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

20.04.2022

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 20.04.2022

_____

Unterschrift

Contents

Acronyms

Notation	Description	Page List
ALU	Arithmetic Logic Unit	i, 4–6
CISC	Complex Instruction Set Computer	3, 4, 6
CPU	Central Processing Unit	1–6, 19
EDiC	Educational Digital Computer	1
EEPROM	Electrically Erasable Programmable Read-Only Memory	6
FPGA	Field Programmable Gate Array	1, 2
MSB	Most Significant Bit	6
PCB	Printed Circuit Board	1
RISC	Reduced Instruction Set Computer	3

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

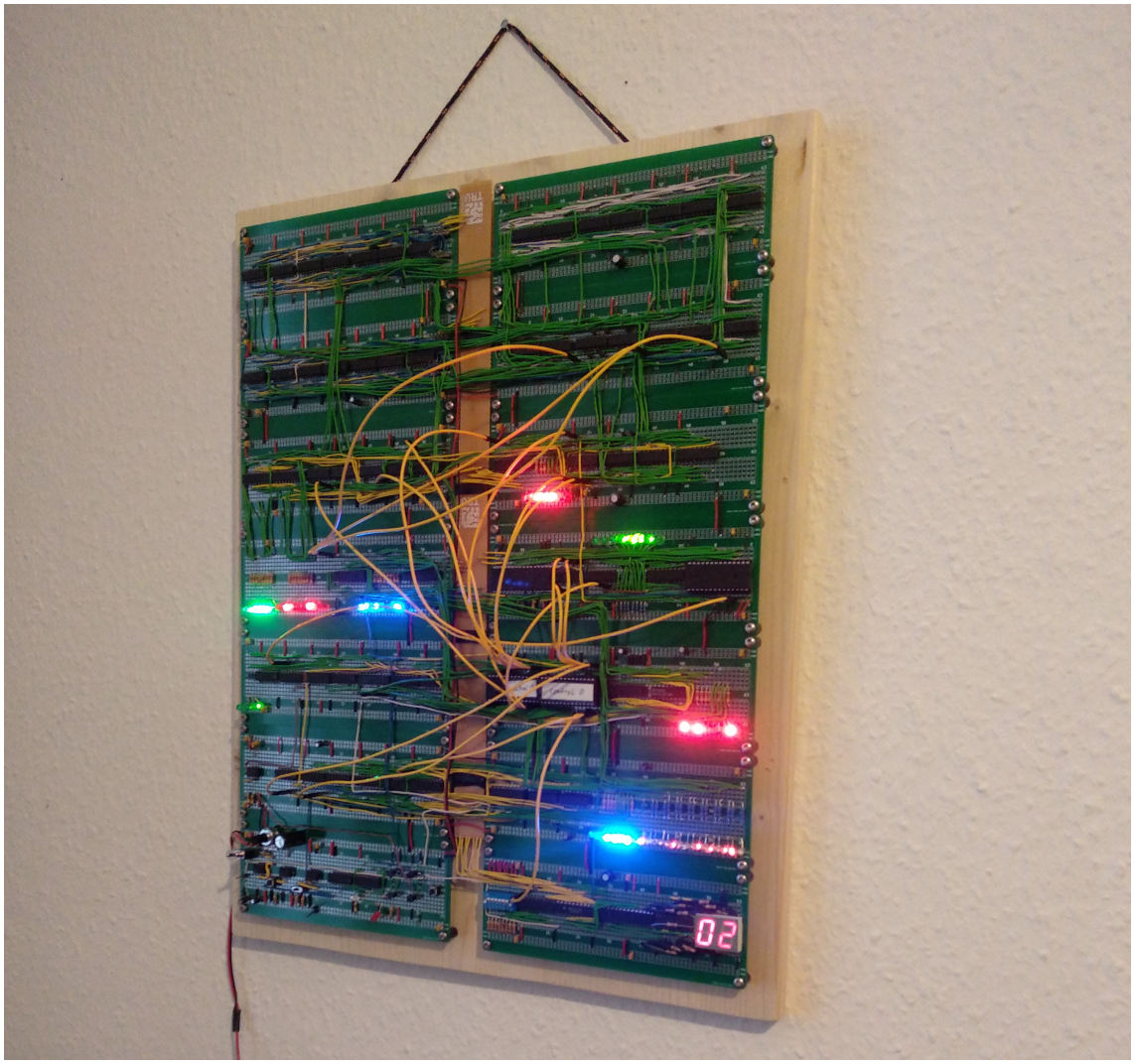

Figure 1.1: The first version of the CPU in its final state.

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* [3]. 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 [3]:

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. 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.

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)).

TODO: include schematics for barrel shifter and full adder

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.

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

2.2.1.5 Program Counter

2.2.1.6 Input & Output

2.2.1.7 Clock

2.2.2 FPGA Simulation

2.2.3 Hardware Build

3 Design Adaptations

3.1 16bit Addresses

3.2 Memory Mapped IO

3.3 Stack Implementation

3.4 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

List of Figures

List of Tables

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