System-on-Chip Design
Finit-State-Machine Data-Path Model

Prof. Dr. Marcel Jacomet
Bern University of Applied Sciences
Bfh-Ti HuCE-microLab, Biel/Bienne
Marcel.Jacomet@bfh.ch

HuCE.ch/microLab

February 16, 2018
Chapter: Finite-State-Machine Data-Path Model

Introduction
Architecture Model
Design Flow
Conclusion
Architecture Philosophy

- FSMD architecture model is composed of 2 blocks
  - finite state machine (FSM)
  - data-path (D)

- goal of FSMD architecture model
  - structured design approach
  - ressource optimization
  - readability, documentation

- FSM characteristics
  - manager
  - controlling, taking decisions, initiating sub-tasks

- data-path characteristics
  - worker, specialist
  - executing, calculating, storing & moving data
Architecture Philosophy

- FSMD architecture model is composed of 2 blocks
  - finite state machine (FSM)
  - data-path (D)
- goal of FSMD architecture model
  - structured design approach
  - ressource optimization
  - readability, documentation
- FSM characteristics
  - manager
  - controlling, taking decisions, initiating sub-tasks
- data-path characteristics
  - worker, specialist
  - executing, calculating, storing & moving data
Architecture Philosophy

- FSMD architecture model is composed of 2 blocks
  - finite state machine (FSM)
  - data-path (D)
- goal of FSMD architecture model
  - structured design approach
  - resource optimization
  - readability, documentation
- FSM characteristics
  - manager
  - controlling, taking decisions, initiating sub-tasks
- data-path characteristics
  - worker, specialist
  - executing, calculating, storing & moving data
Architecture Philosophy

- FSMD architecture model is composed of 2 blocks
  - finite state machine (FSM)
  - data-path (D)
- goal of FSMD architecture model
  - structured design approach
  - resource optimization
  - readability, documentation
- FSM characteristics
  - manager
  - controlling, taking decisions, initiating sub-tasks
- data-path characteristics
  - worker, specialist
  - executing, calculating, storing & moving data
Architecture Model

- FSMD architecture model
  - based on FSM model and data-path model
    - interface: inputs
    - interface: outputs
    - control between FSM and data-path

![FSMD Architecture Model Diagram]

- FSMD
  - data-path (RTL logic)
  - control path (finite state machine)
Architecture Model

- FSMD architecture model
  - based on FSM model and data-path model
  - interface: inputs
  - interface: outputs
  - control between FSM and data-path
Architecture Model

- FSMD architecture model
  - based on FSM model and data-path model
  - interface: inputs
  - interface: outputs
  - control between FSM and data-path

![Diagram of FSMD architecture model]

- FSMD:
  - data-path (RTL logic)
  - control path (finite state machine)
  - inputs (sensors)
  - outputs (actuators)
Architecture Model

- FSMD architecture model
  - based on FSM model and data-path model
  - interface: inputs
  - interface: outputs
  - control between FSM and data-path

![Architecture Model Diagram]
FSM Mealy

- FSM Mealy machine
  - outputs are dependent on inputs and states
  - equations
  - structure (most general)

\[
\begin{align*}
  s[k + 1] &= f(i[k], s[k]) \\
  o[k] &= g(i[k], s[k])
\end{align*}
\]
**FSM Moore**

- **FSM Moore machine**
  - outputs are dependent on states only
  - equations
  - structure (functional restriction)

\[
s[k + 1] = f(i[k], s[k]) \\
o[k] = g(s[k])
\]

```
transition logic
```

```
state reg
```

```
output logic
```

- inputs: \( i[k] \)
- state: \( s[k] \)
- output: \( o[k] \)
FSM Medwedjew

- FSM Moore machine
  - outputs are the states
  - equations
  - structure (no hazards at outputs)

\[
s[k + 1] = f(i[k], s[k]) \\
o[k] = s[k]
\]
Datapath Elements

- A typical data-path consists of 3 types of basic elements
  - communication: bus, multiplexor, de-multiplexor
  - functional units and operators: adder, multiplier, shifter, ALU, comparator, etc.
  - storage: flip-flop, register, register-files, FIFO and memory elements like RAM, etc.
Datapath Memory Elements

- memory elements store new values at every clock cycle
- to give the FSM full control to the datapath, the datapath memory elements need to be upgraded with an *enable* control input

![Diagram of register with enable input](image-url)
Datapath Memory Elements

- memory elements store new values at every clock cycle
- to give the FSM full control to the datapath, the datapath memory elements need to be upgraded with an *enable* control input

```
Datapath Memory Elements

- memory elements store new values at every clock cycle
- to give the FSM full control to the datapath, the datapath memory elements need to be upgraded with an *enable* control input

```

![Datapath Memory Elements Diagram](image)
Datapath Memory Elements

- Memory elements store new values at every clock cycle.
- To give the FSM full control to the datapath, the datapath memory elements need to be upgraded with an *enable* control input.

![Datapath Memory Elements Diagram]

- *Memory elements store new values at every clock cycle.*
- *To give the FSM full control to the datapath, the datapath memory elements need to be upgraded with an enable control input.*

![Datapath Memory Elements Diagram]

- *Memory elements store new values at every clock cycle.*
- *To give the FSM full control to the datapath, the datapath memory elements need to be upgraded with an enable control input.*

![Datapath Memory Elements Diagram]

- *Memory elements store new values at every clock cycle.*
- *To give the FSM full control to the datapath, the datapath memory elements need to be upgraded with an enable control input.*

![Datapath Memory Elements Diagram]

- *Memory elements store new values at every clock cycle.*
- *To give the FSM full control to the datapath, the datapath memory elements need to be upgraded with an enable control input.*

![Datapath Memory Elements Diagram]

- *Memory elements store new values at every clock cycle.*
- *To give the FSM full control to the datapath, the datapath memory elements need to be upgraded with an enable control input.*

![Datapath Memory Elements Diagram]

- *Memory elements store new values at every clock cycle.*
- *To give the FSM full control to the datapath, the datapath memory elements need to be upgraded with an enable control input.*

![Datapath Memory Elements Diagram]

- *Memory elements store new values at every clock cycle.*
- *To give the FSM full control to the datapath, the datapath memory elements need to be upgraded with an enable control input.*

![Datapath Memory Elements Diagram]

- *Memory elements store new values at every clock cycle.*
- *To give the FSM full control to the datapath, the datapath memory elements need to be upgraded with an enable control input.*

![Datapath Memory Elements Diagram]

- *Memory elements store new values at every clock cycle.*
- *To give the FSM full control to the datapath, the datapath memory elements need to be upgraded with an enable control input.*

![Datapath Memory Elements Diagram]

- *Memory elements store new values at every clock cycle.*
- *To give the FSM full control to the datapath, the datapath memory elements need to be upgraded with an enable control input.*

![Datapath Memory Elements Diagram]
FSMD Design Flow

- a tutorial design shall serve as vehicle for a practical approach: Black Jack Player
- a key element in the FSMD design procedure is the interface definition
- design steps

1. definition of the algorithm
2. FSMD interface definition
3. datapath design
4. datapath interface definition
5. FSM interface definition
6. FSM state definition
7. FSM design
8. HDL coding
9. test-bench design and simulation
FSMD Design Flow

▶ a tutorial design shall serve as vehicle for a practical approach: Black Jack Player
▶ a key element in the FSMD design procedure is the interface definition
▶ design steps
  1. definition of the algorithm
  2. FSMD interface definition
  3. datapath design
  4. datapath interface definition
  5. FSM interface definition
  6. FSM state definition
  7. FSM design
  8. HDL coding
  9. test-bench design and simulation
FSMD Design Flow

- a tutorial design shall serve as vehicle for a practical approach: Black Jack Player
- a key element in the FSMD design procedure is the interface definition
- design steps
  1. definition of the algorithm
  2. FSMD interface definition
  3. datapath design
  4. datapath interface definition
  5. FSM interface definition
  6. FSM state definition
  7. FSM design
  8. HDL coding
  9. test-bench design and simulation
a tutorial design shall serve as vehicle for a practical approach: Black Jack Player

a key element in the FSMD design procedure is the interface definition

design steps

1. definition of the algorithm
2. FSMD interface definition
3. datapath design
4. datapath interface definition
5. FSM interface definition
6. FSM state definition
7. FSM design
8. HDL coding
9. test-bench design and simulation
FSMD Design Flow

- a tutorial design shall serve as vehicle for a practical approach: Black Jack Player
- a key element in the FSMD design procedure is the interface definition
- design steps
  1. definition of the algorithm
  2. FSMD interface definition
  3. datapath design
  4. datapath interface definition
  5. FSM interface definition
  6. FSM state definition
  7. FSM design
  8. HDL coding
  9. test-bench design and simulation
FSMD Design Flow

- a tutorial design shall serve as vehicle for a practical approach: Black Jack Player
- a key element in the FSMD design procedure is the interface definition
- design steps
  1. definition of the algorithm
  2. FSMD interface definition
  3. datapath design
  4. datapath interface definition
  5. FSM interface definition
  6. FSM state definition
  7. FSM design
  8. HDL coding
  9. test-bench design and simulation
FSMD Design Flow

- a tutorial design shall serve as vehicle for a practical approach: Black Jack Player
- a key element in the FSMD design procedure is the interface definition
- design steps
  1. definition of the algorithm
  2. FSMD interface definition
  3. datapath design
  4. datapath interface definition
  5. FSM interface definition
  6. FSM state definition
  7. FSM design
  8. HDL coding
  9. test-bench design and simulation
FSMD Design Flow

- a tutorial design shall serve as vehicle for a practical approach: Black Jack Player
- a key element in the FSMD design procedure is the interface definition
- design steps
  1. definition of the algorithm
  2. FSMD interface definition
  3. datapath design
  4. datapath interface definition
  5. FSM interface definition
  6. FSM state definition
  7. FSM design
  8. HDL coding
  9. test-bench design and simulation
FSMD Design Flow

- a tutorial design shall serve as vehicle for a practical approach: Black Jack Player
- a key element in the FSMD design procedure is the interface definition
- design steps
  1. definition of the algorithm
  2. FSMD interface definition
  3. datapath design
  4. datapath interface definition
  5. FSM interface definition
  6. FSM state definition
  7. FSM design
  8. HDL coding
  9. test-bench design and simulation
Design Step 1: Algorithm Definition

- goal of the Black Jack Player
  - get as close as possible to 21 points
  - lost of summed points exceed 21

- game restrictions:
  - the cards have values between 2 to 9, 10 for the boy, queen or king and 11 for the ace

- game rules for the player:
  - he asks for as many cards as he needs
  - the ace can be treated as 1 or as 11 points card

- player behavior (algorithm):
  - he asks for cards as long as the summed-up points do not exceed 16
  - he initially treats the ace always as 11 points card
  - when exceeding 21 points, he treats possible ace a 1 point card to get a second chance
  - optional: when having a 11 point ace on the pile, he risks to ask for cards as long as the summed-up points are below 18 points
Design Step 1: Algorithm Definition

- goal of the Black Jack Player
  - get as close as possible to 21 points
  - lost of summed points exceed 21

- game restrictions:
  - the cards have values between 2 to 9, 10 for the boy, queen or king and 11 for the ace

- game rules for the player:
  - he asks for as many cards as he needs
  - the ace can be treated as 1 or as 11 points card

- player behavior (algorithm):
  - he asks for cards as long as the summed-up points do not exceed 16
  - he initially treats the ace always as 11 points card
  - when exceeding 21 points, he treats possible ace a 1 point card to get a second chance
  - optional: when having a 11 point ace on the pile, he risks to ask for cards as long as the summed-up points are below 18 points
Design Step 1: Algorithm Definition

- goal of the Black Jack Player
  - get as close as possible to 21 points
  - lost of summed points exceed 21

- game restrictions:
  - the cards have values between 2 to 9, 10 for the boy, queen or king and 11 for the ace

- game rules for the player:
  - he asks for as many cards as he needs
  - the ace can be treated as 1 or as 11 points card

- player behavior (algorithm):
  - he asks for cards as long as the summed-up points do not exceed 16
  - he initially treats the ace always as 11 points card
  - when exceeding 21 points, he treats possible ace a 1 point card to get a second chance
  - optional: when having a 11 point ace on the pile, he risks to ask for cards as long as the summed-up points are below 18 points
Design Step 1: Algorithm Definition

▶ goal of the Black Jack Player
  ◦ get as close as possible to 21 points
  ◦ lost of summed points exceed 21

▶ game restrictions:
  ◦ the cards have values between 2 to 9, 10 for the
    boy, queen or king and 11 for the ace

▶ game rules for the player:
  ◦ he asks for as many cards as he needs
  ◦ the ace can be treated as 1 or as 11 points card

▶ player behavior (algorithm):
  ◦ he asks for cards as long as the summed-up points
    do not exceed 16
  ◦ he initially treats the ace always as 11 points card
  ◦ when exceeding 21 points, he treats possible ace a
    1 point card to get a second chance
  ◦ optional: when having a 11 point ace on the pile,
    he risks to ask for cards as long as the summed-up
    points are below 18 points
Design Step 2: Interface Definition

- defining the interface of the overall FSMD
- defining signals: system, data, control, status, signal width
Design Step 2: Interface Definition

- defining the interface of the overall FSMD
- defining signals: **system**, data, control, status, signal width
Design Step 2: Interface Definition

- defining the interface of the overall FSMD
- defining signals: system, data, control, status, signal width
Design Step 2: Interface Definition

- defining the interface of the overall FSMD
- defining signals: system, data, control, status, signal width
Design Step 2: Interface Definition

- defining the interface of the overall FSMD
- defining signals: system, data, control, **status**, signal width

Diagram:
- **FSMD**
- **cardReady**
- **newCard**
- **cardValue**
- **score**
- **clk**
- **lost**
- **finished**
- **start**
Design Step 2: Interface Definition

- defining the interface of the overall FSMD
- defining signals: system, data, control, status, **signal width**

![Diagram of Black Jack Player with signals and interface definitions](image-url)
Design Step 3: Datapath Design

- datapath has to be able to execute all functional operators of the algorithm
- clearly separate control-path and datapath tasks as seen in the manager/worker analogy model
- use memory elements for storing data
- use buses, multiplexers for moving data
- use functional operators for data manipulation and calculations
- use comparators for status information
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths

Design Step 3: Datapath Design (cont)
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

▶ loading card value into register
▶ is card an ace?
▶ accumulating the card values
▶ are the accumulated card values above 16? or above 21?
▶ visualizing score
▶ second chance is ace in pile
▶ defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 3: Datapath Design (cont)

- loading card value into register
- is card an ace?
- accumulating the card values
- are the accumulated card values above 16? or above 21?
- visualizing score
- second chance is ace in pile
- defining datapath widths
Design Step 4: Datapath Interface Definition

- defining the interface of the datapath block
- defining active level of control/status signals
Design Step 4: Datapath Interface Definition

- defining the interface of the datapath block
- defining active level of control/status signals
Design Step 5: FSM Interface Definition

- defining the inputs and outputs of the FSM
- interface FSMD = interface datapath + interface FSM
Design Step 5: FSM Interface Definition

- defining the inputs and outputs of the FSM
- interface FSMD = interface datapath + interface FSM
Design Step 5: FSM Interface Definition

- defining the inputs and outputs of the FSM
- interface FSMD = interface datapath + interface FSM
Design Step 5: FSM Interface Definition

- defining the inputs and outputs of the FSM
- interface FSMD = interface datapath + interface FSM
Design Step 5: FSM Interface Definition

- defining the inputs and outputs of the FSM
- interface FSMD = interface datapath + **interface FSM**

---

**Datapath**

- cardValue \((3:0)\) → FSMD
- clk → FSMD
- start → FSMD

**FSM**

- newCard → FSM
- lost → FSM
- finished → FSM

**Interface FSMD**

- interface datapath + **interface FSM**

---

**Card Value**

- score \((3:0)\)
- newCard
- lost
- finished
Design Step 5: FSM Interface Definition

- defining the inputs and outputs of the FSM
- interface FSMD = interface datapath + interface FSM

FSMD

Datapath

FSM

interface defined
iFSMD = iDP + iFSM

 clk

 start

 cardValue (3:0)
Design Step 5: FSM Interface Definition (cont)

- define input and output signals of FSM
  - FSM inputs: condition signals
  - FSM outputs: datapath and output control signals

![Diagram of FSM interface]

- clk
- rst
- cardReady
- cmp1
- cmp11
- cmp16
- cmp21
- enaLoad
- enaAdd
- enaScore
- newReady
- lost
- finished

Note: The diagram shows the interface of the FSM with its input and output signals.
Design Step 5: FSM Interface Definition (cont)

- define input and output signals of FSM
- FSM inputs: condition signals
- FSM outputs: datapath and output control signals

![Diagram of FSM interface]

- **cmp11**, **cmp16**, **cmp21**, **sel**, **enaLoad**, **enaAdd**, **enaScore**
- **cardReady**
- **newReady**
- **lost**
- **finished**
- **clk**
- **rst**
Design Step 5: FSM Interface Definition (cont)

- define input and output signals of FSM
- FSM inputs: condition signals
- FSM outputs: datapath and output control signals
Design Step 5: FSM Interface Definition (cont)

- define input and output signals of FSM
- FSM inputs: condition signals
- FSM outputs: datapath and output control signals
Design Step 5: FSM Interface Definition (cont)

- define input and output signals of FSM
- FSM inputs: condition signals
- FSM outputs: datapath and output control signals

![FSM Diagram]

**FSM input:** cardReady, cmp11, cmp16, cmp21

**FSM output:** finished, lost, newCard, sel, enaLoad, enaAdd, enaScore
Design Step 6: FSM State Definition

- draw a skeleton state with placeholders for the state name and output signals

<table>
<thead>
<tr>
<th>FSM output:</th>
<th>finished</th>
<th>lost</th>
<th>newCard</th>
<th>sel</th>
<th>enaLoad</th>
<th>enaAdd</th>
<th>enaScore</th>
</tr>
</thead>
</table>
Design Step 6: FSM State Definition

- draw a skeleton state with placeholders for the state name and output signals

<table>
<thead>
<tr>
<th>FSM output:</th>
<th>finished</th>
<th>lost</th>
<th>newCard</th>
<th>sel</th>
<th>enaLoad</th>
<th>enaAdd</th>
<th>enaScore</th>
</tr>
</thead>
</table>

name

output signals: finished, lost, newCard, sel, enaLoad, enaAdd, enaScore
Design Step 7: FSM Design (Timing)

- single clock cycle schema
- More type FSM
- FSMD timing diagramm
  - registered values are available in next state
  - combinational values are available in current state

![FSM Timing Diagram](image-url)
Design Step 7: FSM Design (Timing)

- single clock cycle schema
- More type FSM
- FSMD timing diagramm
  - registered values are available in next state
  - combinational values are available in current state

![FSM Diagram](image-url)
Design Step 7: FSM Design (Timing)

- single clock cycle schema
- More type FSM
- FSMD timing diagramm
  - registered values are available in next state
  - combinational values are available in current state

```
state
LoadRegState CheckValState CheckedState OpenDataState IdleState
clk
enable(FSM)
registers(D)
inform(D)
select(FSM)
data bus (D)
new value
data
```
Design Step 7: FSM Design (Timing)

- single clock cycle schema
- More type FSM
- FSMD timing diagramm
  - registered values are available in next state
  - combinational values are available in current state
Design Step 7: FSM Design (Timing)

- single clock cycle schema
- More type FSM
- FSMD timing diagramm
  - registered values are available in next state
  - combinational values are available in current state
Design Step 7: FSM Design (Timing)

- single clock cycle schema
- More type FSM
- FSMD timing diagramm
  - registered values are available in next state
  - combinational values are available in current state
Design Step 7: FSM Design (Timing)

- single clock cycle schema
- More type FSM
- FSMD timing diagramm
  - registered values are available in next state
  - combinational values are available in current state

```
state
<table>
<thead>
<tr>
<th>LoadRegState</th>
<th>CheckValState</th>
<th>CheckedState</th>
<th>OpenDataState</th>
<th>IdleState</th>
</tr>
</thead>
</table>
clk           |              |              |               |           |
enable(FSM)    |              |              |               |           |
registers(D)   |              |              | new value     |           |
inform(D)      |              |              |               |           |
select(FSM)    |              |              |               |           |
data bus (D)   | data         |              |               |           |
```
Design Step 7: FSM Design (Timing)

- single clock cycle schema
- More type FSM
- FSMD timing diagramm
  - registered values are available in next state
  - combinational values are available in current state

```
state
LoadRegState  CheckValState  CheckedState  OpenDataState  IdleState
clk
enable(FSM)
registers(D)
inform(D)
select(FSM)
data bus (D)
```

- new value
- data
Design Step 7: FSM Design (Timing)

- single clock cycle schema
- More type FSM
- FSMD timing diagramm
  - registered values are available in next state
  - combinational values are available in current state
Design Step 7: FSM Design

- design the Moore type state diagram
- conditions on arrows are FSM inputs
- output values are defined in states
- use lightning arrow for asynchronous reset
Design Step 8: Datapath Coding (VHDL)

- all registers with associated logic are placed in one process (identical clock and asynchronous reset)
- loosely coupled combinatorial logic can be coded with conditional signal assignments

```
process (clk, rst) begin
    if (rst = '0') then
        regLoad <= "00000";
        regAdd <= "00000";
        regScore <= "00000";
    elsif (clk'event and clk = '0') then
        if (enaAdd = '1') then
            regAdd <= regAdd + regLoad;
        end if;
    end if;
    ... end process;
```

cmp11 <= '1' when (regLoad = "01011"), else '0';
cmp16 <= '1' when (regAdd > "10000"), else '0';
cmp21 <= '1' when (regAdd > "10110"), else '0';
Design Step 8: Datapath Coding (VHDL)

- all registers with associated logic are placed in one process (identical clock and asynchronous reset)
- loosely coupled combinatorial logic can be coded with conditional signal assignments

```vhdl
process (clk, rst)
begin
  if (rst = '0') then
    regLoad <= "00000";
    regAdd  <= "00000";
    regScore <= "00000";
  elsif (clk'event and clk = '0') then
    if (enaAdd = '1') then
      regAdd <= regAdd + regLoad;
    end if;
    ...
  end if;
end process;
```

```
cmp11 <= '1' when (regLoad = "01011"), else '0';
cmp16 <= '1' when (regAdd > "10000"), else '0';
cmp21 <= '1' when (regAdd > "10101"), else '0';
```
Design Step 8: Datapath Coding (VHDL)

- all registers with associated logic are placed in one process (identical clock and asynchronous reset)
- loosely coupled combinatorial logic can be coded with conditional signal assignments

```vhdl
process (clk, rst) begin
  if (rst = '0') then
    regLoad <= "00000";
    regAdd <= "00000";
    regScore <= "00000";
  elsif (clk'event and clk = '0') then
    if (enaAdd = '1') then
      regAdd <= regAdd + regLoad;
    end if;

    . . .
  end if;
end process;
```

```
cmp11 <= '1' when (regLoad = "01011"), else '0';
cmp16 <= '1' when (regAdd > "10000"), else '0';
cmp21 <= '1' when (regAdd > "10101"), else '0';
```
Design Step 8: Datapath Coding (VHDL)

- all registers with associated logic are placed in one process (identical clock and asynchronous reset)
- loosely coupled combinatorial logic can be coded with conditional signal assignments

```vhdl
process(clk, rst)
begin
  if (rst = '0') then
    regLoad <= "00000";
    regAdd <= "00000";
    regScore <= "00000";
  elsif (clk'event and clk = '0') then
    if (enaAdd = '1') then
      regAdd <= regAdd + regLoad;
    end if;
    ...
  end if;
end process;
```

```
cmp11 <= '1' when (regLoad = "01011") else '0';
cmp16 <= '1' when (regAdd > "10000") else '0';
cmp21 <= '1' when (regAdd > "10101") else '0';
```
Design Step 8: Datapath Coding (VHDL)

- all registers with associated logic are placed in one process (identical clock and asynchronous reset)
- loosely coupled combinatorial logic can be coded with conditional signal assignments

```vhdl
process(clk, rst)
begin
  if (rst = '0') then
    regLoad <= "00000";
    regAdd <= "00000";
    regScore <= "00000";
  elsif (clk 'event and clk = '0') then
    if (enaAdd = '1') then
      regAdd <= regAdd + regLoad;
    end if;
  end if;
  ...
end process;
```

```vhdl
cmp11 <= '1' when (regLoad = "01011"), else '0';
cmp16 <= '1' when (regAdd > "10000"), else '0';
cmp21 <= '1' when (regAdd > "10101"), else '0';
```
Design Step 8: FSM Coding (VHDL)

- one clocked process is used for the state transition
- one combinatorial process is used for the state dependent output assignment

```vhdl
process (clk, rst) begin
  if (rst = '0') then
    state <= StartState;
  elsif (clk'event and clk = '0') then
    case state is
      when CallCardState =>
        if (cardReady = '1') then
          state <= LoadCardState;
        end if;
      when others =>
        state <= IllegalState;
    end case;
  end if;
end process;
```

```vhdl
process (state) begin
  case state is
    when CallCardState =>
      outvec <= "001—00";
    when others =>
      outvec <= "UUUUUUU";
  end case;
end process;
```

```vhdl```

```vhdl```

```vhdl```

```vhdl```

```vhdl```

```vhdl```

```vhdl```

```vhdl```

```vhdl```

```vhdl```

```vhdl```

```vhdl```

```vhdl```

```vhdl```

```vhdl```

```vhdl```

```vhdl```

```vhdl```
Design Step 8: FSM Coding (VHDL)

- one clocked process is used for the state transition
- one combinatorial process is used for the state dependent output assignment

```vhdl
process(clk, rst)
begin
  if (rst = '0') then
    state <= StartState;
  elsif (clk 'event and clk = '0') then
    case state is
      when CallCardState =>
        if (cardReady = '1') then
          state <= LoadCardState;
        end if;
        ... when others =>
          state <= IllegalState;
        end case;
  end if;
end process;

process(state)
begin
  case state is
    when CallCardState =>
      output <= "001---00";
    when others =>
      output <= "UUUUUU";
  end case;
end process;
```

```vhdl```
```process(clk, rst)
begin
  if (rst = '0') then
    state <= StartState;
  elsif (clk 'event and clk = '0') then
    case state is
      when CallCardState =>
        if (cardReady = '1') then
          state <= LoadCardState;
        end if;
        ... when others =>
          state <= IllegalState;
        end case;
  end if;
end process;
```

```process(state)
begin
  case state is
    when CallCardState =>
      output <= "001---00";
    when others =>
      output <= "UUUUUU";
  end case;
end process;```
Design Step 8: FSM Coding (VHDL)

- one clocked process is used for the state transition
- one combinatorial process is used for the state dependent output assignment

```
process (clk, rst)
begin
  if (rst = '0') then
    state <= StartState;
  elsif (clk'event and clk = '0') then
    case state is
      when CallCardState =>
        if (cardReady = '1') then
          state <= LoadCardState;
        end if;
      ... when others =>
        state <= IllegalState;
    end case;
  end if;
end process;
```

```
process (state)
begin
  case state is
    when CallCardState =>
      outvec <= "001---00";
      ... when others =>
      outvec <= "UUUUUUU";
    finished <= outvec(6);
    lost <= outvec(5);
    ... 
  end case;
end process;
```
Design Step 8: FSM Coding (VHDL)

- one clocked process is used for the state transition
- one combinatorial process is used for the state dependent output assignment

```
process(clk, rest)
begin
  if (rst = '0') then
    state <= StartState;
  elsif (clk 'event and clk = '0') then
    case state is
      when CallCardState =>
        if (cardReady = '1') then
          state <= LoadCardState;
        end if;
      when others =>
        state <= IllegalState;
    end case;
  end if;
end process;

process(state)
begin
  case state is
    when CallCardState =>
      outvec <= "001----00";
    when others =>
      outvec <= "UUUUUUU";
    end case;
  finished <= outvec(6);
  lost <= outvec(5);
end process;
```
Design Step 8: FSM Coding (VHDL)

- one clocked process is used for the state transition
- one combinatorial process is used for the state dependent output assignment

```vhdl
process(clk, rst) begin
  if (rst = '0') then
    state <= StartState;
  elsif (clk'event and clk = '0') then
    case state is
      when CallCardState =>
        if (cardReady = '1') then
          state <= LoadCardState;
        end if;
      when others =>
        state <= IllegalState;
    end case;
  end if;
end process;
```

```vhdl
process(state) begin
  case state is
    when CallCardState =>
      outvec <= "001---00";
    when others =>
      outvec <= "UUUUUUU";
  end case;
end process;
```

```vhdl
process(clk, rst) begin
  if (rst = '0') then
    state <= StartState;
  elsif (clk'event and clk = '0') then
    case state is
      when CallCardState =>
        if (cardReady = '1') then
          state <= LoadCardState;
        end if;
      when others =>
        state <= IllegalState;
    end case;
  end if;
end process;
```

```vhdl
process(state) begin
  case state is
    when CallCardState =>
      outvec <= "001---00";
    when others =>
      outvec <= "UUUUUUU";
  end case;
end process;
```
Design Step 8: Final Design

FSMD

Datapath

FSM

Card Value score (3:0) (4:0)

new Card cardReady

lost

finished

clk

start

Design Step 8: Final Design

FSMD

Datapath

FSM

Card Value score (3:0) (4:0)

new Card cardReady

lost

finished

clk

start
Design Step 8: Final Design

FSMD

Datapath

FSM

cardValue (3:0)

(4:0) score

cardReady

clk

start

newCard

lost

finished
Design Step 9: Testbench

- compare a testbench with the HuCE-microLab:
  - there are chips and boards needed to be tested
  - there is a nice measurement equipment
  - there are skilled and hard working engineers
  - there are no signals coming in or going out of the lab windows
Design Step 9: Testbench

- compare a testbench with the HuCE-microLab:
  - there are chips and boards needed to be tested
  - there is a nice measurement equipment
  - there are skilled and hard working engineers
  - there are no signals coming in or going out of the lab windows

![Diagram of testbench and device-under-test](image-url)
Design Step 9b: Testbench Test-Cycle

- cycle based test
- apply input patterns at beginning of test cycle
- capture response after active clock edge
Design Step 9b: Testbench Test-Cycle

- cycle based test
- apply input patterns at beginning of test cycle
- capture response after active clock edge
Design Step 9b: Testbench Test-Cycle

- cycle based test
  - apply input patterns at beginning of test cycle
  - capture response after active clock edge

![Diagram of test cycle]

**Test Cycle**
- **clk**
- **inputs**
- **outputs**
- **sync**
  - stable
Design Step 9b: Testbench Test-Cycle

- cycle based test
  - apply input patterns at beginning of test cycle
  - capture response after active clock edge
Design Step 9b: Testbench Test-Cycle

- cycle based test
- apply input patterns at beginning of test cycle
- capture response after active clock edge
Design Step 9b: Testbench Test-Cycle

- cycle based test
- apply input patterns at beginning of test cycle
- capture response after active clock edge
Errors and Pitfalls

- asynchronous external inputs to FSM provoke state hazards
  - assume a 0.1 ns hazard can be captured in a state register
  - one out of 100 states is sensitive for a hazard
  - assume 100 MHz clock frequency
  - we might get 10'000 errors per second !!

- input synchronization for all "external" non-synchronous FSM inputs

---

**Diagram Description:**
- **Input:** $i[k]$
- **Transition Logic:** $s[k+1]$
- **State Register:** $s[k]$
- **Output Logic:** $o[k]$

---
Errors and Pitfalls

- asynchronous external inputs to FSM provoke state hazards
  - assume a 0.1 ns hazard can be captured in a state register
  - one out of 100 states is sensitive for a hazard
  - assume 100 MHz clock frequency
  - we might get 10'000 errors per second !!

- input synchronization for all "external" non-synchronous FSM inputs
Errors and Pitfalls

- asynchronous external inputs to FSM provoke state hazards
  - assume a 0.1 ns hazard can be captured in a state register
  - one out of 100 states is sensitive for a hazard
  - assume 100 MHz clock frequency
  - we might get 10'000 errors per second !!

- input synchronization for all "external" non-synchronous FSM inputs
Errors and Pitfalls

- asynchronous external inputs to FSM provoke state hazards
  - assume a 0.1 ns hazard can be captured in a state register
  - one out of 100 states is sensitive for a hazard
  - assume 100 MHz clock frequency
  - we might get 10'000 errors per second !!

- input synchronization for all "external" non-synchronous FSM inputs
Errors and Pitfalls

- asynchronous external inputs to FSM provoke state hazards
  - assume a 0.1 ns hazard can be captured in a state register
  - one out of 100 states is sensitive for a hazard
  - assume 100 MHz clock frequency
  - we might get 10'000 errors per second !!

- input synchronization for all "external" non-synchronous FSM inputs
Conclusion

- FSMD architecture supports structured design approach
- 9 design step approach for FSMD design presented
- task re-distribution between FSM and datapath is crucial
  - example: ace counting in BlackJack player, who should do it?
- manager/specialist analogy is used to assign sub-tasks to control-path (manager) and datapath (specialized workers)
Conclusion

▸ FSMD architecture supports structured design approach
▸ 9 design step approach for FSMD design presented
▸ task re-distribution between FSM and datapath is crucial
  ▸ example: ace counting in BlackJack player, who should do it?
▸ manager/specialist analogy is used to assign sub-tasks to
  control-path (manager) and datapath (specialized workers)
Conclusion

- FSMD architecture supports structured design approach
- 9 design step approach for FSMD design presented
- task re-distribution between FSM and datapath is crucial
  - example: ace counting in BlackJack player, who should do it?
- manager/specialist analogy is used to assign sub-tasks to control-path (manager) and datapath (specialized workers)
Conclusion

- FSMD architecture supports structured design approach
- 9 design step approach for FSMD design presented
- Task re-distribution between FSM and datapath is crucial
  - Example: ace counting in BlackJack player, who should do it?
- Manager/specialist analogy is used to assign sub-tasks to control-path (manager) and datapath (specialized workers)