



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

10.05.2022

# 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 10.05.2022

_____

Unterschrift

# Contents

# Abstract

This thesis covers stuff.

# Kurzfassung

Diese Arbeit umfasst Zeugs.

# 1 Introduction

This thesis covers the development and engineering process of the Educational Digital Computer (EDiC) pictured in figure 1.1. It is a completely novel Central Processing Unit (CPU) architecture built to visualize and show the fundamental workings of any CPU. The EDiC can execute over half a million instructions per second but also features step-by-step, instruction-by-instruction as well as breakpoint capabilities for better understanding of how CPUs work. All components can be tested individually with the help of dedicated test adapters and, therefore, Integrated Circuit (IC) failures can be tracked down and fixed easily. Additionally to the hardware built, the project includes an open source development environment including an assembler, tools to modify the micro-code and also Field Programmable Gate Array (FPGA) simulation and emulation of the hardware [6].

## 1.1 The Beginning

The foundation of this project started at the end of 2020 where I decided to design and build a 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 1.2 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:

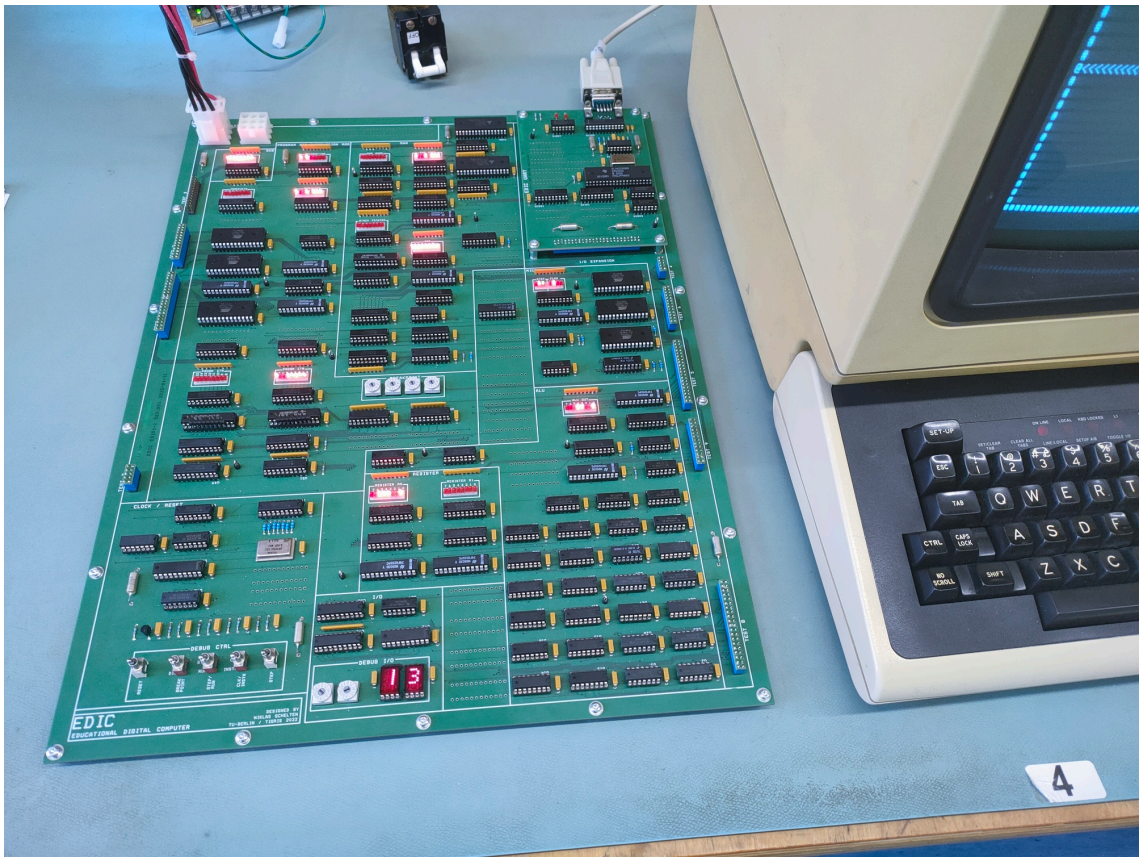



**Figure 1.1:** The final version of the EDiC playing Snake on a VT-100 over an RS-232 I/O card.

- 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, 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

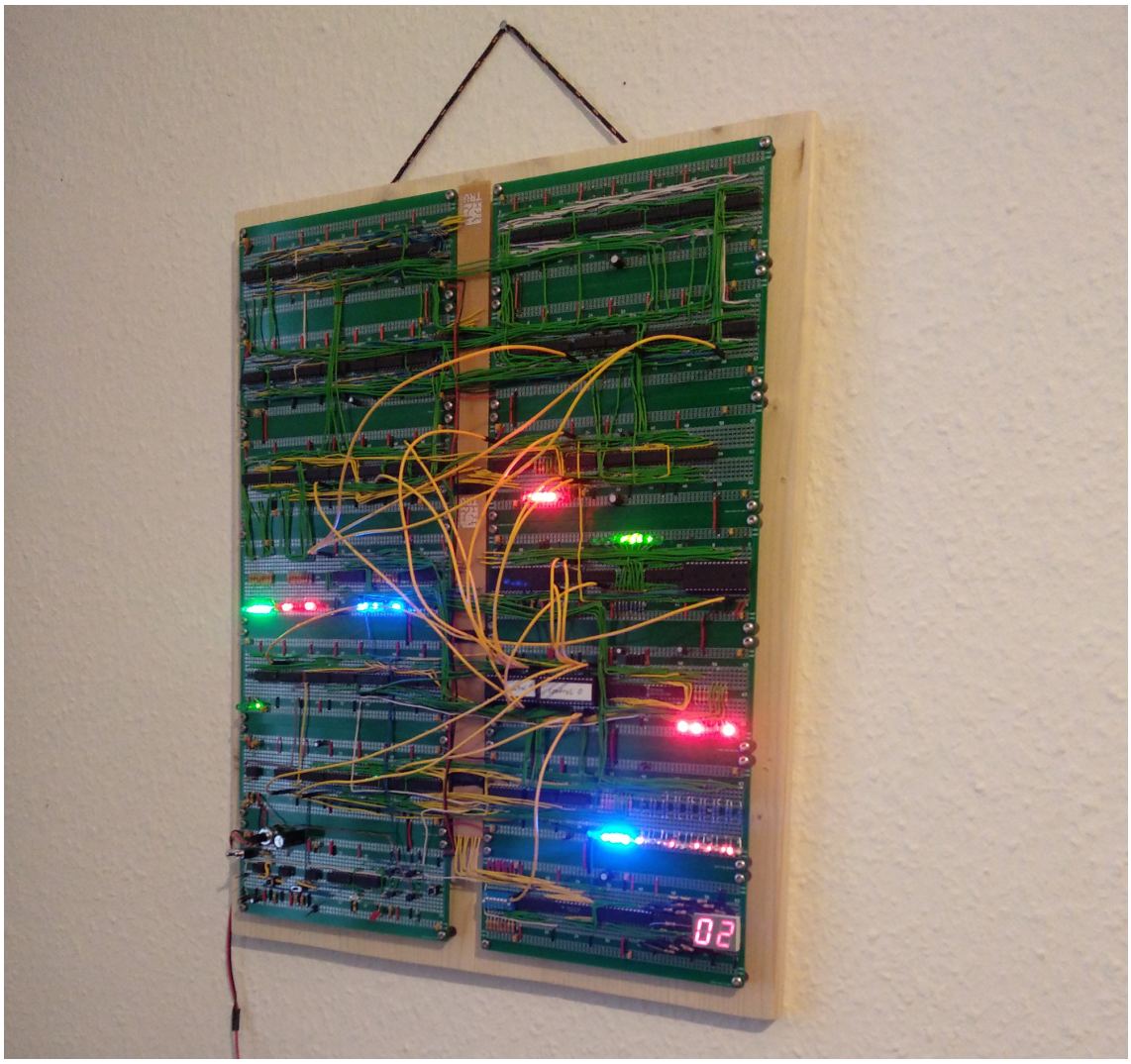


**Figure 1.2:** The first version of the CPU in its final state.

implement the behavior on an 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 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* [11]. 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 [11]:

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 [3].

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] Branch Instructions change the Program Counter (PC) 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 |

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.

## 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

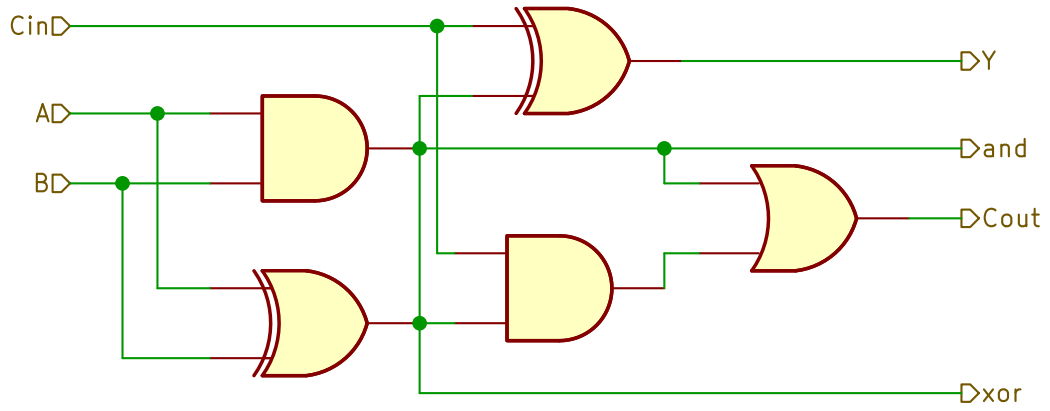



**Figure 2.1:** 1 bit full adder with the usual A, B and Carry inputs and Y and Carry outputs as well as the XOR and AND outputs.

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 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

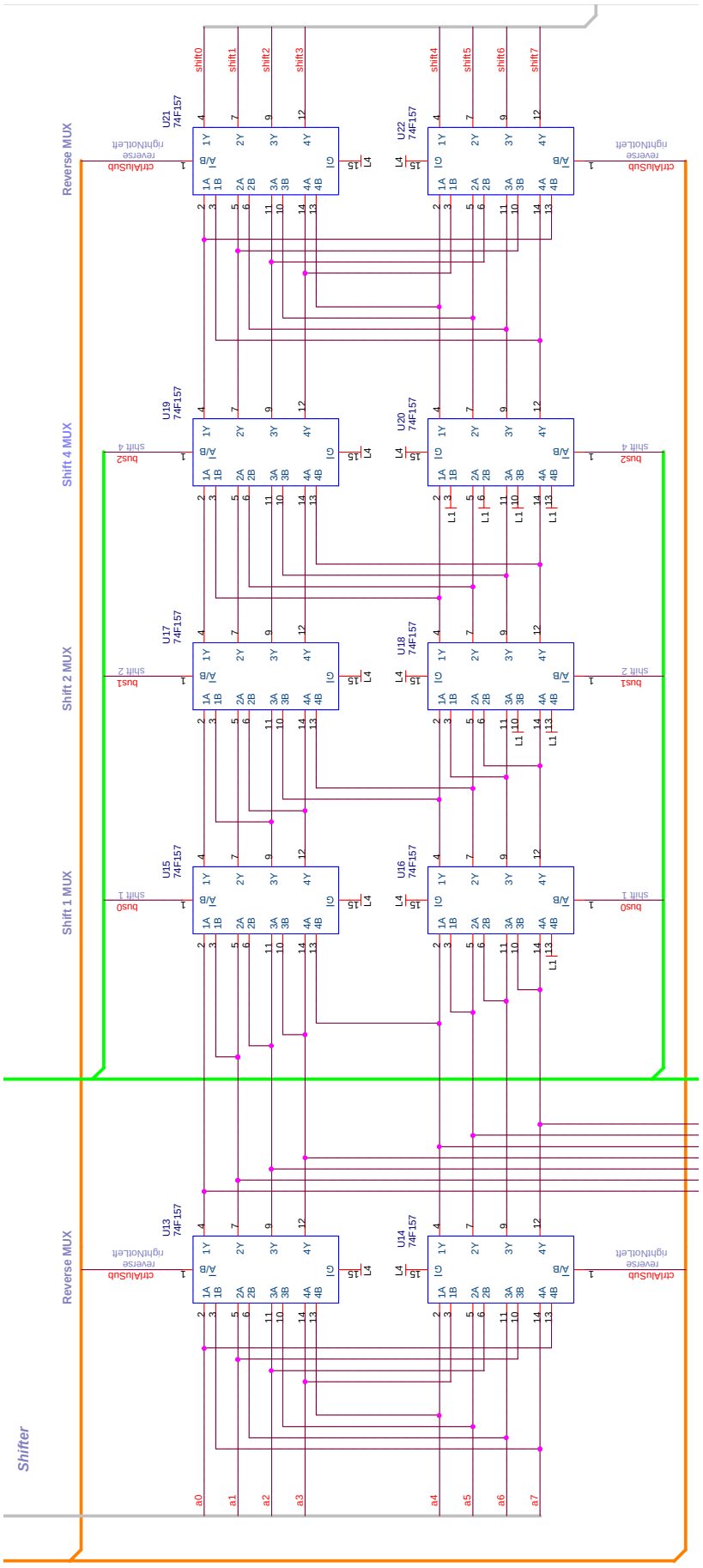



**Figure 2.2:** 8 bit bidirectional barrel shifter.

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.

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 (PC)

The PC 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 [14] 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 [9].

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 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 [5] 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:
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`

```
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.

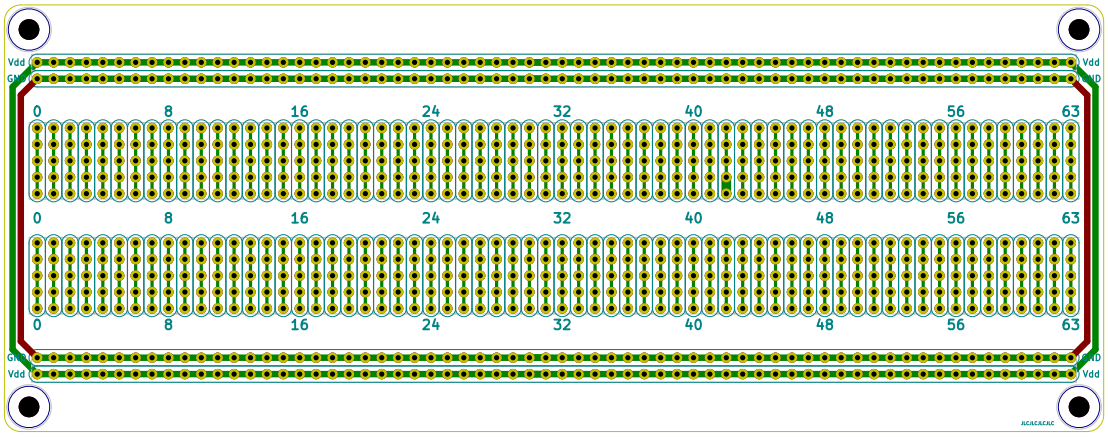



**Figure 2.3:** PCB used for the hardware built of the first version of the CPU.

**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) |

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.

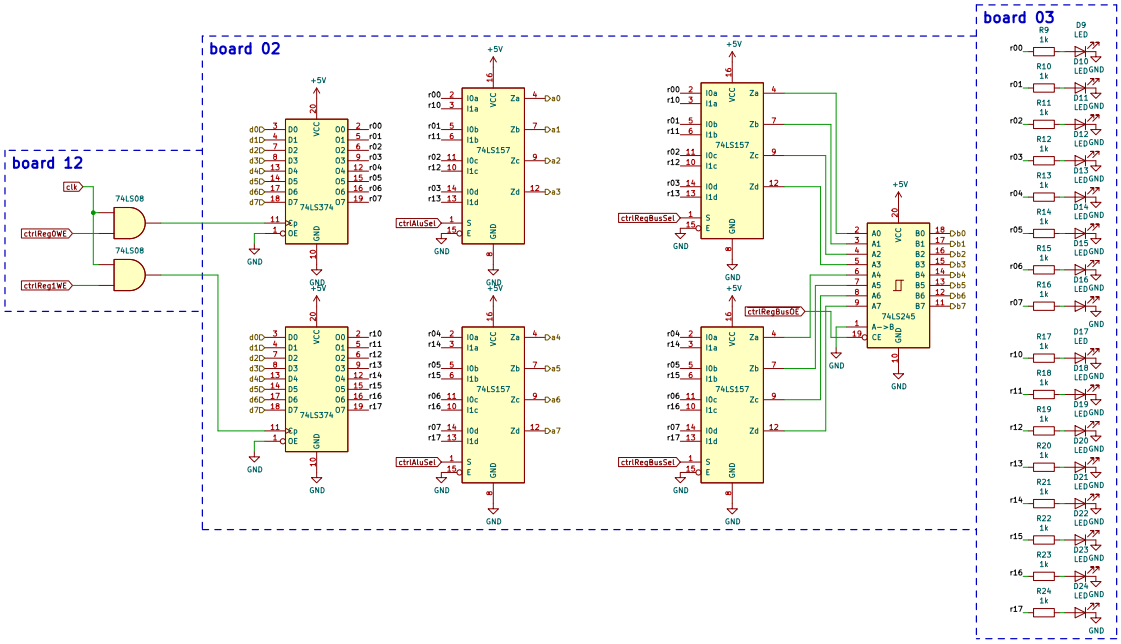



**Figure 2.4:** Register File with two general purpose register, ALU A input, bus driver and LEDs.

#### 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 [4] 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 [8]. 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 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

[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.

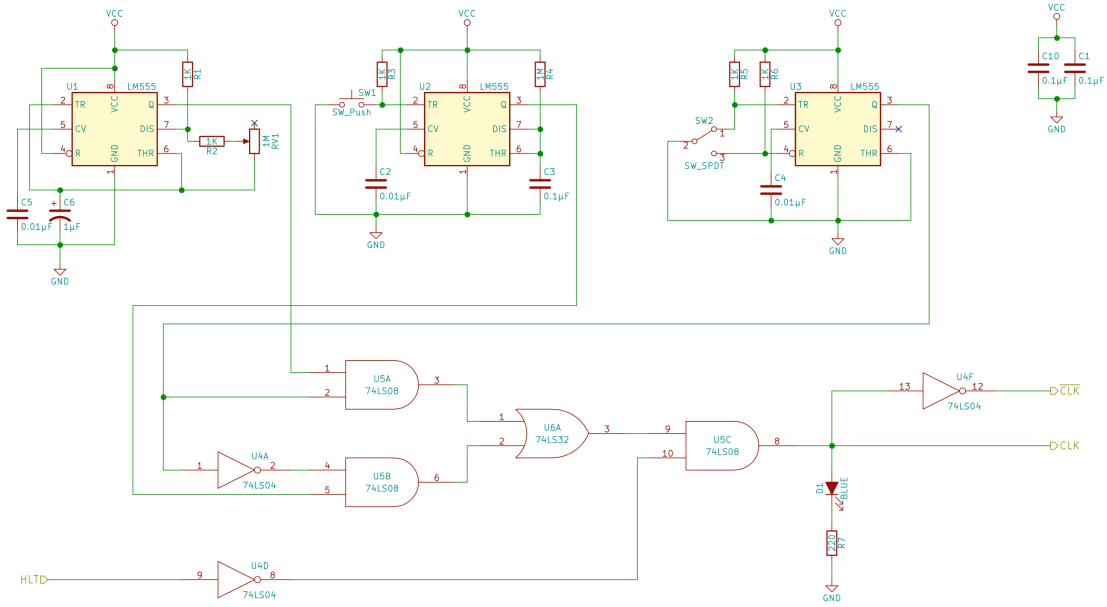



**Figure 2.5:** The schematic of the clock module by Ben Eater [4] which inspired the clock module of the first version of the CPU.

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 Design Adaptations

For building the EDiC the goal was to keep to the same basic infrastructure of the first version CPU. However, many bugs and problems that occurred in the building of it should be addressed and avoided. Furthermore, the CPU should be extended with some important features that were not implemented in the first version.

## 3.1 General Improvements

There are several design decisions, especially in the hardware built, which caused problems or could have caused problems in certain circumstances.

**LED Driver** First of all, all LEDs were directly connected to the logic wires. This does work but the outputs of all logic ICs have a limited current they can provide. For example, the *74LS245* is rated for maximum 20 µA for high-level output and −200 µA for an low-level output [7]. The way the LEDs were connected in the first CPU they use only current when the logic IC outputs a high-level voltage which is rated for 1/10 the output current. When connecting more ICs and one LED the maximum current can easily be exceeded. Therefore, all LEDs of the EDiC are powered with an additional inverting driver, the *74ABT540* which is rated for 50 µA in both directions [12].

**Register IC** The 74 series of logic ICs feature many different registers. The most basic register IC has $n$ D-type flip-flops with respective data inputs and outputs plus one common clock input. On each rising edge of the clock the flip-flops capture the input values and hold them until the next rising edge of the clock. However, often it is required that a register does not capture on every rising edge of the clock. This is done with an additional input, called clock enable. In the first version of the CPU the clock inputs of the registers that needed clock enable were connected to the

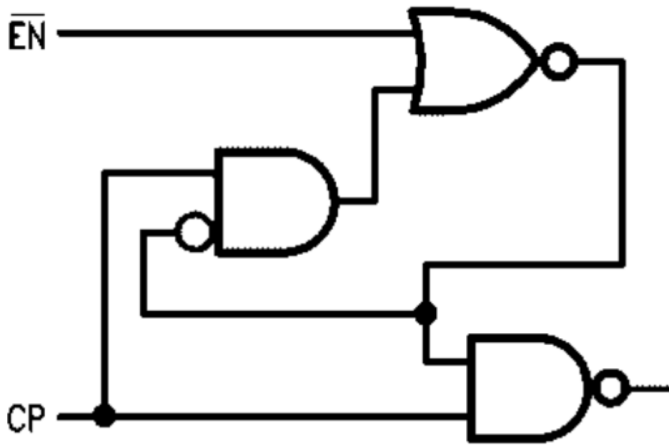
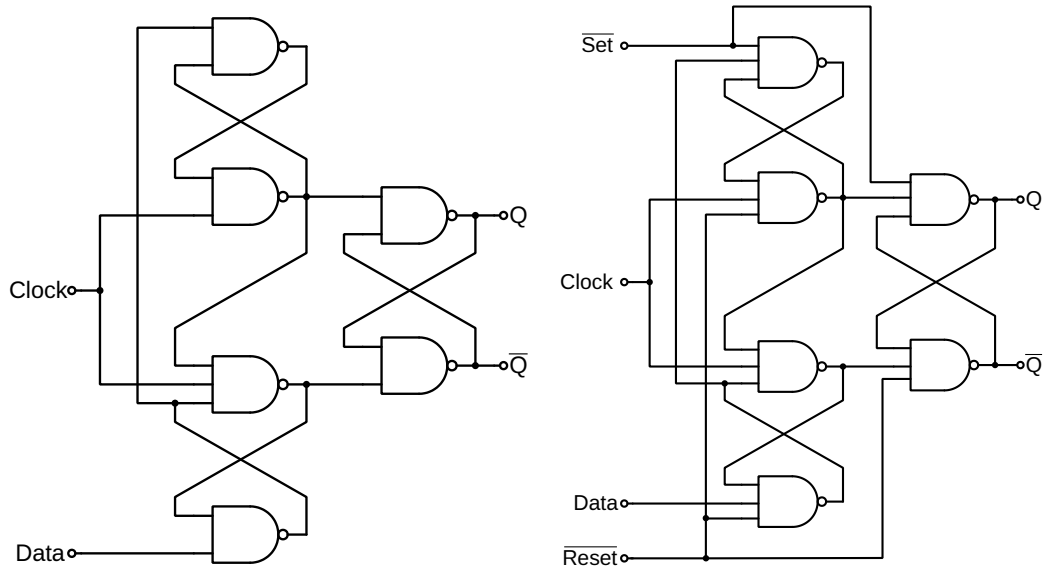



**Figure 3.1:** Clock Enable circuit of the *74F825* IC [2].

output of an AND gate of the clock and a control bit. This has the major drawback that glitches of the enable control signal can propagate to the clock input of the register when the clock is currently high. There are two widely used alternatives to the simple AND gate: The enable input can be used as the select input for an multiplexer to the data input of the flip flop, where it multiplexes between the actual input and the current output. This allows the flip-flop to always capture data but when the enable input is inactive, it recaptures the current output. The drawbacks are that each bit of the register needs a multiplexer at the input and secondly that the flip-flops draw power on every clock pulse, even though no data is captured. The *74F825* logic IC solves this with the circuit shown in figure 3.1. When the $\overline{\text{EN}}$ input is low, the CP input is NAND gate on the right passed the negated CP through[1]. When the $\overline{\text{EN}}$ input is high, on the other hand, the output does not change. This circuit prevents the $\overline{\text{EN}}$ to trigger a falling edge (which would trigger the flip-flops) on the CP output. However, when the $\overline{\text{EN}}$ goes high while the CP input is high, then the output also goes high. This is not directly a problem because the flip-flops only trigger on falling edges but is the reason for timing requirements on the $\overline{\text{EN}}$ input which are discussed in more detail in section 6.1. As the registers store the current state of execution, it is required that the registers start up to a known state. Therefore, some registers feature a clear input (or set input) which forces all flip-flops to 0 (or 1). This is usually accomplished by modifying the classical D-type flip-flop to allow for setting and resetting the internal $\overline{\text{SR}}$ NAND latches as shown in figure 3.2.

The third feature that may be important is a three-state output which allows the register to be directly connected to a bus. It is accomplished by adding a tri-state output driver to the outputs of the flip-flops.

[1]The internal flip-flops of the *74F825* are negative edge triggered




**(a)** Classical D-type flip-flop built out of three $\overline{SR}$ NAND latches [10].

**(b)** D-type flip-flop modified to support $\overline{Clear}$ and $\overline{Set}$ [13].

**Figure 3.2:** Comparison of D-type flip-flops with and without $\overline{Clear}$ and $\overline{Set}$.

The logic IC that was chosen for the EDiC is the *74F825* because it has all three features and is 8 bits wide.

## 3.2 16bit Addresses

One of the major limitation of the 8 bit CPU is the addressable memory space. With only 8 bit for the memory address, the maximum amount of memory addressable is 256 bytes. In the first version of the CPU this limitation was extended a bit by providing 256 bytes of instruction memory besides 256 bytes of read only memory for instruction immediate values and 256 bytes of addressable SRAM. However, especially with a CISC architecture, the limited SRAM memory space limits the overall complexity of programs that can be done. Additionally, more complex programs or even small operating systems are impossible to fit into 256 instructions.

Therefore, it was decided to extend the PC and the memory addresses to 16 bit, which yields 65536 bytes of addressable SRAM and theoretically 65536 instructions[2]. It was decided to not extend the overall bus width to 16 bit because the architectural complexity of the design can increase exponentially with bus width. However, this raises problems of where the 16 bit addresses come from. This is

[2]The largest feasible EEPROM available has only 15 address bits and with that only 32768 words of data.

addressed after explaining the other changes made to the memory module in section 3.5.

## 3.3 Memory Mapped I/O

Input and Output is one of the most important factors of any CPU besides the computing capabilities which are mostly defined by the ALU. Using individual instructions for I/O which directly read from and write to the bus are limiting the usability quite a lot. A common way to extend the I/O capabilities is to use so called Memory Mapped I/O. This works by splitting the address space between actual memory and I/O devices. Then every I/O operation is performed as a usual memory access but the memory chip does not receive the access and the I/O device addressed performs the operation. In the EDiC the memory address is decoded in such a way, that accesses to addresses `0xfe00` to `0xfeff` are performed by any connected I/O devices. For this to work, the lower 8 address bits, the bus and memory control signals - i.e. write enable, read enable and I/O chip enable (active when the upper 8 address bits are `0xff`) - are exposed for I/O devices to connect to.

## 3.4 Stack Implementation

A feature that has been thoroughly missing from the first CPU version is a kind of stack Implementation. The stack is essential to the workings of the programming paradigm *functions*. When calling functions, the return address is usually (automatically) stored on the stack where also local variables can be stored. This allows functions to be called recursively and also simplifies the written assembler compared to simple branching.

However, a typical stack implementation as in modern CPUs architectures like ARM is rather complex. It requires a Stack Pointer (SP) register which needs to be accessible like another general purpose register, including arithmetic operations which is not possible when the bus width is only 8 bits but the SP is 16 bits wide. Therefore, the EDiC uses an unique approach to the stack:

Similarly to the memory mapped I/O it was decided to implement the stack as an 8 bit register which can be incremented and decremented. Every time a memory access is performed where the upper 8 bits of the address equal `0xff`, a 17th address bit is set and the upper 8 address bits are replaced by the current value of the SP. For

example: The SP is currently `0x21` and a memory access to the address `0xff42` is performed. Then the actual address at the memory IC is `0x1_2142`.

This allows each function (which has a unique SP value on the current call stack) to have 256 bytes of function local memory. In the call instruction, the EDiC automatically stores the return address at address `0xffff`, which is `0x1_spff` after translation. To store the whole 16 bit return address, a second memory IC is used in parallel which only needs 256 bytes of storage. In the hardware build of the EDiC the same SRAM IC as for the main memory is used because it is cheaply available. The call and return instructions are further described in section 3.7.

TODO: explain function parameters

## 3.5 Addressing Logic

With increasing the address width to 16 bit and also adding more functionality to the memory access, the addressing logic has become more complex. There are two main sources for memory addresses: The new 16 bit MAR which can be written to from the bus and the 16 bit instruction immediate. As the bus is only 8 bits wide, there is a special instruction to write to the upper 8 bits of the MAR and the lower bits are written in the memory access instruction. This can be used when a memory address is stored in registers and is needed when looping through values in the memory like arrays. When accessing addresses known at compile time, the instruction immediate can be used as an address which has been extended to support 16 bit. These two sources of addresses are then decoded to either select the stack (upper 8 bits equal `0xff`), memory mapped I/O (`0xfe`) or regular memory access. The chip enable of the main memory is only asserted when performing stack and regular memory accesses while the I/O chip enable is only asserted when the upper 8 bits are `0xfe`. Additionally, the 17th address bit is asserted when stack access is performed and the upper 8 bits of the address are replaced with the SP in this case.

## 3.6 Debugging and Breakpoint

An important feature when developing a CPU is debugging capabilities. The initial version could at least step the clock cycle by cycle. As programs get complexer this

feature quickly becomes less useful as each instruction is made of several cycles and when a problem occurs after several hundred instructions it is infeasible to step through all cycles. Additionally, the usual application developer does not want to step through each cycle but rather step through each instruction, assuming that the instruction set works as intended. Another important debugging feature is the use of breakpoints where the CPU halts execution when the PC reaches a specific address.

In the EDiC halting was not realized by stopping the clock completely but rather by inhibiting the instruction step counter increment. This has the advantage that the clock is not abruptly pulled to 0 or 1 and, therefore, no spikes on the clock line can occur. To implement a cycle by cycle stepping mode, the halt signal is de-asserted for only one clock cycle. To step whole instructions, the halt signal is de-asserted until the instruction is finished (marked by a control signal that is inserted at the end of each instruction from the control logic). In breakpoint mode, the halt signal is controlled from a comparator that compares the PC and a 16 bit user input, asserting the halt signal when those two equal. As soon as the CPU halts, the user can then switch to stepping mode and debug the specific instruction of the program.

## 3.7 Final Instruction Set

This section describes all available instructions, what they do and which instruction cycle performs which steps of the instruction. Each instruction starts with the same two cycles for instruction fetching. The following instructions are supported by the hardware:

**ALU operations** The EDiC supports a wide variety of instructions that perform ALU operations. All these operations take two arguments which are used for one of the possible operations shown in table 2.1. Each ALU operation modifies the status flags.

- *Register x Register:* Takes two registers as parameter and the result is stored in the first parameter.

  Cycles:

  1. Both register to ALU A and B input, write enable of ALU result register.
  2. Write content of ALU result register into first parameter register.

- *Register x Register (no write back):* Takes two registers as parameter and the result is only calculated for the status flags.

  Cycles:

  1. Both register to ALU A and B input, write enable of ALU result register.

- *Register x Memory (from Register):* Takes one register as ALU A input and a second register which is used as a memory address for the ALU B input. The result is stored in the first register.

  Cycles:

  1. Second register is stored in the lower 8 bits of the MAR[3].
  2. Address calculations.
  3. First register and memory content as A and B inputs, write enable of the result register.
  4. Write content of ALU result register into first parameter register.

- *Register x Memory (from immediate):* Takes one register as ALU A input and a 16 bit value as immediate which is used as a memory address for the ALU B input. The result is stored in the first register.

  Cycles:

  1. Address calculations.
  2. First register and memory content as A and B inputs, write enable of the result register.
  3. Write content of ALU result register into first parameter register.

- *Register x Memory (from immediate, no write back):* Takes one register as ALU A input and a 16 bit value as immediate which is used as a memory address for the ALU B input. The result is only calculated for the status flags.

  Cycles:

  1. Address calculations.
  2. First register and memory content as A and B inputs, write enable of the result register.

[3]The upper 8 bits of the MAR should be set beforehand

- *Register x Immediate:* Takes one register as ALU A input and an 8 bit value as immediate for the ALU B input. The result is stored in the first register.

  Cycles:

  1. Register and immediate value as A and B inputs and write enable of the result register.
  2. Write content of ALU result register into first parameter register.

- *Register x Immediate (no write back):* Takes one register as ALU A input and an 8 bit value as immediate for the ALU B input. The result is only calculated for the status flags.

  Cycles:

  1. Register and immediate value as A and B inputs and write enable of the result register.

**Memory operations** Some ALU operations also include reading values from memory. The EDiC features a lot more memory operations which are detailed below. As all memory operations may perform memory mapped I/O operations, special care must be taken to allow asynchronous I/O devices to function as well. This means that for each memory access, the address setup and hold must be an individual cycle, resulting in a 3 cycle memory access.

- *Load from register address:* Takes the second register parameter as the lower 8 bits of the memory address and writes the memory content to the first register.

  Cycles:

  1. Second register to lower MAR.
  2. Memory address setup.
  3. Memory read access and write back to first register.
  4. Memory address hold.

- *Load from immediate address:* Takes a 16 bit immediate as the memory address and writes the memory content to the register.

  Cycles:

  1. Memory address setup.
  2. Memory read access and write back to first register.

3. Memory address hold.

- *Load from immediate address with incremented SP:* Takes a 16 bit immediate as the memory address and writes the memory content to the register. However, before the memory access, the SP is incremented and after the access, the SP is decremented again. This is used to access parameters for subfunctions.

  Cycles:

  1. Increment Stack Pointer.
  2. Memory address setup.
  3. Memory read access and write back to first register.
  4. Memory address hold.
  5. Decrement Stack Pointer.

- *Store to register address:* Takes the second register parameter as the lower 8 bits of the memory address and writes the content of the first register to the memory.

  Cycles:

  1. Second register to lower MAR.
  2. Memory address and data setup.
  3. Memory write access.
  4. Memory address and data hold.

- *Store to immediate address:* Takes a 16 bit immediate as the memory address and writes the register content to memory.

  Cycles:

  1. Memory address and data setup.
  2. Memory write access.
  3. Memory address and data hold.

- *Store to immediate address with incremented SP:* Takes a 16 bit immediate as the memory address and writes the register content to memory. However, before the memory access, the SP is incremented and after the access, the SP is decremented again. This is used to access parameters for subfunctions.

  Cycles:

1. Increment Stack Pointer.
2. Memory address and data setup.
3. Memory write access.
4. Memory address and data hold.
5. Decrement Stack Pointer.

- *Set upper 8 bits of MAR from register:* Sets the upper MAR register to the content of the register.

  Cycles:

  1. Register output enable and upper MAR write enable.

- *Set upper 8 bits of MAR from immediate:* Sets the upper MAR register to the 8 bit immediate value.

  Cycles:

  1. Immediate output enable and upper MAR write enable.

**Miscellaneous operations** There are some more operations that are strictly speaking neither ALU nor memory operations like moves and branches.

- *Move between register:* Set the first register to the value of the second.

  Cycles:

  1. Second register output enable and first register write enable.

- *Move immediate to register:* Set the register to the value of the immediate.

  Cycles:

  1. Immediate output enable and first register write enable.

- *Conditionally set PC from immediate:* This is the only conditional operation available. Depending on the current status register the following cycles are either executed or No Operations (NOPs) are executed.

  Cycles:

  1. PC write enable from immediate.

- *Function Call:* Takes a 16 bit address which the PC is set to. The SP is incremented and the return address is stored on the stack.

  Cycles:

1. Increment SP and write `0xffff` into the MAR.
2. Memory address and data (PC) setup.
3. Memory write access.
4. Memory address and data hold.
5. Load PC from instruction immediate.

- *Function Return:* Decrements the SP and the PC is loaded from the return address which is read from the memory.

  Cycles:

  1. Write `0xffff` into the MAR.
  2. Memory address setup.
  3. Memory read access and PC write enable.
  4. Memory address hold.
  5. Decrement SP.

# 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, 6–8, 10, 12, 13, 15, 20, 22–24, 26, 39, 43 |
| CISC | Complex Instruction Set Computer | 5, 6, 8, 19 |
| CPU | Central Processing Unit | 1–3, 5–8, 10–14, 16, 17, 19–22, 39, 41, 43 |
| EDiC | Educational Digital Computer | 1, 2, 11, 14, 16, 17, 19–22, 24, 39 |
| EEPROM | Electrically Erasable Programmable Read-Only Memory | 10, 19 |
| FPGA | Field Programmable Gate Array | 1, 3, 11 |
| HDL | Hardware Description Language | 11, 12 |

| **Notation** | **Description** | **Page List** |
|---|---|---|
| IC | Integrated Circuit | 1, 12, 14, 15, 17–19, 21, 39, 41 |
| LED | Light-Emitting Diode | 14, 17 |
| MAR | Memory Address Register | 10, 21, 23–27 |
| MSB | Most Significant Bit | 8 |
| MUX | Multiplexer | 12 |
| NOP | No Operation | 26 |
| PC | Program Counter | i, 6, 10, 19, 22, 26, 27 |
| PCB | Printed Circuit Board | 2, 12, 14, 39 |
| RISC | Reduced Instruction Set Computer | 5 |
| SP | Stack Pointer | 20, 21, 25–27 |
| SRAM | Static Random-Access Memory | 10, 19, 21 |
| TUB | Technical University Berlin | 12 |
| VHDL | VHSIC Hardware Description Language | 11, 12 |

# List of Figures

# List of Tables

# List of Code Examples

# 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] Fairchild Semiconductor Corporation. *74F8258-Bit D-Type Flip-Flop*. 2000. URL: https://rocelec.widen.net/view/pdf/d4zabtdsls/FAIRS08275-1.pdf?t.download=true&u=5oefqw.

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

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

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

[6] *Git Repository of the EDiC developement*. URL: https://github.com/Nik-Sch/EDiC.

[7] Texas Instruments. *SNx4LS245 Octal Bus Transceivers With 3-State Outputs*. 1976. URL: https://www.ti.com/lit/ds/sdls146b/sdls146b.pdf.

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

[9] 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.

[10] Nolanjshettle. 2013. URL: https://en.wikipedia.org/wiki/File:Edge_triggered_D_flip_flop.svg.

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

[12] Philips Semiconductors. *74ABT540 Octal buffer, inverting (3-State)*. 1998. URL: https://www.mouser.com/datasheet/2/302/74ABT540-62406.pdf.

[13] Stunts1990. 2020. URL: https://en.wikipedia.org/wiki/File:Edge_triggered_D_flip_flop_with_set_and_reset.svg.

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