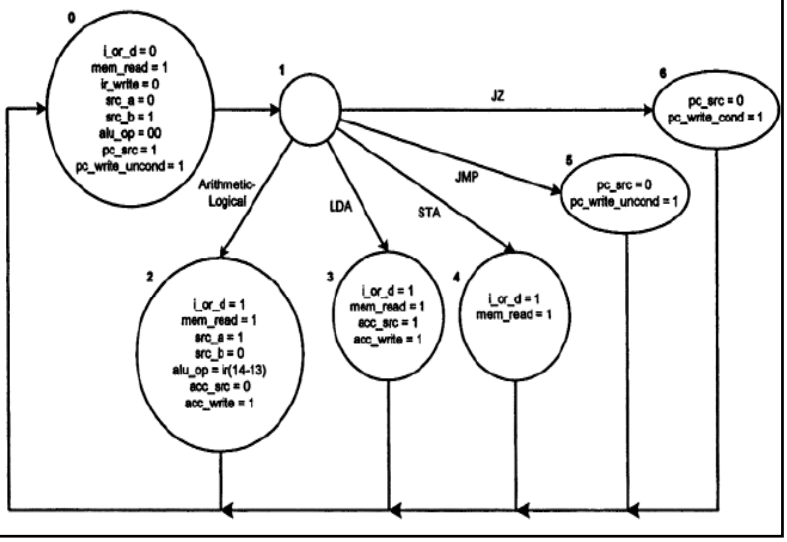
Multi-Cycle Implementation:
Datapath & Controller inter-connection



Multi-Cycle Implementation: Controller design

Controller design as a Moore finite state machine

- each state issues appropriate control signals and specifies the next state
- transition between states are triggered by the edge of the clock
- all control signals in a state are issued by entering the state



Multi-Cycle Implementation: Controller design

State 0: IF step :

- *pc* is applied to memory address input (*i_or_d=0*),
- instruction is read from memory (*mem_read=1*),
- instruction is written into *ir* (*ir_write=1*),
- *pc* is incremented by *1* (*src_a=0, src_b=1, alu_op=00, pc_src=1, pc_write_uncond=1*).

State 1: give enough time to the controller to decode the instruction,

- no control signal is asserted
- instruction decoding specifies the next state according to the type of the instruction being executed

Next State:

Arithmetic-logical instruction / *LDA* instruction /*STA* instruction /*JMP* instruction /*JZ* instruction:

93

Multi-Cycle Implementation: Controller design

Home Assignment: The implementation of the controller requires a state machine that can be implemented by a one-hot state machine or an encoded machine with three flip-flops.

94