ECE 545
Lecture 12

Design of Controllers using Algorithmic State Machine (ASM) Charts
Required reading

• P. Chu, *RTL Hardware Design using VHDL*

*Chapter 11, Register Transfer Methodology: Principle*
Recommended reading

• P. Chu, *RTL Hardware Design using VHDL*

  Chapter 12, *Register Transfer Methodology: Practice*
Slides based partially on

- S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*

  *Chapter 8, Synchronous Sequential Circuits*
  *Sections 8.1-8.5*

  *Chapter 8.10, Algorithmic State Machine (ASM) Charts*

  *Chapter 10.2 Design Examples*
Structure of a Typical Digital System

Datapath (Execution Unit)

Data Inputs

Control & Status Inputs

Control Signals

Datapath (Execution Unit)

Data Outputs

Control & Status Outputs

Controller (Control Unit)

Status Signals
Hardware Design with RTL VHDL

- Pseudocode
- Interface
- Datapath
  - Block diagram
  - VHDL code
- Controller
  - ASM chart
  - VHDL code
Steps of the Design Process
Introduced in Class Today

1. Text description
2. Interface
3. Pseudocode
4. Block diagram of the Datapath
5. Interface with the division into Datapath and Controller
6. ASM chart of the Controller
7. RTL VHDL code of the Datapath, Controller, and Top-Level Unit
8. Testbench of the Datapath, Controller, and Top-Level Unit
9. Functional simulation and debugging
10. Synthesis and post-synthesis simulation
11. Implementation and timing simulation
12. Experimental testing
Generalized FSM

Based on RTL Hardware Design by P. Chu
Generalized FSM

Inputs

Next State function

Present State Register

Next State

clock reset

Mealy Output function

Mealy Outputs

Mealy Outputs

Moore Output function

Moore Outputs
Algorithmic State Machine (ASM) Charts
Algorithmic State Machine –
representation of a Finite State Machine
suitable for FSMs with a larger number of
inputs and outputs compared to FSMs
expressed using state diagrams and state
tables.
ASM Chart

- Flowchart-like diagram
- Provides the same info as a state diagram
- More descriptive, better for complex description
Elements used in ASM charts (1)

(a) State box

(b) Decision box

(c) Conditional output box
State Box

- **State box** – represents a state.
- Equivalent to a node in a state diagram or a row in a state table.
- Contains register transfer actions or output signals
- **Moore-type outputs are listed inside of the box.**
- It is customary to write only the name of the signal that has to be asserted in the given state, e.g., \( z \) instead of \( z=1 \).
- Also, it might be useful to write an action to be taken, e.g., \( \text{count} \leq \text{count} + 1 \), and only later translate it to asserting a control signal that causes a given action to take place (e.g., enable signal of a counter).
Decision Box

- **Decision box** – indicates that a given condition is to be tested and the exit path is to be chosen accordingly. The condition expression may include one or more inputs to the FSM.
Conditional Output Box

- **Conditional output box**
- **Denotes output signals that are of the Mealy type.**
- The condition that determines whether such outputs are generated is specified in the decision box.
Counting: Block diagram + ASM
Counting: Timing waveforms
ASMs representing simple FSMs

- Algorithmic state machines can model both Mealy and Moore Finite State Machines
- They can also model machines that are of the mixed type
Moore FSM – Example 1: State diagram
Moore FSM – Example 1: State table

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
<th>Output $z$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$w = 0$</td>
<td>$w = 1$</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>
ASM Chart for Moore FSM – Example 2
Moore FSM

process(clock, reset)

concurrent statements
Example 1: VHDL code (1)

```vhdl
USE ieee.std_logic_1164.all;

ENTITY simple IS
    PORT ( clock : IN STD_LOGIC;
           resetn : IN STD_LOGIC;
           w      : IN STD_LOGIC;
           z      : OUT STD_LOGIC );
END simple;

ARCHITECTURE Behavior OF simple IS
    TYPE State_type IS (A, B, C);
    SIGNAL y : State_type;
BEGIN
    PROCESS ( resetn, clock )
    BEGIN
        IF resetn = '0' THEN
            y <= A;
        ELSIF rising_edge(clock) THEN
```
Example 1: VHDL code (2)

```vhdl
CASE y IS
    WHEN A =>
        IF w = '1' THEN
            y <= B;
        ELSE
            y <= A;
        END IF;
    WHEN B =>
        IF w = '1' THEN
            y <= C;
        ELSE
            y <= A;
        END IF;
    WHEN C =>
        IF w = '0' THEN
            y <= A;
        ELSE
            y <= C;
        END IF;
END CASE;
```
Example 1: VHDL code (3)

```vhdl
END IF;
END PROCESS;

z <= '1' WHEN y = C ELSE '0';

END Behavior;
```
Mealy FSM – Example 2: State diagram
ASM Chart for Mealy FSM – Example 2
Mealy FSM

`process(clock, reset)`

```
Inputs
```

```
Next State function
```

```
Present State Register
```

```
Next State
```

```
Output function
```

```
Present State
```

```
Outputs
```

```
concurrent statements
```

```
clock
reset
```

```
concurrent statements
```

```
process(clock, reset)
```

```
Inputs
```

```
Next State function
```

```
Present State Register
```

```
Next State
```

```
Output function
```

```
Present State
```

```
Outputs
```

```
concurrent statements
```

```
clock
reset
```

```
concurrent statements
```

```
process(clock, reset)
```

```
Inputs
```

```
Next State function
```

```
Present State Register
```

```
Next State
```

```
Output function
```

```
Present State
```

```
Outputs
```

```
concurrent statements
```

```
clock
reset
```

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

ENTITY Mealy IS
  PORT ( clock : IN STD_LOGIC;
         resetn : IN STD_LOGIC;
         w : IN STD_LOGIC;
         z : OUT STD_LOGIC );
END Mealy;

ARCHITECTURE Behavior OF Mealy IS
  TYPE State_type IS (A, B);
  SIGNAL y : State_type;
BEGIN
  PROCESS ( resetn, clock )
  BEGIN
    IF resetn = '0' THEN
      y <= A;
    ELSIF rising_edge(clock) THEN
      \[ \text{Example 3: VHDL code (1)} \]
  \end{verbatim}
CASE y IS
  WHEN A =>
    IF w = '1' THEN
      y <= B;
    ELSE
      y <= A;
    END IF;
  WHEN B =>
    IF w = '0' THEN
      y <= A;
    ELSE
      y <= B;
    END IF;
END CASE;
Example 3: VHDL code (3)

END IF ;
END PROCESS ;

z <= '1' WHEN (y = B) AND (w='1') ELSE '0' ;

END Behavior ;
Control Unit Example: Arbiter (1)
Control Unit Example: Arbiter (2)
Control Unit Example: Arbiter (3)
ASM Chart for Control Unit - Example 4
Moore FSM

process(clock, reset)

Inputs → Next State function → Next State

clock → Present State Register → Present State
reset → Output function → Outputs

concurrent statements
Example 4: VHDL code (1)

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY arbiter IS
    PORT ( Clock, Resetn : IN    STD_LOGIC ;
          r : IN    STD_LOGIC_VECTOR(1 TO 3) ;
          g : OUT   STD_LOGIC_VECTOR(1 TO 3) ) ;
END arbiter ;

ARCHITECTURE Behavior OF arbiter IS
    TYPE State_type IS (Idle, gnt1, gnt2, gnt3) ;
    SIGNAL y : State_type ;
Example 4: VHDL code (2)

BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN y <= Idle ;
        ELSIF rising_edge(Clock) THEN
            CASE y IS
            WHEN Idle =>
                IF r(1) = '1' THEN y <= gnt1 ;
                ELSIF r(2) = '1' THEN y <= gnt2 ;
                ELSIF r(3) = '1' THEN y <= gnt3 ;
                ELSE y <= Idle ;
                END IF ;

            WHEN gnt1 =>
                IF r(1) = '0' THEN y <= Idle ;
                ELSE y <= gnt1 ;
                END IF ;

            WHEN gnt2 =>
                IF r(2) = '0' THEN y <= Idle ;
                ELSE y <= gnt2 ;
                END IF ;
Example 4: VHDL code (3)

WHEN gnt3 =>
    IF r(3) = '0' THEN y <= Idle ;
    ELSE y <= gnt3 ;
    END IF ;

    END CASE ;

    END IF ;

    END PROCESS ;

    g(1) <= '1' WHEN y = gnt1 ELSE '0' ;
    g(2) <= '1' WHEN y = gnt2 ELSE '0' ;
    g(3) <= '1' WHEN y = gnt3 ELSE '0' ;

    END Behavior ;
Exercise

Assuming ASM chart given on the next slide, supplement timing waveforms given in the answer sheet with the correct values of signals State, g1, g2, g3, in the interval from 0 to 575 ns.
ASM Chart
ASM Block

- One state box
- One or more (optional) decision boxes: with T (1) or F (0) exit paths
- One or more (optional) conditional output boxes: for Mealy outputs

Figure 10.4 ASM block.
ASM Chart Rules

- Difference between a regular flowchart and an ASM chart:
  - Transition governed by clock
  - Transition occurs between ASM blocks

- Basic rules:
  - For a given input combination, there is one unique exit path from the current ASM block
  - Any closed loop in an ASM chart must include a state box

Based on RTL Hardware Design by P. Chu
Incorrect ASM Charts

Based on RTL Hardware Design by P. Chu
Class Exercise 1
STATISTICS
no_1 = no_2 = no_3 = sum = 0
for i=0 to k-1 do
    sum = sum + din
    if din > no_1 then
        no_3 = no_2
        no_2 = no_1
        no_1 = din
    elseif (din > no_2) then
        no_3 = no_2
        no_2 = din
    elseif (din > no_3) then
        no_3 = din
    end if
end for
avr = sum / k
Circuit Interface

- clk
- reset
- din
- go
- Statistics
- done
- dout
- dout_mode
## Interface Table

<table>
<thead>
<tr>
<th>Port</th>
<th>Width</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>1</td>
<td>System clock.</td>
</tr>
<tr>
<td>reset</td>
<td>1</td>
<td>System reset.</td>
</tr>
<tr>
<td>din</td>
<td>n</td>
<td>Input Data.</td>
</tr>
<tr>
<td>go</td>
<td>1</td>
<td>Control signal indicating that the first input will be ready in the next clock cycle. Active for one clock cycle.</td>
</tr>
<tr>
<td>done</td>
<td>1</td>
<td>Signal set to high after the output is ready.</td>
</tr>
<tr>
<td>dout</td>
<td>n</td>
<td>Output dependent on the dout_mode input.</td>
</tr>
<tr>
<td>dout_mode</td>
<td>2</td>
<td>Control signal determining value available at the output. 00: avr, 01: no_1, 10: no_2, 11: no_3.</td>
</tr>
</tbody>
</table>
Block diagram of the Datapath
Interface with the division into the Datapath and the Controller

Datapath
- din
- dout_mode
- n
- 2
- clk
- reset
- gt1
- gt2
- gt3
- zi

Controller
- en1
- en2
- en3
- esum
- enc
- s2
- s3

done
ASM Charts
Class Exercise 2
CIPHER
Pseudocode

Split input I into four words, I3, I2, I1, I0, of the size of w bits each.

A = I3; B = I2; C = I1; D = I0
B = B + S[0]
D = D + S[1]
for i = 1 to r do
    { 
        T = (B*(2B + 1)) <<< k  
        U = (D*(2D + 1)) <<< k  
        A = ((A ⊕ T) <<< U) + S[2i] 
        C = ((C ⊕ U) <<< T) + S[2i + 1] 
        (A, B, C, D) = (B, C, D, A) 
    } 
A = A + S[2r + 2]
C = C + S[2r + 3]
O = (A, B, C, D)
Notation

**w:** word size, e.g., w=8 (constant)

**k:** $\log_2(w)$ (constant)

**A, B, C, D, U, T:** w-bit variables

**I3, I2, I1, I0:** Four w-bit words of the input I

**r:** number of rounds (constant)

**O:** output of the size of 4w bits

**S[j]:** 2r+4 round keys stored in two RAMs.
   - Each key is a w-bit word.
   - The first RAM stores values of $S[j=2i]$, i.e., only round keys with even indices. The second memory stores values of $S[j=2i+1]$, i.e., only round keys with odd indices.
Operations

⊕ : XOR
+ : addition modulo $2^w$
– : subtraction modulo $2^w$
* : multiplication modulo $2^w$
$X \ll Y$ : rotation of $X$ to the left by the number of positions given in $Y$
$X \gg Y$ : rotation of $X$ to the right by the number of positions given in $Y$
Circuit Interface

clk
reset
I
write_I
Sj
Write_Sj
j

4w
w

CIPHER

4w
w

O
DONE

m
# Interface Table

<table>
<thead>
<tr>
<th>Port</th>
<th>Width</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>1</td>
<td>System clock.</td>
</tr>
<tr>
<td>reset</td>
<td>1</td>
<td>System reset – clears internal registers.</td>
</tr>
<tr>
<td>I</td>
<td>4w</td>
<td>Input block.</td>
</tr>
<tr>
<td>write_I</td>
<td>1</td>
<td>Synchronous write control signal for the input block I. After the block I is written to the CIPHER unit, the encryption of I starts automatically.</td>
</tr>
<tr>
<td>Sj</td>
<td>w</td>
<td>Round key S[j] loaded to one of the two internal memories.</td>
</tr>
<tr>
<td>write_Sj</td>
<td>1</td>
<td>Synchronous write control signal for the round key S[j].</td>
</tr>
<tr>
<td>j</td>
<td>m</td>
<td>Index of the round key S[j] loaded using input Sj.</td>
</tr>
<tr>
<td>O</td>
<td>4w</td>
<td>Output block.</td>
</tr>
<tr>
<td>DONE</td>
<td>1</td>
<td>Asserted for one clock cycle when the output O is ready.</td>
</tr>
</tbody>
</table>

**Note:**

m is a size of index j. It is a minimum integer, such that $2^m - 1 \geq 2r + 3$. 
Protocol (1)

An external circuit first loads all round keys $S[0], S[1], S[2], ..., S[2r+2], [2r+3]$ to the two internal memories of the CIPHER unit.

The first memory stores values of $S[j=2i]$, i.e., only round keys with even indices. The second memory stores values of $S[j=2i+1]$, i.e. only round keys with odd indices.

Loading round keys is performed using inputs: $S_j, j, write_S_j, clk$.

Then, the external circuits, loads an input block $I$ to the CIPHER unit, using inputs: $I, write_I, clk$.

After the input block $I$ is loaded to the CIPHER unit, the encryption starts automatically.
Protocol (2)

When the encryption is completed, signal DONE becomes active, and the output O changes to the new value of the ciphertext.

The output O keeps the last value of the ciphertext at the output, until the next encryption is completed. Before the first encryption is completed, this output should be equal to zero.
Assumptions

- $2r+4$ clock cycles are used to load round keys to internal RAMs
- one round of the main for loop of the pseudocode executes in one clock cycle
- you can access only one position of each internal memory of round keys per clock cycle

As a result, the encryption of a single input block $I$ should last $r+2$ clock cycles.
Memories of Round Keys – Contents
Case of $r=2$

$m=3$

\[
\begin{array}{ccc}
000 & 0 & S[0] \\
010 & 1 & S[2] \\
100 & 2 & S[4] \\
110 & 3 & S[6] \\
\end{array}
\]

\[
\begin{array}{ccc}
001 & 0 & S[1] \\
011 & 1 & S[3] \\
101 & 2 & S[5] \\
111 & 3 & S[7] \\
\end{array}
\]

\[2^{3^4} \times w = 4 \times w\]
Memories of Round Keys – Circuit
Counter

en \rightarrow en

ld \rightarrow ld

rst \rightarrow rst

clk \leftarrow clk

m-1 \rightarrow m-1

0 \rightarrow 0

i \rightarrow i

zi \rightarrow zi

==r
Interface with the Division into the Datapath and Controller
Pseudocode

Split input $I$ into four words, $I_3$, $I_2$, $I_1$, $I_0$, of the size of $w$ bits each

$A = I_3; B = I_2; C = I_1; D = I_0$

$B = B + S[0]$  
$D = D + S[1]$

for $i = 1$ to $r$ do

{}  

$T = (B*(2B + 1)) \ll k$
$U = (D*(2D + 1)) \ll k$

$A = ((A \oplus T) \ll U) + S[2i]$
$C = ((C \oplus U) \ll T) + S[2i + 1]$

$(A, B, C, D) = (B, C, D, A)$

{}  

$A = A + S[2r + 2]$  
$C = C + S[2r + 3]$  

$O = (A, B, C, D)$
Pseudocode

Split input I into four words, I3, I2, I1, I0, of the size of w bits each

A = I3
B = I2 + S[0]
C = I1
D = I0 + S[1]

for i = 1 to r do
    { 
    T = \((B \times (2B + 1)) \ll k\)
    U = \((D \times (2D + 1)) \ll k\)
    A' = (A \oplus T) \ll U + S[2i]
    C' = (C \oplus U) \ll T + S[2i + 1]
    (A, B, C, D) = (B, C', D, A')
    }
A = A + S[2r + 2]
C = C + S[2r + 3]
O = (A, B, C, D)
Pseudocode

Split input $I$ into four words, $I_3$, $I_2$, $I_1$, $I_0$, of the size of $w$ bits each

$$A = I_3; \quad B = I_2 + S_{2i};$$
$$C = I_1; \quad D = I_0 + S_{2ip1}$$

for $i = 1 \text{ to } r$ do

$$
\begin{align*}
T &= (B \times (2B + 1)) \ll k \\
U &= (D \times (2D + 1)) \ll k \\
A' &= ((A \oplus T) \ll U) + S[2i] \\
C' &= ((C \oplus U) \ll T) + S[2i + 1] \\
(A, B, C, D) &= (B, C', D, A')
\end{align*}
$$

end for

$$O = (A + S_{2i}, B, C + S_{2ip1}, D)$$
i=0

F write_I
T

A=I3; B=I2+S2i
C=I1; D=I0+S2ip1
i++

S_loop

A=B; B=C'
C=D; D=A'
i++

i==?r
F
T

i++

S_done

O=(A+S2i, B, C+S2ip1, D)
DONE

reset
S_wait

didi

F write_I
T

init, enr
eni

S_loop

enr

S_done

ENO
DONE
\[ A = I^3; B = I^2 + S^2i \]
\[ C = I^1; D = I^0 + S^2i + 1 \]

i++

A = B; B = C'
C = D; D = A'

i++

i == r

O = (A + S^2i, B, C + S^2i + 1, D)

DONE
Alternative Coding Styles
Traditional Coding Style

process(clock, reset)

Inputs

Next State function

Present State Register

Next State

clock reset

Mealy Output function

Moore Output function

Mealy Outputs

Moore Outputs

Mealy Outputs

Moore Outputs

concurrent statements
Alternative Coding Style 1

Process(Present State, Inputs)

Inputs

Next State function

Next State

Present State Register

Process(clock, reset)

clock
reset

Process(Present State, Inputs)

Process(Present State)

Mealy Output function

Mealy Outputs

Mealy Outputs

Moore Output function

Moore Outputs

Moore Outputs
library ieee;
use ieee.std_logic_1164.all;
entity mem_ctrl is
  port (  
     clk, reset: in std_logic;
     mem, rw, burst: in std_logic;
     oe, we, we_me: out std_logic);
end mem_ctrl ;

architecture mult_seg_arch of mem_ctrl is
  type mc_state_type is 
    (idle, read1, read2, read3, read4, write);
  signal state_reg, state_next: mc_state_type;
begin
  -- state register
  process(clk, reset)
  begin
    if (reset='1') then
      state_reg <= idle;
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
    end if;
  end process;
-- next-state logic

process(state_reg, mem, rw, burst)
begin
  case state_reg is
    when idle =>
      if mem='1' then
        if rw='1' then
          state_next <= read1;
        else
          state_next <= write;
        end if;
      else
        state_next <= idle;
      end if;
    when write =>
      state_next <= idle;
  end case;
end process;
when read1 =>
    if (burst='1') then
        state_next <= read2;
    else
        state_next <= idle;
    end if;
when read2 =>
    state_next <= read3;
when read3 =>
    state_next <= read4;
when read4 =>
    state_next <= idle;
end case;
end process;
-- moore output logic
process(state_reg)
begin
  we <= '0';  -- default value
  oe <= '0';  -- default value
  case state_reg is
    when idle =>
      we <= '1';
    when write =>
      we <= '1';
      oe <= '1';
    when read1 =>
      oe <= '1';
    when read2 =>
      oe <= '1';
    when read3 =>
      oe <= '1';
    when read4 =>
      oe <= '1';
  end case;
end process;
— mealy output logic

process(state_reg, mem, rw)
begin
  we_me <= '0'; — default value
  case state_reg is
    when idle =>
      if (mem='1') and (rw='0') then
        we_me <= '1';
      end if;
    when write =>
    when read1 =>
    when read2 =>
    when read3 =>
    when read4 =>
      end case;
  end process;
end mult_seg_arch;
Alternative Coding Style 2

Process(Present State, Inputs)

Process(clk, reset)
library ieee;
use ieee.std_logic_1164.all;
entity mem_ctrl is
port (  
   clk, reset: in std_logic;
   mem, rw, burst: in std_logic;
   oe, we, we_me: out std_logic);
end mem_ctrl ;

architecture mult_seg_arch of mem_ctrl is
  type mc_state_type is
    (idle, read1, read2, read3, read4, write);
  signal state_reg, state_next: mc_state_type;
begin
  -- state register
  process(clk, reset)
  begin
    if (reset='1') then
      state_reg <= idle;
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
    end if;
  end process;
}
process(state_reg, mem, rw, burst)
begin
  oe <= '0';  — default values
  we <= '0';
  we_me <= '0';
  case state_reg is
    when idle =>
      if mem='1' then
        if rw='1' then
          state_next <= read1;
        else
          state_next <= write;
        end if;
      else
        state_next <= idle;
      end if;
    when write =>
      state_next <= idle;
      we <= '1';
when read1 =>
  if (burst='1') then
    state_next <= read2;
  else
    state_next <= idle;
  end if;
  oe <= '1';
when read2 =>
  state_next <= read3;
  oe <= '1';
when read3 =>
  state_next <= read4;
  oe <= '1';
when read4 =>
  state_next <= idle;
  oe <= '1';
end case;
end process;