Creating a Simple CPU Using an FPGA

Artefact

Daniel Myhill

Supervisor: Miss Henfrey

Computer Science

2004 words

June 2023

1 Introduction

Computers have become an integral part of modern life, and so their design is very important. Central Processing Units (CPUs) are an integral part of a computer, executing instructions and controlling the rest of the device. This is why I have chosen to create one, using a Field Programmable Gate Array (FPGA). Due to the extremely complex nature of commercial CPUs, the CPU I have created is far simpler than modern ones, however it still has the same basic functionality. In order to find out the optimal way for me to create this CPU, I investigated the various types of CPU architecture, as well as the hardware implementations I could use. The CPU can perform arithmetic on four stored 32-bit numbers, as well as execute conditional logic and output up to 8 bits at a time using the built in LEDs.

2 Design Process

Before I was able to start creating the CPU, I researched the many types of CPU and I decided to make a Reduced Instruction Set Computer (RISC) CPU on an FPGA.

RISC is a type of CPU architecture in which the of different instructions a computer can execute are kept as simple and minimal as possible while still being able to run any program [1]. This is the alternative to a Complex Instruction Set Computer (CISC), which does not have a limit on the complexity or number of instructions, and so the same program would be described using fewer instructions. Although CISC predates RISC, with each being invented in the 1970s and 80s respectively [1], both are in use today, with large computers such as servers predominantly using CISC, and smaller devices, including smartphones, using RISC [2]. The crucial difference between RISC and CISC for this project is the smaller set of instructions that RISC uses. This allowed less time to be spent on designing an instruction set, and so the focus could be shifted to the CPU architecture itself. It also allowed the CPU architecture to be simpler, as the number of different instructions it was required to execute was smaller.

FPGAs are a type of integrated circuit that contains a large array of configurable logic gates. Logic gates are the most basic building block of any circuit, as each is able to execute extremely simple logic, which can then be connected in order to form a complex circuit. The gates in an FPGA can be configured by the user to create a chip that can do anything it is designed to.

Why I Used an FPGA

I have used a Red Pitaya STEMlab 124-15, which contains a Xilinx Zynq7010 FPGA [3] and other features such as LEDs, to create my CPU. This is due to economic and practical reasons. The re-programmability of FPGAs allowed me to iteratively design and test the CPU architecture, which was extremely practical as I did not have a specific design implementation in mind when I started. This is in contrast to a traditionally produced chip which would have needed to be produced after every change to the design. Additionally, this specific FPGA costs £308 [3], due to the extra functionality it includes which I have not used, while the cost of an FPGA can usually range from $4 for very simple ones to $100,000 for the most high-end versions [4]. In contrast, a traditionally produced chip can cost between $100,000 and $1.5 million upfront, before the chip itself can be mass produced [5].

However, FPGAs are not usually superior to traditional silicon chips. The clock speed of a chip defines the fasted speed at which it can execute instructions. For the FPGA I used, this speed is 125 MHz [3], however chips such as Intel i9s, typically found in modern computers, have clock speeds around 2-6 GHz [6]. This allows them to execute instructions up to 50 times faster. More simple chips also have advantages over FPGAs, namely the cost per unit, as they can be as cheap as $2-3 [5]. For these reasons, FPGAs are commonly used for prototyping, allowing cheap and quick iteration on the architecture of a chip [7].

Despite the drawbacks that are inherent in FPGAs, they are still by far the better option for this CPU as the cost to produce a traditional chip is prohibitive and the possible increase in performance is not necessary.

Design Software

To design the architecture of the CPU, a program called Xilinx Vivado was used. Vivado is an application used to simulate and synthesise designs written in hardware description languages. In this program modules were created using the hardware description language Verilog, created in the early 1980s to describe and simulate electronic circuits in a similar way to how programming languages describe algorithms [8]. Each module describes the function of a specific part of the CPU, such as performing calculations using two numbers, and an example module can be found in Code Listing 1 (see Appendix). These modules were then connected using the tools provided by Vivado, as seen in Figure 2 (see Appendix). Finally, each module and their connections were converted into a set of logic gates by the program, which were then saved into a file. When this file is transferred to the FPGA, the programmable section of the FPGA is connected to form the layout of logic gates described by the file. This allows the FPGA to execute the logic implemented in the modules.

3 Final Design

Architecture

The CPU is able to run programs by following the Fetch-Decode-Execute cycle. First, the CPU ‘Fetches’ an instruction by reading it from where it is stored in memory. Secondly, the instruction is ‘Decoded’, meaning the individual bits of the instruction are interpreted so the

CPU knows what it is meant to do. Finally, the instruction is executed, completing the instruction. Usually, after this is completed, the cycle begins again on the next instruction.

Each part of the cycle runs every time the clock of the CPU activates, which can happen at a rate of up to 3.8kHz. The clock built into the FPGA runs at a rate of 125MHz [3], however it must be slowed down to allow the CPU to execute all of its processes in the time between activations. Although each Fetch-Decode-Execute cycle takes multiple clock cycles to complete, the CPU is still able to execute one instruction per clock cycle by effectively fetching the next instruction while the first instruction is decoding.

Output from the CPU is achieved using the 8 built-in LEDs on the board. Numbers are displayed in binary, with each LED representing a power of 2, from 2^0 on the right to 2^7 on the left. For example, 10 would be represented in binary as 00001010 as it is $2^1 + 2^3$ (2 + 8), therefore the 2nd and 4th LEDs, counting from the right, would be lit when displaying the number 10.

Instruction Set

The instruction set of a CPU is the list of individual instructions that a CPU can execute, and the meaning of each bit in the instruction. For this CPU, each instruction is made up of 32 bits, which was chosen to allow a large array of individual opcodes and options, while still leaving enough space to store large literal values within instructions. The highest 4 represent the instruction itself, allowing 16 unique instructions. For most instructions the next 2 bits represent whether the operand will be interpreted as a literal value or a value in a register. The 2 bits that follow this represent the register the operation should be performed on, and the final 24 bits describe the operand, as shown in Figure 1. See Table 1 for example instructions and their meaning. For the full use of every instruction, see Table 5, in the Appendix.

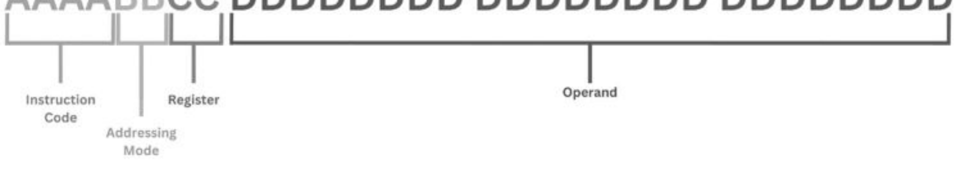

Figure 1. *Typical Instruction Format*

Table 1. *Example Instructions*

Instruction Binary (hexadecimal)	Instruction Meaning
00010100 00001111 01000010 01000000 (140F 4240)	Move the value 1,000,000 into register 0
01110011 00000000 00000000 00000001 (7300 0001)	Add the value in register 1 to register 3
01000000 00000000 00000000 00000010 (4000 0002)	Compare the value in register 0 with register 2
01101000 00000000 00000000 01100100 (6800 0064)	Jump to address 100 if the last comparison was equal

4 Limitations

Although the CPU can fulfil all of its functions as planned, it is still lacking features of commercial CPUs, among other issues.

The most major limitation in its design is the lack of a way to save/load values to memory. Although loading values can be done through using the 'move' instruction, the value must be specified inside the instruction, limiting its value to 16,777,216 while the CPU can handle numbers up to 4,294,967,296. Additionally, the memory location must be run as an instruction, instead of being accessed from afar. Furthermore, values generated by a program cannot be saved to memory in any way, which causes an issue. The CPU only contains 4 registers to store numbers, so any program that requires more than 4 numbers to be stored at once either cannot be run or must be heavily altered.

Another issue the CPU faces is the jump instruction, (instruction 0110). When a jump instruction is followed, the CPU begins reading instructions from the location specified by the instruction, and so the flow of logic should continue from there. However, as multiple instructions have been read into the CPU ahead of their execution to allow parallel processing of the stages of the Fetch-Decode-Execute cycle, the next four instructions after the jump would still be executed. To counteract this, the CPU ignores the next 3 instructions after a jump. This still executes one extra instruction, however this instruction, called a delay slot, is unavoidable without complex logic and is found within many early RISC CPUs, such as the MIPS architecture [9]. Additionally, through stopping unwanted code from running, the CPU sits dormant for 3 clock cycles after jumping, which slows down the CPU somewhat.

Additionally, the output from the CPU is far more limited than commercial CPUs. Most CPUs are able to output values to memory, as well as sending data down the data bus to peripherals, such as monitors which can display the information on a screen. In contrast, this CPU is only able to output data using the 8 LEDs built into the board, which is far less useful.

A final limitation in the CPUs design is the way memory is accessed. Due to the lack of available documentation on the FPGA I used, I was unable to directly access more than two locations in memory. To get around this, I used the built in ARM Cortex-A9 Processor on the

FPGA to move data in its own memory to the locations that could be accessed by my CPU, as requested by the CPU. The program that does this is found in Code Listing 2 (see Appendix). Although this functions the same as directly accessing the memory, it significantly slows the CPU down, as ARM processor cannot move the data as fast as the CPU can process it, and it also means the CPU cannot function without the support of a more capable processor. With sufficient time and documentation I could have made the CPU directly access memory, however I felt that it was not necessary for this project.

5 Examples

Tables 2 and 3 contain example programs for the CPU, and an explanation of each instruction, along with a pseudocode version of the same algorithm. For demonstrations of these programs running on the CPU, refer to videos Example 1 and 2 respectively.

Table 2. *Example Program 1: LED Bouncer*

Instructions	Pseudocode
001 : 1400 0001 # Set r0 to 1	r0 = 1
002 : B400 0001 # Shift r0 left 1	FOREVER:
003 : E000 0000 # Output first 8 bits of r0	WHILE r0 < 128
004 : 1500 000A # Set r1 to 10	SHIFT r0 LEFT 1
005 : 6F00 0064 # Jump to function at address 100	OUTPUT r0
006 : 1700 0007 # Set r3 to 7 (address to return to)	SLEEP 10
007 : 4400 0080 # Compare r0 with binary value 1000 0000	
008 : 6A00 0002 # Jump to address 2 if r0 is less than it	WHILE r0 > 1
009 : 0000 0000 # Delay slot of jump	SHIFT r0 RIGHT 1
010 : C400 0001 # Shift r0 right by 1	OUTPUT r0
011 : E000 0000 # Output r0	SLEEP 10
012 : 1500 000A # Set r1 to 10	
013 : 6700 0064 # Jump to function at address 100	
014 : 1700 000F # Set r3 to 15 (address to return to)	
015 : 4400 0001 # Compare r0 with 1	
016 : 6C00 000A # Jump to address 10 if r0 is greater than it	
017 : 0000 0000 # Delay slot of jump	
018 : 6F00 0002 # Unconditionally jump to address 2	
.	
.	
. # Function that sleeps for approx. r1 ms	
100 : 9500 0122 # Multiply r1 by 122	
101 : A500 03E8 # Divide r1 by 1000	
102 : 7500 0001 # Increase r1 by 1	
103 : 8500 0001 # Decrease r1 by 1	
104 : 4500 0000 # Compare r1 with 0	
105 : 6C00 0067 # Jump to address 103 if r1 is greater than it	
106 : 0000 0000 # Delay slot of jump	
107 : 6700 0003 # Unconditionally jump to address in r3	

Table 3. *Example Program 2: Prime Number Counter*

Instructions	Pseudocode
001 : 1400 0001 # Set r0 to 1	r0 = 1
002 : 7400 0001 # Increase r0 by 1	FOREVER
003 : 4400 0002 # Compare r0 with 2	r0 = r0 + 1
004 : 6900 0007 # Jump to address 7 if r0 is not equal to 2	IF r0 IS NOT 2

```
005 : 0000 0000 # Delay slot of jump
006 : 6F00 0011 # Unconditionally jump to address 17
007 : 1100 0000 # Set r1 to r0
008 : 8500 0001 # Decrease r1 by 1
009 : 1200 0000 # Set r2 to r0
010 : 5200 0001 # Set r2 to the modulus of r2 and r1
011 : 4600 0000 # Compare r2 to 0
012 : 6800 0002 # Jump to address to 2 if r2 is 0
013 : 0000 0000 # Delay slot of jump
014 : 4500 0002 # Compare r1 with 2
015 : 6C00 0008 # Jump to address 8 if r1 is greater than 2
016 : 0000 0000 # Delay slot of jump
017 : E000 0000 # Output r0
018 : 1500 03E8 # Set r1 to 1000
019 : 6F00 0064 # Jump to function at address 100
020 : 1700 0015 # Set r3 to 21 (return address)
021 : 6F00 0002 # Unconditionally jump to address 2
   .
   .
   .                 # Function that sleeps for approx. r1 ms
100 : 9500 0122 # Multiply r1 by 122
101 : A500 03E8 # Divide r1 by 1000
102 : 7500 0001 # Increase r1 by 1
103 : 8500 0001 # Decrease r1 by 1
104 : 4500 0000 # Compare r1 with 0
105 : 6C00 0067 # Jump to address 103 if r1 is greater than it
106 : 0000 0000 # Delay slot of jump
107 : 6700 0003 # Unconditionally jump to address in r3
```

```
  r1 = r0
  WHILE r1 > 2
    r1 = r1 – 1
    r2 = r0 MOD r1
    IF r2 = 0
      NEXT FOREVER LOOP

OUTPUT r0
SLEEP 1000
```

Table 4. *Four Example Outputs of Program 2*

Binary LED output from CPU								**Decimal value of output**
128	**64**	**32**	**16**	**8**	**4**	**2**	**1**	
○	○	○	○	○	○	●	○	2
○	○	○	○	○	○	●	●	3
○	○	○	○	○	●	○	●	5
○	○	○	○	○	●	●	●	7

6 Appendix

Instruction Code	Special Bits				Operand 2	Function
32-29	28	27	26	25	24-1	
0000	MEANINGLESS					No-op, does nothing.
0001	Value type*		Register		Operand 2	Sets the value in Register to the value specified by Operand 2.
0010	MEANINGLESS					No-op, does nothing.
0011	MEANINGLESS					No-op, does nothing.
0100	Value type*		Register		Operand 2	Compares the value in Register with the value specified by Operand 2. (unsigned logic)
0101	Value type*		Register		Operand 2	Sets the value in Register to be the remainder when dividing the value in Register by the value specified by Operand 2.
0110	Value type*	Jump Condition			Operand 2	Jumps to the instruction specified by Operand 2, if the previous comparison met a condition based on the Jump Condition: (The next instruction will run, and so should usually be left blank) 000: Equal 001: Not Equal 010: Less Than 011: Less Than or Equal 100: Greater Than 101: Greater Than or Equal 110: Never 111: Always
0111	Value type*		Register		Operand 2	Adds the value specified by Operand 2 to the value in Register.
1000	Value type*		Register		Operand 2	Subtracts the value specified by Operand 2 from the value in Register.
1001	Value type*		Register		Operand 2	Multiplies the value in Register by the value specified by Operand 2.
1010	Value type*		Register		Operand 2	Divides the value in Register by the value specified by Operand 2, saving the integer part of the result.
1011	Value type*		Register		Operand 2	Shifts the value in Register left by the value specified by Operand 2 (which cannot be negative).
1100	Value type*		Register		Operand 2	Shifts the value in Register right by the value specified by Operand 2 (which cannot be negative).
1101	Value type*		Register		Operand 2	Shifts the value in Register right by the value specified by Operand 2 (which cannot be negative), while preserving the sign.
1110	Value type*		Output settings		Operand 2	Outputs certain bits of the value specified by Operand 2 to the 8 build in LEDs, depending on the Output Settings: 00: bits 1-8 01: bits 9-16 10: bits 17-24 11: bits 25-32
1111	MEANINGLESS					Stops the CPU.

Table 5. *Full instruction set documentation*

*If Value Type is 0, Operand 2 specifies a value in registers 0-3. If it is 1, Operand 2 specifies a literal value.

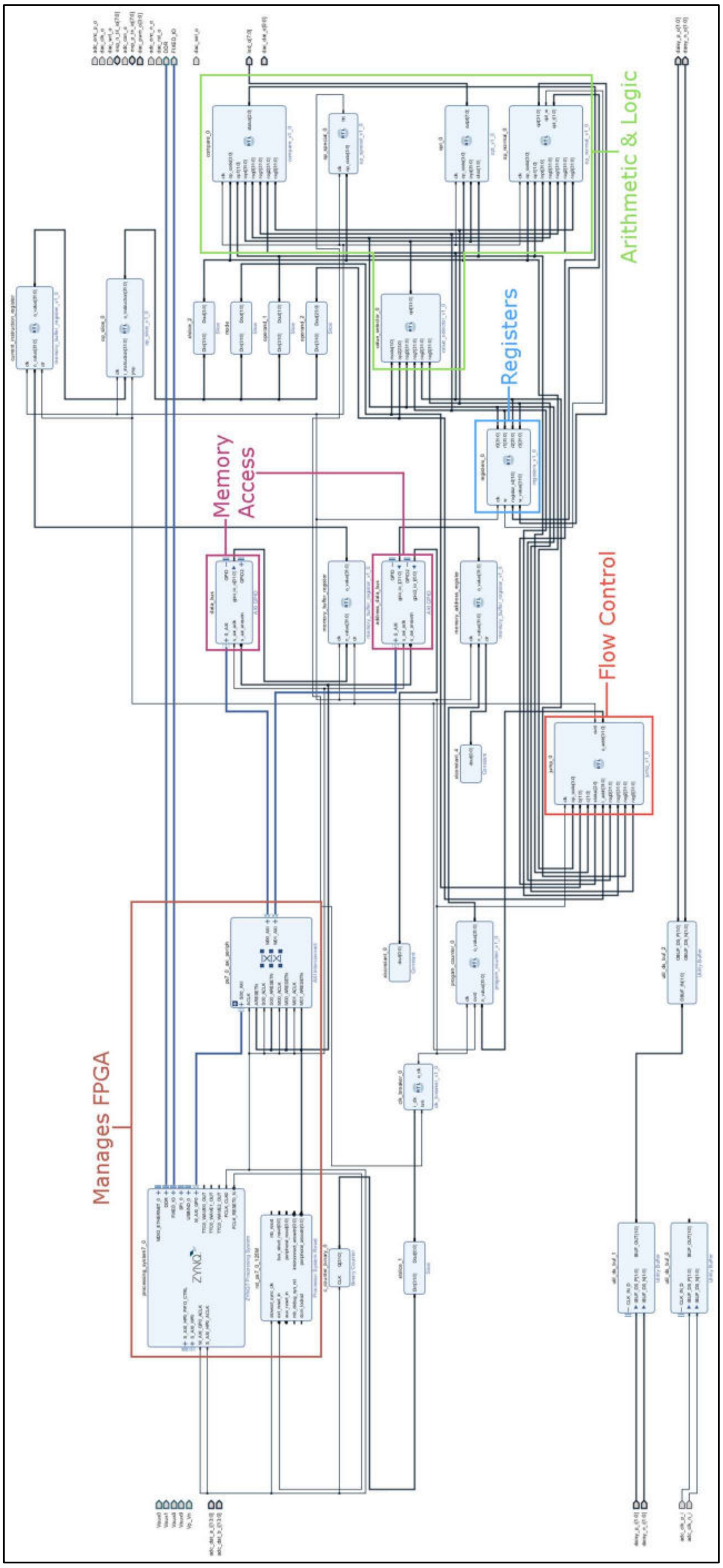

Figure 2. *Vivado Modules and Connections Diagram*

```
module op_normal(input clk, input [3:0] op_code,
                 input [1:0] op1, input [31:0] reg0,
                 input [31:0] reg1, input [31:0] reg2,
                 input [31:0] reg3, output [31:0] opt,
                 output opt_w, output [1:0] opt_r);

     reg [31:0] value = 0;
     reg w = 0;
     reg [1:0] r = 0;

     reg [31:0] stored = 0;

     always @ (posedge clk)
     begin
          w = 0;
          r = op1;
          case (op1)
               0: stored = reg0
               1: stored = reg1
               2: stored = reg2
               3: stored = reg3
          endcase

          case (op_code)
               1: value = inpt;
               5: value = stored % inpt;
               7: value = stored + inpt;
               8: value = stored - inpt;
               9: value = stored * inpt;
               10: value = stored / inpt;
               11: value = stored << inpt;
               12: value = stored >> inpt;
               13: value = $signed(stored) >> inpt;
               default: value = stored;
          endcase
          w = 1;
     end

     assign opt = value;
     assign opt_w = w;
     assign opt_r = r;
endmodule
```

Code Listing 1. *Example Verilog Module*

```
#include <fcntl.h>
#include <signal.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <unistd.h>

static volatile bool keepRunning = true;
const int SIZE = 4000;

void handler(int sig)
{
     keepRunning = false;
     printf("exiting…\n");
}

int main(int argc, char **argv)
{
     int fd;
     int value;
     int w_addr;
     void *data;
     void *addrc;
     char *name = "/dev/mem/";

     signal(SIGINT, handler);

     fd = open(name, O_RDWR);

     if(fd < 0)
     {
          perror("open");
          return 1;
     }

     data = mmap(NULL, sysconf(_SC_PAGESIZE),
                 PROT_READ|PROT_WRITE, MAP_SHARED, fd,
                 0x42000000);

     addrc = mmap(NULL, sysconf(_SC_PAGESIZE),
                 PROT_READ|PROT_WRITE, MAP_SHARED, fd,
                 0x4000000);
```

```
    uint32_t mem[SIZE];

    system("cat /root/cpu.bit > /dev/xdevcfg");

    while (keepRunning)
    {
        int rw = *((uint8_t *)(addrc + 8));
        int addr = *((uint8_t *)(addrc));

        if (rw)
        {
            mem[addr] = *((uint32_t *)(data + 8));
        }
        else
        {
            *((uint32_t *)(data)) = mem[addr];
        }
    }

    munmap(data, sysconf(_SC_PAGESIZE));
    munmap(addrc, sysconf(_SC_PAGESIZE));
    return 0;
}
```

Code Listing 4. *ARM Core Memory Manager Code*

(Based on code obtained from [10]*)*

7 Acknowledgements

I would like to sincerely thank my supervisor, Miss Henfrey, for the feedback and guidance while creating and writing up this project. Additionally, I am very thankful for my computer science teachers, Mr Baker and Mr Gray, for helping me understand the theory behind the design of the project. Finally, I would like to thank Dr Denes for the incredibly useful final feedback on the project.

8 References

[1] P. Kirvan, "RISC (reduced instruction set computer)," Tech Taget, [Online]. Available: https://www.techtarget.com/whatis/definition/RISC-reduced-instruction-set-computer. [Accessed April 2023].

[2] GIGA-BYTE Technology Co., "CISC," [Online]. Available: https://www.gigabyte.com/Glossary/cisc. [Accessed April 2023].

[3] Red Pitaya, "STEMlab 125-14 Board for OEM partners," Red Pitaya, [Online]. Available: https://redpitaya.com/product/stemlab-125-14-board-for-oem-partners/. [Accessed March 2023].

[4] Hillaman Curtis, "FPGA PROGRAMMING AND ITS COST COMPARISON," Hillaman Curtis, [Online]. Available: https://hillmancurtis.com/fpga-programming-and-its-cost-comparison/. [Accessed March 2023].

[5] I. Lankshear, "The Economics of ASICs: At What Point Does a Custom SoC Become Viable?," ElectronicDesign, 15 July 2019. [Online]. Available: https://www.electronicdesign.com/technologies/embedded-revolution/article/21808278/ensilica-the-economics-of-asics-at-what-point-does-a-custom-soc-become-viable. [Accessed March 2023].

[6] Intel, "Intel® Core™ i9 Processors," Intel, March 2023. [Online]. Available: https://www.intel.co.uk/content/www/uk/en/products/details/processors/core/i9/product s.html. [Accessed March 2023].

[7] "FPGA-Based Prototyping," Xilinx, [Online]. Available: https://www.xilinx.com/applications/emulation-prototyping/fpga-based-prototyping.html. [Accessed 27 June 2023].

[8] "Verilog's inventor nabs EDA's Kaufman award," *EETimes,* 11 July 2005.

[9] "MIPS Delay Slot Instructions," Etnus Inc, 2003. [Online]. Available: https://www.jaist.ac.jp/iscenter-/mpc/old-machines/altix3700/opt/toolworks/totalview.6.3.0-1/doc/html/ref_guide/MIPSDelaySlotInstructions.html. [Accessed April 2023].

[10] A. Potočnik, "Red Pitaya FPGA Project 1 - LED Blinker," 6 October 2016. [Online]. Available: http://antonpotocnik.com/>p=487360. [Accessed April 2023].