# The EuroTwo v0.004 Manual

maximilian noppel

28. April 2020

**The following document is WORK IN PROGRESS!**

As compared to the document on the EuroOne this project is also WORK IN PROGRESS. This EuroTwo is much smaller and simpler than EuroOne and the logic was created and simulated within 4 hours. The goal of this project is to implement exactly the CPU described here from Hardware. Therefore I will use relays, resistors, capacitors and some inductions. Maybe small part will be implement with external ICs, for example the RAM and the ROM. This makes it easier to visualize the RAM and and program the ROM. Maybe i will also add a MIR to address MMIOs (UART, ...).

The simple ALU used in this project is already in production as a simple preproject to gain same experiences with this kind of project.

Well, enjoy reading!

## Inhaltsverzeichnis

## 1 Features

- Harvard-Architecture
- Four 4-bit generalpurpose-registers
- One 12-bit programcounter-register
- fixed 16-bit instruction length
- fixed 1 cycles per instruction
- $2^8$ x 4-bit RAM
- $2^1 2$ x 16-bit ROM for instructions
- Only handles unsigned instructions
- Every instruction runs in 1 clock cycle

## 2 Introduction

This documents describes the EuroTwo CPU. The little sister to the bigger EuroOne CPU I simulated in logisim. As the EuroOne projects grows bigger and bigger my goal to implement it in hardware from basic components is no longer a way to go. That's why I started this project of a more simplier and smaller CPU. Also I regained the motivation to implement it from relays, transistors (TTL) or much cooler LED-Transistor-Logic. In the following sections I briefly describe the different components of the CPU and how the work.

## 3 Basic Architecture

The architecture basically is a harvard architecture. This give my the possibility of a much bigger memory for instructions than for the RAM. Also the buswidth can be different. This is nice because I wanted to have 4-bit register. But I dont want to hasle with loading 4-bit words in the instruction registers because I would therefore need even more 4-bit registers. Now I only have 12-bit programcounter pointing to the ROM which provides 16-bit of data to the ControlUnit. The ControlUnit is also not storing the instruction because we have got a separat bus for the execution. This saves components and execution time.

The architecture consists of 3 parts:

- The ControlUnit (CU)
- The ALU
- The Backplane

In the following section I will describe any of them I some details, if I am motivated to write stuff... :D

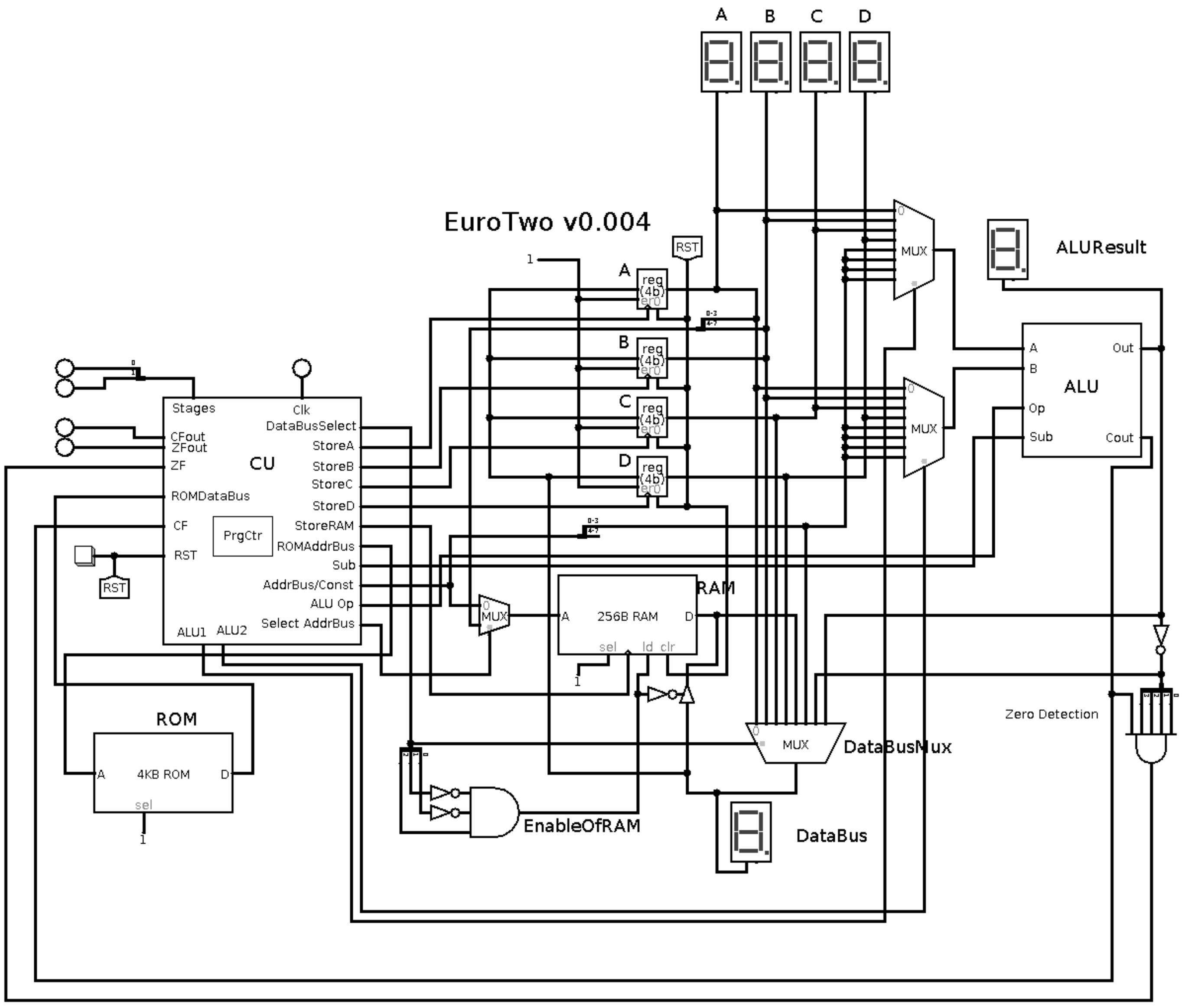

Abbildung 1: Draft of the CPU-architecture

## 4 ALU

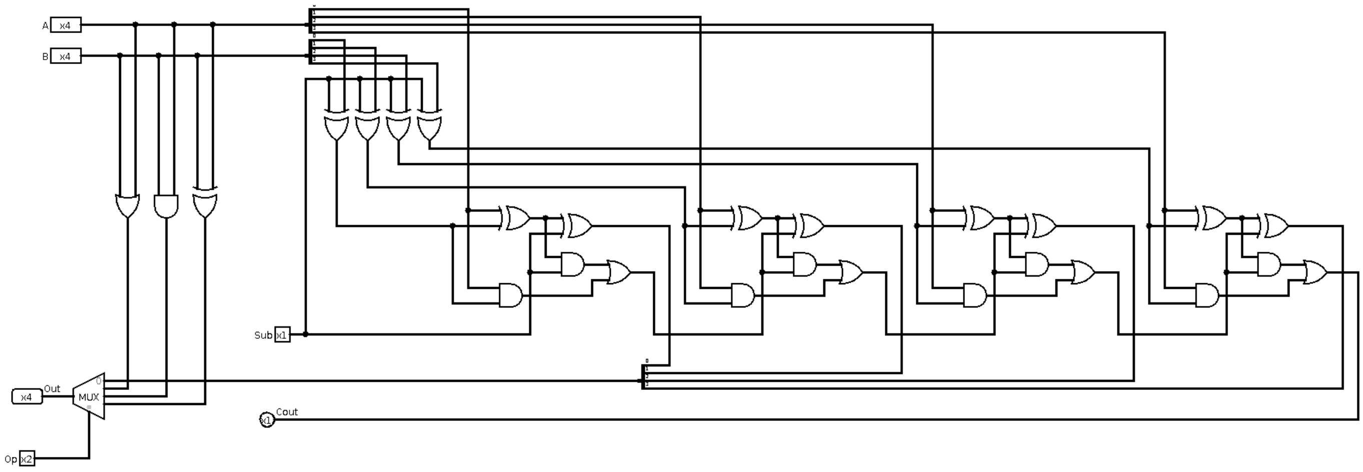

Abbildung 2: Draft of the ALU

### 4.1 Flags

The ALU is outputting the following signals to the CU:

- `CF`: Carry Flag
- `ZF`: Zero Flag

## 5 The Instruction Set

In the following table I present the very simple instruction set for the EuroTwo CPU. One can read the binary representation of the single instruction from figure 3 so I will just give some examples, so that the reader can check if she did it right. There are four types of instructions:

- NoOperation Instruction
- Jump Instruction
- DataFlow Instruction
- Arthm. Instruction

I provide a comprehensive list of all supported instructions with their binary represenation in the appendix **??**. Here comes the examples:

| Instruction | Formal definition | Description |
|---|---|---|
| `ADD r1 r2 r3` | $r3 := r1 + r2 (\mod 16)$ | Sets CF and ZF |
| `SUB r1 r2 r3` | $r3 := r1 - r2 (\mod 16)$ | Sets CF and ZF |
| `ADDC r1 const r2` | $r2 := r1 + c (\mod 16)$ | Sets CF and ZF |
| `SUBC r1 const r2` | $r2 := r1 - c (\mod 16)$ | Sets CF and ZF |
| `MOV r1 r2` | $r2 := r1$ | Move |
| `LD addr r1` | $r1 := [addr]$ | Load from RAM |
| `LDR r1` | $r1 := [rb\|\|ra]$ | Load from RAM |
| `LC const r1` | $r1 := c$ | Load Constant |
| `STR r1 addr` | $[addr] := r1$ | Store to addr |
| `STRR r1` | $[rb\|\|ra] := r1$ | Store to $rb\|\|ra$ |
| `JMP` | $PC := addr$ | |
| `JZ` | $if\ ZF : PC := addr$ | JumpZero, JumpEqual |
| `JA` | $if\ (\neg CF \wedge \neg ZF) : PC := addr$ | JumpGreater |
| `JB` | $if\ CF : PC := addr$ | JumpLess |
| `CMP` | | Sets CF and ZF |
| `NOP` | | No operation |

Tabelle 1: Instruction Set

Every instruction is running in one cycle!

## 6 ControlUnit

The ControlUnit is decoding the incoming instruction. It does not use a instruction register. As I used the harvard architecture and only clock to the PC at the next instruction cycle it simple uses the output of the ROM to decode the instructions. As all instructions are 16-bit long this is quite simple. The following table documentates how the single bit get interpretet:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | Addr | | | | | | | | | | | |
| 0 | 1 | 0 | 1 | Addr | | | | | | | | | | | |
| 0 | 1 | 1 | 0 | Addr | | | | | | | | | | | |
| 0 | 1 | 1 | 1 | Addr | | | | | | | | | | | |
| 1 | RAM Store | | | Reg DB | | | 0,1 | Addr | | | | | | | |
| 1 | Reg Store | | | RAM To DB | | | 0,1 | Addr | | | | | | | |
| 1 | Reg Store | | | Reg DB | | | x | x | x | x | x | x | x | x | x |
| 1 | Reg Store | | | Const To DB | | | x | x | x | x | x | Const | | | |
| 1 | Reg Store | | | ALU To DB | | | 0 | ALU1 | | | ALU2 | | | ALU Op | |
| 1 | Reg Store | | | ALU To DB | | | 1 | ALU1 | | | ALU2 | | | ALU Op | |
| 1 | Reg Store | | | ALU To DB | | | 0 | ALU1 | | | 1 | Const | | | |
| 1 | Reg Store | | | ALU To DB | | | 1 | ALU1 | | | 1 | Const | | | |
| 0 | 0 | x | x | x | x | x | x | x | x | x | x | x | x | x | x |

Abbildung 3: Draft of the instruction decoding

The blue part is describing the typ of the instruction. There are four types a instruction can have:

- No Operation Instruction (`00`)
- Jump Instruction (`01`)
- DataFlow Instruction (`10`)
- Arthm. Instruction (`11`)

This is giving in blue in the bits `[15:14]`. For the arithmetic and dataflow instructions the second bit is ommited. These are disinguished by the databus select bits in light green. Next the gray part. Those 2 bits `[13:12]` are interpreted as condition code for the jump instruction. So the jump is only performed if the condition holds. There are three conditioned instruction and the regular jump instruction that is always performed. Then there are yellow fields. These contain addresses in different length and constants. They are always applied to the busses, independent of the instruction performed. Unless a store happens this is just fine. The purple cells are representing the store line. The current databus can be stored at eight positions:

- unused (`000`)
- unused (`001`)
- Register A (`010`)
- Register B (`011`)
- Register C (`100`)
- Register D (`101`)
- unused (`110`)
- RAM (`111`)

These 3 bits decode the selected store input of the module. The next three bits (in light green) select the module obtaining the databus. The output of this component is applied to the databus. They available components are:

- Register A (`000`)
- Register B (`001`)
- Register C (`010`)
- Register D (`011`)
- RAM (`100`)
- Const (`101`)
- ALU inverted (`110`)
- ALU (`111`)

For the ALU input in bright green and blue, the following 3 bits can be set:

- Register A (`000`)
- Register B (`001`)

- Register C (`010`)
- Register D (`011`)
- Const (`100`)
- Const (`101`)
- Const (`110`)
- Const (`111`)

If the first bit is 1, the following 4 bits `[3:0]` of the instruction are applied to the ALU as constant value.

The dark blueish bit `[8]` is representing if the address in the instruction is applied to the addressbus or the concadenation of register `ra` and `rb`.

The brown bit `[1:0]` are only applied if it is a arthmetic operation and for the second alu input is a register selected.

- Add or Sub (`00`)
- bitwise Or (`01`)
- bitwise And (`10`)
- bitwise Xor (`11`)

Keep in mind that those `AND`, `OR` and `XOR` instructions do also set the `CF` and `ZF` flags.

This whole decoding process is implemented in the following circuit:

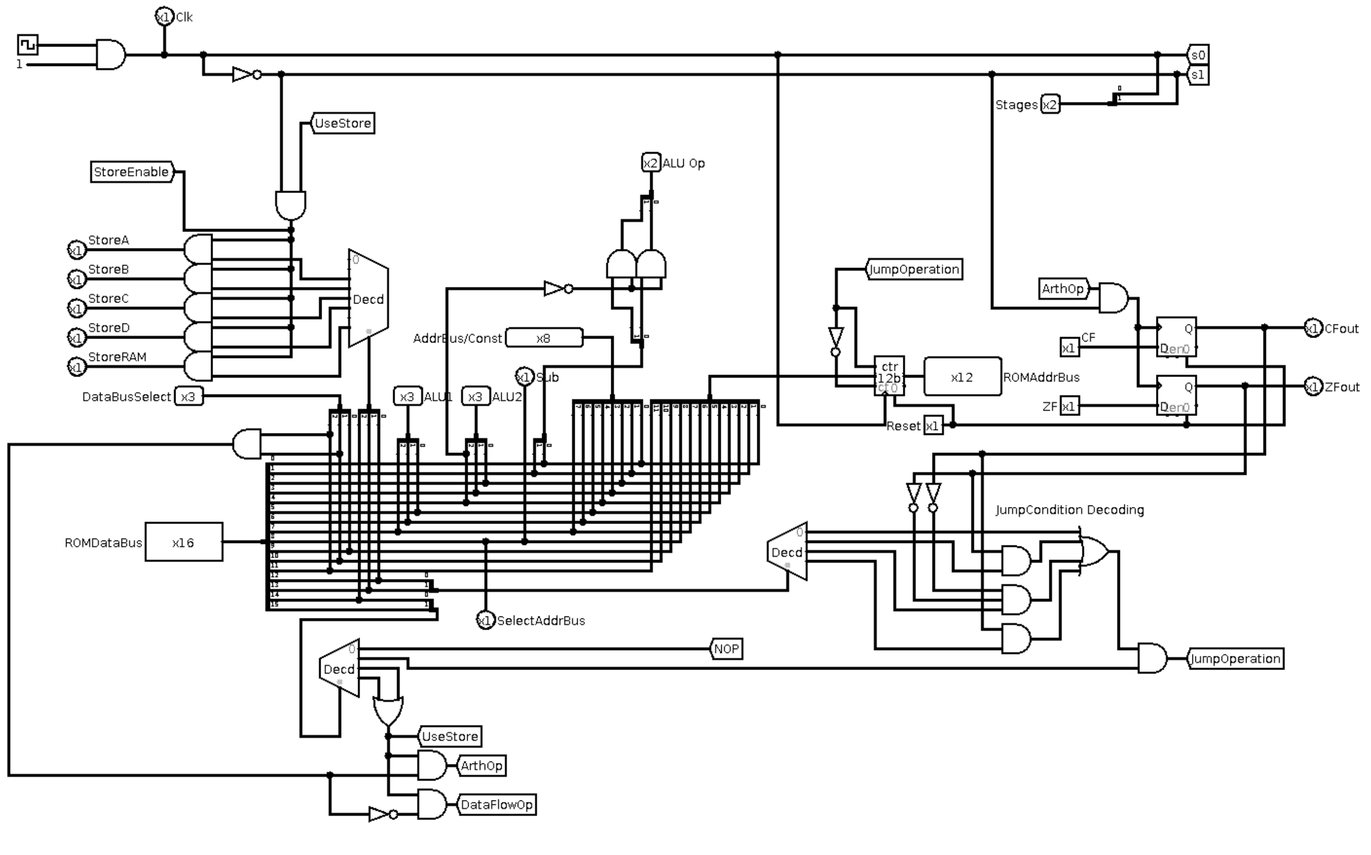

Abbildung 4: Draft of the control unit

[1]

## Literatur

[1] "x86 assembly language - wikipedia."