Course Summary

1. Course overview
2. Intro to PICOBLAZE, C and Number systems and Boolean Algebra
3. Course overview with microprocessor MU0 (I)
4. Course overview with microprocessor MU0 (II)
5. Verilog HDL
6. Digital system components using schematics and Verilog
7. Combinational logic standard forms. Karnaugh maps
8. Combinational ccts and configurable logic devices
9. Simple Sequential circuits, flip flops
10. Sequential circuits, counters, registers, memories
11. Non-ideal effects in digital circuits
12. Finite State Machines
13. Design of FSMs
14. Register Transfer Level Systems (RTL) systems
15. Design of RTL Systems
16. Non-ideal effects in complex digital systems (Karnaugh maps)
17. Complex RTL design
18. The PICOBLAZE Softcore
19. Assembly language programming
20. C and Assembly
21. Other microprocessor architectures
Datapaths
Datapath

Figure 23: Simple datapath with one accumulator [Gajski].
The register may be enhanced by asynchronous Preset and Clear (Reset) signals, which are not controlled by the clock signal.

![Diagram of a register with labels](image)

(a) Graphic symbol

![Schematic of a 4-bit register](image)

(b) Register schematic

Figure 1: A 4-bit register [Gajski].
Figure 2: A 4-bit register with asynchronous Preset and Clear [Gajski].
Shift Register with Load

(a) Graphic symbol

(b) Operation table

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load 0</td>
<td>No change</td>
</tr>
<tr>
<td>Load 1</td>
<td>$I_3$ $I_2$ $I_1$ $I_0$</td>
</tr>
</tbody>
</table>

3213: Digital Systems & Microprocessors: L#14_15
Register files

A register file has $2^N$ registers of $N$ DFFs each.

- The registers are arranged as a 2-dimensional array of register-file cells (RFCs).
- In addition, it has read/write decoders and output driving logic.
- Writing is controlled by the Write-Enable (WE) signal.
  - At any time, we can write into only one register (row), unless it has multiple write ports.
- Reading is controlled by the Read-Enable (RE) signal.
  - At any time, we can read from only one register.
- Reading from and writing into the same register at the same time is not allowed.
- The primary advantage of a register file is regularity, which reduces routing (wiring) complexity.
Register files

(a) Register file cell
(b) Graphic symbol
(c) Logic Schematic

Figure 10: Register file with 1 write port and 1 read port [Gajski].
Random Access Memories

* A RAM is organized as an array of $2^N$ rows with $M$ bits stored in each row.

- The size of the RAM is $2^N \times M$ bits—it has $N$ address lines, $M$ input data lines, and $M$ output data lines (see Fig. 12).
- The input data lines can be the same as the output data lines, i.e., the data lines can be bidirectional.

* A memory cell (MC) can be considered as a clocked D latch with an AND gate and an output driver

- For a static RAM (SRAM), MC is constructed by 6 transistors, using cross-coupled inverters to serve as a latch, and implementing the input AND gate and the output driver with one transistor each.
- For a dynamic RAM (DRAM), MC is constructed by only 1 transistor.
  * The latch is implemented by a capacitor.
  * It needs to be refreshed periodically.
  * It has high density (therefore low cost).
SRAM cell

The 0 or 1 state can be written and read instantly without waiting for a capacitor to fill up or drain (like in DRAM).

When opposite voltages are applied to the column wires, the flip-flop is oriented in one of two directions for a 0 or 1. At that point, the flip-flop becomes a self-perpetuating storage cell as long as a constant voltage is applied.
<table>
<thead>
<tr>
<th>Memory address</th>
<th>Memory content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>Decimal</td>
</tr>
<tr>
<td>0...000</td>
<td>0</td>
</tr>
<tr>
<td>0...001</td>
<td>1</td>
</tr>
<tr>
<td>0...010</td>
<td>2</td>
</tr>
<tr>
<td>0...011</td>
<td>3</td>
</tr>
<tr>
<td>0...100</td>
<td>4</td>
</tr>
<tr>
<td>0...101</td>
<td>5</td>
</tr>
<tr>
<td>0...110</td>
<td>6</td>
</tr>
<tr>
<td>0...111</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1...110</td>
<td>$2^n-2$</td>
</tr>
<tr>
<td>1...111</td>
<td>$2^n-1$</td>
</tr>
</tbody>
</table>

(a) Memory address and content

(b) Graphic symbols

Figure 12: Random-access memory (RAM) [Gajski].
Random Access Memories

* The RAM also has a Chip-Select input and a Read/Write Select Input.

* Both SRAM and DRAM are volatile memories, i.e., their content is lost if the power is shut down.

* ROM, PROM, EPROM, EEPROM, and flash memories are nonvolatile.

* We can connect several memory chips to get one of longer words, or connect several memory chips to get one with more words.
Figure 13: RAM organization [Gajski].
Figure 14: RAM timing [Gajski].
Push Down Stack

* A push-down stack (or simply stack) is a memory component with limited access—data can be accessed through only one location (i.e., the top of the stack).

- When data is to be stored, it is pushed on the stack and stays on top of others.
- When data is to be fetched, it has to be in the top position before it can be popped out of the stack.

* A stack can be implemented by shift registers, with an up-down counter to detect full/empty stack

* It can also be implemented by a RAM—less expensive for a large stack, But need two pointers (implemented by counters)
Figure 17: Push-down stack operations [Gajski].
First in First Out Queue

* A first-in-first-out (FIFO) queue (or simply queue or FIFO) is a memory component with limited access—data can be written through only the head (front) of the queue and read (and removed) through only the tail (back) of the queue.

* A queue can be implemented by shift registers, with an up-down counter to detect full/empty queue.

* It can also be implemented by a RAM—less expensive for a large queue, but need two pointers (implemented by counters)
(a) Queue content before 45 is stored  
(b) Queue content after 45 is stored  
(c) Queue content after 23 is read
Magnitude Comparator

1011 vs 1001?

- Final answer appears on the right.
- Takes time for answer to "ripple" from left to right.

Sage3 Sage2 Sage1 Sage0

\[ a_3 \quad b_3 \quad a_2 \quad b_2 \quad 1 > 0 \quad a_1 \quad b_1 \quad a_0 \quad b_0 \]

1gt in_gt out_gt
1eq in_eq out_eq
1lt in_lt out_lt

1gt in_gt out_gt
1eq in_eq out_eq
1lt in_lt out_lt

1gt in_gt out_gt
1eq in_eq out_eq
1lt in_lt out_lt

1gt in_gt out_gt
1eq in_eq out_eq
1lt in_lt out_lt

AgtB AeqB AltB

3213: Digital Systems & Microprocessors: L#14_15
Datapath Models

- Execution graphs
- Classification according to sequencing
- Organisation of systems: Functional and Control Units
- RTL description: division into datapath and control subsystems
- Analysis and design of datapath systems
Approach

- Digital system divided into **Datapath** and **Control** subsystems.

- The state of a system consists of the contents of its registers.

- Functioning of system performed in a sequence of register transfers synchronous to the clock.

- The register transfer is a transformation performed on a datum while the datum is transferred from one register to another within the datapath.

- The sequencing of the register transfers is controlled by the **control subsystem**
Execution Graphs: Polynomial Evaluation Example

\[ P_7(x) = \sum_{i=0}^{7} p_i x^i \]

TWO ALGORITHMS:

\[ P_7(x) = (((((p_7x + p_6)x + p_5)x + p_4)x + p_3)x + p_2)x + p_1)x + p_0 \]

\[ P_7(x) = (x^2)(x^2)[x^2(p_7x + p_6) + (p_5x + p_4)] + x^2(p_3x + p_2) + (p_1x + p_0) \]
Sequential Execution Graphs

Only one node can be executed at a time

Unfolded

(a)

Loop

(b)

BEGIN

$V = p_7 \times x + p_6$

$V = V \times x + p_5$

$V = V \times x + p_4$

$V = V \times x + p_3$

END

BEGIN

$V = p_7$

$i = 6$

$V = V \times x + p_i$

$i = i - 1$

$i \geq 0$

yes

no

END
Concurrent Execution Graphs

More than one node active at a time

3213: Digital Systems & Microprocessors: L#14_15
In group sequential execution graphs each group can start when the preceding has been executed.

\[ A = x \times x \quad B = p_7 x + p_6 \quad C = p_5 x + p_4 \quad D = p_3 x + p_2 \quad E = p_1 x + p_0 \]

\[ F = A \times A \quad G = B \times A + C \quad H = D \times A + E \]

\[ P_7(x) = G \times F + H \]

(b)
Transfer Concurrent into Sequential Execution Graph

Any concurrent execution graph can be converted into an equivalent sequential graph by sequencing of the concurrent nodes.

\[
\begin{align*}
A &= x \times x \\
B &= p_7 \times x + p_6 \\
C &= p_5 \times x + p_4 \\
D &= p_3 \times x + p_2 \\
E &= p_1 \times x + p_0 \\
F &= A \times A \\
G &= B \times A + C \\
H &= D \times A + E \\
P_7(x) &= G \times F + H
\end{align*}
\]

END
Concurrent vs Sequential (and G.S.) Execution Graphs

- Sequential execution graphs are easier to develop due to ease of tracking data transformations – **we design them using an ALGORITHMIC design approach.**

- Control of sequential execution graphs is simpler because
  - A single arc usually goes from one node to its unique successor
  - A node may have several output arcs but a **conditional branch** determines which path is followed

- **Just like a computer algorithm only pipe-lined**

- Sequencing can be controlled by a FSM in which each node or group of nodes corresponds to one state of the controller - **we consider this approach in a couple of examples**

- Concurrent graphs are potentially faster - **parallel algorithms**

- **Usually need a combination of both to meet timing constraints – Group Sequential**
SEQUENTIAL DATAPATHS

- Think of these as *algorithms* but not exactly like JAVA or C !!!

- Data is stored in memories or registers

- Data fetched from memory after the rising edge of the clock

- Data processed using combinational components between two clock transitions. Data stored back in memory at the second transition

- The algorithm can be controlled by a state machine (FSM) but not always necessary.

- The only practical difference is that a FSM has no memory whereas a datapath has *memory* – usually registers, register files, RAMS, but.....

- **NOT VERILOG reg-variables!!!!!!!!!!!!!!!!!!!!!!!!!!!**

- **Recall how MU0 used registers**
### Figure 24: Datapath with 3-port register file [Gajski].

#### (a) Datapath schematic


#### (b) Table of ALU operations

<table>
<thead>
<tr>
<th>M</th>
<th>S_1</th>
<th>S_0</th>
<th>ALU Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>complement A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>AND</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>EX-OR</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>OR</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>decrement A</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>add</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>subtract</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>increment A</td>
</tr>
</tbody>
</table>

#### (c) Table of shifter operations

<table>
<thead>
<tr>
<th>S_2</th>
<th>S_1</th>
<th>S_0</th>
<th>Shift Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>pass</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>pass</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>not used</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>not used</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>shift left</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>rotate left</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>shift right</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>rotate right</td>
</tr>
</tbody>
</table>

#### (d) Control word

<table>
<thead>
<tr>
<th>IE</th>
<th>Write address</th>
<th>Read address A</th>
<th>Read address B</th>
<th>ALU operation</th>
<th>Shifter operation</th>
<th>OE</th>
</tr>
</thead>
</table>
SEQUENTIAL DATAPATH DESIGN

• In designing a datapath we assume that a control word plus inputs is provided as an input to the system c.f. MU0

• In a pure algorithmic datapath we have to control somehow → add a counter or clocked stack or FSM (RTL?)

• Maybe we should consider algorithmic datapaths for simple tasks. More complex datapaths are promoted to RTLs.

• But RTLs have a datapath anyway

• *The line is blurred....we are just learning new approaches to design*
A Timer or Signal Generator

Design a group sequential datapath to execute an algorithm to timeout a binary counter at a preset count.

A signal generator or frequency synthesiser is a slightly more complex device that outputs any number of signals at (usually but not always fixed) predetermined intervals.

Examples – Numerically controlled oscillators (NCOs), direct digital synthesisers (DDS) and Coordinate Rotating Digital Computers (CORDICS).
SEQUENTIAL DATAPATH DESIGN: EXAMPLE 1

Two registers:
- Target = the preset count for timeout
- Cntr = the current value of the count

Inputs:  
- TIC = the start enable
- Target

Outputs:  
- TOC = the stop flag

Algorithm

```plaintext
if(TIC)
  1. Cntr = 0
  2. Target = T
  3. TOC = 0
while( !(Target^Cntr) )
  4. Cntr = Cntr + 1
end while
  5. TOC=1
end if
```
SEQUENTIAL DATAPATH DESIGN EXAMPLE 1: Design Procedure

- The `control word` corresponds to the line numbers of the algorithm.

- Define some registers and proceed with a normal FSM design and VERILOG coding style.

Important Note – the algorithm begins by initialising the `registers` containing `Cntr, Target and TOC`
A One's Counter

Using a datapath with a register file, design a one's counter that will count the number of 1's in an input bitstream and return the result after completion.
SEQUENTIAL DATAPATH DESIGN: EXAMPLE 2

- Four registers:
  Data = input bitstream,
  Ocount = output = number 1's,
  Mask = 1 (1 bit) (could be a parameter in an FSM)
  Temp = temporary storage register

- Algorithm and registers required

1. Data := Inport
2. Ocount := 0
3. Mask := 1
   while Data ≠ 0 repeat
   4. Temp := Data AND Mask
   5. Ocount := Ocount + Temp
   6. Data := Data >> 1
   end while
7. Outport := Ocount

(a) Basic algorithm for one's count
(b) Register assignment

- Associate the steps of the algorithm with the control words and think of as a sequential algorithm (actually an FSM)

- Not optimised for speed!
### DATAPATH DESIGN EXAMPLE 2

<table>
<thead>
<tr>
<th>Control Words</th>
<th>IE</th>
<th>Write address</th>
<th>Read address A</th>
<th>Read address B</th>
<th>ALU operation</th>
<th>Shifter operation</th>
<th>OE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>R1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>R3</td>
<td>0</td>
<td>0</td>
<td>add</td>
<td>pass</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>R2</td>
<td>0</td>
<td>X</td>
<td>increment</td>
<td>pass</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>R4</td>
<td>R1</td>
<td>R2</td>
<td>AND</td>
<td>pass</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>R3</td>
<td>R3</td>
<td>R4</td>
<td>add</td>
<td>pass</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>R1</td>
<td>R1</td>
<td>0</td>
<td>add</td>
<td>shift right</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>none</td>
<td>R3</td>
<td>0</td>
<td>add</td>
<td>pass</td>
<td>1</td>
</tr>
</tbody>
</table>

(c) Control words for one’s counter

Figure 25: One’s-count algorithm [Gajski].
FSM EXAMPLE 2

Start = 0

S0

Start = 1

S1

Data = Inport

S2

Ocount = 0

S3

Mask = 1

S4

Temp = Data AND Mask

S5

Ocount = Ocount + Temp

S6

Data = Data >> 1 (shift right)

S7

Outport = Ocount

Done = 1
Observations

• FSM provides a good solution to a sequential DATAPATH.

• But note the need for storage devices to imbue the FSM with memory

• Often an FSM is overkill – most of the steps in the algorithm proceed automatically without a control flag input. Other possibilities -
  ➢ a simple counter if the algorithm proceeds without too many jumps. Also a better solution if the algorithm has thousands of automatic (unconditioned / no control flag) steps
  ➢ ROM or stack and state register implementations

• Concurrent and group sequential datapath designs may be required for high speed algorithms to meet timing constraints
The End
Pulse generator: Intelligent Switch debouncer

- When $Sw \rightarrow 0$, a counter is started, $Sd = 0$
- When the counter overflows, $cntd = 1$, $Sd = 1$
- Problem: $Sd = 1$ for many clock cycles
- Could just set $Sd = 1$ in state $S1$

Figure 1: Even odd detector FSM
Pulse generator: Switch debouncer

\[
\begin{align*}
X &= 0 \\
S_0 &\quad Y = 0 \\
X &= 1, \text{ Cntr start} \\
S_1 &\quad Y = 0 \\
&\quad \text{Cntr} = 0 \\
S_2 &\quad Y = 1
\end{align*}
\]
SEQUENTIAL AND RTL SYSTEMS...
Scope (follow Ercegovac)

- Sequential and finite state systems
- State vs time behaviour, Controllers

Register transfer level systems

Design and analysis of RTLs

Examples
Sequential Systems

● Definition

● The state description and time behaviour

● Controllers as sequential systems
Sequential Systems Definition

- **State description:** two functions on finite sets
- **Time description:** function on time sequences.
- Output depends on both current and previous inputs.
- \( z(t) = F(x(0,t)) \), where \( x(t) \) belongs to input set and \( z(t) \) to output set: both finite.
Sequential Systems Time Description

$z(t) = F(x(0, t))$

Note multi-level less compact than labelled lines

3213: Digital Systems & Microprocessors: L#14_15
Synchronous and Asynchronous Systems

According to the time at which inputs and outputs are considered

- **Synchronous**: Inputs/outputs sampled at active edges of a clock. \( x(t_1,t_2) \)

- **Asynchronous**: time variable is continuous
Synchronous and Asynchronous Systems

Figure 7.2: a) SYNCHRONOUS BEHAVIOR. b) ASYNCHRONOUS BEHAVIOR.

- CLOCK
- I/O SEQUENCE $x(t_1, t_2)$

$x(2, 5) = aabc$
Decimal Serial Adder

\[
x | 1638753 \\
y | 3652425 \\
z | 5291178
\]

- **LEAST-SIGNIFICANT DIGIT FIRST (at t=0)**

<table>
<thead>
<tr>
<th>t</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>x(t)</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>y(t)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>z(t)</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>
State Description (of FSMs!)

- State machine must capture influence of all past inputs $x(t_1, t_2)$.
- Do we need to MEMORISE all $x(0,t)$ in order to determine $z$?
- Group values of input sequences into finite number of **classes** such that all time functions with same output are in the same class.
- Thus to determine $z(t)$ we do not need $x(0,t)$. **Only need the class to which the function belongs.**
- Class is kept in an auxiliary variable: the state
Time Description

- Just remember the current inputs and the carry-in.

\[ s(t + 1) = G(s(t), x(t)) \]

\[ z(t) = H(s(t), x(t)) \]

Figure 7.3: OUTPUT AND STATE TRANSITION FUNCTIONS

FSM is synchronous
State Description

- The carry-in $c(t)$ is the state and it has two values.

- The state description has
  - three time variables: input, state and output
  
  \[ \text{Next State} = G(\text{Present input, Present state}) \]

- and,
  - two functions: The state transition function and the output function
  
  \[ \text{Output} = H(\text{Present input, Present state}) \]
Use of Conditional Expressions

Input: \( x(t) \in \{0, 1, 2, 3\} \)
Output: \( z(t) \in \{a, b\} \)
State: \( s(t) \in \{S_0, S_1\} \)
Initial state: \( s(0) = S_0 \)

Functions: The transition and output functions are

\[
s(t + 1) = \begin{cases} 
S_0 & \text{if } (s(t) = S_0 \text{ and } x(t) = 0 \text{ or } x(t) = 2) \\
S_1 & \text{if } (s(t) = S_1 \text{ and } x(t) = 3) \\
S_1 & \text{otherwise}
\end{cases}
\]

\[
z(t) = \begin{cases} 
a & \text{if } s(t) = S_0 \\
b & \text{if } s(t) = S_1
\end{cases}
\]
State Description of Serial Adder

Input: \( x(t), y(t) \in \{0, 1, \ldots, 9\} \)
Output: \( z(t) \in \{0, 1, \ldots, 9\} \)
State: \( s(t) \in \{0, 1\} \) (the carry)
Initial state: \( s(0) = 0 \)

Functions: The transition and output functions are

\[
s(t + 1) = \begin{cases} 
1 & \text{if } x(t) + y(t) + s(t) \geq 10 \\
0 & \text{otherwise}
\end{cases}
\]

\[
z(t) = (x(t) + y(t) + s(t)) \mod 10
\]

EXAMPLE:

<table>
<thead>
<tr>
<th>( t )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x(t) )</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>( y(t) )</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>( s(t) )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( z(t) )</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>
Example of Time Behaviour: (Odds/Evens)

TIME-BEHAVIOR SPECIFICATION:

Input: \( x(t) \in \{a, b\} \)

Output: \( z(t) \in \{0, 1\} \)

Function: \( z(t) = \begin{cases} 1 & \text{if } x(0, t) \text{ contains an even number of } b's \\ 0 & \text{otherwise} \end{cases} \)

I/O SEQUENCE:

<table>
<thead>
<tr>
<th>( t )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x, z )</td>
<td>a, 1</td>
<td>b</td>
<td>0</td>
<td>b, 1</td>
<td>a, 1</td>
<td>b, 0</td>
<td>a</td>
<td>0</td>
</tr>
</tbody>
</table>
Example of State Description: (Odds/Evens)

Input: \( x(t) \in \{ a, b \} \)
Output: \( z(t) \in \{ 0, 1 \} \)
State: \( s(t) \in \{ \text{EVEN, ODD} \} \)
Initial state: \( s(0) = \text{EVEN} \)

Functions: Transition and output functions

<table>
<thead>
<tr>
<th>( PS )</th>
<th>( x(t) = a )</th>
<th>( x(t) = b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEN</td>
<td>EVEN, 1</td>
<td>ODD, 0</td>
</tr>
<tr>
<td>ODD</td>
<td>ODD, 0</td>
<td>EVEN, 1</td>
</tr>
<tr>
<td>NS, ( z(t) )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Time Behaviour from FSM

STATE DESCRIPTION $\Rightarrow$ I/O SEQUENCE (Example 7.10)

Initial state: $s(0) = S_2$

Functions: Transition and output functions are:

<table>
<thead>
<tr>
<th>PS</th>
<th>$x(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a$</td>
</tr>
<tr>
<td>$S_0$</td>
<td>$S_0$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>$S_2$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$S_2$</td>
</tr>
<tr>
<td>$S_3$</td>
<td>$S_0$</td>
</tr>
<tr>
<td></td>
<td>$NS$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$t$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>$s$</td>
<td>$S_2$</td>
<td>$S_2$</td>
<td>$S_3$</td>
<td>$S_2$</td>
<td>$S_2$</td>
</tr>
<tr>
<td>$z$</td>
<td>q</td>
<td>q</td>
<td>p</td>
<td>q</td>
<td></td>
</tr>
</tbody>
</table>
Obtain FSM from Time Behaviour

- One can always determine the time behaviour given an FSM
- Can one always deduce an FSM from time behaviour? - NO!
NOT ALL TIME-BEHAVIORS ARE REALIZABLE:

\[ z(t) = \begin{cases} 1 & \text{if } x(0, t) \text{ has same number of 0's and 1's} \\ 0 & \text{otherwise} \end{cases} \]

\[ s(t) = \text{DIFFERENCE BETWEEN NUMBER OF 1'S AND 0'S} \]

\[ s(t + 1) = \begin{cases} s(t) + 1 & \text{if } x(t) = 1 \\ s(t) - 1 & \text{otherwise} \end{cases} \]

\[ z(t) = \begin{cases} 1 & \text{if } s(t) = 0 \\ 0 & \text{otherwise} \end{cases} \]

⇒ DIFFERENCE UNBOUNDED: NOT A FINITE-STATE SYSTEM
Obtain FSM from Time Behaviour

If the time behaviour can be described by an FSM then the following procedure applies:

- Determine a set of states representing the classes of events – quite system dependent
- Determine the transition function.
- Determine the output function.
Example: Pattern recogniser

- Current output indicates whether the input subsequence ending at time $t$ corresponds to a pattern,
- A sequential system that recognises the pattern $P=\{p_0, p_1, \ldots, p_{m-1}\}$ has the following time description,
- Input: $x(t)$ in $\{0,1\}$
- Output: $z(t)$ in $\{0,1\}$
- Function: 
  - $z(t) = 1$, if $x(t-m+1, t) = P$
  - $z(t) = 0$ otherwise
Example 7.11

Input: \( x(t) \in \{0, 1\} \)

Output: \( z(t) \in \{0, 1\} \)

Function: \[
z(t) = \begin{cases} 
1 & \text{if } x(t - 3, t) = 1101 \\
0 & \text{otherwise}
\end{cases}
\]

**PATTERN DETECTOR ⇒ DETECT SUBPATTERNS**

<table>
<thead>
<tr>
<th>State</th>
<th>Indicates That</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_{\text{init}} )</td>
<td>Initial state; also no subpattern</td>
</tr>
<tr>
<td>( S_1 )</td>
<td>First symbol (1) of pattern has been detected</td>
</tr>
<tr>
<td>( S_{11} )</td>
<td>Subpattern 11 has been detected</td>
</tr>
<tr>
<td>( S_{110} )</td>
<td>Subpattern 110 has been detected</td>
</tr>
</tbody>
</table>

![State Transition Diagram]
Controllers
Controllers

• The state description is primary

• FSM producing **control signals**

• Control signals determine actions performed in other parts of the system

• **Autonomous Controller**: fixed sequence of states independent of inputs
AUTONOMOUS Controller

\[ S_0 \rightarrow \text{"Wait" state} \]

\[ S_1/C1 \rightarrow \text{C1 -- control signal to initialize the system} \]

\[ S_2/C2 \rightarrow \text{C2 -- control signal to input } x \text{ and } y \text{ to the system} \]

\[ S_3 \rightarrow \text{Compute } z = f(x, y) \]

\[ S_4/C3 \rightarrow \text{C3 -- control signal to output } z \text{ from the system} \]
NON-AUTONOMOUS Controller

The rest of digital system

Sequential Machine Controller

coin
return
sum<75
sum=75
sum>75
change_available

release_candy
return_change
return_all_coins

Quarters, dimes, or nickels only
Return
Change available
Candy bar release

Note: \( \text{coin} \cdot \text{return} = 0 \)
## MU0 Controller (autonomous)

### FSM State Transition Table

<table>
<thead>
<tr>
<th>state</th>
<th>F[2:0]</th>
<th>Next state</th>
<th>IREn</th>
<th>PCEn</th>
<th>AccEn</th>
<th>M[1:0]</th>
<th>Xsel</th>
<th>Ysel</th>
<th>Asel</th>
<th>Ren</th>
<th>Wen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>xxx</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>xx</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>110</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>111</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>xx</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Notes:
- N and Z are the Negative and Zero state of the Accumulator, respectively.
- (used to reduce the size of the table, as drawn)
- If a value is not going to be latched it doesn’t matter what it is! (e.g. ALU output for STO)
- STP operates by remaining in its evaluation state.
MU0 Controller (autonomous)
RTL SYSTEMS
Introduction

- Execution graphs
- Classification according to sequencing
- Organisation of systems: Functional and Control Units
- RTL description: division into data and control subsystems
- Analysis and design of RTL systems
**RTL approach**

- Digital system divided into **Data** and **Control** subsystems.
- The state of a system consists of the contents of its registers, usually vast if we include the datapath.
- Functioning of system performed in a sequence of register transfers synchronous to the clock.
- The register transfer is a transformation performed on a datum while the datum is transferred from one register to another.
- The sequencing of the register transfers is controlled by the **control subsystem**
Execution Graphs: Polynomial Evaluation Example

\[ P_7(x) = \sum_{i=0}^{7} p_i x^i \]

TWO ALGORITHMS:

\[ P_7(x) = (((((p_7x + p_6)x + p_5)x + p_4)x + p_3)x + p_2)x + p_1)x + p_0 \]

\[ P_7(x) = (x^2)(x^2)[x^2(p_7x + p_6) + (p_5x + p_4)] + x^2(p_3x + p_2) + (p_1x + p_0) \]
Sequential Execution Graphs

Only one node can be executed at a time

Unfolded

\[ V = p_7 \times x + p_6 \]
\[ V = V \times x + p_5 \]
\[ V = V \times x + p_4 \]
\[ V = V \times x + p_3 \]

Loop

\[ V = p_7 \]
\[ i = 6 \]
\[ V = V \times x + p_i \]
\[ i = i - 1 \]
\[ i \geq 0 \]

(a)
Concurrent Execution Graphs

More than one node active at a time

3213: Digital Systems & Microprocessors: L#14_15
In **group sequential** execution graphs each group can start when the preceding has been executed.

\[
\begin{align*}
A &= x \times x \\
B &= p_7 x + p_6 \\
C &= p_5 x + p_4 \\
D &= p_3 x + p_2 \\
E &= p_1 x + p_0 \\
F &= A \times A \\
G &= B \times A + C \\
H &= D \times A + E \\
P_7(x) &= G \times F + H
\end{align*}
\]
Transfer Concurrent into Sequential Execution Graph

Any concurrent execution graph can be converted into an equivalent sequential graph by sequencing of the concurrent nodes.

BEGIN

\[ A = x \times x \rightarrow B = p_7 x + p_6 \rightarrow C = p_5 x + p_4 \rightarrow D = p_3 x + p_2 \rightarrow E = p_1 x + p_0 \]

\[ F = A \times A \rightarrow G = B \times A + C \rightarrow H = D \times A + E \]

\[ P_7(x) = G \times F + H \]

END
Concurrent vs Sequential (and G.S.) Execution Graphs

- Sequential execution graphs are easier to develop due to ease of tracking data transformations.
- Control of sequential execution graphs is simpler because:
  - A single arc usually goes from one node to its unique successor.
  - A node may have several output arcs but a conditional branch determines which path is followed.
- Sequencing can be controlled by a FSM in which each node or group of nodes corresponds to one state of the controller.
- Concurrent graphs are potentially faster.
RTL Design Example:

A Morse Code Translator
Morse Code

Transmits telegraphic information using rhythm.

Morse code uses a standardized sequence of short and long elements (dots and dashes) to represent information.

Originally invented by Samuel F. B. Morse's (April 27, 1791 – April 2, 1872) for his electric telegraph in the early 1840s, Morse code was also extensively used for early radio communication beginning in the 1890s.

Variable length of the Morse characters made it hard to adapt to automated circuits.

Morse code mostly used now by amateur radio operators, although it is no longer a requirement for licensing in many countries.
Morse Code

Morse code is transmitted using just two states (on and off) so it was an early form of a digital code.

Visualize any Morse code sequence as a combination of the following five elements (quinary):

1. short mark, dot or 'dit' (·) — 1
2. longer mark, dash or 'dah' (–) — 111
3. intra-character gap (between the dots and dashes within a character) — 0
4. short gap (between letters) — 000
5. medium gap (between words) — 00000000
International Morse Code

1. A dash is equal to three dots.
2. The space between parts of the same letter is equal to one dot.
3. The space between two letters is equal to three dots.
4. The space between two words is equal to seven dots.

M O R S E  C O D E

A B C D E F G H I J K L M N O P Q R S T

U V W X Y Z

1 2 3 4 5 6 7 8 9 0
Pegasus Implementation?

Use three push buttons
Design Concept

Morse code not amenable to digital implementation due to its syncopated asynchronous human element -> Make it amenable by assigning PEGASUS push buttons to the dots, dashes and spaces.

-> Breaks rules by making Morse code a ternary (actually quaternary) as opposed to a quinary code.- dot, dash, space is all we need!

Not so useless after all – could use for texting?

PB0  PB1  PB2
DOT  DASH  SPACE
Design Concept

- Let PEGASUS push buttons (PBs) do *dot, dash, space*. **We do not need inter-dit-dah space.**

- Need a separate interface (*ButtonIface.v*) to convert PB inputs to some for RTL system.
  - Issues – debounce the Pbs a must, need to sync outputs
  - DOT, DASH, SPACE, NULL work a bit like KEY/NOKEY in RP document.

- Will not treat the display driver (leave this to HLAB3)
Data Path Design

Data path has two modules

Register *KeyHoldReg.v* registers DOTS and DASHES

Register *Morse2Char.v* outputs an alphanumeric character for each Morse code sequence

A Look Up Table (LUT) to store characters given DOTS and DASHES – incorporate this into *Morse2Char.v* (cut corner)

Always control these by binary controls from FSM
ButtonIface.v

clk

PB0 (dot)

PB1 (dash)

PB2 (space)

beeps

| NULL | DASH | NULL | SPACE |

3213: Digital Systems & Microprocessors: L#14_15
always @ (posedge clk) begin

if (rst) begin
    morse <= 16'hffff;
end else if (inc) begin
    morse <= {morse[13:2], beeps[1], beeps[0], 1'b0, 1'b0};
end else morse <= morse;

end
Morse Functional Description

Inputs:

Beeps = \{\text{DOT, DASH, SPACE, NULL}\}

Outputs:

Rst, Inc, Chen
Controller State Diagram

- Sinit
  - Space/Null: $\text{rst}=1$, $\text{inc}=0$, $\text{chen}=0$
  - dot/dash: $\text{rst}=0$, $\text{inc}=1$, $\text{chen}=0$

- S0
  - dot/dash: $\text{rst}=0$, $\text{inc}=1$, $\text{chen}=0$
  - Null: $\text{rst}=0$, $\text{inc}=0$, $\text{chen}=0$

- S1
  - Space: $\text{rst}=0$, $\text{inc}=0$, $\text{chen}=1$

3213: Digital Systems & Microprocessors: L#14_15
## Controller State Transition Table

<table>
<thead>
<tr>
<th>State</th>
<th>Inputs</th>
<th>Snext</th>
<th>rst</th>
<th>inc</th>
<th>chen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sinit</td>
<td>dot/dash</td>
<td>S0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Sinit</td>
<td>Space/NULL</td>
<td>Sinit</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S0</td>
<td>dot/dash</td>
<td>S0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>S0</td>
<td>NULL</td>
<td>S0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S0</td>
<td>Space</td>
<td>S1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>X</td>
<td>Sinit</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Output: print S.O.S