Digital Electronics

Seyran Balasanyan
Mane Aghagulyan
Heinz-Dietrich Wuttke
Karsten Henke

Bachelor Embedded Systems
Year
Group

DesIRE

Table of Contents

List of illustrations

List of used abbreviations and symbols

CPLD	Complex Programmable Logic Device
CLB	Configurable logic blocks
$\overline{CS}$	Chip select
FF	Flip-Flop
FPGA	Field Programmable Gate Array
GAL	Gate Array Logic
LUT	Look Up Table
PLA	Programmable Logic Array
ROM	Read Only Memory

Introduction

The course “Digital Electronics” describes the fundamental basics of digital hardware, starting from Boolean algebra and their representation in digital electronic circuits. Step by step more complex circuits are introduced from a functional and thereafter from a structural point of view.

Thus a systematic overview about digital electronics is given that ranges from basic systems on gate-level, combinational circuits like decoders, multiplexers and de-multiplexers to sequential circuits like latches, flip-flops, registers and counters and finally to programmable circuits like ROM, PLA, GAL, CPLD and FPGA.

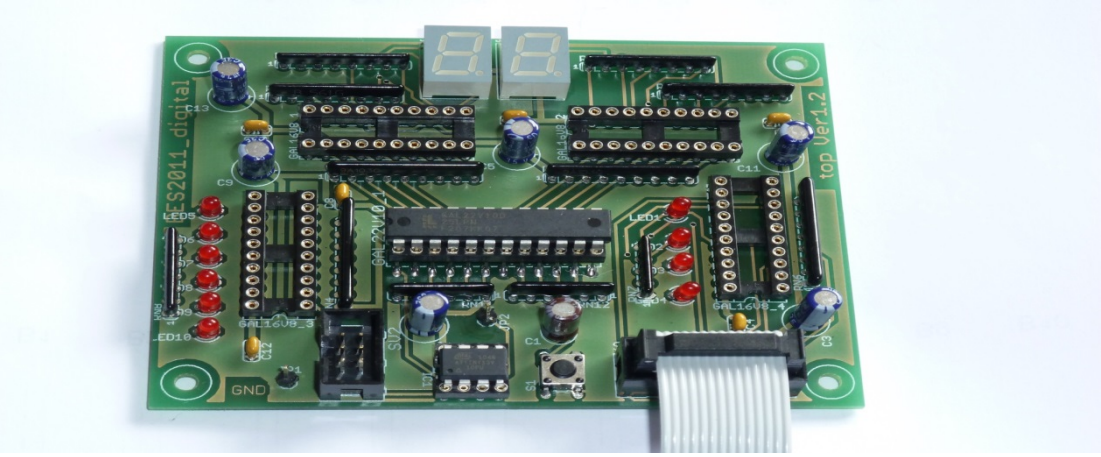


Figure 0-1: Board with a CPLD

The course is a prerequisite for the course “Digital System Design” as well as for the hands-on laboratory work in the “GOLDi” remote labs of the IUT.

The course is delivered in a blended learning style, containing online lectures, homework tasks and presence phases for consultation. It ends with an oral examination of about 20 minutes.

After finishing the course, students are able to identify combinational and sequential digital circuits. They can describe their structure as well as their function in a formalized manner. The students can analyze digital circuit diagrams and transfer them into a formal description based on Boolean algebra.

Learning outcomes

Analyze

- Students should be able to analyze digital circuit diagrams and transfer them into a formal description based on Boolean algebra.

Realize

- Students are able to realize simple combinational and sequential circuits based on standard structures.
- They should be able to design a proper schematic, develop their own library components, route the printed circuit board, in 2D and 3D, use design rules, according to the supplier of their choice and deliver all necessary output document for actual ordering and production of the board and components used.

Optimize

- Students should make use of the tools to check their design and improve errors and warnings
- Students should be able to use the tools to check for crosstalk and high-frequent reflections and adjust their design accordingly.

Communicate

- They should communicate on the theory behind different aspects of digital electronics.
- They should make and defend their choices in design, according to the description methods used and components used.
- They should also be able to communicate on the subject and answer questions on a theoretical basis.

Perform professionally

- The student should be able to find suitable information in the course material and through web resources and use the information found accordingly.
- Schematics, PCB routing and library development should be done in a clear and orderly fashion.
- The students should be able to work together and divide the workload independently.

Think business-like

- Students should realize the economic repercussions of their design choices, whether this is related to their cost of components, the cost of the PCB production or the cost of automated assembly of the bare board.

Assessments

The theory will be assessed in an oral exam.
There are questions from the fundamental part of the course and from the design of digital electronics.

The digital electronics project and lab work is assessed during the presentation of the design work. The work is evaluated on the schematics, the library content and usage, the PLD description and the output generation.

1 Introduction to Programmable Logic

1.1 History

A programmable logic device (PLD) is an electronic component used to build reconfigurable digital circuits. Unlike a logic gate, which has a fixed function, a PLD has an undefined function at the time of manufacture. Before the PLD can be used in a circuit it must be programmed, that is, reconfigured.

Logic devices can generally be classified into two broad categories - fixed and programmable. The circuits in a fixed logic device are permanent and they will perform one function or set of functions - once manufactured, they cannot be changed. On the other hand, PLDs are standard, off the shelf parts that offer customers a wide range of logic capacity, features, speed, and voltage characteristics, and these devices can be changed at any time after manufacture to perform any number of functions.

In order to give design flexibility, Signetics (which was later purchased by Philips Semiconductor. and then eventually by Xilinx, Inc.), developed the idea of two programmable planes. These two programmable planes provided any combination of 'AND' and 'OR' gates and sharing of AND terms across multiple OR's. This Programmable Logic Array (PLA) architecture was very flexible but slow.

Other comparable architectures followed, such as the Programmable Array of Logic (PAL) and Generic Array of Logic (GAL). These categories of device are often referred to as the Simple PLD (SPLD). The PAL architecture is similar to the PLA except that the OR gate array is fixed.

Also, PROMs (programmable ROMs), EPROMs (ultraviolet-erasable PROMs) and EEPROMs (electrically erasable PROMs) are available that can be programmed using a standard PROM programmer without requiring specialised hardware or software.

In 1971, General Electric Company (GE) was developing a programmable logic device based on the new PROM technology. This experimental device improved on IBM's ROAM by allowing multilevel logic. Intel had just introduced the floating-gate UV erasable PROM so the researcher at GE incorporated that technology. The GE device was the first erasable PLD ever developed, predating the Altera EPLD by over a decade. GE obtained several early patents on programmable logic devices. In 1973 National Semiconductor introduced a mask-programmable PLA device with 14 inputs and 8 outputs with no memory registers. This was more popular than the TI part but cost of making the metal mask limited its use. The device is significant because it was the basis for the field programmable logic array produced by Signetics in 1975.

In order to give design flexibility, Signetics (which was later purchased by Philips Semiconductor. and then eventually by Xilinx, Inc.), developed the idea of two

programmable planes. These two programmable planes provided any combination of 'AND' and 'OR' gates and sharing of AND terms across multiple OR's. This Programmable Logic Array (PLA) architecture was very flexible but slow.

Other comparable architectures followed, such as the Programmable Array of Logic (PAL) and Generic Array of Logic (GAL). These categories of device are often referred to as the Simple PLD (SPLD). The PAL architecture is similar to the PLA except that the OR gate array is fixed. Figure 1-1 shows the basic arrangement for the PAL.

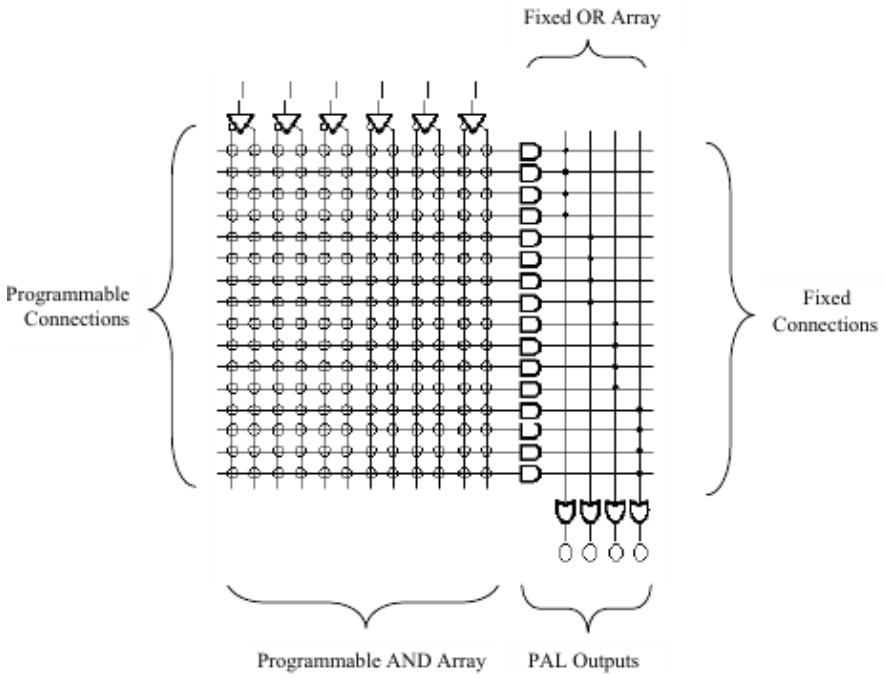


Figure 1-1: PAL Architecture

In the early 1980's, this trend continued, programmable devices consisted of PALs and a more Complex PLD known as the (CPLD), which were essentially small sets of AND-OR planes with cross-point switches to connect them plus registers in order to create sequential logic circuits such as state machines. These devices contained the equivalent of hundreds of gates of logic and were used primarily to replace digital glue logic. Towards the end of the 1980's, Xilinx Inc. and Altera Corporation, started to invest heavily in PLD technologies.

Xilinx Inc. developed the SRAM-based Field Programmable Gate Array (FPGA) that could hold from 1,000 to more than 5,000 logic gates. Altera following its success with their Complex Programmable Logic Device (CPLD) followed with an SRAM-based FPGA.

For FPGAs, SRAM is the dominant technology, although antifuse is used for applications where the protection of intellectual property is required. Antifuse also has some power consumption advantages over SRAM. Actel has introduced flash memory-based FPGAs that have the speed, size, and nonvolatility advantages of antifuse technology while using a more standard process that's easier to

manufacture. The latest trend in FPGAs is the inclusion of specialised hardware macros in the form of hard cores.

Field Programmable Analog Arrays (FPAAs) in recent years have become more popular in certain electronic system designs but not to the extent of their digital counterparts. In an FPAA, the designer has the ability to design analogue functions such as gain stages and filters, without having to work at the level of components such as op amps, capacitors and resistors. FPAAs are finding use in the areas of audio, sensors, PID controllers and communications.

1.2 Overview

The current range of available programmable device types includes devices ranging from small devices capable of implementing only a handful of logic equations to large FPGAs and CPLDs (see Figure 1-2) that can hold an entire processor core, including peripherals.

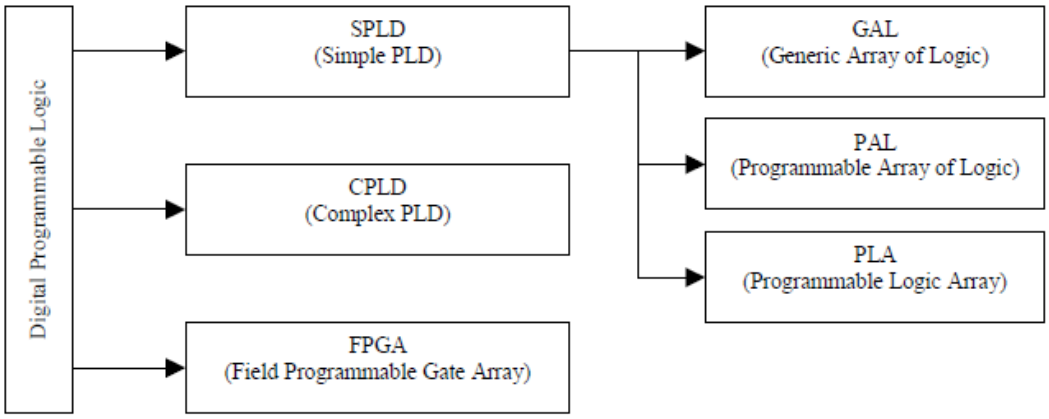


Figure 1-2: Programmable Device Types

1.3 Simple Programmable Logic Devices (SPLDs)

At the lower end of complexity are the original SPLDs. These were the first ICs that could be used to implement a flexible digital logic design in hardware. This effectively allowed for a few 74-series TTL parts (ANDs, ORs, and NOTs) to be replaced with a single SPLD.

Other names encountered for this class of device are:

- Programmable Logic Array (PLA)
- Programmable Array Logic (PAL)
- Generic Array Logic (GAL).

A single IC requires less board area, and power than multiple ICs. The design inside the chip is flexible, so a change in the logic doesn't require any rewiring of the board.

As these ICs contain only a small amount of logic, they don't have much relevance in today's semiconductor market.

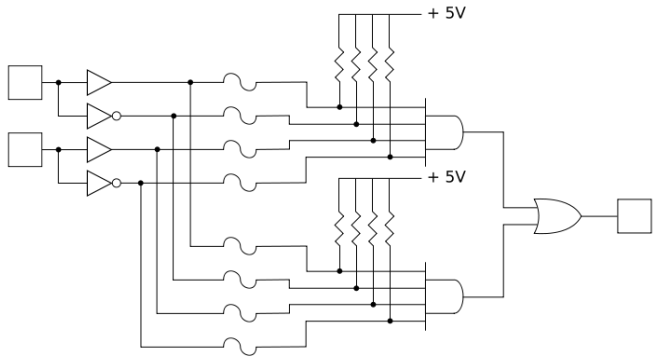


Figure 1-3: Programmable Device Types

1.4 Programmable Logic Array (PLA)

PLA is a kind of programmable logic device used to implement combinational logic circuits. The PLA has a set of programmable AND gate planes, which link to a set of programmable OR gate planes, which can then be conditionally complemented to produce an output. This layout allows for a large number of logic functions to be synthesized in the sum of products (and sometimes product of sums) canonical forms.

PLA's differ from Programmable Array Logic devices (PALs and GALs) in that both the AND and OR gate planes are programmable.

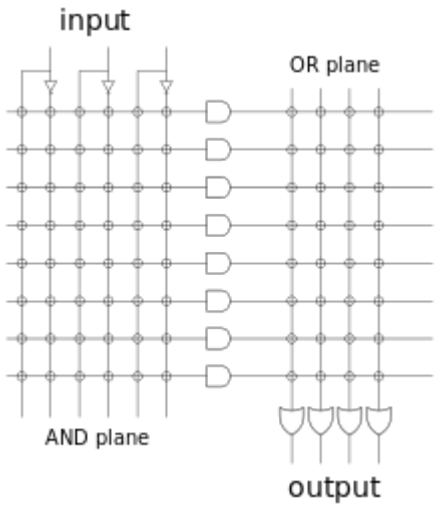


Figure 1-4: PLA schematic example

1.5 Programmable Array of Logic (PAL)

PAL devices have arrays of transistor cells arranged in a "fixed-OR, programmable-AND" plane used to implement "sum-of-products" binary logic equations for each of the outputs in terms of the inputs and either synchronous or asynchronous feedback from the outputs.

MMI introduced a breakthrough device in 1978, the Programmable Array Logic or PAL. The architecture was simpler than that of Signetics FPLA because it omitted the programmable OR array. This made the parts faster, smaller and cheaper. They were available in 20 pin 300 mil DIP packages while the FPLAs came in 28 pin 600 mil packages. The PAL Handbook demystified the design process. The PALASM design software (PAL Assembler) converted the engineers' Boolean equations into the fuse pattern required to program the part. The PAL devices were soon second-sourced by National Semiconductor, Texas Instruments and AMD.

After MMI succeeded with the 20-pin PAL parts, AMD introduced the 24-pin 22V10 PAL with additional features. After buying out MMI (1987), AMD spun off a consolidated operation as Vantis, and that business was acquired by Lattice Semiconductor in 1999.

Figure 1-5: MMI PAL 16R6 in 20-pin DIP

1.6 Generic Array of Logic (GAL)

An innovation of the PAL was the generic array logic device, or GAL, invented by Lattice Semiconductor in 1985. This device has the same logical properties as the PAL but can be erased and reprogrammed. The GAL is very useful in the prototyping stage of a design, when any bugs in the logic can be corrected by reprogramming. GALs are programmed and reprogrammed using a PAL programmer, or by using the in-circuit programming technique on supporting chips.

Lattice GALs combine CMOS and electrically erasable (E2) floating gate technology for a high-speed, low-power logic device.

A similar device called a PEEL (programmable electrically erasable logic) was introduced by the International CMOS Technology (ICT) Corporation.

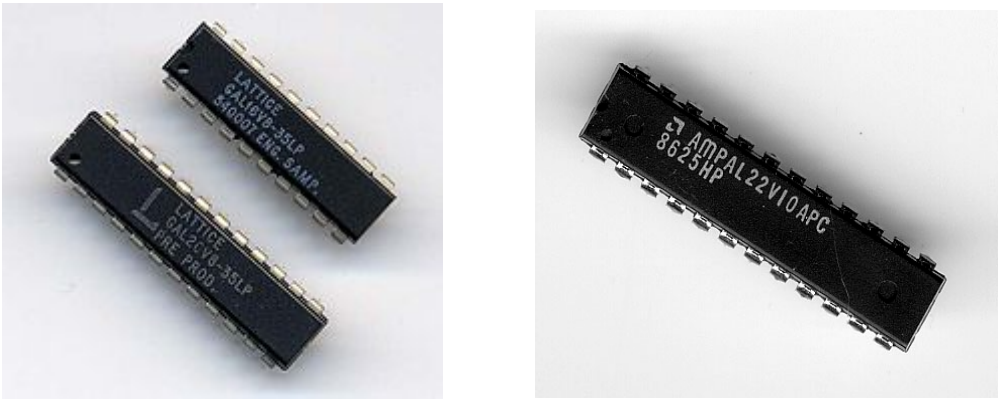


Figure 1-6: Lattice GAL 16V8 and 20V8 and AMD 22V10 in 24-pin DIP

1.7 Complex Programmable Logic Devices (CPLDs)

PALs and GALs are available only in small sizes, equivalent to a few hundred logic gates.

As the ability to integrate more circuitry within an IC, the PLD manufacturers evolved their products into larger (logically, but not necessarily physically) parts called CPLDs. For most practical purposes, CPLDs can be thought of as multiple PLDs (plus some programmable interconnect) in a single chip.

The larger size of a CPLD allows you to implement either more logic equations or a more complicated design. CPLDs can replace thousands, or even hundreds of thousands, of logic gates.

Some of the CPLD features are in common with PALs:

- Non-volatile configuration memory. Unlike many FPGAs, an external configuration ROM isn't required, and the CPLD can function immediately on system start-up.
- For many legacy CPLD devices, routing constrains most logic blocks to have input and output signals connected to external pins, reducing opportunities for internal state storage and deeply layered logic. This is usually not a factor for larger CPLDs and newer CPLD product families.

Other features are in common with FPGAs:

- Large number of gates available. CPLDs typically have the equivalent of thousands to tens of thousands of logic gates, allowing implementation of moderately complicated data processing devices. PALs typically have a few hundred gate equivalents at most, while FPGAs typically range from tens of thousands to several million.
- Some provisions for logic more flexible than sum-of-product expressions, including complicated feedback paths between macro cells, and specialized

logic for implementing various commonly used functions, such as integer arithmetic.

The most noticeable difference between a large CPLD and a small FPGA is the presence of on-chip non-volatile memory in the CPLD. The characteristic of non-volatility makes the CPLD devices used in modern digital designs for performing "boot loader" functions before handing over control to other devices not having this capability. A good example is where a CPLD is used to load configuration data for an FPGA from non-volatile memory.

Figure 1-7 contains a block diagram of a simple hypothetical CPLD.

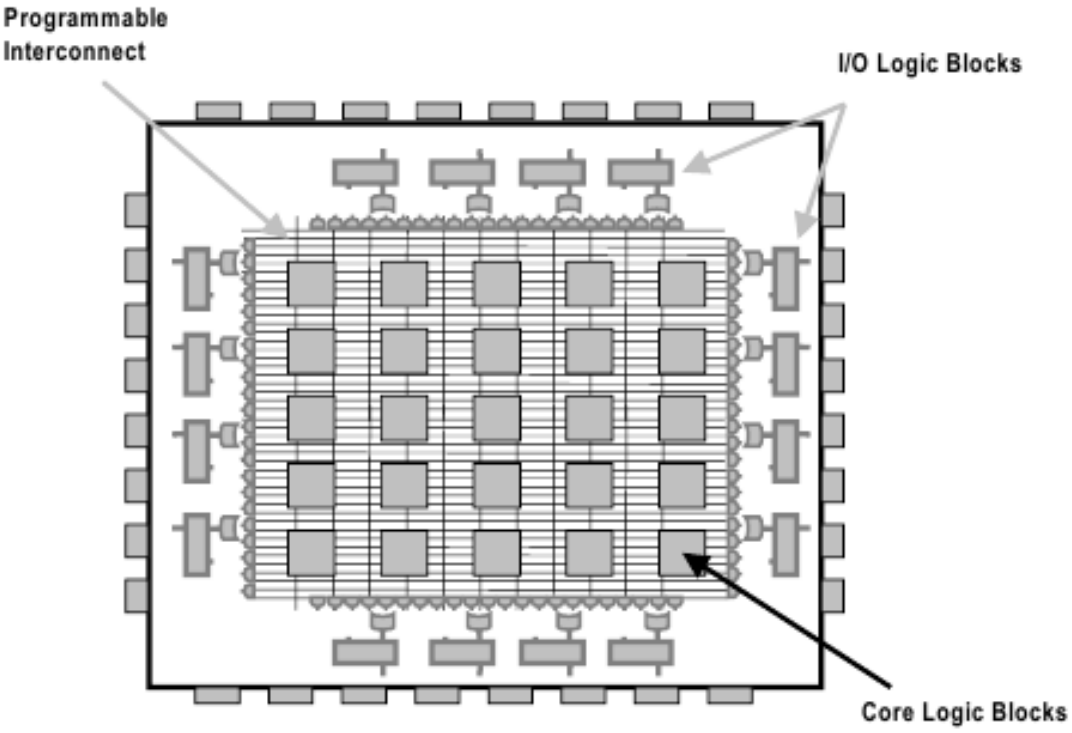


Figure 1-7: Internal Structure of a CPLD

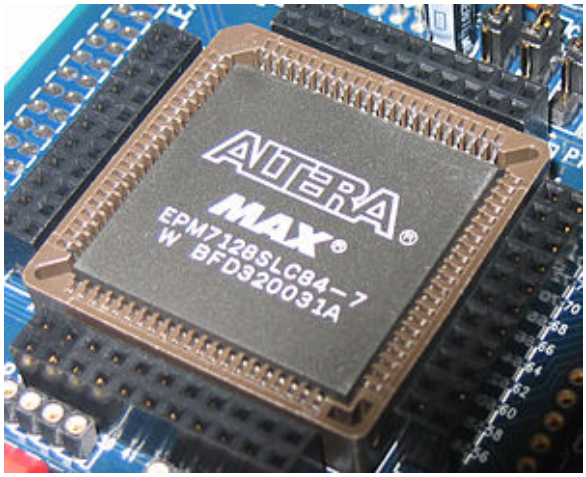


Figure 1-8: Altera MAX 7000-series CPLD with 2500 gates

Each of the logic blocks shown is the equivalent of one PLD, these logic blocks are themselves comprised of macrocells and interconnect wiring, just like an ordinary PLD. Unlike the programmable interconnect within a PLD, the switch matrix within a CPLD may or may not be fully connected. The high-end (in terms of numbers of gates), there is also a lot of overlap in potential applications with FPGAs. Traditionally, CPLDs have been chosen over FPGAs whenever high-performance logic is required. Because of its less flexible internal architecture, the delay through a CPLD (measured in nanoseconds) is more predictable and usually shorter.

Some CPLDs are programmed using a PAL programmer, but this method becomes inconvenient for devices with hundreds of pins. A second method of programming is to solder the device to its printed circuit board, then feed it with a serial data stream from a personal computer. The CPLD contains a circuit that decodes the data stream and configures the CPLD to perform its specified logic function. Some manufacturers (including Altera and Microsemi) use JTAG to program CPLD's in-circuit from JAM files.

1.8 Field Programmable Gate Arrays (FPGAs)

A FPGA (see Figure 1-9) is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC) (circuit diagrams were previously used to specify the configuration, as they were for ASICs, but this is increasingly rare).

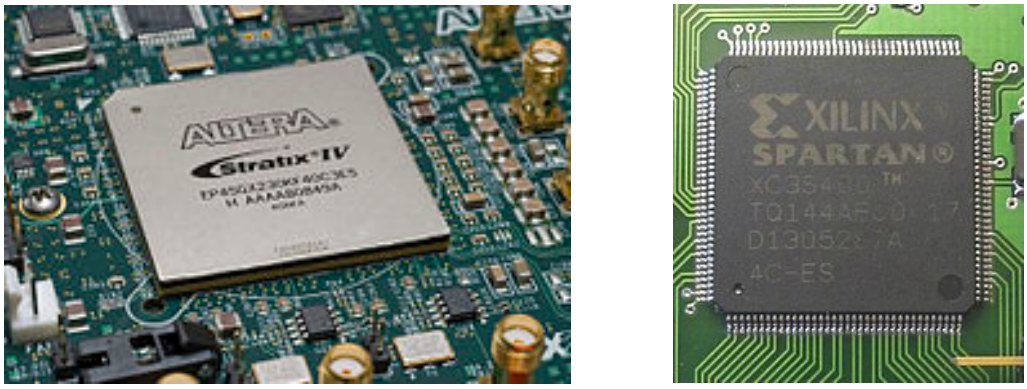


Figure 1-9: FPGA from Altera and Xilinx

FPGAs can be used to implement a wide range of digital logic circuits and systems. One common use is to prototype hardware that will eventually find its way into an ASIC. However, there is no reason why the FPGA cannot be used in the final product. Finally it depends on the relative weights between development cost and production cost for a particular project. It costs significantly more to develop an ASIC, but the cost per chip may be lower in the long run. The development of the FPGA was distinct from the PLD/CPLD evolution just described. This is apparent

when you look at the structures inside. Figure 1-10 and Figure 1-11 illustrate a typical FPGA architecture.

There are three key parts of its structure: logic blocks, interconnect, and I/O blocks. The I/O blocks form a ring around the outer edge of the part. Each of these provides individually selectable input, output, or bi-directional access to one of the general-purpose I/O pins on the exterior of the FPGA package. Inside the ring of I/O blocks lies a rectangular array of programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together" – somewhat like many (changeable) logic gates that can be inter-wired in (many) different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory. Connecting I/O blocks to logic blocks is the programmable interconnect wiring.

The logic blocks within an FPGA can be small and simple, but the macrocells in a PLD are more complex with so-called fine-grained and coarse-grained architectures for larger and more complex designs. However, they are never as large as an entire PLD, the logic blocks of a CPLD contain multiple macrocells, but the logic blocks in an FPGA are generally nothing more than a couple of logic gates or a look-up table and a flip-flop. Because of all the extra flip-flops, the architecture of an FPGA is much more flexible than that of a CPLD. This makes FPGAs better in register-heavy and pipelined applications.

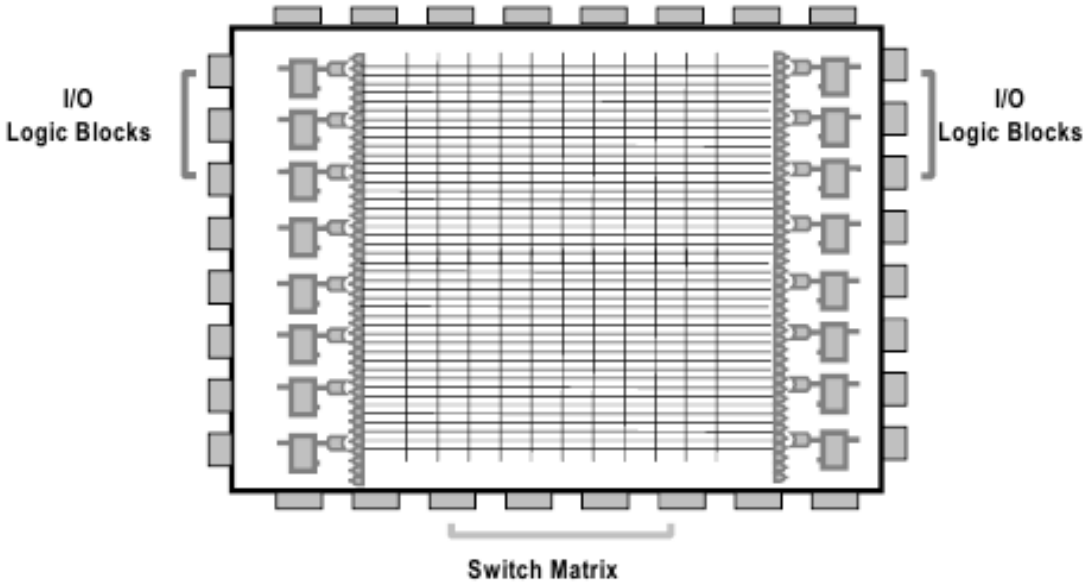


Figure 1-10: Internal Structure of a FPGA

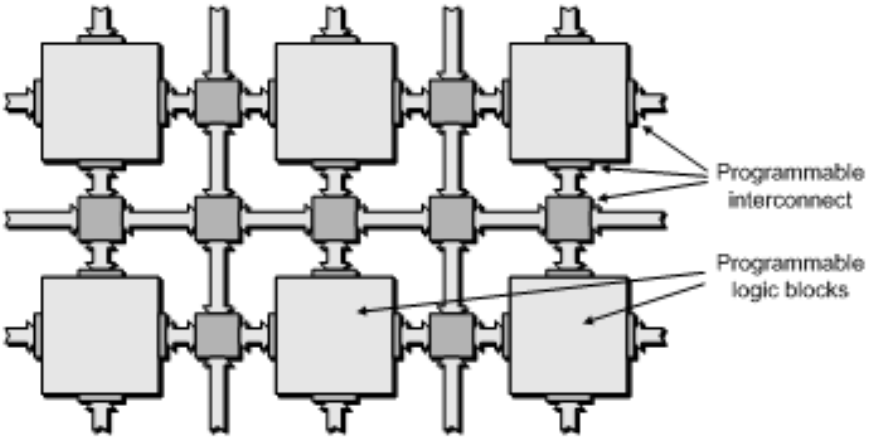


Figure 1-11: FPGA Underlying Structure

They are also often used in place of a processor-plus-software solution, particular where the processing of input data streams must be performed at a very fast pace. In addition, FPGAs are usually denser (more gates in a given area) and cost less than a CPLD equivalent, so they might be the best choice for larger logic designs.

Contemporary FPGAs have large resources of logic gates and RAM blocks to implement complex digital computations. As FPGA designs employ very fast I/Os and bidirectional data buses it becomes a challenge to verify correct timing of valid data within setup time and hold time. Floor planning enables resources allocation within FPGA to meet these time constraints.

Technically speaking an FPGA can be used to solve any problem which is computable. This is trivially proven by the fact FPGA can be used to implement a Soft microprocessor. Their advantage lies in that they are sometimes significantly faster for some applications due to their parallel nature and optimality in terms of the number of gates used for a certain process.

Specific applications of FPGAs include digital signal processing, software-defined radio, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation, radio astronomy, metal detection and a growing range of other areas.

FPGAs originally began as competitors to CPLDs and competed in a similar space, that of glue logic for PCBs. As their size, capabilities, and speed increased, they began to take over larger and larger functions to the state where some are now marketed as full systems on chips (SoC). Particularly with the introduction of dedicated multipliers into FPGA architectures in the late 1990s, applications which had traditionally been the sole reserve of DSPs began to incorporate FPGAs instead.

Traditionally, FPGAs have been reserved for specific vertical applications where the volume of production is small. For these low-volume applications, the premium that companies pay in hardware costs per unit for a programmable chip

is more affordable than the development resources spent on creating an ASIC for a low-volume application. Today, new cost and performance dynamics have broadened the range of viable applications.

2 Boolean Functions and Methods of their Minimization

2.1 Boolean Algebra

In 1854 George Boole introduced a systematic treatment of logic and developed for this purpose an algebraic system known as symbolic logic, or Boolean algebra. Boolean algebra is a branch of mathematics and it can be used to describe the manipulation and processing of binary information. The two-valued Boolean algebra has important application in the design of modern computing systems.

Boolean algebra uses ***Boolean variables*** and ***Boolean operators***. Boolean variables are binary variables and Boolean operators are logical operators. Some examples of Boolean variables are A, B, C, a, b, c, X, Y, and Z. There are three basic logical operators: AND, OR, and NOT. A ***Boolean expression*** is a combination of Boolean variables and Boolean operators. There are many Boolean expressions that are logically equivalent to one another. There are called ***equivalent expressions***.

A ***Boolean function*** typically has one or more input variables and produces a result that is based on these input values. The result can have a value of 0 or 1.

2.1.1 Truth Tables

A Boolean function can be uniquely and completely described using a ***truth table*** of n input variables. A truth table lists all possible values of input combinations of the function and the corresponding values of the outputs for all these input combinations. A truth table is a useful visual tool for defining the input-output relationship of binary variables in a Boolean function. A function of n variables has 2^n rows of possible input combinations, each row specifying the value of the function for a different combination. A truth table can be used to represent one or more functions.

Example:

x	y	F	a	b	F_1	F_2
0	0	0	0	0	0	0
0	1	1	0	1	0	1
1	0	1	1	0	0	1
1	1	1	1	1	1	0

2.1.2 Basic Boolean Laws (T1-T10):

Note that every law has two expressions, (a) and (b). This is known as *duality*. These are obtained by changing every AND (.) to OR (+), every OR (+) to AND (.) and all 1's to 0's and vice-versa.

- The duality principle states that if two Boolean expressions are equal, then their duals are also equal.

It has become conventional to drop the (AND symbol) i.e. A.B is written as AB.

T1: Commutative Laws

(a) $x + y = y + x$ (b) $x \cdot y = y \cdot x$

T2: Associative Laws

(a) $(x + y) + z = x + (y + z)$ (b) $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

T3: Distributive Law

(a) $x \cdot (y + z) = x \cdot y + x \cdot z$ (b) $x + (y \cdot z) = (x + y) \cdot (x + z)$

T4: Idempotent Laws

(a) $x + x = x$ (b) $x \cdot x = x$

T5: Absorption Laws

(a) $x + xy = x$ (b) $x \cdot (x + y) = x$

Laws of Boolean Addition and Multiplication

T6 (a): $x + 0 = x$ (***Identity law in OR form***)

(b): $x \cdot 0 = 0$ (***Null law in AND form***)

T7 (a): $x + 1 = 1$ (***Null law in OR form***)

(b): $x \cdot 1 = x$ (***Identity law in AND form***)

T8: Laws of Complementarity

(a) $x + \bar{x} = 1$ (***Inverse law in OR form***)

(b) $x \cdot \bar{x} = 0$ (***Inverse law in AND form***)

T9: (a) $x + \bar{x} \cdot y = x + y$ (b) $x \cdot (\bar{x} + y) = x \cdot y$

T10: DeMorgan's Laws

(a) $\overline{x + y} = \bar{x} \cdot \bar{y}$ (b) $\overline{x \cdot y} = \bar{x} + \bar{y}$

2.1.3 Representations of Boolean Functions

A Boolean function can be described or represented by any one of the following:

- A Boolean expression (one of many equivalent Boolean expressions)
- A truth table
- A circuit diagram (one of many equivalent circuit diagrams)

A Boolean function can be represented by many different equivalent expressions: a sum of product terms, a sum of minterms, a product of sum terms, or a product of maxterms. To help eliminate potential confusion, logical designers specify a Boolean function using a ***standardized form*** called ***canonical form***. A canonical form can be either a sum of minterms or a product of maxterms as described below.

A ***minterm*** is a product term that includes all variables of a function and each variable is either in uncomplemented form or in complemented (inversed) form. A minterm is also called a ***canonical product term***. A minterm is a product term, but a product term may or may not be a minterm.

A ***maxterm*** is a sum term of all variables in which each variable is either in complemented form or in uncomplemented form. A maxterm is also called a ***canonical sum term***. A maxterm is a sum term, but a sum term may or may not be a maxterm.

The following are examples of product term, minterm, sum term, and maxterm for a function of three variables a, b, and c:

product terms: $a, ac, \bar{b}c, abc, \bar{a}bc, \bar{a}\bar{b}\bar{c}, \ldots$

minterms: $a\bar{b}c, abc, \bar{a}\bar{b}c, \bar{a}\bar{b}\bar{c}, \ldots$

sum terms: $a, (a+b), (b+c), (\bar{a}+b), (\bar{a}+\bar{b}), \ldots$

maxterms: $(a+b+c), (a+\bar{b}+c), (\bar{a}+\bar{b}+\bar{c}) \ldots$

A minterm is represented by the symbol m_j, where the subscript j is the decimal equivalent of the minterm. A maxterm is represented by the symbol M_j, where the subscript j is the decimal equivalent of the maxterm.

Example (Exclusive OR function, XOR):

a	b	F	minterms	maxterms
0	0	0		$M_0 = (a + b)$
0	1	1	$m_1 = \bar{a}b$	
1	0	1	$m_2 = a\bar{b}$	
1	1	0		$M_3 = (\bar{a} + \bar{b})$

Either the sum of minterms or the product of maxterms as shown below can represent the Exclusive OR function:

Sum of minterms (Canonical SOP ("sum of products") form):

$F(a,b) = \bar{a}b + a\bar{b}$ (shorthand notation $F(a,b) = m_1 + m_2$; $F(a,b) = \sum m(1,2)$)

Product of maxterms (Canonical POS ("products of sum") form):

$F(a,b) = (a + b) \cdot (\bar{a} + \bar{b})$ (shorthand notation $F(a,b) = M_0 + M_3$;

$F(a,b) = \prod M(0,3)$

The expressions $\bar{a}b + a\bar{b}$ and $(a + b) \cdot (\bar{a} + \bar{b})$ are equivalent expressions for the XOR function.

The standard representation of a Boolean function is either a sum of minterms or a product of maxterms. However, Boolean functions are most frequently represented by a sum of product terms or a sum of minterms. These standard representations make the minimization procedure easier.

2.1.4 Equivalent Expressions and Equivalent Circuits

A Boolean function may be represented by several equivalent expressions. The equivalence of two Boolean expressions can be proved or disproved by comparing their truth tables as shown below:

a	b	$\bar{a}$	$\bar{b}$	$\bar{a}b$	$a\bar{b}$	$a+b$	$\bar{a}+\bar{b}$	$\bar{a}b+a\bar{b}$	$(a+b)(\bar{a}+\bar{b})$
0	0	1	1	0	0	0	1	0	0
0	1	1	0	1	0	1	1	1	1
1	0	0	1	0	1	1	1	1	1
1	1	0	0	0	0	1	0	0	0

Therefore, $\bar{a}b + a\bar{b} = (a + b) \cdot (\bar{a} + \bar{b})$. The expressions $\bar{a}b + a\bar{b}$ and $(a + b) \cdot (\bar{a} + \bar{b})$ are equivalent expressions for the XOR function. Converting one Boolean expression to another Boolean expression by Boolean algebraic manipulation also proves the equivalence of the two Boolean expressions:

$(a + b) \cdot (\bar{a} + \bar{b}) = a\bar{a} + a\bar{b} + \bar{a}b + b\bar{b} = 0 + \bar{a}b + a\bar{b} + 0 = \bar{a}b + a\bar{b}$.

Therefore, $\bar{a}b + a\bar{b} = (a+b)\cdot(\bar{a}+\bar{b})$

2.2 Minimization of Boolean Functions

The standard sum of minterms representation is usually not a minimized expression. A circuit designer generally starts with a Boolean function and finds a simpler but equivalent one. From the minimized expression, a circuit for that function can be constructed. Minimizing terms and expressions can be important because electrical circuits consist of individual components that are implemented for each term or literal for a given expression. This allows designers to make use of fewer components, thus reducing the cost of a particular system.

The following two approaches can be used for simplification of a Boolean expression:

1. Karnaugh map method
2. Algebraic method (using Boolean algebra rules)

2.2.1 Minimization of Boolean Expressions Using Karnaugh Maps

A Karnaugh map (K-map) method is a graphical way of minimizing a Boolean expression based on the rule of complementation. It works well if to use for expressions with not more than 4 variables. Hardware components of a computer are built of logic circuits to perform various logic functions. A logic function can be represented by a truth table, a circuit diagram, or a Boolean expression. The truth table of a function is unique. A function, however, can be represented by many equivalent Boolean expressions. A Boolean algebraic manipulation or a K-map may simplify a Boolean expression. The algebraic manipulation method is often difficult because it lacks specific rules for predicting each succeeding step in the manipulative process. The K- map method provides a simple, straightforward procedure for minimizing Boolean expressions.

The idea behind a K-map (Karnaugh, 1953) is to draw an expression's truth table as a matrix in such a way that each row and each column of the matrix puts minterms that differ in the value of a single variable adjacent to each other. Then, by grouping adjacent cells of the matrix, you can identify product terms that eliminate all complemented literals, resulting in a minimized version of the expression. The K- map can also be described as a special arrangement of a truth table.

The diagram below illustrates the correspondence between the K- map and the truth table for the general case of a two variable problem.

A	B	F
0	0	a
0	1	b
1	0	c
1	1	d

Truth Table.

A / B	0	1
0	a	b
1	c	d

F.

The values inside the squares are copied from the output column of the truth table, therefore there is one square in the map for every row in the truth table. Around the edge of the K- map are the values of the two input variable. A is along the top and B is down the left hand side. The diagram below explains this

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Truth Table.

A / B	0	1
0	0	1
1	1	1

F.

The values around the edge of the map can be thought of as coordinates. So as an example, the square on the top right hand corner of the map in the above diagram has coordinates A=1 and B=0. This square corresponds to the row in the truth table where A=1 and B=0 and F=1. Note that the value in the F column represents a particular function to which the K- map corresponds.

The following is a K-map minimization procedure for obtaining a minimal expression directly from a truth table. The map is a diagram made up of cells containing 1s (minterms).

Karnaugh Map Minimization Procedure includes the following steps:

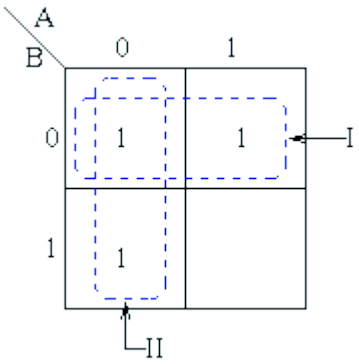
1. Construct a K-map.
2. Find all groups of horizontal or vertical adjacent cells that contain 1.
    a. Each group must be either rectangular or square with 1, 2, 4, 8, or 16 cells.
    b. Each group should be as large as possible.
    c. Each cell with 1 on the K-map must be covered at least once. The same cell can be included in several groups if necessary.
    d. Select the least number of groups so as to cover all the 1's.
    e. Adjacency applies to both vertical and horizontal borders.

3. Translate each group into a product term.
   (Any variable whose value changes from cell to cell drops out from the term)
4. Sum all the product terms.

Example 2.1 Consider the following expression

$$Z = f(A,B) = \overline{A}\overline{B} + A\overline{B} + \overline{A}B,$$

plotted on the K- map:



Pairs of 1's are *grouped* as shown above, and the simplified answer is obtained by using the following steps:

Note that two groups can be formed for the example given above, bearing in mind that the largest rectangular clusters that can be made consist of two 1s. Notice that a 1 can belong to more than one group.

The first group labeled I, consists of two 1s which correspond to A = 0, B = 0 and A = 1, B = 0. Put in another way, all squares in this example that correspond to the area of the map where B = 0 contains 1s, independent of the value of A. So when B = 0 the output is 1. The expression of the output will contain the term $\overline{B}$.

For group labeled II corresponds to the area of the map where A = 0. The group can therefore be defined as $\overline{A}$. This implies that when A = 0 the output is 1. The output is therefore 1 whenever B = 0 and A = 0. Hence the simplified answer is $Z = \overline{A} + \overline{B}$

Example 2.2 We will use the function with the following truth table as a running example:

a	b	c	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

This truth table can also be represented as the list of minterms. That is, the truth table has a 1 in the Y column for the rows where the binary number represented by the values of a, b, and c is one of the numbers listed. The other two rows (0 and 3) have a 0 in the Y column, and thus are not minterms.

- One standard way to represent any Boolean function is called "sum of products" (SOP) or, more formally, *disjunctive normal form*. In this form, the function is written as the logical OR (indicated by +) of a set of AND terms, one per minterm.

For example, the disjunctive normal form for our sample function would be:

$$Y = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}\bar{c} + a\bar{b}c + ab\bar{c} + abc$$

There is also a *conjunctive normal form,* which represents an expression as a product of sums rather than as a sum of products. The material presented below can be extended to deal with conjunctive normal forms rather than disjunctive normal forms. We'll leave that as one of those classic "exercises left for the student," and deal with just disjunctive normal forms.

Figure 2-1 shows how the minterms in truth tables are placed in a K-map grid for both 3 and 4 variable expressions.

Looking at the 3 variable map on the left in Figure 2-1, note that minterm 0 (000_2) is just above minterm 4 (100_2). This arrangement means that if both minterms 0 and 4 occur in a function, the first variable (the one named *a* in Figure 2-1) appears in both its true and complemented form, and can be eliminated.

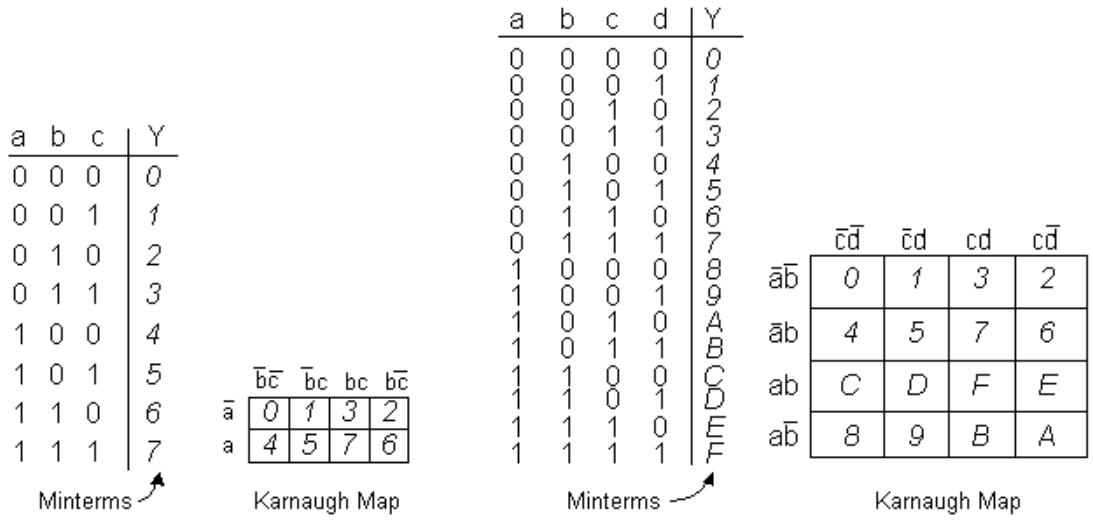


Figure 2-1: Truth tables and K- maps

The top row of the K- map is labeled with *a'* and the lower row with *a*: Any minterms appearing in the top row have the literal *a'* in them, and any minterms in the bottom row have the literal *a* in them. At the same time, note that each column has the same values for the variables *b* and *c*. Also, the columns are arranged in an order so that only one variable changes value as you go from one column to the next. Thus, the first two columns differ in the value of *c*, the second and third columns differ in the value of *b*, and the third and fourth columns differ in the value of *c* again. Furthermore, the first and fourth columns are "next to" each other as well, because they differ from each other just in the value of *b*.

The right side of Figure 2-1 shows that this same pattern (adjacent *columns* differ by the value of a single variable) applies to the *rows* of a K-map too: The first and second rows of that map differ in the value of *b*, the second and third differ in the value of *a*, the third and fourth differ in the value of *b*, and the first and fourth differ in the value of *a*.

Figure 2-2 shows our sample function drawn as a K-map. Minterms 1 and 2 are in the second and fourth columns of the top row, while minterms 4, 5, 7, and 6 appear from left to right in the four columns of the bottom row.

A K- map is used to produce a minimal sum of products implementation of an expression by drawing rectangles around groups of adjacent minterms in the map; each rectangle will correspond to one product term, and the full expression will be constructed as the OR (sum) of all the product terms. The goal is to have as few product terms as possible, which implies that each product term will account for as many minterms as possible.

a	b	c	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

	$\bar{b}\bar{c}$	$\bar{b}c$	bc	$b\bar{c}$
$\bar{a}$	0	1	0	1
a	1	1	1	1

Figure 2-2: Truth table and K- map

Here are the rules for drawing the rectangles:

- Every minterm must be inside at least one rectangle, but there must not be any zeros inside any rectangles.
- Every rectangle has to be as large as possible.
- Rectangles may wrap around to include cells in both the leftmost and rightmost columns; likewise for the top and bottom rows.
- The number of minterms enclosed in a rectangle must be a power of two (1, 2, 4, 8, or 16 minterms for 4-variable maps).
- Some functions have "don't care" conditions, which are combinations of inputs that will never occur, resulting in cases where it doesn't matter whether the output is a zero or a one. Where these don't care conditions appear in a K- map (usually indicated by X's instead of ones or zeros), they may be included inside rectangles or not depending on what will make the rectangles as few and as large as possible.

Figure 2-3 shows the rectangles for our sample function, following the procedure outlined above:

	$\bar{b}\bar{c}$	$\bar{b}c$	bc	$b\bar{c}$
$\bar{a}$	0	1	0	1
a	1	1	1	1

Figure 2-3: K-Map for 3 inputs

The largest rectangle (the bottom row) corresponds to the product term *a*. By enclosing four minterms, two variables have been eliminated resulting in a single product term with a single variable. The rectangle in the second column encloses two minterms, eliminating one variable (*a*) from that product term. Similarly, the rectangle in the fourth column eliminates *a* from that product term. The resulting sum of products function is: $y = a + \bar{b}c + b\bar{c}$.

Every time you double the number of minterms inside a rectangle, you eliminate one variable from the resulting product term. Every doubling corresponds to applying the rule of complementation again.

2.2.2 Algebraic Minimization

This is an approach where you can transform one Boolean expression into an equivalent expression by applying Boolean theorems and laws. It should be noted that there are no fixed rules that can be used to minimize a given expression. It is left to an individuals' ability to apply Boolean theorems and laws in order to minimize a function.

Example 2.3 Minimization the following functions using algebraic method:

$$Z = (A + \bar{B} + \bar{C})(A + \bar{B}C)$$
$$Z = AA + A\bar{B}C + A\bar{B} + \bar{B}\bar{B}C + A\bar{C} + \bar{B}C\bar{C}$$
$$Z = A(1 + \bar{B}C + \bar{B} + \bar{C}) + \bar{B}C + \bar{B}C\bar{C} \quad \text{from laws T8b and Tb}$$
$$Z = A + \bar{B}C \quad \text{from laws T8a, T8b and T9b}$$

Example 2.4 Minimization of $F(a,b) = \overline{a+b} + a\bar{b}$ using the algebraic method:

$$\overline{a+b} + a\bar{b} = \bar{a}\bar{b} + a\bar{b} = (\bar{a}+a)\bar{b} = (1)\bar{b} = \bar{b}$$

The algebraic manipulation method is easy and simple for this Boolean expression

Minimizing an expression algebraically involves repeatedly applying the rule of complementation, starting with the disjunctive normal form of the function, and ending with a set of product terms called pri*me implicants.* A prime implicant is a product term that will generate ones only for combinations of inputs that are minterms of the disjunctive normal form of the function, and which cannot be further reduced by combining with any other term. They correspond to the rectangles in a K-map.

It takes two steps to minimize our sample function. The following chart shows the original disjunctive normal form of the function as Step 0 and shows what reductions are performed for the other two passes.

Step 0: $\bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}\bar{c} + a\bar{b}c + ab\bar{c} + abc$

Step 1: $\bar{a}\bar{b}c + a\bar{b}c$ reduces to $\bar{b}c$

$\bar{a}b\bar{c} + ab\bar{c}$ reduces to $\bar{b}c$

$a\bar{b}c + a\bar{b}\bar{c}$ reduces to $a\bar{b}$

$ab\bar{c} + abc$ reduces to ab

$\bar{b}c + \bar{b}c + a\bar{b} + ab$

Step 2: $a\bar{b} + ab$ reduces to a

$a + \bar{b}c + \bar{b}c = a + \bar{b}c$

The rules to follow for each step are:

- Each term present in one step must be combined with another term if possible.
- Any terms that cannot be combined are carried forward unchanged to the next step.

- A term that has already been used once in a pass should be used again if it will allow another term to be reduced. For example, in Step 1 above, $a\bar{b}c$ is used twice, and so is $ab\bar{c}$.

The rule about reusing terms in a step corresponds to circling some minterms more than once in a K-map. The two minterms that are reused in Pass 1 above are exactly the same two that are circled twice in the K-map of Figure 2-3.

Once the prime implicants of an expression have been determined, a minimal subset of them has to be selected. Picking a minimal subset of prime implicants relies on the notion of minterms being "covered" by prime implicants. For our sample function, the prime implicant *a* covers minterms 4, 5, 6, and 7; the prime implicant $\bar{b}c$ covers minterms 1 and 5, and the prime implicant $b\bar{c}$ covers minterms 2 and 6. In this example, we need all three prime implicants in order to cover all the minterms at least one, and the expression shown at the end of Step 2 is the minimized form for our sample function.

But whenever there is more than one minimal form for an expression, the different forms will correspond to different subsets of the complete set of prime implicants. For any one of the minimal forms there will be extra prime implicants that have to be discarded.

The following procedure describes a way to determine one minimal form of an expression after all the prime implicants have been determined.

- For every minterm that is covered by just one prime implicant, that prime implicant must be included in the minimal form. These minterms are called *essential prime implicants* because it is essential to include them in the minimization.

For our sample function, minterms 2, 3, and 4 are each covered by exactly one prime implicant, so all three of the prime implicants are essential, there is just one minimized form, and there is nothing more to do.

- Make a list of all minterms that are not covered by any of the essential prime implicants.

- Make a list of unused prime implicants. Order this list by the number of literals each prime implicant contains.

- If any remaining minterms are covered by just one of the remaining prime implicants, the corresponding prime implicants must be added to the minimization, and all the minterms they cover must be removed from the list of uncovered minterms.

- If there are any uncovered minterms add the unused prime implicant with the smallest number of literals to the minimization, and remove all minterms that are covered by this prime implicant from the list of uncovered minterms.

If two or more prime implicants have equally small numbers of literals, there is more than one minimal solution. Finding them all involves systematically substituting each of the tied prime implicants into the different minimal forms being generated.

- Remove all prime implicants that fail to cover any of the remaining minterms from the list of unused prime implicants.

- Repeat the previous three steps until all minterms have been covered.

3 Combinational Logic

3.1 Introduction

Digital logic gates as the basic building blocks of digital circuits and systems. These circuit elements will receive and produce binary digital values (0/1) and the operation of a digital logic gate is to modify the input bit logic values in order to produce the required output logic values.

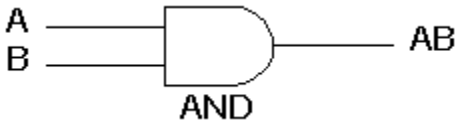
Boolean functions may be practically implemented by using electronic gates. The following points are important to understand.

- Electronic gates require a power supply.
- Gate **INPUTS** are driven by voltages having two nominal values, e.g. 0V and 5V representing logic 0 and logic 1 respectively.
- The **OUTPUT** of a gate provides two nominal values of voltage only, e.g. 0V and 5V representing logic 0 and logic 1 respectively. In general, there is only one output to a logic gate except in some special cases.
- There is always a time delay between an input being applied and the output responding.

3.2 Combinational Logic Gates

Digital systems are said to be constructed by using logic gates. These gates are the AND, OR, NOT, NAND, NOR, EXOR and EXNOR gates. The basic operations are described below with the aid of truth tables.

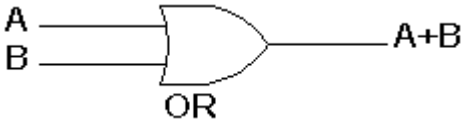
AND gate



2 Input AND gate		
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

The AND gate is an electronic circuit that gives a **high** output (1) only if **all** its inputs are high. A dot (.) is used to show the AND operation i.e. A.B. Bear in mind that this dot is sometimes omitted i.e. AB

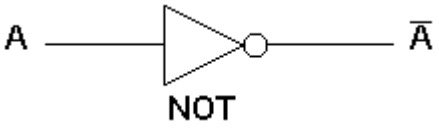
OR gate



2 Input OR gate		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

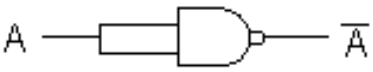
The OR gate is an electronic circuit that gives a high output (1) if **one or more** of its inputs are high. A plus (+) is used to show the OR operation.

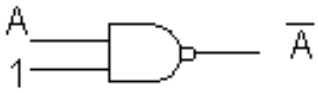
NOT gate



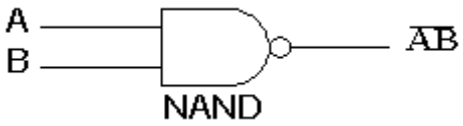
NOT gate	
A	$\overline{A}$
0	1
1	0

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an ***inverter.*** If the input variable is A the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top, as shown at the outputs. The diagrams below show two ways that the NAND logic gate can be configured to produce a NOT gate. It also can be done using NOR logic gates in the same way.





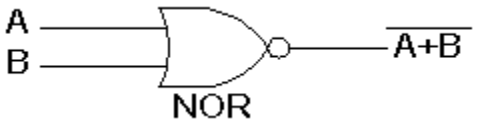
NAND gate



2 Input NAND gate		
A	B	$\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

This is a NOT-AND gate which is equal to an AND gate followed by a NOT gate. The outputs of all NAND gates are high if **any** of the inputs are low. The symbol is an AND gate with a small circle on the output. The small circle represents inversion.

NOR gate

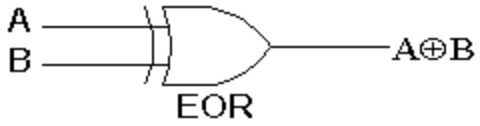


2 Input NOR gate		
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate. The outputs of all NOR gates are low if **any** of the inputs are high.

The symbol is an OR gate with a small circle on the output. The small circle represents inversion.

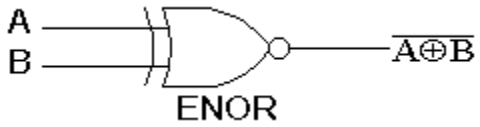
EXOR gate



2 Input EXOR gate		
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

The '**Exclusive-OR**' gate is a circuit which will give a high output if **either, but not both**, of its two inputs are high. An encircled plus sign (⊕) is used to show the EOR operation.

EXNOR gate



2 Input EXNOR gate		
A	B	$\overline{A⊕B}$
0	0	1
0	1	0
1	0	0
1	1	1

The '**Exclusive-NOR'** gate circuit does the opposite to the EOR gate. It will give a low output if **either, but not both**, of its two inputs are high. The symbol is an EXOR gate with a small circle on the output. The small circle represents inversion.

The NAND and NOR gates are called *universal functions* since with either one the AND and OR functions and NOT can be generated.

Note:

A function in *sum of products* (SOP) form can be implemented using NAND gates by replacing all AND and OR gates by NAND gates.

A function in *product of sums* (POS) form can be implemented using NOR gates by replacing all AND and OR gates by NOR gates.

Example 3.1

A NAND gate can be used as a NOT gate using either of the following wiring configurations.

(You can check this out using a truth table.)

The logic gates are represented by:

- Logic symbol, used when drawing a circuit diagram.
- Boolean Equation, defines the operation of the gate in terms of Boolean logic. Used in Boolean algebra expressions.
- Truth Table, shows the output logic value for all possible input combinations in a tabular format.

Logic can be either positive logic or negative logic:

- Positive logic

An output is logic 1 when a condition set by the inputs is true.

- Negative logic

An output is logic 0 when a condition set by the inputs is true.

In the following discussions, positive logic will be used unless otherwise stated.

3.3 Combinational logic design using Truth Tables

A truth-table is a table that, for a given logic function, identifies the possible logic inputs and the corresponding logic outputs. For combinational logic, when the input logic levels change, there would ideally be an instantaneous change in the logic output. However, in real circuits, there will be a small but finite delay due to the logic gates and the interconnection between the logic gates.

The general form of a 3-input truth-table is shown in Figure 3-1. Here, the three inputs are A, B and C. The output is Z.

The input binary values change in increments of 1 from a low value of 0 (decimal) to 7 (decimal) in a direction from top to bottom – this is a straight binary count. In the following discussions, the input values will change in this manner. The number of possible combinations of input will be dependent on the number of logic inputs. There will be 2n possible input combinations where n is the number of inputs (in Figure 3-1, n = 3).

In this example, the logic value of the output will only be logic 1 when the input is either 4 or 5 (decimal).

The resulting Boolean expression will be referred to as a sum of products. Where the output is logic 1 in a row, then this will be added to the Boolean expression as the AND of the input values (when the input is a 1, the input term is used: when the input is a 0, the inverse (NOT) of the input is used). Each row term will then be OR-ed.

Decimal Equivalent of Binary Input

Input Value
0
1
2
3
4
5
6
7

TruthTable

A	B	C	Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Figure 3-1: 3 Input Truth Table

For the example in Figure 3-1, then the resulting Boolean expression is:

$$Z = \left(A \cdot \overline{B} \cdot \overline{C}\right) + \left(A \cdot \overline{B} \cdot C\right) \qquad (3.1)$$

This expression can be reduced by considering where terms can be removed. The expression can be re-written as:

$$Z = \left(A \cdot \overline{B}\right) \cdot \left(\overline{C} \cdot C\right) \qquad (3.2)$$

which reduces to:

$$Z = \left(A \cdot \overline{B}\right) \qquad (3.3)$$

This can also be seen by looking at the truth-table directly:

Z is logic 1 when A is logic 1 AND B is logic 0 AND C is either a logic 0 or logic 1.

As C can be either logic value, it doesn't matter what the logic value is and so can be removed from the expression.

The required Boolean logic expression can be represented by the truth-table and from this table a reduced expression (if possible) could be developed, either by looking into the patterns of output logic values directly from the truth-table, or by a combination of the truth-table and Boolean logic manipulation.

An important logic design that is created from the basic logic gates is the half-adder, see Figure 3-2. This is a design with two inputs (A and B) and two outputs (Sum and Carry-Out (Cout)). This cell performs the addition of the two binary input numbers and produces sum and carry-out terms.

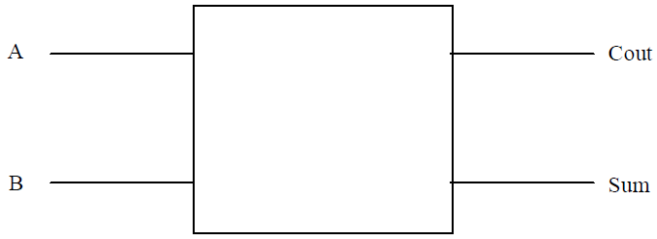


Figure 3-2: Half-adder cell

The truth-table for this design is shown in Figure 3-3.

A	B	Sum	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 3-3: Half-adder cell truth table

From viewing the truth-table, then the Sum output is only logic 1 when either, but not both, inputs are logic 1:

$$Sum = \left(\overline{A} \cdot B\right) + \left(A \cdot \overline{B}\right) \qquad (3.4)$$

This is actually the Exclusive-OR function, so:

$$Sum = A \oplus B \qquad (3.5)$$

From viewing the Cout output in the truth table, then the output is logic 1 only when all two inputs are logic 1 (i.e. A AND B):

$$Cout = A \cdot B \qquad (3.6)$$

This can be drawn as a logic diagram as shown in Figure 3-4.

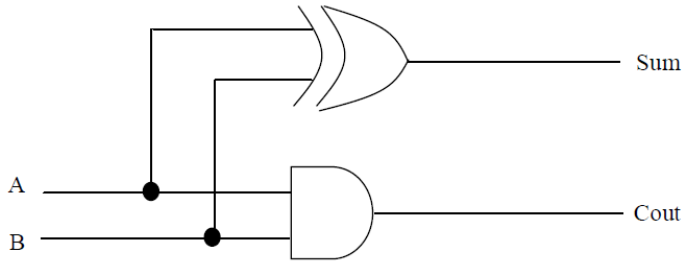


Figure 3-4: Half-adder logic diagram

An extension of the half-adder is the full adder, see Figure 3-5.

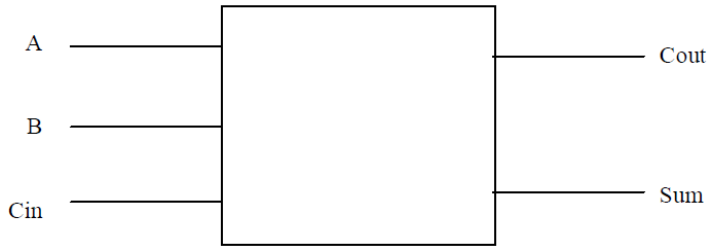


Figure 3-5: Full-adder cell

The truth-table is shown in Figure 3-6.

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 3-6: Full-adder cell truth table

This now has the two original inputs to be added (A and B), but also now there is a carry-in bit (C in). The Boolean logic for this cell is:

$$Sum = A \oplus (B \oplus Cin) \qquad (3.7)$$

$$Cout = Cin \cdot A \cdot B \qquad (3.8)$$

Conclusion:

This lecture has introduced the design of combinational logic circuits using truth tables and Karnaugh Maps. Examples were provided how these representations of a logic function can be derived and analyzed. The aim of the design process should be to minimize the amount of logic required to implement the Boolean expression.

4 Flip-Flop Types

4.1 Introduction

Flip-flops are digital logic circuits that can be in one of two states. Flip-flops maintain their state indefinitely until an input pulse called a trigger is received. When a trigger is received, the flip-flop outputs change state according to defined rules and remain in those states until another trigger is received. Flip-flop circuits are interconnected to form the logic gates for the digital integrated circuits (IC s) used in memory chips and microprocessors. Flip-flops can be used to store one bit, or binary digit, of data.

The first electronic flip-flop was invented in 1919 by William Henry Eccles and Frank Wilfred Jordan. It used vacuum tubes and was initially called the *Eccles-Jordan trigger circuit*.

Flip-flops can be divided into common types: the **SR** ("set-reset"), **D** ("data" or "delay"), **T** ("toggle"), and **JK** types are the common ones. The behavior of a particular type can be described by what is termed the characteristic equation, which derives the "next" (i.e., after the next clock pulse) output, Q_{next} in terms of the input signal(s) and/or the current output, Q .

4.2 SR Flip-flop

4.2.1 Simple set-reset latches

4.2.1.1 SR NOR latch

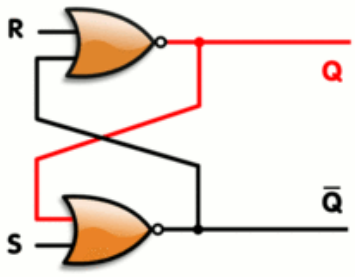


Figure 4-1: SR latch, constructed from a pair of NOR gates

(red and black mean logical '1' and '0', respectively)

When using static gates as building blocks, the most fundamental latch is the simple *SR latch*, where S and R stand for *set* and *reset*. It can be constructed from a pair of cross-coupled NOR logic gates. The stored bit is present on the output marked Q.

While the S and R inputs are both low, feedback maintains the Q and Q' outputs in a constant state, with Q' the complement of Q. If S (*Set*) is pulsed high while R (*Reset*) is held low, then the Q output is forced high, and stays high when S returns to low; similarly, if R is pulsed high while S is held low, then the Q output is forced low, and stays low when R returns to low.

The R = S = 1 combination is called a **restricted combination** or a **forbidden state** because, as both NOR gates then output zeros, it breaks the logical equation Q = **not** Q. The combination is also inappropriate in circuits where *both* inputs may go low *simultaneously* (i.e. a transition from *restricted* to *keep*). The output would lock at either 1 or 0 depending on the propagation time relations between the gates (a race condition).

SR latch operation							
Characteristic table				Excitation table			
S	**R**	**Q_{next}**	**Action**	**Q**	**Q_{next}**	**S**	**R**
0	0	Q	hold state	0	0	0	X
0	1	0	reset	0	1	1	0
1	0	1	set	1	0	0	1
1	1	X	not allowed	1	1	X	0

Note: X means don't care*, that is, either 0 or 1 is a valid value.*

Characteristic: Q+ = R'Q + R'S or Q+ = R'Q + S.

To overcome the restricted combination, one can add gates to the inputs that would convert (S, R) = (1, 1) to one of the non-restricted combinations. That can be:

- Q = 1 (1,0) – referred to as an *S (dominated)-latch*
- Q = 0 (0,1) – referred to as an *R (dominated)-latch*
- Keep state (0,0) – referred to as an *E-latch*

Alternatively, the restricted combination can be made to *toggle* the output. The result is the JK latch.

4.2.1.2 SR NAND latch

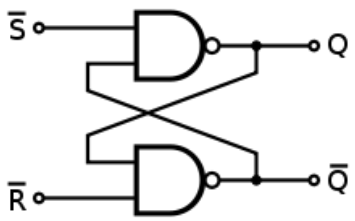


Figure 4-2: SR latch

This is an alternate model of the simple SR latch which is built with NAND logic gates. *Set* and *reset* now become active low signals, denoted S and R respectively. Otherwise, operation is identical to that of the SR latch. Historically, SR-latches have been predominant despite the notational inconvenience of active-low inputs.

SR latch operation		
S	**R**	**Action**
0	0	Restricted combination
0	1	Q = 1
1	0	Q = 0
1	1	No Change

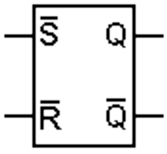


Figure 4-3: Symbol for an S-R-NAND latch

4.2.1.3 JK latch

The JK latch is much less frequently used than the JK flip-flop. The JK latch follows the following state table:

JK latch truth table			
J	**K**	**Q_{next}**	**Comment**
0	0	Q	No change
0	1	0	Reset
1	0	1	Set
1	1	Q	Toggle

Hence, the JK latch is an SR latch that is made to *toggle* its output (oscillate between 0 and 1) when passed the input combination of 1,1. Unlike the JK flip-flop, the 1,1 input combination for the JK latch is not very useful because there is no clock that directs toggling.

4.2.2 Gated latches and conditional transparency

Latches are designed to be *transparent.* That is, input signal changes cause immediate changes in output; when several *transparent* latches follow each other, using the same enable signal, signals can propagate through all of them at once. Alternatively, additional logic can be added to a simple transparent latch to make it *non-transparent* or *opaque* when another input (an "enable" input) is not asserted. By following a *transparent-high* latch with a *transparent-low* (or *opaque-high*) latch, a master–slave flip-flop is implemented.

4.2.2.1 Gated SR latch

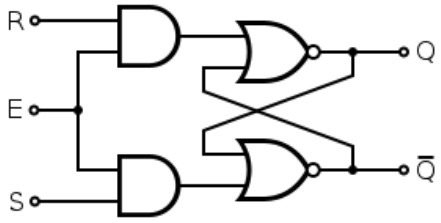


Figure 4-4: Gated SR latch circuit diagram from NOR gates

A ***synchronous SR latch*** (sometimes *clocked SR flip-flop*) can be made by adding a second level of NAND gates to the inverted SR latch (or a second level of AND gates to the direct SR latch). The extra NAND gates further invert the inputs so the simple SR latch becomes a gated SR latch (and a simple SR latch would transform into a gated SR latch with inverted enable).

With E high (*enable* true), the signals can pass through the input gates to the encapsulated latch; all signal combinations except for (0, 0) = *hold* then immediately reproduce on the (Q,Q') output, i.e. the latch is *transparent*.

With E low (*enable* false) the latch is *closed (opaque)* and remains in the state it was left the last time E was high.

The *enable* input is sometimes a clock signal, but more often a read or write strobe.

Gated SR latch operation	
E/C	**Action**
0	No action (keep state)
1	The same as non-clocked SR latch

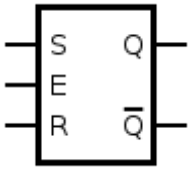


Figure 4-5: Symbol for a gated SR latch

4.2.2.2 Gated D latch

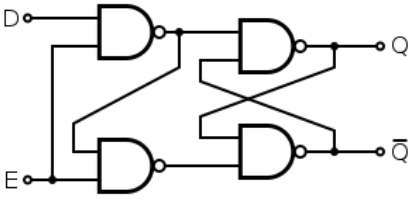


Figure 4-6: D-type transparent latch based on an SR NAND latch

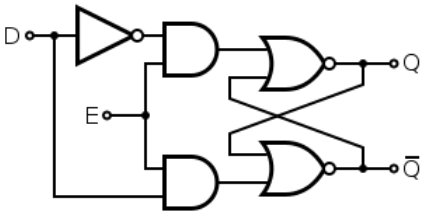


Figure 4-7: Gated D latch based on an SR NOR latch

This latch exploits the fact that, in the two active input combinations (01 and 10) of a gated SR latch, R is the complement of S. The input NAND stage converts the two D input states (0 and 1) to these two input combinations for the next SR latch by inverting the data input signal. The low state of the *enable*-signal produces the inactive "11" combination. Thus a gated D-latch may be considered as a *one-input synchronous SR latch*. This configuration prevents application of the restricted input combination. It is also known as *transparent latch*, *data latch*, or simply *gated latch*. It has a *data* input and an *enable*-signal (sometimes named *clock*, or *control*). The word *transparent* comes from the fact that, when the enable input is on, the signal propagates directly through the circuit, from the input D to the output Q.

Transparent latches are typically used as I/O ports or in asynchronous systems, or in synchronous two-phase systems (synchronous systems that use a two-phase clock), where two latches operating on different clock phases prevent data transparency as in a master–slave flip-flop.

Gated D latch truth table

E/C	D		Q	Q	Comment
0	X		Q_{prev}	Q_{prev}	No change
1	0		0	1	Reset
1	1		1	0	Set

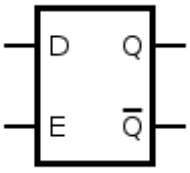


Figure 4-8: Symbol for a gated D latch

The truth table shows that when the enable/clock input is 0, the D input has no effect on the output. When E/C is high, the output equals D.

4.2.2.3 Earle latch

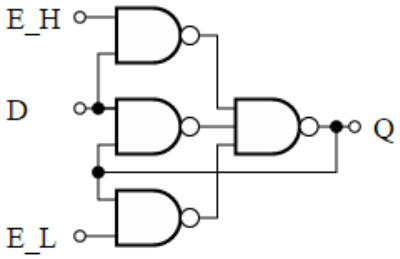


Figure 4-9: Earle latch uses complementary enable inputs

The classic gated latch designs have some undesirable characteristics. They require double-rail logic or an inverter. The input-to-output propagation may take up to three gate delays. The input-to-output propagation is not constant – some outputs take two gate delays while others take three.

Designers looked for alternatives. A successful alternative is the Earle latch. It requires only a single data input, and its output takes a constant two gate delays. In addition, the two gate levels of the Earle latch can, in some cases, be merged with the last two gate levels of the circuits driving the latch because many common computational circuits have an OR layer followed by an AND layer as their last two levels. Merging the latch function can implement the latch with no additional gate delays. The merge is commonly exploited in the design of pipelined computers, and, in fact, was originally developed by J. G. Earle to be used in the IBM System/360 Model 91 for that purpose.

The Earle latch is hazard free. If the middle NAND gate is omitted, then one gets the **polarity hold latch**, which is commonly used because it demands less logic. However, it is susceptible to logic hazard. Intentionally skewing the clock signal can avoid the hazard.

4.2.3 D flip-flop

The D flip-flop is widely used. It is also known as a "data" or "delay" flip-flop.

The D flip-flop captures the value of the D-input at a definite portion of the clock cycle (such as the rising edge of the clock). That captured value becomes the Q output. At other times, the output Q does not change. The D flip-flop can be viewed as a memory cell, a zero-order hold, or a delay line.

Truth table:

Clock	D	Q_{next}
Rising edge	0	0
Rising edge	1	1
Non-Rising	X	Q

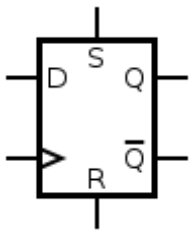


Figure 4-10: D flip-flop symbol

('X' denotes a *Don't care* condition, meaning the signal is irrelevant)

Most D-type flip-flops in ICs have the capability to be forced to the set or reset state (which ignores the D and clock inputs), much like an SR flip-flop. Usually, the illegal S = R = 1 condition is resolved in D-type flip-flops. By setting S = R = 0, the flip-flop can be used as described above. Here is the truth table for the others S and R possible configurations:

Inputs				Outputs	
S	R	D	>	Q	Q'
0	1	X	X	0	1
1	0	X	X	1	0
1	1	X	X	1	1

4.2.3.1 Classical positive-edge-triggered D flip-flop

This circuit consists of two stages implemented by SR NAND latches. The input stage (the two latches on the left) processes the clock and data signals to ensure correct input signals for the output stage (the single latch on the right). If the clock is low, both the output signals of the input stage are high regardless of the data input; the output latch is unaffected and it stores the previous state. When the clock signal changes from low to high, only one of the output voltages (depending on the data signal) goes low and sets/resets the output latch: if D = 0, the lower output becomes low; if D = 1, the upper output becomes low. If the clock signal continues staying high, the outputs keep their states regardless of the data input and force the output latch to stay in the corresponding state as the input logical zero remains active while the clock is high. Hence the role of the output latch is to store the data only while the clock is low.

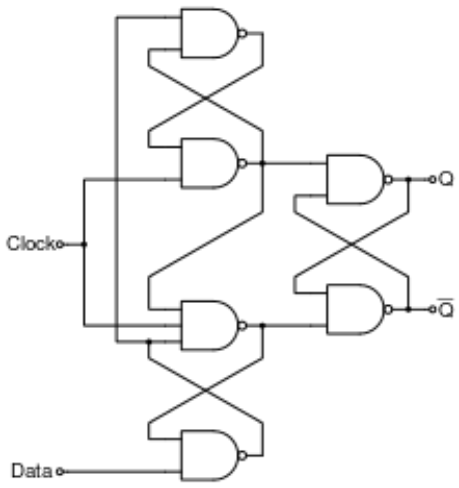


Figure 4-11: Positive-edge-triggered D flip-flop

The circuit is closely related to the gated D latch as both the circuits convert the two D input states (0 and 1) to two input combinations (01 and 10) for the output SR latch by inverting the data input signal (both the circuits split the single D signal in two complementary S and R signals). The difference is that in the gated D latch simple NAND logical gates are used while in the positive-edge-triggered D flip-flop SR NAND latches are used for this purpose. The role of these latches is to "lock" the active output producing low voltage (a logical zero); thus the positive-edge-triggered D flip-flop can be thought of as a gated D latch with latched input gates.

4.2.3.2 Master–slave edge-triggered D flip-flop

A master–slave D flip-flop is created by connecting two gated D latches in series, and inverting the *enable* input to one of them. It is called master–slave because the second latch in the series only changes in response to a change in the first (master) latch.

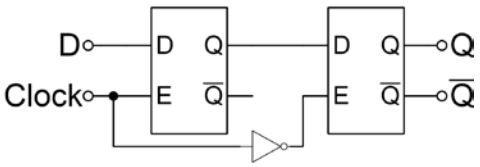


Figure 4-12: Falling edge master–slave D flip-flop

It responds on the falling edge of the *enable* input (usually a clock)

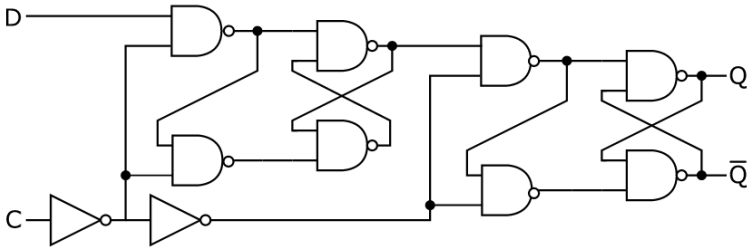


Figure 4-13: Rising edge master–slave D flip-flop

It responds on the rising edge of the enable input (usually a clock)

For a positive-edge triggered master–slave D flip-flop, when the clock signal is low (logical 0) the "enable" seen by the first or "master" D latch (the inverted clock signal) is high (logical 1). This allows the "master" latch to store the input value when the clock signal transitions from low to high. As the clock signal goes high (0 to 1) the inverted "enable" of the first latch goes low (1 to 0) and the value seen at the input to the master latch is "locked". Nearly simultaneously, the twice inverted "enable" of the second or "slave" D latch transitions from low to high (0 to 1) with the clock signal. This allows the signal captured at the rising edge of the clock by the now "locked" master latch to pass through the "slave" latch. When the clock signal returns to low (1 to 0), the output of the "slave" latch is "locked", and the value seen at the last rising edge of the clock is held while the "master" latch begins to accept new values in preparation for the next rising clock edge. By removing the leftmost inverter in the circuit at side, a D-type flip-flop that strobes on the *falling edge* of a clock signal can be obtained. This has a truth table like this:

D	Q	>	Q_{next}
0	X	Falling	0
1	X	Falling	1

4.2.3.3 Edge-triggered dynamic D storage element

An efficient functional alternative to a D flip-flop can be made with dynamic circuits (where information is stored in a capacitance) as long as it is clocked often enough; while not a true flip-flop, it is still called a flip-flop for its functional role. While the master–slave D element is triggered on the edge of a clock, its components are each triggered by clock levels. The "edge-triggered D flip-flop", as it is called even though it is not a true flip-flop, does not have the master–slave properties.

Edge-triggered D flip-flops are often implemented in integrated high-speed operations using dynamic logic. This means that the digital output is stored on parasitic device capacitance while the device is not transitioning. This design of dynamic flip flops also enables simple resetting since the reset operation can be performed by simply discharging one or more internal nodes. A common dynamic flip-flop variety is the true single-phase clock (TSPC) type which performs the flip-flop operation with little power and at high speeds. However, dynamic flip-flops will typically not work at static or low clock speeds: given enough time, leakage paths may discharge the parasitic capacitance enough to cause the flip-flop to enter invalid states.

4.2.4 T flip-flop

If the T input is high, the T flip-flop changes state ("toggles") whenever the clock input is strobed. If the T input is low, the flip-flop holds the previous value. This behavior is described by the characteristic equation:

$$Q_{next} = T \oplus Q = T\overline{Q} + \overline{T}Q$$ (expanding the XOR operator)

and can be described in a truth table:

T flip-flop operation							
Characteristic table				**Excitation table**			
T	Q	Q_{next}	**Comment**	Q	Q_{next}	T	**Comment**
0	0	0	hold state (no clk)	0	0	0	No change
0	1	1	hold state (no clk)	1	1	0	No change
1	0	1	toggle	0	1	1	Complement
1	1	0	toggle	1	0	1	Complement

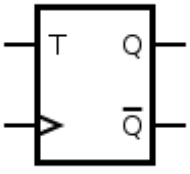


Figure 4-14: Symbol for a T-type flip-flop

When T is held high, the toggle flip-flop divides the clock frequency by two; that is, if clock frequency is 4 MHz, the output frequency obtained from the flip-flop will be 2 MHz. This "divide by" feature has application in various types of digital counters. A T flip-flop can also be built using a JK flip-flop (J & K pins are connected together and act as T) or D flip-flop (T input and $Q_{previous}$ is connected to the D input through an XOR gate).

4.2.5 JK flip-flop

The JK flip-flop augments the behavior of the SR flip-flop (J=Set, K=Reset) by interpreting the S = R = 1 condition as a "flip" or toggle command. Specifically, the combination J = 1, K = 0 is a command to set the flip-flop; the combination J = 0, K = 1 is a command to reset the flip-flop; and the combination J = K = 1 is a command to toggle the flip-flop, i.e., change its output to the logical complement of its current value. Setting J = K = 0 does NOT result in a D flip-flop, but rather, will hold the current state. To synthesize a D flip-flop, simply set K equal to the complement of J. Similarly, to synthesize a T flip-flop, set K equal to J. The JK flip-flop is therefore a universal flip-flop, because it can be configured to work as an SR flip-flop, a D flip-flop, or a T flip-flop.

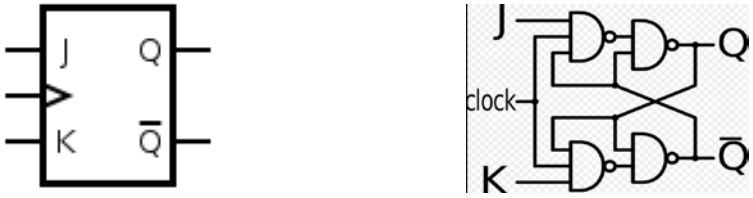


Figure 4-15: Symbol and NAND implementation for a JK flip-flop

The characteristic equation of the JK flip-flop is:

$$Q_{next} = J\overline{Q} + \overline{K}Q$$

and the corresponding truth table is:

JK flip-flop operation								
Characteristic table				**Excitation table**				
J	**K**	**Comment**	**Q_{next}**	**Q**	**J**	**K**	**Comment**	**Q_{next}**
0	0	hold state	Q	0	0	X	No Change	0
0	1	reset	0	0	1	X	Set	1
1	0	set	1	1	X	1	Reset	0
1	1	toggle	Q	1	X	0	No Change	1

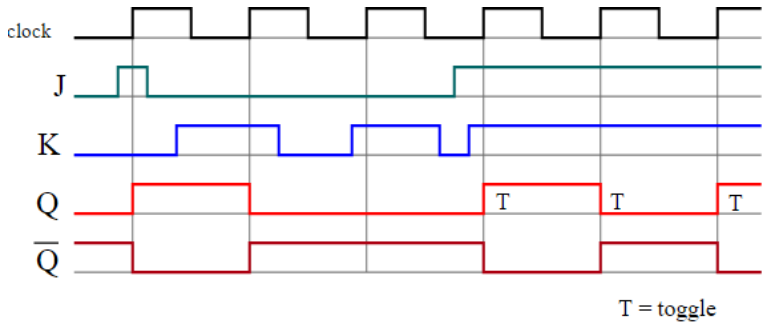


Figure 4-16: JK flip-flop timing diagram

4.2.6 Timing consideration

Setup time (t_{su}) is the minimum amount of time the data signal should be held steady **before** the clock event so that the data is reliably sampled by the clock. This applies to synchronous input signals to the flip-flop.

Hold time (t_h) is the minimum amount of time the data signal should be held steady **after** the clock event so that the data are reliably sampled. This applies to synchronous input signals to the flip-flop.

Synchronous signals (like Data) should be held steady from the set-up time to the hold time, where both times are relative to the clock signal.

Recovery time is like setup time for asynchronous ports (set, reset). It is the time available between the asynchronous signals going inactive and the active clock edge (clock to output **(t_{co})**).

Removal time is like hold time for asynchronous ports (set, reset). It is the time between active clock edge and asynchronous signal going inactive.

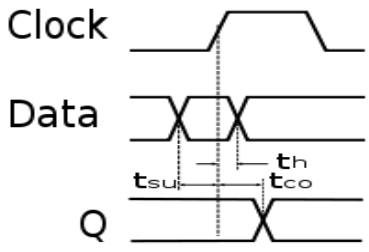


Figure 4-17: Flip-flop timing parameters

5 FSM-based Design of Digital Control Systems

5.1 Introduction to Finite State Machines

Up to now, every circuit that was presented was a combinatorial circuit. That means that its output is dependent only by its current inputs. Previous inputs for that type of circuits have no effect on the output. Multiplexers, decoders, code converters and adders cannot perform operations sequentially; i. e. an adder cannot count sequentially without changing the inputs after each addition. However, there are many applications where there is a need for our circuits to have "memory"; to remember previous inputs and calculate their outputs according to them. A circuit whose output depends not only on the present input but also on the history of the input is called a sequential circuit.

Combinational logic devices lack memory elements. They store results from a preceding state of operation and support it as an input to the following operation. With storage elements an adder becomes to an accumulator. Memory elements are indispensable in a sequential circuit.

Such circuits are called sequential machines or state machines. They possess memory and can perform time dependent sequences of logic signals controlled by present and past input information. Most sequential machines are synchronous because the data processing from a present state to the next state is controlled by a system clock. Usually, the number of states is limited according to their dedicated purpose, so they are called finite state machines (FSM). FSMs are used in many digital systems to control the behavior of systems and dataflow paths. Examples of FSM include control units and sequencers

There are two types of state machines as classified by the types of outputs generated from each. The first is the Moore State Machine where the outputs are only a function of the present state, the second is the Mealy State Machine where one or more of the outputs are a function of the present state and one or more of the inputs (see Figure 5-1).

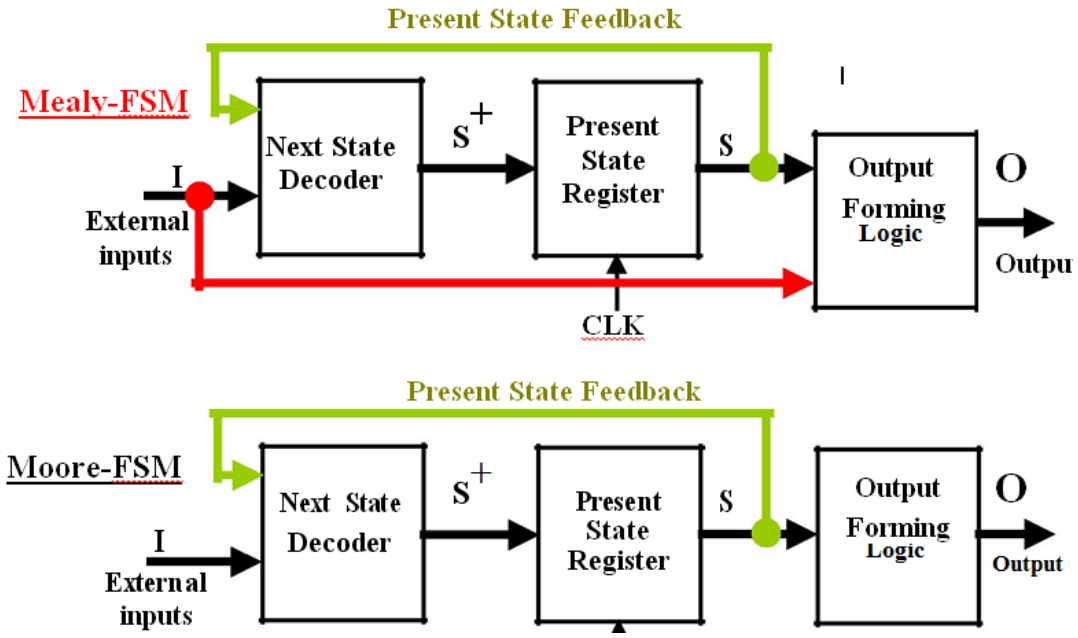


Figure 5-1: Mealy and Moore Finite State Machines

5.2 Design of Finite State Machines

The state machine design consists of:

- Identifying the count sequence to undertake.
- Identifying the type of bi-stable to use.
- Identifying the available logic gates to use in the combinational logic.
- Determining the combinational logic.

The state machine design will be viewed in the following representations:

- As a logic diagram;
- As a state transition table (the same appearance as a combinational logic truth table);
- As a state transition diagram.

In this section we will learn how to design and build such sequential circuits. In order to see how this procedure works, we will use an example, on which we will study this topic.

Let's suppose we have a digital quiz game that works on a clock and reads an input from a manual button. However, we want the switch to transmit only one HIGH pulse to the circuit. If we hook the button directly on the game circuit it will transmit HIGH for as few clock cycles as our finger can achieve. On a common clock frequency our finger can never be fast enough.

The design procedure has specific steps that must be followed in order to get the work done:

Step 1

The first step of the design procedure is to define with simple but clear words what we want our circuit to do:

"Our mission is to design a secondary circuit that will transmit a HIGH pulse with duration of only one cycle when the manual button is pressed, and won't transmit another pulse until the button is depressed and pressed again."

Step 2

The next step is to design a State Diagram. This is a diagram that is made from circles and arrows and describes visually the operation of our circuit. In mathematic terms, this diagram that describes the operation of our sequential circuit is a Finite State Machine.

Make a note that this is a Moore Finite State Machine. Its output is a function of only its current state, not its input. That is in contrast with the Mealy Finite State Machine, where input affects the output. In this tutorial, only the Moore Finite State Machine will be examined.

The State Diagram of our circuit is shown in Figure 5-2.

Every circle represents a "state", a well-defined condition that our machine can be found at.

In the upper half of the circle we describe that condition. The description helps us remember what our circuit is supposed to do at that condition.

- The first circle is the "stand-by" condition. This is where our circuit starts from and where it waits for another button press.
- The second circle is the condition where the button has just been just pressed and our circuit needs to transmit a HIGH pulse.
- The third circle is the condition where our circuit waits for the button to be released before it returns to the "stand-by" condition.

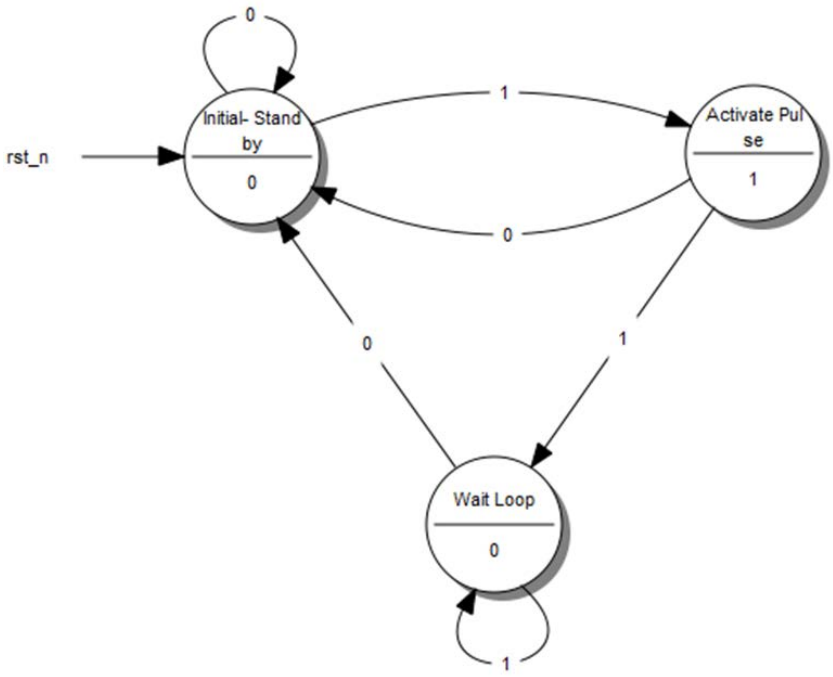


Figure 5-2: State Transition Diagram

In the lower part of the circle is the output of our circuit. If we want our circuit to transmit a High on a specific state, we put a 1 on that state. Otherwise we put a 0.

Every arrow represents a "transition" from one state to another. A transition happens once every clock cycle. Depending on the current Input, we may go to a different state each time. Notice the number in the middle of every arrow. This is the current Input.

For example, when we are in the "Initial-Stan by" state and we "read" a 1, the diagram tells us that we have to go to the "Activate Pulse" state. If we read a 0 we must stay on the "Initial-Stand by" state.

So, what does our "Machine" do exactly? It starts from the "Initial - Stand by" state and waits until a 1 is read at the Input. Then it goes to the "Activate Pulse" state and transmits a HIGH pulse on its output. If the button keeps being pressed, the circuit goes to the third state, the "Wait Loop". There it waits until the button is released (Input goes 0) while transmitting a LOW on the output. Then it's all over again!

This is possibly the most difficult part of the design procedure, because it cannot be described by simple steps. It takes experience and a bit of sharp thinking in order to set up a State Diagram, but the rest is just a set of predetermined steps.

Step 3

Next, we replace the words that describe the different states of the diagram with binary numbers. We start the enumeration from 0 which is assigned on the initial

state. We then continue the enumeration with any state we like, until all states have their number.

The result looks something like this (see Figure 5-3):

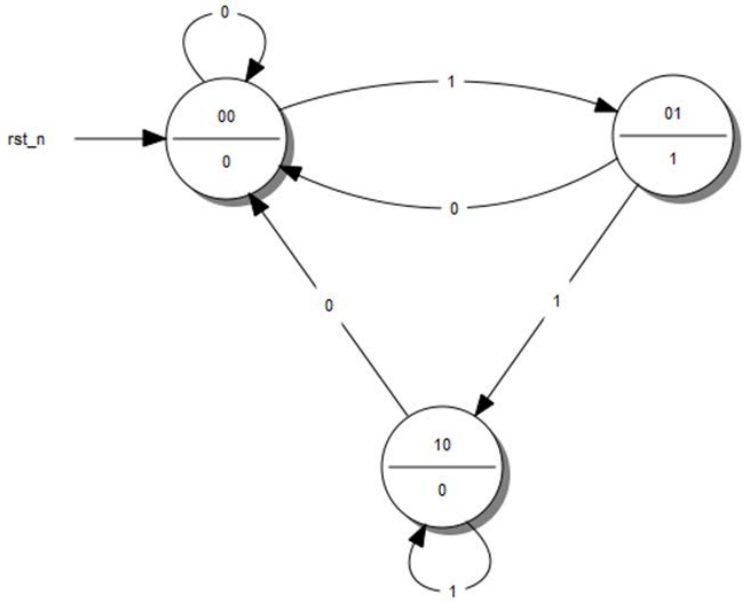


Figure 5-3: State Transition Diagram with Coded States

Step 4

Afterwards, we fill the State **Transition** Table (Figure 5-4). This table has a very specific form. I will give the table of our example and use it to explain how to fill it in.

Current State		Input	Next State		Outputs
A	B	I	A_{next}	B_{next}	Y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	1	1	0	0
1	1	0	X	X	X
1	1	1	X	X	X

Figure 5-4: State Transition Table

The first columns are as many as the bits of the highest number we assigned the State Diagram. If we had 5 states, we would have used up to the number 100, which means we would use 3 columns. For our example, we used up to the number 10, so only 2 columns will be needed. These columns describe the Current State of our circuit.

To the right of the Current State columns we write the Input Columns. These will be as many as our Input variables. Our example has only one Input.

Next, we write the Next State Columns. These are as many as the Current State columns.

Finally, we write the Outputs Columns. These are as many as our outputs. Our example has only one output. Since we have built a More Finite State Machine, the output is dependent on only the current input states. This is the reason the outputs column has two 1: to result in an output Boolean function that is independent of input I. Keep on reading for further details.

The Current State and Input columns are the Inputs of our table. We fill them in with all the **binary numbers**

- **from 0 to $2^{(\text{Number of Current State columns + Number of Input columns})} - 1$**.

It is simpler than it sounds fortunately. Usually there will be more rows than the actual States we have created in the State Diagram, but that's ok.

Each row of the Next State columns is filled as follows: We fill it in with the state that we reach when, in the State Diagram, from the Current State of the same row we follow the Input of the same row. If have to fill in a row whose Current State number doesn't correspond to any actual State in the State Diagram we fill it with Don't Care terms (X). After all, we don't care where we can go from a State that doesn't exist. We wouldn't be there in the first place! Again it is simpler than it sounds.

The outputs column is filled by the output of the corresponding Current State in the State Diagram. The State Table is complete! It describes the behavior of our circuit as fully as the State Diagram does.

Step 5a

The next step is to take that theoretical "Machine" and implement it in a circuit. Most often than not, this implementation involves Flip Flops. This guide is dedicated to this kind of implementation and will describe the procedure for both D - Flip Flops as well as JK - Flip Flops. T - Flip Flops will not be included as they are too similar to the two previous cases.

The selection of the Flip Flop to use is arbitrary and usually is determined by cost factors. The best choice is to perform both analysis and decide which type of Flip Flop results in minimum number of logic gates and lesser cost.

First we will examine how we implement our "Machine" with D-Flip Flops.

We will need as many D - Flip Flops as the State columns, 2 in our example. For every Flip Flop we will add one more column in our State Table (Figure 5-5) with the name of the Flip Flop's input, "D" for this case. The column that corresponds

to each Flip Flop describes what input we must give the Flip Flop in order to go from the Current State to the Next State. For the D - Flip Flop this is easy: The necessary input is equal to the Next State. In the rows that contain X's we fill X's in this column as well.

Current State		Input	Next State		Outputs	Flip Flop Inputs	
A	B	I	A_{next}	B_{next}	Y	D_A	D_B
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	1
0	1	0	0	0	1	0	0
0	1	1	1	0	1	1	0
1	0	0	0	0	0	0	0
1	0	1	1	0	0	1	0
1	1	0	X	X	X	X	X
1	1	1	X	X	X	X	X

Figure 5-5: State Table with D - Flip Flop Excitations

Step 5b

We can do the same steps with JK - Flip Flops. There are some differences however. A JK - Flip Flop has two inputs, therefore we need to add two columns for each Flip Flop. The content of each cell is dictated by the JK's excitation table (Figure 5-6 and Figure 5-7).

Q	Q_{next}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Figure 5-6: JK - Flip Flop Excitation Table

This table says that if we want to go from State Q to State Qnext, we need to use the specific input for each terminal. For example, to go from 0 to 1, we need to feed J with 1 and we don't care which input we feed to terminal K.

Current State		Input	Next State		Outputs	Flip Flop Inputs			
A	B	I	A_{next}	B_{next}	Y	J_A	K_A	J_B	K_B
0	0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	0	X	1	X
0	1	0	0	0	1	0	X	X	1
0	1	1	1	0	1	1	X	X	1
1	0	0	0	0	0	X	1	0	X
1	0	1	1	0	0	X	0	0	X
1	1	0	X	X	X	X	X	X	X
1	1	1	X	X	X	X	X	X	X

Figure 5-7: State Table with JK - Flip Flop Excitations

Step 6

We are in the final stage of our procedure. What remains, is to determine the Boolean functions that produce the inputs of our Flip Flops and the Output. We will extract one Boolean function for each Flip Flop input we have. This can be done with a Karnaugh map (Figure 5-8).

D_A

A \ BI	00	01	11	10
0	0	0	1	0
1	0	1	X	X

D_B

A \ BI	00	01	11	10
0	0	1	0	0
1	0	0	X	X

Figure 5-8: Karnaugh maps for the D - Flip Flop Inputs

The input variables of this map are the Current State variables as well as the Inputs.

That said, the input functions for our D - Flip Flops are the following:

$$D_A = A \cdot I + B \cdot I = (A+B) \cdot I$$
$$D_B = \overline{A} \cdot \overline{B} \cdot I$$

If we chose to use JK - Flip Flops our functions would be the following: (Figure 5-9).

J_A

A \ BI	00	01	11	10
0	0	0	1	0
1	X	X	X	X

K_A

A \ BI	00	01	11	10
0	X	X	X	X
1	1	0	X	X

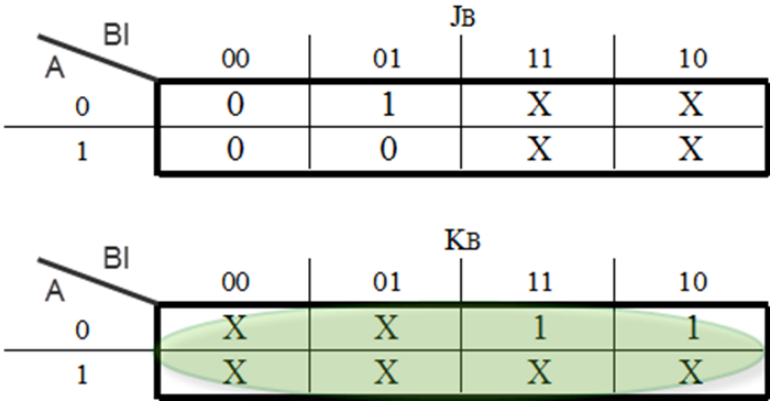


Figure 5-9: Karnaugh maps for the JK - Flip Flop Input

$$J_A = B \cdot I$$
$$K_A = \overline{I}$$
$$J_B = \overline{A} \cdot I$$
$$K_B = 1$$

A Karnaugh map will be used to determine the function of the Output as well: (Figure 5-10)

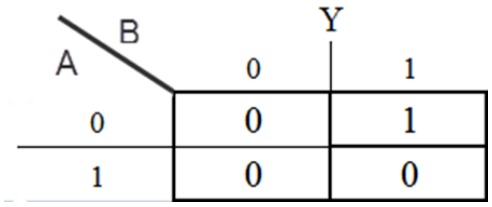


Figure 5-10: Karnaugh map for the Output variable Y

$$Y = \overline{A} \cdot B$$

Step 7

After designing our Finite State Machine it realized in the form of the sequential circuits. We place the Flip Flops and use logic gates to form the Boolean functions that we calculated. The gates take input from the output of the Flip Flops and the Input of the circuit. It is necessary to connect the clock to the Flip Flops.

The D - Flip Flop version is shown in Figure 5-11.

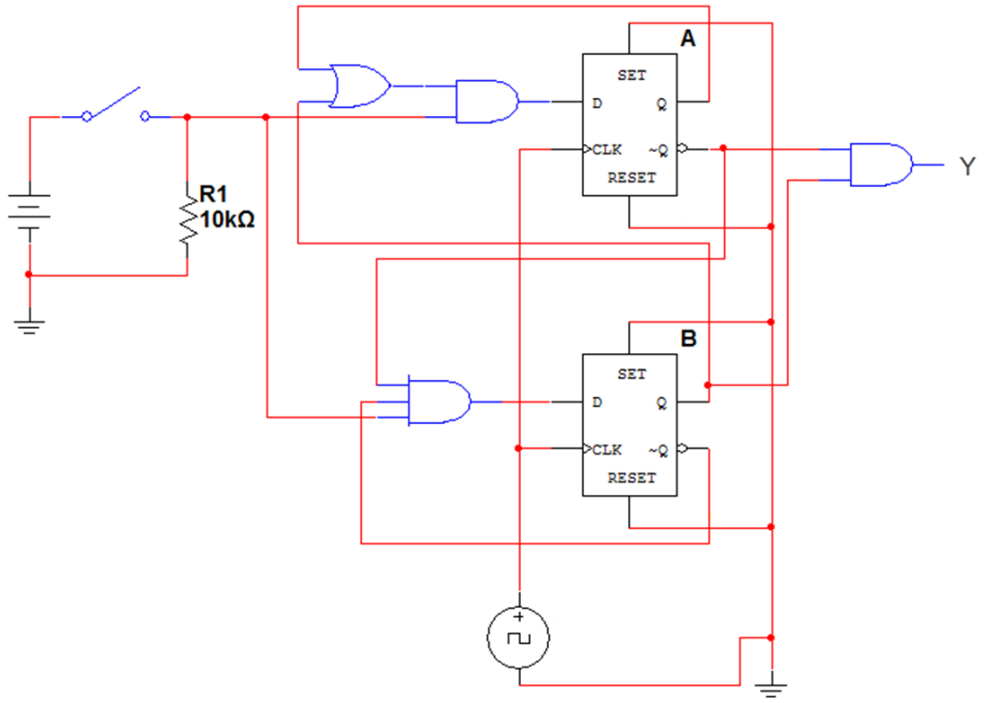


Figure 5-11: Completed D - Flip Flop Sequential Circuit

The JK - Flip Flop version is shown in Figure 5-12.

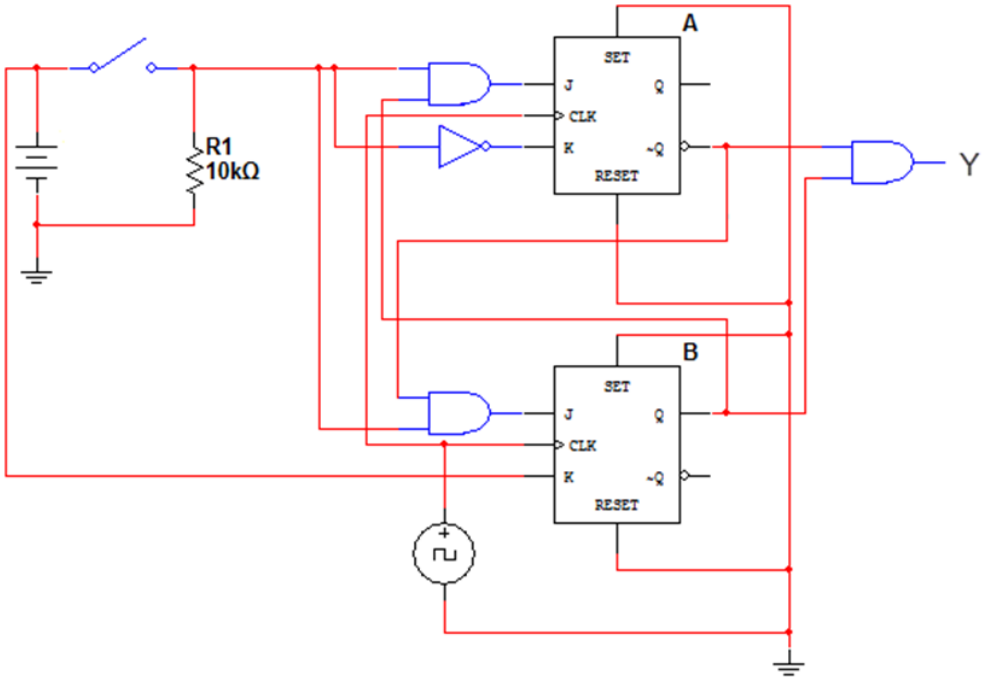


Figure 5-12: Completed JK - Flip Flop Sequential Circuit

We have successfully designed and constructed a Sequential Circuit. At first it might seem a daunting task, but after practice and repetition the procedure will become trivial. Sequential Circuits can come in handy as control parts of bigger

circuits and can perform any sequential logic task that we can think of. The sky is the limit! (or the circuit board, at least)

Conclusion

A Sequential Logic function has a "memory" feature and takes into account past inputs in order to decide on the output.
The Finite State Machine is an abstract mathematical model of a sequential logic function. It has finite inputs, outputs and number of states.
FSMs are implemented in real-life circuits through the use of Flip Flops
The implementation procedure needs a specific order of steps (algorithm), in order to be carried out.

6 Exercises

6.1 Control Questions

Obligatory tasks

Answer the control Questions for each chapter!.

Chapter 1

What is a Programmable Logic Device?

What kinds of programmable logic devices are available?

Describe differences between types of PLDs!

In which areas PLD are applied?

Chapter 2

What is a Boolean function?

What are the Basic Boolean Laws?

How a Boolean function can be described or represented?

What is a truth table?

In what ways a Boolean function can be minimized?

Chapter 3

What is a combinational circuit?

What is a truth table?

What are a basic combinational logic gates?

What is an adder?

Chapter 4

What is a flip flop?

What types of a flip flops are there and how do they differ from one to another?

Of which elements flip flops consist?

_____

Where flip flops can be used?

What time characteristics have flip flops?

Chapter 5

What is a Finite State Machine?

What distinguishes Mealy State Machine from Moore State Machine?

Of which steps consists the design flow of a state machine?

How state machines can be represented?

6.2 Multiple Choice Quiz

Obligatory tasks
Go to the web-site and fulfill the multiple choice test!
If you are offline, prepare a list of Question number and solution!

(Interactive Quiz - http://www.ee.surrey.ac.uk/Projects/Labview/gatesfunc/index.html#quiz)

This is a multiple choice quiz. You need to click the appropriate radio button after each question to make your choice.

1. What does an EXOR gate do?

a) Give a high output when one or more of its inputs are high

b) Give a high output when only one of its inputs is high

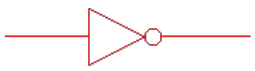
c) Give a low output when one or more of its inputs are high

d) Give a low output when only one of its inputs is high

2. Which of the following symbols represents a NOR gate?

a)

b)



c)

d)

3. Which one of the following truth tables represents the behavior a NAND gate?

a)

2 Input NAND gate		
A	B	$\overline{A.B}$
0	0	1
0	1	0
1	0	0
1	1	0

b)

2 Input NAND gate		
A	B	$\overline{A.B}$
0	0	0
0	1	1
1	0	1
1	1	0

4. What does connecting the inputs of NAND and NOR gates do?

a) Help produce multi-input gates

b) Produce and EXNOR gate

c) Produce a NOT gate

d) Damage the gates

5. How do you make a NAND gate out of an AND gate by using inverters (NOT gates)?

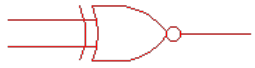
a) Invert the output from the AND gate

b) Invert both the inputs to the AND gate

c) Invert one of the inputs to the AND gate

d) Invert both the inputs and the output of the AND gate

6. What type of logic gate does this symbol represent?



a) Exclusive OR

b) Exclusive NOR

c) OR

d) NOR

---

7. How do you make a NOR gate out of an NAND gate by using inverters (NOT gates)?

a) Invert the output from the NAND gate

b) Invert both the inputs to the NAND gate

c) Invert one of the inputs to the NAND gate

d) Invert both the inputs and output of the NAND gate

8. What type of logic gate's behavior does this truth table represent?

?			
A	B	C	?
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

a) 2 input OR

b) 3 input OR

c) 3 input EXOR

d) 4 input EXOR

9. What type of logic gate does this logic circuit configuration produce?

a) NAND gate

b) NOR gate

c) EXOR gate

d) EXNOR gate

10. How do you make an NAND gate out of an OR gate using inverters (NOT gates)?

a) Invert the output from the OR gate

b) Invert both inputs to the OR gate

c) Invert one of the inputs to the OR gate

d) Invert both the inputs and output of the OR gate

11. Which of the following expressions is in the product-of-sums form?

a. $(A + B)(C + D)$

b. $(AB)(CD)$

c. $AB(CD)$

d. $AB + CD$

12. Which of the following is an important feature of the sum-of-products form of expressions?

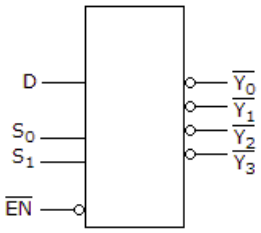
a. All logic circuits are reduced to nothing more than simple AND and OR operations.

b. The delay times are greatly reduced over other forms.

c. No signal must pass through more than two gates, not including inverters.

d. The maximum number of gates that any signal must pass through is reduced by a factor of two.

13. For the device shown here, assume the D input is LOW, both S inputs are LOW, and the $\overline{EN}$ input is LOW. What is the status of the $\overline{Y}$ outputs?



a. All are HIGH.

b. All are LOW.

c. All but $\overline{Y_0}$ are LOW.

d. All but $\overline{Y_0}$ are HIGH.

14. An output gate is connected to four input gates; the circuit does not function. Preliminary tests with the DMM indicate that the power is applied; scope tests show that the primary input gate has a pulsing signal, while the interconnecting node has no signal. The four load gates are all on different ICs. Which instrument will best help isolate the problem?

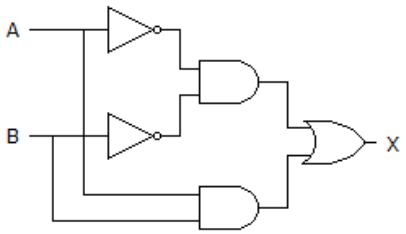
    a. Current tracer

    b. Logic probe

    c. Oscilloscope

    d. Logic analyzer

15. Which of the following logic expressions represents the logic diagram shown?



    a. $X = A\,\overline{B} + \overline{A}\,B$

    b. $X = \overline{\overline{A}\,\overline{B}} + A\,B$

    c. $X = \overline{A\,B} + \overline{A}\,\overline{B}$

    d. $X = \overline{A}\,\overline{B} + A\,B$

16. Which of the K-maps given below represents the expression $X = AC + BC + B$?

	$\overline{C}$	C
$\overline{A}\,\overline{B}$	1	1
$\overline{A}\,B$	1	1
$A\,B$	0	0
$A\,\overline{B}$	0	0

a.

	$\overline{C}$	C
$\overline{A}\,\overline{B}$	0	1
$\overline{A}\,B$	0	0
$A\,B$	1	1
$A\,\overline{B}$	1	1

b.

	$\overline{C}$	C
$\overline{A}\,\overline{B}$	0	0
$\overline{A}\,B$	1	1
$A\,B$	1	1
$A\,\overline{B}$	0	1

c.

	$\overline{C}$	C
$\overline{A}\,\overline{B}$	1	1
$\overline{A}\,B$	0	1
$A\,B$	0	1
$A\,\overline{B}$	1	1

d.

17. A decoder can be used as a demultiplexer by:

a. tying all enable pins LOW

b. tying all data-select lines LOW

c. tying all data-select lines HIGH

d. using the input lines for data selection and an enable line for data input

18 How many 4-bit parallel adders would be required to add two binary numbers each representing decimal numbers up through 300_{10}?

a. 1 **b.** 2

c. 3 **d.** 4

19. Which statement below best describes a Karnaugh map?

a. A Karnaugh map can be used to replace Boolean rules.

b. The Karnaugh map eliminates the need for using NAND and NOR gates.

c. Variable complements can be eliminated by using Karnaugh maps.

d. Karnaugh maps provide a visual approach to simplifying Boolean expressions.

6.3 Examples

Additional information

- Give here additional information.

Combinational logic design using Karnaugh Maps: Karnaugh maps (K-maps) provide a means to display the combinational logic circuit by the use of a map. This map is a grid (a 4 x 2 grid for a 3-input logic circuit and a 4 x 4 grid for a 4-input logic circuit). From this map, Boolean minimization can be undertaken and a minimal expression obtained. The K-maps for a 3-input and 4-input logic circuit are shown in Figure 6-1.

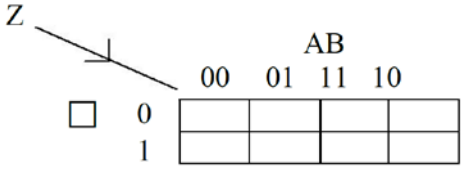


Figure 6-1: 3-input K-map

The inputs are A, B and C: The output is Z.

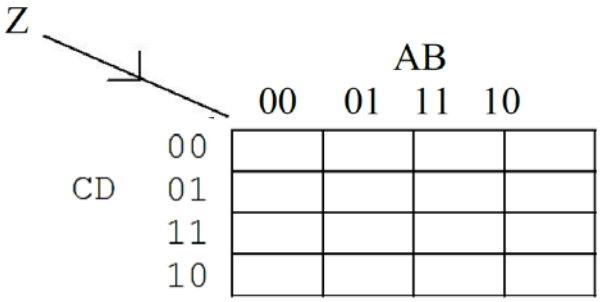


Figure 6-2: 4-input K-map

The inputs are A, B, C and D: The output is Z.

The logic value (0 or 1) for the output is placed in the corresponding square of the input logic values. Logic 1's are grouped to form the largest possible group:

• For the 3-input Karnaugh Map, this is 8, then 4, then 2, then 1.

• For the 4-input Karnaugh Map, this is 16, then 8, then 4, then 2, then 1.

Groups of 16, 8, 4 and 2 can be simplified. A group of 1 cannot be simplified. The groups must be adjacent so that the group forms a square or rectangle. The edges of the Karnaugh map (top/bottom and left/right) are considered to be adjacent. Loops may also overlap.

For a 4-input Karnaugh Map, the idea is shown in Figure 6-3.

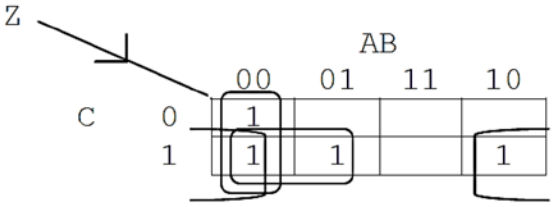


Figure 6-3: 3-input Karnaugh Map example

Here, there are three loops of two. The logic expression is derived from this by considering each loop in turn and AND-ing the input values together (when the input is a 1, the input term is used: when the input is a 0, the inverse (NOT) of the input is used). Each AND term is OR-ed together. If in a loop, a value can be either 0 or 1, it is not needed and so removed from the expression. For example, in Figure 6-3, the resulting Boolean expression is:

$$Z = (\overline{A} \cdot \overline{B}) + (\overline{A} \cdot C) + (\overline{B} \cdot C) \qquad (3.1)$$

The truth-table representation of the logic function in the Karnaugh Map is shown in Figure 6-4.

A	B	C	Z
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Figure 6-4: 3-input K-map example

For a 4-input K-map, the idea is shown in Figure 6-5.

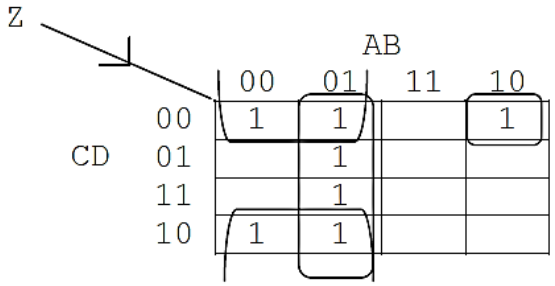


Figure 6-5: 4-input K-map example

Here, there are two loops of four and one loop of one.

The logic expression is derived from this by considering each loop in turn and AND-ing the input values together (when the input is a 1, the input term is used: when the input is a 0, the inverse (NOT) of the input is used). Each AND term is OR-ed together. If in a loop, a value can be either 0 or 1, it is not needed and so removed from the expression.

For example, in Figure 6-5, the resulting Boolean expression is:

$$Z = \left(\overline{A} \cdot B\right) + \left(\overline{A} \cdot \overline{D}\right) + \left(A \cdot \overline{B} \cdot \overline{C} \cdot \overline{D}\right) \quad (3.2)$$

The truth-table representation of the logic function in the Karnaugh Map is shown in Figure 6-6.

A	B	C	D	Z
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Figure 6-6: 4-input Karnaugh Map example

6.4 Laboratory Work

Obligatory tasks

Prepare the documents for the experiments!
Laboratory work includes implementation of following experiments:

Experiment 1: « Create a parallel machine for the physical system «3-Floor Elevator »» in GOLDI environment.

To implement the experiment 1 students have to use the **example of a parallel machine for the physical system «3-Axis-Portal»** developed at Ilmenau University of Technology (http://goldi-labs.net/index.php?Site=41).

Experiment 2: «Quartus II Project for «Production cell» model»

To implement this experiment on GOLDI students should prepare the Quartus-project for selected physical system (**«Productuon cell» model**) by using the example-projects for the physical system «3AxisPortal» developed at Ilmenau University of Technology (http://goldi-labs.net/index.php?Site=40**)**

6.5 Practical Work

Obligatory tasks

During the practical work students are required to solve tasks on sections «Boolean functions and methods of their minimization" and "Combinational logic», to study laboratory GOLDI, gain practical experience in the construction of an automaton graph of electromechanical models of GOLDI using Graphical Interactive FSM Tool (GIFT)

7 Lab work

7.1 The Interactive Hybrid Online laboratory GOLDI

The Hybrid Online Lab **GOLDI** (**G**rid of **O**nline **L**ab **D**evices **I**lmenau) was developed at the Department of Integrated Communication Systems at the Ilmenau University of Technology. It provides a tool set, supporting all design steps for complex control tasks.

To check the functionality of the whole design, some special simulation and validation features are included as integral part of the GOLDI, such as:

- usage of simulation models of the physical system for visual prototyping;
- step by step and parallel execution of these prototypes;
- visualization of the simulation process with the tools also used for specification;
- features for test pattern generation and code generation for hardware and software synthesis.

GOLDI is used for teaching practical lessons as well as giving hands-on experiences for the development of embedded systems. This is done using remote lessons via the Internet and has the advantage that courses can be offered internationally world-wide. Additionally, even for local students, the lab offers extended opening hours. Besides the advantages for students, this also reduces the costs for academic teaching and improves the quality by offering more practical training possibilities.

GOLDI offers three different specification and control mechanisms to handle the physical systems (electro-mechanical models, e.g. **Elevator**, **Production Cell, 3-Axis-Portal**) in the lab room.

Web-Control Kit for a FSM oriented control: In this case, the student will use Finite State Machines as specification technique, based upon an automaton graph. By accessing the Web browser interface, he is able to enter his created control algorithm (in the form of Boolean equations), handle the laboratory procedure (initialize, start, stop and reset) and change environmental variables if necessary. The physical system will be controlled by the interpreter running inside the client (implemented e.g. as Java applet on the student's PC at home). In this case, only the input and output signals of the model will be transferred via Internet.

Web-based microprocessor control unit for a software oriented design: Students can implement their design task directly into a microcontroller for a software oriented specification. Therefore, they use common (non-commercial) development tools, (e. g. from Microchip or Atmel to develop C software projects). After compilation, the generated software control algorithm is transferred via GOLDI Web-interface to the remote lab, where the code is programmed into the microcontroller. Then, the student can begin with his experiment, to check if his algorithm fulfills the requirements of the given control task.

Web-based FPGA control unit for a hardware oriented design: If a student prefers an exclusive hardware oriented design, using a hardware description language like VHDL as specification technique, he can prepare his design task with common (non-commercial) development tools - e.g. from Altera or Xilinx. The generated bit file is uploaded via the GOLDI Web-interface to the remote lab, where an FPGA will be programmed. After programming the connected FPGA, the FPGA board operates as control unit for the designed control algorithm, and the student can start his experiment.

Figure 7-1 gives an overview about the GOLDI. The infrastructure is based on a universal grid concept which guaranties a reliable, flexible as well as robust usage of this online lab. The server side infrastructure (remote lab) consists of three parts:

- an internal serial ***remote lab bus*** to interconnect all parts of the remote lab, realized as controller area network (CAN) bus,
- a ***bus protection unit*** (BPU) to interface the control units to the remote lab bus and to protect the bus from blockage, misuse and damage as well as
- a ***physical system protection unit*** (PSPU), which protects the physical systems (the electro-mechanical models in the remote lab) against deliberate damage or accidentally wrong control commands and which offers different access and control mechanisms.

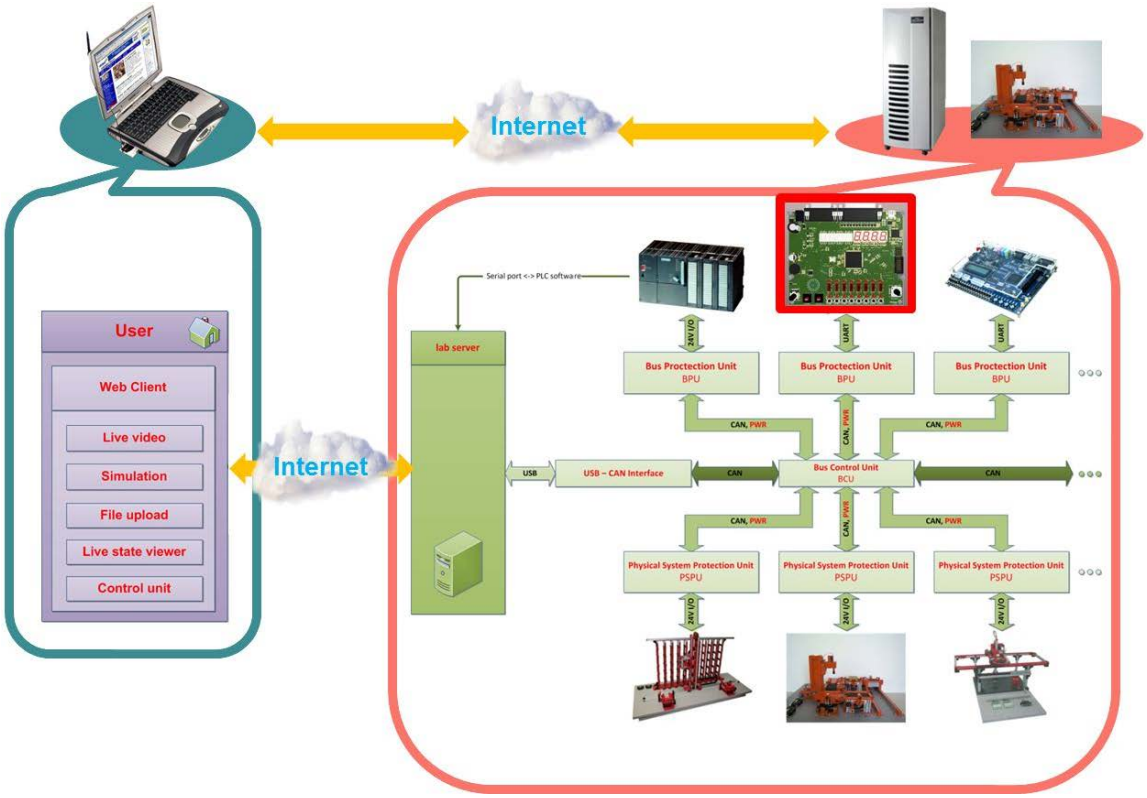


Figure 7-1: Overview of the Lab GOLDI infrastructure

The complete schematic view (client and server side) of the GOLDI architecture is shown in Figure 7-2.

Lab GOLDI is provided Web-based lab experiments in the field of digital system design including the basics of Boolean algebra, combinational logic and simple sequential circuits, various minimization techniques for logical expressions, dynamic effects in combinational and sequential circuits and the design of digital control systems based on Finite State Machines (FSM). Finally is offered different methods and tool concepts to create, implement and validate digital systems to solve complex design tasks.

The interconnection between the Web-control units and the selected physical systems during a remote lab experiment as well as the webcam handling is done by the lab server as part of the remote lab infrastructure. During a running experiment, the client application will interact directly with the online lab grid infrastructure. Based on this infrastructure are offered following operation modes:

- Stand-alone Mode (visual prototyping);
- Remote Control Mode (via Web-client);
- Remote Control Mode (via control unit);

- Virtual Control Mode (visual prototyping);
- Virtual Control Mode (test mode);

- Local Control Mode (via control unit);
- Local Control Mode (manually);

- Rapid Prototyping Mode;
- Visitor Mode.

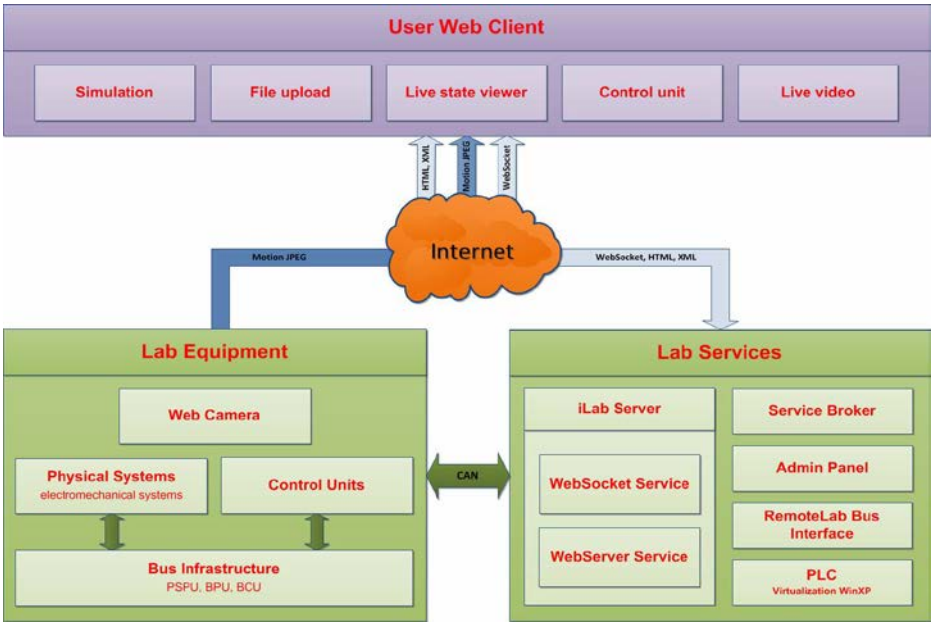


Figure 7-2: Schematic view of the remote lab components

7.2 Laboratory experiment 1: «Creating a parallel machine for the «3-Floor Elevator » model»

Introduction

To implement above mentioned experiment students have to use the **example of a parallel machine for the physical system** «**3-Axis-Portal**» developed at the Department of Integrated Communication Systems at the Ilmenau University of Technology (http://goldi-labs.net/index.php?Site=41).

Graphical Interactive FSM Tool: To design a finite state machine students have to use the graphical interactive FSM tool (GIFT). This tool can create a FSM in form of a state machine, a transition table, a machine table or z-equations. For more information about GIFT and the various forms of a FSM, you can read the user guide to GIFT.

Click here to open the Graphical Interactive FSM Tool (GIFT).

Example of a parallel machine for the physical system «3-Axis-Portal»

In this example we create a parallel machine, which describes the following behavior of the «3-Axis-Portal». The crane starts in the X-,Y-Position. First of all the crane drives to the right until it reaches the right position on the X-axis. Then the crane drives down and up again in Z-direction. Once the crane has finished its movement upwards it drives to the left until it reaches the left position on the X-axis.

We need to create two separate machines in GIFT, which we adapt to each other afterwards. The first machine controls the movement on the X-axis (left-right). With the help of the State Diagram and the Transition Table we create the machine and generate the corresponding z-/y-equations.

The input-variable "x3" is only a placeholder for the state Z0 ("up") from machine A1, which will be defined later. The other variables correspond to the sensors and actuators from the physical system «3AxisPortal».

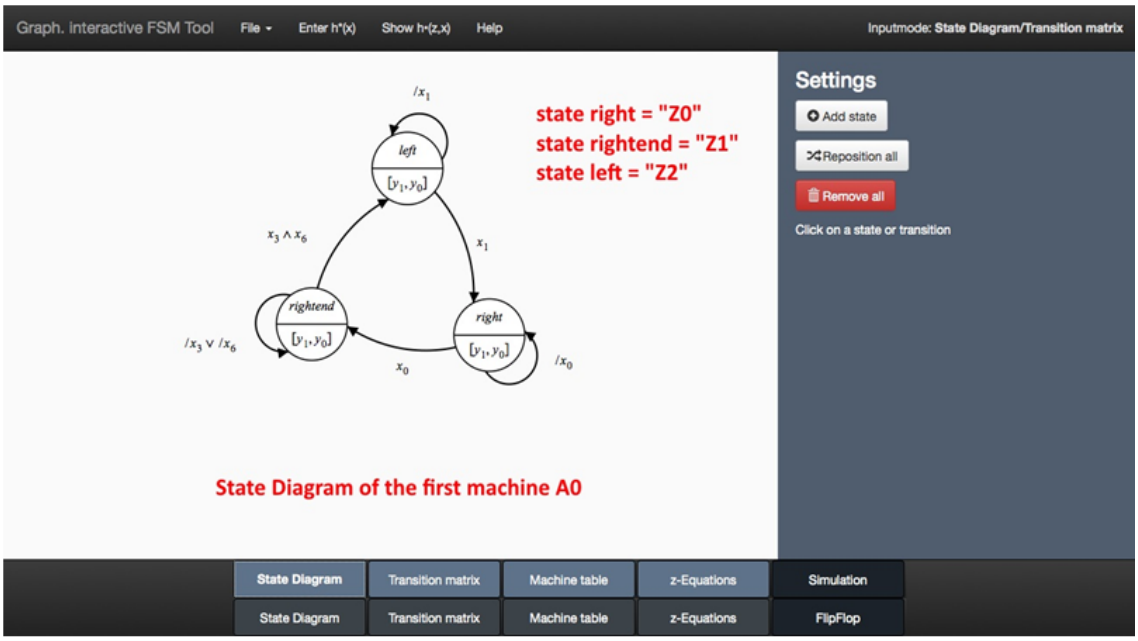


Figure 7-3: State Diagram of machine A0

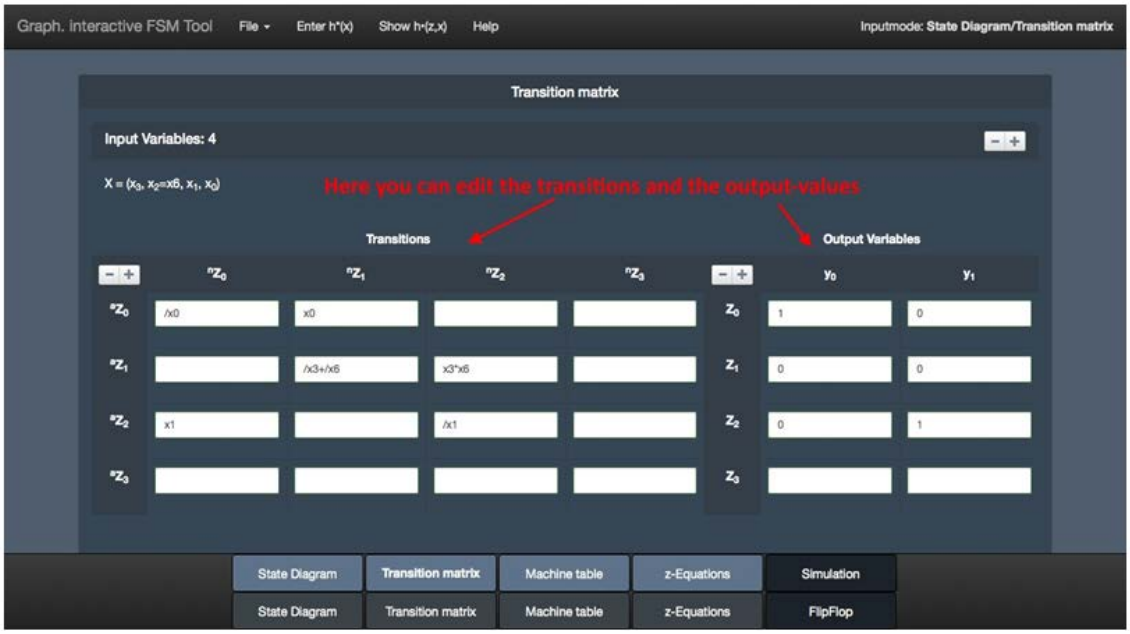


Figure 7-4: Transition Table of machine A0

The first machine A0 can be imported into the experiment with the *ECP Export* from GIFT. Therefore click on *ECP Export* in the File-menu in GIFT and click on the *Import from GIFT*-Button in the File-menu in the experiment. Then you have to name your machine and adapt the variables from GIFT to the physical system in a pop-up window.

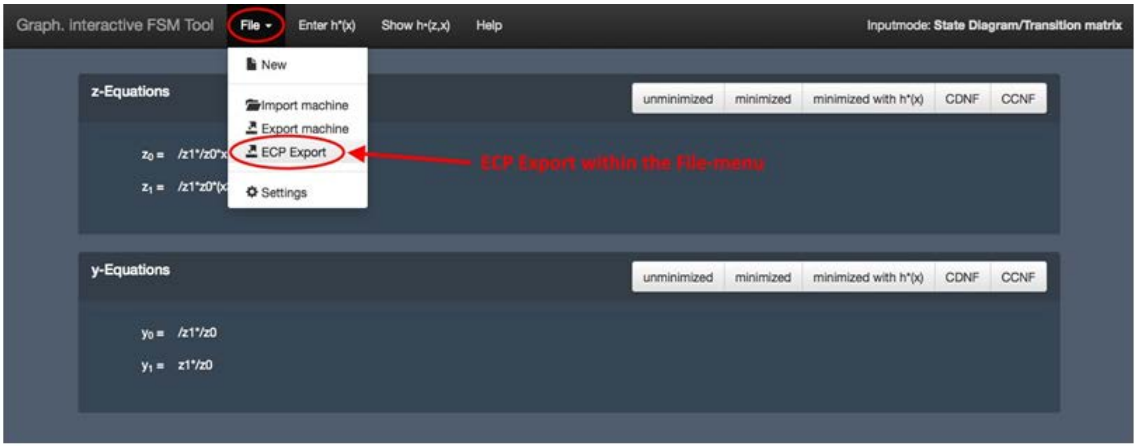


Figure 7-5: ECP-Export in GIFT

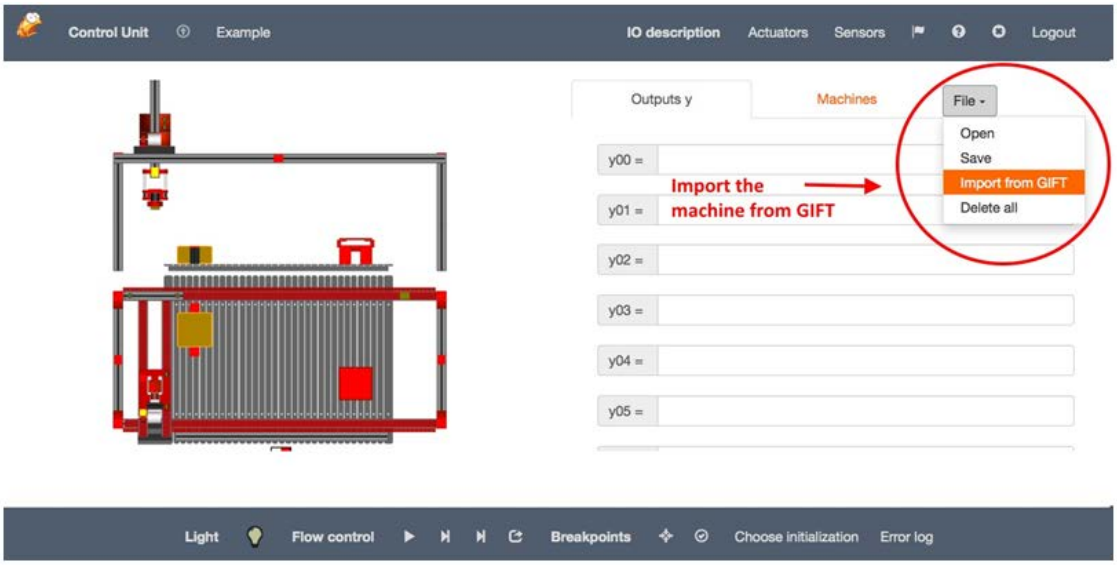


Figure 7-6: Import from GIFT in experiment

Name the first machine "A0" and assign the variables to the sensors and actuators from the system. In this example link the variables x6 to X6, x1 to X1, x0 to X0, y1 to Y1 and y0 to Y0. The variable x3 is still a placeholder, so you can assign it to any sensor of the system that you do not need (here: x3 to X9).

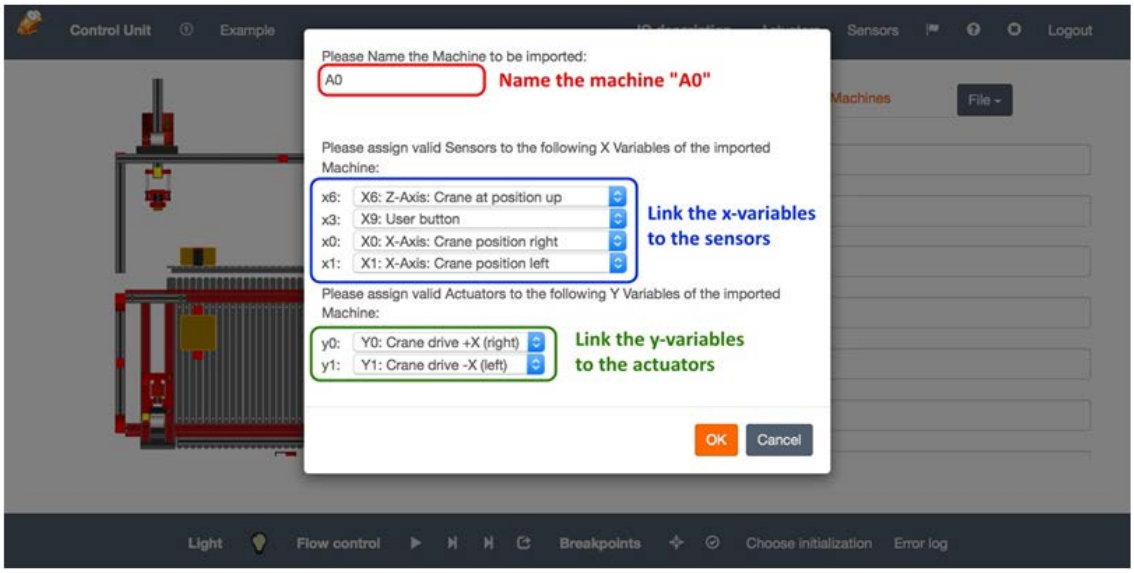


Figure 7-7: Settings for imported machine in the experiment

After importing the first machine A0 into the experiment, add a new machine A1 and change back again to GIFT to create the second machine A1 similar to the machine A0, which controls the movement on the Z-axis (down-up). This time you cannot import the machine via ECP into the experiment, so you have to copy the z-/y-equations from GIFT and insert them into the correct input field in the experiment.

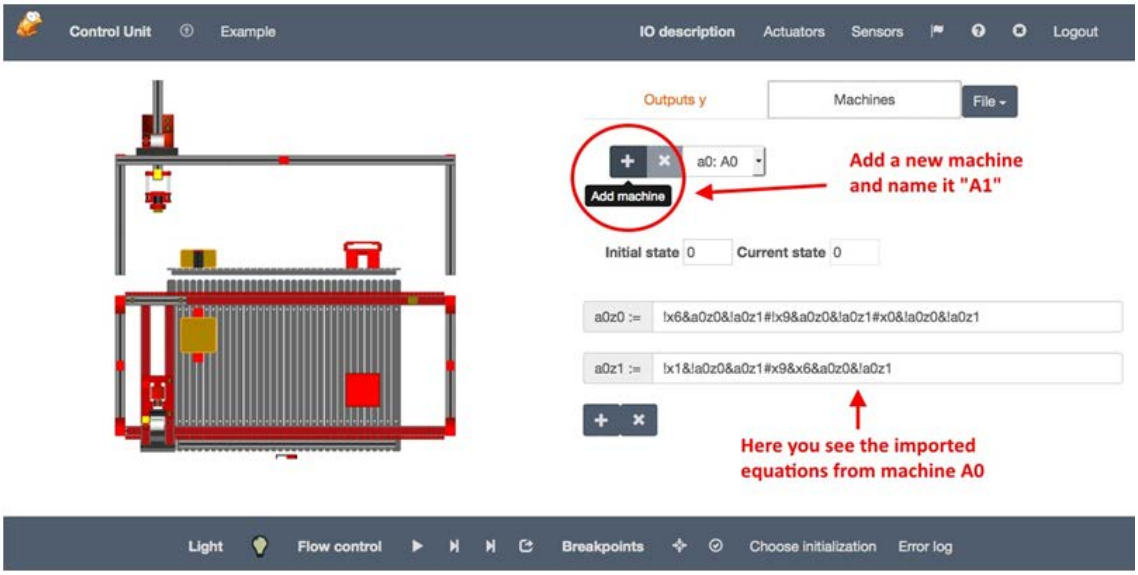


Figure 7-8: Add a new machine (A1)

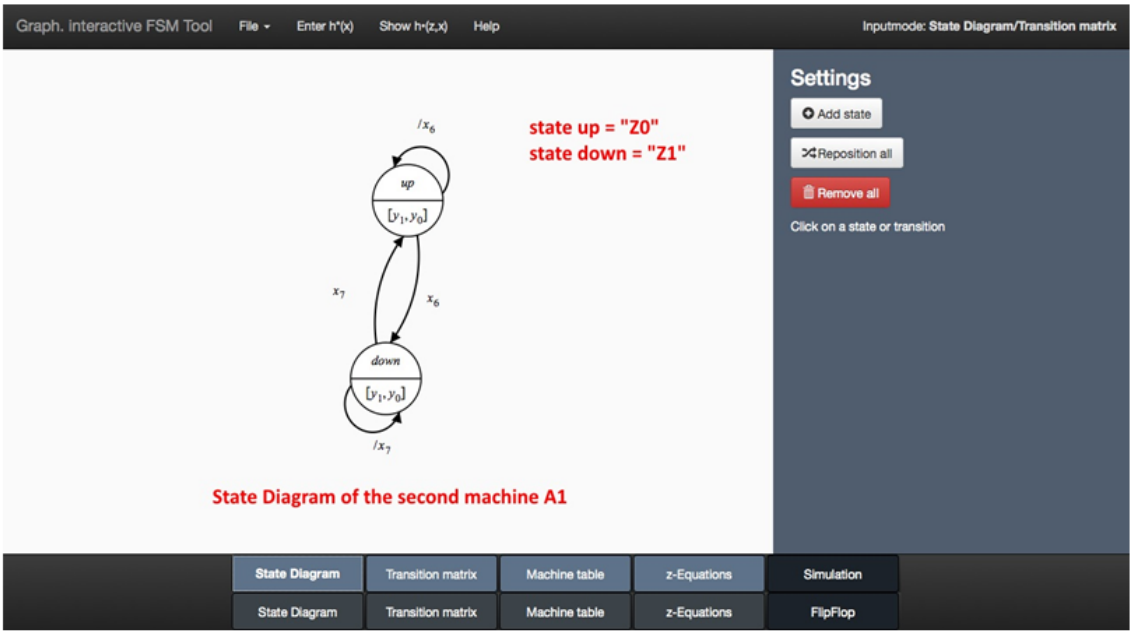


Figure 7-9: State Diagram of machine A1

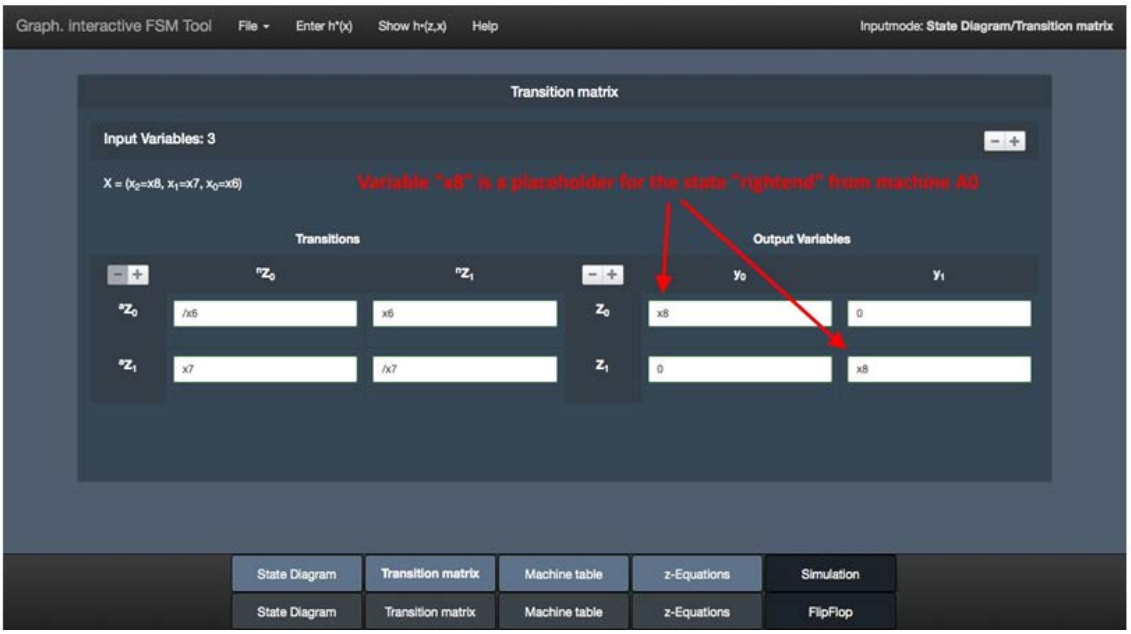


Figure 7-10: Transition Table of machine A1

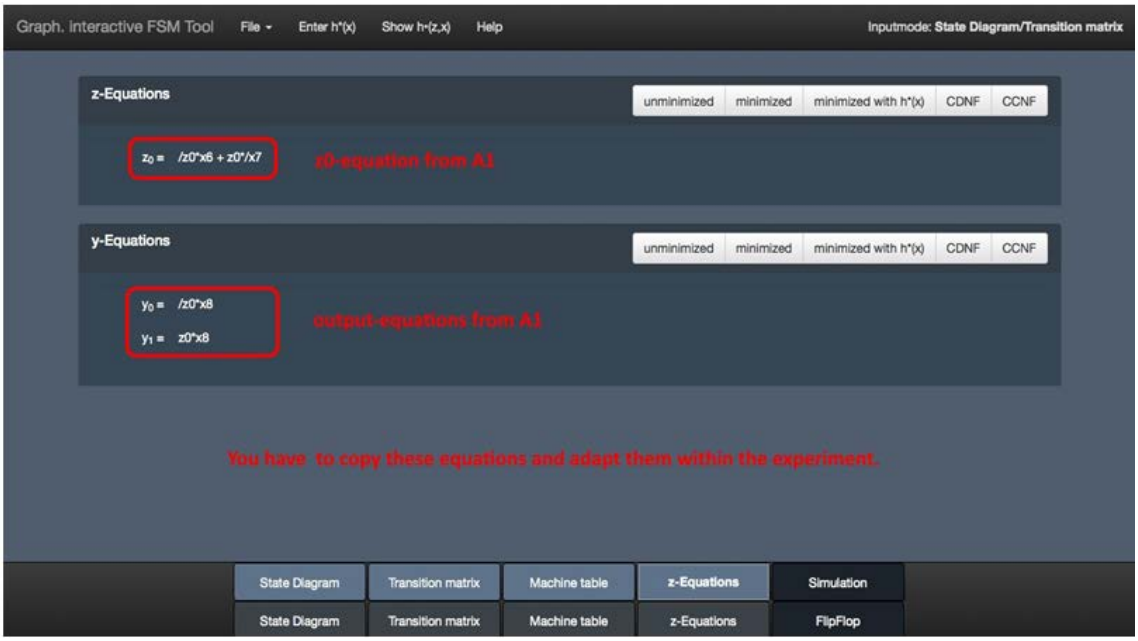


Figure 7-11: z-/y-equations of machine A1

Now you have to copy the z0-equation into the "a1z0"-field from machine A1, the y0-equation into the "y04"-field and the y1-equation into the "y05"-field from the "Outputs y" and adapt the equations. Change "z0" to "a1z0", "/" to "!", "*" to "&", "+" to "#" and "x8" to "!a0z1&a0z0". Furthermore you have to change the placeholder-variable "x9" from machine A0 to "!a1z0".

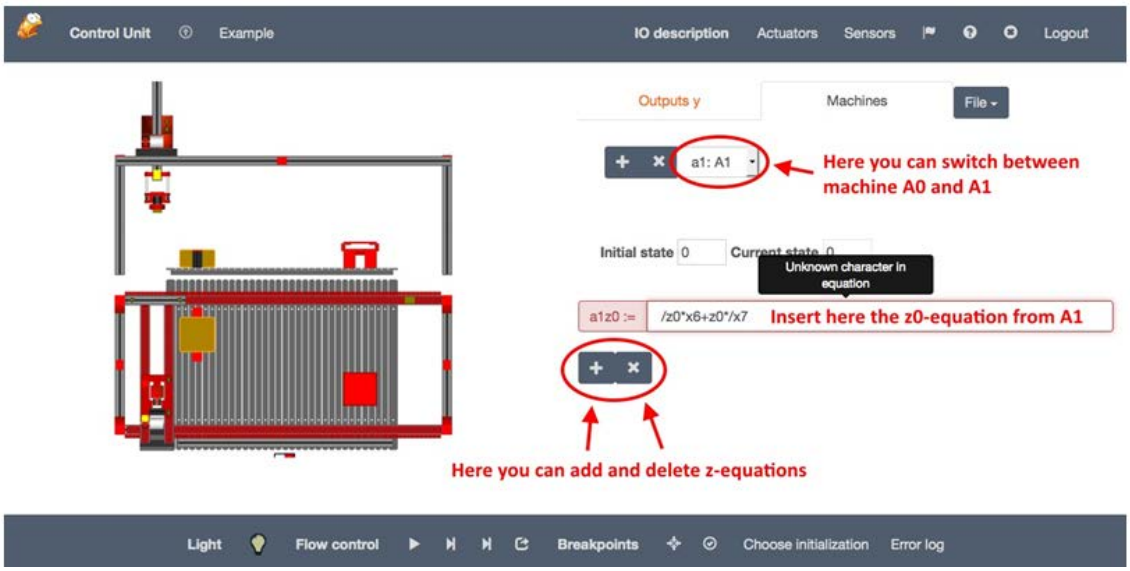


Figure 7-12: Copy the z0-equation from A1

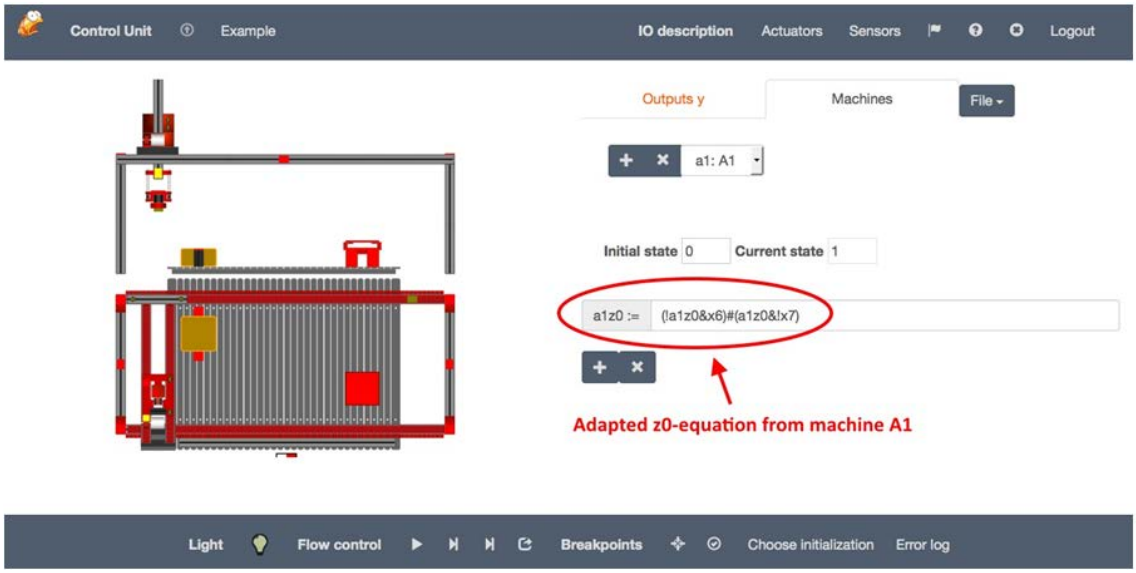


Figure 7-13: Adapt the z0-equation from A1

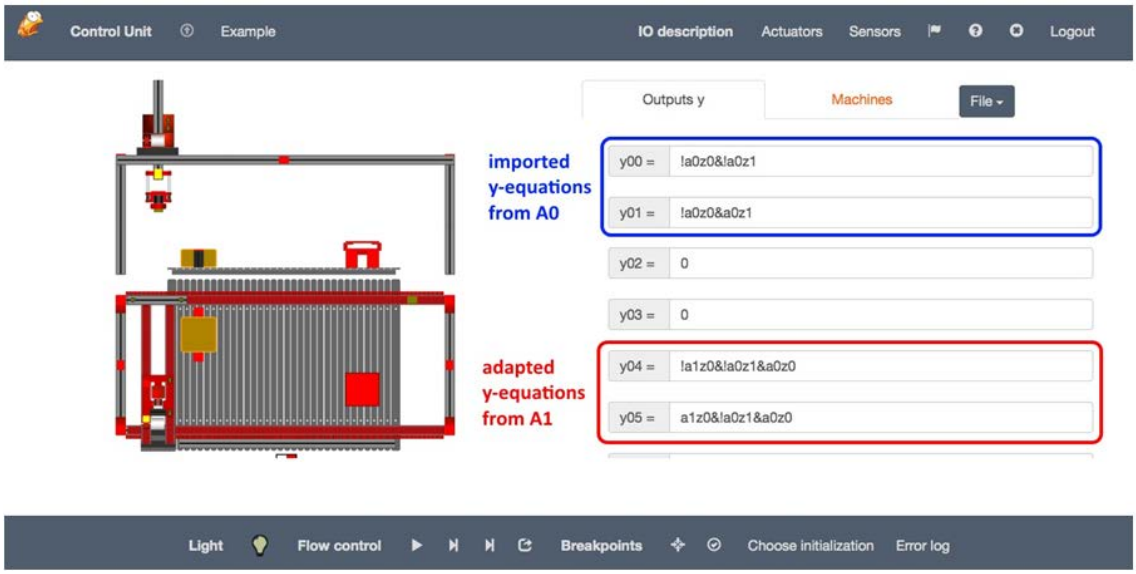


Figure 7-14: Copy and adapt the output-equations

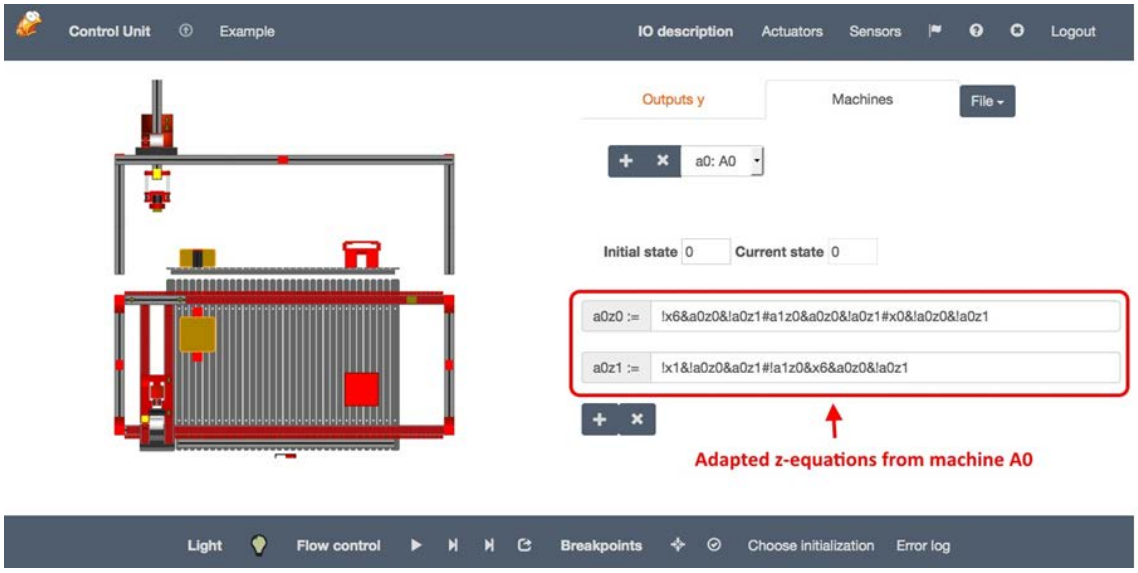


Figure 7-15: Adapted z-equations of machine A0

When you have created and adapted both machines A0 and A1 in the experiment, you can test the functionality of the parallel machine by clicking on the "Start"-button. If everything is correct, you can see the described movement of the "3-Axis-Portal" in the animation on the left.

7.3 Laboratory experiment 2: «Quartus-project for the «Production cell» model»

Introduction

To implement this experiment on GOLDi students should prepare the Quartus-project for selected physical system (**«Production cell» model**) by using the example-projects for the physical system «3AxisPortal» developed at the Department of Integrated Communication Systems at the IImenau University of Technology (http://goldi-labs.net/index.php?Site=40**).**

In this experiment students will use one of ALTERA's MAX V CPLDs **(MAX V 5M570ZT100C5)**. The architecture of these CPLDs integrates external functions such as flash, RAM, oscillators and phase-locked loops. Furthermore the CPLD we use delivers 570 logic elements and 100 pins for a particular package.

Quartus II: To adapt the files for programming the PLD in an experiment students need the software **Quartus II Web-Edition** from ALTERA. After creating your own user account at ALTERA's website, it is possible to download the free license version of Quartus II here:

http://dl.altera.com/?edition=web

Download the latest version of Quartus II (version 15.0). You can decide whether you want to download the Combined Files or the Individual Files. With the Individual Files you only have to download in Device the MAX II, MAX V device support, store all in the same directory. After the download is finished, all will be installed by starting the Quartus Setup.

If you want to get a detailed introduction how to install Quartus II and how to handle the software, you can follow these links:

Quartus II - An Introduction
Quartus II handbook
Altera Software Installation and Licensing

How to get files to start an experiment

To start this experiment with a PLD on GOLDi students should prepare the Quartus-project for selected physical system (**«Production cell» model**). They have the choice either to start with a textual design or with a graphical design in Quartus. Here students can download the example-projects for the physical system «3AxisPortal» («PortalCrane-Mealy»):

Textual Design: PortalCrane_Mealy
Graphical Design: PortalCrane_Mealy

These projects include all files you need to conduct an experiment. You only have to modify the UserDesign-files and the TopLevelDesign-files to adapt the downloaded project to your experiment. For more information read the following introduction to a Quartus design project on the basis of the «PortalCrane-Mealy» example.

7.4 Quartus project example «PortalCrane-Mealy»

In this example we use the graphical design project to demonstrate how to use Quartus II for GOLDi.

Step 1: Creating a project in Quartus II

- Download the rar-archive Graphical Design: PortalCrane_Mealy and extract the files. The created folder *"Example_TextualDesign"* is the project folder.
- Start Quartus II and click on **Open Project**. Select the qpf-file *"BPU_ProgrammableLogicDevice.qpf"* in your project folder and click *"Open"*.

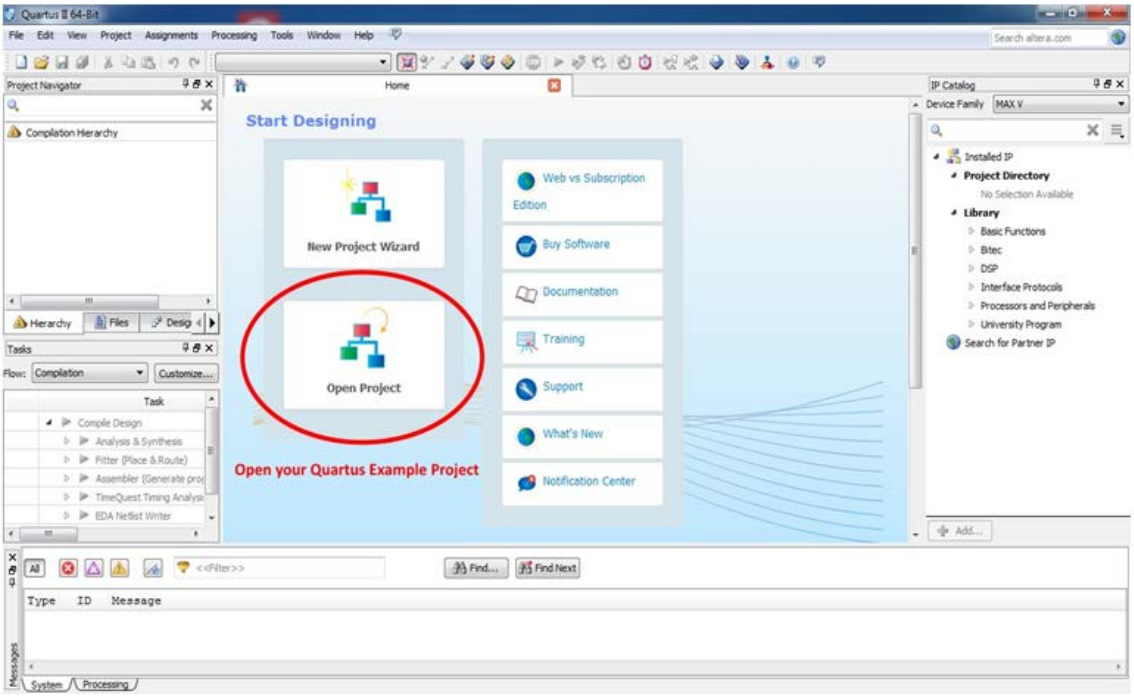


Figure 7-16: Quartus II user interface

- In the **Project Navigator** you can see the files of the Quartus project. It has mainly the following structure:

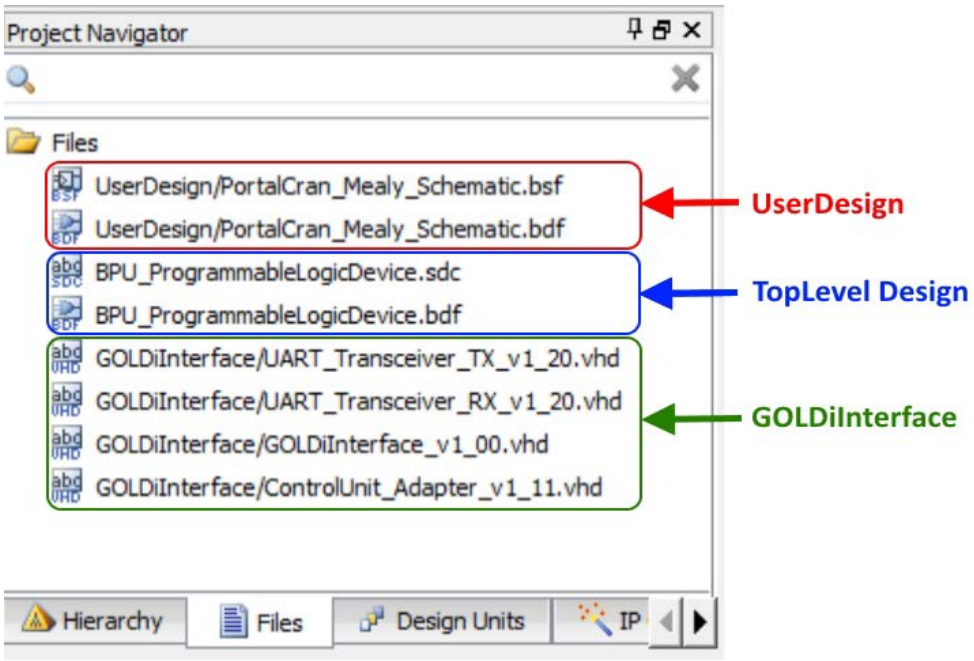


Figure 7-17: Quartus II project navigator

- The folder **GOLDiInterface** contains all the files which are necessary for the communication within the GOLDi infrastructure. These files are ready to use - no modifications are necessary. They will be included automatically.

- The folder **UserDesign** contains the actual control algorithm. It can be used a plain VHDL description (e.g. *PortalCrane_Mealy_FSM.vhd*), a design as automaton graph by using the integrated FSM graph editor (e.g. *PortalCrane_Mealy_FSM.smf*) or a schematic design (e.g.*PortalCrane_Mealy_Schematic.bdf*). In this case we use the schematic design.
- The **TopLevel Design** is the connection between the GOLDiInterface and the UserDesign. It can be implemented as a graphical block design (e.g.*BPU_ProgrammableLogicDevice.bdf*) or as a plain VHDL description (e.g. *BPU_ProgrammableLogicDevice.vhd*). In this case we use the graphical block design.

Step 2: Adapt the files to your project

The following files must be adapted according the actual control algorithm and the selected physical system:

- the **UserDesign** file according the used specification (*.vhd, *.smf or *.bdf)
- the **TopLevel Design** file BPU_ProgrammableLogicDevice (*.vhd or *.bdf)

Graphical Design

1. **UserDesign file** *./UserDesign/PortalCrane_Mealy_Schematic.bdf*

   In this file you have to adapt the schematic design to your control algorithm. The bdf-file generates the corresponding symbol in the file*./UserDesign/PortalCrane_Mealy_Schematic.bsf*.

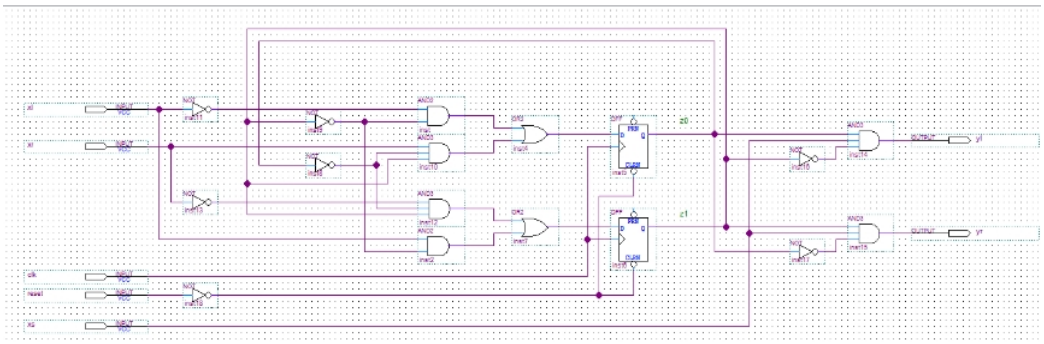


Figure 7-18: Schematic of the UserDesign (*.bdf)

2. **TopLevelDesign file** *./BPU_ProgrammableLogicDevice.bdf*

   Within this file you must insert the symbol, which was generated from the UserDesign in the file *./UserDesign/PortalCrane_Mealy_Schematic.bsf*. Furthermore the inputs and outputs of the symbol must be interconnected with the corresponding signal bits of the *GOLDiInterface_v1_00* sensor and actuator bits according the pin description of the physical system.

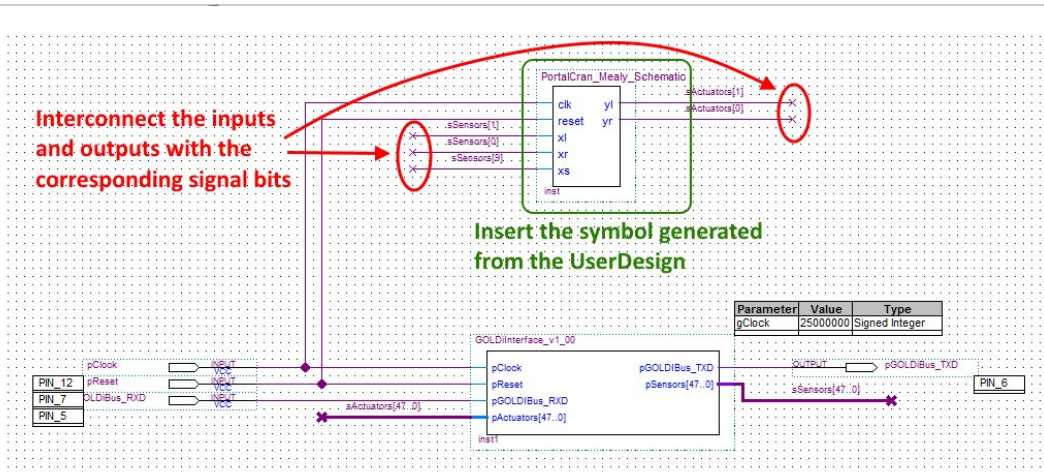


Figure 7-19: Top Level file for a schematic design (*.bdf)

Textual Design

1. **UserDesign file** *./UserDesign/PortalCrane_Mealy_FSM.vhd*

The following VHDL description was automatically generated from the user design automaton graph */UserDesign/PortalCrane_Mealy_FSM.smf*, which you can create by using the integrated State Machine Wizard.

```
-- Copyright (C) 1991-2014 Altera Corporation. All rights reserved.
-- Generated by Quartus II Version 14.1.0 Build 186 12/03/2014 SJ Web Edition
-- Created on Fri Aug 07 13:12:32 2015

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY PortalCrane_Mealy_FSM IS
    PORT (
        clock   : IN STD_LOGIC;
        reset   : IN STD_LOGIC := '0';
        xl      : IN STD_LOGIC := '0';
        xr      : IN STD_LOGIC := '0';
        xs      : IN STD_LOGIC := '0';
        yl      : OUT STD_LOGIC;
        yr      : OUT STD_LOGIC
    );
END PortalCrane_Mealy_FSM;

ARCHITECTURE BEHAVIOR OF PortalCrane_Mealy_FSM IS
    TYPE type_fstate IS (Z0,Z1,Z2);
    SIGNAL fstate : type_fstate;
    SIGNAL reg_fstate : type_fstate;
BEGIN
    PROCESS (clock,reg_fstate)
    BEGIN
        IF (clock='1' AND clock'event) THEN
    END PROCESS;
```

Figure 7-20: Textual description of the User Design in VHDL

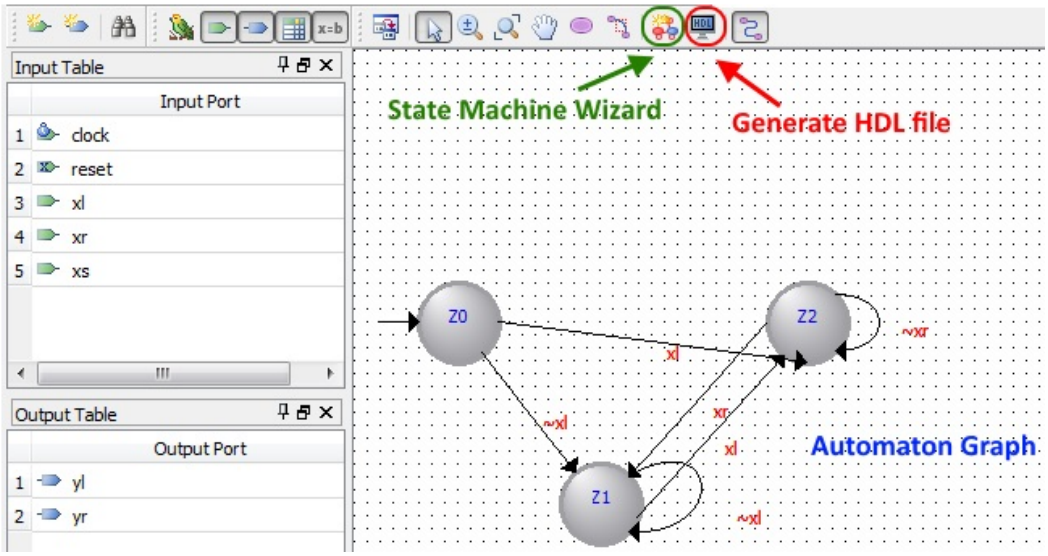


Figure 7-21: UserDesign automaton graph

1. **TopLevelDesign file** *./BPU_ProgrammableLogicDevice.vhd*

Within the TopLevelDesign file you have to modify the component description of your implemented control algorithm and the port mapping of the input and output signals to the corresponding sensor and actuator bits according the interface description of the physical system.

```
    -- ##############################################################################################
    -- ##############################################################################################
    -- ##                                                                                          ##
    -- ## User design component description => your implementation part!                           ##
    -- ##                                                                                          ##
    -- ##############################################################################################
    -- ##############################################################################################

    component PortalCrane_Mealy_FSM IS
        PORT        (
                        clock                   : IN STD_LOGIC;
                        reset                   : IN STD_LOGIC := '0';
                        xl                      : IN STD_LOGIC := '0';
                        xr                      : IN STD_LOGIC := '0';
                        xs                      : IN STD_LOGIC := '0';
                        yl                      : OUT STD_LOGIC;
                        yr                      : OUT STD_LOGIC
                    );
    END component;

begin

    -- ##############################################################################################
    -- ##############################################################################################
    -- ##                                                                                          ##
    -- ## User state machine => your implementation part!                                         ##
    -- ##                                                                                          ##
    -- ##############################################################################################
    -- ##############################################################################################

    PortalCrane_Inst : PortalCrane_Mealy_FSM
    PORT MAP(
                clock   => pClock,
                reset   => pReset,
                xl      => sSensors(1),
                xr      => sSensors(0),
                xs      => sSensors(9),
                yl      => sActuators(1),
                yr      => sActuators(0)
```

component description of your control algorithm

port mapping of the input and output signals

Figure 7-22: TopLevel file for a textual design in VHDL

Step 3: Complete the project

- After adapting the files, you have to compile your project in Quartus II. If errors occur, you have to correct them according to the error message.
- If your data is correct, you can upload the Outputfile (*.pof) from the new folder "output_files" to the experiment.

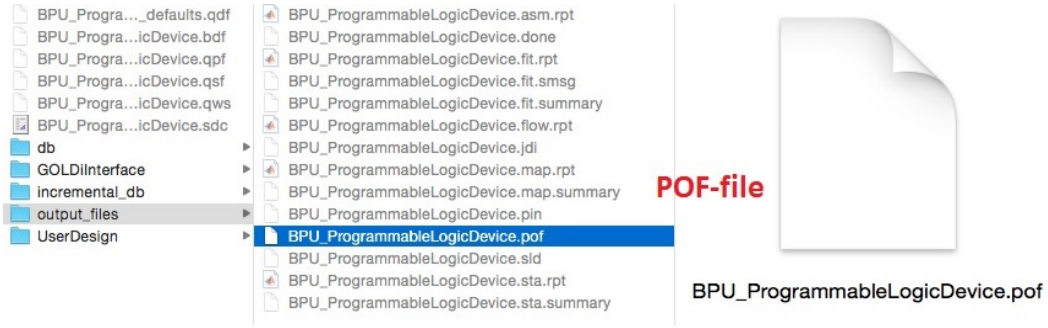


Figure 7-23: pof-file within the folder "output_files"

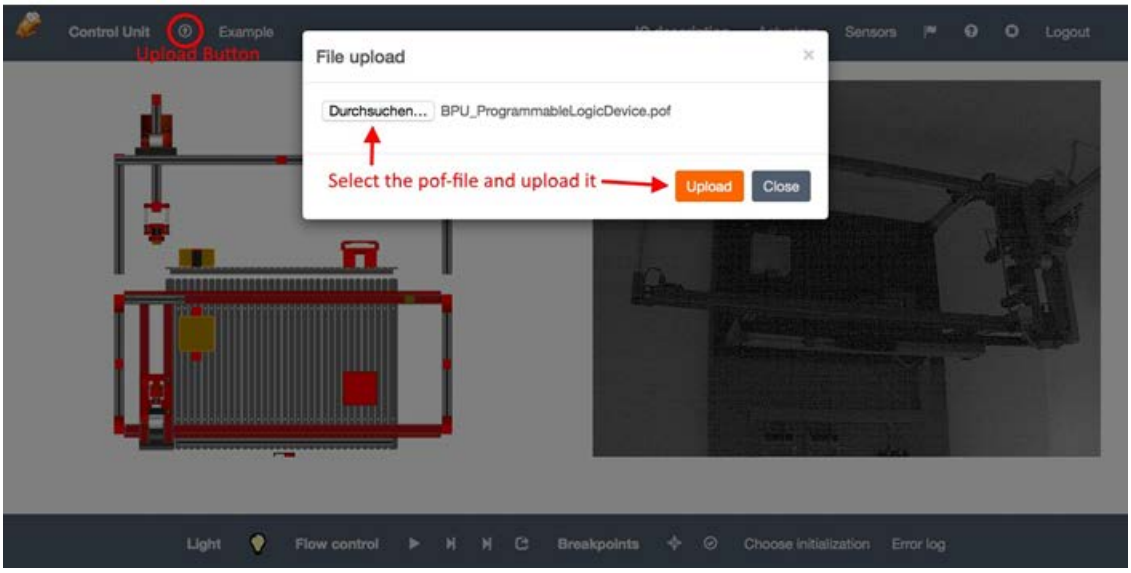


Figure 7-24: Upload the output file to the experiment

- For further information how to test your project within your chosen physical system you can read the description to "Start Experiment".

8 Practical Work (Individual Tasks)

Obligatory tasks

In preparation for the lab work, solve the following tasks:

Task 1. From the given truth table receive a completely normal disjunctive form of logic functions.

Task 2. From the given truth table receive a completely conjunctive normal form of logic functions.

Task 3. Reduce the logic functions represented as completely normal disjunctive form using a Karnaugh map.

Task 4. Reduce received Boolean functions using the algebraic method.

Decimal digit	Input signal				Output signal						
	a	*b*	*c*	*d*	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8
0	0	0	0	0	0	1	0	1	1	1	0
1	0	0	0	1	0	1	0	0	0	1	0
2	0	0	1	0	1	0	0	0	1	0	0
3	0	0	1	1	1	0	1	0	0	0	0
4	0	1	0	0	0	1	1	1	1	0	1
5	0	1	0	1	1	1	1	1	0	0	1
6	0	1	1	0	0	0	0	0	1	1	1
7	0	1	1	1	1	0	0	1	0	1	1
8	1	0	0	0	1	1	0	1	1	0	0
9	1	0	0	1	1	1	1	0	0	0	0
10	1	0	1	0	1	0	1	1	1	1	1
11	1	0	1	1	1	0	1	0	0	1	1
12	1	1	0	0	1	1	0	1	1	1	0
13	1	1	0	1	0	1	0	0	0	1	0
14	1	1	1	0	0	0	0	1	1	0	1
15	1	1	1	1	0	0	1	0	0	0	1

Task 5. A combinational logic circuit is to have four input (A, B, C and D) and four outputs (OUT1, OUT2, OUT3 and OUT4), where:

• OUT1 is logic 1 when all inputs are logic 1.

• OUT2 is logic 1 when A = B and C = D.

• OUT3 is logic 1 when any one of the inputs is logic 1.

• OUT is logic 1 when exactly two inputs are logic 1 and the other two inputs are logic 0.

For this design, develop the:

1. Truth-table.

2. Boolean logic expression.

3. Logic diagram.

Use Karnaugh Maps in order to aid in the development of a suitable Boolean logic expression.

Task 6. Full-adder cells are to be used to develop a 4-bit full-adder design.

1. Truth-table.

2. Boolean logic expression.

3. Logic diagram.

Use Karnaugh Maps in order to aid in the development of a suitable Boolean logic expression.

Task 7. A 3-input straight binary value is to be converted into 3-bit gray code format using combinational logic. For this design, develop the:

1. Truth-table.

2. Boolean logic expression.

3. Logic diagram.

Use Karnaugh Maps in order to aid in the development of a suitable Boolean logic expression.

Task 8. A 3-input straight binary value (representing 010 to +710) is to be converted into a 2's complement code format (where an input of 01 produces an output of –410 and an input of +710 represents +310) using combinational logic. For this design, develop the:

1. Truth-table.

2. Boolean logic expression.

3. Logic diagram.

Use Karnaugh Maps in order to aid in the development of a suitable Boolean logic expression.

Task 9. A combinational logic circuit is to be designed to multiply two binary numbers. Each number is 2-bits wide representing decimal numbers 0, 1, 2 and 3. The output is to be 4-bits wide. For this design, develop the:

1. Truth-table.

2. Boolean logic expression.

3. Logic diagram.

Use Karnaugh Maps in order to aid in the development of a suitable Boolean logic expression.

Task 10. Design a 3-bit gray code counter using positive-edge-triggered D-Type bistables (bistable, in which the circuit is stable in either state). The bistables are to be externally reset using an active low reset signal.

For this design, develop the:

1. State transition table.

2. State transition diagram.

3. Boolean logic expression for the combinational logic.

4. Logic diagram.

Task 11. Design a 4-bit gray code counter using positive-edge-triggered D-Type bistables. The bistables are to be externally reset using an active low reset signal.

For this design, develop the:

1. State transition table.

2. State transition diagram.

3. Boolean logic expression for the combinational logic.

4. Logic diagram.

Task 12. Design a 3-bit counter using positive-edge-triggered D-Type bistables that will follow the following count sequence:

0 1 3 5 7 0

The bistables are to be externally reset using an active low reset signal.

For this design, develop the:

1. State transition table.

2. State transition diagram.

3. Boolean logic expression for the combinational logic.

4. Logic diagram.

Task 13. Design a 3-bit counter using positive-edge-triggered D-Type bistables that will follow the following count sequence:

0 2 4 6 0

The bistables are to be externally reset using an active low reset signal.

For this design, develop the:

1. State transition table.

2. State transition diagram.

3. Boolean logic expression for the combinational logic.

4. Logic diagram.

Task 14. Design a 4-bit counter using positive-edge-triggered D-Type bistables that will follow the following count sequence: 0 2 4 6 8 10 12 14 0.

The bistables are to be externally reset using an active low reset signal.

For this design, develop the:

1. State transition table.

2. State transition diagram.

3. Boolean logic expression for the combinational logic.

4. Logic diagram.

Task 15. A 4-bit straight binary counter is to be designed with an external control signal such that when the control signal is logic 1, then the counter counts as normal. However, when the control signal is logic 0, then the counter is to remain in its current state. The bistables are to be externally reset using an active low reset signal.

For this design, develop the:

1. State transition table.

2. State transition diagram.

3. Boolean logic expression for the combinational logic.

4. Logic diagram.

Task 16. Design a 3-bit counter with an external control signal such that when the control signal is logic 1, the counter is a straight binary counter. However, when the control signal is logic 0, then the counter is a gray code counter.

The bistables are to be externally reset using an active low reset signal.

For this design, develop the:

1. State transition table.

2. State transition diagram.

3. Boolean logic expression for the combinational logic.

4. Logic diagram

Conclusion

The course “Digital Electronics” described the fundamental basics of digital hardware, starting from Boolean algebra and their representation in digital electronic circuits. Step by step more complex circuits were introduced from a functional and thereafter from a structural point of view.

After finishing the course, the students have a systematic overview about digital electronics that ranges from basic systems on gate-level, combinational circuits like decoders, multiplexers and de-multiplexers to sequential circuits like latches, flip-flops, registers and counters and finally to programmable circuits like ROM, PLA, GAL, CPLD and FPGA.

The course is a prerequisite for the course “Digital System Design” as well as for the hands-on laboratory work in the “GOLDi” remote labs of the IUT.

The course was delivered in a blended learning style, containing online lectures, homework tasks and presence phases for consultation.

After finishing the course, students are able to identify combinational and sequential digital circuits. They can describe their structure as well as their function in a formalized manner. The students can analyze digital circuit diagrams and transfer them into a formal description based on Boolean algebra.

So they are prepared to solve tasks in the remote lab GOLDi to control physical systems with digital control algorithms based on Finite State Machines on different platforms like microcontrollers and FPGAs.

Literature

1. Grout, Ian. *Digital Systems Design with FPGAs and CPLDs*. Amsterdam: Elsevier / Newnes, 2008. ISBN: 978-0-7506-8397-5.
2. Wakerly, John F. Digital Design: Principles and Practices. Upper Saddle River, NJ: Prentice Hall, 2001.
3. Floyd, Thomas L. Digital Fundamentals. Taipei: Central Book, 1986.
4. Salina, Muhamad, and Muhammad Nazir Mohammed Khalid. Digital Electronics. Oxford: Oxford University, 2013.
5. Floyd, Thomas L. Digital Fundamentals. Taipei: Central Book, 1986.
6. Dueck, Robert K., and Ken Reid. Digital Electronics. Clifton Park, NY: Delmar/Cengage Learning, 2012.
7. Kharate, G. K. Digital Electronics. New Delhi: Oxford UP, 2012.
8. Prager R., Flack T. Combinational Logic , 2009.
9. Questions and answers: http://www.allaboutcircuits.com/worksheets/flipflop.html
10. Quiz: https://filebox.ece.vt.edu/~jgtront/introcomp/flipflop_quiz.swf
11. Davis J., Reese R. Finite State Machine Datapath Design, Optimization, and Implementation. Synthesis Lectures on Digital Circuits and Systems. 2008, 123 p.
12. Peter Minns, Ian Elliott FSM–based Digital Design Using Verilog HDL (+ CD-ROM), John Wiley and Sons, 2008.
13. All about Circuits: http://www.allaboutcircuits.com/vol_4/chpt_11/5.html
14. Hybride online Lab GOLDi, TU Ilmenau: http://goldi-labs.net/