Master Thesis

Design and Implementation of a Model CPU with Basic Logic Chips and related Development Environment for Educational Purposes

created by
Niklas Schelten
Matrikel: 376314

First examiner:	Prof. Dr.-Ing. Reinhold Orglmeister, Chair of Electronics and Medical Signal Processing, Technische Universität Berlin
Second examiner:	Prof. Dr.-Ing. Clemens Gühmann, Chair of Electronic Measurement and Diagnostic Technology, Technische Universität Berlin
Supervisor:	Dipl.-Ing. Henry Westphal, Tigris Elektronik GmbH

26.04.2022

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 26.04.2022

_____

Unterschrift

Contents

1 Introduction **1**

2 Previous Work **3**

2.1 Design Goal . . . 3

2.2 Implementation . . . 5

2.2.1 Modules . . . 5

2.2.1.1 Arithmetic Logic Unit (ALU) . . . 5

2.2.1.2 Register File . . . 6

2.2.1.3 Control Logic . . . 6

2.2.1.4 Memory . . . 8

2.2.1.5 Program Counter . . . 8

2.2.1.6 Input & Output . . . 8

2.2.1.7 Clock & Reset . . . 8

2.2.2 FPGA Simulation . . . 9

2.2.2.1 Language Choice . . . 9

2.2.3 Hardware Build . . . 9

2.2.3.1 Clock Module . . . 12

3 Design Adaptations **15**

3.1 General Improvements . . . 15

3.2 16bit Addresses . . . 15

3.3 Memory Mapped IO . . . 15

3.4 Stack Implementation . . . 15

3.5 Debugging and Breakpoint . . . 15

4 Software Development Environment **17**

4.1 Micro-Code Generation . . . 17

4.2 Assembler . . . 17

4.2.1 Syntax Definition for VS Code . . . 17

5 FPGA Model **19**

5.1 CPU Architecture Overview . . . 19

5.2 Behavioral Simulation . . . 19

5.3 Behavioral Implementation . . . 19

5.4 Chip-level Implementation . . . 19

5.4.1 Conversation Script . . . 19

6 Hardware Design **21**
6.1 Timing Analysis 21
6.2 Commissioning 21
6.2.1 Test Adapter 21

7 Conclusion and Future Work **23**

Acronyms **25**

List of Figures **27**

List of Tables **29**

List of Code Examples **31**

Bibliography **33**

Abstract

This thesis covers stuff.

Kurzfassung

Diese Arbeit umfasst Zeugs.

1 Introduction

The foundation of this thesis started at the end of 2020 where I decided to design and build a Central Processing Unit (CPU) from scratch. In many university courses we would discuss some parts of a CPU like different approaches to binary adders or pipelining concepts but never would we build a complete CPU including the control logic. Due to a Covid-19 lockdown I had enough time at my hands and after 6 years of study, I felt like I had the expertise to complete this project.

At the end of January 2021 I succeeded with the actual hardware built and the CPU was able to execute a prime factorization of 7 bit numbers. Figure 1.1 depicts the final hardware build. Its design ideas, implementation and flaws are shown in chapter 2.

Through the university module "Mixed-Signal-Baugruppen" I got to know Henry Westphal in summer 2021. He established a company that builds mixed-signal-electronics and, therefore, has a deep understanding of analog and digital circuitry. As he heard of my plans to build a future version of my CPU he was immediately interested and we wanted to rebuild a CPU with some changes:

- The general architecture should remain similar to the existing CPU with only changes where it was necessary.
- The objective was no longer only to create a functioning CPU, this was already accomplished, but the build should be such that it could be used for education.
- It should be more reliable, more capable and its components should be easily distinguishable. Therefore, it is to be build on a large Printed Circuit Board (PCB).
- There should be a generic interface for extension cards, i.e. IO Devices.

How the, now called, Educational Digital Computer (EDiC) differs from its predecessor is presented in chapter 3.

To achieve the goal of the EDiC being educational it is important to not only build the hardware but to also provide a Software Environment to, for example, write applications. This is presented in chapter 4.

An important step in the design of the EDiC was to thoroughly simulate and implement the behavior on an Field Programmable Gate Array (FPGA). I firstly simulated the behavior and after the hardware schematic was finished, we built a script to convert the exported netlist to verilog to simulate the CPU on chip level. The process and

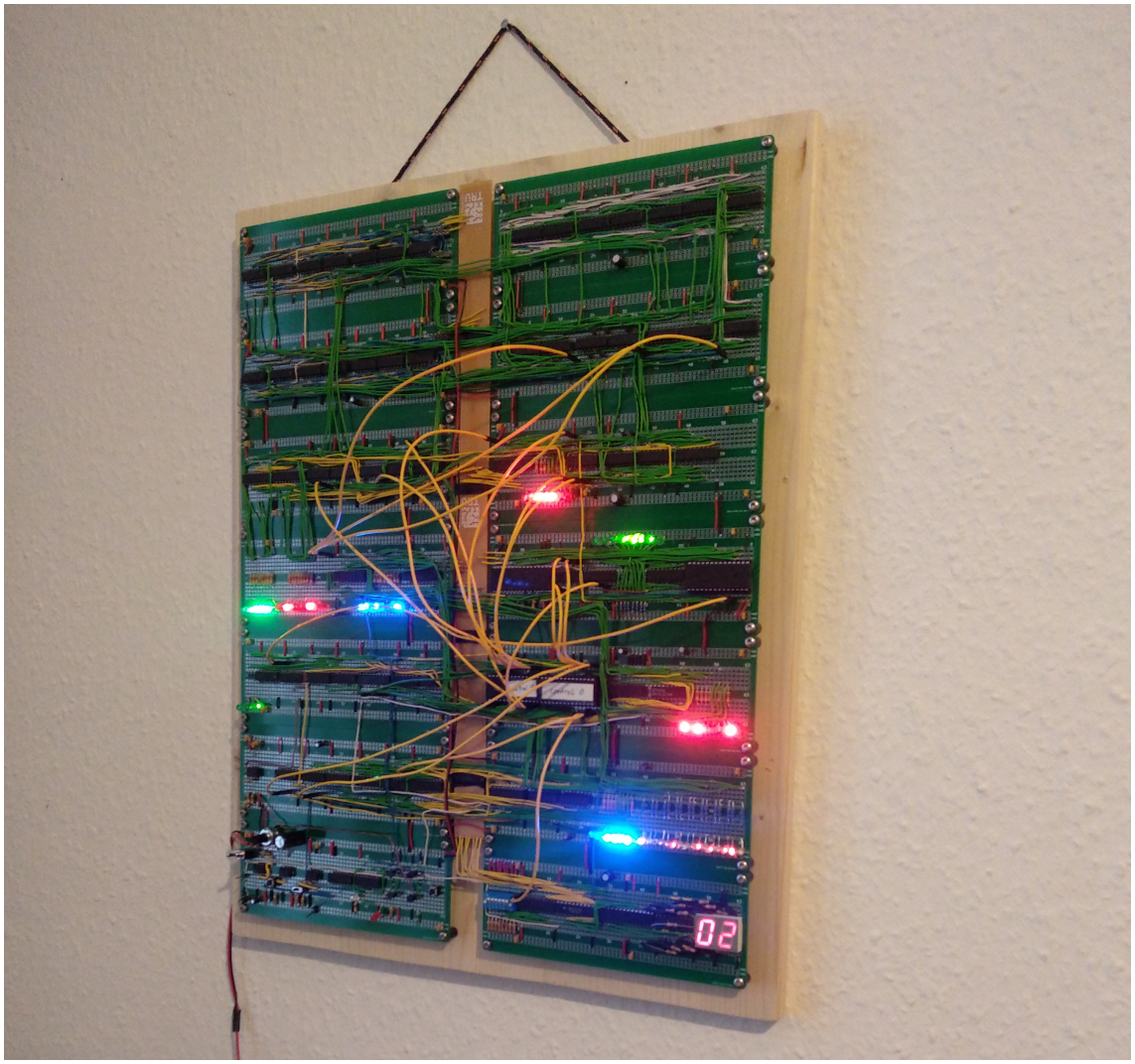

Figure 1.1: The first version of the CPU in its final state.

differences between the FPGA design and the actual hardware are presented in chapter 5.

Chapter 6 describes the final hardware assembly, commissioning and timing analysis to determine the final clock frequency.

The final conclusion and future improvements are given in chapter 7.

2 Previous Work

This chapter provides an overview of what my workings on the initial CPU included.

2.1 Design Goal

My initial goal was to design a CPU from scratch without explicitly looking at how other architectures had solved occurring problems. This meant I was to rely on the knowledge I had achieved until then and came up with the following specifications I wanted my CPU to fulfill:

8 bit bus width Most current era CPUs employ a 32 bit or 64 bit bus to handle large numbers and large amounts of data. It was clear to me that I needed to settle for a smaller bus when building a CPU by hand. Some early CPUs worked with only 4 bits but to not be as limited I finally chose to use an 8 bit bus.

Datapath Architecture - Multicycle CISC In most CPUs an instruction is not done in one clock cycle but it is divided into several steps that are done in sequence. There are two general approaches that are called *Multicycle* and *Pipelining* [7]. Multicycle means that all the steps of one instruction are performed sequentially and a new instruction is only dispatched after the previous instruction is finished. This is usually used when implementing Complex Instruction Set Computers (CISCs), where one instruction can be very capable [1]. For example a add instruction in CISC could fetch operands from memory, execute the add and write the result back to memory. Reduced Instruction Set Computers (RISCs) on the other hand would need independent instruction to load operands from memory into registers, do the addition and write the result back to memory.

In Pipelining there a fixed steps each instruction goes through in a defined order and the intermediate results are stored in so called pipeline registers. Each pipeline step is constructed in such a way that it does not intervene with the others. Therefore, it is possible to dispatch a new instruction each cycle even though the previous instruction is not yet finished. A typically 5-step pipeline would consist of the following steps [7]:

1. **Instruction Fetch**: The instruction is retrieved from memory and stored in a register.
2. **Instruction Decode**: The fetched instruction is decoded into control signals (and instruction specific data) for all the components of the CPU.
3. **Execute**: If arithmetic or logical operations are part of the instruction, they are performed.
4. **Memory Access**: Results are written to the memory and/or data is read from memory.
5. **Writeback**: The results are written back to the registers.

However good the performance of a pipelined CPU is, it also comes with challenges. Those include a greater resource usage because all intermediate results need to be stored in pipeline registers. Additionally, branch instructions[1] pose a great challenge because at the moment, the CPU execute the branch the next instructions have already been dispatched. This means that the pipeline needs to be flushed (i.e. cleared), performance is lost and more logic is required. It also noteworthy that branch prediction and pipeline flushes can be quite vulnerable as recently shown in CVE-2017-5753 with the Spectre bug [2].

Therefore, I decided to build my CPU as a Multicycle CISC.

Single-Bus Oriented The decision for a Multicycle CPU also enabled the architecture to be single-bus oriented. This means that all modules (e.g. the Arithmetic Logic Unit (ALU) and the memory) are connected to a central bus for data transfer. The central bus is then used as a multi-directional data communication. To allow this in hardware, all components that drive the bus (i.e. "send" data) need to have a tri-state driver. A tri-state driver can either drive the bus with a defined '0' or '1' or high impedance which allows other tri-state drivers on the same bus to drive it. That way an instruction which fetches a word from the memory from an address stored in a register and stores it in a register could consist of the following steps:

1. Instruction Fetch
2. Instruction Decode
3. Memory Address from register over *bus* to memory module
4. Memory Access
5. Data from memory module over *bus* to register

With such an architecture it is possible to avoid large multiplexers and keep the overall architecture simple.

[1] Branch Instructions change the program counter and with that the location from which the next instruction is to be fetched. This is required for conditional and looped execution.

Table 2.1: Summary of the available alu operations.

aluOp[1]	aluOp[0]	aluSub	Resulting Operation
0	0	0	$(A + B)$ Addition
0	0	1	$(A - B)$ Subtraction
0	1	0	$(A \wedge B)$ AND
0	1	1	$(A \wedge \overline{B})$
1	0	0	$(A \veebar B)$ XOR
1	0	1	$(\overline{A \veebar B})$ XNOR
1	1	0	$(A \gg B)$ logical shift right
1	1	1	$(A \ll B)$ logical shift left

2.2 Implementation

As mentioned above, the CPU is divided into multiple modules which are only connected over the bus apart from control signals and one other exception.

2.2.1 Modules

The original design was split into 7 independent modules of varying complexity.

2.2.1.1 Arithmetic Logic Unit (ALU)

The ALU is capable of 4 different operations plus inverting the B input for two's complement subtraction. Therefore, there are three control signals which control the operation: two alu-operation bits plus one subtract bit. The B input is XORed with the aluSub bit which results in the B input being inverted when aluSub is '1' and otherwise B remains the same. The aluSub bit is also connected to the carry input of the adder and with that, results in a two's complement subtraction. All possible operations are shown in table 2.1. The adder is a simple ripple carry adder for its simplicity and the XOR and AND operation from the half-adders are also used as the logic operations. A complete 1 bit full-adder is shown in figure 2.1.

It was desirable to include a barrel shifter to have the possibility to improve multiply operation with a shift and add approach instead of repeated addition. The barrel shifter works by 3 consecutive multiplexers to shift by 1, 2 or 4 bit to the right that are controlled by the first 3 bit of the (not inverted) B input. To also allow shifting to the left there is one multiplexer before the three shift multiplexers to invert the order of bits and another

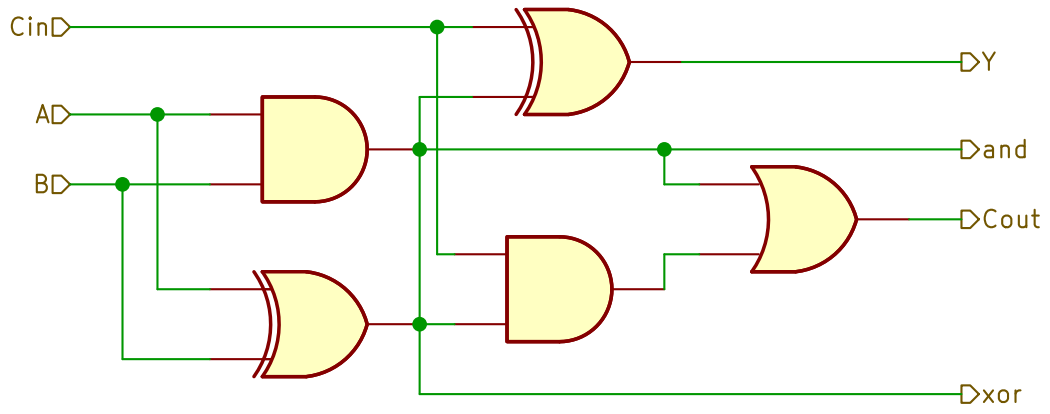

Figure 2.1: 1 bit full adder with the usual A, B and Carry inputs and Y and Carry outputs as well as the XOR and AND outputs.

one after the shifting to reorder the bits. In figure 2.2 a bidirectional barrel shifter implemented with the `74F157` is visualized. The `74F157` implements four 2 to 1 multiplexer and, therefore, 2 chips are needed for a full 8 bit 2 to 1 multiplexer.

The multiplexed result is stored in an 8 bit ALU result register. For conditional execution the ALU includes two flags: Not Zero (at least one bit is one) and negative (the Most Significant Bit (MSB)).

2.2.1.2 Register File

As is typical with CISCs the CPU does not need many general purpose registers and the register file can be kept simple with only two registers. The register file has one write port (from the bus) and two read ports of which one reads to the bus and the other is directly connected to the *A* input of the ALU.

2.2.1.3 Control Logic

The control logic's job is to decode the current instruction and provide all the control signals for each cycle for any instruction. For keeping track which cycle of each instruction is currently executing a 3 bit synchronous counter is needed. Each control signal could be derived by a logical circuitry with 13 inputs: 8 bits instruction, 2 bits ALU flags and 3 bits cycle counter. However, designing these logic circuits is a lot of work, takes up a lot of space and cannot be changed easily later on. (For example when finding a bug in one instruction) Therefore, an Electrically Erasable Programmable Read-Only Memory (EEPROM) is used where the 13 bits that define one cycle of one specific instruction are used as addresses. The control signals then are the data bits of the word that is stored at the specific address in the EEPROM.

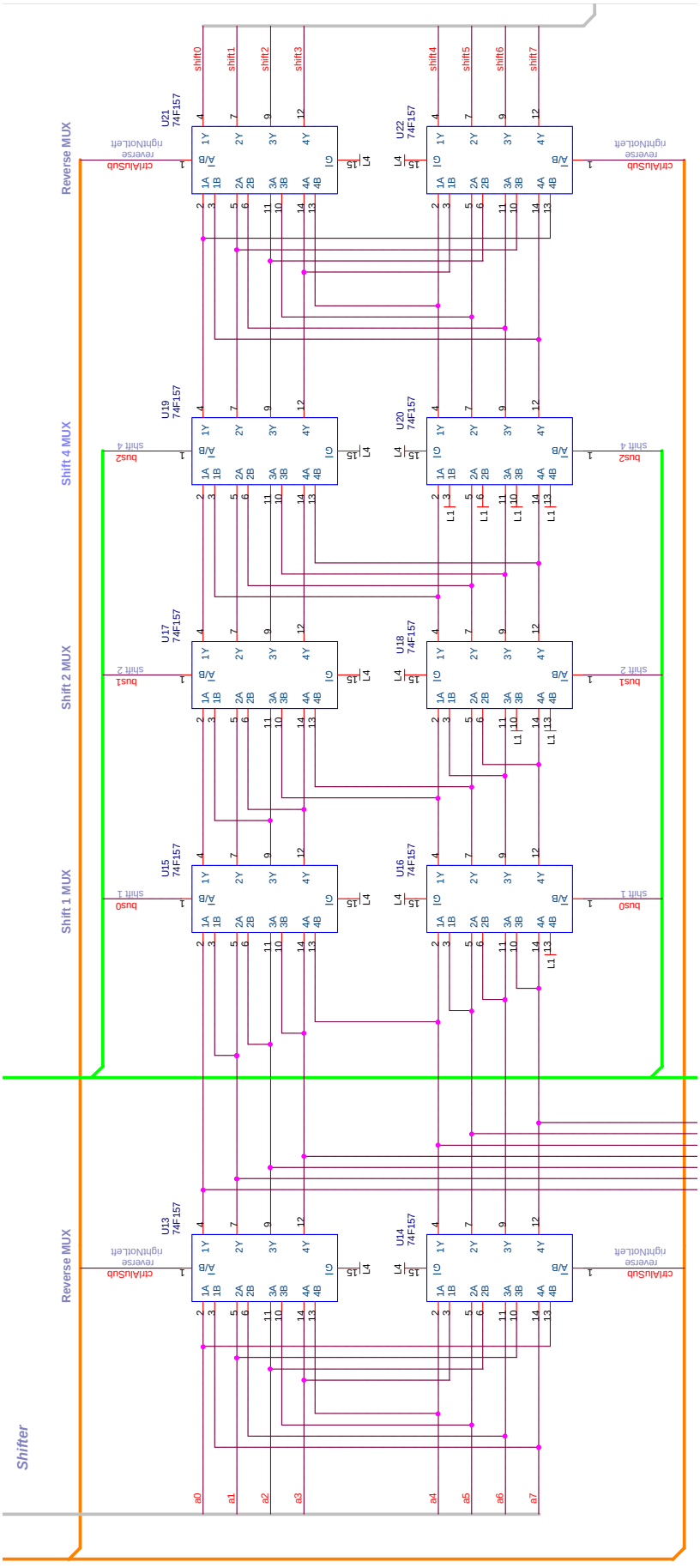

Figure 2.2: 8 bit bidirectional barrel shifter.

The first two cycles of each instruction need to be taken in special consideration because the instruction register is not yet loaded with the next instruction, because it is still being fetched and decoded. However, the instruction fetch and decode are always the same for each instruction, which means that all memory locations where the cycle counter is equal to 0 or 1 (the first two instructions) are filled with the control signals for an instruction fetch and decode.

2.2.1.4 Memory

The memory module contains the main memory of the CPU in form of an asynchronous Static Random-Access Memory (SRAM) and the instruction memory as an EEPROM. This way a new program can be loaded into the CPU by reprogramming the EEPROM. Both the SRAM and the EEPROM have their data lanes connected to the bus for reading from and writing to the memory. The address is controlled via an Memory Address Register (MAR) from the bus.

2.2.1.5 Program Counter

The program counter is a special register with two main functionalities:
Usually it increments by one after each instruction. However, when a branch is executed it needs to load the branch address from the bus. For this an 8 bit increment similar to the cycle counter from the Control Logic section 2.2.1.3 is multiplexed with the bus and used as the input data for the register.

2.2.1.6 Input & Output

This first version of the CPU included very rudimentary I/O logic. For input it provides an 8 bit DIP-Switch connected to the bus and for output there is a register with a 2 digit 7-segment display.

2.2.1.7 Clock & Reset

The function of the clock module is threefold:
It provides a clock for the whole CPU while also providing an active-low reset for some of the registers to provide a defined starting condition. The clock has two modes. One to run continuously and another where the clock can be manually advanced. Additionally, there is a halt instruction which stops the CPU clock until a button is pressed.

2.2.2 FPGA Simulation

The goal of the FPGA simulation is to proof the general workings of the CPU architecture. There was no attempt made to provide a chip-level simulation of the hardware build but rather to provide a top-level behavioral model. The chosen development environment is the AMD - Xilinx Vivado [8] as it is freely available and provides an advanced simulation environment while providing the possibility to synthesize for relatively cheap FPGAs.

One major problem with tri-state bus logic for FPGAs is that most current era FPGAs do not feature tri-state bus drivers in the logic. Most FPGAs do have bidirectional tri-state transceiver for I/O logic but not for internal logic routing. However, the Hardware Description Languages (HDLs) (both VHSIC Hardware Description Language (VHDL) and Verilog) support tri-state logic and the Xilinx Simulation tool also does. As the first CPU was only simulated, this was not a problem and the tri-state logic could be used the same way as in the hardware build. Chapter 5 describes how tri-state logic is solved for the synthesis of the EDiC.

2.2.2.1 Language Choice

There are two main HDLs: Verilog and VHDL. Both are widely supported and used and can also be used in parallel in the same design. At the Technical University Berlin (TUB) VHDL is taught and in Germany it also used more often. However, in general both are used about equally often [6].

As I only knew VHDL and very basic concepts of Verilog, I decided to start in Verilog to get to know the differences. Code Example 2.1 shows the Verilog Code for the ALU module (without the module definition to fit on one page). Lines 24-28 describe the synchronous, positive-edge-triggered alu output register with a write enable. The 8 XOR for the B input are described by lines 39-42 and lines 44-61 show an combinatorial process for the alu operation and multiplexing. The lines 50-58 describe the bidirectional barrel shifter with 3 shift steps (by 1, 2, and 4) and the reverse Multiplexers (MUXs) in front and at the end.

2.2.3 Hardware Build

The idea is to use Integrated Circuits (ICs) of the well known 74 series for the logic functionality. Initially, it was planned to build the CPU similar to the 8-bit CPU project by Ben Eater [4] out of breadboards. However, breadboards make for great prototyping but are known for their not-ideal connectivity and wires can come loose quite easily. Especially when using about 15 boards errors due to bad connections are prone to happen. On the other side, the proper approach would be to design a PCB for the CPU. I decided against it for two main reasons:

```
24 always @(posedge i_clk) begin
25   if (i_aluWr) begin
26     r_y <= s_y;
27   end
28 end
29
30 transmitter inst_tx(
31   .a(r_y),
32   .b(o_y),
33   .noe(i_noe)
34 );
35
36 assign o_negative = r_y[7];
37 assign o_nZero = | r_y;
38
39 genvar i;
40 for (i = 0; i < 8; i++) begin
41   assign s_b[i] = i_b[i] ^ i_subShiftDir;
42 end
43
44 always @* begin
45   case (i_aluOp)
46     2'b00: s_y <= i_a + s_b + i_subShiftDir;
47     2'b01: s_y <= i_a & s_b;
48     2'b10: s_y <= i_a ^ s_b;
49     2'b11: begin
50       s_a = i_subShiftDir
51         ? {i_a[0], i_a[1], i_a[2], i_a[3], i_a[4], i_a[5],
 ↪  i_a[6], i_a[7]}
52         : i_a;
53       s_shift1 = i_b[0] ? s_a >> 1 : s_a;
54       s_shift2 = i_b[1] ? s_shift1 >> 2 : s_shift1;
55       s_shift3 = i_b[2] ? s_shift2 >> 4 : s_shift2;
56       s_y <= i_subShiftDir
57         ? {s_shift3[0], s_shift3[1], s_shift3[2],
 ↪  s_shift3[3], s_shift3[4], s_shift3[5], s_shift3[6],
 ↪  s_shift3[7]}
58         : s_shift3;
59     end
60   endcase
61 end
```

Code Example 2.1: (System)-Verilog Code for the ALU of the first CPU version.

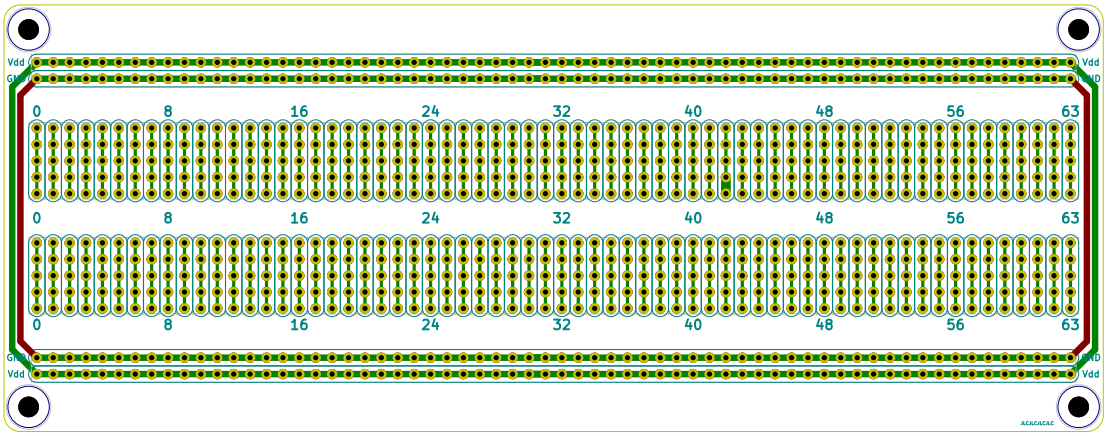

Figure 2.3: PCB used for the hardware built of the first version of the CPU.

Table 2.2: All logic ICs used in the first CPU version.

Quantity	IC	Function
23	74LS157	quad 2 to 1 multiplexer
11	74LS08	quad AND gate
9	74LS86	quad XOR gate
6	74LS32	quad OR gate
6	74LS374	octal register (tri-state output)
5	74LS245	octal bus transceiver (tri-state output)
4	74LS273	octal register with asynchronous clear
4	NA555P	555 timer
3	74LS00	quad NAND gate
3	28C64	32k x 8 bit EEPROM (tri-state output)
1	AS6C1008-55PCN	128k x 8 bit SRAM (tri-state output)

I had little to no experience designing PCBs, from layouting to placing and routing. Additionally, I never worked with the 74 series ICs and wanted to work with them in an easier to change environment similar to breadboards. Secondly, designing such a large PCB would have been very costly compared to what my financial plans for this project were.

Therefore, I decided to find a solution in the middle: A more permanent solution than breadboards but not already fully wired on a big PCB. I designed a small PCB that is very similar to a breadboard with some minor tweaks to make it better suited for my goal. It is shown in figure 2.3. I could order 25 of these boards from JLCPCB[2] for only 34 USD shipped to Germany making it by far the cheapest option.

[2] `https://jlcpcb.com`

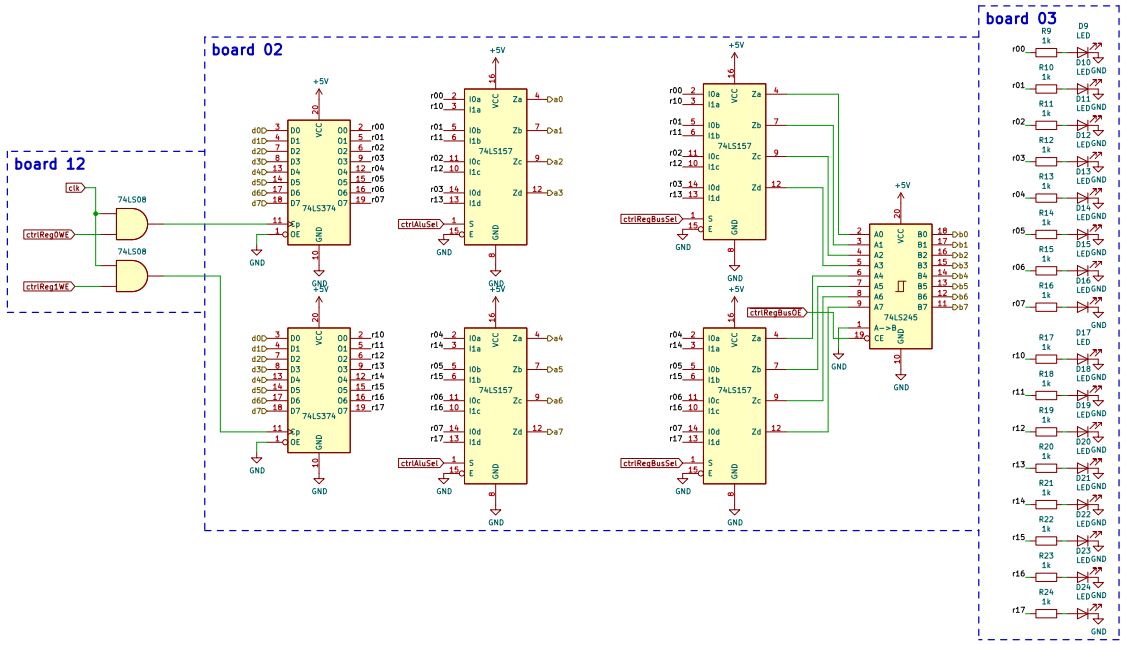

Figure 2.4: Register File with two general purpose register, ALU A input, bus driver and LEDs.

The logic ICs that have been used in the CPU are listed in table 2.2. To make it easier to debug and also to visualize what the CPU is calculating, most registers have Light-Emitting Diodes (LEDs) attached to their outputs via resistors. The layout of the register file is shown in figure 2.4 as an example. It can be seen that one breadboard-like PCB holds 7 logic ICs and the LEDs were placed on another board. The clock pulse of the registers (`r0/1_cp`) comes from another board where the clock has been ANDed with multiple `clockEnable` control signals (not all shown). Problems of this kind of design and also how they were resolved for the EDiC is presented in section 3.1.

2.2.3.1 Clock Module

One clock module which cannot be simulated as well as the others is the clock module. Its circuit is heavily inspired by the clock module of the above mentioned series by Ben Eater [3] which exploits three possible use cases of the well known 555 timer. The 555 timer IC is a massively used IC which features voltage dividers, two comparator, one SR flip-flop, an output driver and a discharge transistor [5]. As shown in figure 2.5 the three use cases for the 555 timer are in the astable (left), monostable (middle) and bistable (right) configuration.

The astable configuration works by charging C6 over R1, R2 and RV1 until a upper threshold voltage is reached which resets the internal flip-flop. This allows the capacitor to discharge to the 555 internal ground until a lower threshold is reached and the

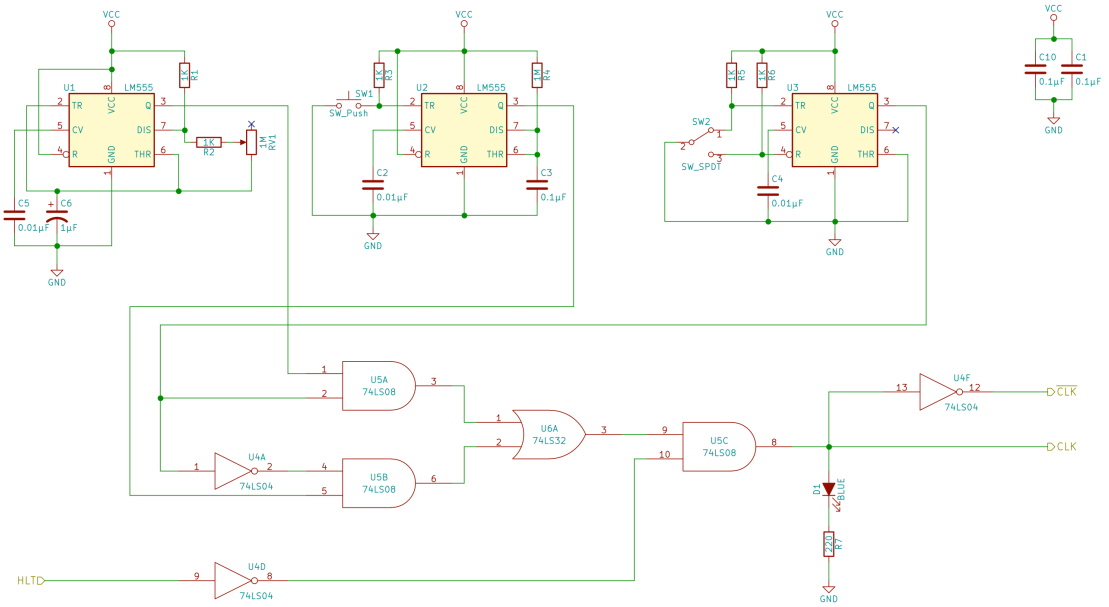

Figure 2.5: The schematic of the clock module by Ben Eater [3] which inspired the clock module of the first version of the CPU.

flip-flop is set again to recharge the capacitor. Depending on the capacitor and resistor dimensions this creates a never ending cycle of sets and resets of the flip-flop and in combination with the output buffer a clock. By using a potentiometer as RV1 it is possible to control the clock frequency[3].

The monostable configuration is used to debounce a button press to be used for stepping the clock one cycle at a time. It creates an active high pulse when the button is pressed once. All consecutive button presses (or bounces) only prolong the pulse but do not trigger another pulse. If the button press would not be debounced it is possible that one press of the button results in multiple edges on the clock line which in turn can result in undefined and unknown behavior. This was a cause for a small bug in the commissioning of the EDiC as presented in section 6.2.

The bistable configuration is used to switch between the two clock sources (astable and monostable). It also uses the internal flip-flop to debounce the switch.

The lower logic gates are used to multiplex the two clock sources and additionally halt the clock when such an instruction is executed.

[3]The duty cycle of the clock is also affected but for the low frequencies I used this circuit (<1kHz) this has no effect on the logic circuit.

3 Design Adaptations

3.1 General Improvements

TODO: LED Driver

TODO: Clock Enable

TODO: 8 bit LEDs/resistors

3.2 16bit Addresses

3.3 Memory Mapped IO

3.4 Stack Implementation

3.5 Debugging and Breakpoint

4 Software Development Environment

4.1 Micro-Code Generation

4.2 Assembler

4.2.1 Syntax Definition for VS Code

5 FPGA Model

5.1 CPU Architecture Overview

5.2 Behavioral Simulation

5.3 Behavioral Implementation

5.4 Chip-level Implementation

5.4.1 Conversation Script

6 Hardware Design

6.1 Timing Analysis

6.2 Commissioning

6.2.1 Test Adapter

7 Conclusion and Future Work

Acronyms

Notation	Description	Page List
ALU	Arithmetic Logic Unit	i, 4–6, 9, 10, 12, 27, 31
CISC	Complex Instruction Set Computer	3, 4, 6
CPU	Central Processing Unit	1–6, 8–13, 27, 29, 31
EDiC	Educational Digital Computer	1, 9, 12, 13
EEPROM	Electrically Erasable Programmable Read-Only Memory	6, 8
FPGA	Field Programmable Gate Array	1, 2, 9
HDL	Hardware Description Language	9
IC	Integrated Circuit	9, 11, 12, 29
LED	Light-Emitting Diode	12
MAR	Memory Address Register	8
MSB	Most Significant Bit	6
MUX	Multiplexer	9
PCB	Printed Circuit Board	1, 9, 11, 12, 27
RISC	Reduced Instruction Set Computer	3

Notation	Description	Page List
SRAM	Static Random-Access Memory	8
TUB	Technical University Berlin	9
VHDL	VHSIC Hardware Description Language	9

List of Figures

1.1 The first version of the CPU in its final state. 2

2.1 1 bit full adder with the usual A, B and Carry inputs and Y and Carry outputs as well as the XOR and AND outputs. 6
2.2 8 bit bidirectional barrel shifter. 7
2.3 PCB used for the hardware built of the first version of the CPU. 11
2.4 Register File with two general purpose register, ALU A input, bus driver and LEDs. 12
2.5 The schematic of the clock module by Ben Eater [3] which inspired the clock module of the first version of the CPU. 13

List of Tables

2.1 Summary of the available alu operations. 5
2.2 All logic ICs used in the first CPU version. 11

List of Code Examples

2.1 (System)-Verilog Code for the ALU of the first CPU version. 10

Bibliography

[1] Crystal Chen, Greg Novick, and Kirk Shimano. *RISC Architecture*. 2000. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/.

[2] *CVE-2017-5753.* 2018. URL: https://www.cve.org/CVERecord?id=CVE-2017-5753.

[3] Ben Eater. *8-bit computer: Clock Module*. 2016. URL: https://eater.net/8bit/clock.

[4] Ben Eater. *Build an 8-bit computer from scratch*. 2018. URL: https://eater.net/8bit.

[5] Doug Lowe. *Electronics All-in-One For Dummies, 2nd Edition*. eng. 2017. ISBN: 978-1-119-32079-1.

[6] NandLand. *VHDL vs. Verilog - Which language should you use for your FPGA and ASIC designs?* NandLand. 2014. URL: https://www.nandland.com/articles/vhdl-or-verilog-for-fpga-asic.html.

[7] David Patterson and John LeRoy Hennessy. *Rechnerorganisation und Rechnerentwurf: Die Hardware/Software-Schnittstelle*. eng. De Gruyter Studium. De Gruyter, 2016. ISBN: 3110446057.

[8] AMD - Xilinx. *Vivado ML Editions - Free Vivado Standard Edition*. AMD - Xilinx. 2022. URL: https://www.xilinx.com/products/design-tools/vivado.html.