The EuroTwo v0.002 Manual

MAXIMILIAN NOPPEL

7. März 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
- Three 4-bit generalpurpose-registers
- One 8-bit programmcounter-register
- fixed 12-bit instruction length
- fixed 5[1] cycles per instruction
- 16 x 4-bit RAM
- 255 x 12-bit ROM for instructions
- Only handles unsigned instructions

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 8-bit programmcounter pointing to the ROM which provides 12-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 safes components and execution time.

The architecture consists of 4 parts:

- The ControlUnit (CU)
- The Sequencer (Seq)
- 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

[1] Maybe I can optimize it to only require 3 or 4 cycles.

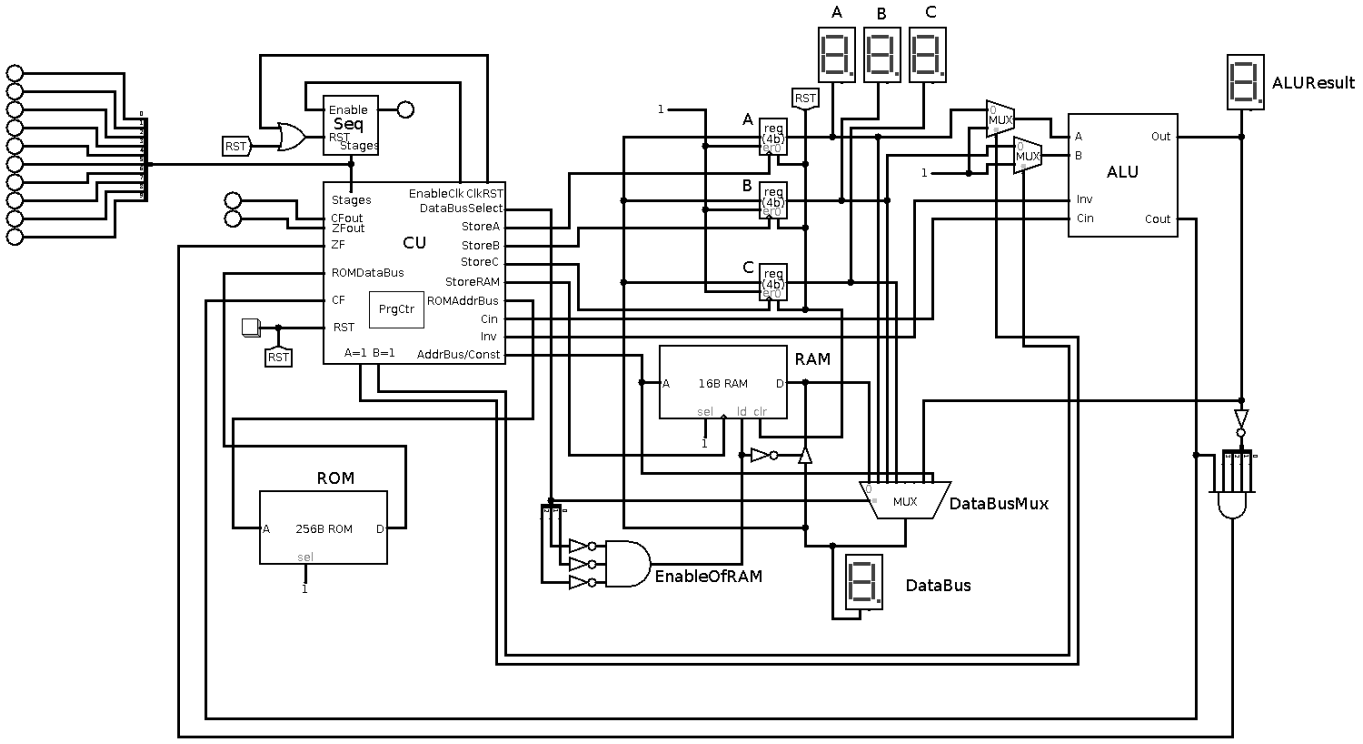

Abbildung 1: Draft of the CPU-architecture

4 ALU

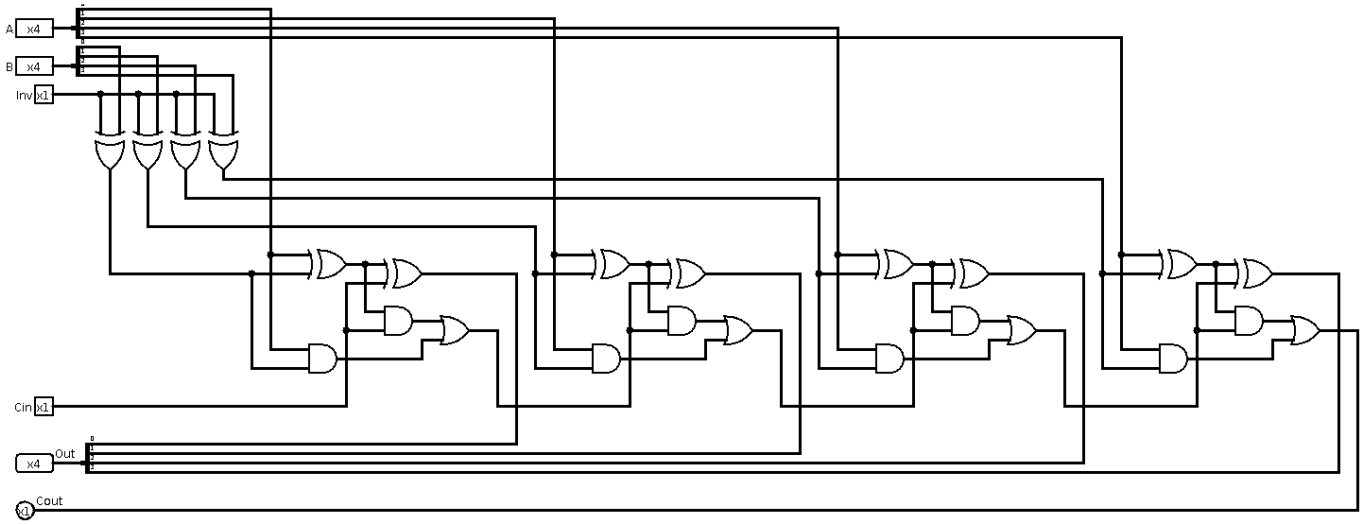

Abbildung 2: Draft of the ALU

4.1 Flags

The ALU is outputting the following signals to the CU:

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

5 Sequencer

The sequencer is responsible to handle the clock and generating stages from it. It integrated a counter to count through the stages. Currently only the stages 0 to 4 are used. Maybe I can get it working with less stages.

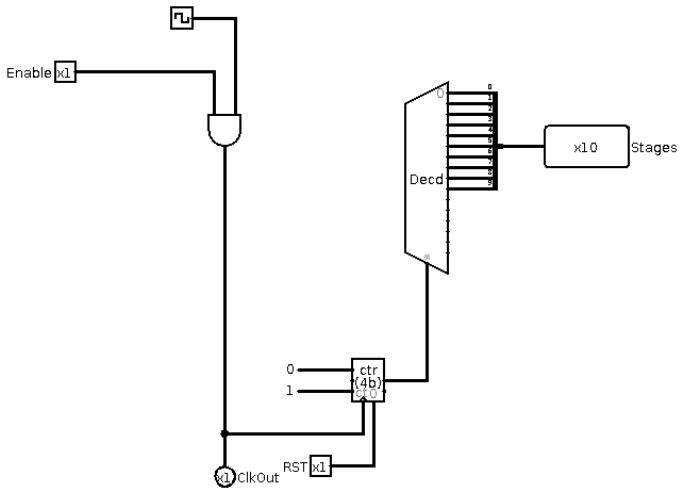

Abbildung 3: Draft of the Seq

Currently the stages are used as follows:

`s0`	Clocking the programmcounter (depending on last instruction either increment or load a address)
`s1`	Time to let the thing go on (for relays)
`s2`	The store signals applied to the decoders are enabled now
`s3`	Time for relays
`s4`	Reset counter and start in stage 0

Tabelle 1: Used stages

6 The Instruction Set

In the following table I present the very simple instruction set for the EuroTwo CPU. Each instruction is given in different flavours depending on which operators it is applied. The Assembler language may consider this.

7 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 12-bit long this is quite simple. The following table documentates how the single bit get interpretet:

Instruction	Formal definition	Binary representation	Description
ADD	$C := A + B(\mod 16)$	0xDE0	Sets CF and ZF
SUB	$C := A - B(\mod 16)$	0xDE3	Sets CF and ZF
ADD1A	$C := A + 1(\mod 16)$	0xDE4	Sets CF and ZF
ADD1B	$C := 1 + B(\mod 16)$	0xDE8	Sets CF and ZF
SUB1A	$C := A - 1(\mod 16)$	0xDE7	Sets CF and ZF
MOV	$B := A$	0x910	Move
MOV	$A := B$	0x850	Move
MOV	$B := C$	0x930	Move
MOV	$A := C$	0x8B0	Move
LD	$A := [addr]$	0x88 <4bit addr>	Load from DataMemory
LD	$B := [addr]$	0x90 <4bit addr>	Load from DataMemory
LD	$C := [addr]$	0x98 <4bit addr>	Load from DataMemory
LC	$A := c$	0x8F <4bit constant>	Load Constant
LC	$B := c$	0x97 <4bit constant>	Load Constant
LC	$C := c$	0x9F <4bit constant>	Load Constant
STR	$[addr] := A$	0x81 <4bit addr>	Store
STR	$[addr] := B$	0x82 <4bit addr>	Store
STR	$[addr] := C$	0x83 <4bit addr>	Store
JMP	$PC := addr$	0x4 <8bit addr>	
JZ	$if\ ZF : PC := addr$	0x5 <8bit addr>	JumpZero, JumpEqual
JA	$if\ (\neg CF \wedge \neg ZF) : PC := addr$	0x6 <8bit addr>	JumpGreater
JB	$if\ CF : PC := addr$	0x7 <8bit addr>	JumpLess
CMP		0xE63	Sets CF and ZF
NOP		0x000	No operation

Tabelle 2: Instruction Set

		11	10	9	8	7	6	5	4	3	2	1	0
4	**JMP**	0	1	0	0	Addr							
5	**JZ**	0	1	0	1	Addr							
6	**JA**	0	1	1	0	Addr							
7	**JB**	0	1	1	1	Addr							
	STR	1	0	x	0	0	Reg DB			Addr			
	LD	1	0	x	Reg Store		0	0	0	Addr			
	MOV	1	0	x	Reg Store		Reg DB			x	x	x	x
	LC	1	0	x	Reg Store		1	1	1	Const			
DE0	**ADD**	1	1	x	1	1	1	1	0	0	0	0	0
DE3	**SUB**	1	1	x	1	1	1	1	0	0	0	1	1
DE4	**ADD1A**	1	1	x	1	1	1	1	0	0	1	0	0
DE8	**ADD1B**	1	1	x	1	1	1	1	0	1	0	0	0
DE7	**SUB1A**	1	1	x	1	1	1	1	0	0	1	1	1
E63	**CMP**	1	1	1	x	x	1	1	0	0	0	1	1
000	**NOP**	0	0	x	x	x	x	x	x	x	x	x	x
		Type			Store		DataBus Select			A=1	B=1	Inv	Cin
										4-Bit Const			
										4-Bit Addr			
				Condition	8-Bit Addr								

Abbildung 4: Draft of the instruction decoding

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

- NoOperation Instruction (00)
- Jump Instruction (01)
- DataFlow Instruction (10)
- Arthm. Instruction (11)

This is giving in blue in the bits [11:10]. Next the gray part. Those 2 bits [9:8] are interpreted as ConditionCode

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, in dependent of the instruction performed. Unless a store happens this is just fine. The green cells are representing two things. First the 2 bit that select the store line. The current databus can be stored at four positions:

- RAM (`00`)
- Register A (`01`)
- Register B (`10`)
- Register C (`11`)

These 2 bits are decoded the selected store input of the module. The 3 bits follow to selected the module obtaining the databus. The output of this component is applied to the databus. They available components are:

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

At last I want to mention the pinkish cells. Each column is represention one signal that is either directly connected to the ALU inputs or is selecting the input for the ALU. For simpler instructions and loops one can overwrite the ALU with ones for the A and B input. This give the two instructions `ADD1A` and `ADD1B`. Anyway the result is written to C. Also this gives a DEC-like instruction `SUB1A` if we overwrite B with 1 and execute a substration.

This whole decoding process is implemented in the following circuit:

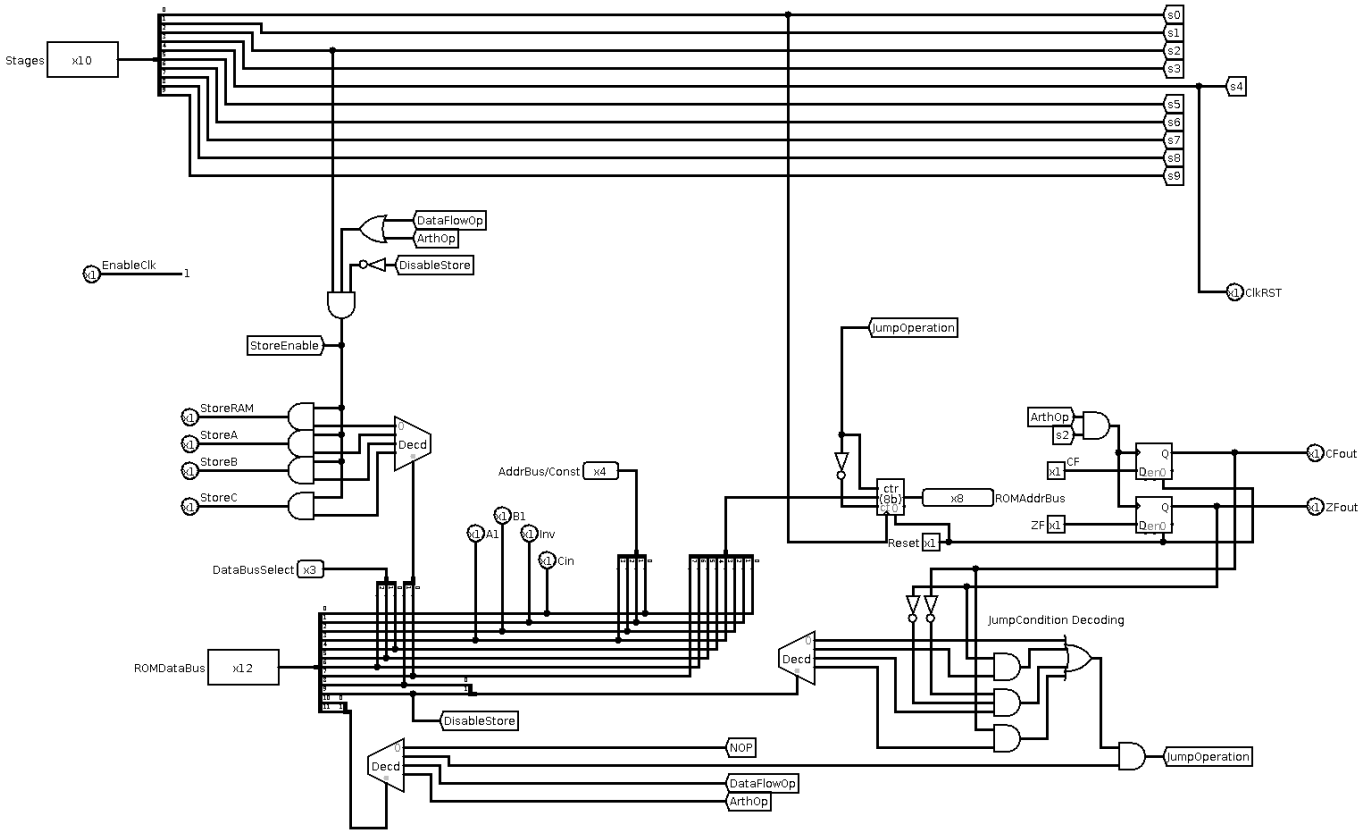

Abbildung 5: Draft of the control unit

8 Required gates and components

The logisim simulation gives me a simple table of my used gates and components. Unfortunately the bigger components like decoders and registers are listed seperatly. Thus there will be much more gates required to implement the registers, decoders, counters and D-FF.

Type	Sequencer	CU	ALU	Remaining	Total
NOT Gate	0	4	0	5	9
AND Gate	1	10	8	2	21
OR Gate	0	2	4	1	7
XOR Gate	0	0	12	0	12
Controlled Buffer	0	0	0	1	1
Decoder	1	3	0	0	4
Register	0	0	0	3	3
D-FF	0	2	0	0	2
Counter	1	1	0	0	2

Tabelle 3: Used gates and components

[1]

Literatur

[1] “x86 assembly language - wikipedia.”