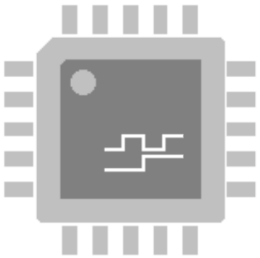
Computer Architecture



Laurentiu-Cristian Duca
Anton Duca

Computer Architecture

Copyright © 2020

All rights reserved to the authors. This book is released under CC BY-SA 2.0 license.

Every effort has been made in order to make this book and its contents accurate and complete, but the information contained in this book is provided as is, without warranty, either expressed or implied. Neither the authors, nor the publisher will be held liable to any loss or damages arising directly or indirectly from the information contained in this book.

Foreword

This book is intended to be used by the students of all ages who want to learn about computer architecture. Its main scope is to provide an overview about computer systems implementation and how they work. It presents how the processors, buses and drivers are implemented on FPGA chips. The presented materials represent an accumulated knowledge in more than 10 years experience that we have in this field.

As prerequisites, the readers should have programming skills and be familiar with the C programming language. The level is beginner to intermediate.

The theories presented here are practically illustrated with free and open source projects. To download the projects, please navigate to the following address: https://github.com/laurentiuduca. Further, every chapter is provided with a specific set of exercises.

The projects have been implemented and tested on the cheapest SBC on the market, Raspberry pi zero WH and a Digilent FPGA board, using the free Xilinx ISE Webpack or Xilinx Vivado Webpack software programs – but using any FPGA board and other software to run the projects should be a similar process.

The examples are written in C, Verilog and VHDL languages. Free tools available on Linux and Windows like Icarus Verilog and the ghdl VHDL tool were used in simulations along with gtkwave. Tutorials on how to use these tools are presented in the book.

Finally, we hope to enjoy reading and rise up your computer knowledge level with the demonstrations presented here.

The authors

Contents

1. Introduction

The CPU is the computer brain. Learning how the CPU, memory and drivers work will make you understand how a computer works.

The CPU synchronizes its activity with a signal named clock which turns on and off with a given frequency. The clock frequency dictates the rate at which the CPU executes instructions and common values have MHz or GHz order.

The CPU has pins which connect him to the system components. The simplified architecture of a classic computer machine is presented in Fig. 1-1.

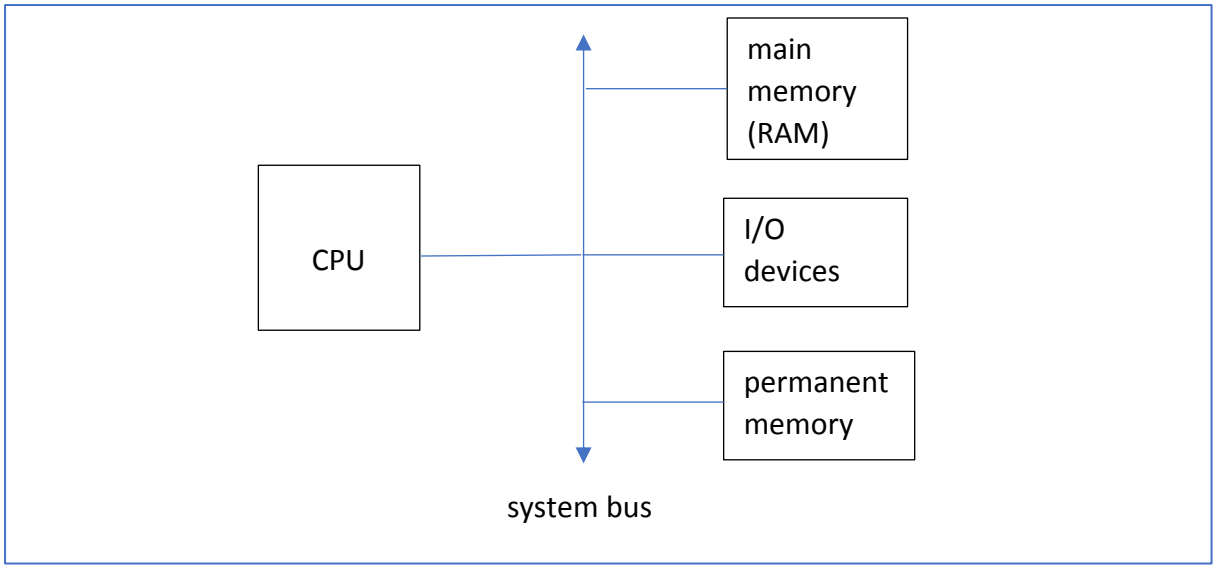


Fig. 1-1. Architecture of a classic computer

Programs are sets of instructions which are stored in the permanent memory. The CPU can load programs from the permanent memory to RAM memory. At runtime, the CPU loads and executes programs instruction by instruction.

In a PC, the RAM memory is much faster than the permanent memory but it is much slower than the CPU. RAM stands for Random Access Memory and contains the data processed by the CPU; its contents are volatile: after power off the RAM content is lost, so the need of the permanent (non-volatile) memory.

In the memory data is organized in addresses; each address corresponds to a piece of data. Although regularly the CPU accesses RAM data sequentially with respect to the RAM addresses, it can also access it in a random order and many times "jumps" from an address to another.

Computer systems internally represents data in base 2, using 0s and 1s. In this context we define a bit which can be 0 or 1 and a byte which contains 8 bits. In computer systems

data is organized in bytes. Regular values of the length of the CPU instruction are 8, 16, 32, 64 and 128. If the length of the CPU instruction is 32 then we have a so called 32bits CPU.

In order for the CPU to execute an instruction it must first load it from memory by setting on the system bus the address of that instruction. The CPU can also write data to memory by specifying the address where it wants the data to be written and setting a special signal called write (WR).

The data in RAM can be instructions, stand alone values or addresses to other instructions. The CPU accesses RAM, permanent memory and I/O devices by setting specific addresses on the system bus.

The total numbers of addresses a CPU can access is called CPU address space. If an I/O device can be accessed at a particular address then it is said that the I/O device it is "mapped" at that address. Examples of I/O devices include the network card, USB devices, printers, monitors, keyboard, mouse, sensors.

Each character that can be printed on the basic screen (like letters and numbers) has associated in the ASCII table a specific value; this table also contains values for non-printable characters like carriage return '\r' (code 13) and line feed '\n' (code 10). ASCII contains a total of 256 characters values.

Each CPU has its own set of instructions that it understands and a set of registers (will see more about registers in the next chapter). The load instruction loads a piece of data from the memory to a register. The store instruction saves in memory the value of a register.

The ALU (arithmetic logical unit) is an execution unit of the CPU; in ALU operations, register values can be added, subtracted, multiplied, divided, compared, shifted, etc. There are also flow control instructions which may change the address of the next instruction that will be executed by the CPU; these are called jump or branch instructions.

It is worth noting the CPU control unit (CU) which divides instructions into specific values for CPU components (for example ALU): for example it commands the ALU to add two values and then it saves the result in a specific register or loads a data word from memory, which is contained at the sum resulted address.

The computers which have a von Neumann architecture store the instructions and data in the same memory; the Harvard architecture specifies different memories for instructions and data.

In the next chapters will go deeply into how CPUs, memories and buses are built and how they function.

2. From transistor to register

Computer systems internally represents and operates with data in base 2, using 0s and 1s. The CPU is made by logic gates and registers which are made of transistors. Fig. 2-1 presents the main type of logic gates: AND, NAND, OR, NOR, XOR, XNOR which operate on two inputs and INV and tri-state buffer which operate on one input. In practice, the 1 logic may be 5V or 3.3V or 1.8V and 0 logic is 0V.

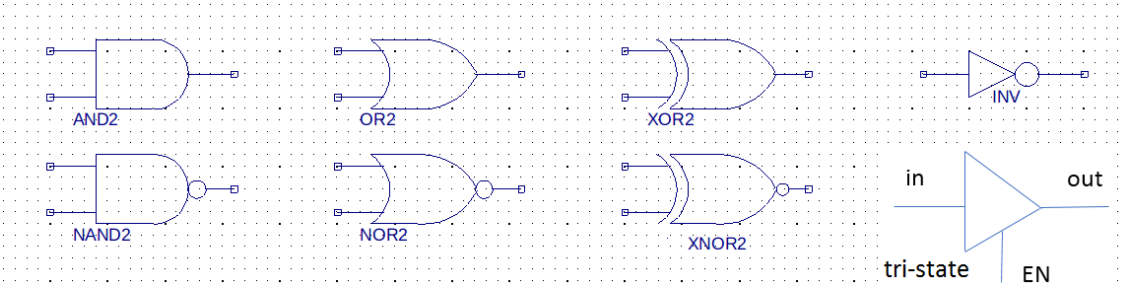


Fig. 2-1. Logic gates

The truth table of a logic gate is a table that contains the output for each of the input combinations. The truth tables of the above logic gates are presented in table 2-1 and table 2-2. Consider a and b the inputs.

a	b		AND	NAND	OR	NOR	XOR	XNOR
0	0		0	1	0	1	0	1
0	1		0	1	1	0	1	0
1	0		0	1	1	0	1	0
1	1		1	0	1	0	0	1

Table 2-1. Truth table for the 2 input gates

in	EN	TS
0	0	high impedance
1	0	high impedance
0	1	0
1	1	1

Table 1-2. Truth table for the tri-state buffer

The INV (inverter) is very simple: INV(0)=1 and INV(1)=0. The tri-state buffer acts like a switch: when EN is 0, the output is in high impedance (switch off); when EN is 1, the output equals to the input.

It's worth mentioning the DeMorgan's law. If we note AND(a,b) = a&b, OR(a,b) = a|b and INV(a) = !a, then DeMorgan's law says that !(a&b) = !a | !b and !(a|b) = !a & !b. Also note that using logic gates any discrete (digital) function can be implemented.

In Fig. 2-2 is illustrated the role of the CMOS transistor in designing logic gates. Here is represented the INV gate but the rest of the gates have a similar concept. A CMOS transistor can be n-channel or p-channel. The transistor acts like a switch. When gate is 1, the n-channel transistor is switched on; when gate is 0, the n-channel transistor is switched off. When gate is 1, the p-channel transistor is switched off; when gate is 0, the p-channel transistor is switched on.

The inverter circuit is made with two transistors, one n-channel and one p-channel:
- when the input is 1, the p-channel transistor is switched off and the n-channel is on; so, the output of the inverter is GND (Ground, 0V) which means logic 0;
- when the input is 0, the p-channel transistor is on and the n-channel is off; so, the output of the inverter is V_{DD} which means logic 1.

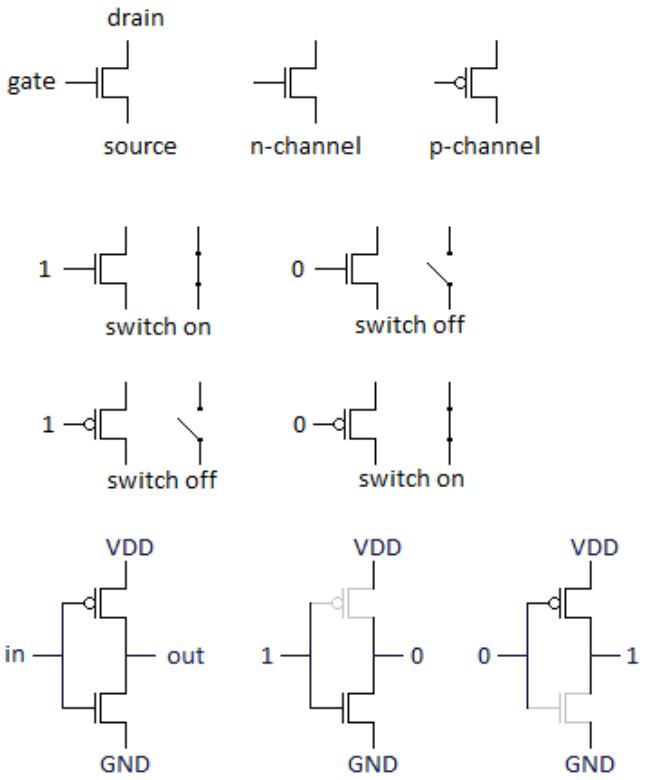


Fig. 2-2. The inverter circuit made with CMOS transistors

The main advantage of CMOS transistors over bipolar transistors is the considerable smaller power dissipation and, as a follow up, the CMOS technology is used in computer chip design.

The multiplexer (MUX) with two inputs (in0, in1) one selector (sel) and one output (out) made with logic gates is presented in Fig. 2-3. Its truth table is shown in table 2-3.

sel	out
0	in0
1	in1

Table 2-3. The truth table of MUX(2:1)

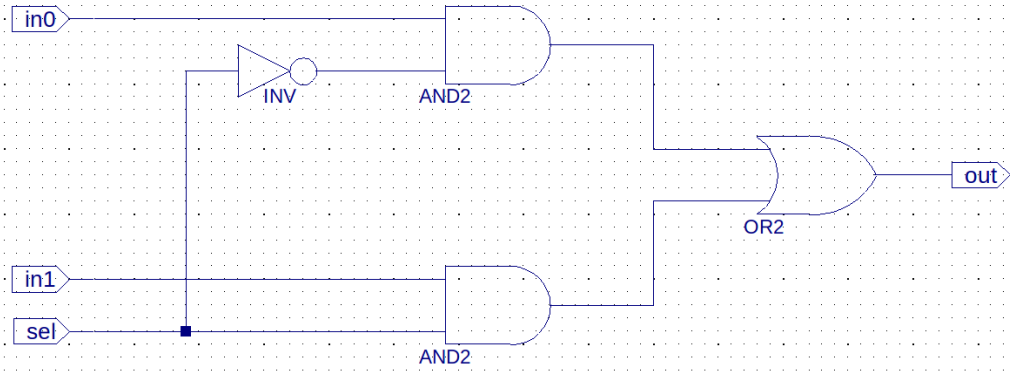


Fig. 2-3. A simple multiplexer

When sel is 0, the upper AND gate has the inputs in0 and 1 – so, its output is in0, while the down AND gate has the inputs in1 and 0 – so, its output is 0; the OR gate has inputs in0 and 0, so the output of the multiplexer is in0. When sel is 1, the theory is similar and the out is in1.

Using one MUX, a latch can be made. This is illustrated in Fig. 2-4. The latch's function is the following: when cmd is 1, the output out1 is d. When cmd is 0, out1 is in0 which is tied to out1 – so the value of the latch remains the same.

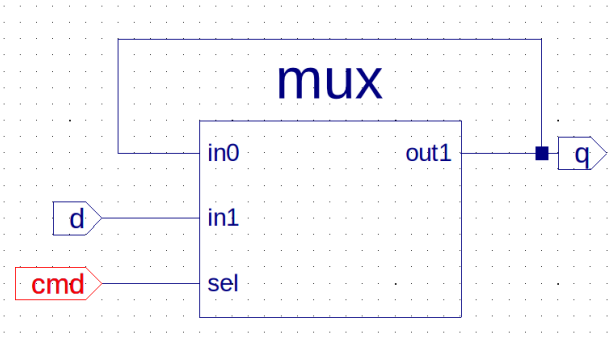


Fig. 2-4. Latch made with MUX

Using two latches, a D flip-flop (which represents a one-bit register) can be made. Normally, the input of the D flip-flop is noted as D and the output of the D flip-flop is noted as Q, but here, cause of commodity reasons we have noted them as d and q. This is illustrated in Fig. 2-5. The function of the D flip-flop is represented in table 1-4: the output of the flip-flop at moment (t+1) is set to its input at moment (t).

t	t+1
d	q
0	0
1	1

Table 1-4. D flip-flop function

When clk is 0, the first latch output is d while the second latch output remains the same. When clk is 1, the first latch output remains to its previous value (d) while the second latch output is set to the first latch output (d). We say that the register output q is set to d on the clk rising (positive) edge (when the clk changes from 0 to 1). With a similar logic it can be designed a register that is active on the falling (negative) clk edge.

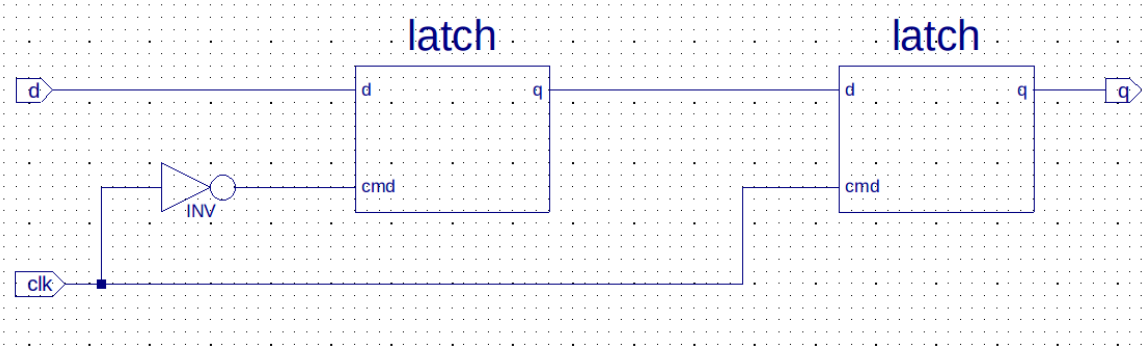


Fig. 2-5. A one-bit register made with two latches

A simulation scenario of the presented theory is shown in Fig. 2-6: d and clk are inputs, q1 is the output of the first latch and q is the output of the second latch. When the clk is low, q1 <= d; when the clk is high, q <= q1. Here we consider that the output q1 is set to d with some delay, and q is set to q1 with a similar delay.

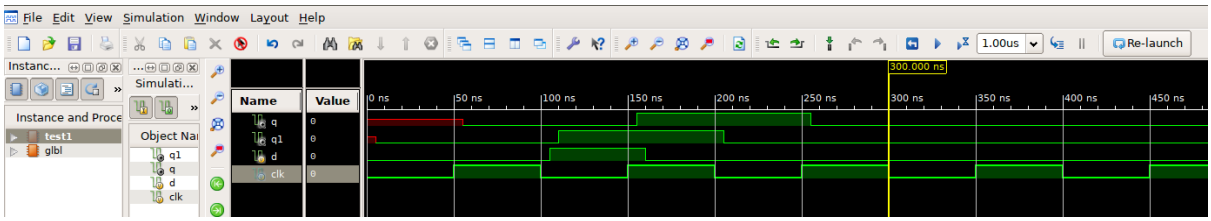


Fig. 2-6. Simulation of the one bit register

In order for the D flip-flop to function correctly, d must be setup before rising clk edge with an amount of time denoted as setup time and also d must be hold after the rising clk edge with an amount of time denoted as hold time. This is illustrated in Fig. 2-7.

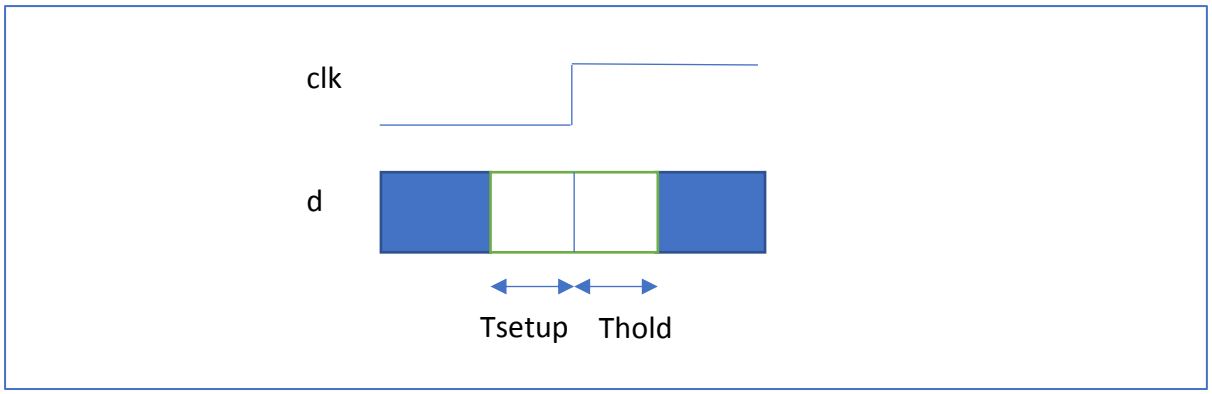


Fig. 2-7. D flip-flop Tsetup and Thold

The D flip-flop it is a sequential type circuit because it depends on the timing of the inputs. The classic logic gates are combinational type circuits because they are time independent circuits.

When designing hardware drivers, we must be aware of the clock skew and the clock jitter. The clock skew is the difference (or the variation) from the moments when the same clock edge arrives at two or more different flip flops. The clock jitter is the difference from the effective moment when the clock edge arrives at a flip flop and the ideal moment when the clock edge should have arrived at the same flip flop.

3. Computer arithmetic

3.1 Base 2, base 10 and base 16 for natural and positive numbers

Let x be a natural number and b a numerical base; then x can be written in base b as:

$$x_{(b)} = a_n a_{n-1} a_{n-2} \dots a_2 a_1 a_0$$

Each of the digits $a_n, a_{n-1}, a_{n-2}, \dots a_2, a_1, a_0$ is less than b and greater or equal to 0.

The value x can be converted in base 10 with the following formula:

$$y_{(10)} = x_{(b)} = a_n * b^n + a_{n-1} * b^{n-1} + \dots + a_1 b^1 + a_0 b^0$$

For example, 21 in base 16 can be written as $2*16^1+1*16^0 = 32 + 1 = 33$ in base 10; we write it as $21_{(16)}=33_{(10)}$.

In base 2 only digits 0 and 1 are used. The addition rules are the following: 0+0=0, 0+1=1, 1+0=1, 1+1=10. Note that 10 in base 2 is $1*2^1 + 0*2^0$ and equals to 2 in base 10; we write it as: $10_{(2)}=2_{(10)}$. Similarly, the number 1001 in base 2 is $1*2^3 + 0*2^2 + 0*2^1 + 1*2^0$ and equals to 9 in base 10; we write it as: $1001_{(2)} = 9_{(10)}$.

To convert a number x written in base 10 to base 2, the following algorithm can be used:

```
int i = 0, n = 0;
do {
  a[n] = x mod 2;
  x = (int) x / 2;
  n = n + 1;
} while (x > 0);
// the number in base 2 is is a_{n-1}a_{n-2}...a_1a_0
for (i = n-1; i >= 0; i--)
  printf("%d", a[i]);
```

Listing 3-1. Convertion from base 10 to base 2

For example let's convert x=20 from base 10 to base 2:

```
n=0: a[0] = 20 mod 2 = 0; x= 20 / 2 = 10.
n=1: a[1] = 10 mod 2 = 0; x = 10 / 2 = 5.
n=2: a[2] = 5 mod 2 = 1;   x = 5 / 2 = 2.
n=3: a[3] = 2 mod 2 = 0;  x = 2 / 2 = 1.
n=4: a[4] = 1 mod 2 = 1; x = 1 / 2 = 0.
Result: 20 in base 2 = 10100
```

Listing 3-2. Converting 20 from base 10 to base 2

Another important base is base 16. The first 16 natural numbers are noted in base 16 as 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. This is represented in table 3-1.

Base 16	Base 10	Base 2
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
a	10	1010
b	11	1011
c	12	1100
d	13	1101
e	14	1110
f	15	1111

Table 3-1. Base 16, base 10 and base 2 representation for the first 16 natural numbers

Conversion from base 16 to base 2 is simple: for each digit in base 16 write its conversion in base 2. Example: $a5_{(16)} = 1010\ 0101_{(2)}$.

Conversion from base 2 to base 16 is also immediate: for each four digits in base 2 write a digit in base 16. Example: $01001100_{(2)} = 4c_{(16)}$.

3.2 Application – A modulo 5 counter

In the following example we will implement a counter modulo 5. That means that will count repeatedly from 0 to 4 in base 2: 000, 001, 010, 011, 100, 000, 001, …

We will memorize the counter value in three D flip-flops. The following table shows the next value for each of the flip-flops considering known the current value.

decimal value (t)	period (t) q2q1q0	period (t+1) q2q1q0	decimal value (t+1)
0	000	001	1
1	001	010	2
2	010	011	3
3	011	100	4
4	100	000	0

Table 3-2. Computing next value for the modulo 5 counter

Considering table 3-2, we can compute the equations for the D flip-flops:

```
q2(t+1) = !q2 & q1 & q0
q1(t+1) = (!q2 & !q1 & q0) | (!q2 & q1 & !q0)
q0(t+1) = (!q2 & !q1 & !q0) | (!q2 & q1 & !q0)
```

Listing 3-1. Counter modulo 5 equations

The equations were computed considering the following principle which is presented for q1(t+1) as example. Look in the (t+1) column where the bit corresponding to q1 is 1 and choose from the (t) column the corresponding lines; these lines are 001 and 010 (corresponding to decimal value at period t, 1 and 2):

- the first line is q2q1q0=001 and we put in the q1(t+1) equation the formula (!q2 & !q1 & q0) because the value for q2 is 0, the value for q1 is 0 and the value for q0 is 1;

- the second line is q2q1q0=010 and we put in the q1(t+1) equation the formula (!q2 & q1 & !q0) because the value for q2 is 0, the value for q1 is 1 and the value for q0 is 0.

The resulting schematic is shown in Fig. 3-1. There were chosen flip-flops with reset signal (noted FDR), in order to have a reset state of 000 for the counter.

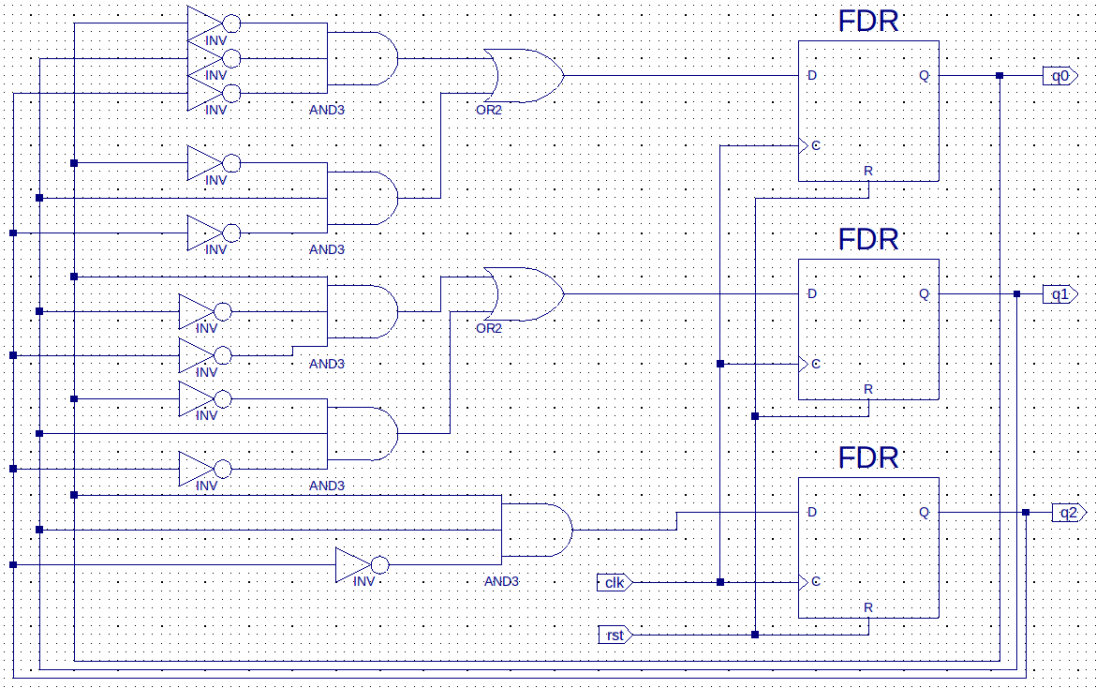


Fig. 3-1. Modulo 5 counter scheme.

The simulation of the modulo 5 counter is shown in Fig. 3-2. When rst is 1, q2q1q0=0 and then at each clock rising edge the counter advances. This happens until the counter q2q1q0 reaches value 4 and then the counter starts again with 0.

Fig. 3-2. Counter modulo 5 simulation

Exercise 3-1. Implement a counter modulo 6 scheme.

3.3 Application – full adder

A 1-bit full adder is presented in Fig. 3-3.

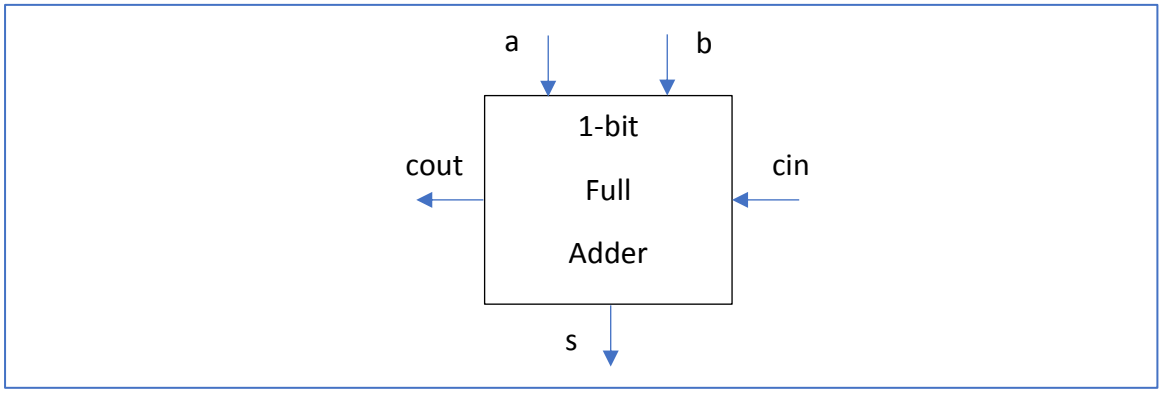


Fig. 3-3. A 1-bit full adder diagram

The truth table for the 1-bit full adder is represented in the table below. Here we abbreviate carry-in with cin and carry-out with cout.

a	0	0	0	0	1	1	1	1
b	0	0	1	1	0	0	1	1
cin	0	1	0	1	0	1	0	1
s	0	1	1	0	1	0	0	1
cout	0	0	0	1	0	1	1	1

Table 3-2. Truth table for 1-bit full adder

The outputs logic functions based on the inputs are (here we note xor as ^):

s = a ^ b ^ cin

cout = (a & b) | (a & cin) | (b & cin)

More 1-bit full adders can be chained in order to compute a sum on more bits. The chaining is made by pinning carry-out of a 1-bit adder to carry-in of the next 1-bit adder.

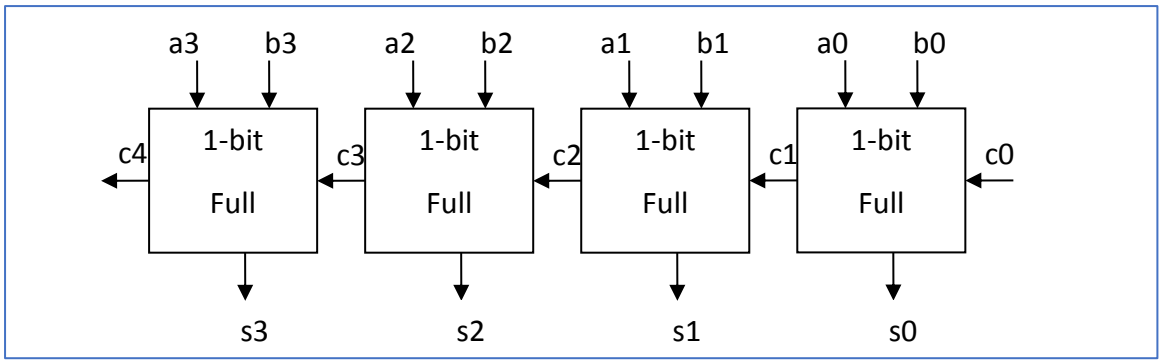


Fig. 3-4. 4-bit adder

This chained architecture for the adder although looks simple it can be inefficient for summing large numbers (represented on many bits) because in order to be able to compute rank i it is needed to wait the computing of the rank i-1 carry-out and so on. In industry there are more efficient adders (like carry look ahead adder) used but these are not discussed here – the scope was to present the concept.

3.3 Signed numbers arithmetic

In computers, signed numbers are represented in two's complement notation. In two's complement notation, the left most bit of a considered signed number represents its sign: if it is 0 then the number is positive; if it is 1 then the number is negative.

Let a and b two numbers represented in two's complement notation (from now on we abbreviate it with cc); then (a - b)cc is identical to (a)cc + (-b)cc.

Let b > 0 a number represented on n bits; then (-b)cc = 2^n – b. General conversion:

$$(-b)cc = \sim b + 1$$

Here we denoted ~b as a bitwise inverter for b. For example if b=0010, then ~b=1101; another example: if b=1110, then ~b=0001.

Let b a number represented on four bits (for simplicity). The following table shows the conversion of (b)cc to (-b)cc. Here we noted (-b)cc with mb. From table 3-3 we can compute the logic equations for mb3, mb2, mb1, mb0. For example the mb3 bit has the formula: mb3=~b3 & (b2 | b1 | b0); see the explanation in chapter 3.2 where we computed the equations for the modulo 5 counter.

value	3210=b	3210=mb	value
0	0000	0000	0
1	0001	1111	-1 = 15
2	0010	1110	-2 = 14
3	0011	1101	-3 = 13
4	0100	1100	-4 = 12
5	0101	1011	-5 = 11
6	0110	1010	-6 = 10
7	0111	1001	-7 = 9

Table 3-3. Positive to negative conversion

In table 3-4 is presented negative to positive conversion.

3210=b	val	val	3210=mb
0000	0	0	0000
1111	-1 = 15	1	0001
1110	-2 = 14	2	0010
1101	-3 = 13	3	0011
1100	-4 = 12	4	0100
1011	-5 = 11	5	0101
1010	-6 = 10	6	0110
1001	-7 = 9	7	0111
1000	-8 = 8	-8	1000

Table 3-4. Negative to positive conversion

Note that in the case of four bit representation, (-(-8)cc)cc is (-8)cc. This is also the case of C compilers; for example, consider the following C program sequence:

```
{char c, d;
  c = -128;
  d = -c;
  /* now d is -128, because signed numbers on 8 bits are -128...0…127 */
  printf("%d",d);
}
```

Listing 3-2. Natural negative numbers in C language

When adding or subtracting signed numbers we can see if the result is valid or not (and in this case we have overflow). For example let's consider two 4-bit signed numbers a and b. We note cout[i] as carry-out of a[i]+b[i]. We can now define the overflow equation: of=cout[3] ^ cout[2]. See examples in table 3-5.

a=2, b=5, r=a+b=7 0010 a 0101 b 0000 cout 0111 r of=0	a=7, b=5, r=a+(-b)=2 0111 a 1011 (-b) 1111 cout 0010 r of=0	a=5, b=7, r=a+(-b)=-2 0101 a 1001 (-b) 0001 cout 1110 r of=0	a=5, b=7, r=a+b=12, 0101 a 0111 b 0111 cout 1100 r=-4 of=1	a=-5, b=-7, r=a+b=-12, 1011 a 1001 b 1011 cout 0100 r=4 of=1

Table 3-5. 4-bit signed number addition and subtraction examples

3.4 Floating point arithmetic

The IEEE 754 standard specifies how floating point numbers are defined. It specifies the single, double and quad precision standards. In all the standards, the data structure (the number itself) is represented as a tuple of sign (S), exponent(E) and mantissa(M):

S	E	M

Fig. 3-5. Floating point number representation

In the single precision format, S size is 1bit (0 means positive and 1 means negative), E size is 8 bits and M size is 23 bits. The value of the stored number is:

$$Value = (-1)^S * 2^{E-127} * (1.M)$$

Thus, the mantissa has one more bit which is "hidden" and is always 1 in a normalized value.

The value of E is expressed in excess 127, which means that E is an unsigned number between 0…255. The actual exponent is computed as: exp = E – 127. Some values are reserved (see [IEEE754]):
- E=0 and M=0 represents number zero;
- E=0 and M!=0 represents a denormalized number (as a consequence of an error);
- E=max and M=0 represents (plus or minus) infinity;
- E=max and M!=0 represents NaN – not a number (when dividing zero by zero).

The minimum value that can be represented is roughly 2^{-126} and the maximum is about 2^{127}.

For example, to represent 0.25 in single precision:
- S=0 (positive number);
- consider that $0.25_{(10)}=0.01_{(2)}$, because $0.01_{(2)}=0*2^0+0*2^{-1}+1*2^{-2}=0.25_{(10)}$
- normalize mantissa: $0.01 = 1.0 * 2^{-2}$, so mantissa=1.0 which implies M=0, and will subtract 2 from E
- E=127+(-2)=125=01111101

0	01111101	00000000000000000000000

Fig. 3-5. The 0.25 floating point number representation

In order to work with floating point numbers, we must define two operators: shift left and shift right. Shift left is noted as << and shift right is noted as >>. They receive two inputs: a number N and an argument a.

- shift left is: N << a = N * 2^a
- shift right is: N >> a = N / 2^a

Examples: N=10110. Then: N >> 1 = 1011, N >> 2 = 101, N << 1 = 101100, N << 2 = 1011000.

We explain the floating point **addition algorithm** by using an example: compute N_1+N_2 where N_1=100, N_2=0.25.

100 is 0|10000101|10010000000000000000000 and

0.25 is 0|01111101|00000000000000000000000

The exponents differ, so the smaller which is E_2 must be increased to E_1, and as a drawback M_2 decreases: mantissa2 (which is $1.M_2$) will be shift right with E_1-E_2 positions, which is 133 – 125 = 8:

10000000000000000000000 >> 8 = 00000000100000000000000

Now the mantissas are added (we consider the hidden bit also):

11001000000000000000000+
00000000100000000000000=
11001000100000000000000

The resulted mantissa is normalized.

If the resulted mantissa would be too small, we must normalize it by shifting the it left bit by bit until it has 1 to the leftmost position and decrease the result exponent each time the result mantissa is shifted.

If the resulted mantissa would be too big, we must normalize it by shifting it right with one position and increment the result exponent once.

In our example the resulted mantissa is already normalized, so after we extract the hidden bit, the addition result is:

sum=0|10000101|10010001000000000000000

In the floating point addition and multiplication algorithms, if the resulted exponent is greater than maximum we have an overflow and if it is smaller than minimum we have an underflow error.

The floating point **multiplication algorithm** of two numbers N_1 and N_2 (whose values are different than zero) is:

- resultS=S_1 ^ S_2 ;
- multiply the mantissas; the result of 24bit mantissas multiplication is a 48 bits number with two bits after the binary point; this number will be truncated to 24 bits;
- resultE=E_1+E_2-127, and verify for underflow and overflow;
- if the result mantissa needs normalization then shift it right with one position and increment the result exponent (and verify for overflow).

Let's see an example: N_1=5, N_2=9.5.

N_1= 0|10000001|01000000000000000000000

N_2= 0|10000010|00110000000000000000000

- multiply the mantissas:

101000000000000000000000*

100110000000000000000000=

010111110000000000000000000000000000000000000000

- add the exponents:

E_1+E_2-127=10000001+10000010-01111111=10000100

- mantissa is normalized, so result is 47.5:

0|10000100|01111100000000000000000

Finally, we mention that in the double precision format, S size is 1 bit, E size is 11 bits and M is 52 bits wide and in gcc <float.h>, FLT_MAX_EXP=128, FLT_MIN_EXP=-125, FLT_MAX_10_EXP=38, FLT_MIN_10_EXP=-37, DBL_MAX=1024, DBL_MIN=-1021, DBL_MAX_10_EXP=308, DBL_MIN_10_EXP=-307.

The IEEE-754 floating point representation can not represent all real numbers; for those real numbers that can not be represented, IEEE-754 will have an approximation equivalent.

In order to easy convert decimal values to IEEE754 floating point, one can use the IEEE-754 Floating Point Converter site; see reference [FloatConv].

In [Dowson12] is presented a free and open source IEEE 754 floating-point implementation in Verilog, which I recommend for further reading along with an easy to follow floating point arithmetic presentation that can be found in [Shaaban99].

4. FPGAs and FPGA tools

4.1 FPGA structure

FPGAs (field programmable gate arrays) are chips made by inter-connected configurable elements. FPGAs are reconfigurable. ASICs (application specific integrated circuits) are chips meant for only one specific application and once created, their structure can not be altered. Comparing an ASIC with an FPGA created for the same application, the ASIC has fewer transistors than the FPGA, so from this point of view the ASICs are cheaper – but the application design costs are higher in case of ASICs because the developers must be sure that their design is correct. In this way, using FPGAs is better for low volume products and for prototyping.

The reconfigurable elements that an FPGA is made of are named look-up-tables or shortly LUTS. Let's present the structure of a LUT starting from an example. Let f be a function with two inputs and one output. The inputs and the output are 1bit wide. The truth table for the f function is shown in table 4-1. The outputs f_0, f_1, f_2 and f_3 are considered to be known. The structure of the LUT implementing f is presented in Fig. 4-1 (here the D flip-flop is noted FD).

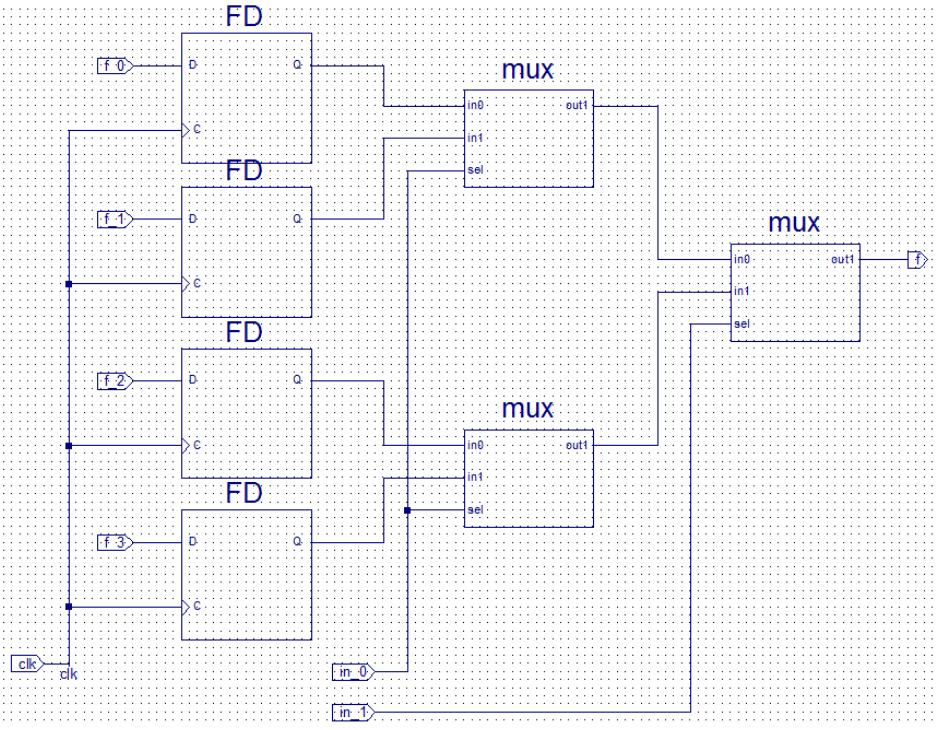


Fig. 4-1. The structure of the 2 input LUT

in0	in1	f
0	0	f_0
0	1	f_1
1	0	f_2
1	1	f_3

Table 4-1. The truth table for the f function

The tools and the processes involved in FPGA configuration building are different than the tools involved in a CPU machine code generation. Fig. 4-2 shows the CPU machine code generation process while Fig. 4-3 shows the FPGA configuration building process.

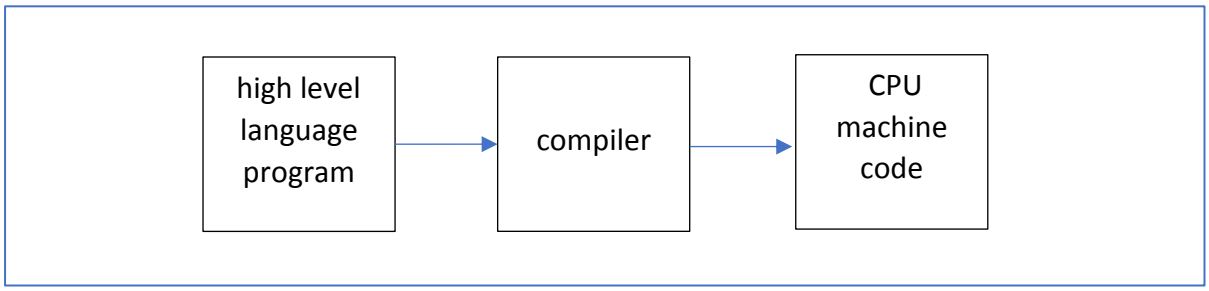


Fig. 4-2. CPU machine code generation process

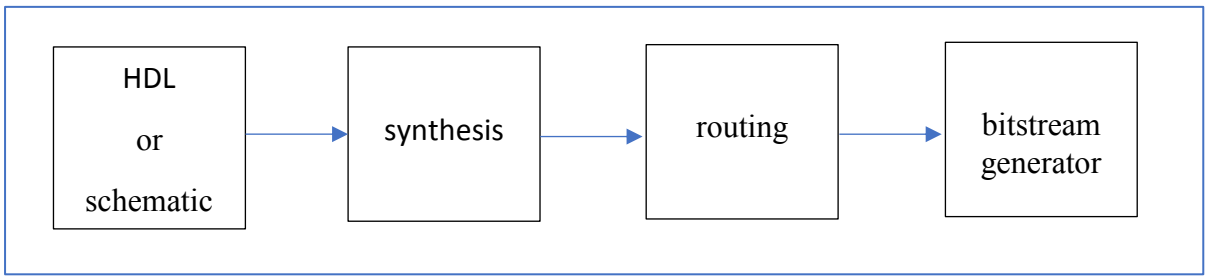


Fig. 4-3. FPGA bitstream generation process

We use the Xilinx naming for the final file that is used to re/configure the FPGA: bitstream. The initial step is creating a module in a hardware description language (HDL) or a scheme (as we saw in the modulo 5 counter previously presented). The synthesis process produces a netlist which contains connected gates and flip-flops. The routing process maps these elements to FPGA resources (LUTs, flip-flops, etc). Finally, the bitstream generator generates the file that will be used to configure the FPGA to do what we specified in the HDL file.

Inside the FPGA, LUTs can also be configured as distributed memory elements. It's worth noting that beside LUTs, modern FPGAs also incorporates configurable memory BRAMs (block RAMs) and DSPs in their structures.

The main FPGA producers and their tools are presented in the following table.

Companies	FPGAs	Partially free software tools
Xilinx	Spartan, Artix, Kintex, Virtex	Xilinx ISE webpack, Xilinx Vivado webpack
Intel (Altera)	Cyclone, Arria, Stratix	Quartus Prime Lite Edition
Lattice	ECP, ice40	Lattice Diamond, IceStorm (Yosys, Arachne-pnr)

Table 4-2. FPGA producers, chips and tools

The most used HDL languages are Verilog and VHDL. Free simulation tools are Icarus for Verilog HDL and ghdl for VHDL which work with gtkwave. Non free FPGA and HDL software tools are Synopsis, Cadence, Modelsim, Matlab, LabView, etc.

4.2 Xilinx Vivado tutorial

In the following we present a Xilinx Vivado HL Webpack tutorial in order to have an overall view about the FPGA design process. The Xilinx Vivado installation kit can be downloaded from the Xilinx website [XilinxVivado].

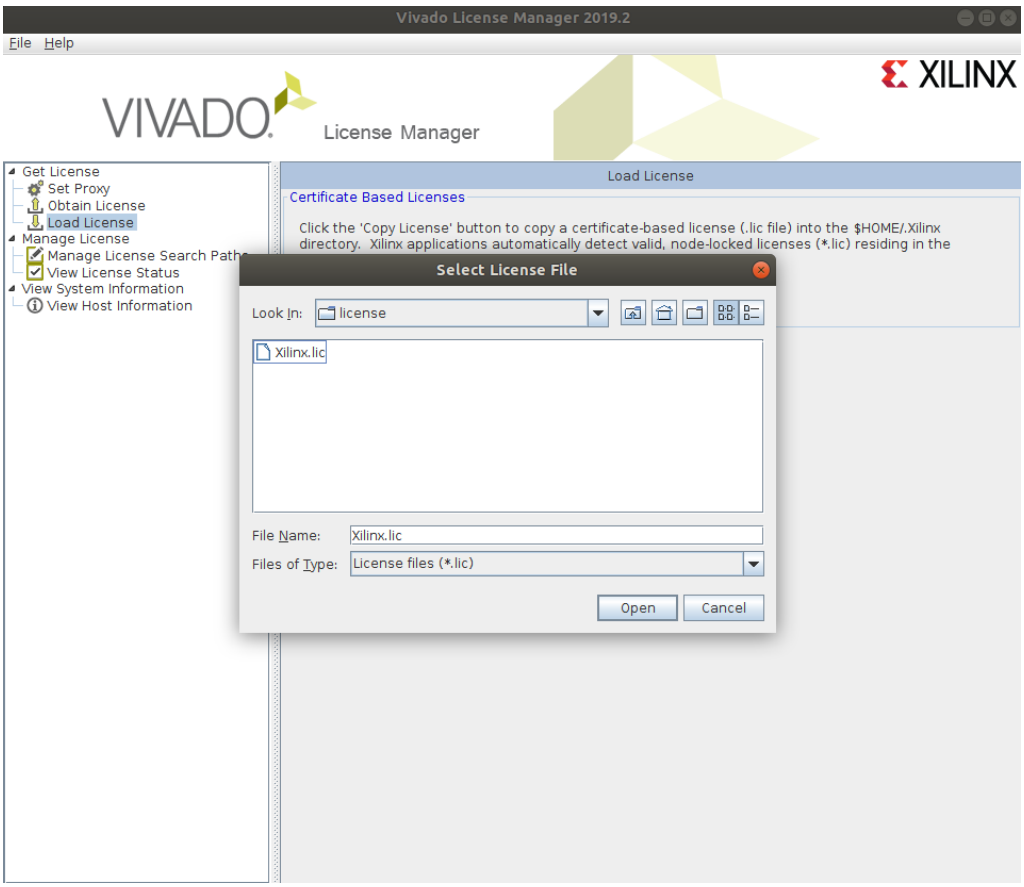


Fig. 4-4. Xilinx Vivado license installation

A free license for Xilinx Vivado HL Webpack can be obtained from the Xilinx website [XilinxLic]. License installation can be done using the Vivado License Manager program

accessible from example at C:\Xilinx\Vivado\2019.2\bin\vlm.bat in Windows and /opt/Xilinx/Vivado/2019.2/bin/vlm in Linux. Click on Load license and then Copy License and select the file Xilinx.lic obtained from the Xilinx website.

Starting Xilinx Vivado can be done in Windows by running C:\Xilinx\Vivado\2019.2\bin\vivado.bat and in Linux by first setting up the running environment with "source /opt/Xilinx/Vivado/2019.2/settings64.sh" from a terminal and then running "vivado" in the same terminal. To create a new project, click Create New Project, then click Next, introduce project name "project_1" and select the path where the project will be saved and then click Next.

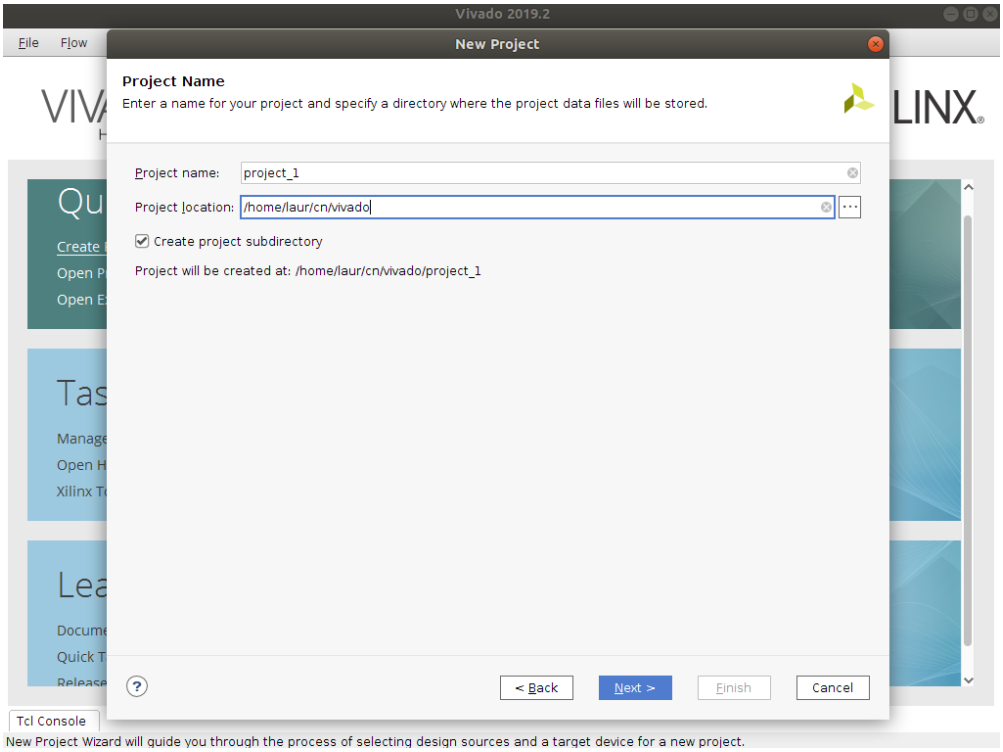


Fig. 4-5a. Create new project in Xilinx Vivado

We choose RTL project and click Next.

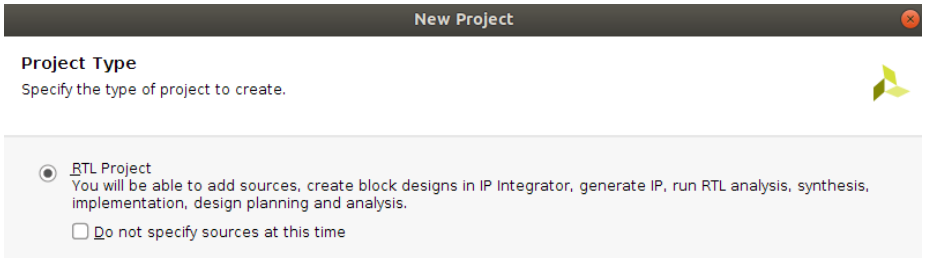


Fig. 4-5b. Create new project in Xilinx Vivado

When the windows Add sources appears, click Next. Do the same for Add existing IP and Add constraints, until the window to select FPGA family appears. Here we use the Digilent Nexys A7 FPGA board which has a XC7A100T-1CSG324C FPGA. So, we choose family: Artix 7, package: CSG324, speed grade: -3, select XC7A100TCSG324-3 and click Next and then Finish.

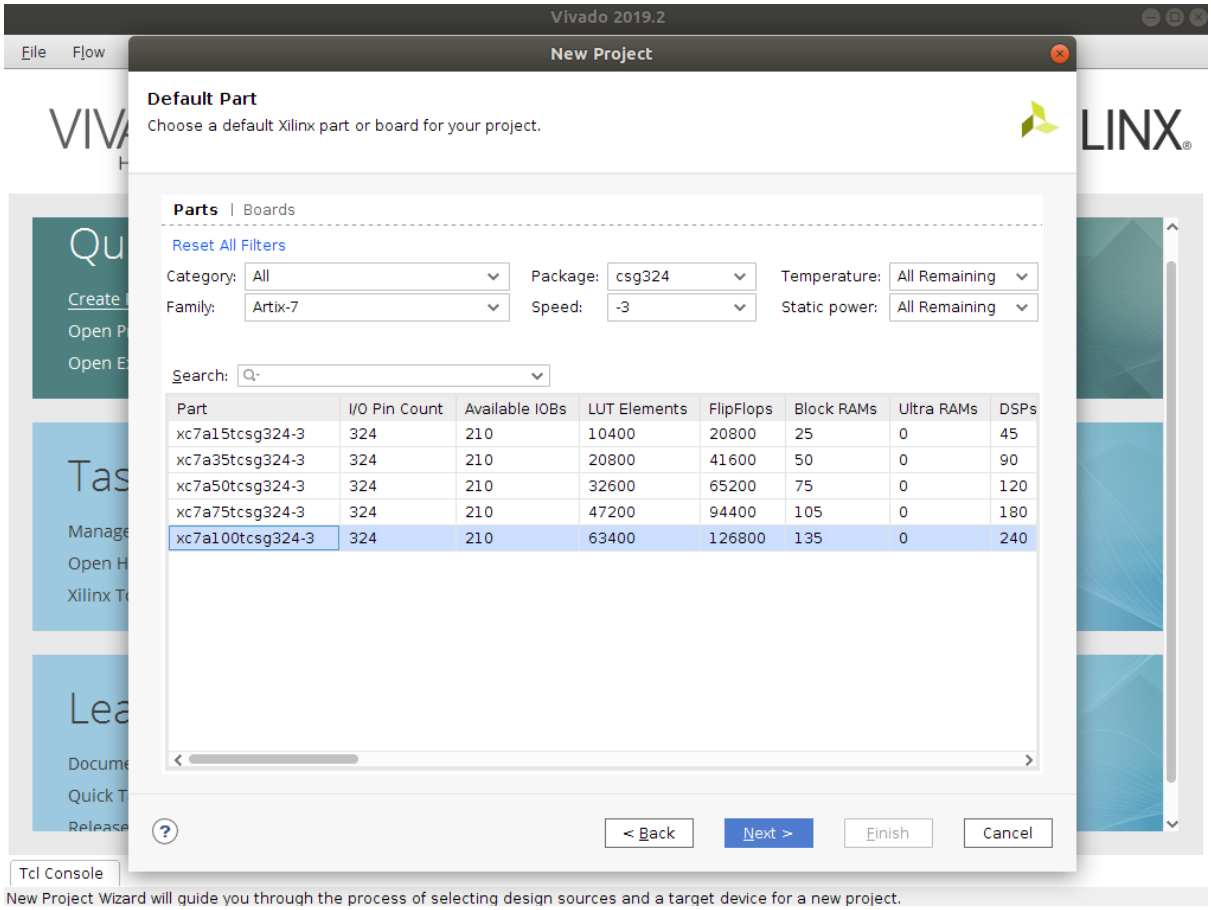


Fig. 4-5c. Selecting the FPGA family in Xilinx Vivado

To create a Verilog source module, click Design Sources and then Add Sources.

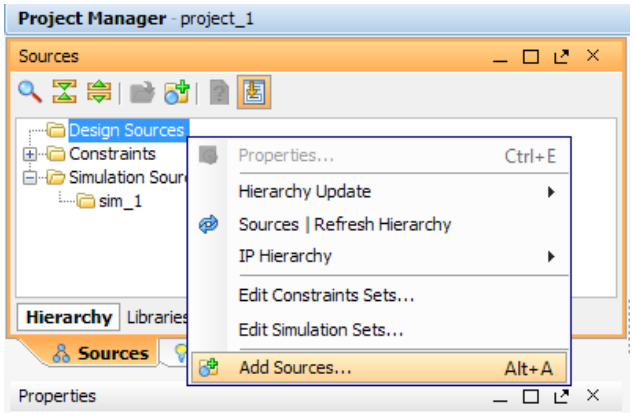


Fig. 4-6a. Creating a Verilog source module in Xilinx Vivado

In the Add Sources window select Add or create design sources and click Next.

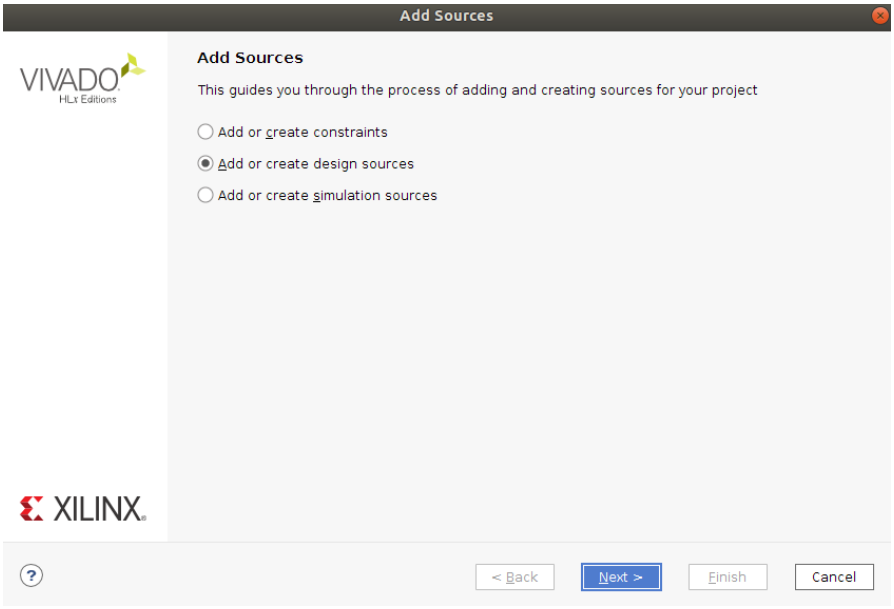


Fig. 4-6b. Creating a Verilog source module in Xilinx Vivado

Click Create File and then introduce the name of the Verilog source, m1. Click OK and Finish.

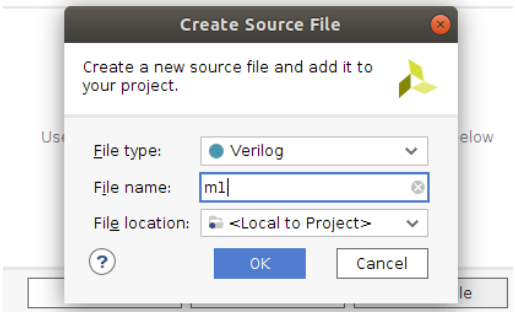


Fig. 4-6c. Creating a Verilog source module in Xilinx Vivado

In the Define Module window, click Ok and Finish.

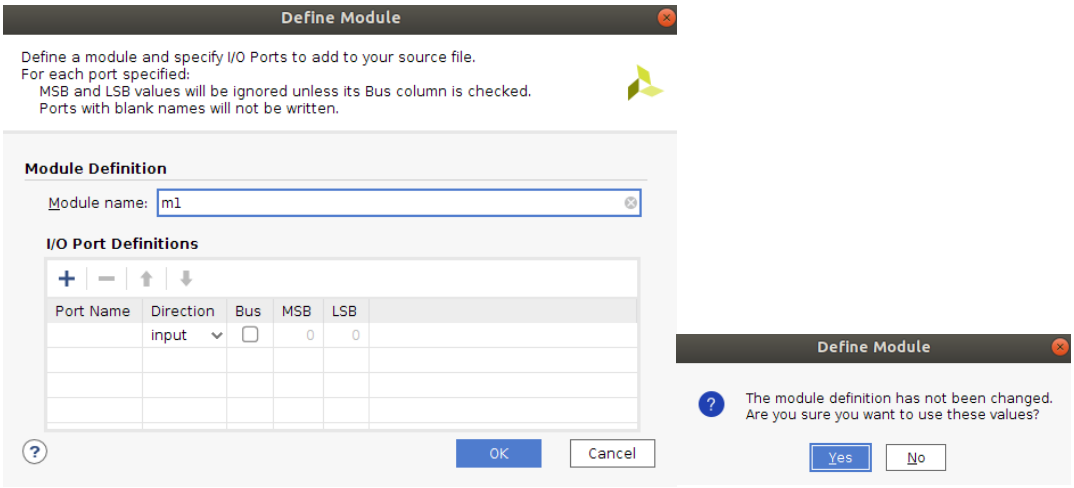


Fig. 4-6d. Creating a Verilog source module in Xilinx Vivado

To edit the new created source, double click m1.v. We edit the source like in the following figure.

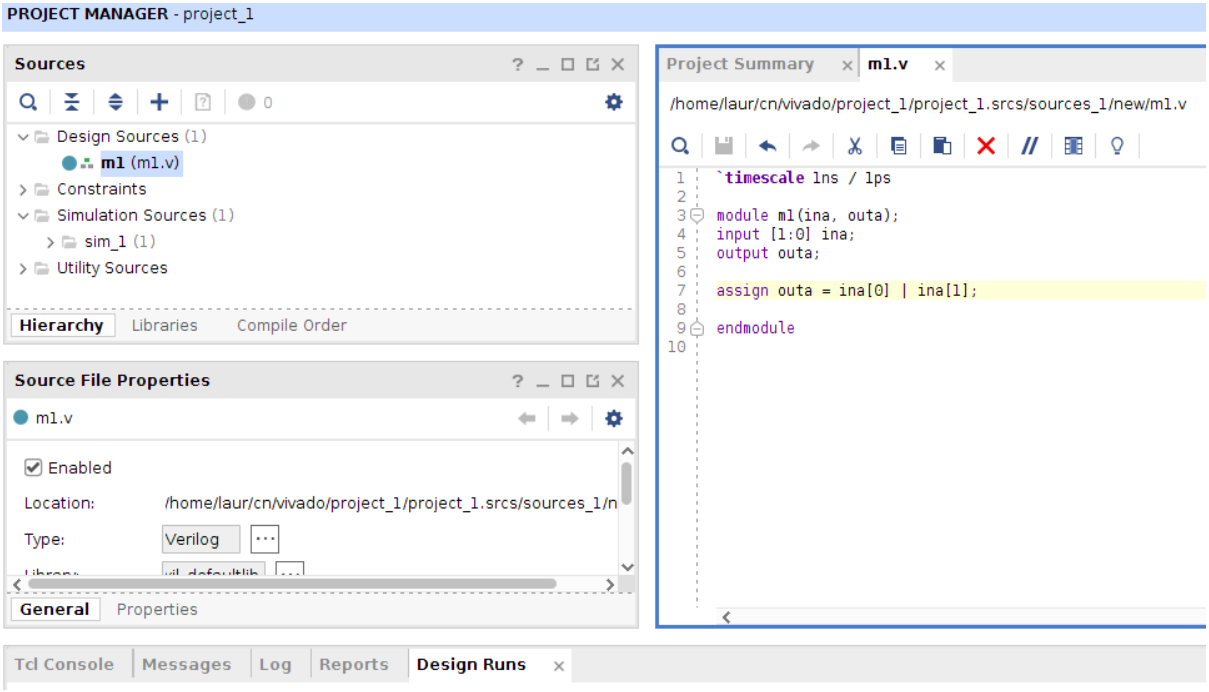


Fig. 4-6e. Creating a Verilog source module in Xilinx Vivado

To create a test bench (will talk about this later in detail), right click on Simulation Sources and select Add Sources.

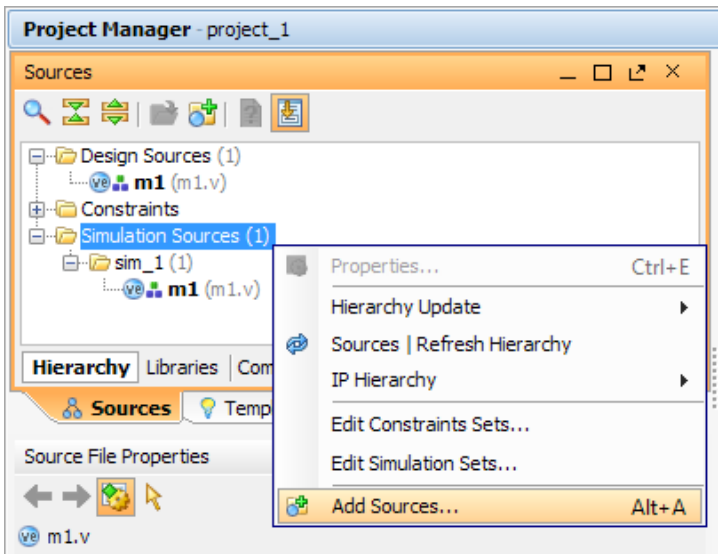


Fig. 4-7a. Creating a simulation source

In the Add Sources window, select Add or create simulations sources.

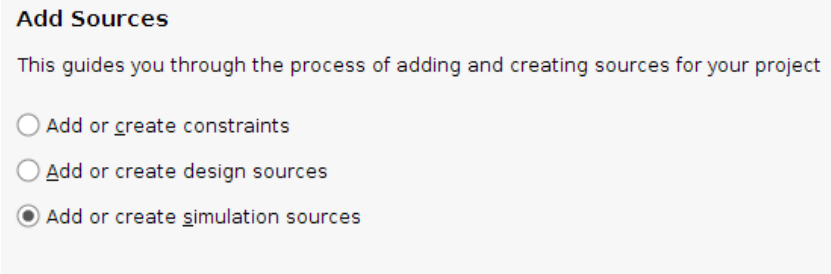


Fig. 4-7b. Creating a simulation source

Click Create File. Enter the test name as t1 and click OK, Finish.

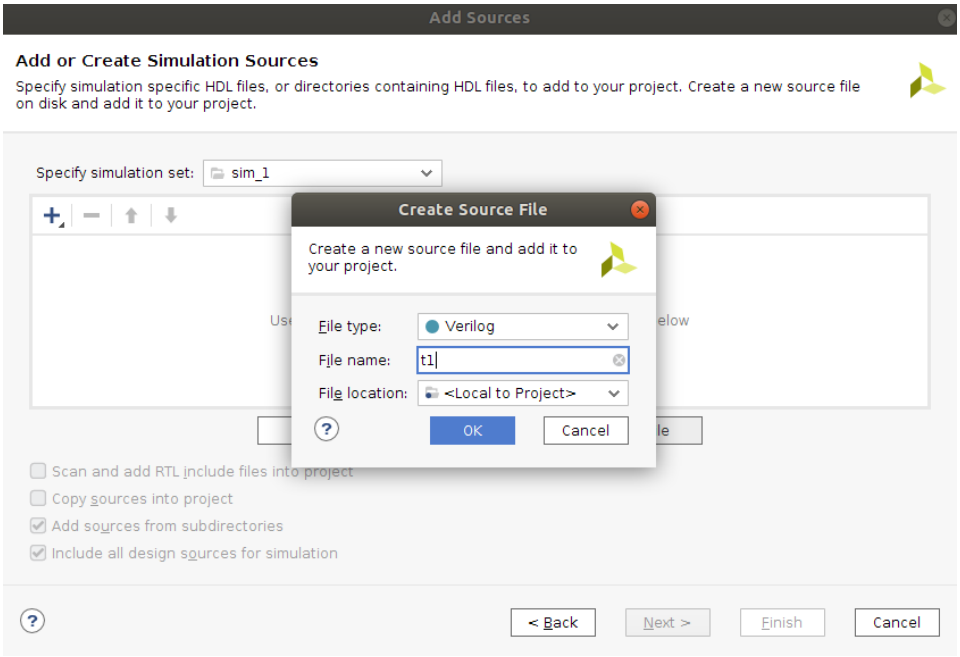


Fig. 4-7c. Creating a simulation source

In the Define Module window click OK and Yes.

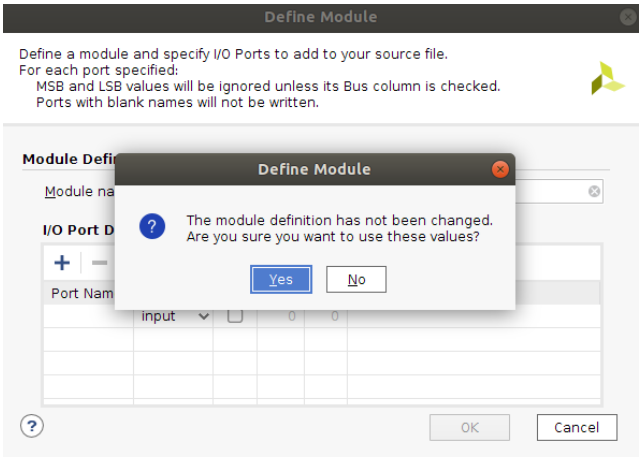


Fig. 4-7d. Creating a simulation source

Double click t1.v and edit it like in the figure below.

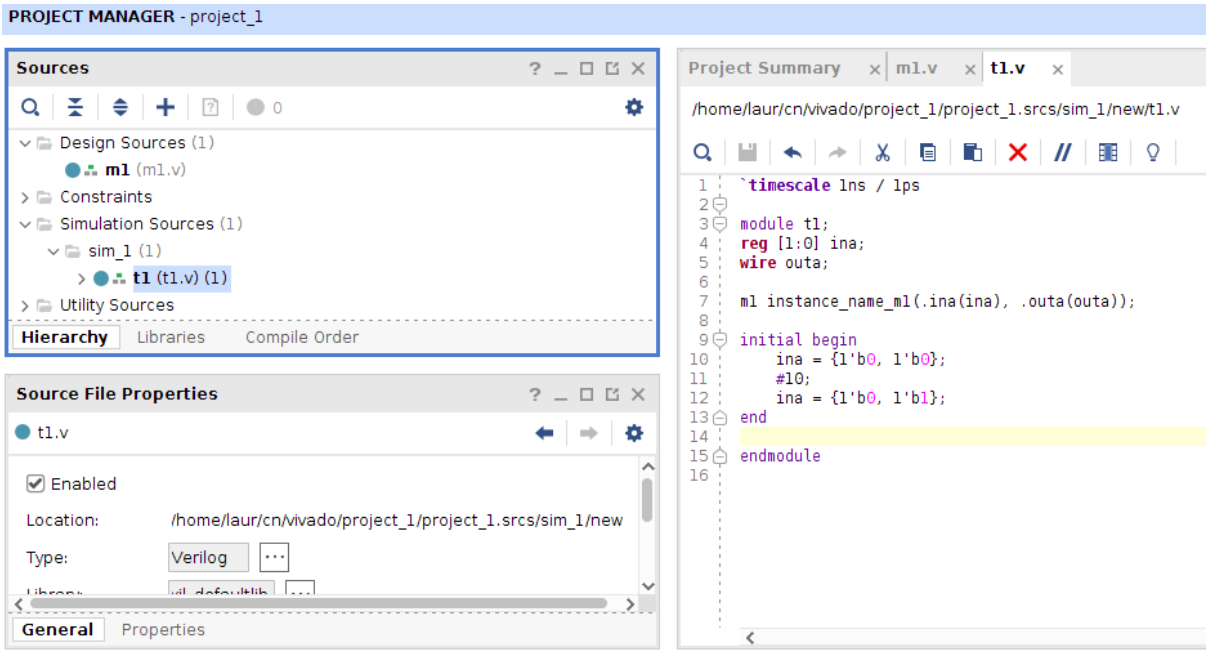


Fig. 4-7e. Creating a simulation source

We start the simulation by selecting sim_1, Run Simulation, Run Behavioral Simulation.

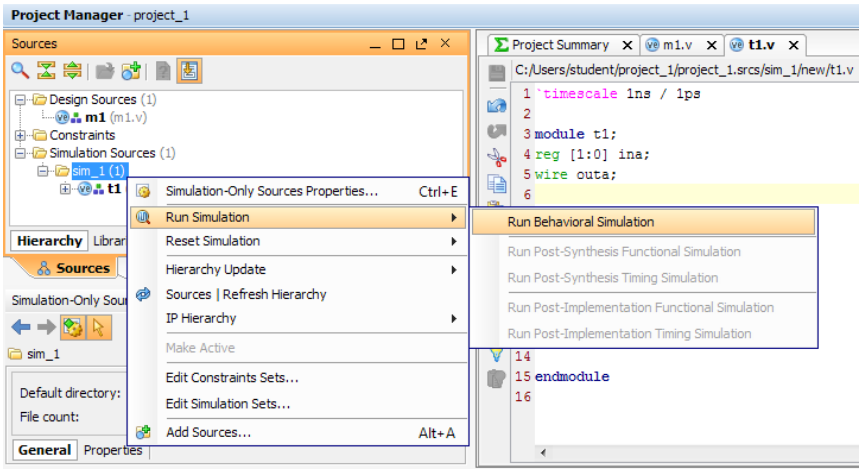


Fig. 4-8. Running the simulation

The simulation is presented in the following figure. Because in m1.v, outa was defined as (ina[0] | ina[1]) then outa will be 1 when either ina[0] or ina[1] is 1.

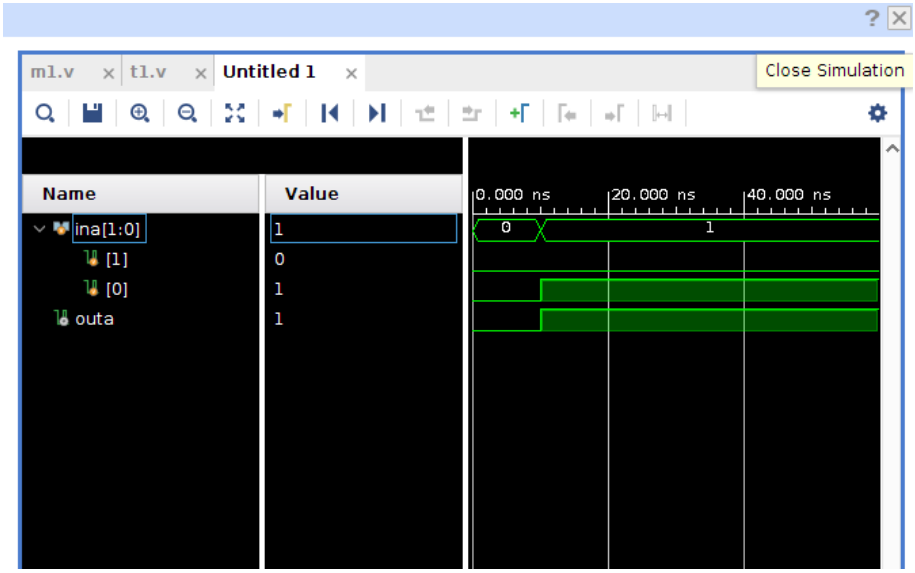


Fig. 4-9. Simulation results

To close the simulation, click on the X from the up-right corner.

To implement the design into the FPGA chip, we will assign ina[0] to the FPGA pin which is tied to switch 0, ina[1] to switch 1 and outa to led 0 on the Digilent Nexys A7 FPGA board. We must download the Xilinx design constraints file from the address https://github.com/Digilent/digilent-xdc/blob/master/Nexys-4-DDR-Master.xdc. After that, click on Design Sources, Add Sources.

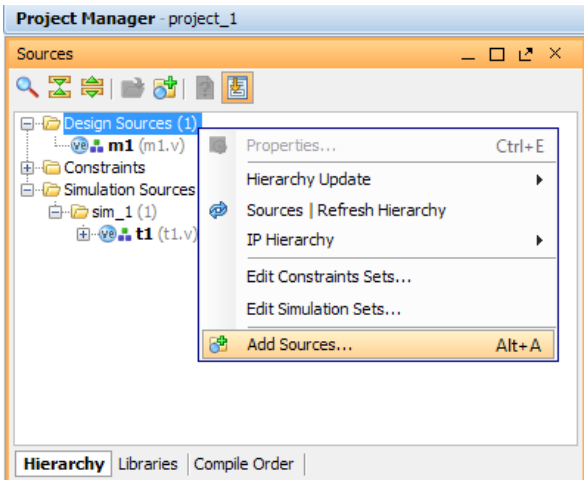


Fig. 4-10a. Creating a constraints file

Click Add or create constraints and Next.

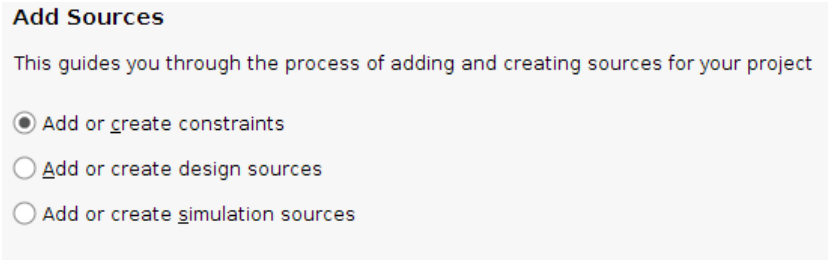


Fig. 4-10b. Creating a constraints file

Click Create File end enter c1, the name of the constraints file. Click OK, Finish.

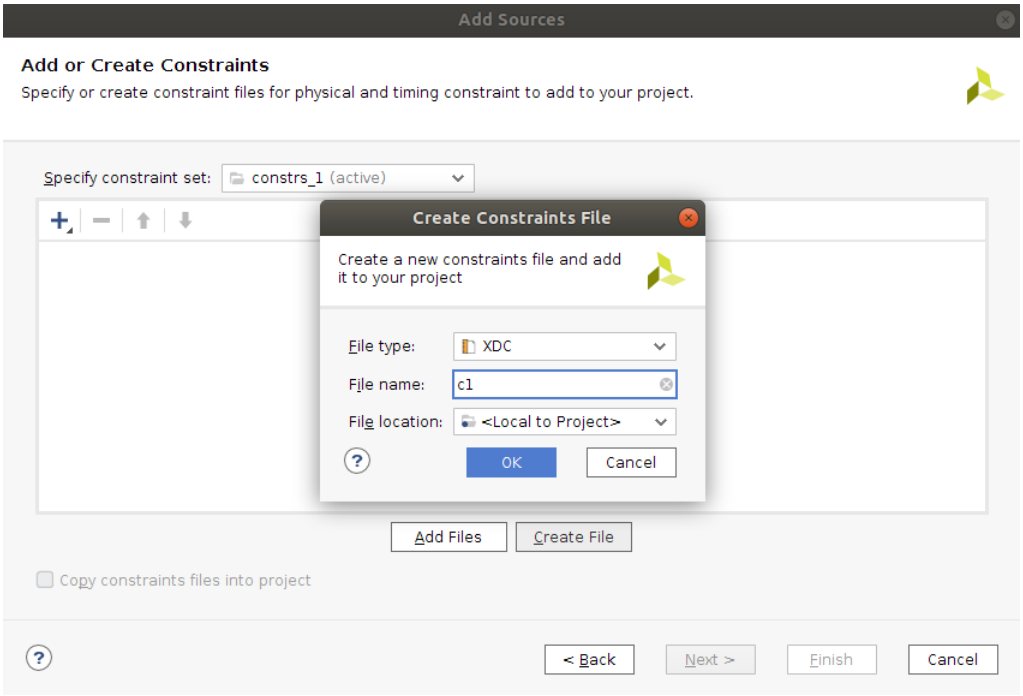


Fig. 4-10c. Creating a constraints file

Copy into c1.xdc the lines from Nexys-4-DDR-Master.xdc which contain SW[0], SW[1] and LED[0] and modify them like in the following figure (we put ina[0] instead of SW[0], ina[1] instead of SW[1] and outa instead of LED[0]).

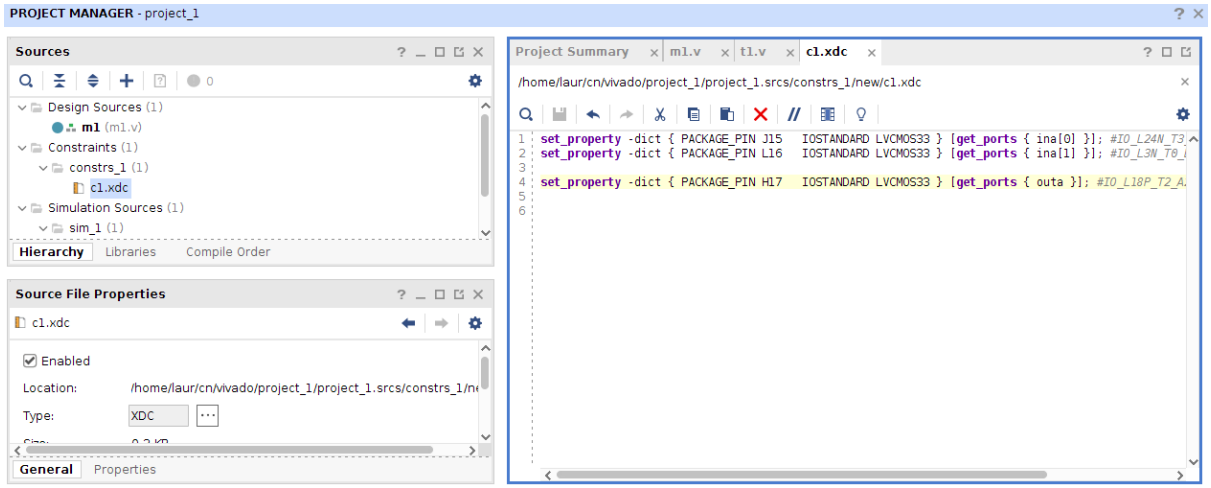


Fig. 4-11. Editing the constraints file

Now we can run the Synthesis tool. Click on m1.v and click Run Synthesis (left down corner).

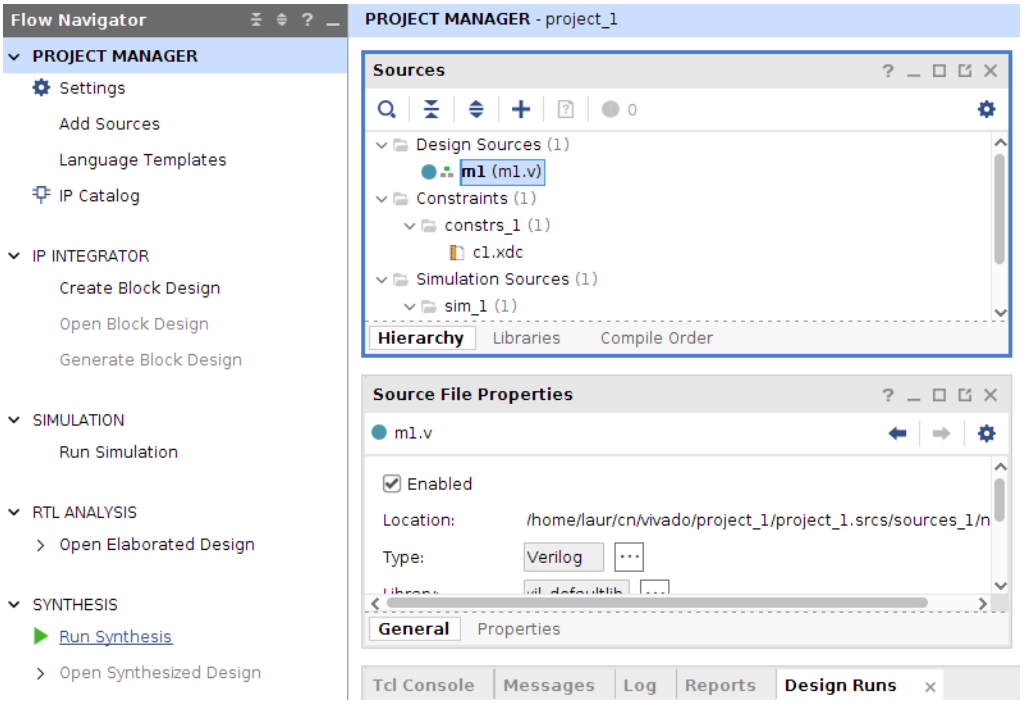


Fig. 4-12. Running Synthesis.

In the Launch Runs window, click OK.

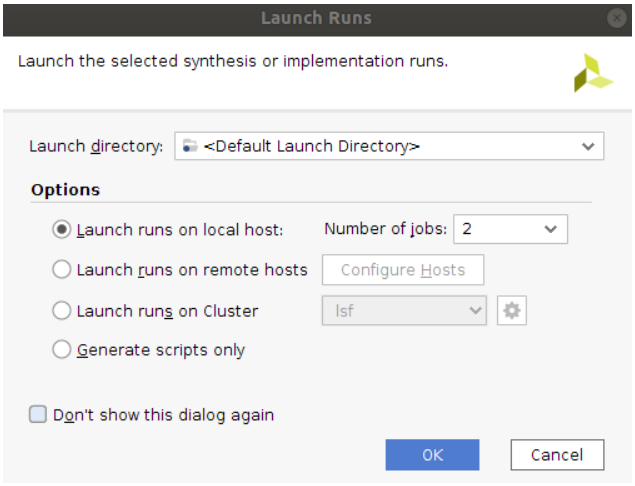


Fig. 4-13. Selecting the host to run processes

After the Synthesis process ends up successfully, we can run the Place and route process (Implementation) and then Generate the bitstream.

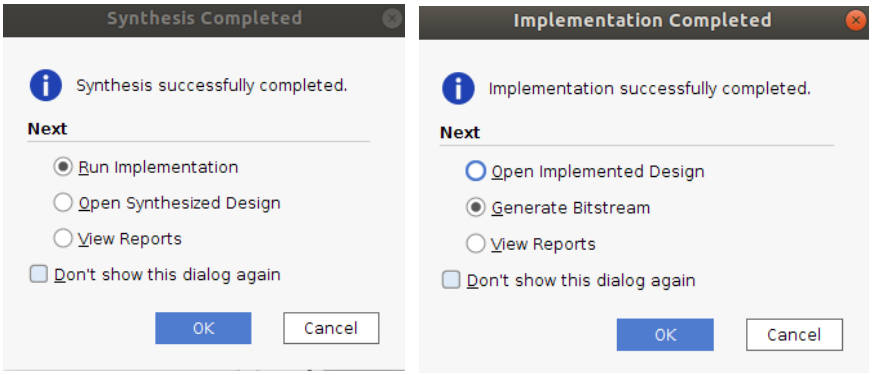


Fig. 4-14. Running Implementation and Generate Bitstream

After Generate Bitstream successfully completes it is time to open the Hardware Manager. But before that, connect the FPGA board to the host computer USB port.

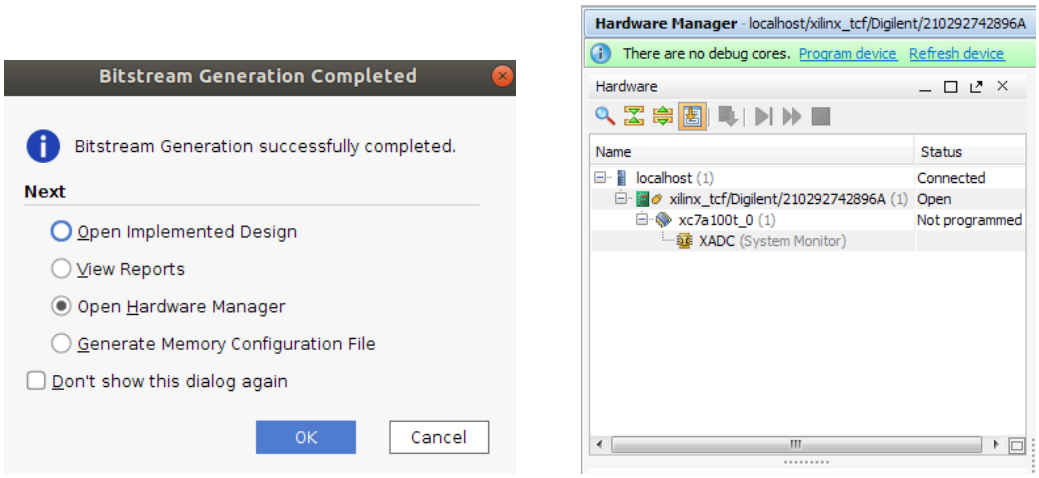


Fig. 4-15. Open the Hardware Manager

In the case of our Nexys A7 FPGA board, the blue jumper must select JTAG (center board pins). Click on Hardware Manager ➔ Open Target (left down corner) and select Autoconnect then Program device. After that, select the generated bitstream (".bit" file) and click Program. The software lets us know when programming is done.

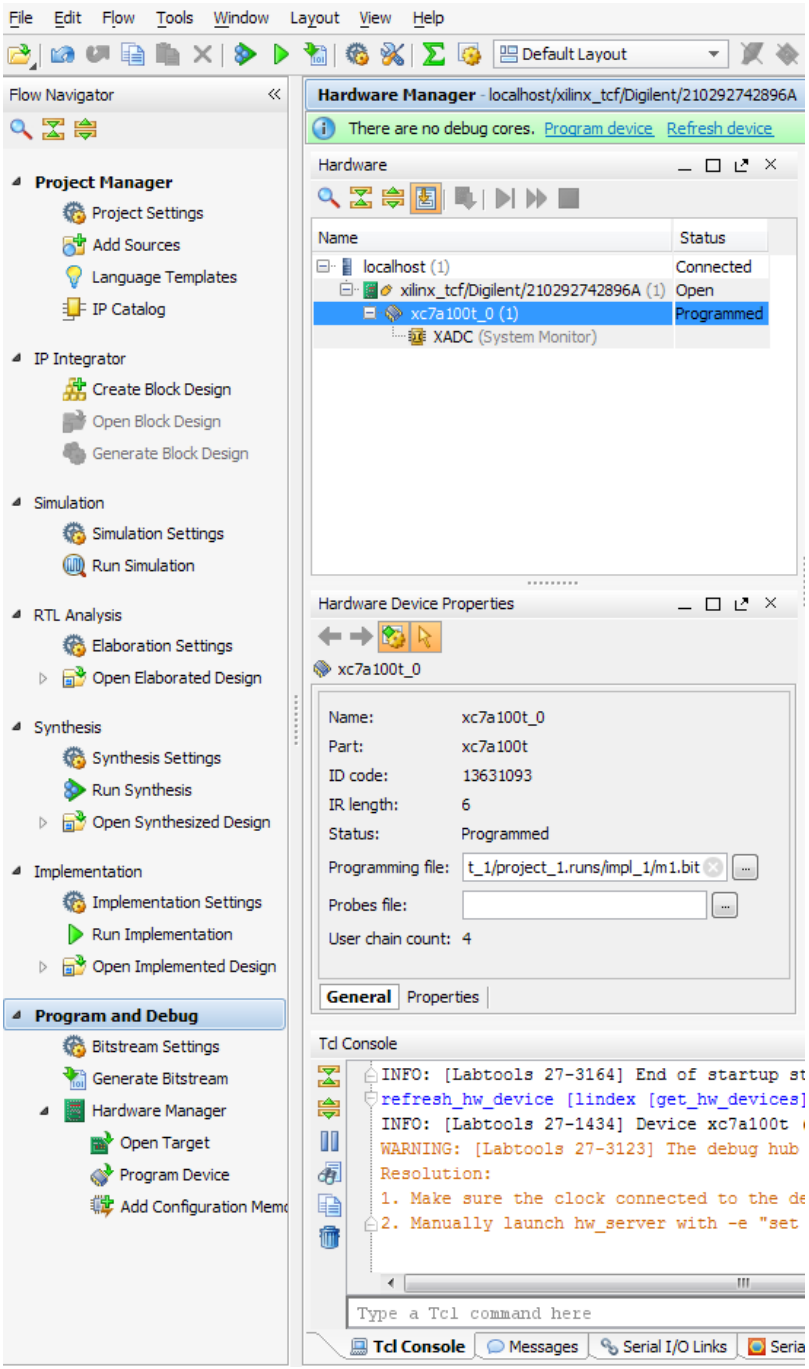


Fig. 4-16. Programming the biststream

Digilent has renamed the Nexys4 DDR board to Nexys A7. We can see that the led 0 illuminates when switch 1 or switch zero is turned on (right down corner).

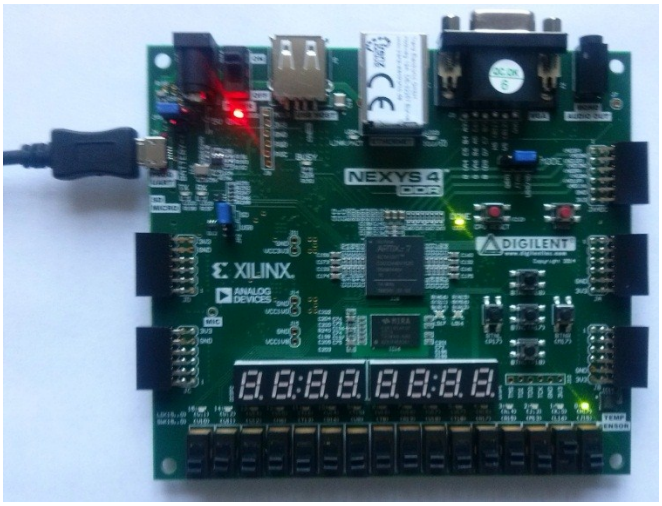


Fig. 4-18. Running the FPGA design

5. HDL essentials

5.1 Mealy and Moore finite state machines

HDL means hardware description language and it is a language dedicated for hardware specification which is similar to the software high level languages. Instead specifying the hardware scheme, one can write it into an HDL and use program tools to compile it. HDL programs are one of the following two categories: simulation code and synthesizable code. Simulation code (also called test) is not synthesizable; synthesizable code can be simulated, but through a test. When writing synthesizable code we must follow some patterns that are known to be understood by the synthesis tools – in this way there is some kind of warranty that the generated bit file will do what we want when programming it into the FPGA chip. These patterns include Mealy and Moore FSMs.

The scheme of a Mealy finite state machine (FSM) is presented in Fig. 5-1. A Mealy FSM is made by:

- registers, which are used to memorize the FSM current state during the current clock period;
- combinational logic circuits which are used to compute the FSM next state;
- output logic which is computed as a function of inputs and current state.

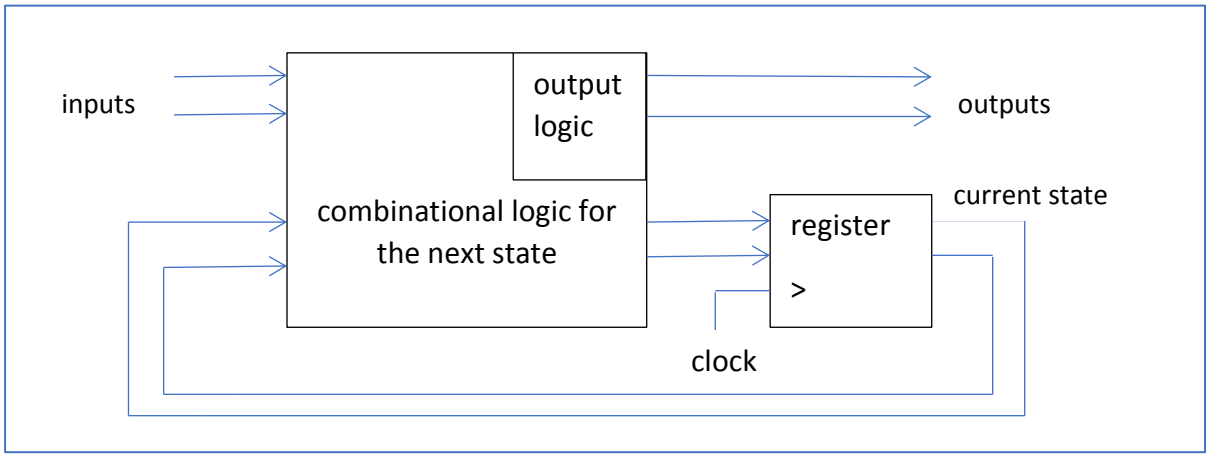


Fig. 5-1. Mealy FSM structure

The scheme of a Moore FSM is presented Fig. 5-2. A Moore FSM is made by:

- registers, which are used to memorize the FSM current state during the current clock period;
- combinational logic circuits which are used to compute the FSM next state;
- output logic which is computed only as a function the FSM current state.

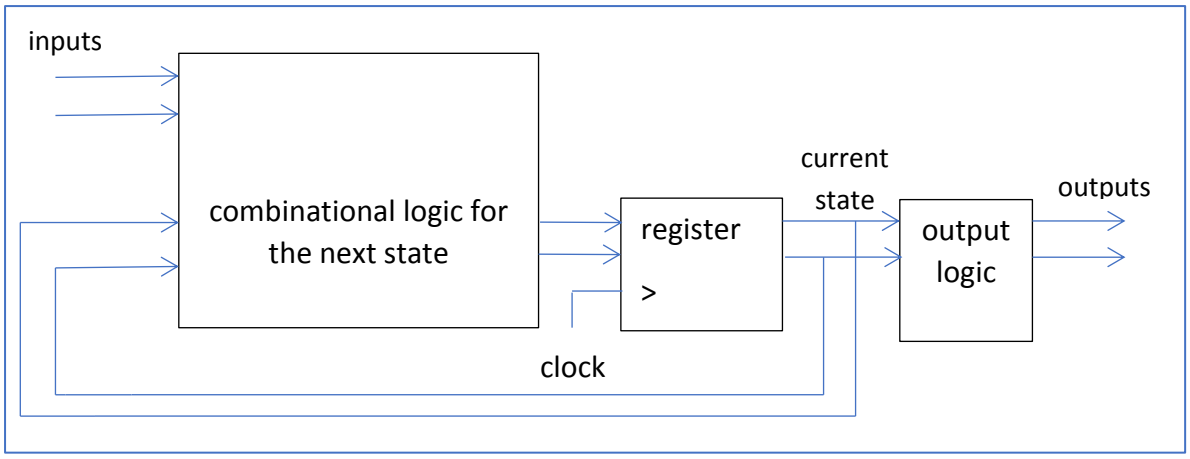


Fig. 5-2. Moore FSM structure

5.2 Application – single pulse FSM

In the following we will describe HDL programs and languages on examples. The first example is the single pulse FSM which receives as input a single-bit wide signal (called ub) which may have value 1 contiguous for several clock periods, but the single pulse FSM output (called ubsing) may be 1 only one clock period and is synchronized to the input. The logic is presented in the Fig. 5-3.

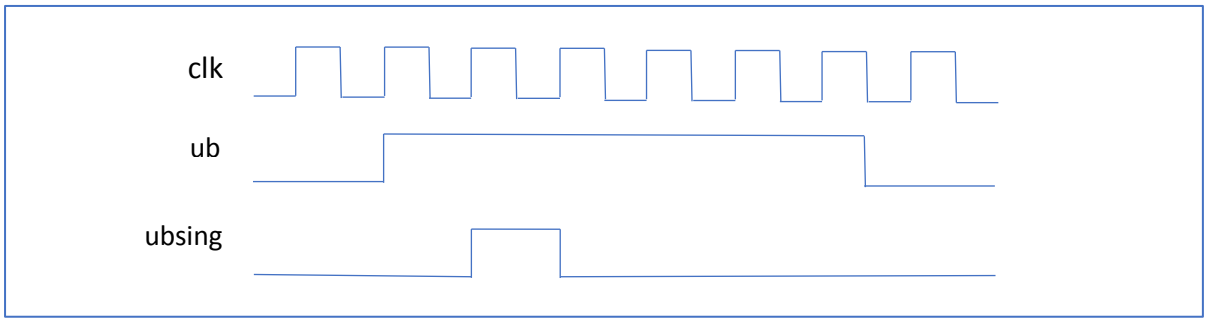


Fig. 5-3. Single pulse FSM function

5.2.1 Verilog and Icarus

Verilog is one of the famous HDL language from the industry. A good quick reference guide is available at [VerilogRef]. Synthesizable Verilog code patterns include the following:

- instances of synthesizable modules;
- any construction using the assign keyword;
- always blocks which respect a standard format, known by the synthesis tool.

Programs in Verilog are called modules. The solution module to our application starts with the following lines:

```
module single_pulse(clk, rst_l, ub, ubsing);
input clk, rst_l;
input ub;
output ubsing;

reg next_state, state;
reg ubsing_reg, next_ubsing_reg;
```

Listing 5-1a. single_pulse Verilog module

The first line contains the module name and its port list. The following declarations establish clk, rst_l and ub as inputs and ubsing as output. Then it follows the declaration of the variables that are used in the FSM. Note that the "reg" declaration it does not necessarily mean that the variables will be registers, but it means that the variables may be the left side of an always block inner assignment (will see that shortly). Also note that the variables are 1bit wide because they are not vectors.

The next line permanently assigns the output ubsing to the value of ubsing_reg:

```
assign ubsing = ubsing_reg;
```

Listing 5-1b. single_pulse Verilog module

Next comes the always block which depends on clock and reset. The always block respects the pattern for register inferring by the synthesis tool for the state and ubsing_reg variables. "<=" means attribution. Please note that on reset (which is active low) the variables are inititalized to 0; the reset sequence takes place on the falling edge of rst_l. "posedge clk" means that the registers will make the attribution "out<=in" on the rising edge of the clock.

```
always @(posedge clk or negedge rst_l)
begin
        if (~rst_l) begin
                        state <= 0;
                        ubsing_reg <= 0;
        end else begin
                        state <= next_state;
                        ubsing_reg <= next_ubsing_reg;
        end
end
```

Listing 5-1c. single_pulse Verilog module

Next comes another always block which represents the combinational logic which computes the next state of the FSM. See Fig. 5-4d. The sensitivity list of the always block is (*) which means that whenever a variable of a right side of an attribution inside the always block changes, the always block will be executed (this happens in simulation too). The first two assignments establish the implicit values for next_state (which is the value of the state register) and next_ubsing_reg (which is 0); if in the case-sequence there is no attribution on any of these variables, then the (missing) variable's value will be the implicit one.

```
always @(*)
begin
                next_state <= state;
                next_ubsing_reg <= 0;
                case (state)
                0: if (ub == 1) begin
                                next_state <= 1;
                                next_ubsing_reg <= 1;
                        end
                1: if (ub == 0)
                                next_state <= 0;
                endcase
end
endmodule
```

Listing 5-1d. single_pulse Verilog module

The logic is simple: if in state 0 ub becomes 1, then the next state will be 1 and the ubsing_reg will become 1 at the next posedge clk. In state 1, ubsing_reg is 0 and we are waiting for ub to become 0 (remember that ub was 1 when we changed the state); immediately after this we go in state 0. This will be illustrated in simulation.

This is a Mealy FSM, because ubsing output depends on the state and the ub input.

The same behavior would be obtained if we use assign instructions for next_state and next_ubsing_reg instead of the combinational always block, but in this case we must declare these variables as wire. Also, remark that assign uses "=".

```
wire next_state;
wire next_ubsing_reg;

assign next_state = (state == 0) ? ((ub == 1) ? 1 : 0) : ((ub == 0) ? 0 : 1);
assign next_ubsing_reg = ((state == 0) && (ub == 1)) ? 1 : 0;
```

Listing 5-1e. single_pulse Verilog module

In order to implement the solution with a Moore FSM, we must make the output ubsing as a function of state only. Also, state and next_state must be vectors, because we need more than two states.

```
reg [1:0] state, next_state;

always @(*)
begin
                next_state <= state;
                next_ubsing_reg <= 0;
                case (state)
                0: if (ub == 1)
                                next_state <= 1;
                1: begin
```

```
                    next_ubsing_reg <= 1;
                    if (ub == 0)
                                next_state <= 0;
                          else
                                next_state <= 2;
                end
                2: if (ub == 0)
                                next_state <= 0;
                endcase
end
```

Listing 5-2. single_pulse Verilog module with Moore FSM

In order to simulate the single_pulse module we must create a test module which will instantiate the tested module. The test module is shown in the figure below. The first line is the timescale directive which establishes the delay interval of the instruction preceded by the "#" character. Next is the instantiation of the tested module (note that ubsing is wire in the test because is tied to the ubsing output of the single_pulse module) and the "initial" sequence which describes the simulation stimulus. Also remark the clock signal simulator from the always block which, in this case, from 5 to 5ns inverts the value of the clock signal.

```
`timescale 1ns / 1ps

module test;
reg clk, rst_l;
reg ub;
wire ubsing;

single_pulse sp1(.clk(clk), .rst_l(rst_l), .ub(ub), .ubsing(ubsing));

always #5 clk = ~clk;

initial begin
        ub = 0;
        clk = 0;
        rst_l = 0;
        #25;
        rst_l = 1;
        #30;
        ub = 1;
        #50;
        ub = 0;
end

`ifdef __ICARUS__
 initial
 begin
        $dumpfile("single_pulse.vcd");
```

```
        $dumpvars(0, test);
        #200;
        $finish;
 end
`endif

endmodule
```

Listing 5-3. single_pulse Verilog module test

The simulation was made using Icarus Verilog simulator and gtkwave waveform viewer. Icarus dumps a file "single_pulse.vcd" which contains the transitions of all the signals from module test and modules instantiated by test. "$finish" ends simulation (here after 200ns). Gtkwave is launched with the command "gtkwave single_pulse.vcd". Fig. 5-4 shows the simulation for the Mealy FSM while Fig. 5-5 shows the simulation for the Moore FSM.

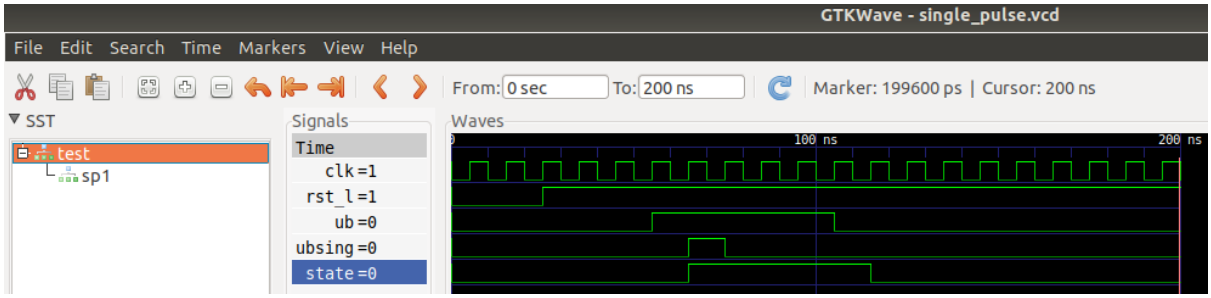


Fig. 5-4. single_pulse Mealy FSM simulation

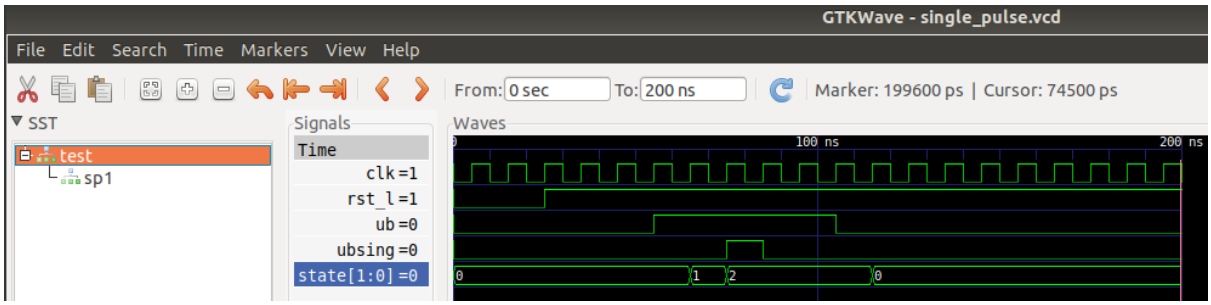


Fig. 5-5. single_pulse Moore FSM simulation

For Icarus and gtkwave installation on Ubuntu Linux just run the commands:

```
sudo apt-get install iverilog gtkwave
```

Listing 5-4. Installing Icarus and gtkwave on Ubuntu Linux

Icarus and gtkwave in Linux can be used from a terminal with the commands from Listing 5-5.

```
iverilog -o single_pulse single_pulse.v test.v
./single_pulse
gtkwave single_pulse.vcd &
```

Listing 5-5. Using Icarus and gtkwave on Linux

In gtkwave, drag and drop signals from SST to Signals tab to view them in simulation.

For Windows, download and install the Icarus kit which incorporates gtkwave from [IcarusBleyer]. We suppose that the installation is made in c:\iverilog path. here are the commands to use Icarus and gtkwave on Windows:

```
C:\iverilog\bin\iverilog.exe -o simplevvp single_pulse.v test.v
C:\iverilog\bin\vvp.exe simplevvp
C:\iverilog\gtkwave\bin\gtkwave.exe single_pulse.vcd
```

Listing 5-6. Using Icarus and gtkwave on Windows

In gtkwave after setting which signals should appear in simulation, one can use the commands Write Save File (Ctrl-S) to save the list of shown simulation signals and Read Save File (Ctrl-O) to read the list of signals to be included in the simulation. This way simplifies using gtkwave successively in multiple runs.

5.2.2 VHDL and ghdl

VHDL is also one of the famous HDL language in industry. Its reference guide is available at [VHDLRef]. As with Verilog, VHDL synthesizable constructions include:

- instances of entities and components whose architecture is synthesizable;
- any stand-alone (not in process) construction using the "<=" operator;
- process blocks which respect a standard format, known by the synthesis tool.

In the following, we will show how to implement in VHDL a solution for the current application. First, we include the IEEE library which describes the definitions of logic values to be used in our source:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
```

Listing 5-7a. single_pulse VHDL source

Then it follows the entity declaration which defines the ports of our source.

```
entity single_pulse is
port(    clk, rst_l: in std_logic;
                ub: in std_logic;
                ubsing: out std_logic
);
end single_pulse;
```

Listing 5-7b. single_pulse VHDL source

Then it follows the architecture implementation of the single_pulse entity.

```
architecture single_pulse_arch of single_pulse is

  signal next_state, state: std_logic;
  signal ubsing_reg, next_ubsing_reg: std_logic;
```

```
begin
        ubsing <= ubsing_reg;
```

Listing 5-7c. single_pulse VHDL source

In the architecture are declared the state logic variables and ubsing_reg value is assigned to ubsing.

Then it follows the process that defines the registers state and ubsing_reg. Similar to Verilog, rst_l is active on 0 and the register attribution is done on the clock rising edge.

```
state_reg: process(clk, rst_l)
begin
                if (rst_l='0') then
                        state <= '0';
                        ubsing_reg <= '0';
                elsif (rising_edge(clk)) then
                                state <= next_state;
                                ubsing_reg <= next_ubsing_reg;
                end if;
end process;
```

Listing 5-7d. single_pulse VHDL source

The combinational logic for computing the next state ends up the single_pulse architecture implementation. Here, the process is called whenever state or ub changes. Similar to Verilog, the first two assignments establish the implicit values for next_state and next_ubsing_reg and then the case sequence establishes the FSM behavior (which was described in the previous Verilog section). Note that "--" defines a VHDL line comment.

```
comb_logic: process(state, ub)
        begin
                next_state <= state;
                next_ubsing_reg <= '0';
                case state is
                        when '0' =>
                                if (ub = '1') then
                                        next_state <= '1';
                                        next_ubsing_reg <= '1';
                                end if;
                        when '1' =>
                                if (ub = '0') then
                                        next_state <= '0';
                                end if;
                        when others =>
                                -- this is forced by the vhdl compiler
                                --next_state <= '0';
                                --next_ubsing_reg <= '0';
```

```
                end case;
  end process;

end single_pulse_arch;
```

Listing 5-7e. single_pulse VHDL source

This was the Mealy VHDL FSM implementation. For Moore FSM, the state and next_state variables must be vectors:

```
signal next_state, state: std_logic_vector(1 downto 0);
```

Listing 5-8a. single_pulse VHDL source for Moore FSM

And in the processes that use these variables, we work with vector constants:

```
comb_logic: process(state, ub)
        begin
                next_state <= state;
                next_ubsing_reg <= '0';
                case state is
                        when "00" =>
                                if (ub = '1') then
                                        next_state <= "01";
                                end if;
                        when "01" =>
                                next_ubsing_reg <= '1';
                                if (ub = '0') then
                                        next_state <= "00";
                                else
                                        next_state <= "10";
                                end if;
                        when "10" =>
                                if (ub = '0') then
                                        next_state <= "00";
                                end if;
                        when others =>
                                -- this is forced by the vhdl compiler
                                --next_state <= “00”;
                                --next_ubsing_reg <= '0';
                end case;
```

Listing 5-8b. single_pulse VHDL source for Moore FSM

The VHDL simulation test begins with IEEE library inclusion and entity declaration.

```
library  ieee;
use ieee.std_logic_1164.all;

entity single_pulse_tb is
end single_pulse_tb;
```

Listing 5-8b. single_pulse VHDL source testbench

Then it follows the architecture implementation of the single_pulse_tb, where the single_pulse component is declared. In the body of the architecture, the single_pulse unit is instantiated.

```
architecture TB of single_pulse_tb is

signal T_clk: std_logic;
signal T_rst_l:  std_logic;
signal T_ub: std_logic;
signal T_ubsing: std_logic;
signal stop_simulation: std_logic := '0';

component single_pulse
port(    clk:              in std_logic;
                rst_l:          in std_logic;
                ub:                     in std_logic;
                ubsing:         out std_logic
);
end component;
begin

  U_single_pulse:  single_pulse  port  map(rst_l=>T_rst_l,  clk=>T_clk,  ub=>T_ub,
ubsing=>T_ubsing);
```

Listing 5-8c. single_pulse VHDL source testbench

Then the clock simulation process and the stimulus implementation processes are implemented. Instructions are self explanatory.

```
  process
  begin
                if stop_simulation = '0' then
                        T_clk <= '1';                   -- clock cycle 10 ns
                        wait for 5 ns;
                        T_clk <= '0';
                        wait for 5 ns;
                else
                        wait;
                end if;
        end process;

  process
  begin
        T_rst_l <= '0';
        T_ub <= '0';
        wait for 20 ns;
        T_rst_l <= '1';
```

```
        wait for 30 ns;
        T_ub <= '1';
        wait for 50 ns;
        T_ub <= '0';
        wait for 50 ns;

        -- stop simulation.
        --assert false report "end of simulation" severity failure;
        stop_simulation <= '1';
        wait;

   end process;

end TB;
```

Listing 5-8d. single_pulse VHDL source testbench

The testbench ends with defining the configuration for the architecture.

```
configuration CFG_TB of single_pulse_tb is
        for TB
        end for;
end CFG_TB;
```

Listing 5-8e. single_pulse VHDL source testbench

The simulation of the Mealy FSM is shown in the figure below.

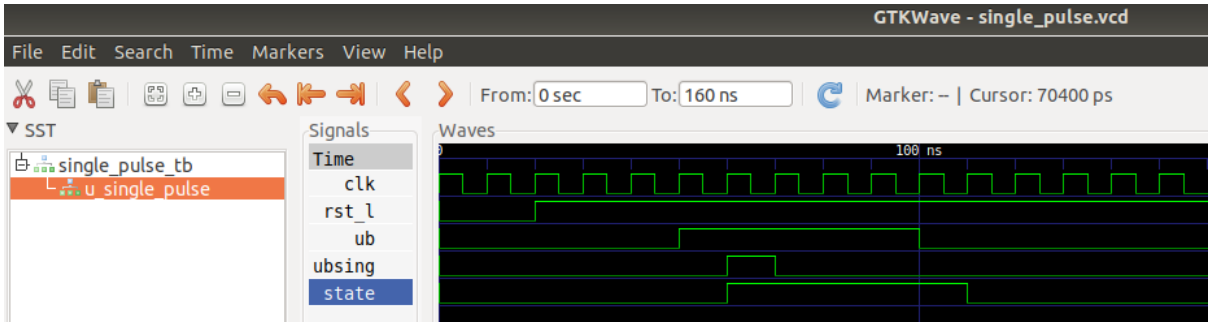


Fig. 5-6. single_pulse Mealy FSM VHDL simulation

The simulation of the Moore FSM is shown in the next figure.

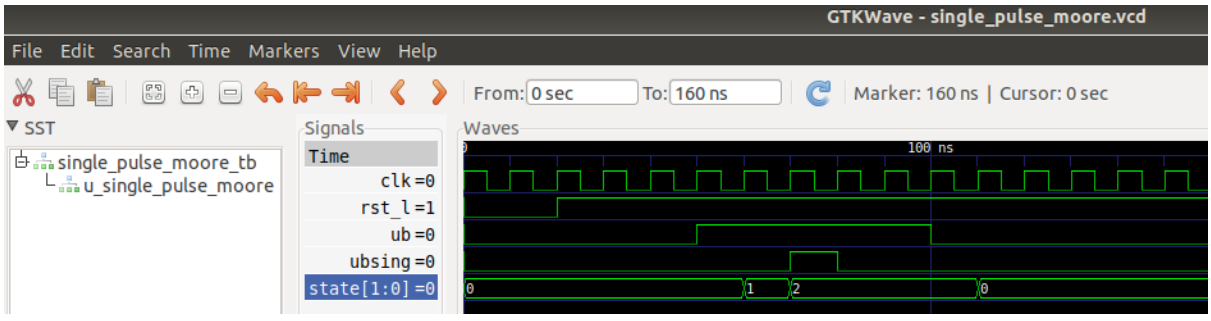


Fig. 5-6. single_pulse Moore FSM VHDL simulation

The simulation was made using the ghdl VHDL simulator. This simulator is free and available for Windows, MacOS and Linux. It can be downloaded from the github website [GhdlGit] as source or releases kit packages. Please notice that ghdl works in conjunction with gtkwave (which on Windows is installed when one installs Icarus – see the Verilog chapter).

Installing ghdl on Ubuntu Linux from source can be done with the following script:

```
sudo apt install llvm
sudo apt install gnat
git clone https:/github.com/ghdl/ghdl
cd ghdl
   ./configure --prefix=/usr/local
   make
   sudo make install
```

Listing 5-9. Ghdl installation on Ubuntu Linux

Compiling and running a simulation using ghdl on Linux can be done with the following script (in Windows is similarly) which can be launched from a terminal as (it works because the test entity name is single_pulse_tb): "./ghdl-script.sh single_pulse".

```
#!/bin/bash
set -x
ghdl --clean
rm *.cf *.vcd *.ghw
echo $1
set -e
ghdl -a $1.vhd $1_tb.vhd
ghdl -e $1_tb
ghdl -r $1_tb --vcd=$1.vcd
gtkwave $1.vcd $1.vcd.savefile &
```

Listing 5-10. Running a VHDL simulation with ghdl on Linux

On Windows one must first install to "C:" mingw64 (gcc port to Windows), add its bin folder to the PATH system variable and then copy the ghdl binaries to an accessible folder. Then with the following ghdl script the simulation can be run: "./ghdl-script.bat single_pulse"

```
echo %1
del *.cf *.vcd *.ghw
c:\ghdl\bin\ghdl.exe --clean
c:\ghdl\bin\ghdl.exe -a %1.vhd %1_tb.vhd
c:\ghdl\bin\ghdl.exe -e %1_tb
c:\ghdl\bin\ghdl.exe -r %1_tb --vcd=%1_tb.vcd
c:\iverilog\gtkwave\bin\gtkwave.exe %1_tb.vcd %1.vcd.savefile
```

Listing 5-11. Running a VHDL simulation with ghdl on Windows

5.3 Application – simple multiplication unit

In the following, will implement a simple multiplication unit for positive (unsigned) numbers. Let's consider the following example: the multiplicand m=5, the multiplier r=6, the product prod=5*6=30. The product can be computed as in the following figure:

<table>
<tr>
<td>0101*
0110

0000
0101
0101
0000

0011110</td>
<td>Algorithm 1.
prod=0;
a=m;
if(r != 0) begin
if(r[0])
prod += a;
a = a << 1;
r = r >> 1;
end</td>
<td>Algorithm 2.
prod=0;
a=m;
for(i=0; i<r; i++)
prod += a;</td>
</tr>
</table>

Fig. 5-7. Simple multiplication algorithms

In Fig. 5-7 are shown two multiplication algorithms. Algorithm 1 has complexity proportional to how many binary digits has r, while algorithm 2 has complexity of r value. Definitely, for large numbers, algorithm 1 has lower complexity. So, we choose to implement algorithm 1.

5.3.1 Verilog implementation

The multiplication module has the following interface:

```
module mul(clk, rst, m, r, prod, start, done);

parameter N_BITS=4;

input clk, rst, start;
input [N_BITS-1:0] m, r;
output done;
output [2*N_BITS-1:0] prod;
reg [2*N_BITS-1:0] prod, a, next_prod, next_a;
reg state, next_state;
reg [N_BITS-1:0] raux, next_raux;
wire done;
```

Listing 5-12a. The Verilog multiplication module

The register code is shown in the following listing. The initial state is 1.

```
always @(posedge clk or posedge rst)
begin
        if(rst) begin
                prod <= 0;
```

```
                state <= 1;
                a <= 0;
                raux <= 0;
        end else begin
                state <= next_state;
                prod <= next_prod;
                a <= next_a;
                raux <= next_raux;
        end
end
```

Listing 5-12b. The Verilog multiplication module

The next state and output logic are computed in the following sequence. Note that Verilog syntax is similar to C. The only different instruction is the next_a logic computation in state 1, "next_a <= {{N_BITS{1'b0}}, m};". Here "{}" means concatenation, "1'b0" means one-bit logic value 0 expressed in base 2 and "{N_BITS{1'b0}}" means 0 value written on N_BITS.

```
always @(*)
begin
        next_a <= a;
        next_prod <= prod;
        next_state <= state;
        next_raux <= raux;
        case(state)
        0:if(raux != 0) begin
                next_raux <= raux >> 1;
                if(raux[0])
                        next_prod <= prod + a;
                next_a <= a << 1;
        end else
                next_state <= 1;
        1: if(start) begin
                next_state <= 0;
                next_a <= {{N_BITS{1'b0}}, m};
                next_prod <= 0;
                next_raux <= r;
        end
        endcase
end
```

Listing 5-12c. The Verilog multiplication module

The done variable has the following logic:

```
assign done = (state == 1);
```

Listing 5-12d. The Verilog multiplication module

The test module instantiates the mul module like in the following listing:

```
module testmul;

	// Inputs
	reg clk, rst, start;
	reg [3:0] m, r;
	// Outputs
	wire [7:0] prod;
	wire done;

	// Instantiate the Unit Under Test (UUT)
	mul #(4) uut (
		.clk(clk), .rst(rst),
		.m(m), .r(r), .prod(prod),
		.start(start), .done(done));
```

Listing 5-13a. The Verilog test module

The clock oscillator and stimulus is shown in the following listing:

```
	always #5 clk = ~clk;

	initial begin
		// Initialize Inputs
		clk = 0; rst = 1; m = 5; r = 6; start = 0;
		// Wait for global reset to finish
		#25;
		rst = 0;
		#10;
		// Add stimulus here
		start = 1;
		#20;
		start = 0;
	end
endmodule
```

Listing 5-13b. The Verilog test module

The simulation is shown in the following figure. Here is shown how raux is right shifted with one bit and a is left shifted with one bit at each algorithm iteration. Also, a is added to prod if raux[0] is one.

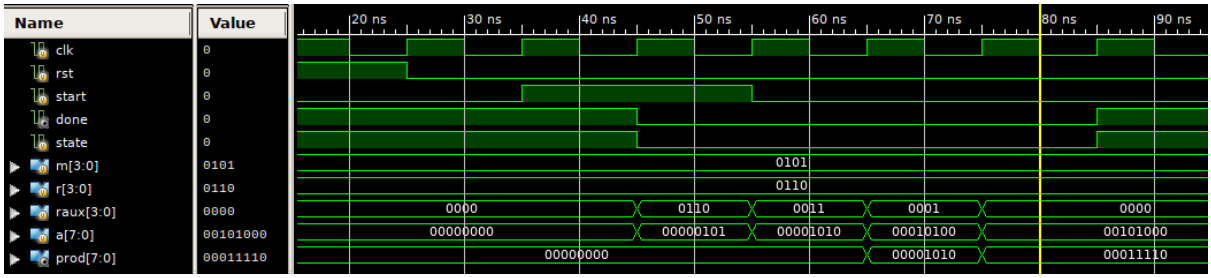


Fig. 5-8. Simulating the multiplication unit

5.3.2 VHDL implementation

Here we use a VHDL package to define our constants:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
package common_mul is
constant N_BITS: integer :=4;
constant PROD_BITS: integer :=(2*N_BITS);
constant N_BITS_ZERO: std_logic_vector((N_BITS-1) downto 0) := (others => '0');
end common_mul;
```

Listing 5-14. The VHDL package for multiplication unit

In the VHDL multiplication unit, include this package:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.common_mul.all;
entity mul is
port(    clk, rst, start:   in std_logic;
                    m, r:in std_logic_vector(N_BITS-1 downto 0);
                    prod:out std_logic_vector((2*N_BITS)-1 downto 0);
                    done: out std_logic
);
end mul;
```

Listing 5-15a. The VHDL multiplication unit

The VHDL architecture of multiplication starts with:

```
architecture mul_arch of mul is

        signal state, next_state: std_logic;
        signal a, next_prod, next_a, prod_signal: std_logic_vector((2*N_BITS-1) downto
0);
        signal raux, next_raux: std_logic_vector ((N_BITS-1) downto 0);

begin

        done <= '1' when (state = '1') else '0';
        prod <= prod_signal;
```

```
state_reg: process(clk, rst)
begin
        if (rst='1') then
                prod_signal <= (others => '0');
                state <= '1';
                a <= (others => '0');
                raux <= (others => '0');
        elsif (rising_edge(clk)) then
                state <= next_state;
                prod_signal <= next_prod;
                a <= next_a;
                raux <= next_raux;
        end if;
end process;
```

Listing 5-15b. The VHDL multiplication unit

Please note that the output variable prod is tied to the inner variable prod_signal, because it is not allowed to modify prod in a process.

The combinational logic for state and output computing is:

```
comb_logic: process(state, raux, m, r, start)
begin
        next_a <= a;
        next_prod <= prod_signal;
        next_state <= state;
        next_raux <= raux;
        case state is
        when '0' =>
                if(raux /= std_logic_vector(to_unsigned(0, N_BITS))) then
                        next_raux <= '0' & raux(N_BITS-1 downto 1);
                        if(raux(0) = '1') then
                                next_prod <=
std_logic_vector(unsigned(prod_signal) + unsigned(a));
                        end if;
                        next_a <= a((2*N_BITS)-2 downto 0) & '0';
                else
                        next_state <= '1';
                end if;
        when '1' =>
                if(start = '1') then
                        next_state <= '0';
                        next_a <= N_BITS_ZERO & m;
                        next_prod <= (others => '0');
                        next_raux <= r;
                end if;
        when others =>
```

```
                        -- this is forced by the vhdl compiler
                end case;
        end process;

end mul_arch;
```

Listing 5-15c. The VHDL multiplication unit

To verify that raux is not null, we convert to unsigned the value 0 on N_BITS and then convert it to std_logic_vector: “if(raux /= std_logic_vector(to_unsigned(0, N_BITS)))”. next_raux is set to raux shifted right one position, where “&” means concatenation: “next_raux <= '0' & raux(N_BITS-1 downto 1);”. Also, note that next_prod is computed as: “next_prod <= std_logic_vector(unsigned(prod_signal) + unsigned(a));”.

The VHDL test is:

```
LIBRARY ieee;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.common_mul.all;

ENTITY testmul IS
END testmul;

ARCHITECTURE behavior OF testmul IS

   COMPONENT mul
   PORT(
       clk, rst, start : IN  std_logic;
       m, r : IN  std_logic_vector((N_BITS-1) downto 0);
       prod : OUT  std_logic_vector(((2*N_BITS)-1) downto 0);
       done : OUT  std_logic
      );
   END COMPONENT;

  signal clk : std_logic := '0';
  signal rst : std_logic := '1';
  signal start : std_logic := '0';
  signal m : std_logic_vector((N_BITS-1) downto 0) := (others => '0');
  signal r : std_logic_vector((N_BITS-1) downto 0) := (others => '0');
  signal prod : std_logic_vector(((2*N_BITS)-1) downto 0);
  signal done : std_logic;

  constant clk_period : time := 10 ns;

BEGIN
  -- Instantiate the Unit Under Test (UUT)
  uut: mul PORT MAP (
```

```
        clk => clk, rst => rst, start => start,
        m => m, r => r, prod => prod,
        done => done);

   -- Clock process definitions
   clk_process :process
   begin
                clk <= '0';
                wait for clk_period/2;
                clk <= '1';
                wait for clk_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
                m <= std_logic_vector(to_unsigned(5, N_BITS));
                r <= std_logic_vector(to_unsigned(6, N_BITS));
                rst <= '1';
     wait for clk_period*2;

     -- insert stimulus here
                rst <= '0';
                wait for clk_period;
                start <= '1';
     wait for clk_period*2;
                start <= '0';
     wait;
   end process;
END;
```

Listing 5-16. The VHDL multiplication test unit

The VHDL simulation is identical to Verilog simulation.

Exercise 5-1. Implement in Verilog and VHDL the Booth multiply algorithm for signed numbers. One can find the algorithm description on Wikipedia website [WikiBooth].

Exercise 5-2. Implement in Verilog and VHDL the IEEE 754 floating point addition algorithm.

Exercise 5-3. Implement in Verilog and VHDL the IEEE 754 floating point multiplication algorithm. One can use the "*" operator for multiplying integer numbers.

Exercise 5-4. Implement in Verilog and VHDL the XTEA encryption algorithm [XTEA].

Exercise 5-5. Use the FloPoCo [FloPoCo] project to implement a unit which computes (a+b)*c where a, b, c are signals of IEEE 754 type.

6. UART

6.1 The UART protocol

Between the host (personal) computer and an FPGA board we can communicate via the UART protocol. Today, the PL2303 cable can be connected to the host USB port and in this way, the host computer will see it as an extra UART port. The PL2303 cable is shown in the figure below. The black pin represents GND, the white pin is RXD of the host, the green pin is TXD of the host and the red pin represents Vcc (usually 5V). In order to have communication between the host computer and FPGA board, host TXD must be connected to FPGA RXD, host RXD to FPGA TXD and host GND to FPGA board's GND; the red pin remains unconnected.

Fig. 6-1. The PL2303 USB to UART cable

The UART protocol is asynchronous. The transmitter and the receiver do not have a common clock signal; instead each have set an inner clock and both of them have the same frequency. The communication takes place between a transmitter and a receiver and is shown in the figure below in which the transmitter sends an octet to the receiver. When the line is idle, TXD value is 1. Before starting to send the data octet (D[7:0]), the start bit (a logic zero) is put on the line; then it follow the octet bits from D[0] to D[7] and, in the end, a stop bit (a logic one). After the stop bit, the line is put in idle state. In UART configuration is possible to add an extra bit to the line named parity bit whose value is a logical XOR of the data bits; this helps in detecting possible errors that may appear on the line. We will not use parity bit.

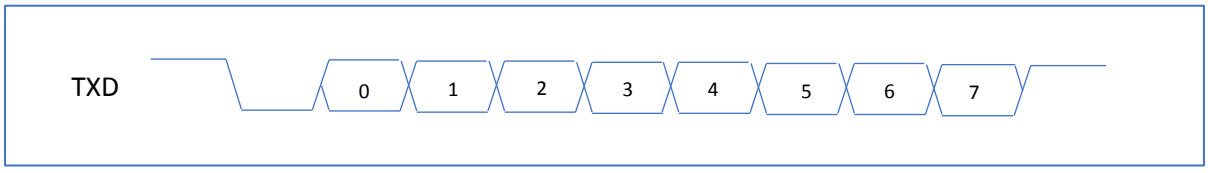


Fig. 6-2. The UART protocol

On the host computer, for the UART handling a free Java based library was used, named jssc (Java Simple Serial Connector) which is ported on Windows, MacOS and Linux. Jssc is available at [JSSClib]. If one has Java JRE installed (can run the command: "java –version")

it can run our application class already compiled. The commands to compile (needs Java Development Kit which is free) and run the Java UART application are:

Compile on Linux	Compile on Windows
export CLASSPATH="jssc.jar:." javac UARTSendReceive.java	set CLASSPATH=jssc.jar;. javac UARTSendReceive.java
Run on Linux	Run on Windows
export CLASSPATH="jssc.jar:."	set CLASSPATH=jssc.jar;.
java UARTSendReceive <port> <baudrate> <parity (0|1)> <char> Examples: java UARTSendReceive COM5 9600 0 a java UARTSendReceive /dev/ttyUSB0 115200 0 a	

Table 6-1. Compiling and running UARTSendReceive Java application

The Java application mainly consists in a few methods, here described. The first one is "attach()" which sets up the serial port with the user provided configuration.

```
public boolean attach(String portName, String strBaudRate, String parity) {
        serialPort = new SerialPort(portName);

        byte pb[]=parity.getBytes();
        int b=pb[0]=='1'?SerialPort.PARITY_ODD:SerialPort.PARITY_NONE;
        try {
                serialPort.openPort();//Open serial port
                int baudrate=Integer.parseInt(strBaudRate);
                serialPort.setParams(baudrate,
                  SerialPort.DATABITS_8,
                  SerialPort.STOPBITS_1,
                  b);
        } catch (SerialPortException ex) {
                ex.printStackTrace(System.out);
                return false;
        }

        return true;
}
```

Listing 6-1a. UARTSendReceive.java

The second method is "sendReceive()" which implements the transfer itself. Note that the host computer has a FIFO based UART where it saves the received octets.

```
public void sendReceive(String msg) throws IOException, SerialPortException {
        byte rawByte[]=new byte[1];
        rawByte = msg.getBytes();

        System.out.println("Sending...");
        serialPort.writeBytes(rawByte);
```

```
        // Reading from the serial port
        System.out.println("Reading");
        byte readByte[] = serialPort.readBytes(1);
        System.out.printf("Read: '%c'=0x%x\n", (char) readByte[0], (int) readByte[0]);
}
```

Listing 6-1b. UARTSendReceive.java

6.2 Verilog implementation

The current implementation is based on the free and open source ucore project available at [Zhangfeifi06]. The UART clock is set at 115200bps. Considering FPGA board clock at 50MHz, the UART clock frequency is 50MHz / (115200 * 16). We will see what for the additional division with 16 is used to.

6.2.1 The receiver

The receiver has 3 states: STA_IDLE, STA_CHECK_START_BIT, STA_RECEIVE. The first two states are implemented by the following sequence:

```
 always @(posedge clk_i or posedge rst_i)
 begin
  if(rst_i)
    /* init all to 0 */
  else begin
        if(baud_clk_posedge)
                case (reg_sta)
                STA_IDLE:
                begin
                 num_of_rec <= 4'd0;
                 count     <= 4'd0;
                 if(!rxd_i)
                        reg_sta  <= STA_CHECK_START_BIT;//recive a start bit
                 else
                        reg_sta  <= STA_IDLE;
                end
                STA_CHECK_START_BIT:
                begin
                 if(count >= 7)
                 begin
                        count   <= 0;
                        if(!rxd_i) begin
                        //has passed 8 clk and rxd_i is still zero,then start bit has been
confirmed
                                rdy_o   <= 1'b0;
                                reg_sta <= STA_RECEIVE;
```

```
                    end
                    else
                            reg_sta <= STA_IDLE;
                end
                else begin
                        reg_sta <= STA_CHECK_START_BIT;
                        count   <= count +1;
                end
               end
```

Listing 6-2a. UART Verilog receiver

The process is clear: in idle state, the receiver interrogates the RXD line (which is 1) and waits for the start bit (which is 0). If it founds a 0 then it jumps to the STA_CHECK_START_BIT. Here, it waits 8 periods of baud_clk and after that interrogates once more the line: if it is 0, then clearly we have a start bit; if it is 1 then we consider that it was a spike on the line, and go back to idle state – to wait for the start bit.

First time that we came in the STA_RECEIVE state we were in the middle of start bit (remember that in STA_CHECK_START_BIT we have waited 8 periods); so, wait 16 periods to arrive in the middle of first data bit. Then we sample the data bit line RXD (first data bit is D0). The attribution "rsr <= {rxd_i,rsr[7:1]};" shifts right the content of rsr. Considering that last data bit is D7 then last time, rxd_i=D7 and first time, rxd_i=D0; so, when we finish to receive all bits we have in rsr=D7…D0. After all bits are received signalize receiver is ready and go back to state idle.

```
STA_RECEIVE:
begin
        {count_c,count} <= count +1;
        //has passed 16 clk after the last bit has been checked,sampling a bit
        if(count_c)
        begin
                if(num_of_rec <=4'd7) begin //sampling the received bit
                        rsr       <= {rxd_i,rsr[7:1]};
                        num_of_rec <= num_of_rec +1;
                        reg_sta    <= STA_RECEIVE;
                end
                else begin//sampling the stop bit
                        data_o  <= rsr;
                        rdy_o   <= 1'b1;
                        reg_sta    <= STA_IDLE;
                end
        end
end
endcase
```

Listing 6-2b. UART Verilog receiver

6.2.2 The transmitter

The transmitter also uses three states: STA_IDLE = 0, STA_TRANS = 1, STA_FINISH = 2; we kept only the first two states.

The behavior in the idle state is shown below. In the idle state, the transmitter is waiting for the wen_i user send command while keeping the txd_o line to 1; when this arrives, the txd_o line is set to 0 (start bit) and it jumps to STA_TRANS state.

```
always @(posedge clk_i or posedge rst_i)
 begin
  if(rst_i) begin
   txd_o      <= 1'b1;
   tre_o      <= 1'b1;
   /* all other are 0 */
  end
  else begin
       if(baud_clk_posedge)
                case(reg_sta)
                STA_IDLE:
                begin
                 num_of_trans   <= 4'd0;
                 count         <= 4'd0;
                 count_c       <= 1'b0;
                 if(wen_i)
                 begin
                        tsr         <= data_i;
                        tre_o       <= 1'b0;
                        txd_o       <= 1'b0;// transmit the start bit
                        reg_sta     <= STA_TRANS;
                 end else
                        reg_sta     <= STA_IDLE;
                end
```

Listing 6-3a. UART Verilog transmitter

The STA_TRANS is shown below. First bit is start bit. It waits 16 periods of baud_clk, time when the current bit of data is kept on the line. Then, if the number of transmitted bits is less than or equal to 8, sets “tsr <= {1'b1,tsr[7:1]};” which shifts left tsr and adds a 1 in position tsr[7] which is a stop bit; the txd_o transmission line is set to tsr[0]. When num_of_trans becomes 9, the stop bit was sent too, and the transmitter jumps to idle state.

```
                STA_TRANS:
                begin
                 {count_c,count} <= count + 1;
                 if(count_c)
                 begin
```

```
                        if(num_of_trans <=4'd8)
                        begin
                                //note ,when num_of_trans==8 ,we transmit the stop bit
                                tsr        <= {1'b1,tsr[7:1]};
                                txd_o      <= tsr[0];
                                num_of_trans <= num_of_trans+1;
                                reg_sta    <= STA_TRANS;
                        end
                        else begin
                                txd_o      <= 1'b1;
                                tre_o      <= 1'b1;
                                reg_sta    <= STA_IDLE;
                        end
                 end
                end
                endcase
```

Listing 6-3b. UART Verilog transmitter

When the java application UARTSendReceive is launched on the host computer, it will first send the ASCII code of a character to the FPGA UART driver which will send back to the computer the ASCII code of the received character plus one. The sep_baud_uart module is used to establish this scenario and also show on the FPGA board leds the ASCII code.

```
module sep_baud_uart(sys_clk, reset, txd_o, rxd_i, char_leds);
…
uart_of_verifla iUART (sys_clk, sys_rst_l, baud_clk_posedge,
                                // Transmitter
                                txd_o,  wen_i, data_i, tre_o,
                                // Receiver
                                rxd_i, data_o, rdy_o);

// we transmit immediately after we receive.
single_pulse sp1 (sys_clk, sys_rst_l, baud_clk_posedge, rdy_o, wen_i);
assign char_leds = data_i;
always @(posedge sys_clk or negedge sys_rst_l)
begin
        if(~sys_rst_l)
                data_i = 8'h6E;
        else
                data_i = data_o+1;
end
endmodule
```

Listing 6-4. UART Verilog sep_baud_uart example top module

To verify that the communication between the host computer and the FPGA board works via the UART, the following test was done:

```
$ sudo ./run.sh UARTSendReceive /dev/ttyUSB0 115200 0 a
CLASSPATH=jssc.jar:.
port = /dev/ttyUSB0
baudrate = 115200
parity = 0
char = a
Sending…
Done sending.
Reading
Read: ‘b’=0x62
```

Fig. 6-3. Testing UART between host computer and FPGA board

The simulation in Xilinx ISE (which can easy be reproduced in Icarus) of the UART Verilog implementation is shown in Fig 6-4. clk_i, rst_i, data_i and wen_i are set in the test. Here is sent the ASCII character 0x61 which is ‘a’. In the simulation test, the signal rxd_i of the UART receiver was tied to the signal txd_o of the transmitter.

```
module test_txd;
…
u_xmit_of_verifla txd1(.clk_i(clk_i), .rst_i(rst_i), .baud_clk_posedge(baud_clk_posedge),
                       .data_i(data_i), .wen_i(wen_i), .txd_o(txd_o), .tre_o(tre_o));
u_rec_of_verifla rxd1(.clk_i(clk_i), .rst_i(rst_i), .baud_clk_posedge(baud_clk_posedge),
                       .rxd_i(txd_o), .rdy_o(rdy_o), .data_o(data_o));
baud_of_verifla
  baud1(.sys_clk(clk_i), .sys_rst_l(sys_rst_l), .baud_clk_posedge(baud_clk_posedge));

  always #5 clk_i = ~clk_i;

  initial begin
            clk_i = 0; rst_i = 1; data_i <= 0; wen_i = 0;
            #2000;
            rst_i = 0;
            #4000;
            data_i <= 8'h61;
            wen_i <= 1;
            #2000;
            wen_i <= 0;
            …
```

Listing 6-5. test_txd module for UART simulation in Verilog simulator

In the simulation Fig. 6-4, first reg_sta corresponds to the transmitter and the second to the receiver state.

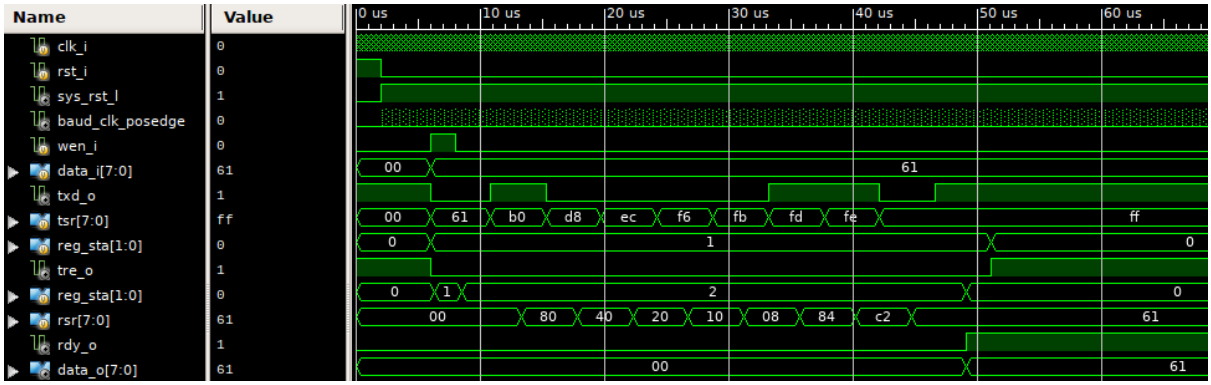


Fig. 6-4. The UART Verilog simulation

6.2 VHDL

The VHDL implementation of UART communication was made by converting the Verilog implementation to VHDL. All variables and signals are the same. Please consult our github repository for VHDL implementation of UART.

Exercise 6-1. Implement the UART unit with parity bit in communication for both Verilog and VHDL.

7. openVeriFLA logic analyzer

7.1 openVeriFLA architecture

openVeriFLA is an FPGA logic analyzer. This project helps in on-board testing and debugging of the FPGA projects. This is done by real-time capturing and then graphically displaying the signals transitions that happen inside the FPGA chip. Having a didactic scope, openVeriFLA is designed and tested on and for small projects.

openVeriFLA is distributed under the GNU GPL license (the UART sources have a more generous license – written in the source code). The host computer software is written in Java, so it is platform independent. The HDL code is written in Verilog and VHDL, in both languages being fully supported.

The main architecture of the openVeriFLA logic analyzer is shown in the figure below. The logic analyzer has two sides, the FPGA part and the host computer one. These communicates via the host computer USB-to-UART interface cable to the FPGA board.

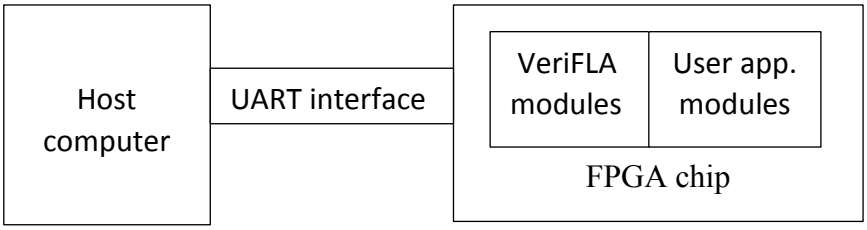


Fig. 7-1. Interfacing the logic analyzer

In order to use the logic analyzer, the openVeriFLA FPGA modules must be implemented in the FPGA chip along with the user application. The openVeriFLA modules capture the signal transitions of the monitored lines and send the data capture to the host computer for graphical visualization and future analyze.

The host computer part of the application is implemented in the Java language. The java application receives the captured data and saves it on the disk in a file named capture.v. This file is a behavioral (simulation) Verilog HDL file. A Verilog HDL simulator with a graphical viewer for the signals is necessary in order to simulate capture.v and view the captured data.

The interaction between FPGA and the host computer is illustrated in Fig. 7-2. For now, important is the fact that the host may send the run command to the monitor, in order to start a new capture and send it back.

As shown in this figure, the FPGA side of the logic analyzer is made by three components. These are:

- the monitor module which handles the data capturing process
- the computer-input and send-capture modules which handle the high level part of the communication between FPGA and the host computer

- the UART modules (not shown here) particular to the host computer-FPGA interface protocol.

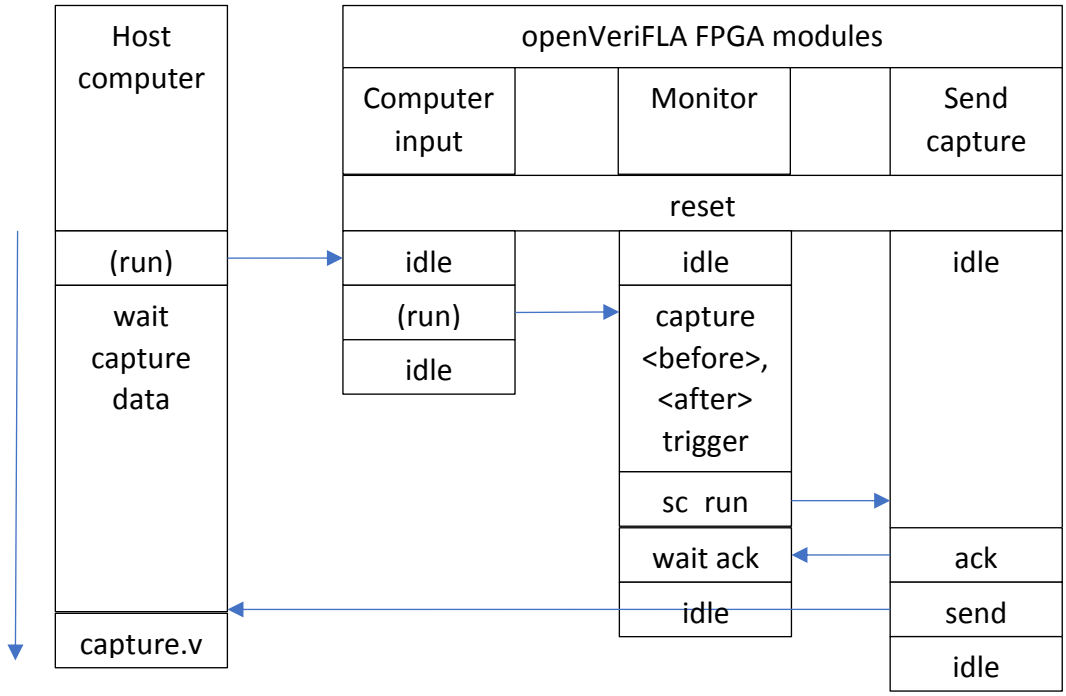


Fig. 7-2. openVeriFLA running flow

The data captured from the monitorized lines is kept in a memory buffer that must be available for the openVeriFLA modules. The memory buffer that comes with openVeriFLA by default is implemented in memory_of_verifla.v. The memory is kept in the mem array and addrb is used for reading and addra for writing.

```
module memory_of_verifla (clk, rst_l, addra, wea, dina, addrb, doutb);

`include "common_internal_verifla.v"

input rst_l, clk, wea;
input [LA_MEM_ADDRESS_BITS-1:0] addra, addrb;
output [LA_MEM_WORDLEN_BITS-1:0] doutb;
input [LA_MEM_WORDLEN_BITS-1:0] dina;

reg [LA_MEM_WORDLEN_BITS-1:0] mem[LA_MEM_LAST_ADDR:0];

//assign doutb = mem[addrb];
// This works too as a consequence of send_capture_of_verifla architecture.
reg [LA_MEM_WORDLEN_BITS-1:0] doutb;
always @(posedge clk or negedge rst_l)
if(~rst_l)
        doutb <= LA_MEM_EMPTY_SLOT;
else
```

```
        doutb <= mem[addrb];

always @(posedge clk)
begin
                if(wea) begin
                        mem[addra] <= dina;
                end
end
initial begin:INIT_SECTION
        integer i;
        for(i=0; i<=LA_MEM_LAST_ADDR; i=i+1)
                mem[i] <= LA_MEM_EMPTY_SLOT;
        //$readmemh("mem2018-2.mif", mem);
end
endmodule
```

Listing 7-1. memory_of_verifla.v

The memory buffer used for storing data is organized as in Fig. 7-3. A special moment in the process of data capturing is the trigger event. This is the moment when signals of the monitored lines match a user defined value. Before the trigger event, the data is stored in a circular FIFO queue named “before trigger queue”. At the end of the memory buffer it is stored the pointer to the tail of the circular queue. After the trigger event, the data is stored in a standard FIFO. When the “after trigger queue” is full, the data capture is sent to the host computer, where the user will analyze it.

A memory word contains a value of the captured data lines and the time that these data lines are constant. There are also reserved words which may specify:
- an empty and not used memory cell (LA_MEM_EMPTY_SLOT)
- the pointer to the tail of the before trigger queue which is stored in the last memory word.

The memory size and memory word length are parameterizable. The control-panel of the logic analyzer is the common_internal_verifla.v file. The other parameters of this file will be explained later.

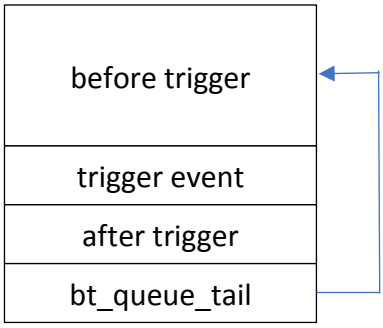


Fig. 7-3. Memory organization

7.2 Application - Simple counters capture

In order to test openVeriFLA, one will need as hardware an FPGA board and a PL2303TA USB to TTL serial converter as the one described in the UART chapter.

Instantiating the openVeriFLA top module in a HDL module is shown in the listing below. Please note that cntb and cnta can have arbitrary width.

```
module counters(cntb,
        clk, reset,
        //top_of_verifla transceiver
        uart_XMIT_dataH, uart_REC_dataH);

input clk, reset;
output [7:0] cntb;
//top_of_verifla transceiver
input uart_REC_dataH;
output uart_XMIT_dataH;

// Simple counters
reg [7:0] cntb, cnta;
always @(posedge clk or posedge reset)
begin
        if(reset) begin
                cntb = 0;
                cnta = 0;
        end else begin
                if((cnta & 1) && (cntb < 16'hf0))
                        cntb = cntb+1;
                cnta = cnta+1;
        end
end

// VeriFLA
top_of_verifla verifla (.clk(clk), .rst_l(!reset), .sys_run(1'b1),
                                .data_in({cntb, cnta}),
                                // Transceiver
                                .uart_XMIT_dataH(uart_XMIT_dataH),
                                .uart_REC_dataH(uart_REC_dataH));
Endmodule
```

Listing 7-2. Instantiating openVeriFLA in the counters module

One must instantiate top_of_verifla module and pass the following signals to openVeriFLA:

- clk, which is the board clock
- rst_l, which is the top_of_verifla reset signal and is active low
- sys_run, which instructs openVeriFLA whether to immediately start a data capture or wait for the user run command

- data_in which contains the signals from the counters module that will be captured
- uart_XMIT_dataH which is the openVeriFLA serial transmission line (similar to txd_o from the UART chapter)
- uart_REC_dataH which is the openVeriFLA serial reception line (similar to rxd_i from the UART chapter)

The signal transitions are captured on-the-fly by the openVeriFLA modules and then will be sent to the host computer, where will be prepared to be graphically displayed.

The FPGA board clock frequency (in Hz) must be written in the inc_of_verifla.v file before synthesis; this is required by the UART modules.

Note that openVeriFLA samples data @(posedge clk).

Part of the openVeriFLA synthesis report of the Xilinx ISE tools is shown in the table below (for the counters example):

Xilinx Spartan 3E 1600	
Minimum clock period:	9.089 ns
Number of Slices:	2% (394)
Number of Slice Flip Flops:	1% (242)
Number of 4 input LUTs:	2% (677)
Number of bonded IOBs:	4% (12)
Number of GCLKs:	4% (1 of 24)

Table 7-1. openVeriFLA FPGA required resources

The host computer should have at least Java Runtime Environment installed (JRE) for running already compiled code or recommended the Java Development Kit (JDK) for compiling. First, the Verifla.java source must be compiled by running compile.sh on Linux (with bash) or compile.bat on Windows; this will generate the VeriFLA.class. In order to receive the grabbed data from the FPGA chip, the VeriFLA.class is run on the host computer. The communication with the openVeriFLA modules is made via the usb-to-serial interface between the host computer and the FPGA development board; the VeriFLA class uses the jssc.jar UART library. This way, the signals capture will be sent to the host computer and saved in a form which can be displayed graphically.

On Linux, the VeriFLA.class is run with the following command (on Windows, one must replace sudo ./run.sh with run.bat):

```
$ sudo ./run.sh VeriFLA verifla_properties_counters.txt
or
$ sudo ./run.sh VeriFLA verifla_properties_counters.txt 1
```

Fig. 7-4. Running openVeriFLA on the host computer

These scripts include in CLASSPATH the path to jssc.jar.

In the first case, VeriFLA waits for data captured to arrive on the UART serial line, while in the second case, it first sends to the FPGA the command run which instructs it to start a new capture and send it on the UART serial line.

After the class is run as shown, the openVeriFLA modules are instructed to start a new capture and after the capture is finished, to send the capture to the host computer.

Now, these modules wait for signal events on the monitorized lines.

The java application gets the capture and builds the capture.v Verilog file. After this, the capture.v can be added and simulated in a Verilog simulator (e.g. Xilinx ISE or Icarus) project. The result is shown in the figures below.

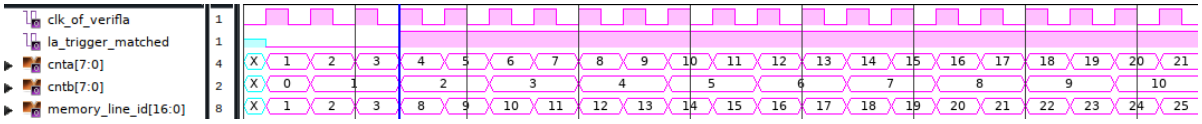


Fig. 7-5a. Simulating a capture.v file

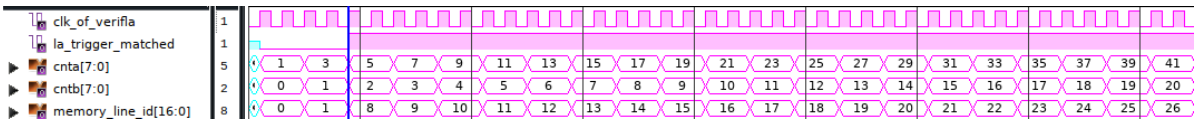


Fig. 7-5b. Simulating a capture.v file

The la_trigger_matched shows the moment when the trigger event appeared and memory_line_id is the index in the captured data memory (used as debug).

In the first figure, the monitor was configured to capture cntb and cnta whenever a bit of these signal changes. The trigger moment was set such as cntb=2 and cnta=4 (LA_TRIGGER_VALUE=16'h0204, LA_TRIGGER_MASK=16'hffff, LA_TRACE_MASK=16'hffff).

In the second figure, the monitor was configured (LA_TRIGGER_VALUE=16'h0200, LA_TRIGGER_MASK=16'hff00, LA_TRACE_MASK=16'hff00) to capture cntb and cnta only when cntb changes. So we can store in the same memory higher values of cntb and cnta (for example when memory_line_id is 20, in the first figure cntb is 8 and cnta is 16 and in the second figure cntb is 14 and cnta is 29). It must be mentioned the fact that in the second figure the trigger event appears when cntb=2 and, for this value, it corresponds two values of cnta (4 and 5) - the last value being kept by openverifla.

In the capture.v simulation, run command was necessary, to reach the $stop instruction of the capture.v.

7.3 Configuration parameters

7.3.1 Host computer parameters

The java application takes its parameters from a properties file. This file contains general parameters and application-specific parameters, like the names of the signals to be captured. An example is the verifla_properties_counters.txt file which is tuned for the counters example. Each parameter name starts with “LA.” (here this prefix is trimmed). The important parameters are:

- the UART serial portName and baudRate;
- memWords represents the size of the memory used to store the capture
- data input width and indentical samples bits (clones) must be multiples of 8 and are stored in dataWordLenBits and clonesWordLenBits;
- the index in memory where the trigger event appeared is stored in triggerMatchMemAddr; it also delimits the before and after trigger queues;
- the Verilog signals passed to top_of_verifla module are grouped. Each group of signals is defined by a number of group parameters. First is groupName which should be the same as the Verilog signal name. The groupSize represents the number of the signal lines in the group and is the same with the size of the Verilog signal. Sum of the groupSize parameter from all groups must be equal to totalSignals. The groupEndian specifies if the data represented by the group is in big-endian or low-endian format. Each group has an unique id specified at the end of the each parameter.
- timescaleUnit, timescalePrecision used for the Verilog timescale line in capture.v and clockPeriod is the period of the development board clock.

7.3.2 The FPGA parameters file

The clock frequency of openVeriFLA and the UART baudrate must be set in the inc_of_verifla.v file. This is used by the UART modules to compute the uart_clk. If the clock frequency of openVeriFLA is lower than 50 Mhz, then the baudrate must be lower than 115200 (for example 9600).

The control-panel of the logic analyzer is the common_internal_verifla.v file. It contains the configurable parameters of the logic analyzer.
- LA_MEM_WORDLEN_BITS represents the length in bits of a memory word; it is made of LA_DATA_INPUT_WORDLEN_BITS (the length in bits of data input) and LA_IDENTICAL_SAMPLES_BITS (the length in bits of the identical samples number) which means the number of clock periods that the data remains constant;
- LA_MEM_EMPTY_SLOT is the value that sets every memory line when cleaning the memory
- LA_TRIGGER_MASK specifies the bits to be considered when checking for the trigger value; it is used to mask the LA_TRIGGER_VALUE and the capture data when these two are compared.
- in LA_TRACE_MASK, the signals that are with 0 will be traced only when one or more signals that are with 1 change;
- LA_TRIGGER_MATCH_MEM_ADDR is the index in memory where the trigger event appeared;
- when the memory is full or it were captured LA_MAX_SAMPLES_AFTER_TRIGGER samples, the data capture is sent to the host computer.
- in order to represent an interval of time slots when the monitored lines are constant, the parameter LA_MAX_IDENTICAL_SAMPLES is the maximum identical samples number allowed to be stored in a memory word (it is built on LA_IDENTICAL_SAMPLES_BITS).

7.4 VHDL openVeriFLA

The Verilog sources were translated line by line in VHDL. Every .v file is .vhd in the VHDL sources. Everything specified in this manual for Verilog is valid in the VHDL implementation.

8. Buses

8.1 Buildroot and raspberry pi

In this chapter we will use the raspberry pi zero WH (and we will refer it as raspberry pi zero or more simple raspberry pi) computer to communicate with the FPGA board and this way we create a testbench to verify our implementations. The raspberry pi zero WH is a single board computer (shown in Fig. 8-1.) with one core CPU Broadcom 2835 (arm-v6), 512MB RAM, USB, UART, SPI, I^2C and other interfaces; it also has microSD slot where a microSD card with an operating system can be plugged in. On raspberry pi zero, Linux was ported and we will show how to install a Linux handy version in order to run our test programs on the raspberry pi zero computer.

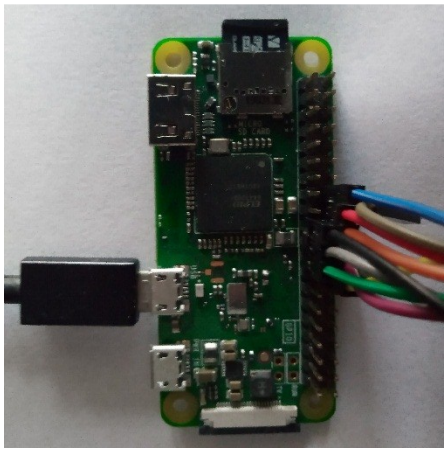


Fig. 8-1. The raspberry-pi zero WH

All new raspberry pi boards (including raspberry pi zero WH) have a GPIO header (in the right of the Fig. 8-1) which contains 40 pins (2 columns of 20 pins each). Some pins are dedicated for UART, some for SPI, I2C and so on; conforming to the reference [RaspberryPiPins] each pin's function is shown in table 8-1.

Function	Pin	Pin	Function
3V3 power	1	2	5V power
SDA	3	4	5V power
SCL	5	6	GND
GPCLK0	7	8	UART TXD
GND	9	10	UART RXD
GPIO 17	11	12	PCM_CLK
GPIO 27	13	14	GND
GPIO 22	15	16	GPIO 23
3V3 power	17	18	GPIO 24
MOSI	19	20	GND
MISO	21	22	GPIO 25
SCLK	23	24	CE0

GND	25	26	CE1
ID_SD	27	28	ID_SC
GPIO 5	29	30	GND
GPIO 6	31	32	PWM0
PWM1	33	34	GND
PCM_FS	35	36	GPIO 16
GPIO 26	37	38	PCM_DIN
GND	39	40	PCM_DOUT

Table 8-1. Raspberry Pi pin functions [RaspberryPiPins]

Buildroot is a Linux build system and it runs on Linux. If you have Windows installed you can download VirtualBox from [VirtualBoxSite] and install it, then create a virtual machine on which you install Ubuntu Linux. In Ubuntu Linux we can work with Buildroot.

Buildroot can be downloaded from [Buildroot] as an archive (buildroot-2019.02.tar.gz). To extract the archive just type in a terminal: “tar -xf buildroot-2019.02.tar.gz”. We can prepare Buildroot to build a Linux system for raspberry pi zero WH by running the following command in the Buildroot folder: “make raspberrypi0w_defconfig”. Then we configure our raspberry pi zero WH distribution with the command “make menuconfig” and selecting the following menu options (one can use the space key to select or deselect a menu entry, the ‘/’ key for search and ‘?’ for help):

```
Build options
   Enable Compiler Cache (for fast recompile)
Toolchain
   C library (glibc)
   Enable C++ support
   Build cross gdb for the host
System configuration
   Hostname: buildroot
   System banner: buildroot
   Root password: student
   Init system
      systemv
   /bin/sh bash
   /dev management (Dynamic using devtmpfs + eudev)
   Run a getty login prompt after boot
Kernel
   Defconfig name (bcmrpi_defconfig)
   In-tree Device Tree Source file names (bcm2708-rpi-zero-w)
Target Packages
   Show packages that are also provided by busybox
   Networking applications
     Dropbear
       Client programs
       Dropbear small
```

```
    Libraries
      Hardware handling
        bcm2835
        wiringpi (for GPIO access)
  Shell and utilities
    bash
    screen (UART access)
    sudo
    time
    which
  Text editors and viewers
    nano
    vim
  Debugging, profiling and benchmark
   spidev-test
   gdb -> gdb server
Filesystem images
  ext2/3/4
  exactsize
    200M
```

Listing 8-1. Buildroot raspberry pi zero WH configuration

After configuring Buildroot, we can now build our new Linux "distribution" with the command: "make". The build process takes more than one hour on an Intel i5 processor.

When the build finishes, using an USB to microSD reader insert the microSD card in the laptop. To see what drive is the microSD use the command "lsblk" and for example if the microSD has 8GB then look for a line like "sdb 8:16 1 7,5G 0 disk"; this means that the microSD card is seen as "/dev/sdb" (in the following we assume that the microSD card is seen like "/dev/sdb"). Then we must be sure that the microSD card is unmounted. To see the mounted drives run "mount" and look for "/dev/sdb1" and "/dev/sdb2"; if the microSD is mounted the output will contain something like in the figure below:

```
/dev/sdb2 on /media/laur/d0429413-ed47-4f39-aa63-a44928abb071 type ext4
(rw,nosuid,nodev,relatime,data=ordered,uhelper=udisks2)
/dev/sdb1 on /media/laur/81F9-08B3 type vfat
(rw,nosuid,nodev,relatime,uid=1000,gid=1000,fmask=0022,dmask=0022,codepage=437,i
ocharset=iso8859-1,shortname=mixed,showexec,utf8,flush,errors=remount-
ro,uhelper=udisks2)
```

Listing. 8-2. Partial list of mounted devices in Linux

To unmount the microSD card, assuming that is mounted in "/media/laur/" like in the above listing, use the following command: "umount /media/laur/*". However, it is a good practice to look in the Linux documentation for mounting and unmounting drives.

To transfer the new Linux built system on the microSD , from the Buildroot folder (note that here we assume that the microSD drive is seen as "/dev/sdb" and is not mounted):

```
sudo dd bs=4M if=./output/images/sdcard.img of=/dev/sdb conv=fsync status=progress
```

Listing 8-3. Copying the Linux built system to microSD.

Now we have a ready to run Linux system for raspberry pi zero WH on the microSD. The microSD now has two partitions:
- the boot partition which contains the Linux kernel and some other files from which the most important for now is the raspberry pi config.txt file;
- the root file system that contains the files to be ran on the raspberry pi.

Both partitions are mounted automatically in Ubuntu Linux when the microSD card is plugged into the laptop.

Note that Buildroot also builds a cross compiler on the host (laptop) for compiling C programs in order to be run on the raspberry pi zero WH computer. We will use it; its path relative to the buildroot folder is "output/host/bin".

In order to be able to connect to our raspberry pi zero board from a Linux laptop we must plug in a USB-to-UART cable in the laptop's USB port and the UART wires in the raspberry pi zero as follows: black wire to GND (pin6), white wire to TXD (pin8) and green wire to RXD (pin10). Also, on the boot partition of the microSD card, we have to edit config.txt as in the listing below. Then plug in the microSD card into raspberry pi, power on the raspberry pi and on laptop:
- on Ubuntu Linux enter in a terminal the commands: "sudo apt-get install minicom" (if minicom is not installed) and then "sudo minicom -D /dev/ttyUSB0";
- on Windows use Tera Term (you have to install it) to connect via USB to raspberry pi.

```
# See http://buildroot.org/manual.html#rootfs-custom
# and http://elinux.org/RPiconfig for a description of config.txt syntax

kernel=zImage

# To use an external initramfs file
#initramfs rootfs.cpio.gz

# Disable overscan assuming the display supports displaying the full resolution
# If the text shown on the screen disappears off the edge, comment this out
disable_overscan=1

# How much memory in MB to assign to the GPU on Pi models having
# 256, 512 or 1024 MB total memory
gpu_mem_256=100
gpu_mem_512=100
gpu_mem_1024=100

dtparam=spi=on
dtparam=i2c_arm=on
dtoverlay=dwc2
enable_uart=1

# fixes rpi3 ttyAMA0 serial console
```

```
dtoverlay=pi3-miniuart-bt

# get rid of bcm2835-aux-uart 20215040.serial: irq not found - -517
dtoverlay=pi3-disable-bt
```

Listing 8-4. Raspberry pi zero config.txt

8.2 The Wishbone system bus

Wishbone is a system bus and sits by default between CPU and memory or I/O devices. Wishbone is defined in [WishboneSpec] as being a logic bus and it does not deal with electrical information. It permits to attach to the same system bus many masters and slaves. Note that in a master-slave bus, any transfer is initiated by a master.

The main signals between a master and a slave on the Wishbone bus are shown in Fig. 8-2. The master first sets wb.addr, wb.we and wb.dato and then sets wb.stb which will tell the slave that the wb.addr is valid; if it wants to, the master uses wb.cyc to make more than one transfer with the slave. The slave first sets wb.dati and then wb.ack which tells the master that the current master request was successfully accomplished.

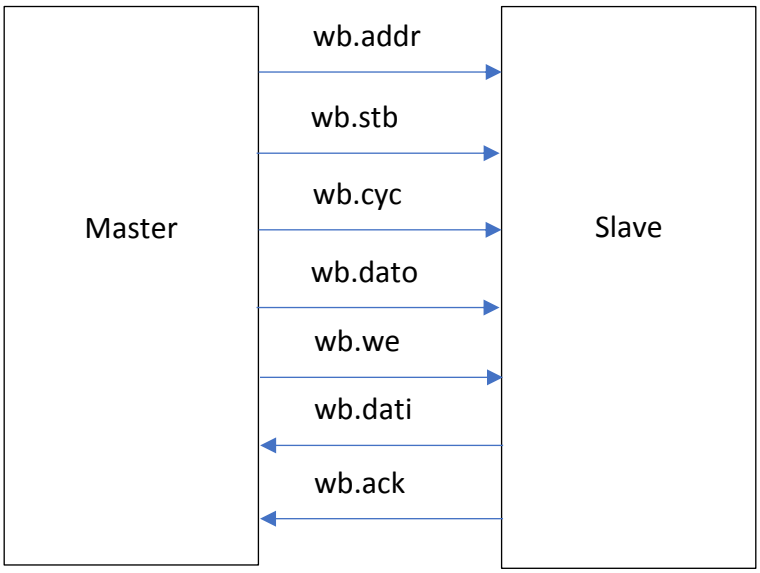


Fig. 8-2. Wishbone main signals

Because on the Wishbone bus can be present many masters and slaves, it is necessary an arbitration scheme, which is shown in Fig. 8-3.

One single master has access to the bus at a given moment. This master is selected by A1. The master that gains the access from the arbiter will have the signal wb.ack valid.
The other units of type master will have the signal wb.ack on 0. One single slave can be addressed at a given moment of time. Each slave has an allocated address. The slave that will be addressed will have the signals wb.stb and wb.cyc on 1. The other slaves have the wb.stb signal on 0.

In the following we will use the free and open source Daniel Wiklund Wishbone implementation. We trimmed it to two masters and two slaves to make the source more readable. wb_top.v is the Wishbone interface to masters and slaves. It uses wb_arb.v to select one master, if there are more requests. The wishbone arbiter code is shown in listing 8-5 and is self explanatory.

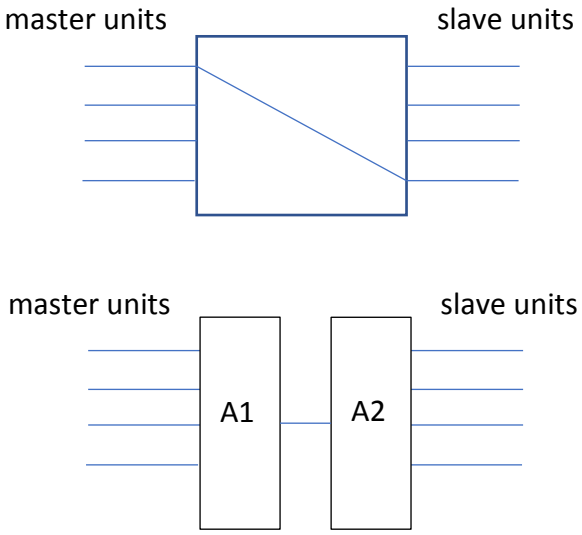


Fig. 8-3. Interfacing master and slave units on the Wishbone bus

```
module wb_arb(clk_i, rst_i, req_i, gnt_o);
…
always @ (posedge clk_i)
  if (rst_i) state <= #1 grant0;
  else       state <= #1 next_state;

assign gnt_o[0] = (state == 3'h0);
assign gnt_o[1] = (state == 3'h1);

always @ (state or req_i) begin
  case(state)
    grant0:
      if (req_i[0]) next_state = grant0;
                    else if (req_i[1]) next_state = grant1;
                    else next_state = grant0;
    grant1:
      if (req_i[1]) next_state = grant1;
                    else if (req_i[0]) next_state = grant0;
                    else next_state = grant1;
  endcase
end
endmodule
```

Listing 8-5. Wishbone arbiter

The wb_top module contains the addresses of the slaves.

```
module wb_top( ...  );

parameter s0_addr_width = 8;
parameter s0_addr = 8'h40;
parameter s1_addr_width = 8;
parameter s1_addr = 8'h90;
...

// Master 0
wire [`mbusw-1:0] m0_signal;
assign m0_signal = {m0_adr_i, m0_dat_i, m0_sel_i, m0_we_i, m0_cyc_i, m0_stb_i,
m0_cab_i, m0_cti_i, m0_bte_i};
…
assign m0_dat_o = s_signal[34:3];
assign m0_ack_o = s_signal[2] & gnt[0];

// Master 1
...
assign m1_ack_o = s_signal[2] & gnt[1];

// Slave 0
wire [`sbusw-1:0] s0_signal;
assign s0_signal = {s0_dat_i, s0_ack_i, s0_err_i, s0_rty_i};
...
assign s0_adr_o = m_signal[76:45];
assign s0_dat_o = m_signal[44:13];
assign s0_we_o  = m_signal[8];
assign s0_cyc_o = m_signal[7];
assign s0_stb_o = m_signal[6] & m_signal[7] & ssel[0];

// Slave 1
...
assign s1_cyc_o = m_signal[7];
assign s1_stb_o = m_signal[6] & m_signal[7] & ssel[1];
...

assign m_signal = gnt[0] ? m0_signal :
                           m1_signal;

assign s_signal = ssel[0] ? s0_signal :
                           ssel[1] ?        s1_signal : s0_signal;

assign ssel     = gnt[0] ? m0_ssel :
                           gnt[1] ? m1_ssel : 10'b0;
```

```
assign m0_ssel[0] = (m0_adr_i[s0_addr_width-1:0] == s0_addr);
assign m0_ssel[1] = (m0_adr_i[s1_addr_width-1:0] == s1_addr);
assign m1_ssel[0] = (m1_adr_i[s0_addr_width-1:0] == s0_addr);
assign m1_ssel[1] = (m1_adr_i[s1_addr_width-1:0] == s1_addr);

wb_arb arb(.clk_i(clk_i), .rst_i(rst_i), .req_i({m1_cyc_i, m0_cyc_i}), .gnt_o(gnt));

assign grant_o = gnt;

endmodule
```

Listing 8-6. Wishbone wb_top.v module

In wb_top.v we have:
- m0_ssel[0] is 1 if master 0 address corresponds to slave 0 otherwise is 0;
- the ssel variable will be m0_ssel if the arbiter selects master 0 or m1_ssel otherwise;
- m_signal is master 0 signal if the arbiter selects it or else master 1 signal;
- s_signal is slave 0 signal if slave 0 is selected, otherwise is slave 1 signal if slave 1 is selected;
- s0_stb_o is a function of ssel[0] and is 0 when the slave 0 is not selected;
- s0_stb_o when is valid is m_signal[6] & m_signal[7] which are the master stb and cyc signals;
- s1_stb_o is similar to s0_stb_o;
- m0_ack_o is s_signal[2] & gnt[0], so is 1 only when the arbiter grants the bus to master 0 and the selected slave sends ack to master 0;
- m1_ack_o is similar to m0_ack_o.

A simplified job for a master module was implemented in wb_master.v. It selects the slave with slave_address and it requires a write with master_param as data.

```
module wb_master (…);

parameter master_param = `dw'd0;
parameter slave_address = `aw'd0;
reg transfer_done;

always @(posedge clk_i or posedge rst_i)
begin
        if(rst_i) begin
                transfer_done = 0;
                m_cyc_o = 0; m_stb_o = 0; m_we_o = 0;
                …
        end else if(!transfer_done) begin
                if(m_ack_i) begin
                        transfer_done = 1;
                        m_cyc_o = 0; m_stb_o = 0; m_we_o = 0;
                end else begin
                        m_cyc_o = 1; m_stb_o = 1; m_we_o = 1;
                        // select all bytes
```

```
                        m_sel_o = {`selw{1'b1}};
                        m_data_o = master_param;
                        m_addr_o = slave_address;
                end
        end
  end

endmodule
```

Listing 8-7. A simplified wishbone master

A simplified wishbone slave is shown in wb_slave.v. It has no internal memory. If the stb, cyc and we signals are active, will not save the master data (as it should, because we is 1) because the slave has no internal memory but will return to master the slave data and ack.

```
module wb_slave      (…);

  parameter slave_param = `dw'd0;

  always @(posedge clk_i or posedge rst_i)
        begin
                if(rst_i) begin
                        slave_ack_o = 0;
                        slave_dat_o = 0;
                end else begin
                        slave_ack_o = slave_stb_i && slave_cyc_i;
                        if (slave_cyc_i && slave_stb_i && slave_we_i)
                        begin
                                slave_dat_o = slave_param;
                        end
                end
  end
endmodule
```

Listing 8-8. A simplified wishbone slave

In order to simulate, we create top.v which instantiates the masters, slaves and wishbone modules and in the instantiations sets their parameters. Master 0 parameters are address 40h and data 20h, master 1 parameters are address 90h and data 30h. In the wishbone bus instantiation is set address width to 8bits, slave 0 address 40h and slave 1 address 90h. Slave 0 parameter is data 25h and slave 1 parameter is data 35h (h from hexadecimal base).

```
module top;
/* set clock oscillator and reset sequence */
…
// Instantiate masters
wb_master #(`dw'h20, `aw'h40) master0 (
    .clk_i(clk_i), .rst_i(rst_i), .m_cyc_o(m0_cyc), .m_stb_o(m0_stb), …);
```

```
wb_master #(`dw'h30, `aw'h90) master1 (
    .clk_i(clk_i), .rst_i(rst_i), .m_cyc_o(m0_cyc), .m_stb_o(m0_stb), …);
// Instantiate slaves
wb_slave #(`dw'h25) slave0 (
    .clk_i(clk_i), .rst_i(rst_i), .slave_we_i(s0_we), .slave_stb_i(s0_stb), …);
wb_slave #(`dw'h35) slave1 (
    .clk_i(clk_i), .rst_i(rst_i), .slave_we_i(s0_we), .slave_stb_i(s0_stb), …);

// Instantiate wishbone bus
wb_top #(8, 8'h40, 8, 8'h90) wb_bus (
   .clk_i(clk_i), .rst_i(rst_i), …);
endmodule
```

Listing 8-9. Instantiating masters, slaves and wishbone modules

Simulation of the top module is shown in the figure below. After the reset sequence, both masters request the bus (req_i=11). Master 0 is the first which receives bus grant; it sets stb, cyc, we, data out (20h) and slave 0 address (40h) on the bus and receives from the slave data 25h and ack. Then, the bus arbiter grants the bus to master 1 which was waiting for slave 1 ack. The slave 1 receives sees its address (90h) on the wishbone bus, because the master 1 accesses slave 1 (sending to it data 30h) and returns with data (35h) and ack to master 1.

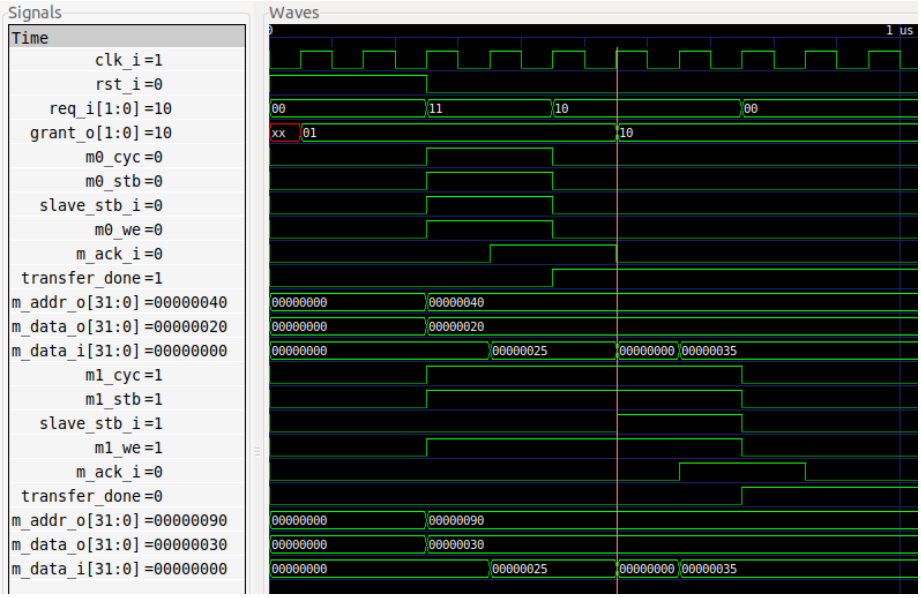


Fig. 8-4. Simulating the wishbone project

8.3 The SPI bus

SPI is a 4-wire master-slave bus and its protocol depicted in Fig. 8-5 (CLK is clock, SS is slave select, MOSI is master out slave in, MISO is master in slave out) may vary depending on configuration parameters (clock polarity CPOL and phase CPHA). In our implementation was used SPI MODE 0 (CPOL=0, CPHA=0). The signals MOSI and MISO may change on SCK falling edge and are constant on the SCK rising edge. The data bits can be sent LSb (least significant bit) first or MSb (most significant bit) first, this is configurable. In our implementation we have chosen MSb. The slave is selected when SS is 0.

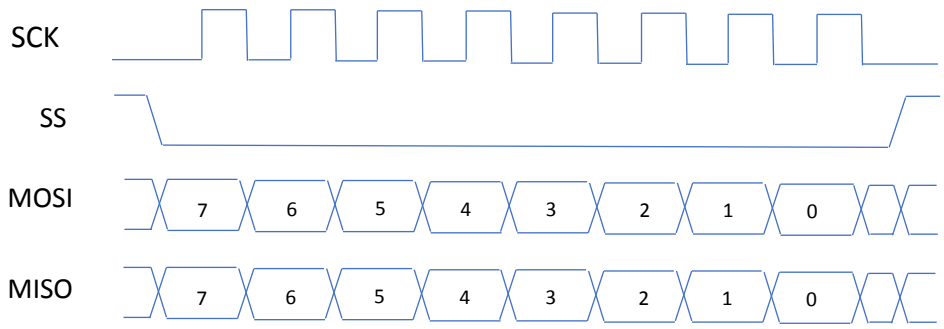


Fig. 8-5. SPI bus transfer protocol

The SPI bus can have only one master and several slaves. A scheme with one master and two slaves is shown in the figure below. The slaves share SCK, MOSI and MISO and have dedicated SS: SS0 to slave 0 and SS1 to slave 1.

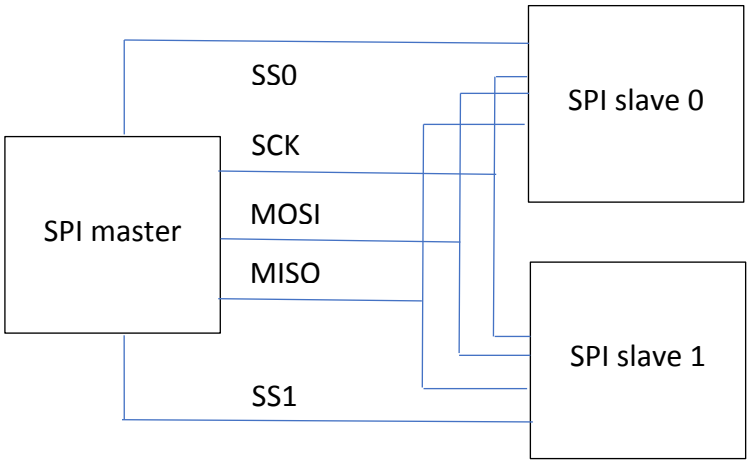


Fig 8-6. SPI master with two slaves

We present our SPI Verilog implementation. The SPI implementation is done in spi_slave_lcd.v which is shown in the listing below. SCK_rise will be 1 when SCK changes from 0 to 1. If SSEL is 1 then we announce that output is not valid and set i to 7.

```
module spi_slave_lcd(clk, SCK, SSEL, MOSI, MISO, output_valid, byte_data_received, byte_data_send);
…
reg [2:0] SCK_vec=3'b000;
always @(posedge clk)
	SCK_vec <= {SCK_vec[1:0], SCK};
wire SCK_rise=(SCK_vec[2:1]==2'b01);

// SPI mode 0
always @(posedge clk) begin
	if (~SSEL) begin
		if(SCK_rise) begin
			byte_data_received[i] <= MOSI;
			if(i == 0) begin
				i <= 7;
				output_valid <= 1;
			end
			else begin
				output_valid <= 0;
				i <= i-1;
			end
		end
		if (SCK_vec[1] == 0) begin
			MISO <= byte_data_send[i];
		end
	end else begin
		i <= 7;
		output_valid <= 0;
	end
end
```

Listing 8-10. spi_slave_lcd.v

When SSEL is set to 0, the slave is selected. If SCK_rise is 1 it means MOSI is valid and we save it in byte_data_received[i] (which now is 7 which means we are receiving MSb first); output_valid is set to 0 and i is decremented because now we are waiting for data bit 6. When i reaches 0, it means we received all the bits and set output_valid to 1 (the octet is in byte_data_received) and reset i to 7.

If SCK_vec[1] is 0 it means we can set MISO (because we modify it when SPI clock is 0) and we set it to byte_data_send[i].

For testing our spi_slave_lcd module, we made the following scenario (see the figure below): the raspberry pi zero WH is connected to the FPGA board through SPI pins; the laptop is connected to the raspberry pi zero using USB-to-UART cable, so we can have a tty console on the raspberry pi zero. On raspberry pi zero, CE0 and CE1 are SS0 and SS1. We will use the first SPI which have the following pins: MOSI=pin19, MISO=pin21, SCK=pin23 and SS=CE0=pin24.

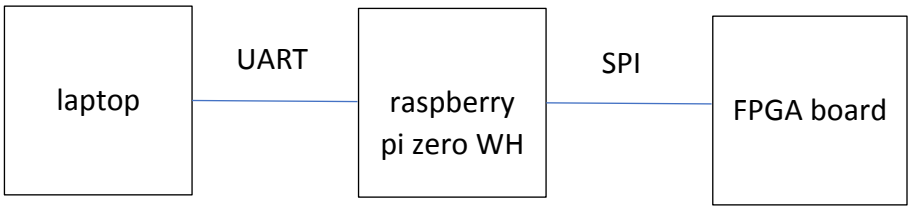


Fig. 8-7. SPI testing scheme

In order to use the SPI in the FPGA, we must instantiate spi_slave_lcd. The module that makes the work is spi_checker.v and a simple version of it is shown in the listing below.

```
module spi_checker(clk, rst, SCK, SSEL, MOSI, MISO, leds, btn);
…
// Debouncer
`define DEB_LEN 10
// SCK
reg sck_deb=1'b0;
reg [`DEB_LEN-1:0] sck_pipe={`DEB_LEN{1'b0}};
always @(posedge clk) begin
   sck_pipe <= {sck_pipe[`DEB_LEN-2:0], SCK};
   if (&sck_pipe[`DEB_LEN-1:1] == 1'b1)
     sck_deb <= 1'b1;
   else if (|sck_pipe[`DEB_LEN-1:1] == 1'b0)
     sck_deb <= 1'b0;
end
// SSEL
/* debouncer similar to SCK */

spi_slave_lcd si (clk, sck_deb, ssel_deb, MOSI, MISO, spi_output_valid,
byte_data_received, byte_data_send);

assign byte_data_send = 8'h61; // ASCII code for ‘a’.
assign leds = byte_data_received;

endmodule
```

Listing 8-11. spi_checker.v

The only thing to note here is the SCK debouncing variable which considers that when SCK and SS change on the cable from raspberry pi zero to the FPGA board, these signals may have spikes. We consider that when changing from 0 to 1 the signal must have 10 consecutive system clock periods the value 1; and when it changes from 1 to 0, the signal must have 10 consecutive system clock periods the value 0. Note that the system clock (signal clk) has a much shorter period than the SPI clock.

We have to write a C program for raspberry pi to communicate with the FPGA board via SPI. The program is shown in the listing below in a simplified form; we call it spi-user.c. The code is self explanatory.

```
#include …
#define TRANSFER_SIZE 1

int spi_config(int fd)
{
   int spi_mode, freq, bits_per_word, ret, lsb_first;

   spi_mode = SPI_MODE_0;
   if((ret = ioctl(fd, SPI_IOC_WR_MODE, &spi_mode)) < 0) {
      perror("SPI_IOC_WR_MODE");
      return -1;
   }
   bits_per_word = 8;
   if((ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits_per_word)) < 0) {
      perror("SPI_IOC_WR_BITS_PER_WORD");
      return -1;
   }
   /* use MSB first instead of LSB first */
   lsb_first = 0;
   if((ret = ioctl(fd, SPI_IOC_WR_LSB_FIRST, &lsb_first)) < 0) {
      perror("SPI_IOC_WR_LSB_FIRST");
      return -1;
   }
   freq = 50000;
   if((ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &freq)) < 0 ) {
      perror("SPI_IOC_WR_MAX_SPEED_HZ");
      return -1;
   }

   return 0;
}

void spi_run(int fd)
{
   int i, ret;
   char buf_send[TRANSFER_SIZE] = {0, };
   char buf_recv[TRANSFER_SIZE] = {0, };

   struct spi_ioc_transfer msg[1] = {
      [0] = {
         .tx_buf = (unsigned long)buf_send,
         .rx_buf = (unsigned long)buf_recv,
         .len = TRANSFER_SIZE,
```

```
        .cs_change = 1, /* change CS between transfers */
        .delay_usecs = 0, /* delay after each transfer */
        .bits_per_word = 8,
    },
  };
  for(i = 0; i < TRANSFER_SIZE; i++) {
          buf_recv[i] = 0;
          buf_send[i] = i+1;
  }
  // SPI_IOC_MESSAGE(1) => one SPI transfer
  ret = ioctl(fd, SPI_IOC_MESSAGE(1), &msg);
  …
}
```

Listing 8-12. spi-user.c

Because the raspberry pi runs Linux, we will use the Buildroot cross compiler generated in chapter 8.1. In Ubuntu Linux go to the spi-user.c folder and run the commands (replace the buildroot cross compiler path with the one that you have generated):

```
export PATH=/home/laur/lucru/raspberry-pi/buildroot/buildroot-2019.02.rpi/output/host/bin:$PATH
arm-linux-gcc -c -o spi-user.o spi-user.c
arm-linux-gcc -o spi-user.out spi-user.o
```

Figure 8-8. Compiling spi-user.c

After we have spi-user.out we must transfer it to the microSD card that has the Linux system built with Buildroot in chapter 8.1. Insert the microSD card into the laptop via the USB-to-microSD device and on Ubuntu Linux the microSD is automatically mounted. Use the command: "cp spi-user.out /path/to/microSD/rootfs/partition/root/". Unmount the card and plug it into raspberry pi. Plug in the raspberry pi and, on the laptop UART console enter the command "/root/spi-user.out". Now the transfer will be made between raspberry pi SPI and FPGA board. You should see on the raspberry pi terminal:

```
# modprobe spi-bcm2835.ko && modprobe spidev.ko
# ./spi-user.out
device=/dev/spidev0.0
ioctl returns 1
i=0 rx=61
```

Listing 8-13. Running the SPI test

8.4 The I2C bus

I2C is a two wire bus invented by Philips Semiconductors (now NXP). Its specification is available at [I2Cspec]. The I2C uses only two wires, namely SDA (serial data) and SCL (serial clock). The masters and the slaves are connected to these two wires. The speed of the I2C bus is slow (standard 100 Kbps, full 400Kbps, fast 1Mbps, high 3.2Mbps) compared to SPI bus which can support clock frequencies up to tens of MHz.

When SCL is high, SDA is valid; when SCL is low, SDA may change. A transfer initiated by the master has a start sequence (SDA transition from high to low when SCL is high) and a stop sequence (SDA transition from low to high when SCL is high).

First 7 bits after the start sequence of a I2C transfer represent the slave address; so, I2C can theoretically have a maximum of 128 slaves. Data on the I2C line is always 8 bits wide and is transmitted as MSb first. On an I2C transfer the master can send data to slave (write) or receive data from slave (read). The read/write selector is a bit that follows the slave address and is 1 for reading and 0 for writing. After the read/write bit is the slave ACK bit which means that the slave is ready to proceed with the transfer.

In order to make an I2C transfer the following scheme is used.

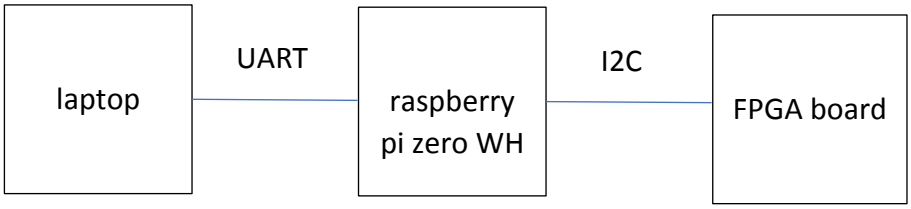


Fig. 8-8. I2C testing scheme

The program that was compiled for raspberry pi zero is shown in the following listing.

```
#include …

#define I2C_ADDR 0x72

int main (void) {
   char buffer[3];
   int fd;
   int i;

   fd = open("/dev/i2c-1", O_RDWR);
   if (fd < 0) {
      printf("Error opening file: %s\n", strerror(errno));
      return 1;
   }
```

```
    if (ioctl(fd, I2C_SLAVE, I2C_ADDR) < 0) {
        printf("ioctl error: %s\n", strerror(errno));
        return 1;
    }

    buffer[0]=0x25;
    write(fd, buffer, 1);

    buffer[0]=buffer[1]=buffer[2]=0;
    read(fd, buffer, 1);
    for(i=0; i<1; i++)
        printf("buffer[%d]=0x%02X\n", i, buffer[i]);
    return 0;
}
```

Listing 8-14. i2c_master.c

It can be seen that the slave address is 72h and data to be sent is 25h. The master first makes a write transfer and then a read transfer with the FPGA board. The i2c_master.c can be compiled the same way we compiled spi-user.c from the SPI chapter, then must be copied to the microSD that contains Linux and finally be plugged into raspberry pi zero. In order to run the test, on raspberry pi zero we have to enter the following commands:

```
# modprobe i2c-dev
# /root/i2c_master.out
```

Listing 8-15. Running i2c_master.out

The write transfer was caught with openVeriFLA logic analyzer and the waves of the two I2C wires are shown in the figure below. It can be seen that:
- transfer begins with the start condition (between memory_line_id 8 and 9);
- the address is 72h (between bitcount 0 and 6);
- the transfer type is write (we have SDA 0 for bitcount 7);
- after the transfer type is present slave ack (bitcount 8);
- the sent data is 25h (between bitcount 9 and 16);
- then follows slave-ack (bitcount 17);
- then follows stop condition (between memory_line_id 59 and 60).

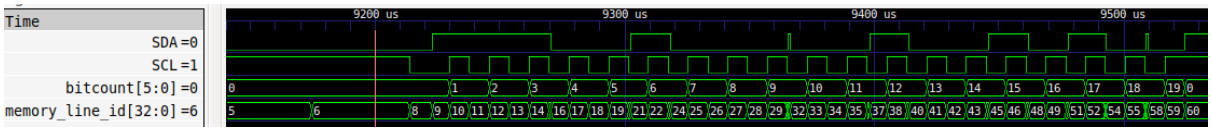


Fig. 8-9. The I2C write transfer

The read transfer was caught with openVeriFLA logic analyzer and the waves of the two I2C wires are shown in the figure below. It can be seen that:
- transfer begins with the start condition (between memory_line_id 61 and 62);
- the address is 72h (between bitcount 0 and 6);
- the transfer type is read (we have SDA 1 for bitcount 7);

- after the transfer type is present slave ack (bitcount 8);
- the sent data is 25h (between bitcount 9 and 16);
- then follows master-ack (bitcount 17);
- then follows stop condition (between memory_line_id 111 and 112);

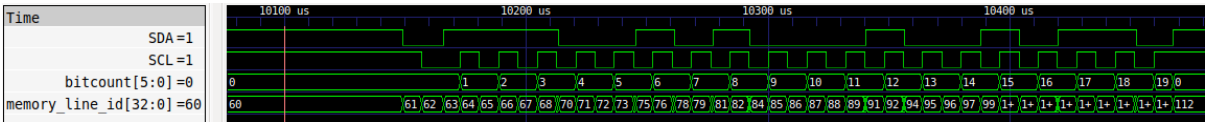


Fig. 8-10. The I2C read transfer

The I2C Verilog unit was adapted from the Alex Forencich I2C slave implementation (which has the free MIT license).

The start and stop condition in a simplified form is shown below. The important thing is that the signal incycle tells us if we are inside the transfer.

```
//Detect start and stop
reg start = 1'b0;
always @(posedge clk)
	if(SDA_negedge)
		start = sclDeb;
reg stop = 1'b0;
always @(posedge clk)
	if(SDA_posedge)
		stop = sclDeb;

//Set cycle state
reg incycle = 1'b0;
always @(posedge clk)
	if (start)
	begin
		if (incycle == 1'b0)
			incycle = 1'b1;
	end
	else if (stop)
	begin
		if (incycle == 1'b1)
			incycle = 1'b0;
	end
```

Listing 8-16a. i2c_slave.v

Address and incoming data handling is made in the sequence below. All the data is strobed only when SCL_posedge is 1 (which means that SDA is stable). The master provided address is saved in the address variable. After the address is read from the SDA_vec line, in the rw variable is stored the type of the operation that the master wants. If the address matches, the variable addressmatch is set to 1. If the master wants to write (rw is 0), then the received data is saved in the variable datain bit by bit.

```
always @(posedge clk)
	if (~incycle)
	begin
		//Reset the bit counter at the end of a sequence
		bitcount <= 0;
	end else if(SCL_posedge) begin
		//Get the address
		if (bitcount < 7)
			address[6 - bitcount] <= SDA_vec[1];
		else if (bitcount == 7)
		begin
			rw <= SDA_vec[1];
			addressmatch <= (slaveaddress == address) ? 1'b1 : 1'b0;
		end else if ((bitcount >= 9) && (bitcount <= 16) && (~rw))
			//Receive data (currently only one byte)
			datain[16 - bitcount] <= SDA_vec[1];
		bitcount <= bitcount + 1;
	end
```

Listing 8-16b. i2c_slave.v

The ACK and outgoing data logic is shown in the following listing, where:

- all the data is set when the SCL is zero;
- the variable SDA is assigned to sdadata which is high impedance (1'bz) when the master drives it and binary logic when the slave drives it; this is the way to deal with inout signals;
- ACKs are sent if the address matches and the bitcount is 17 and rw is 0 or when bitcount is 8;
- data is sent only if bitcount is greater or equal to 9 and currvalue is 0 (because valuecnt is 1).

```
parameter valuecnt = 3'b001;
reg sdadata = 1'bz;
reg [2:0] currvalue = 0;
always @(posedge clk)
	if(SCL_negedge) begin
		//ACK's
		if (((bitcount == 8) | ((bitcount == 17) & ~rw)) & (addressmatch)) begin
			sdadata <= 1'b0;
			currvalue <= 0;
		end
		//Data
		else if ((bitcount >= 9) & (rw) & (addressmatch) & (currvalue < valuecnt))
begin
			//Send Data
			if (((bitcount - 9) - (currvalue * 9)) == 8)
			begin
				//Release SDA so master can ACK/NAK
```

```
                                sdadata <= 1'bz;
                                currvalue <= currvalue + 1;
                        end else
                                //Modify this to send actual data, currently echoing
incomming data valuecnt times.
                                sdadata <= datain[7 - ((bitcount - 9) - (currvalue * 9))];
                end else
                        //Nothing (cause nothing tastes like fresca)
                        sdadata <= 1'bz;
        end
assign SDA = sdadata;
```

Listing 8-16c. i2c_slave.v

Exercise 8-1. Use the openverifla analyzer to catch the SPI unit signals.

Exercise 8-2. Convert the SPI driver to VHDL.

Exercise 8-3. Convert the I2C driver to VHDL.

9. The central processing unit

9.1 A simple pipeline

When talking about processors we are interested what is the total time that a program takes to be run. The total program time has the following formula:

program_time = number_of_instructions * clock_cycles_per_instruction * clock_cycle_time

We now introduce the pipeline concept by using an example. Let's consider the following register.v which defines a register unit.

```
module register(clk, rst, next, out);
input clk, rst;
input [15:0] next;
output [15:0] out;
reg [15:0] out;

always @(posedge clk or posedge rst)
begin
	if(rst)
		out = 0;
	else
		out = next;
end
endmodule
```

Listing 9-1. register.v

Now we define pipeline_biss.v which has three similar modules pipeline_biss_et1, pipeline_biss_et2 and pipeline_biss_et3.

```
module pipeline_biss_et1 (clk, rst, a, b);
input clk, rst;
input [15:0] a;
output [15:0] b;
wire  [15:0] b;
reg [15:0] next_b;

register r1(.clk(clk), .rst(rst), .next(next_b), .out(b));

always @(*)
	if(a != 0)
		next_b = a + 10;  // numbers are by default in base 10.
	else
		next_b = 0;
endmodule
```

```
module pipeline_biss_et2 (clk, rst, b, c);
… next_c = b + 100;
module pipeline_biss_et3 (clk, rst, c, d);
… next_d = c + 1000;  // numbers are by default in base 10.
```

Listing 9-2. pipeline_biss.v

The test that instantiates these modules is test_pipeline_biss.v and is shown in the following listing. We have:

- the input of ps_et1 is a and output is b;
- the input of ps_et2 is b and output is c;
- the input of ps_et3 is c and output is d.

```
module test_pipeline_biss;
…
   pipeline_biss_et1 ps_et1 (.clk(clk), .rst(rst), .a(a), .b(b));
   pipeline_biss_et2 ps_et2 (.clk(clk), .rst(rst), .b(b), .c(c));
   pipeline_biss_et3 ps_et3 (.clk(clk), .rst(rst), .c(c), .d(d));

   initial begin
      clk = 1'b0;
      a = 0;
      rst = 1'b1;      #(2*PERIOD);
      rst = 1'b0;      #PERIOD;

      a = 1;      #PERIOD;
      a = 2;      #PERIOD;
      a = 3;      #PERIOD;
      a = 4;      #PERIOD;
      a = 5;      #PERIOD;
      a = 6;      #PERIOD;
      a = 0;      #PERIOD;
   end

endmodule
```

Listing 9-3. test_pipeline_biss.v

The nice thing about these three modules is that they “run” in parallel. It means that in the same time when ps_et1 computes b, ps_et2 computes c and ps_et3 computes d. The simulation is shown in the figure below.

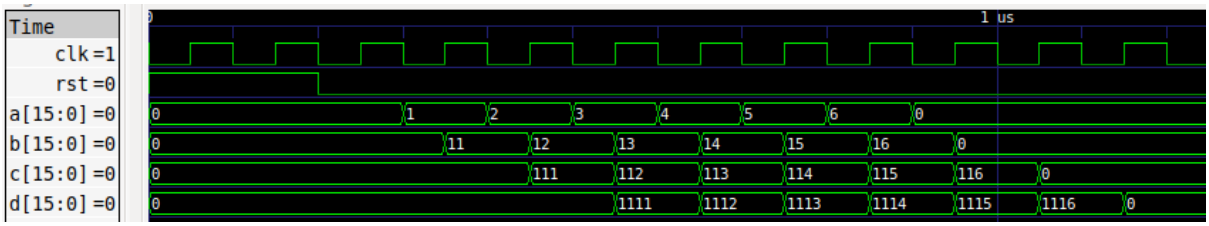


Fig. 9-1. Simple pipeline simulation

Let's consider the clock cycles corresponding to the signal a from 1 to 6. Each module does 6 addition operations; a total of 18 addition operations. If the modules would "run" serially then the necessary time to execute the 18 operations would take 18 clock cycles. But, because the three modules run in parallel the total number of clock cycles is 8, beginning with the cycle when b becomes 11 and ending with the cycle when d becomes 1116.

That's what pipeline does: define units that execute in parallel, so, in the end we have fewer clock cycles per program instruction.

Let's look now at what a processor does for executing an instruction:
- it loads the instruction for memory (this stage is known as "fetch");
- it "decodes" the instruction (see what type of instruction is and what are the parameters);
- it executes the instruction (the "execute" stage);
- it writes data to memory (the "memory" stage) or loads data from memory, if the instruction requires this
- it writes the result to register set (the "write back" stage) if the instruction requires this.

If we use pipeline then instead of 5 cycles per instruction (the width of our pipeline) we will have 1 cycle per instruction. But, we must consider that in reality, the pipeline stages are not totally independent (and they need to synchronize at some points) so there is something more than 1 cycle per instruction (usually 1.x or 2.x CPI).

9.2 Cache, MMU and interrupts

The "link" between the processor core and main memory is made via the MMU (memory management unit) and cache in a scheme similar to Fig. 9-2.

The main memory (RAM) is smaller than the permanent memory but is much faster; the same applies to cache memory: is smaller but much faster than the main memory. The cache is much bigger than the processor register set, but much slower.

The **cache** memory stores instructions and data words that are most often accessed by the processor core. Because cache is smaller than main memory, it can not contain all main memory; we call cache miss the situation when the processor wants to access the contents of a memory location whose contents are not in cache; we call cache hit the situation when the processor wants to access the contents of a memory location whose contents are present in cache. Usually cache hit ratio can vary between 60% and 95%, depending on the cache architecture.

The most probable memory words that are to be accessed again are the memory words placed in main memory at addresses:
- near the address currently requested by the processor;
- that were already accessed by the processor.

That is why we normally do not replace a single memory location in case of a miss, but a whole block of memory locations.

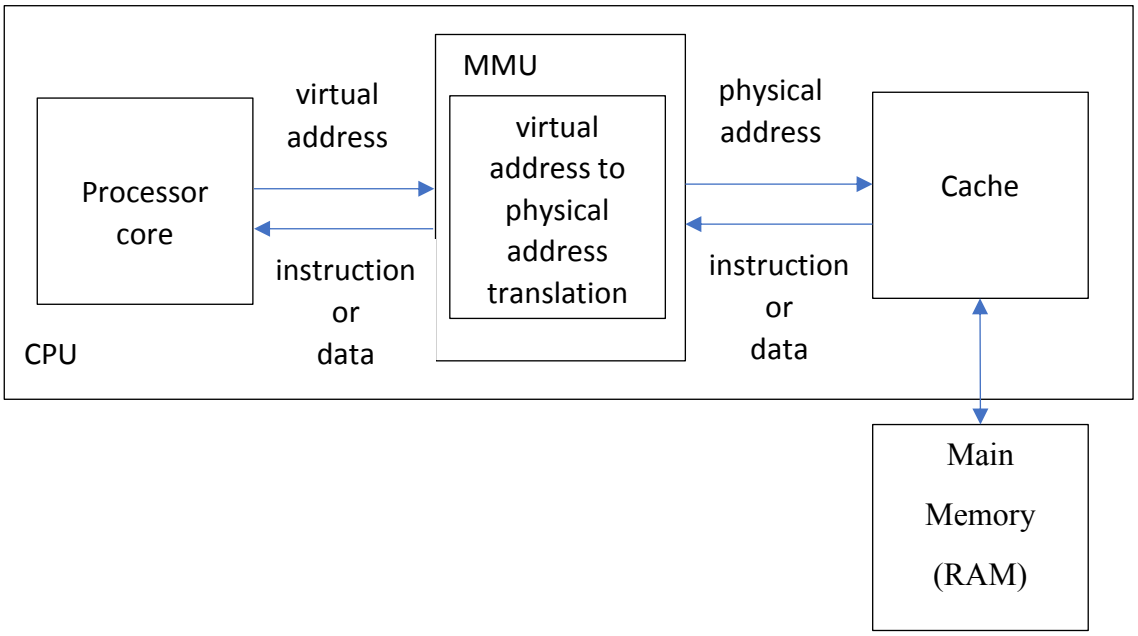


Fig. 9-2. Processor, MMU, cache and main RAM memory

Now comes the question: what to replace from memory cache in case of a miss? There are more alternatives available and the most frequent are to replace the LRU (least recently used) cache entry or use the FIFO (first in first out) replacement scheme.

Mapping is the process that transfers the main memory locations to cache memory. There are three generally used mapping methods: direct mapping, full associative mapping and set associative mapping.

When using full associative mapping, the main memory can be mapped anywhere in the cache memory. A cache line contains the pair address and data of the mapped memory location. Its disadvantage is the fact that it is slow, because in order to verify that a memory location is mapped in the cache memory we must check all the cache memory locations to find out if it is there.

When using direct mapping, multiple blocks of memory can be mapped in the same cache memory block. For example, when using a 2048 blocks main memory and a 64 blocks cache memory, we can map 2048/64 = 32 main memory blocks in the same cache memory block. In this case, the processor address is split into an index and a tag. The cache is indexed by the index part of the address and contains the tag and the value of a memory location. For example:

- memory addresses 01000 and 02000 have the tags 01 and 02 and the index 000, so it can be stored only at cache index 000;
- let's suppose that the word of value 1234 at address 02000 is stored in cache; then the cache entry 000 contains tag 02 and value 1234.

The advantage of using direct mapping is that searching whether a memory location is present in cache is very fast. The drawback is when the processor accesses memory locations which have the same index (and obviously different tags): the cache miss rate increases very much.

Set associative mapping is a trade-off between direct mapping and associative mapping. We “break” the cache memory in separate sets. Inside the set, the mapping is associative. A memory location is first mapped onto a set and then placed into any cache entry of the set. If the set contains only one memory location, then set associative mapping is identical to direct mapping. There are implementations with 2-way or 4-way set associative mapping.

When writing a word of data to memory, the cache can use the policies write-through (when the data is written in cache and also in memory) or write-back (when data is written only in cache and will be written in memory only when the data will be taken out of the cache as a follow up to a cache miss).

Virtual memory creates the illusion of a big memory. The main memory acts as a cache for the permanent memory. CPU programs use virtual addresses. Each program has its own virtual to physical mapping. Two programs can use the same virtual addresses to access completely different data. A program can not vitiate the memory used by another program.

Cache uses blocks and main memory uses pages; for example, the size of one page can be 4096 bytes. The equivalent of the cache miss is page fault. The operating system (for example, Linux) maintains the page table for each process in main (physical) memory. A page table is a vector and contains an entry for each page of the process. The page table also contains a validity bit which tells if the page is present in main memory or is only in the permanent (non-volatile) memory.

The virtual address is made by the virtual page number (VPN) and the page offset. The physical address is made by the physical page number (PPN) and the page offset. VPN is the index in the page table. The translation between a virtual address to a physical address is made by the **MMU**. The translation process is illustrated in the figure below.

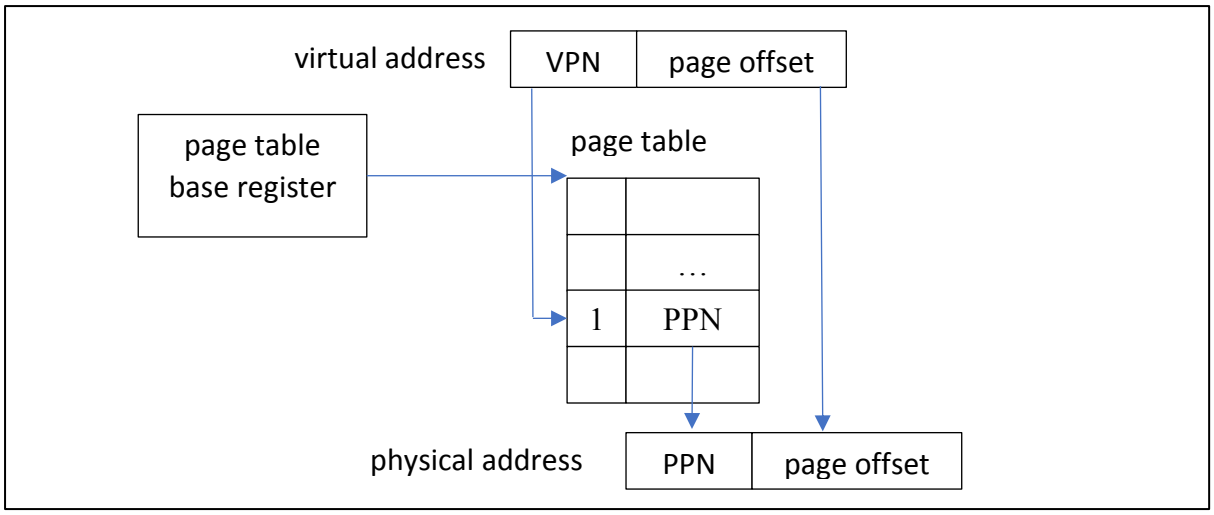


Fig. 9-3 Virtual to physical page translation

The MMU uses the page table base register which points to the beginning of the page table. The VPN is an index in the page table and the PPN at this index will be chosen. If the valid bit is one, it means that the page is in main memory. The physical address is computed by concatenating PPN and page offset. Modern operating systems use several level page tables (for example Linux which uses 3, 4 or 5 level page tables depending on the system architecture).

The MMU has its own cache of physical addresses that were the result of virtual addresses translation. This cache is named translation lookaside buffer TLB and has over 90% hit rate.

If an I/O device wants to signal an event to the CPU it can do it by using the **interrupts** system. A simple interrupt-based system is shown in the figure below. When the CPU is running a program and an I/O device raises an interrupt, the CPU disable interrupts, saves the current program state, acknowledges the interrupt and runs the interrupt handlers registered with the corresponding interrupt. After the interrupt handlers finish, the CPU restores the state of the interrupted program, enables interrupts and continues with the interrupted program. In this way asynchronous requests from I/O devices are served by the CPU. The interrupt controller may be connected to the processor data bus.

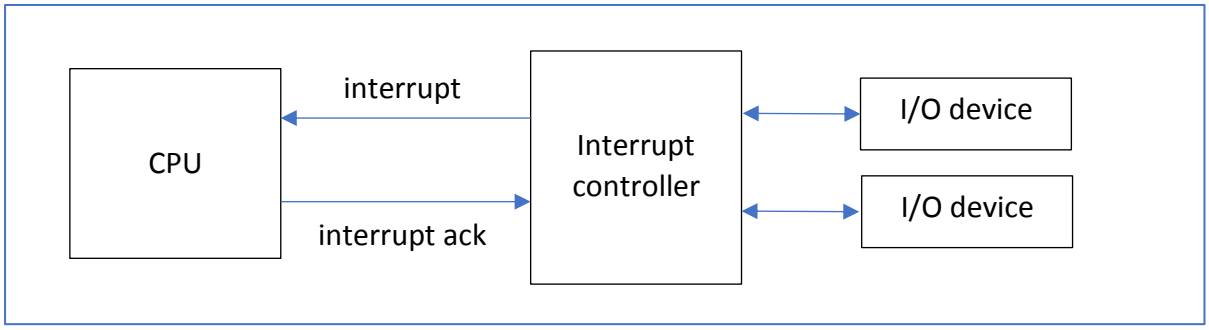


Fig. 9-4. Interrupt based system

9.3 RISCV instruction set architecture

RISC-V (pronounced “risk-five”) is an instruction-set architecture (ISA) meant to be the standard free and open architecture for industry implementations as its authors declared in the RISCV instruction set manual.

The RISCV privileged ISA covers the aspects and functionality of RISCV systems required for running operating systems and attaching external devices; it is not described in this book.

The RISCV instruction set manual, volume I covers the unprivileged ISA and is available in [RV]. In the following we will use this manual to describe the RISCV instruction set. The manual defines instruction set for CPU architectures RV32i or RV64i plus standard extensions (a)tomics, (m)ultiplication and division, (f)loat, (d)ouble.

The minimum, mandatory set of RISC-V instructions is the integer instruction set. (Indicated with 'i' or capital 'I'.) This set by itself can implement a simplified general-purpose computer, with full software support, including a general-purpose compiler. A computer design may add further subsets: Integer multiplication and division (set 'M'), atomic instructions for handling real-time concurrency ('A'), IEEE Floating point ('F') with Double-precision ('D') and Quad-precision ('Q') options. A computer with all of these instruction sets, an 'RVIAFDP' is said to be 'general-purpose' summarized as 'G'.

RV32I is a 32bit CPU architecture and contains 40 unique instructions and 32 general purpose registers. RV32E reduces the integer register count to 16 general-purpose registers, (x0–x15), where x0 is a dedicated zero register. RV32E is only used with a soft-float calling convention.

The Application Binary Interface (ABI) defines the mechanisms that describe how the functions are invoked, how the parameters are passed between functions, how the returned value is provided to the caller function, etc. Supported ABIs are ilp32 (32-bit soft-float), ilp32f (32-bit with single-precision in registers and double in memory), ilp32d (32-bit hard-float), ilp32e (embedded), ilp64, ilp64f and ilp64d (the same but with 64-bit long and pointers).

A word of memory is defined as 32 bits (4 bytes), a half word is 16 bits (2 bytes), a double word is 64 bits (8 bytes) and a quad word is 128 bits (16 bytes).

The following table describes the registers and their ABI names for RV32I.

Register	ABI name	Description	Saver
x0	zero	Hard-wired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5	t0	Temporary / alternate link register	Caller

x6-x7	t1-2	Temporaries	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-1	Function arguments / return values	Caller
x12-x17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller

Table 9-1. RV32I registers [RV]

"The standard software calling convention uses register x1 to hold the return address for a call, with register x5 available as an alternate link register. The standard calling convention uses register x2 as the stack pointer" [RV].

The following table lists the RV32I instructions type.

Instruction bits							Instruction Type
31…25	24…20	19…15	14…12	11…7	6…0		
func7	rs2	rs1	func3	rd	opcode		R-type
imm[11:0]		rs1	func3	rd	opcode		I-type
imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode		S-type
imm[12,10:5]	rs2	rs1	func3	imm[4:1,11]	opcode		B-type
imm[31:12]				rd	opcode		U-type
imm[20,10:1,11,19:12]				rd	opcode		J-type

Table 9-2. RV32I instructions type [RV]

The RISC-V ISA keeps the source (rs1 and rs2) and destination (rd) registers at the same position in all formats to simplify decoding. The imm field represents an integer value written in base 2. The sign bit for all immediate values is always in bit 31 of the instruction to speed sign-extension circuitry.

Instruction bits							Instruction
31…25	24…20	19…15	14…12	11…7	6…0		
0000000	rs2	rs1	110	Rd	0110011		OR
imm[11:0]		rs1	000	rd	0010011		ADDI
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011		SW
imm[12,10:5]	rs2	rs1	100	imm[4:1,11]	1100011		BLT
imm[20,10:1,11,19:12]				rd	1101111		JAL

Table 9-3. Examples of RV32I instructions [RV]

"RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers. Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective address is obtained by adding register rs1 to the sign-extended 12-bit offset. Loads copy a value from memory to register rd" [RV]. Stores copy the value in register rs2 to memory (variants: store word SW, store half SH, store byte SB). See the example at the end of the Darkriscv CPU dedicated chapter.

“The LW instruction loads a 32-bit value from memory into rd. LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in rd. LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in rd. LB and LBU are defined analogously for 8-bit values” [RV].

“In a system for which endianness is byte-address invariant, the following property holds: if a byte is stored to memory at some address in some endianness, then a byte-sized load from that address in any endianness returns the stored value.

In a little-endian configuration, multibyte stores write the least-significant register byte at the lowest memory byte address, followed by the other register bytes in ascending order of their significance. Loads similarly transfer the contents of the lesser memory byte addresses to the less-significant register bytes.

In a big-endian configuration, multibyte stores write the most-significant register byte at the lowest memory byte address, followed by the other register bytes in descending order of their significance. Loads similarly transfer the contents of the greater memory byte addresses to the less-significant register bytes” [RV].

“ADDI adds the sign-extended 12-bit immediate to register rs1; arithmetic overflow is ignored and the result is placed in register rd. The NOP instruction does not change any architecturally visible state, except for advancing the program counter (PC – address of the instruction currently executed) and incrementing any applicable performance counters. NOP is encoded as ADDI x0, x0, 0” [RV]. Similar instructions to ADDI are ANDI, ORI, XORI.

OR performs bitwise OR logical operation between register rs1 and rs2 and places the result in register rd. Similar are AND, XOR; ADD, SUB performs adding and subtracting between rs1 and rs2.

“The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. JAL stores the address of the instruction following the jump (pc+4) into register rd” [RV].

Branch instructions compare two registers rs1 and rs2; for example, BLT takes the branch if the register rs1 is less than register rs2. “The 12-bit B-immediate encodes signed offsets in multiples of 2 bytes. The offset is sign-extended and added to the address of the branch instruction to give the target address” [RV].

A list with all instructions is shown in [RV] in the chapter “RV32/64G Instruction Set Listings” and instructions are explained in detail in chapter “RV32I Base Integer Instruction Set”.

9.4 darkriscv implementation of the RISCV

9.4.1 Introduction to darkriscv

Many RISCV open source and commercial implementations exists on the market: RV32EC_P2, RV32IC_P5, RV32EC_FMP5, rocket, freedom, Berkeley Out-of-Order Machine (BOOM), ORCA, RI5CY, Ibex (formerly Zero-riscy), Ariane, Riscy Processors, RiscyOO, Lizard, Minerva, OPenV/mriscv, VexRiscv, Roa Logic RV12, SCR1, SCR3, SCR4, SCR5, SCR7, Hummingbird E200, Shakti, ReonV, PicoRV32, MR1, SERV, SweRV EH1, Reve-R, Bk3, Bk5, Bk7, DarkRISCV, RPU, RV01, N22, N25F, D25F, A25, A25MP, NX25F, AX25, AX25MP, Instant SoC, Taiga, Maestro, XuanTie C910, XuanTie E902, BM-310, BI-350, BI-651, BI-671, SSRV, RSD, Pluto [RVCoreList].

In this chapter, we present a modified (simplified) version of the darkriscv processor implementation whose original code is available at [DarkRV].

Darkriscv is a “bare metal” hardware platform where instructions have full access to the physical address space. Moreover, the simplified darkriscv processor supports a single thread. It has not MMU, cache and interrupt controller; the processor directly use physical memory addresses. Is defined as little-endian and it implements RV32I or RV32E, depending of its configuration. Also, we use the HARVARD memory architecture instead of von Neumann so the memory is split in code (ROM) and data (RAM).

The GNU C compiler (gcc) that was ported to RISCV can be compiled for Darkriscv. “Although the fence*, e* and crg* instructions are not implemented, the gcc appears to not use of that instructions and they are not available in the core” [DarkRV].

For RV32E, building the compilation toolchain is shown in the listing below. When compiling gcc the following error may occur: “Error: non-constant .uleb128 is not supported”. The darkriscv does not currently support large memory configurations and the current firmware does not use libgcc. So, ignore the error.

```
git clone –depth=1 git://gcc.gnu.org/git/gcc.git gcc
git clone –depth=1 git://sourceware.org/git/binutils-gdb.git
git clone –depth=1 git://sourceware.org/git/newlib-cygwin.git
mkdir combined
cd combined
ln -s ../newlib-cygwin/* .
ln -sf ../binutils-gdb/* .
ln -sf ../gcc/* .
mkdir build
cd build
../configure –target=riscv32-embedded-elf –enable-languages=c –disable-shared –disable-threads –disable-multilib –disable-gdb –disable-libssp –with-newlib  --with-arch-rv32e –with-abi=ilp32e –prefix=/usr/local/share/gcc-riscv32-embedded-elf
make -j4
make
sudo make install
```

Listing 9-4. Compiling gcc for darkriscv RV32E [DarkRV]

The following listing shows how to verify that the GNU toolchain was successfully installed.

```
export PATH=$PATH:/usr/local/share/gcc-riscv32-embedded-elf/bin
riscv32-embedded-elf-gcc -v
```

Listing 9-5. Verifying gcc installation for darkriscv RV32E

Compiling and verifying gcc for darkriscv RV32I is similar to RV32E, with the following specific steps.

```
../configure --target=riscv32-unknown-elf --enable-languages=c --disable-shared --disable-threads --disable-multilib --disable-gdb --disable-libssp --with-newlib --with-arch=rv32ima --with-abi=ilp32 --prefix=/usr/local/share/gcc-riscv32-unknown-elf
```

Listing 9-6. Compiling gcc for darkriscv RV32I [DarkRV]

```
export PATH=$PATH:/usr/local/share/gcc-riscv32-unknown-elf/bin
riscv32-unknown-elf-gcc -v
```

Listing 9-7. Verifying gcc installation for darkriscv RV32I

The darkriscv sources are the following.

Folder	Sources	Details
./	Makefile	System makefile
sim	Makefile, darksimv.v	Simulation module
rtl	darkriscv.v, darksocv.v, darkuart.v, config.vh	CPU, SOC, UART, config
src	Makefile banner.c, stdio.c, io.c, boot.c, main.c io.h, stdio.h	Firmware C files
src	darksocv.ld.src,	Memory sections linker script

Table 9-4. darkriscv sources

The program executed by the processor sits in ROM. The current memory map in the linker script is the follow:
0x00000000: 4KB ROM
0x00001000: 4KB RAM

Also, the linker maps the I/O memory space starting from the address 0x80000000.

The RAM memory contains the .data area, the .bss area (after the .data and initialized with zero), the .rodada and the stack area at the end of RAM. The stack area is at the end of RAM and the stack pointer register SP decreases from end of RAM towards the start of RAM. The ROM memory contains the .text section which is the firmware program code.

We now present how to simulate darkriscv RV32E in Icarus. First change directory to darkriscv-e folder and then enter the following commands.

```
export PATH=$PATH:/usr/local/share/gcc-riscv32-embedded-elf/bin
make clean
make
```

Listing 9-8. Simulating darkriscv-e

The result should be similar to the one in the figure below.

```
reset (startup)
LEDFF=0000
LEDFF=0005
          vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
            vvvvvvvvvvvvvvvvvvvvvvvvvvvv
rrrrrrrrrrrrr      vvvvvvvvvvvvvvvvvvvvvvvvvv
rrrrrrrrrrrrrrr     vvvvvvvvvvvvvvvvvvvvvvvv
rrrrrrrrrrrrrrrrr   vvvvvvvvvvvvvvvvvvvvvvvvv
rrrrrrrrrrrrrrrrr   vvvvvvvvvvvvvvvvvvvvvvvvv
rrrrrrrrrrrrrrrrr   vvvvvvvvvvvvvvvvvvvvvvvvv
rrrrrrrrrrrrrrr     vvvvvvvvvvvvvvvvvvvvvv
rrrrrrrrrrrrr      vvvvvvvvvvvvvvvvvvvvv
rr            vvvvvvvvvvvvvvvvvvvv
rr          vvvvvvvvvvvvvvvvvvvvvv      rr
rrrr     vvvvvvvvvvvvvvvvvvvvvvvv      rrrr
rrrrrr     vvvvvvvvvvvvvvvvvvvvv      rrrrrr
rrrrrrrr     vvvvvvvvvvvvvvvvv      rrrrrrrr
rrrrrrrrr     vvvvvvvvvvvvv      rrrrrrrrr
rrrrrrrrrrr     vvvvvvvvv      rrrrrrrrrrr
rrrrrrrrrrrrr     vvvvvv      rrrrrrrrrrrrr
rrrrrrrrrrrrrrr     vv      rrrrrrrrrrrrrrr
rrrrrrrrrrrrrrrrr         rrrrrrrrrrrrrrrrr
rrrrrrrrrrrrrrrrrrr      rrrrrrrrrrrrrrrrrrr
rrrrrrrrrrrrrrrrrrrrr  rrrrrrrrrrrrrrrrrrrrr

     INSTRUCTION SETS WANT TO BE FREE

boot0: text@0 data@4096 stack@8192
board: simulation only (id=0)
build: darkriscv fw build Wed, 01 Apr 2020 13:08:56 +0300
core0: darkriscv@50.0MHz with rv32e+MAC
uart0: 115200 bps (div=434)
Welcome to DarkRISCV!
end simulation
echo simulation ok.
simulation ok.
```

Fig. 9-5. Simulating darkriscv in Icarus

9.4.2 darkriscv Verilog sources

Now we explain the Verilog sources. The simplified version of Darkriscv is mainly contained in two files: darksoc.v, darkrisc.v; keep this files open while reading our explanations. There is a third file, named darkuart.v – which implements UART and the config.vh file which contains parameter values.

In darksoc.v, the memory is implemented similar to how we implemented memory in openVeriFLA. Most of the synthesis tools (for example Xilinx synthesis tool XST) recognize the memory format and will generate block ram memory (BRAM); other tools may generate distributed ram from FPGA. The memory mapping described above is implemented in darksoc.v; look for DATA_ADDR[31] signal which is 1 when CPU accesses I/O devices.

The darkriscv.v contains three stage pipeline working with a single clock phase: in this case the instruction is fetch from a block ram in the first clock, decoded in the second clock and executed in the third clock. The code is self explanatory. We consider __3STAGE__ defined in config.vh and cache is disabled.

The register set is stored in the RG matrix variable. During reset, RESMODE is used to initialize RG[2] which is SP to the __RESETSP__ value (8192 in config.vh). PC=0 takes two periods after reset, but flush_instr_pipe on reset is initialized to 2, so the pipeline do not execute.

The ROM receives from processor INSTR_ADDR (which is NXPC2) and returns INSTR_DATA; this is the instruction fetch stage. At the next clock posedge the instruction is decoded. When the instruction reaches the execute stage, the PC points to its address (as it should), because at each posedge CLK, PC<=NXPC and NXPC<=NXPC2.

In the decoded stage REG_INSTR_DATA is set, and for every instruction type there is a boolean variable (starts with IS_INSTR_, for example IS_INSTR_JAL) which is set if the current instruction is of that type. If the instruction type is JAL, JALR or branch then the pipeline needs to be flushed and flush_instr_pipe is set accordingly; as a follow-up, each variable corresponding to an instruction type is set to NOP (0), so the data-memory is not read or write anymore and RG[DPTR] is not modified - only modified registers are PC, NXPC and NXPC2 - so, other instructions will be fetched and decoded.

A special case of instruction is LD (load) which reads from data-memory; because it needs four cycles (Instruction Decode; RD=1, DATA_ADDR valid; RAMFF valid (RAMFF is DATA_INPUT in processor); write RAMFF to destination register), a special variable named HLT is set to 1 (set in darksoc.v and used in darkriscv.v) for one clock period, and, as a follow up, REG_INSTR_DATA and the variables which decode the instruction type and PC, NXPC, NXPC2 remain the same; in this period, DATA_INPUT becomes valid and LDATA is set to DATA_INPUT - and on the next period, RG[DPTR] is set to LDATA.

9.4.2 darkriscv C firmware sources

The most important C source are boot.c, io.c and stdio.c. In io.c, it is declared the variable "volatile struct DARKIO *io;" which has the volatile keyword, so the compiler does not make optimizations on it. The boot() function, which is the firmware entry point is shown in the listing below. The io pointer is initiated to address 0x80000000 where the I/O space is mapped in the system.

```
void boot(void)
{
   int tmp=1;

   /* io is used by stdio routines (putchar, printf) */
   io = (volatile struct DARKIO *)(0x80000000);
   io->led = 5;

   while(1)
   {
      banner();
      printf("boot0: text@%d data@%d stack@%d\n",
         (unsigned int)boot, (unsigned int)&utimers, (unsigned int)&tmp+16);
       main();
   }
}
```

Listing 9-9. The boot() function

The instruction "io->led=5;" sets the values of the leds on the FPGA board to 5 (if we implement our system onto an FPGA board, of course).

It is worth noting that in io.h it is defined the struct DARKIO variable which is the type of the io pointer and it establishes, for the firmware, the offsets from the address of the io pointer where the UART, LEDs and GPIOs are mapped.

```
struct DARKIO {

   unsigned char board_id; // 00
   unsigned char board_cm; // 01
   unsigned char board_ck; // 02
   unsigned char irq;      // 03

   struct DARKUART {

      unsigned char  stat; // 04
      unsigned char  fifo; // 05
      unsigned short baud; // 06/07

   } uart;
```

```
    unsigned short led;     // 08/09
    unsigned short gpio;    // 0a/0b

    unsigned timer;         // 0c
};
```

Listing 9-10. The DARKIO structure

The main loop in boot() is showing the RISCV banner, then print the addresses of the text, data and stack sections on the UART console and finally calls main() which prints some messages on the UART console and then:
- ends simulation, in case of simulating, by using the instruction "*((unsigned int*)0x80000004) = 1;" because in darksoc.v when writing at this address it is called the Verilog instruction $finish;
- runs an infinite "while(1);" loop when implementing the system in the FPGA.

In stdio.c, the important functions printf(), getchar() and putchar() are defined. These use the io pointer to access the UART console.

```
int getchar(void) {
  while((io->uart.stat&2)==0); // uart empty, wait...
  return io->uart.fifo;
}
int putchar(int c) {
  if(c=='\n')
  {
    while(io->uart.stat&1); // uart busy, wait...
    io->uart.fifo = '\r';
  }
  while(io->uart.stat&1); // uart busy, wait...
  return io->uart.fifo = c;
}
```

Listing 9-11. The getchar() and putchar() functions

After compiling the sources by issuing the "make" command in the src folder, the darksocv.rom.mem file contains the program code which is detailed in src/darksocv.lst. The code (.text section) starts with the boot() function:

```
Disassembly of section .text:
00000000 <boot>:
   0:   ff010113          addi    sp,sp,-16
   4:   00100793           li    a5,1
   8:   00f12023          sw    a5,0(sp)
```

Listing 9-12. Starting of the .text section

At the beginning of the chapter we have simulated darksocv-e in Icarus and now we have the darksocv.vcd generated file. Using gtkwave, run the command "gtkwave

sim/darksocv.vcd --save=sim/write-save-file.txt.gtkw" (or "gtkwave sim/darksoc.vcd" and add the signals yourself) and see that in simulation, between 3600ns and 3700ns we have:
- RG[2]=00001FF0, which is SP;
- RG[15], which is register a5, is set to 1 because the value of the tmp variable from the boot() function is 1; the code is optimized and the tmp variable is kept in register a5;
- stores RG[15] to SP RAM address, which is RAM[1020] because each RAM word is 32 bits (4 bytes) and SP is $1FF0_{16}=8176_{10}$ = 4096 (address where RAM starts) + 4080 (which is 4*1020).

Exercise 9-1. Configure darkriscv in RV32I variant.

Exercise 9-2. Implement darkriscv processor on the FPGA board.

Exercise 9-3. Implement darkriscv processor in VHDL.

References

1. [Buildroot] Buildroot Linux builder, https://buildroot.org/download.html, last accessed 2020/03.

2. [DarkRV] Marcelo Samsoniuk, DarkRISCV Opensource RISC-V implemented from scratch, https://github.com/darklife/darkriscv, last accessed 2020/04.

3. [FloPoCo] Floating Point Coprocessor – FPGA Arithmetic the way it should be, http://flopoco.gforge.inria.fr, last accessed 2020/04.

4. [I2Cspec] I2C specification, NXP Semiconductors
https://www.nxp.com/docs/en/user-guide/UM10204.pdf, last accessed 2020/03

5. [IEEE754] IEEE Standard for Binary Floating-Point Arithmetic,
https://standards.ieee.org/standard/754-1985.html

6. [Dowson12] J. Dowson, Synthesizable IEEE 754 floating point library in Verilog,
https://github.com/dawsonjon/fpu

7. [FloatConv] IEEE-754 Floating Point Converter,
https://www.h-schmidt.net/FloatConverter/IEEE754.html, last accessed 2020/03.

8. [GhdlGit] Ghdl VHDL simulator, https://github.com/ghdl/ghdl, last accessed 2020/03.

9. [IcarusBleyer] Icarus Verilog for Windows, http://bleyer.org/icarus, last accessed 2020/03.

10. [JSSClib] Java Simple Serial Connector home page,

https://github.com/java-native/jssc/releases, last accessed 2020/03.

11. [PicoRV] Clifford Wolf, PicoRV32 - A Size-Optimized RISC-V CPU, https://github.com/cliffordwolf/picorv32, last accessed 2020/04.

12. [RaspberryPiPins] Raspberry Pi GPIO pins,

https://www.raspberrypi.org/documentation/usage/gpio, last accessed 2020/03.

13. [RV] Andrew Waterman and Krste Asanovic, The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, Berkley University of California, https://riscv.org/specifications/isa-spec-pdf, last access 2020/04.

14. [RVCoreList] RISC-V cores and SoC Overview, https://github.com/riscv/riscv-cores-list, last accessed 2020/04.

15. [Shaaban99] M. Shaaban, Floating Point Arithmetic Using The IEEE 754 Standard Revisited,

http://meseec.ce.rit.edu/eecc250-winter99/250-1-27-2000.pdf

16. [VerilogRef] Verilog HDL quick reference guide,

https://sutherland-hdl.com/pdfs/verilog_2001_ref_guide.pdf, last accessed 2020/03.

17. [VHDLRef] IEEE Standard for VHDL Language Reference Manual,

https://standards.ieee.org/standard/1076-2019.html, last accessed 2020/03.

18. [VirtualBoxSite] VirtualBox virtualization software, https://www.virtualbox.org, last accessed 2020/03.

19. [WikiBooth] Booth's multiplication algorithm,

https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm, last accessed 2020/03.

20. [WishboneSpec] Opencores, WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores,

https://cdn.opencores.org/downloads/wbspec_b4.pdf, last accessed 2020/03.

21. [XilinxLic] Xilinx Product Licensing, https://www.xilinx.com/getlicense, last accessed 2020/03.

22. [XilinxVivado] Vivado Design Suite Evaluation and WebPACK,

https://www.xilinx.com/products/design-tools/vivado/vivado-webpack.html, last accessed 2020/03.

23. [XTEA] The XTEA encryption algorithm, https://en.wikipedia.org/wiki/XTEA, last accessed 2020/04.

24. [Zhangfeifei06] Zhang Fei Fei, UART implementation in Verilog,

https://opencores.org/projects/ucore, last accessed 2020/03.