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

07.07.2022

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 07.07.2022

____

Unterschrift

Abstract

This thesis covers the implementation of the Educational Digital Computer (EDiC), a model Central Processing Unit (CPU) which is to be used for teaching the workings of modern digital general purpose processors. For educational purposes the novel CPU Instruction Set Architecture (ISA) is accompanied by an extensive software development environment. The thesis justifies the architectural design decisions which lead to the creation of this 8 bit Complex Instruction Set Computer (CISC) multi-cycle CPU with a 16 bit address space and comprehensive Input / Output (I/O) support. The breakdown of the CPU into seven independent modules simplifies the process of understanding the details of the CPU. Additionally, the choice of Transistor-transistor logic (TTL) Integrated Circuits (ICs) of the 74 family takes the learning focus towards the digital level without complicating the design with analog behavior as Register-transistor logic (RTL) would.

For the functional verification, a behavioral as well as a chip-level Field Programmable Gate Array (FPGA) implementation is performed. The component verification is eased with specially developed test adapters which allow for bit by bit testing of all ICs and in-depth debugging. With a detailed timing analysis it is ensured that the EDiC does not run into unpredictable timing problems.

Kurzfassung

Diese Arbeit beschreibt die Entwicklung und Implementierung des Educational Digital Computer (EDiC), einer Model Central Processing Unit (CPU), welche speziell für die Lehre entwickelt wurde. Sie soll dabei helfen, die Funktionsweise eines modernen, allgemein benutzbaren Prozessors zu vermitteln. Dafür wird die neu entwickelte Instruction Set Architecture (ISA) durch eine ausführliche Entwicklungsumgebung unterstützt. Alle Designentscheidungen, welche zu diesem 8 bit Complex Instruction Set Computer (CISC) mit einem 16 bit Adressraum beigetragen haben, werden ausführlich erklärt und abgewogen. Die Aufteilung in insgesamt sieben größtenteils unabhängige Module vereinfacht das Verständnis der Details der CPU deutlich. Das Verständnis der CPU wird zusätzlich durch Wahl der Transistor-transistor logic (TTL) Integrated Circuits (ICs) aus der 74er Familie deutlich vereinfacht, weil hier der Fokus auf der digitalen Ebene liegt und der Betrachter sich nicht, wie zum Beispiel bei Register-transistor logic (RTL), mut analogen Nebeneffekte beschäftigen muss.

Um die Funktionalität zu verifizieren, wurden zwei Field Programmable Gate Array (FPGA)-Implementierungen durchgeführt. Die erste Implementierung, modeliert nur das Verhalten der Schaltung, während die zweite Implementierung auf IC-level den Schaltplan verifiziert. Die Verifikation der einzelnen Hardware-Komponenten wird durch speziell für den EDiC entwickelte Test Adapter deutlich vereinfacht. Diese ermöglichen es, all ICs Bit für Bit zu testen und die Schaltung ausführlich zu debuggen. Weiterhin wird durch eine detaillierte Timing-Analyse sichergestellt, dass beim EDiC keine unvorhergesehenen Timing-Probleme auftreten werden.

Acknowledgments

First and foremost, I want to thank Henry Westphal, my supervisor. His experience in designing digital circuits and his prior works in creating model CPUs were very valuable in creating the EDiC. Henry Westphal especially helped by copying the schematic to OrCAD, helping with the layout and in general very beneficial advices for the thesis.

Furthermore, many friends and fellow students – namely Anton Schulte, Jasper v. Blankenburg and Leonard Hackel – deserve my special thanks for their invaluable support, both academic and personal.

Most importantly, I am very grateful for all the support I received from my parents, my brother and especially my girlfriend, Nele Heindorf.

Contents

1 Introduction

This thesis covers the development and engineering process of the Educational Digital Computer (EDiC) which is pictured in figure 1.1. It is a completely novel Central Processing Unit (CPU) architecture built in order to visualize and demonstrate the fundamental workings of any CPU. The EDiC can execute over half a million instructions per second and also features step-by-step debugging as well as breakpoint capabilities to enable a better understanding of how CPUs work. All components can be tested individually with the help of dedicated test adapters and, thus, Integrated Circuit (IC) failures can be tracked down and fixed easily. Additionally, to the hardware build, the project includes an open source development environment including an assembler, tools to modify the microcode and also Field Programmable Gate Array (FPGA) simulation and emulation of the hardware [13].

1.1 Background

1.1.1 Short History on Computing

The history of computing hardware goes back to ancient times when people used devices like the abacus which simplifies calculations like additions or multiplications. Starting from the end of the 19th century, analog computers were developed which used continuous physical phenomena to explore complex problems. One of the first widespread analog computers was created by Sir William Thomson (Lord Kelvin) which predicted tide levels for particular locations by using a set of pulleys and wires. [7] Even though analog computers could perform very complex operations like solving differential equations [19] they also had the major drawback that, due to their analog and continues nature, it was not possible to exactly recreate a calculation.

The idea of modern, digital computers was firstly theorized by Alan Turing in his paper *On Computable Numbers* in 1936. [30] He introduced the notion of a universal (Turing) machine which describes a machine that is provable capable of computing

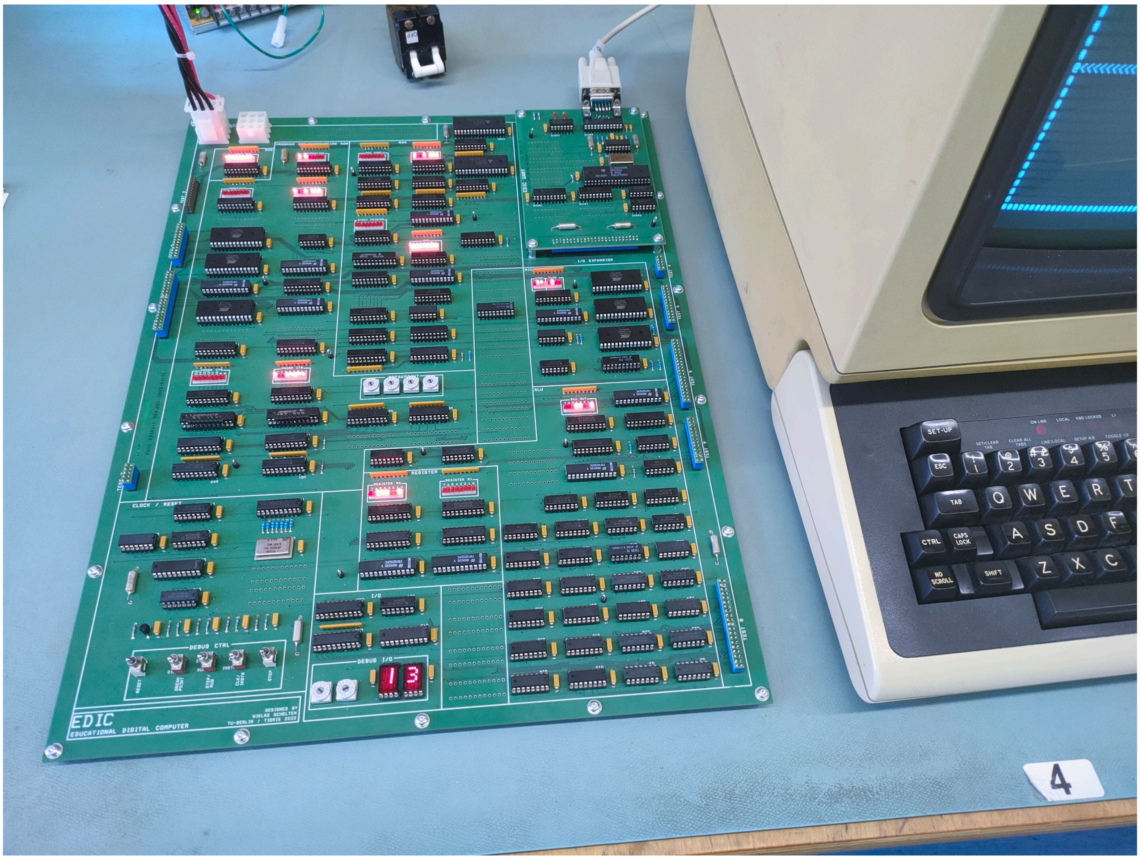


Figure 1.1: The final version of the EDiC playing Snake on a VT-100 over an RS-232 I/O card.

everything that is computable. All the computers today are as capable as a turing machine which is expressed by calling them *turing complete*. This is with exception from their finite memory and limited number range. The first digital computers from the mid 20th century were mechanical or electromechanical machines which combined basic switches like relays and mostly mechanical memory. As fully electrical computers increased the switching frequencies, a lot of different number formats where emerging: Opposed to analog computers where one signal, e.g. a voltage, *represents* a value, it now needs to *encode* a value. In the nowadays common binary system one signal encodes either a 0 or a 1 (for example a low and high voltage) but a lot of different number systems where used like bi-quinary[1].

A variety of technologies where developed for fully electric computers like vacuum tubes or transistors. After using discrete transistors, the advent of ICs in the late 50th lead to a rapid acceleration of computer complexity and speed while reducing the power consumption drastically. The first semiconductor IC was invented in 1959 [32, p. 221] with its first application being in military and space industries in the

[1]Bi-quinary has one quinary signal encoding 0-4 or 5-9 depending on one binary signal encoding a low or high number. This allows two signals to encode a decimal digit similarly to some abacuses.

early 60th. The series of ICs which is the most relevant to this thesis is the Transistor-transistor logic (TTL) 74 series. It is a successor of one of the first TTL ICs developed by Texas Instruments in 1964 for military applications: The 5400 series. [33] The 5400 series of ICs was specified for a temperature range from $-55\,°C$ to $125\,°C$ and came in a ceramic surface-mounted device (SMD) to reach the high temperature range and be low weight. Each package included a set of basic logic circuits like four 2-input NAND gates in the 5400N. In 1966 the first ICs of the 74 series were released which had the same functions but with a reduced temperature rating of $0\,°C$ to $70\,°C$ and often came in plastic packaging for commercial applications. For cost reduction and easier usage, they often came in dual in-line package (DIP) and later on also in plastic packages. In contrast to previous Register-transistor logic (RTL), these TTL ICs were capable of higher switching frequencies and lower power consumption due to a second transistor driving the high voltage level. See section 1.1.3 for a more in depth description of the workings of a TTL gate. As the 74 family of ICs became larger with more complex ICs, more advanced technologies, such as Complementary metal-oxide-semiconductor (CMOS), were also introduced into the family to further reduce the power consumption or increase the switching speeds.

With further advances in the complexity and integration of computing nodes, the first microprocessors were developed in the 70s with the famous Intel 4004 and 8080 in 1971 and 1974, respectively. These processors combine all the logic required for a general purpose CPU into one IC usually exposing interfaces for connecting memories and user Input / Output (I/O) logic.

1.1.2 Technology Selection for the EDiC

The design goal for the EDiC was to create a CPU which aids the teaching of how CPUs generally work. To build a custom CPU, many of the above-mentioned technologies were used for computer design, however, not all of them are equally suited for a model CPU. It was decided to use TTL ICs of the 74 family in the EDiC for several reasons:

- *Complexity:* The ICs of this family are complex enough to make it possible to build complex systems as a general-purpose CPU with only about 100 chips. On the other hand, each individual IC is easy to understand since it is kept quite simple, for example the 7400 has a basic interface of 12 pins for the four 2-input NAND gates plus GND and +5V pins.
- *Speed:* In contrast to previous technologies such as electromechanical relays or RTL, the 74 series is a lot faster, particularly the 74F subseries which is

mainly used in the EDiC. It is feasible to create complex designs with the 74F series with a clock frequency of several MHz. However, at the same time, the clock frequency is not too high, so that special care must be taken when designing the Printed Circuit Board (PCB) which would be the case with higher frequency signals.

- *Simplicity:* Working with the ICs is fairly easy: No special tools – except a soldering iron and oscilloscope – are required to assemble and test the system. Especially the usage of sockets for the DIPs simplifies the build because no IC can overheat while soldering and all the ICs can be replaced later on or while testing.

In contrast to previous and also later technology, the TTL also stands out as the best suited one. When trying to build a CPU out of discrete transistors, not only the logical level needs to be respected but a lot of static and dynamic behavior of the transistors needs to be analyzed. This complicates the design and prevents students from comprehending the CPU on its logical level. On the other side, more modern technologies became so abstract and complex to use that the comprehension of the internal workings of the CPU could also be lost. For example, when choosing FPGAs as the driving technology, the work surrounding the technology quickly becomes more complex than the CPU itself. The FPGA ICs require special voltage levels and special care with the high frequency clock traces, are hard to solder with small pins, require complex build toolchains, cannot be debugged with an oscilloscope and so on.

Thus, the TTL was the ideal technology level for creating a model CPU which helps students understanding the workings of CPUs.

1.1.3 Workings of TTL

Figure 1.2 shows the internals of one of the four NAND gates inside the 7400 IC. The multi-emitter transistor V_1 functions as the logical NAND gate while the transistors V_2, V_3 and V_4 in combination with the diode V_5 form the "totem-pole" output stage. If both inputs are high, a small collector current is drawn by both inputs because V_1 is in reverse-active mode. The current through R_1 "activates" V_2 which in turn "activates" V_4 due to the current steering effect, where the current flows through the one parallel voltage-stable element with the lowest threshold voltage. In this case, the current flows through R_2, V_2 collector-emitter and V_4 base-emitter rather than through V_3 collector-emitter, V_5 and V_4 collector-emitter. Therefore, V_4 drives the output with a low voltage.

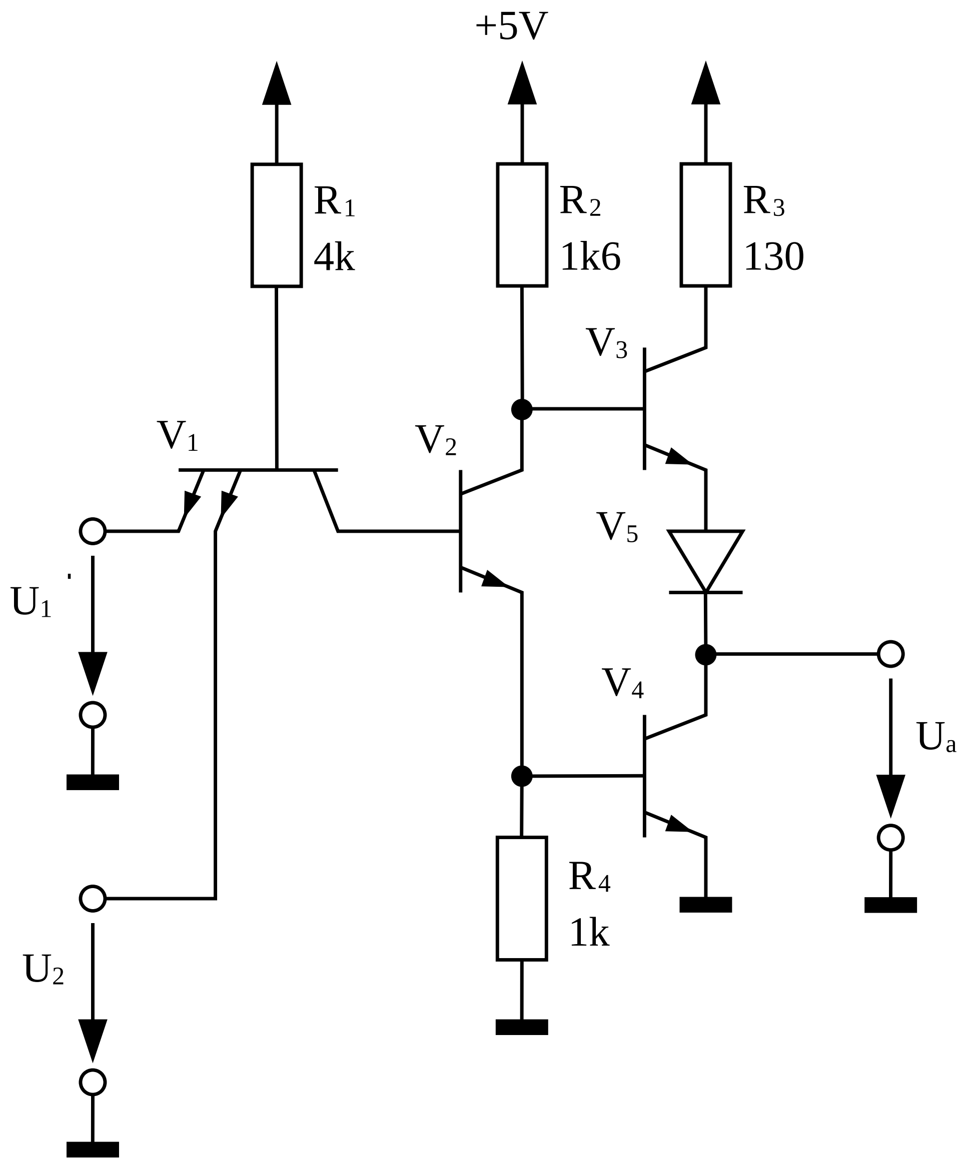


Figure 1.2: TTL NAND with "totem-pole" output stage as in the 7400 IC. [23]

If one of the inputs is low, on the other hand, the current steering effect turns off V_2 since the current flows through R_1 and V_1 base-emitter rather than R_1, V_1 base-collector V_2 base-emitter and R_4. Hence, the above-mentioned current steering effect on the output stage no longer takes effect and V_3 drives the output high through the diode V_5.

The advantage of this "totem-pole" output stage in comparison to a more simple output stage with the collector of V_2 effectively being the output is, that a very low output resistance can be achieved (only the small R_3) which allows the output to drive more inputs of other logic gates. Additionally, the speed is drastically increased because the high output is actively driven instead of pulled up via a resistor as in RTL. However, the voltage drop over V_3 and V_5 also have the effect that the high output voltage is only about 3.5 V in contrast to the simpler approach where almost 5 V can be achieved.

1.2 Thesis Structure

Firstly, the following chapter 2 will give an in-depth explanation of the CPU architecture. It includes an analysis of the design goals and explanations of how they can be achieved. The individual modules are described as well as how they work together to execute any instruction.

Foccusing on usability, the chapter 3 gives an overview of the software environment which eases the development of programs for the EDiC. Furthermore, it also features a tool with which the microcode for instructions can be changed, or new instructions can be added which is especially important looking at the educational purpose of the model CPU.

Subsequently, chapter 4 gives a short background to FPGAs and then covers all the important aspects of the FPGA model which was created to verify the architecture. With a chip-level FPGA implementation it was possible to not only verify the architecture but also the schematic of the EDiC on the logical level.

In chapter 5 the hardware design is finally detailed. Besides an explanation of the schematics (attached in appendix A), the chapter also contains information on the development process of the PCB design.

After the PCB was designed and produced, the process of testing the components and verifying all instructions is shown in chapter 6.

A final conclusion and possibilities for further work are then given in chapter 7.

2 Architecture

Designing and building a general purpose CPU includes a lot of architectural decisions which will influence how well the CPU performs, how complex it is and so on. The goal for the EDiC was to build a CPU that is capable of interacting with extensible I/O devices such as the VT-100 and, at the same time, simple enough to easily understand its workings, so that it is suited to be used in education.

2.1 Design Decisions

First of all, there are several decisions about the general structure of a CPU that need to be made. These decisions greatly influence how the EDiC can be structured into modules and how the final hardware build is set up. Another important factor towards architectural structure is the fact that the final hardware build of the CPU is based on the 74-series of TTL ICs.

2.1.1 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. This, however, is not feasible when using 74-series ICs and at the same time targeting a hardware build that is easy to understand. Some early CPUs built with similar ICs worked with only 4 bits. This can work very well for specific applications but for the most arithmetic computations and data handling 8 bits are more practical. The EDiC will, therefore, use an 8 bit bus for data with an integer range of -128 to 127 or 0 to 255 for unsigned integers.

One of the major limitations of an overall 8 bit bus is the addressable memory space. With only 8 bit for the memory address, the maximum amount of memory addressable is 256 bytes. In a first prototype of the CPU the memory space was tripled by providing 256 bytes of instruction memory besides 256 bytes of Read-Only Memory (ROM) for instruction immediate values and 256 bytes of addressable Static

Random-Access Memory (SRAM). However, especially with a Complex Instruction Set Computer (CISC) architecture (see section 2.1.2), the limited SRAM memory space greatly limits the overall complexity of programs that can be executed. Additionally, more complex programs or even small operating systems are impossible to fit into 256 instructions.

Therefore, it was decided to extend the Program Counter (PC) and the memory addresses to 16 bit, which yields 65536 bytes of addressable SRAM and theoretically 65536 instructions[1]. However, this raises problems of where the 16 bit addresses should come from when all the registers and the memory only store 8 bit. The solution for the EDiC is presented in section 2.2.5.3 when explaining the different modules of the EDiC.

2.1.2 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* [26]. 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 CISCs, where one instruction can be very capable [6]. For example an 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 three independent instructions to load operands from the memory into registers, do the addition and write the result back to memory.

In Pipelining there are fixed steps that 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, in theory, possible to dispatch a new instruction each cycle even though the previous instruction is not yet finished. A typical 5-step pipeline would consist of the following steps [26]:

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.

[1] The largest feasible Electrically Erasable Programmable Read-Only Memory (EEPROM) available used for instruction memory has only 15 address bits and with that only 32768 8 bit words of data.

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 certain challenges. Those include a greater resource usage since all intermediate results need to be stored in pipeline registers. Additionally, branch instructions[2] pose a greater challenge because the moment the CPU executes the branch, the following 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 is also noteworthy that branch prediction and pipeline flushes can be quite vulnerable as recently shown in CVE-2017-5753 with the Spectre bug [9].

Therefore, the EDiC is built as a Multicycle CISC.

2.1.3 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) or 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’ (low voltage) or ‘1’ (high voltage) or not drive the bus (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, adds a register value to it 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 ALU input

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

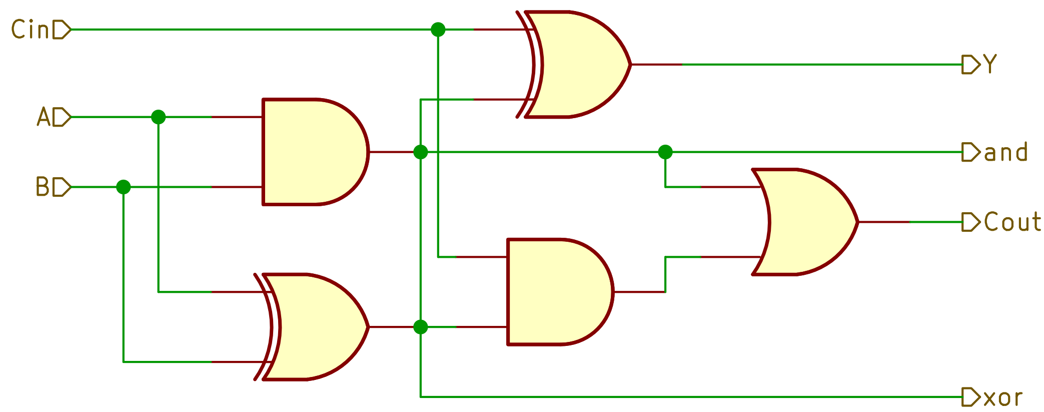


Figure 2.1: 1 bit full adder with the usual A, B and Carry inputs and Y and Carry outputs as well as the XOR and AND outputs.

6. ALU operation
7. Data from ALU output over *bus* to register

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

2.2 Modules

The design has been split into seven rather independent modules of varying complexity which mainly interface with each other over the bus and control signals.

2.2.1 Arithmetic Logic Unit (ALU)

An ALU is the computational core of any CPU as it performs all the calculations. The ALU of the EDiC is by design simple with only four different operations plus an option to invert the second input. The result of the ALU is stored in a result register which can drive the bus to store the result in a register or memory. For simplicity, the first input of the ALU (A input) is directly connected to the register file (section 2.2.2) and only the second input (B input) is accessible from the bus. This limits the possibilities of instructions, however, if both inputs should have been driven by the bus, every ALU instruction would have taken three instead of two cycles limited by the bus (first cycle A input, second cycle B input, third cycle result).

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

The ALU consists of an 8 bit ripple carry full adder and a barrel shifter. The operations are controlled by three control signals: The first two bits select which ALU operation to perform, and the third bit modifies the operation to perform. The possible operations are shown in table 2.1. For the adder, the third bit inverts the B input when active (All input bits are XORed with the control bit) and is used as the carry in of the adder. This negates the B input in two's complement and, therefore, subtracts it from the A input. For the barrel shifter, the third bit reverses the shift direction.

The XOR and AND operations shown in table 2.1 are chosen because they are already implemented in the half-adders and no additional logic is required to implement them. A complete 1 bit full-adder of the EDiC 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 three consecutive multiplexers shifting by 1, 2 or 4 bits to the right that are controlled by the first 3 bits 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, two chips are needed for a full 8 bit 2 to 1 multiplexer.

The ALU also provides four flags which are used for condition execution. The Zero (all result bits are zero) and Negative (The most significant bit (MSB) of the result) flag are both very easy to derive and were the only ones included in the prototype. However, the experience of programming for the CPU showed that it is desirable to

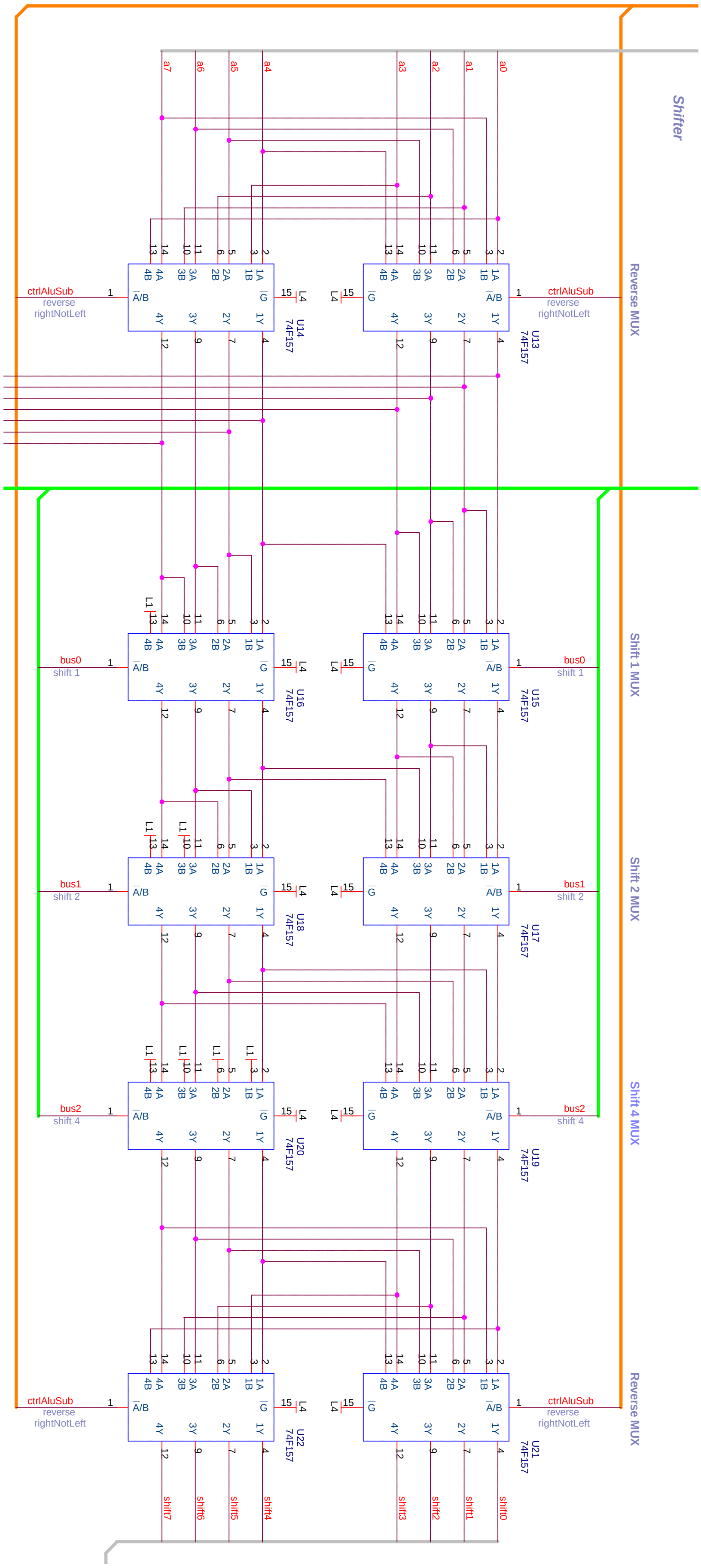


Figure 2.2: 8 bit bidirectional barrel shifter.

be able to work with more advanced ALU flags when programming more complex functions. Having only Zero and Negative Flags, for example, does not allow unsigned operations of the full width[3] which is especially important with only 8 data bits. It limits unsigned operations to only 0-127 even though the ALU would be capable of calculations with 0-255.

A lot of modern CPUs feature many flags with the Intel 64® and IA-32 CPU having about 20 different flags [17, Section 3.4.3]. However, the popular ARM Architecture has a rather unique but very capable system for conditional execution which relies on only the four most used ALU flags. The EDiC uses the same flags and their functions are as follows:

- **N** The *Negative* flag indicates that the result is negative and is set if the 8th bit of the ALU result (MSB) is `'1'`.
- **Z** The *Zero* flag indicates that the result is `'0'` and is set if all 8 result bits are `'0'`.
- **V** The *Overflow* flag indicates that an overflow occurred and is set if the carry in and carry out of the 8th full-adder are different. This detects arithmetic overflows for signed two's-complement calculations.
- **C** The *Carry* flag is the carry-out bit of the adder for adding and subtracting. For logical operations (`XOR` and `AND`) the carry flag has no meaning and for shifting operations it equals the last bit that was "carried out" (or is unchanged if shifting by 0 bits).

2.2.2 Register File

As it 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. All ports can access both registers.

2.2.3 Program Counter (PC) & Instruction Register

The PC is a special 16 bit register which is used to store the address for the current instruction. Usually it is incremented by one for each instruction. However, it is

[3] An overflow detection is not possible and with that a greater or less than comparison cannot be done.

also possible to load the PC from an instruction immediate (see below) or from the memory (section 2.2.5). The first option is used for branch instruction while the second option is used for returning from a function, which is explained in more detail in section 2.2.5.2. The value of the PC is used as an address for the instruction EEPROMs and can also be driven to the memory for storing the return address for function calls.

Each instruction of the EDiC is stored in a 24 bit register of which 8 bits are the instruction and 16 bits represent an optional instruction immediate. It can be used as an address for the memory/PC (16 bit) or as data (8 bit) driven to the bus. The instruction is directly forwarded to the control logic (section 2.2.4).

2.2.4 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. What kind of control signals exist in the EDiC is explained after all the modules are described in section 2.3. For keeping track which cycle of each instruction is currently executing, a 3 bit synchronous counter is used. Each control signal could be derived by a logical circuitry with 13 inputs: 8 bits instruction, 4 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. Therefore, an 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. How the EEPROM is programmed with the correct data is explained in depth in section 3.1.

One special case are the 3 bit ALU opcodes. They are not decoded the usual way from the instruction but are directly taken from the three least significant bits (LSBs) of the instruction. This is done to reduce the storage requirements for the decoding EEPROMs. For instructions that use the ALU, the 3 LSBs need to be set accordingly but for all other instructions, the three bits can be used as usual for decoding the instruction because it does not matter what the combinatorial part of the ALU does.

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, as 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 are filled with the control signals for an instruction fetch and decode.

2.2.5 Memory

The memory module became the most complex module because it includes not only the main memory of the CPU in form of an asynchronous SRAM but also includes a lot of addressing logic for the 16 bit addresses.

The addressing logic is required because the EDiC has 16 bit addresses with only an 8 bits data bus. However, the EDiC also features memory mapped I/O and a stack implementation which further complicate the addressing logic. Both these features and the addressing logic is described below.

2.2.5.1 Memory Mapped I/O

I/O is one of the most important factors of any CPU besides the computing capabilities which are mostly defined by the ALU. The first prototype showed that 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.

2.2.5.2 Stack Implementation

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

However, a typical stack implementation as in modern CPU architectures like ARM is rather complex. It requires a Stack Pointer (SP) register which is usually accessible like any other general purpose register and can be directly used as an address. This includes using it as an operand for arithmetic operations which is not possible when the bus width is only 8 bits but the SP needs to be 16 bits wide to be used as an address. Therefore, the EDiC uses a unique approach to the stack:

Similarly to the memory mapped I/O it was decided to implement the SP as an 8 bit register which can be incremented and decremented at function calls and returns, respectively. 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 (that 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 (next PC value) at address `0xffff`, which is `0x1_{sp}ff` 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, and the build is simplified by not using more different components. The call and return instructions are further described in section 2.4.

Usually, the stack is also used to store parameters for a function call. In the EDiC, this can be achieved by providing a special *store* and *load* instruction which access the stack memory with an increment SP. This way it is possible to store parameters before calling a function, and it is also possible to retrieve modified values after the call[4]. The calling convention for the EDiC is further described in section 3.2.1.

2.2.5.3 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. Figure 2.3 shows a simplified block diagram of how the addressing logic works. For easier understanding

[4]This is important when a function takes memory pointers as parameters and modifies the memory content. For example a string parsing function could take a pointer to the start of the string, parse some characters as a number, return its number representation and modify the parameter such that it points to where the parsing stopped.

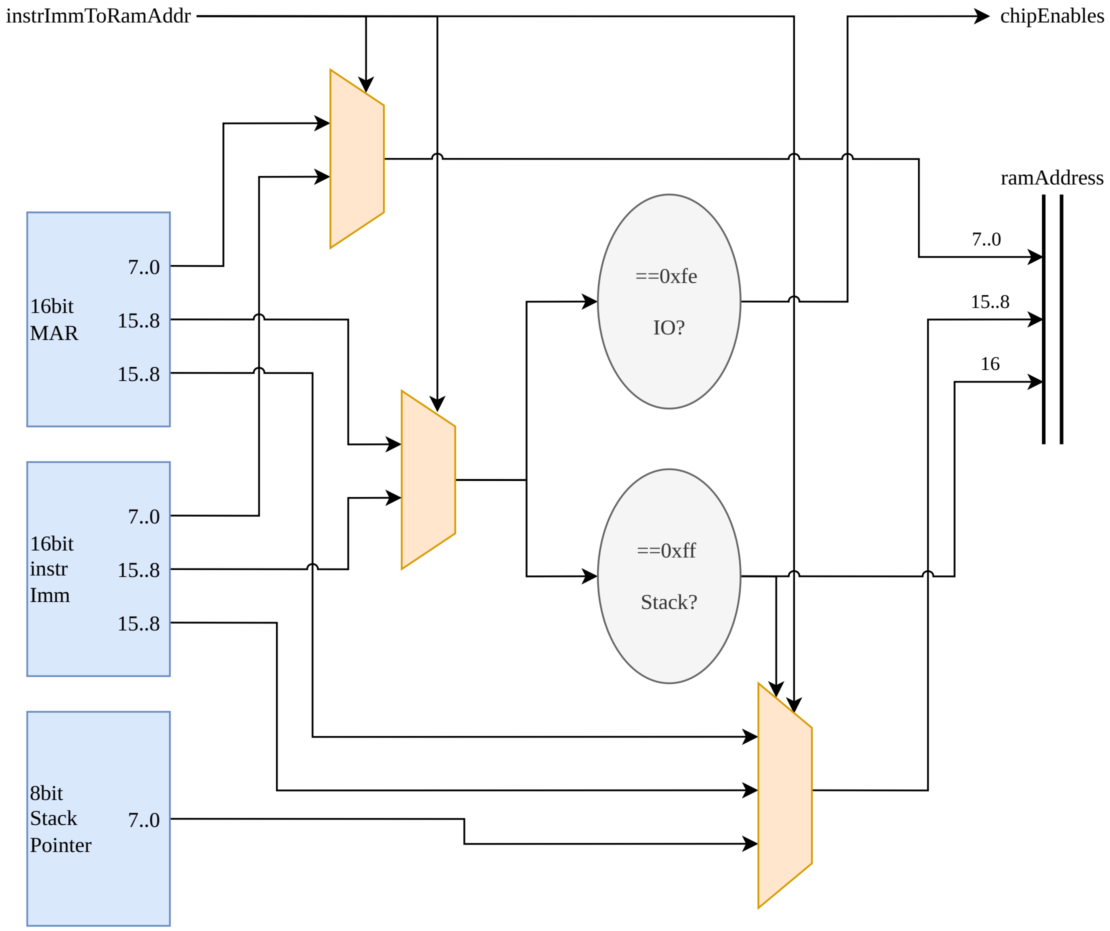


Figure 2.3: Simplified block diagram of the addressing logic for the EDiC.

multiplexers are used, even though the hardware build of the EDiC uses tri-state busses as explained in section 5.1.4.

There are two main sources for memory addresses: The new 16 bit Memory Address Register (MAR) which can be written to from the bus and secondly 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 which are known at compile time, the 16 bit instruction immediate can be directly used as an address, preserving the MAR. 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.

2.2.6 Input & Output

The EDiC can interface with different I/O devices connected to it via the memory mapped I/O. For evaluation and debugging, the EDiC includes one I/O device at address `0x00` which can be read from and written to. The value to be read can be selected by the user with two hexadecimal 8 bit switch and the values written to the address `0x00` are displayed with a two digit display. This allows simple programs to run independently of external I/O devices.

2.2.7 Clock, Reset & Debugging

An important feature when developing a CPU is debugging capabilities. The initial prototype could at least step the clock cycle by cycle. However, as programs get more complex 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 from incrementing. 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 deasserted for only one clock cycle, which in turn increments the step counter only once. To step whole instructions, the halt signal is deasserted until the instruction is finished (marked by a control signal that is asserted 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 this two equal. As soon as the CPU halts, the user can then switch to stepping mode and debug the specific instruction of the program. The user can freely switch between these modes with switches and buttons provided on the lower side of the PCB in figure 1.1.

2.3 Control Signals

The EDiC has 24 control signals which define what the current cycle does:

- `aluYNWE` - ALU output register write-enable (active low): Connects to the clock-enable input of the ALU output register.
- `aluNOE` - ALU output-enable (active low): Enables the tri-state buffer to drive the bus with the value of the ALU output register.
- `reg0NWE` - Register 0 write-enable (active low): Connects to the clock-enable input of the register 0.
- `reg1NWE` - Register 1 write-enable (active low): Connects to the clock-enable input of the register 1.
- `regAluSel` - Register Select for the ALU A input: When 0, sets register 0 as A input to the ALU, otherwise, register 1.
- `reg0BusNOE` - Register 0 bus output-enable (active low): Drives the bus with the value of register 0.
- `reg1BusNOE` - Register 1 bus output-enable (active low): Drives the bus with the value of register 1.
- `memPCFromImm` - load data for PC from instruction immediate: Selects the load input of the PC to be from the instruction immediate instead of from the memory.
- `memPCNEn` - PC enable (active low): enables the PC to load data or increment by one depending on the next control signal.
- `memPCLoadN` - PC load and not increment (active low): When 0 load the PC with the data specified by `memPCFromImm`, otherwise, increment the PC by one.
- `memPCToRamN` - PC output-enable (active low): Drives the bus and `ram2data` with the value of the PC.
- `memSPNEn` - SP-enable (active low): enable the SP to be incremented or decremented depending on the next control signal.
- `memSPUp` - SP increment not decrement: When 1, increment the SP, otherwise, decrement.
- `memInstrNWE` - Instruction write-enable (active low): Connects to the clock-enable input of the instruction register.

- `memInstrNOE` - Instruction output-enable (active low): Drives the bus with the lower 8 bits of the instruction immediate.
- `memMar0NWE` - MAR bits 7..0 write-enable (active low): Connects to the clock-enable input of the lower 8 bits of the MAR.
- `memMar1NWE` - MAR bits 15..8 write-enable (active low): Connects to the clock-enable input of the upper 8 bits of the MAR.
- `memInstrImmToRamAddr` - Random-Access Memory (RAM) address from instruction immediate and not MAR: When 1, use the instruction immediate as address for the memory, otherwise, use the MAR content.
- `memRamNWE` - Memory write-enable (active low): Connects to the write-enable input of the SRAM and I/O.
- `memRamNOE` - Memory output-enable (active low): Drives the bus with the value of the SRAM or I/O depending on the memory address (section 2.2.5).
- `instrFinishedN` - Instruction finished (active low): Is asserted at the last active cycle of the instruction to reset the step counter to `0x00`[5].
- `busFFNOE`[6] - Drive `0xff` to the bus (active low): Connects to the output-enable input of the constant `0xff` driver.

2.4 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 parameters of each instruction and how the instructions are programmed is shown in section 3.2.2.

2.4.1 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.

[5]The instruction finished signal is also used for the debugger to detect the end of an instruction and halt when stepping through instructions and not single cycles.

[6]Was added in the component verification and is explained in section 6.2.3.

- *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[7].
  2. Address decoding.
  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 decoding.
  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:

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

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

- *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.

2.4.2 Memory operations

Some ALU operations also include reading values from memory. However, 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 sub-functions.

  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 sub-functions.

  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.

2.4.3 Miscellaneous operations

There are some more operations that are neither ALU nor memory operations like move and branch instructions.

- *Move between registers:* 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.

3 Software Development Environment

When just providing the hardware, the CPU can hardly be used. It is possible to write programs by hand by writing single bytes to the EEPROMs that hold the program. However, it is quite infeasible to write complex programs this way. While manually writing programs byte by byte is doable, writing the content of the EEPROMs which hold the microcode by hand is quite infeasible.

Therefore, the EDiC comes with two main software utilities that form the software development environment in sections 3.1 and 3.2.

3.1 Microcode Generation

The goal is to define all the available instructions and what they perform in which instruction step and then have a program automatically generate the bit-files for the EEPROM. This approach allows simple modifications to the existing microcode if a bug was found or a new instruction should be added. The file format which defines the microcode has to be human and machine-readable as it should be easily edited by hand and also be read by the tool that generates the bit-files. A very common file format for tasks like this is JavaScript Object Notation (JSON) [18] which is widely used in the computer industry. Besides basic types as strings and numbers, it allows arrays with square brackets (`[]`) and objects with curly braces (`{}`). Each object contains key value pairs and everything can be nested as desired. For the EDiC microcode generation CoffeeScript-Object-Notation (CSON) was used which is very similar to JSON but is slightly easier to write by hand because its syntax is changed a bit:

- It allows comments which is extensively used to ease the understanding of individual instruction steps.
- Braces and commas are not required.

```
interface IMicrocodeFile {
  signals: [
    {
      name: string;
      noOp: 0 | 1;
    }
  ];

  instructionFetch: [
    {
      [signalName: string]: 0 | 1;
    }
  ];

  instructions: [
    {
      op: string;
      cycles: [
        {
          [signalName: string]: 0 | 1 | 'r' | 's' | '!r' | '!s';
        }
      ];
    }
  ];
};
```

Code Example 3.1: Schema of the Microcode Definition CSON-File [5] as a TypeScript [21] Type definition.

- Keys do not require string quotation marks.

The schema for the file describing the microcode is shown in code example 3.1. Some examples for the fields are listed below:

Signals The signals array consists of objects that define the available control signals and the default value of the control signal. Code Example 3.2a defines the *not write enable signal for register 0* control signal and defines the default state as high. This means, when this control signal is not specified in an instruction it will stay high and, therefore, register 0 will not be written.

Instruction Fetch This array defines the steps that are performed at the beginning of each instruction to fetch the new instruction and decode it. Each object

```
{
  name: 'reg0NWE'
  noOp: 1
}
```

(a) Register 0 write enable control signal.

```
instructionFetch: [
  { # write instruction
    memInstrNWE: 0
  }
  { # increment PC
    memPCNEn: 0
    memPCLoadN: 1
  }
]
```

(b) Instruction fetch and decode cycles.

Code Example 3.2: Example definitions of one control signal and the instruction fetch cycles for the microcode generation.

represents one step and consists of key value pairs that define one control signal.

In code example 3.2b the first instruction cycle specifies only the *instruction not write enable* to be low and with this write the instruction into the instruction register. Secondly, the PC is incremented by setting *PC not enable* to low and *PC not load* to high.

Instructions The instructions are an array of all available instructions. Each instruction is defined as an op code, which is the 8 bit instruction in binary format. However, if it was only possible to define the 8 bit as 0s and 1s, instructions which only differ in the register used would need to be specified separately which is very error-prone. Therefore, it is allowed to specify the bit that determines if register 0 or 1 is used to be set to `'r'` or `'s'` and then multiple instructions are generated. The `cycles` array defines the steps each instruction does in the same way as the `instructionFetch` array does. However, as the value of individual control signals may depend on which register is specified in the op code, it is also possible to specify `'r'`, `'!r'`, `'s'` or `'!s'`.

Code Example 3.3a defines the move immediate to register instruction for both register at the same time. The *instruction immediate not output enable* is low and either register 0 or register 1 is written to. This definition would be equal to code example 3.3b.

This example is quite simple, however, instructions with two registers as arguments would result in four times the same definition and duplication can always result in inconsistencies. The same idea is also used for the ALU operations. The ALU operation control signals are not generated by the microcode but are rather the

```
{
  op: '1111100r' # r = imm
  cycles: [
    { # imm to bus to r
      reg0NWE: 'r'
      reg1NWE: '!r'
      memInstrNOE: 0
    }
  ]
}
```

(a) Definition using 'r' in the opcode.

```
[
  {
    op: '11111000' # r0 = imm
    cycles: [
      { # imm to bus to r0
        reg0NWE: 0
        reg1NWE: 1
        memInstrNOE: 0
      }
    ]
  }
  {
    op: '11111001' # r1 = imm
    cycles: [
      { # imm to bus to r1
        reg0NWE: 1
        reg1NWE: 0
        memInstrNOE: 0
      }
    ]
  }
]
```

(b) Equivalent definition of both separate instructions.

Code Example 3.3: Definition of the move immediate to register instruction for the microcode generation.

three least significant bits of the instruction. Therefore, all instructions using the ALU can have the exact same control signals stored in the microcode EEPROM. To avoid 8 definitions of the same instructions, the op code can contain `'alu'` and all 8 instructions are generated. Code example 3.4 for example defines the ALU operation with two registers and defines all 32 instructions with the op codes `'00000000'` to `'00011111'`.

There is one final specialty built into the Microcode Generator: The EDiC has a branch instruction which is either executed or treated as a no-operation depending on the current state of the ALU flags. For all other instructions, the flags are ignored, and the instructions are always executed. For this special instruction, the last four bits replaced with `flag` define at which state of the ALU flags, the branch should be executed. The possible conditions are heavily inspired by the conditional execution of ARM CPUs[10] as the ALU flag architecture is very similar. The possible values for the `flag` field and their meanings are listed in table 3.1. Especially for a

```
{
  op: '000rsalu' # r = r x s (alu)
  cycles: [
    { # r x s into alu
      aluYNWE: 0
      reg0BusNOE: 's'
      reg1BusNOE: '!s'
      regAluSel: 'r'
    }
    { # alu into r
      aluNOE: 0
      reg0NWE: 'r'
      reg1NWE: '!r'
    }
  ]
}
```

Code Example 3.4: Definition of the ALU operation with two register arguments for the microcode generation.

```
{
  op: '1010flag' # pc := imm
  cycles: [
    { # imm to pc
      memPCNEn: 0
      memPCLoadN: 0
      memPCFromImm: 1
    }
  ]
}
```

Code Example 3.5: Definition of the branch instructions.

CPU with only 8 bits it is important to support unsigned and signed operations and with a complex microcode it is no problem to support all the different branch instructions and facilitate the application design. Code example 3.5 defines the branch instructions.

3.2 Assembler

The second software that is similarly important is the assembler. An assembler translates human-readable instructions into machine code, i.e. the bits that are stored in the instruction EEPROMs. For the EDiC each instruction is 24 bits wide, with

Table 3.1: All available branch instructions with their op-code and microcode translation based on the ALU flags explained in section 2.2.1.

`flag` (OP-Code)	Assembler Instruction	ALU flags	Interpretation
`0000`	`jmp/bal/b`	Any	Always
`0001`	`beq`	`Z==1`	Equal
`0010`	`bne`	`Z==0`	Not Equal
`0011`	`bcs/bhs`	`C==1`	Unsigned $\geq$
`0100`	`bcc/blo`	`C==0`	Unsigned $<$
`0101`	`bmi`	`N==1`	Negative
`0110`	`bpl`	`N==0`	Positive or Zero
`0111`	`bvs`	`V==1`	Overflow
`1000`	`bvc`	`V==0`	No overflow
`1001`	`bhi`	`C==1` and `Z==0`	Unsigned $>$
`1010`	`bls`	`C==0` or `Z==1`	Unsigned $\leq$
`1011`	`bge`	`N==V`	Signed $\geq$
`1100`	`blt`	`N!=V`	Signed $<$
`1101`	`bgt`	`Z==0` and `N==V`	Signed $>$
`1110`	`ble`	`Z==0` or `N!=V`	Signed $\leq$
`1111`	`-`	Any	Never (Not used)

8 bits instruction op code and 8 or 16 bits immediate value. Even though assemblers usually only translate instructions one by one, they can have quite advanced features. With an assembler, the programmer is no longer required to know the specific op codes for all instructions and set individual bits of the instructions which is very error-prone. The assembler for the EDiC, therefore, allows easier programming with a simple text-based assembly syntax similar to the well-known ARM syntax.

Code examples 3.6 and 3.7 show the translation that the assembler does. In particular, code example 3.6 shows the assembler program that is written by the programmer and code example 3.7 summarizes what values are stored in the program EEPROM.

The full assembler code used in the demonstration in figure 1.1 is attached in appendix A.

```
PRNG_SEED = 0x0000
SIMPLE_IO = 0xfe00

prng:
  ldr r0, [PRNG_SEED]
  subs r0, 0
  beq prngDoEor
  lsl r0, 1
  beq prngNoEor
  bcc prngNoEor
prngDoEor:
  xor r0, 0x1d
prngNoEor:
  str r0, [PRNG_SEED]
ret

start:
  mov r0, 0
  str r0, [PRNG_SEED]
  prng_loop:
    call prng
    str r0, [SIMPLE_IO]
  b prng_loop
```

Code Example 3.6: Pseudo Random Number Generator (PRNG) written in the EDiC Assembler.

3.2.1 Calling conventions

Even though calling conventions are strictly speaking not a feature of the assembler, it is an important factor to keep in mind in functional programming. Calling conventions are a set of rules which caller (the instructions calling a subroutine) and callee (the subroutine that is called) should usually follow.

Parameters Typically, the first parameters from the caller to the callee are passed in registers, which avoids long memory operations for storing and loading the parameters. As the EDiC's memory operations cannot stall they are not slower than register operation and the EDiC has only 2 registers. Therefore, only the very first argument is passed in `r0` and all further arguments are passed in the memory. The parameters are stored on the stack of the callee starting at stack address `0x00` (`0xff00` as memory address).

```
0x0000 - op: 10100000, imm: 0x000a - b 0x0a
0x0001 - op: 11110000, imm: 0x0000 - ldr r0, [0x00]
0x0002 - op: 10010001, imm: 0x0000 - subs r0, 0x00
0x0003 - op: 10100001, imm: 0x0007 - beq 0x07
0x0004 - op: 10000111, imm: 0x0001 - lsl r0, 0x01
0x0005 - op: 10100001, imm: 0x0008 - beq 0x08
0x0006 - op: 10100100, imm: 0x0008 - bcc 0x08
0x0007 - op: 10000100, imm: 0x001d - xor r0, 0x1d
0x0008 - op: 11110010, imm: 0x0000 - str r0, [0x00]
0x0009 - op: 10110001, imm: ------ - ret
0x000a - op: 11111000, imm: 0x0000 - mov r0, 0
0x000b - op: 11110010, imm: 0x0000 - str r0, [0x00]
0x000c - op: 10110000, imm: 0x0001 - call 0x01
0x000d - op: 11110010, imm: 0xfe00 - str r0, [0xfe00]
0x000e - op: 10100000, imm: 0x000c - b 0x0c
```

Code Example 3.7: The output of the PRNG of code example 3.6. The first 16 bits are the memory address, then 8 bits for the instruction op-code and 16 bits for the instruction immediate follow. For reference, the original instruction with all the variables replaced is appended.

Return value The return value is to be placed in `r0`. If a return value larger than 8 bit (or multiple 8 bit values) are to be returned, the caller may pass a pointer to a memory location as a parameter and the callee works on the memory content pointed to.

Preservation The register `r1` can be used as a function local variable and, therefore, has to be preserved by any callee. This is usually done by storing the content on the stack at the beginning of the function and restoring it from the stack at the end of the function.

3.2.2 Available Instructions

This section summarizes all available instructions and which parameters they take. All instructions start with the operation followed by up to two parameters which are separated by a comma.

There are four different parameter types. It can either be a register specified as `r0` or `r1`. The register value can also be passed as the address to a memory operation with `[r0]`.

Immediate values can also be specified as value or as address with brackets around the immediate value. However, the syntax for immediate values is more complex, as the assembler can parse decimal (positive and negative) as well as hexadecimal numbers. Additionally, variables can be used which are further explained in section 3.2.3.

When specifying a value, the immediate can range between -127 and 255 (two's complement and unsigned) and when used as an address it can range between 0 and 0xfffe (65534). The upper limit is not 0xffff because that address is reserved for the return address and should not be overwritten.

3.2.2.1 ALU Instructions

The following ALU instructions are available:

- add
- sub
- and
- eor
- xor
- xnor
- lsr
- lsl

ALU instructions always take two parameters. The first parameter is the left hand side operand and the register where the result is stored in, and the second parameter is the right hand side operand. The following operand combinations are possible:

- Two registers

  `sub r0, r1` does: $r_0 := r_0 - r_1$

- One register and one register as memory address

  `lsr r1, [r0]` does: $r_1 := r_1 \gg \text{mem}[r_0]$

- One register and an immediate value

  `xor r0, 0x0f` does: $r_0 := r_0 \veebar 15$

- One register and an immediate value as memory address

  `add r1, [0x0542]` does: $r_1 := r_1 + \text{mem}[1346]$

All the ALU instructions can have an 's' as suffix which has the effect that the result of the operation is not written to the first operand. This is useful when a calculation is only performed to update the ALU flags, but the register value should be preserved. This results in a special ALU instruction: `cmp` which is an alias to `subs` which is typically used to compare to values and perform a branch instruction based on the result.

The code

```
cmp r0, 10 // equal to subs r0, 10
blt 0x42
```

compares the `r0` register with the value 10 and if $r0 < 10$ branches to instruction at address 66 and preserves the content of `r0`.

3.2.2.2 Memory Instructions

The following memory instructions are supported:

- str
- ldr
- sts
- lds
- stf
- ldf
- sma

The two common instructions are `str` and `ldr` which are *store* and *load* operations. These two instructions take two parameters: The first is the register used in the store or load operation and the second is the memory address. They either take a 16 bit immediate address which is used as the full address for the access or a register as address. As the registers are only 8 bits, the register value is only used for the lower 8 bits of the address and the upper 8 bits are the value of the MAR. The upper 8 bits of the MAR can be set with the `sma` instruction which takes either a register or an 8 bit immediate value.

The `lds` and `sts` instructions are used for accessing the stack. They only take immediate addresses and the assembler makes sure that the upper 8 bits of the address are `0xff` to always access the stack.

The `ldf` and `stf` instructions work very similar in only accessing the stack. However, before the memory access, the SP is incremented and after the access, it is restored. This way, it is possible to access parameters of a function that is called.

Some examples:

`ldr r0, [0xabba]`	Loads the value from address `0xabba` into `r0`
`str r1, [0xc0de]`	Stores the value in `r1` to address `0xc0de`

`sma 0xca`
`mov r0, 0xfe`
`ldr r0, [r0]`
	Loads the value from address `0xcafe` into `r0`

`lds r1, [0x42]`	Loads the value from address `0xff42` which is translated into `0x{sp}42` into `r1`
`stf r0, [0xab]`	Stores the value in `r0` to address `0xffab` with incremented SP which is translated into `0x{sp+1}ab`

3.2.2.3 Miscellaneous Instructions

There are four more instructions that are essential:

- mov
- b
- call
- ret

The `mov` instruction either takes two registers or one register and an 8 bit immediate value as parameters. When specifying two registers, the content of the second register is copied to the first register. Otherwise, the immediate value is stored in the register. The branch (`b`) instruction takes a 16 bit immediate value which is used as the new PC content. It is the only conditional instruction that is available in the EDiC instruction set. The second column of table 3.1 lists all the possible suffixes for conditional branches and their meanings. If the condition is met, the branch is executed, otherwise the instruction has no effect.

The `call` instruction also takes a 16 bit immediate address which is the destination address for the call. In contrast to the branch instruction, the call is not conditional (i.e. it is always executed) and has the side effect of incrementing the SP and storing the current PC on the stack at address `0x{sp}ff`.

The `ret` instruction is used at the end of a function without any parameters to restore the PC from the stack at address `0x{sp}ff` and decrement the SP again.

Some examples:

`mov r0, 0xda`	Sets `r0` to `0xda`
`mov r1, r0`	Copies the value of `r0` to `r1`
`cmp r0, 10` `blt 0x42`	Branches to address (sets the PC to) `0x42` if the value of `r0` is smaller than 10
`call 0x100`	Calls a function at address `0x100`
`ret`	Returns from a function to the caller

3.2.3 Constants

One main improvement that an assembler allows over manually setting the instruction bits is the use of constants in the code. They can be declared to represent a value and then used similarly to variables of higher level languages instead of hard coded numbers. The EDiC assembler supports three kinds of constants: Value constants, labels and string constants.

3.2.3.1 Value constants

Value constants are the easiest kind of constants available. The first two lines of code examples 3.6 and 3.8 both declare a value constant that is used exactly like in higher level languages. In each instruction that takes an immediate value the immediate value can be specified with the name of the constant and the value of the constant is then used instead. In code example 3.8 line 5 (`ldr r0, [PRNG_SEED]`) is assembled into the same instruction as `ldr r0, [0x00]`. Constant declarations have the format `<name> = <value>`.

These value constants can be used to make the code easier to understand. For example `str r0, [SIMPLE_IO]` makes it clearer that the value of r0 is not stored in some memory location but rather send to some I/O device (in this case the internal I/O register from section 2.2.6). It also prevents errors where a typo in an address causes unintended behavior of the code.

3.2.3.2 Labels

Instruction labels are often used in assembler languages and are very important. They are declared by specifying a label name followed by a colon and hold the address of the next instruction. Then, they can be used as immediate values for branch and call instructions to jump to the instruction followed by the label declaration. As

```
1  PRNG_SEED = 0x0000          | // no instruction
2  SIMPLE_IO = 0xfe00          | // no instruction
3                              | b 0x0a // inserted by assembler
4  prng:                       | // no instruction
5    ldr r0, [PRNG_SEED]       |   ldr r0, [0x00]
6    subs r0, 0                |   subs r0, 0
7    beq prngDoEor             |   beq 0x07
8    lsl r0, 1                 |   lsl r0, 1
9    beq prngNoEor             |   beq 0x08
10   bcc prngNoEor             |   bcc 0x08
11 prngDoEor:                  | // no instruction
12   xor r0, 0x1d              |   xor r0, 0x1d
13 prngNoEor:                  | // no instruction
14   str r0, [PRNG_SEED]       |   str r0, [0x00]
15 ret                         | ret
16                             |
17 start:                      | // no instruction
18   mov r0, 0                 |   mov r0, 0
19   str r0, [PRNG_SEED]       |   str r0, [0x00]
20   prng_loop:                | // no instruction
21     call prng               |     call 0x01
22     str r0, [SIMPLE_IO]     |     str r0, [0xfe00]
23   b prng_loop               |   b 0x0c
```

Code Example 3.8: The PRNG of code example 3.6 with the constants and labels resolved.

seen in code example 3.8 the line 21 (`call prng`) is assembled into the instruction `call 0x01` which is the location of the instruction after the declarations of the `prng` label (`ldr r0, [PRNG_SEED]`).

The load instruction from line 5 is actually the first instruction of the PRNG algorithm, however, it is not assembled as the first instruction. This is due to a special label being declared in the code at line 17. When the `start` label is declared, then a new instruction is inserted at the beginning which unconditionally branches to the instruction after the start label. This can be seen in code example 3.7 where the first instruction is a `b 0x0a` because the first instruction after the start label got assembled to the address `0x0a`. The use of the start label comes especially clear in the section 3.2.4.

3.2.3.3 String constants

The third constant is rather advanced and uses very EDiC specific features. It allows the definition of character strings with a maximum length of 255 chars which can

```
1  include "prng.s"
2  include "uart_16c550.s"
3  0x20.WON_STRING = "You won!!! Score: "
4  lost:
5    // [...]
6    // output the lost string
7    mov r0, WON_STRING
8    call outputString
9    // output the score
10   ldr r0, [SNAKE_LENGTH]
11   call outputDecimal
12   // [...]
13
14 // r0: address of string
15 outputString:
16   str r1, [0xfffe]
17   sts r0, [0x00]
18   mov r1, 0
19   outputStringLoop:
20     lds r0, [0x00]
21     sma r0
22     ldr r0, [r1]
23     cmp r0, 0
24     beq outputStringEnd
25     call uart_write
26     add r1, 1
27     cmp r1, 255
28     bne outputStringLoop
29   outputStringEnd:
30   ldr r1, [0xfffe]
31 ret
```

Code Example 3.9: Excerpts of the Snake assembler program used in the demo in figure 1.1.

later be used. Differently to the value constants of section 3.2.3.1 strings cannot be used as parameters for instructions directly, because a string is a rather complex data structured in the context of assemblers. In the EDiC assembler a string can be defined as shown in code example 3.9 line 3 with the syntax <address>.<name> = "<value>". In the example a string constant with the name “WON_STRING” is defined to have the content “You won!!! Score: ‘’ at the address 0x20. The EDiC assembler treats a string as an NULL-terminated array of characters, meaning that the characters are stored consecutively in memory and after the last character a NULL-byte is stored to mark the end of the string. The address of a string constant

```
mov r0, 0x59 // 'Y'
str r0, [0x2000]
mov r0, 0x6f // 'o'
str r0, [0x2001]
mov r0, 0x75 // 'u'
str r0, [0x2002]
mov r0, 0x20 // ' '
str r0, [0x2003]
mov r0, 0x77 // 'w'
str r0, [0x2004]
mov r0, 0x6f // 'o'
str r0, [0x2005]
mov r0, 0x6e // 'n'
str r0, [0x2006]
mov r0, 0x21 // '!'
str r0, [0x2007]
mov r0, 0x21 // '!'
str r0, [0x2008]
mov r0, 0x21 // '!'
str r0, [0x2009]
mov r0, 0x20 // ' '
str r0, [0x200a]
mov r0, 0x53 // 'S'
str r0, [0x200b]
mov r0, 0x63 // 'c'
str r0, [0x200c]
mov r0, 0x6f // 'o'
str r0, [0x200d]
mov r0, 0x72 // 'r'
str r0, [0x200e]
mov r0, 0x65 // 'e'
str r0, [0x200f]
mov r0, 0x3a // ':'
str r0, [0x2010]
mov r0, 0x20 // ' '
str r0, [0x2011]
mov r0, 0x0 // NULL-byte
str r0, [0x2012]
mov r0, 0 // restore r0
```

Code Example 3.10: The instructions resulting from the string definition of code example 3.9 line 4.

actually defines the upper 8 bits of the address where the string is stored and is also the value of the constant itself. This means that the string in the example is actually stored at addresses `0x2000` to `0x2012` (18 characters plus 1 NULL-byte) and `mov r0, WON_STRING` in line 7 is equivalent to `mov r0, 0x20`. As the assembler has no direct control over the memory contents like, for example, the ARM assembler, each string declarations results in two instructions per character that are inserted at the start of the program[1] as shown in code example 3.10.

Code example 3.9 lines 15 to 31 show a function that gets the upper 8 bits of the string address as a parameter in `r0`. It outputs the characters one by one in a loop until the NULL-byte is reached. To retrieve each character, firstly the `sma` instruction is called with the MSBs of the address and then the `ldr` instruction with the loop register `r1` as an address argument is called. The character (in `r0`) is then passed as an argument to the `uart_write` function.

[1]Before the `b start` instruction that is inserted when a start label exists.

3.2.4 File imports

An important factor of software development is reusability. This also holds for assembler development and is the reason why the EDiC assembler supports including other assembler files. Including files can be used to write a utility library and then importing its functions for multiple projects. This way, a bug fix in the utility library will be fixed across all projects at the same time.

As can be seen in code example 3.9 lines 1 and 2, the EDiC assembler supports the `include` keyword followed by a relative or absolute filename in double quotes. Before assembling a file, all the include statements are replaced with the content of the file specified. All the constants and labels are imported as usual with some exceptions:

- The start label of all included files is discarded, and the main file is required to provide a start label. Otherwise, the starting point is ambiguous and probably not where the programmer expects it.
- Constants from included files can be overwritten in the main file. This can be useful when value constants hold memory locations of global variables that need to be repositioned in the main file. This also shows why it is important to use value constants for memory locations of global variables.

3.2.5 Syntax Definition for VS Code

Syntax Highlighting has become a very important factor for software development as Integrated Development Environments (IDEs) grow more capable. The highlighting is usually done by firstly, parsing the syntax and associating parts of the text file with specific categories and, secondly, assigning styles like font color to these categories. This way, a programmer can select a global color scheme which will define colors for different categories for all programming languages. When applied correctly, code in different languages becomes easier to recognize because variables are always colored the same way, no matter the language. The syntax parser, however, needs to be selected for each file type and categorize the file content correctly.

Even though the EDiC syntax is similar to the ARM syntax, it is not syntactically identical which makes syntax highlighting in editors difficult. As can be seen in code example 3.9 line 3, the ARM syntax definition used for the highlighting in this document is not perfect (The leading 0 is red, and the string is not colored correctly).

```
include "prng.s"
include "uart_16c550.s"
0x20.WON_STRING = "You won!!! Score: "
lost:
  // [ ... ]
  // output the lost string
  mov r0, WON_STRING
  call outputString
  // output the score
  ldr r0, [SNAKE_LENGTH]
  call outputDecimal
  // [ ... ]

// r0: address of string
outputString:
  str r1, [0xfffe]
  sts r0, [0x00]
  mov r1, 0
  outputStringLoop:
    lds r0, [0x00]
    sma r0
    ldr r0, [r1]
    cmp r0, 0
    beq outputStringEnd
    call uart_write
    add r1, 1
    cmp r1, 255
    bne outputStringLoop
  outputStringEnd:
  ldr r1, [0xfffe]
ret
```

Figure 3.1: The syntax highlighting with the EDiC Visual Studio Code Extension and the Atom One Light Theme [3].

As Visual Studio Code [22] is one of the leading extensible code editors, an extension for EDiC assembler has been developed and published for the EDiC [27]. The code example 3.9 is shown again in figure 3.1 with the syntax highlighting of the developed extension. The extension itself mainly consists of a TextMate language definition [20] and configuration files to work correctly with Visual Studio Code. TextMate is a tokenization engine which works with a structured collection of regular expressions as language definitions.

4 FPGA Model

The goal of the FPGA model is to proof the general workings of the CPU architecture before finalizing the hardware layout and PCB design. With the design running on an actual FPGA it is also possible to debug and test extension cards without the actual hardware of the EDiC.

4.1 FPGA Background

An FPGA can be seen as an intermediary between Application-Specific Integrated Circuits (ASICs) and general purpose CPUs. It allows for a lot more design flexibility in contrast to ASICs by being reprogrammable but at the same time has similar applications. The first FPGA was released by Altera in 1984 which featured a quartz window to erase the Erasable Programmable Read-Only Memory (EPROM) cells that hold the configuration. It only had eight macrocells and a maximum frequency of about 30MHz [4]. Today's FPGAs can have several million logic elements with several hundred MBs of Block RAM (BRAM), more than a thousand floating-point Digital Signal Processors (DSPs) and usual frequencies of more than 200MHz. However, the general idea of how FPGAs work stayed the same:

Field Programmable means that the FPGA can be programmed in the application field, even though configure is the better word to be used.
Gate Array stands for an array of logic gates which make up the FPGA. These logic gates can then be freely routed by the developer and with that different logic functions can be implemented.

FPGAs are built out of so-called Configurable Logic Blocks (CLBs) which can be connected with each other to create larger designs. Such a CLB contains several elements like Lookup Tables (LUTs), registers and Multiplexers (MUXs) which allows one CLB to provide different functionality as needed. Each LUT can encode any kind of multi-bit boolean functionality. Figure 4.1 shows how a 2-bit LUT is built out of three 2-to-1 MUXs. Depending on the input values of the SRAM into the MUXs, a different logic function can be implemented. For example: For a `NAND` function, the

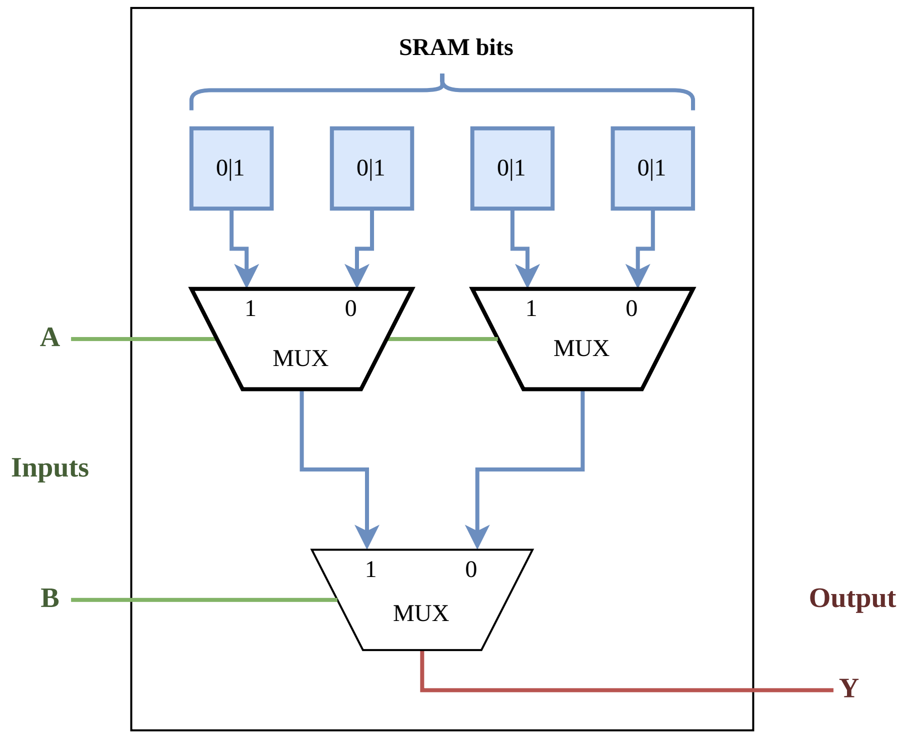


Figure 4.1: Internal structure of a 2-bit LUT

SRAM is loaded with the bits `0111`. In FPGAs these LUTs usually take 4-6 bit inputs and can, therefore, implement more complex logic functions.

Combining these LUTs with registers, complex hardware DSPs and a lot more advanced hardware, modern FPGAs are very capable and complex devices that are increasingly used in prototyping and low to medium quantity products. There are several cheaply available FPGAs development boards that are very well suited for a FPGA prototype of the EDiC.

4.2 FPGA Choices

For the EDiC the Nexys A7 development board [11] with the AMD-Xilinx Artix 7 XC7A100T-1CSG324C FPGA has been chosen. Its synthesis tool is the AMD-Xilinx Vivado [34] which is available as a free version and includes an advanced simulation environment.

4.2.1 Language Choice

There are two main Hardware Description Languages (HDLs): Verilog and VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL).

```
assign s_cin[0] = i_ctrlAluSub;
for (i = 0; i < 8; i=i+1) begin
  assign s_yXor[i] = i_a[i] ^ s_b[i];
  assign s_yAnd[i] = i_a[i] & s_b[i];
  assign s_yAdder[i] = s_cin[i] ^ s_yXor[i];
  assign s_cin[i + 1] = s_yAnd[i] | (s_cin[i] & s_yXor[i]);
end
```

Code Example 4.1: Behavioral Verilog Description of the Adder (including XOR and AND) of the ALU module.

Both are widely supported and can also be used in the same project with the help of mixed-language compilation. At the Technical University Berlin (TUB) VHDL is taught, however, in general both are used about equally often [24]. As Verilog is often cited as being less verbose and, therefore, easier to write and understand it was chosen as the hardware description language.

Code example 4.1 shows the Adder described in Verilog as an example. It iterates over all 8 bits, calculates the XOR and AND results and based on these and the carry input, the bit result and the carry output is calculated.

4.2.2 Tri-state Logic in 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 slices. Most FPGAs do have bidirectional tri-state transceiver for I/O but not for internal logic routing. However, the HDLs (both VHDL and Verilog) support tri-state logic and the AMD-Xilinx Simulation tool also does. Therefore, a simulation with tri-state logic would work, but it cannot be synthesized.

This is solved with a custom module for each tri-state network "tristatenet.v". Each tri-state driver exposes the current data and output enable signal to the tri-state module which then has only one output which represents the value of the net. If none of the drivers have an active output enable, the output is `0xff`; if one of the drivers has an active output enable, the output represents its value. Furthermore, if more than one driver has an active output enable, an error is raised. The module's logic representation for a tri-state net with two inputs is shown in figure 4.2. The output `o_noe` is only active (low) if exactly one input `i_noe` is active (low) and depending on it, the data output is selected. For this FPGA this tri-state logic module is implemented with one `LUT4` primitive per output data bit.

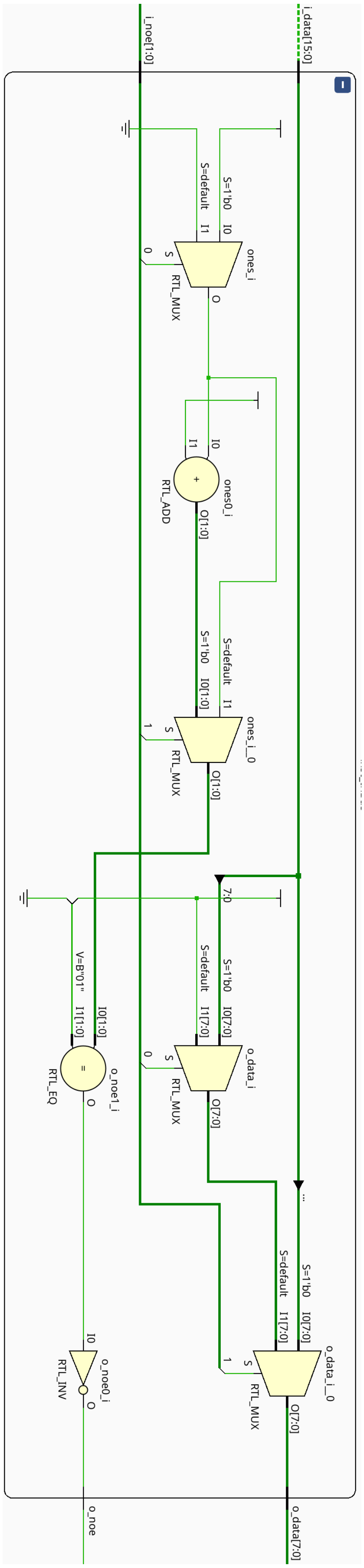


Figure 4.2: The elaborated tri-state module with two 8 bit inputs.

```
mov r1, 0x12
add r1, 0x2f
end:
b end
```

Code Example 4.2: The code for the waveform example of figure 4.3.

4.3 Behavioral Implementation

Two kinds of FPGA designs were developed during the development. The first is a behavioral description of the whole CPU and, therefore, only models the general workings of a module but does not describe the individual chips that are used in the final hardware assembly. The description in code example 4.1, for example, is a behavioral description as it only describes the logical level of what should happen. This is quite useful for development, because it is quickly changed and bugs are fixed more efficiently as opposed to a chip-level model.

To visualize how a behavioral simulation looks like, a simulation of the code in code example 4.2 is shown in figure 4.3. The first instruction (`mov r1, 0x12`) starts at 1 µs where the instruction step counter is 0 and the instruction fetch is executed. Step 1 increments the program counter and starts the instruction decoding. The `mov` instruction only consists of one step and, therefore, the `ctrInstrFinishedN` signal is asserted in step 2 together with the control signals of the actual instruction. Due to `ctrInstrFinishedN`, the step counter is reset to 0 and the second instruction (`pc==1`) is executed. After the instruction fetch steps, the ALU adds `0x12` and `0x42` at 2 µs and writes the result into `r1` at `step==3`. The third instruction then just branches to itself, resulting in an infinite loop.

4.4 Chip-level Implementation

With the behavioral simulation working, the hardware schematic can be developed. The schematic as well as the placing and routing for the PCB is described in chapter 5. However, for the EDiC it was decided to add another verification step after developing the schematic. From the schematic a netlist is generated which is usually used to summarize all the components and connections in a machine-readable format for the software that does placing and routing. In this case, a tool was written which converts a given netlist into a Verilog file which can be compiled and synthesized by Vivado.

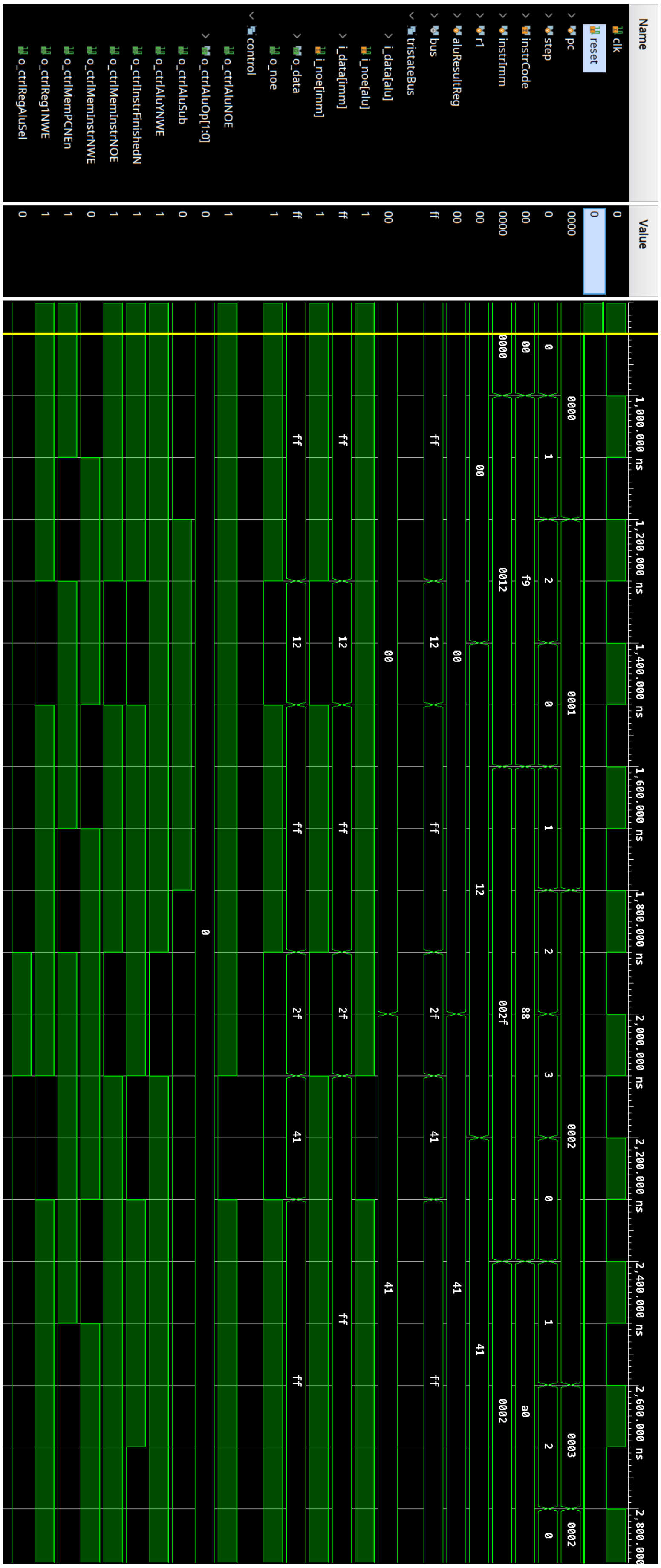


Figure 4.3: Waveform of the relevant signals for setting a register to 0x12 and adding 0x2f to it (Assembler code is shown in code example 4.2).

```
(instance U54
 (viewRef NetlistView
  (cellRef &74AS867_0
  (libraryRef OrCAD_LIB))) (designator "U54")
 (property PCB Footprint (string "DIP.100/24/W.300/L1.175"))
 (property Name (string "I656203"))
 (property Value (string "74AS867"))
 (portInstance &3)
 (portInstance &4)
 (portInstance &5)
 (portInstance &6)
 (portInstance &7)
 (portInstance &8)
 (portInstance &9)
 (portInstance &10)
 (portInstance &14)
 (portInstance &22)
 (portInstance &21)
 (portInstance &20)
 (portInstance &19)
 (portInstance &18)
 (portInstance &17)
 (portInstance &16)
 (portInstance &15)
 (portInstance &13)
 (portInstance &24)
 (portInstance &12)
 (portInstance &11)
 (portInstance &23)
 (portInstance &1)
 (portInstance &2))
```

Code Example 4.3: An EDIF definition of an instance as exported by OrCAD/CAPTURE.

4.4.1 Conversation Script

The netlist file used is an `*.edn` which is exported by OrCAD/CAPTURE version 9.2.1.148. It follows the Electronic Design Interchange Format (EDIF) and contains a list of all instances (i.e. ICs and other components) with port numbers and a second list of all nets (connections between ports). The conversion script consists of a parser which analyzes such a netlist. The parsed netlist is then further processed until a Verilog file can be created. The generated Verilog file only consists of wire definitions and module instantiations. Each of the instantiated modules has its own,

```

(net PCINO
 (joined
  (portRef &18 (instanceRef U52))
  (portRef &18 (instanceRef U51))
  (portRef &3 (instanceRef U54)))
 (property Name (string "PCINO")))
```

Code Example 4.4: An EDIF definition of a net as exported by OrCAD/CAPTURE.

manually written implementation. The implementation for an `74F08` (quad AND gate) is, for example, shown in code example 4.5.

Code example 4.3 specifies the instance U54 which is an `74AS867`. The format also specifies the port numbers, but they are not processed by the parser because they are not required. Code example 4.4 then specifies a net with the name `PCINO` which connects U52 port 18 with U51 port 18 and U54 port 3. In this case U52 and U51 are both `74F245` octal bus transceivers where port 18 is the B0 tri-state output port and U54 is a `74AS867` (synchronous up/down counter with load) where port 3 is the D0 input port. Depending on the control signals of U51 and U52 this net connects either the 0th bit of the bus or the instruction immediate with the 0th bit of the load input of the PC. Internally, the list of instances and list of nets are combined into a list of instances where each instance contains a mapping of port numbers to connected nets.

The parser discards all components except logic ICs (ID starting with ‘U’) and 0 resistors. The schematic includes some 0 resistors between control signals to be able to rewire them more easily on the PCB if needed. As they essentially behave as direct connections, the nets on either side of one 0 resistor are merged.

The basic instances are easily converted to Verilog instantiations. However, there are some obstacles that need to be taken with more advanced instances.

4.4.1.1 EEPROM

The six EEPROMs (three for the instructionROM and three for the microcode) need to be instantiated with the correct data loaded into them. Those six instantiations are identified by the unit ID and the wires are then connected to one of the custom

```
// quad and https://www.ti.com/lit/ds/symlink/sn74ls08.pdf
module ic74x08(
input wire port1,
input wire port2,
output wire port3,
input wire port4,
input wire port5,
output wire port6,
input wire port7,
output wire port8,
input wire port9,
input wire port10,
output wire port11,
input wire port12,
input wire port13,
input wire port14
);

assign port3 = port1 & port2;
assign port6 = port4 & port5;
assign port8 = port9 & port10;
assign port11 = port12 & port13;

endmodule
```

Code Example 4.5: Verilog implementation for the 74F08 IC.

AMD-Xilinx ROM IP Cores which are configured with the respective initial values. The addresses for one ROM instantiation are used and then all 24 data ports from the three EEPROMs are connected resulting in a Verilog instantiation as shown in code example 4.6.

4.4.1.2 Tri-state Ports

Some ICs provide tri-state ports. As discussed above, they cannot be implemented on FPGAs and, therefore, need to be converted. The same tri-state net component as in the behavioral implementation is used. However, for this to work, each bidirectional port of the ICs needs to be replaced by one input and one output port. Also, one output enable port needs to be added. Then the output port that replaced the bidirectional port is connected to an input of the tri-state net instance and a new net is created for each tri-state net which is the actual value of the net (the output of the tri-state net module). The tri-state net for the `PCINO` signal (code example 4.4) is represented by the instantiation shown in code example 4.7.

```
microCodeRom inst_microCodeRom (
  .clka(i_asyncEEPROMSpecialClock),
  .addra({MC_A14, MC_A13, MC_A12, MC_A11, MC_A10, MC_A9, MC_A8,
↪  MC_A7, MC_A6, CTRLALUOP1_SRC, CTRLALUOP0_SRC, CTRLALUSUB_SRC,
↪  MC_A2, MC_A1, MC_A0}),
  .douta({unconnected_U87_19, unconnected_U87_18,
↪  unconnected_U87_17, CTRLINSTRFINISHED_SRC,
↪  CTRLMEMPCTORAM_SRC, CTRLMEMPCFROMIMM_SRC, CTRLMEMPCEN_SRC,
↪  CTRLMEMRAMOE_SRC, CTRLMEMRAMWE_SRC,
↪  CTRLMEMINSTRIMMTORAMADDR_SRC, CTRLMEMMAR1WE_SRC,
↪  CTRLMEMMAR0WE_SRC, CTRLMEMINSTROE_SRC, CTRLMEMINSTRWE_SRC,
↪  CTRLMEMSPEN_SRC, CTRLMEMSPUP_SRC, CTRLMEMPCLOAD_SRC,
↪  CTRLREG1BUSOE_SRC, CTRLREG0BUSOE_SRC, CTRLREGALUSEL_SRC,
↪  CTRLREG1WE_SRC, CTRLREG0WE_SRC, CTRLALUOE_SRC,
↪  CTRLALUYWE_SRC})
);
```

Code Example 4.6: Verilog instantiation of the microcode ROM generated out of three EEPROM instantiations.

4.4.1.3 RAM and EEPROM clock

Another problem with the FPGA implementation in general is that both, the SRAM and EEPROM chips used, are asynchronous and the FPGA only has synchronous logic elements. In the behavioral implementation, exact timings were no requirement and, therefore, the memory and ROMs were clocked with the inverse clock, mimicking an asynchronous behavior. However, for the exact netlist FPGA implementation this is not a good way to imitate the behavior. Therefore, the exact delays of both chips were calculated with the help of the datasheets, and they are both clocked with a custom clock that is out of phase with the global logic clock by the exact amount of the delay.

This means, that the clock inputs of the memory and EEPROM instantiations are replaced with the corresponding custom clock as can be seen in code example 4.6.

Figure 4.4 visualizes how the main clock (`CLK1` in the waveform) and the clock for the ROM (`asyncEEPROMSpecialClock` in the waveform) differ in phase. Consequently, the step register and with it the address for the microcode ROM change with a rising edge of the main clock while all the control signals which are outputs of the ROM change with the rising edge of the phase shifted clock.

```
tristatenet #(
  .INPUT_COUNT(2)
) inst_triBusPCIN0 (
  .i_data({PCIN0_U51, PCIN0_U52}),
  .i_noe({U51_b_noe, U52_b_noe}),
  .o_data(PCIN0),
  .o_noe(PCIN0_noe)
);
```

Code Example 4.7: Verilog instantiation for the tri-state Net `PCIN0`.

4.4.1.4 Assignments

There are some connections which are unique to the FPGA implementation and, therefore, are not contained in the netlist. These are mainly the inputs and outputs of the CPU for the I/O extensions, the user buttons (reset, step etc.), clock oscillator, breakpoint addresses and so on. However, one exception are the `L1-L4` and `H1-H4` nets which are static nets connected to ground or 5V through resistors. They are used for logic inputs of ICs instead of directly using GND or 5V to ease the error fixing on the PCB. When connected to the pane, there is no trace that can be scratched through if the pin needs to be connected to another net. Hence, the pin usually needs to be drilled out, and the new pin needs to be insulated through the hole because it should not connect to any exposed inner layers.

In the FPGA design the nets `L1-L4` and `H1-H4` are, therefore, assigned to 0 or 1 respectively.

4.4.1.5 Display Driver

The hardware build will feature two displays for the built-in I/O which can be directly addressed with 4 bits to display hexadecimal digits. The FPGA development board on the other hand, features simpler and more common 7-segment displays. In total, there are eight 7-segment displays, two of which are used as the built-in I/O and the others are used for debugging when the CPU is halted. The wiring of the displays is shown in figure 4.5. To address these displays, a custom display driver has been developed which has eight 4 bit data inputs for the digits plus eight inputs for the dots between the digits. Additionally, 8 bits encode which 7-segment displays should be illuminated. It loops through the digits by setting each anode to `'0'` individually and setting the cathodes according to the corresponding input bits. This way, each display is illuminated for 2 ms which makes it look like all displays are illuminated all the time.

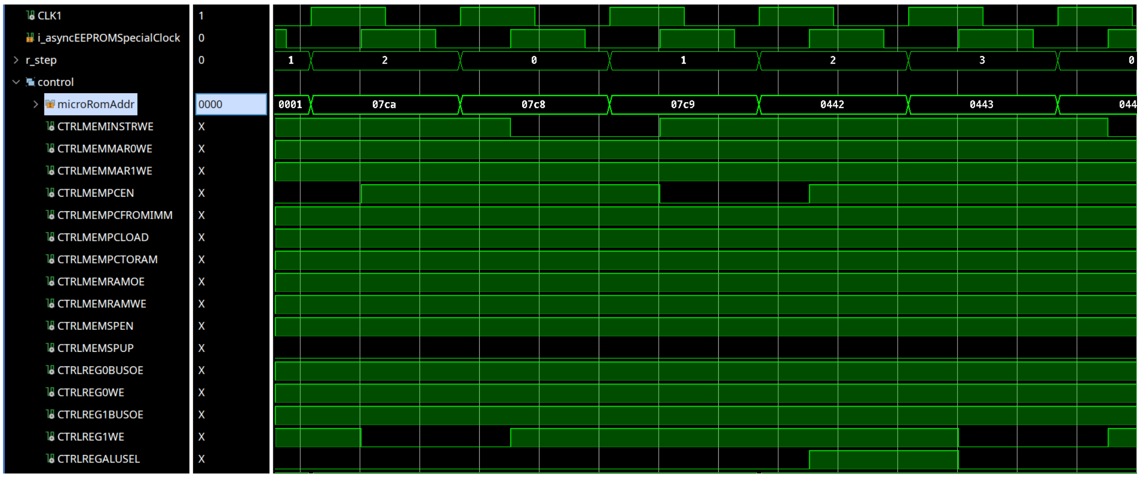


Figure 4.4: Waveform showing the clock used for the FPGA ROM to mimic the asynchronous behavior of the EEPROMs.

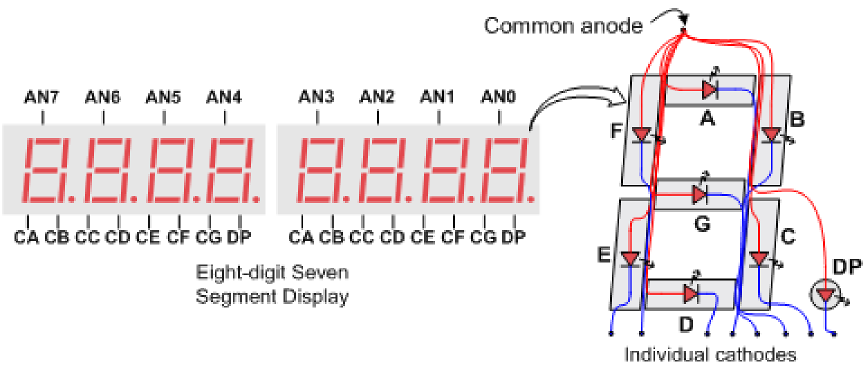


Figure 4.5: Overview of the eight 7-segment displays of the Nexys A7 development board [12].

4.4.2 RS232 I/O Extension Debugging

One goal of creating a logical replication of the CPU on a FPGA was to verify the I/O extension cards before ordering the large PCB for the CPU. All extension cards will be daughter boards and sit on top of the main PCB in a smaller form factor. The following logical connections are passed through pin headers:

- Bus (8 bits, bidirectional)
- I/O Address (lower 8 bits, to I/O)
- Control Signals:
  - $\overline{\texttt{ioCE}}$: active when the upper 8 RAM address bits equal `0xfe`.
  - $\overline{\texttt{ctrlMemRamWE}}$: write enable signal. Write should only happen when $\overline{\texttt{ioCE}}$ is active.
  - $\overline{\texttt{ctrlMemRamOE}}$: output enable signal. Read should only happen when $\overline{\texttt{ioCE}}$ is active.
  - `clk`: Clock signal.
  - $\overline{\texttt{reset}}$: Reset signal.

Additionally, ground and 5V is passed through the connector. However, the Nexys A7 FPGA development board only features digital 3V3 connections on the side. Thus, an adapter board is required which converts the 3V3 voltages to 5V and the other way around while providing the correct pin locations for the Nexys A7 board and the daughter board. Its schematic is shown in figure 4.6 where the `74LVC8T245` is used as a voltage converting buffer. For the control signals and addresses, its direction is always from the A port (3V3) to the B port. The direction pin of the bus buffer, however, is low (from B to A) when both, output enable and chip enable, signals are active.

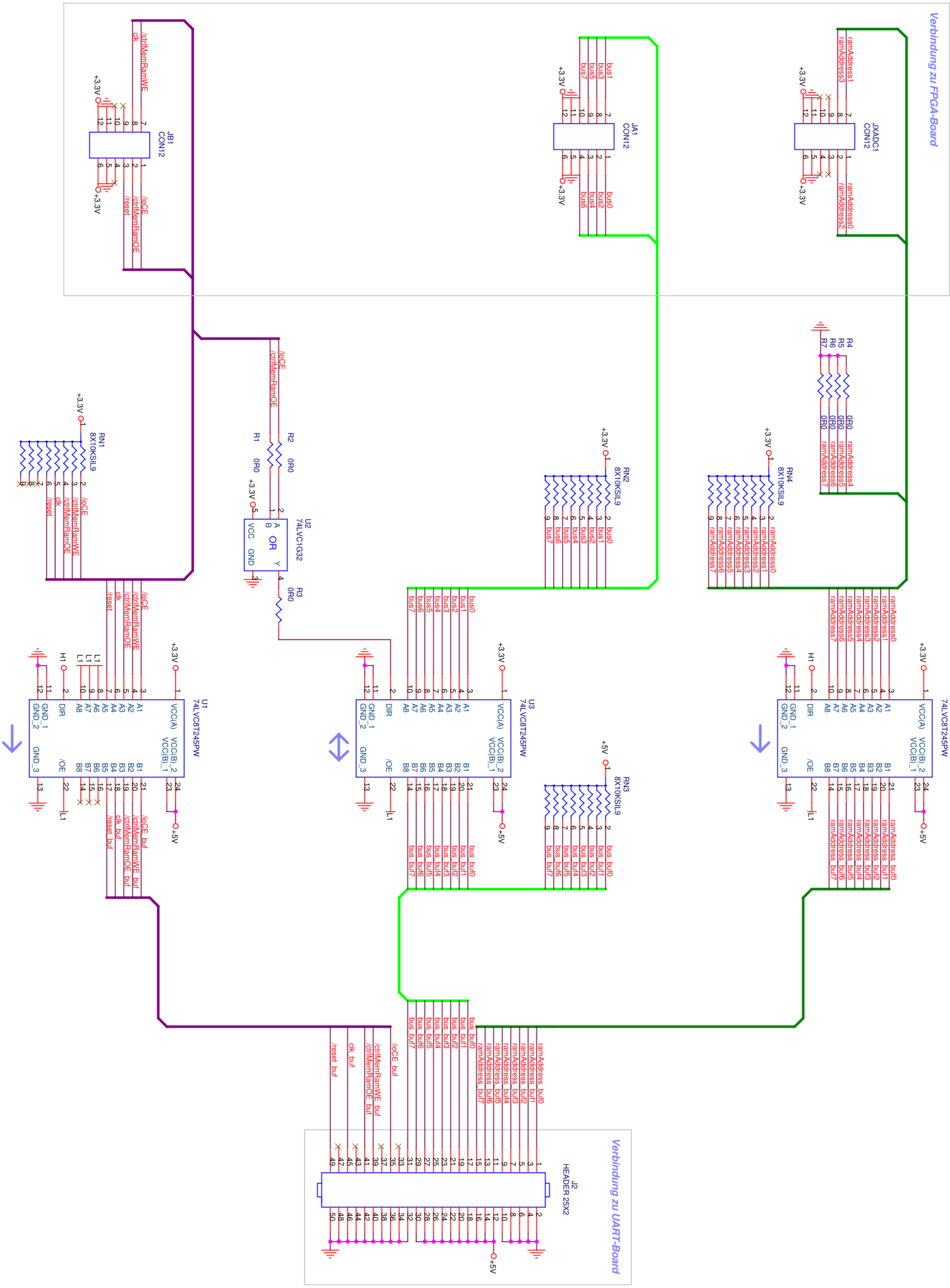


Figure 4.6: The Schematic for the 3V3 to 5V conversion cards with the FPGA development board.

5 Hardware Design

After the FPGA behavioral simulation has been successful, the hardware design process is started. The initial step is designing a schematic (section 5.1) which is followed by the netlist simulation and the placing and routing of components and wires in section 5.2. Additionally, a timing analysis is performed to ensure that the clock frequency is as high as it is possible without risking any misbehavior.

5.1 Schematic

The full schematics of the hardware design can be found in appendix A in figures A.1 to A.7. The schematic is created in such a way that the logical connections are easy to understand. Each IC has its pins arranged for easy understanding and the connections have meaningful names to easier understand the logic.

The 74 series of ICs is used for the EDiC. However, a lot of decisions need to be made in choosing the correct ICs as explained in detail below.

5.1.1 Register Comparison

The 74 series of logic ICs feature many 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 data on every rising edge of the clock. This is done with an additional input, called *clock enable*. Implementing the clock enable with a basic AND gate of the clock and a control bit has the major drawback that glitches of the enable control signal can propagate to the clock input of the register and, therefore, falsely trigger the register. Additionally, every logic gate has a propagation latency which would result in a clock signal being delayed at some ICs which, in turn, may lead to timing problems. There are two widely used alternatives to the simple AND

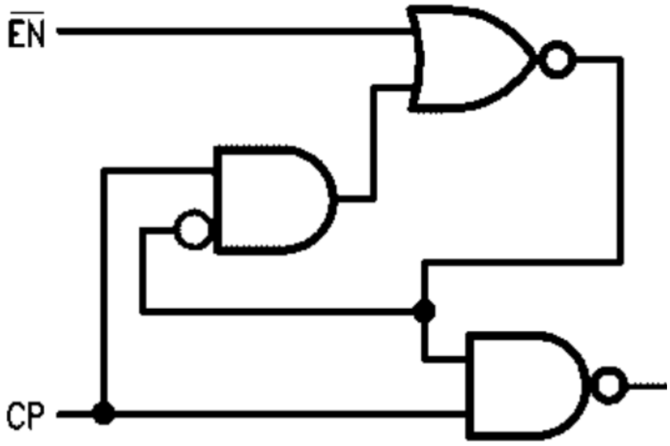


Figure 5.1: Clock Enable circuit of the *74F825* IC [8].

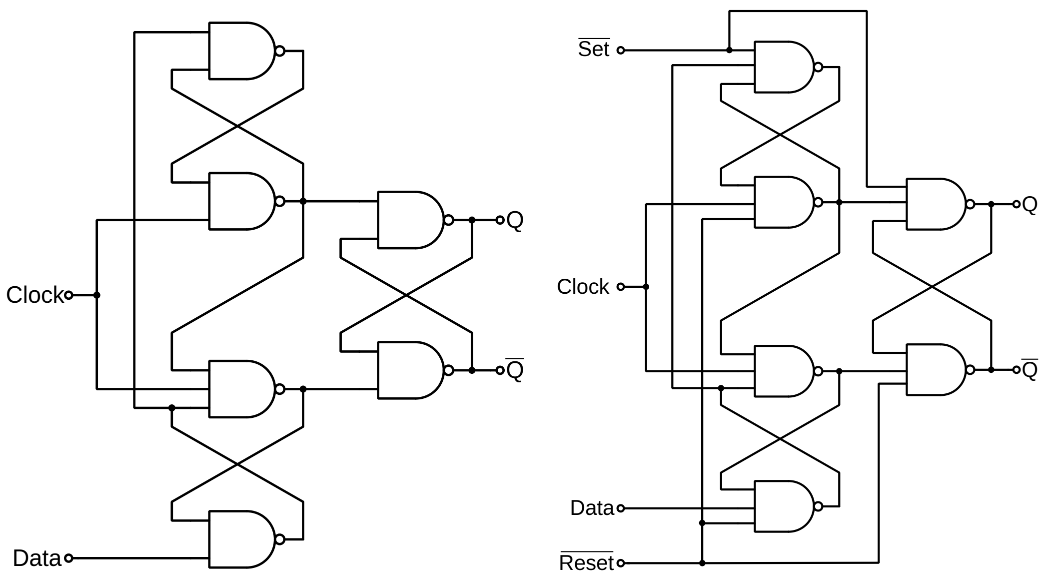
gate: The enable input can be used as the select input for a multiplexer to the data input of the flip-flop, where it multiplexes between the actual input and the current output. This allows the flip-flop to always capture data but when the enable input is inactive, it recaptures the current output. The drawbacks are that each bit of the register needs a multiplexer at the input and, furthermore, that the flip-flops draw power on every clock pulse, even though no new data is captured. The *74F825* logic IC solves this with the circuit shown in figure 5.1. When the $\overline{\text{EN}}$ input is low, the NAND gate on the right negates the CP[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 5.3.

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 an asynchronous 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 5.2.

A third feature that may be important is a tri-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.

The register that was chosen for the EDiC is the *74F825* because it has all three

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



(a) Classical D-type flip-flop built out of three $\overline{\text{SR}}$ NAND latches [25].

(b) D-type flip-flop modified to support $\overline{\text{Clear}}$ and $\overline{\text{Set}}$ [29].

Figure 5.2: Comparison of D-type flip-flops with and without $\overline{\text{Clear}}$ and $\overline{\text{Set}}$.

features and is 8 bits wide. However, three other kinds of registers are also sparsely used in the EDiC:

- The *74AS867* is a more advanced synchronous counter register which is used for the PC and SP. They are described below.
- The *74F374* register only features the output enable and is used once where no additional control logic is required.
- The *74F273* is used for the built-in I/O to mimic the typical asynchronous extension cards and for the buffering of user control inputs (stepping etc.) because only a reset is required there.

5.1.2 LED Driver

The EDiC features many LEDs showing the register contents to aid the understanding of the workings of a CPU. However, naively connecting the LEDs to the logic outputs of registers may lead to unwanted behavior because the outputs of all logic ICs have a limited current they can provide. This leads to the usage of specific buffers for the LEDs. Additionally, the current rating usually is higher for low-level output due to the internal workings of the output buffer. For example, the B outputs of a *74F245* non-inverting buffer are rated for maximum $-15\,\text{mA}$ for high-level output and $64\,\text{mA}$ for a low-level output [16]. Therefore, connecting the anode of a LED via a current limiting resistor to the output of a non-inverting buffer and the cathode

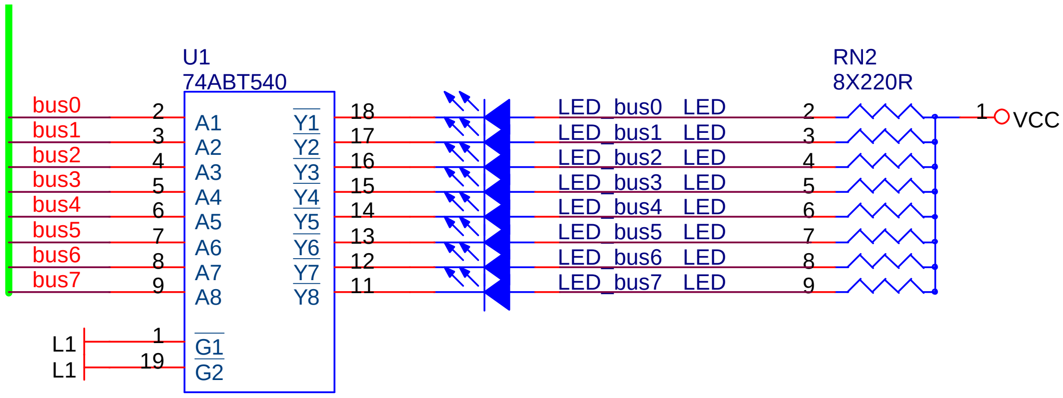


Figure 5.3: The Light-Emitting Diode (LED) connection of the main bus with the *74ABT540* as a driver.

to GND will not be ideal. To be able to draw more current from the buffer and thus having brighter LEDs, inverting buffers are used, and the LEDs are connected “backwards” as shown in figure 5.3. The *74ABT540* is the IC used as LED buffer in the EDiC with a low-level current rating of 64 mA [28]. The cathodes of the LEDs are then connected to the *74ABT540* and the anodes are connected through current limiting resistors to V_{cc}.

5.1.3 Program Counter & Instruction EEPROMs

Figure A.1 contains the PC (U54 and U55) with the instruction EEPROMs (U62, U67, U69) and the registers to store the instruction. The PC can be incremented or loaded from either an instruction immediate (U50 and U52) for branching or the SRAM (U49 and U51) for returning from a function call, visualized in figure 5.4. To facilitate these operations, the *74AS867* is used which is an 8 bit synchronous counter with loading and asynchronous clear capabilities that can be cascaded with a ripple carry output. The PC is then used as the address to the instruction EEPROMs and can also be saved to the SRAMs. As the main memory is only 8 bits wide but the PC is 16 bit wide, a second SRAM IC is used to store the upper bits of the PC in the case of a function call (see section 5.1.4). The PC is, additionally, used as A inputs to the *74F521* (U53 and U60) comparators to detect when a breakpoint is reached. The 8 bit comparators can be cascaded via the enable input to compare 16 bit values. The B input is selected by the user with four hexadecimal digit switches.

The function of the “Test” block between the output of the instruction EEPROMs

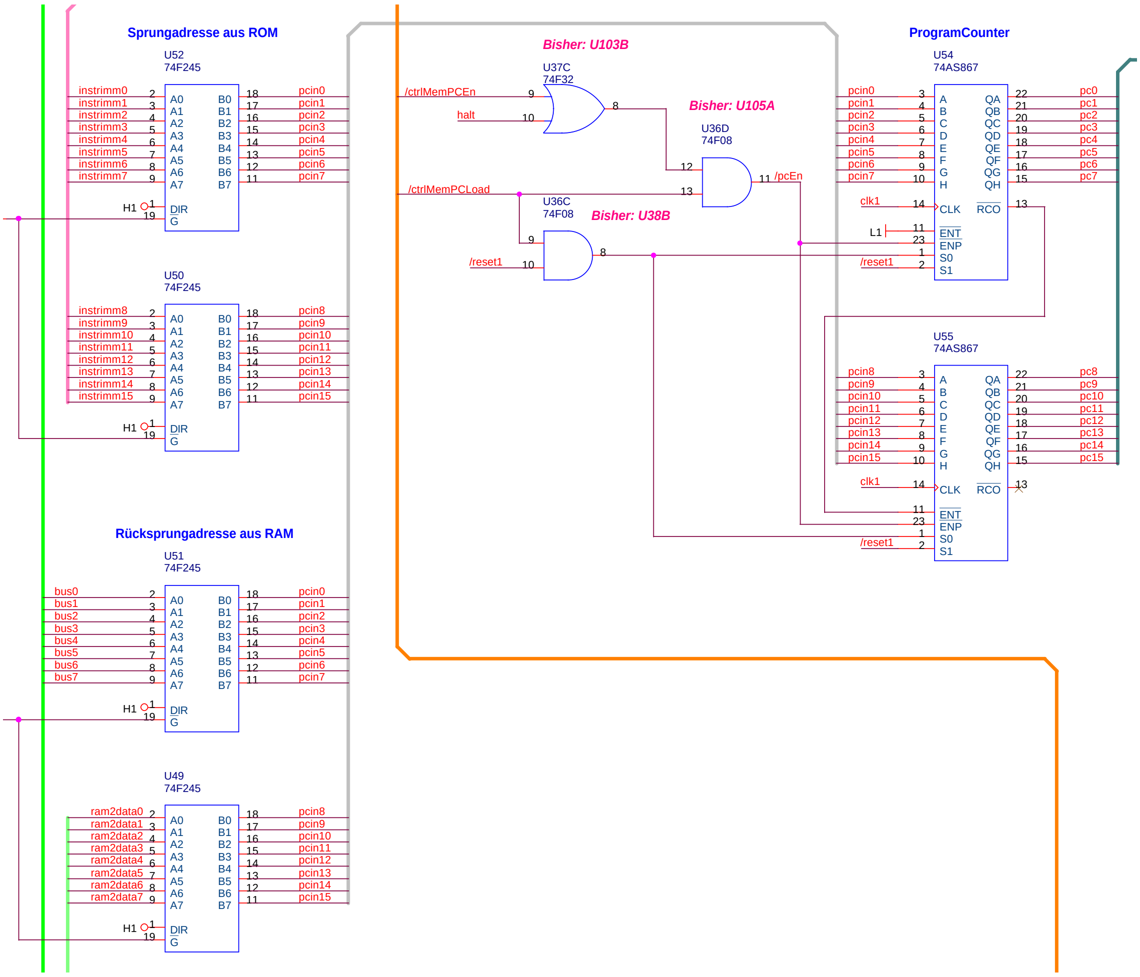


Figure 5.4: The inputs of the PC to enable incrementing, branching and returning from functions.

and the instruction registers is explained in section 6.1. For understanding the function of the schematic, it can be assumed that it shorts the connections on the left with the corresponding connections on the right. The lowest of the 3 instruction registers (U64) holds the instruction code which is used in the section 5.1.5. The upper two registers (U70 and U71) hold the immediate value which can be used as an address in the section 5.1.4, as a branch address for the PC and the lower 8 bits can be used as immediate value on the bus (U75).

All 5 registers have LEDs connected to them as described in section 5.1.2.

5.1.4 Memory

The memory module (figure A.2) features three registers used for the address logic: The MAR (U68 and U63) is a 16 bit register where the lower and upper 8 bits can be loaded independently of the bus. The SP (U56) is a *74AS867* counter register similar to the PC but only 8 bits wide and wired differently to only allow incrementing and decrementing. The three different kinds of memory accesses are decoded from the upper 8 address bits which either come from the instruction immediate (U74) or the MAR (U73):

- *I/O access:* When the upper 8 bits equal `0xfe` (U79), the I/O chip enable (CE) signal is asserted and the SRAM CE is deasserted.
- *Stack access:* When the upper 8 bits equal `0xff` (U76), the stack memory is selected. Then the upper 8 bits of the address is replaced by the SP and a 17th address bit is asserted to access the stack memory.

The address is then driven by several bus drivers according to the decoding logic (U61, U63, U65, U66 and U72).

The actual SRAM ICs (U77 and U100) have voltage levels which are not quite compatible with the standard *74F* ICs [14] which is why all the signals connecting to them are buffered with the *74ACT245* [15] (U201, U202, U203, U204 and U205).

5.1.5 Control Logic

Figure A.3 contains two registers for the address of the microcode EEPROMs (U85, U86 and U87) of which the data pins are the control signals (section 2.3). The first registers (U83) is used as a synchronous 3 bit step counter which increments each cycle except when the halt signal is asserted. The instruction finished control signal will reset the step counter to 0 at the next cycle. U83 also registers the four ALU flags and U84 registers the instruction to synchronize all address bits for the EEPROMs.

5.1.6 Clock and Reset

Figure A.4 contains the oscillator (X1) whose frequency is determined in section 5.3 and an active low reset controller (U34) which resets on power-on and can be combined with a user reset switch (SW1301). The clock and reset is buffered with an *74ABT245* for minimal latency. To avoid glitches (see section 6.2) on the four user inputs, a low pass and a Schmitt trigger and two registers are used. A multiplexer (U39) generates the halt signal from the debug user inputs and the instruction finished control signal to implement the logic described in section 2.2.7

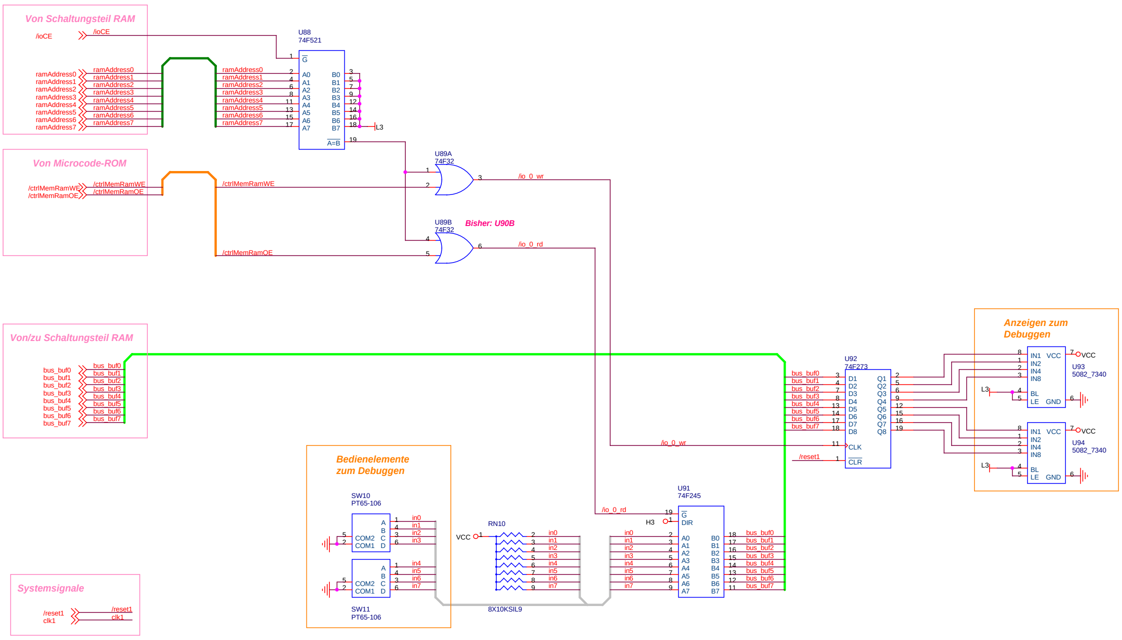


Figure 5.5: The basic I/O included in the EDiC with one register and one 8 bit display.

5.1.7 Built-In I/O

The built-in I/O (figures 5.5 and A.5) consists of one register to hold the output value (U92) which is connected to two hexadecimal displays (U93 and U94). For input two hexadecimal switches (SW10 and SW11) are used with a bus driver (U91). To control the register clock pulse and the output enable of the bus driver, the I/O CE is combined with the I/O write enable and I/O output enable and the I/O address is compared with `0x00` (U88).

5.1.8 Register Set and ALU output

The register set in figure A.6 consists of two registers (U40 and U41) which can be loaded from the bus. The register outputs can drive the bus (U44 and U45) and are multiplexed for the A input of the ALU (U42 and U43). After the combinatorial ALU (section 5.1.9), the four operation results are multiplexed (U5, U6, U7 and U8) and stored in the ALU output register (U9). Even though the ALU output register features output enable inputs, an individual bus driver is used (U10) because the content of the ALU output register should be displayed to the user (U11). The carry flags are also multiplexed (U101) as the carry flag from the shift operation is generated independently. The overflow flag is generated in the combinatorial schematic of the ALU, the negative flag is just the MSB of the output and the zero flag is deduced from a comparison with zero (U12). All four flags are then stored in a register (U97).

5.1.9 Combinatorial ALU

Figure A.7 shows the ripple carry adder on the left composed out of 8 full-adder and with subtracting capabilities. The barrel shifter on the right side is explained in depth in section 2.2.1. The carry flag resulting from a shift operation should always represent the last bit which was shifted out of the 8 bits and should be unchanged when shifting by 0. This is accomplished with another multiplexer (U102).

5.2 Placing and Routing

After designing the schematic all the components need to be placed on the PCB. All logic ICs are listed in table 5.1. For placing and routing several factors are important. As the goal of the EDiC is to be easy to understand for future students, all components were not only placed to optimize the wiring, but a focus was to place components of the same modules close together. Additionally, extra space was left to mark each module and to name all the LEDs on the silkscreen of the PCB for easier reference. Figure 5.7 is a rendering showing the traces (red and green), silkscreen (violet) and through-holes/vias (green/yellow).

Especially on larger PCBs and designs with quickly switching power consumption it is important to ensure good power delivery to all components. In the case of the EDiC this is achieved by using a 4-layer PCB with the two internals layers being filled with GND and 5V planes. The top and bottom layer are then used for logical connections where the most efficient wiring can be done when one layer mostly has vertical wires (red traces in figure 5.7) and the other layer mostly horizontal wires (green traces).

The usual problem when not using solid GND planes is the implicit low pass filter resulting from the inductance. When using dedicated GND wires to all the ICs the current flows through one trace towards the IC and through another trace back to the source. This leads to an electric field around both traces which in turn may induce current in neighboring traces and most importantly stores energy and acts as an RL-low-pass. If one would reduce the space between those traces until those traces

Table 5.1: All logic ICs used in the EDiC.

IC	Quantity	Function
74F245	17	Tri-state Octal Bus Transceiver
74ABT540	14	Inverting Octal Buffer (LED Driver)
74F157	12	Quad 2 to 1 multiplexer
74F825	10	Octal register with Tri-state, Asynchronous Clear and Clock Enable
74F86	7	Quad XOR
74F08	7	Quad AND
74F521	6	8 bit Inverting Comparator with Enable
28C256	6	EEPROM with 15 address bits
74ACT245	5	Octal Bus Transceiver used for SRAM
74F153	4	Dual 4 to 1 multiplexer
74F32	4	Quad OR
74F151	3	8 to 1 multiplexer
74AS867	3	Synchronous 8 bit cascaded counter with loading
74F273	2	Octal register with clear
74F04	2	Hex Inverter
AS6C4008	2	SRAM with 19 address bits
5082_7340	2	hexadecimal display
74ACT14	1	Hex Inverter with Schmitt Trigger
DS1813-10	1	Reset Generator
74F374	1	Octal register with output enable
74ABT245	1	Bus driver used for clock and reset
Sum	110	-

are directly beneath each other, the electrical field would be minimal. Figure 5.6 demonstrates this by visualizing the H-field around two wires in two scenarios. The blue plot assumes the two wires to be 20 mm apart (at 40 and 60 mm) and it can be seen that the opposite current flow leads to an amplified magnetic field between the two wires and also a magnetic field to the left and right of the wires. However, when moving the wires closer together, the magnetic field in between the two wires vanishes and the field to the left and right is reduced by a lot because the positive field to the right of the left wire cancels the negative field to the right of the right wire. In the extreme case of both wires being perfectly aligned in this dimension (being on top of each other for example), the magnetic field would cancel out completely.

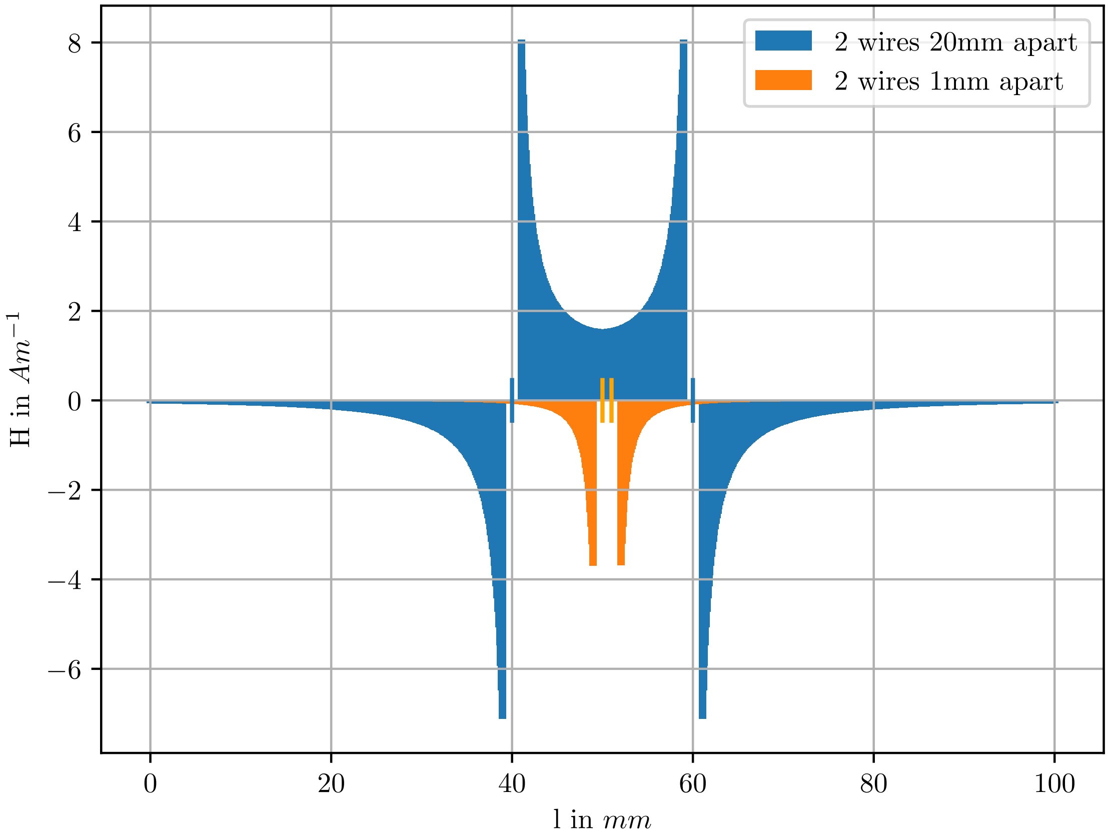


Figure 5.6: The magnetic field resulting from two wires of 1 mm thickness with 50 mA of current flowing opposite directions. For the blue plot, the centers of the two wires are 20 mm apart and for the orange only 1 mm. [1, page 187]

This is exactly what is accomplished by using a GND plane: Assuming a trace takes a 90-degree turn to reach the IC and then the current flows through the GND plane back to the source. The current on the GND plane could flow everywhere on the plane, especially the direct (diagonal) path with the least resistance or the path directly under the trace which has the least inductive reactance because the magnetic field is a lot smaller. The overall impedance is defined in section 5.2 where the capacitive reactance (X_C) can be ignored in our case. The inductive impedance, on the other hand, is dependent on the frequency and the inductance. [2, page 45]

$$Z = R + j(X_L + X_C) \tag{5.1}$$

$$Z = R + j(2\pi f L + X_C) \tag{5.2}$$

Thus, for higher frequencies, the inductance has a greater impact on the overall impedance and the current in the GND plane will flow closer to the trace to reduce the inductive reactance. For lower frequencies, the inductive reactance becomes negligible and the lower resistance of the direct (diagonal) path prevails.

Another important factor to bear in mind is to route the traces in a way that makes it possible to access every trace on the PCB. It is always possible that some bug was not detected in the schematic design or netlist simulation. If a bug has been found it may be required to cut a trace and rewire it (see chapter 6). Therefore, logical wires are always placed on the top or bottom plane and on the top plane traces will never be completely covered by ICs. If, for example, connecting pins 1 and 24 of a 24 pin IC (they are directly opposite of each other) they should not be connected directly but a detour should be taken to expose the trace from under the IC.

It is also possible that a new IC needs to be placed on the PCB to fix a bug, like an extra register or bus driver. Therefore, through holes are provided to allow spare ICs to be placed at convenient locations throughout the PCB.

TODO: Insert picture of the PCB + Testadapter.

5.3 Timing Analysis

To figure out what the maximum frequency is at which the EDiC can operate on, a detailed timing analysis was performed. The timing analysis computes the path with the longest propagation delay which is called the critical path. The delay of the critical path can then be used as a baseline for choosing the correct frequency.

Figure 5.8 visualizes how the propagation delays work: Each IC has delays which are specified in the datasheet. In the example of figure 5.8, a value of register r_0 goes through a combinatorial path and is then stored in register r_1. The registers have a propagation delay t_p which specifies the time from a rising edge of the clock to the output (Q). In theory, it is also important to hold the input data of a register for the specified hold delay t_h, however, in the EDiC this is no problem. Then the combinatorial path also has propagation delays from inputs to outputs which need to be added up (t_c). At the next register, a setup time t_s has to be met which specifies the amount of time the input data needs to be stable before the rising edge.

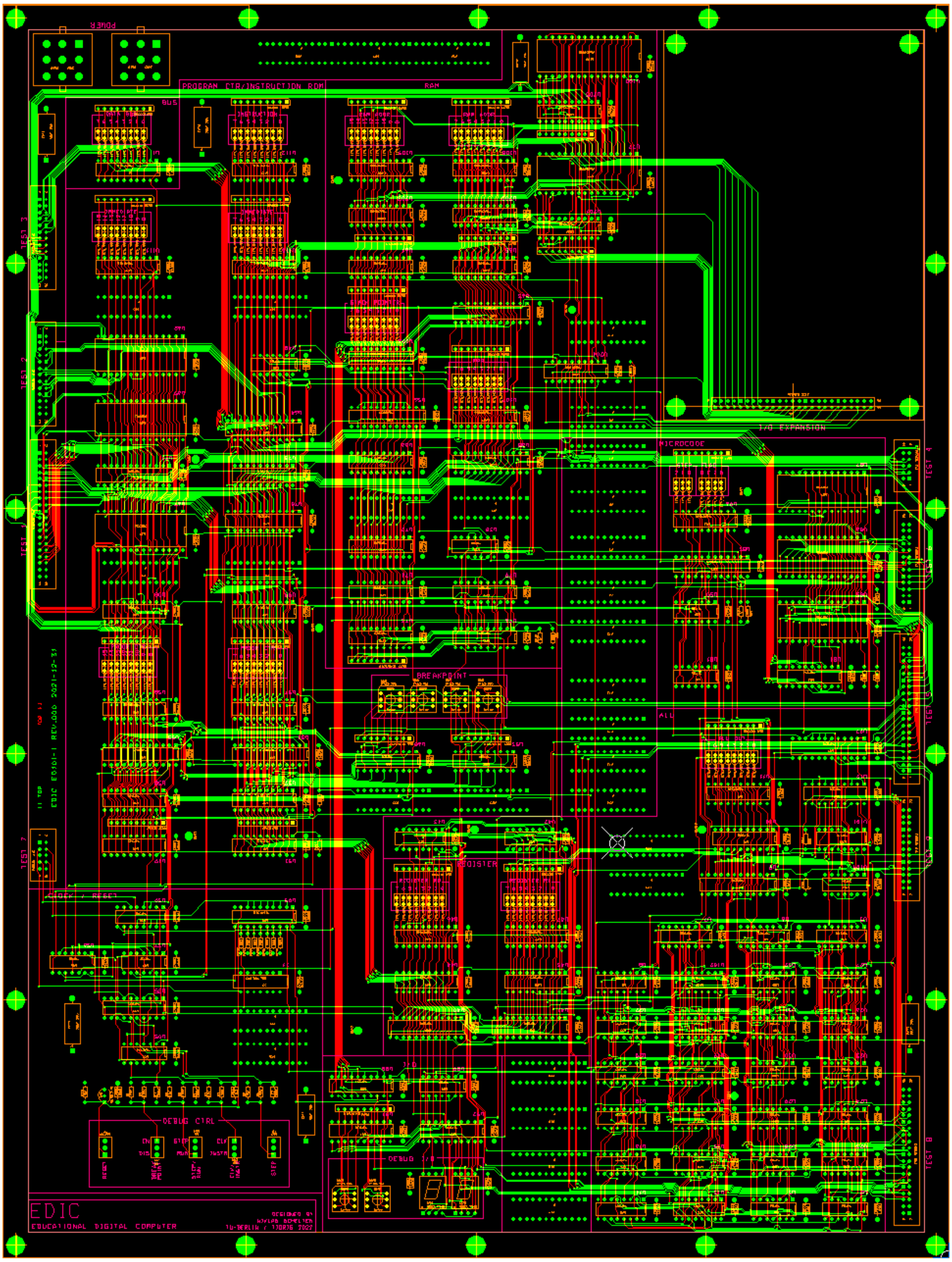


Figure 5.7: Rendering with all components placed and all the traces routed on the two signal layers (green and red).

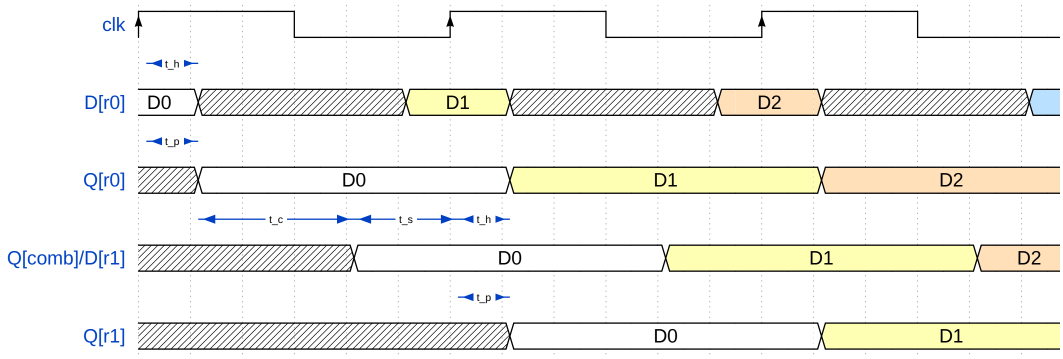


Figure 5.8: Timing relations for a combinatorial datapath between two registers.

The figures 5.9, B.1 and B.2 show three timing analysis for the EDiC. Each block represents one IC with the corresponding delay. The first row shows the unit number from the schematic, the second line the type of IC, the third shows the kind of delay and the fourth shows the amount of time. The kind of delay of a buffer can for example be d→ q which means input data to output data delay or oe→ q which is the time from asserting output enable until the data is valid. The delay time is always the worst case time as specified in the datasheet[2]. A vertical double line represents a point where multiple delay paths must be met until the execution can continue. In figure 5.9 for example, the propagation delay of register U83 (flags and step register) and register U84 (instruction) must both be over until the address for the EEPROMs U85, U86 and U87 are valid. At these points the maximum of the merging delay paths is used as the starting point for the next path. The maximum delay up to this point is also printed at the top. Additionally, some paths are labeled for clarity. All the delays of the critical path (the path that takes the longest from one starting point to one end point) are marked in red.

Figure 5.9 shows the basic latency path for control signals and the common bus driver ICs. The latencies inside the register set and program counter are negligible and, therefore, only the memory module with the complex address decoding and the ALU is further examined. For the memory module (figure B.1), there are two critical paths: The first comes from the `memInstrToRamAddr` control signal, through the stack selection logic to the memory address and finally to the buffered output data of the SRAM on the bus (281.3 ns). The second has the same origin but represents the writing option of the SRAM (272.2 ns).

[2]Propagation typically varies with the temperature and age of the IC and by taking the worst case time (maximum) it is assured that no timing bugs occur due to e.g. temperature changes.

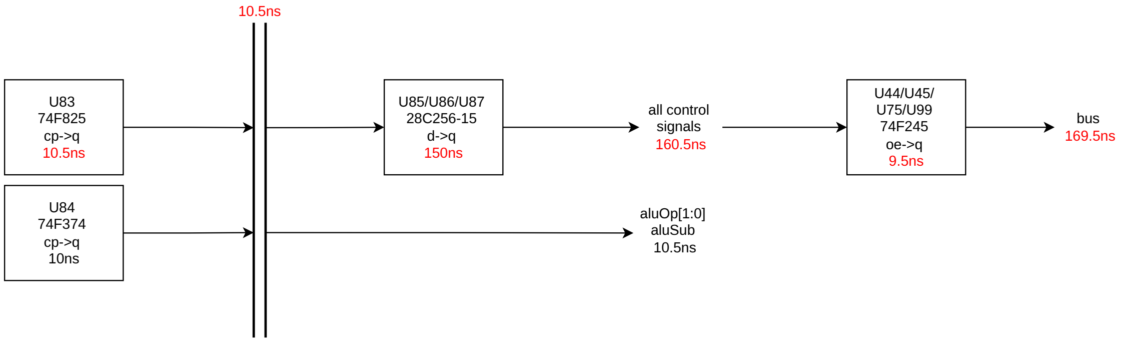


Figure 5.9: Timing analysis for the control signals.

The ALU latency path (figure B.2) is more complex which is mainly due to the ripple carry adder. Consequently, the critical path comes from the bus, through all the carry flags to the final adder result. After the result multiplexer the longest path is from the zero flag to the ALU flag register U97 (313.9 ns).

Theoretically, it is possible for one instruction to read a value from the SRAM and using it in the same instruction as an input to the ALU. This would replace the baseline delay of the bus input in figure B.2 (169.5 ns) with 281.3 ns and, therefore, enlarge the total worst case latency to

$$281.3\,\mathrm{ns} - 169.5\,\mathrm{ns} + 313.9\,\mathrm{ns} = 425.7\,\mathrm{ns} \tag{5.3}$$

Notwithstanding, because the EDiC is a multicycle CPU it is easily possible to assign two cycles to all ALU operations where the B operand is read from the memory. With this trick, the overall critical path is the maximum of 313.9 ns and ${}^{425.7\,\mathrm{ns}}/_{2}$ which is 313.9 ns. With a safety margin of 30% it is feasible to choose an oscillator with a frequency of 2.4 MHz:

$$2.4\,\mathrm{MHz} \leq \frac{1}{1.3 \cdot 313.9\,\mathrm{ns}} = 2.45\,\mathrm{MHz} \tag{5.4}$$

When dealing with circuits that are designed to run right at the critical path or with very long trace lengths another factor needs to be taken into account which is neglected here. Each signal has a latency in traces and when the clock arrives at the second register earlier than at the first register, the time allowed for the combinatorial path is shorter than the theoretical clock period. However, for the EDiC this is not a problem as shown in section 6.2.2.

6 Initial Hardware Test & Component Verification

Even though the netlist was simulated and tested in the FPGA implementation, there is no guarantee that the hardware will work out of the box. Therefore, all bits of all components are to be tested individually to make sure there are no wiring problems which would result in bugs which are hard to pinpoint and debug. Testing single ICs, especially with tri-state logic, is significantly easier when incrementally adding ICs to the PCB. That way one driver of a tri-state net can be verified and when adding another driver to the net, problems can be pinpointed to the new IC because the first one was known to work correctly. Therefore, all ICs are placed inside sockets. This way all resistors, LEDs and sockets can be soldered to the PCB at once and all the ICs can be placed in their sockets consecutively. Another reason for using sockets is that it is easier to change an IC if it is faulty or breaks in the future.

6.1 Test Adapter

To facilitate the testing, all signal loop backs can be disrupted to individually test parts of the circuit. These are the control signals including clock and reset, the instruction register inputs and the ALU results + flags are interrupted by connectors at the side of the PCB which can connect to test adapter boards for testing. For the debug signals that are not busses the test adapter has LEDs to display the state of the EDiC and DIP Switches to set each debug signal to a known value. Additionally, the main bus and ram2data bus is connected to a test adapter but not interrupted. The test adapter has LEDs for displaying the state and a bus driver with DIP Switches. An example is shown in figure 6.1 for the ALU result. If not testing, the connectors can be bridged with shorting connectors which short the pins from the left to the right column (except the two tri-state busses).

Testing the individual components becomes very simple this way. For example the instruction registers:

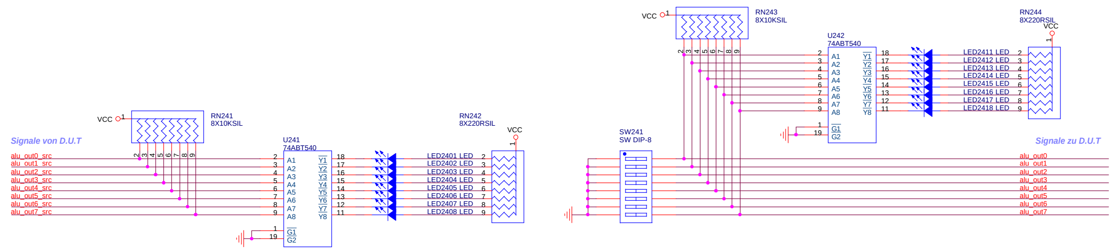


Figure 6.1: An excerpt of the test adapters showing the display and driver for the disruption of the ALU result.

1. Set each bit of the data input individually from the test adapter
2. Assert the `memInstrNWE` control signal (set to 0)
3. Trigger a clock pulse
4. Verify that the output lines equal the input set on the test adapter

All ICs, including the EEPROMs and SRAMs can be directly tested in a couple simple steps this way.

The advantage of individually testing all the bits of all ICs is that in integration testing one can assume that the problem is not with one specific IC.

6.2 Potential Complications

As is normal with large designs, there were some potential problems found in the EDiC which needed fixing.

6.2.1 Shifter - Carry Flag

The detailed testing with the test adapter did reveal one bug which was not revealed in the netlist simulation because it did not occur in any simulated program. The carry flag of a shift operation is determined by the 8 to 1 multiplexer U102 in figure A.7 with the first 3 bits of the bus as select bits. The carry should always be the last bit that was shifted out of the 8 bit word. This way, the input D1 (a0) is set as the new carry flag when shifting by 1 bit, D2 (a1) when shifting by 2 bits and so on. However, the input D7 is connected to a7 and not a6 as it should be. This results in a wrong carry flag when shifting by 7 bits in either direction.

In the netlist simulation this bug was not found because there was no circumstance where a value with differing bits 7 and 8 was shifted by 7 bits and the carry flag being used in the next instruction. This kind of bugs can go undetected for a very long time and are very hard to pinpoint with a fully running CPU.

6.2.2 Clock reflection

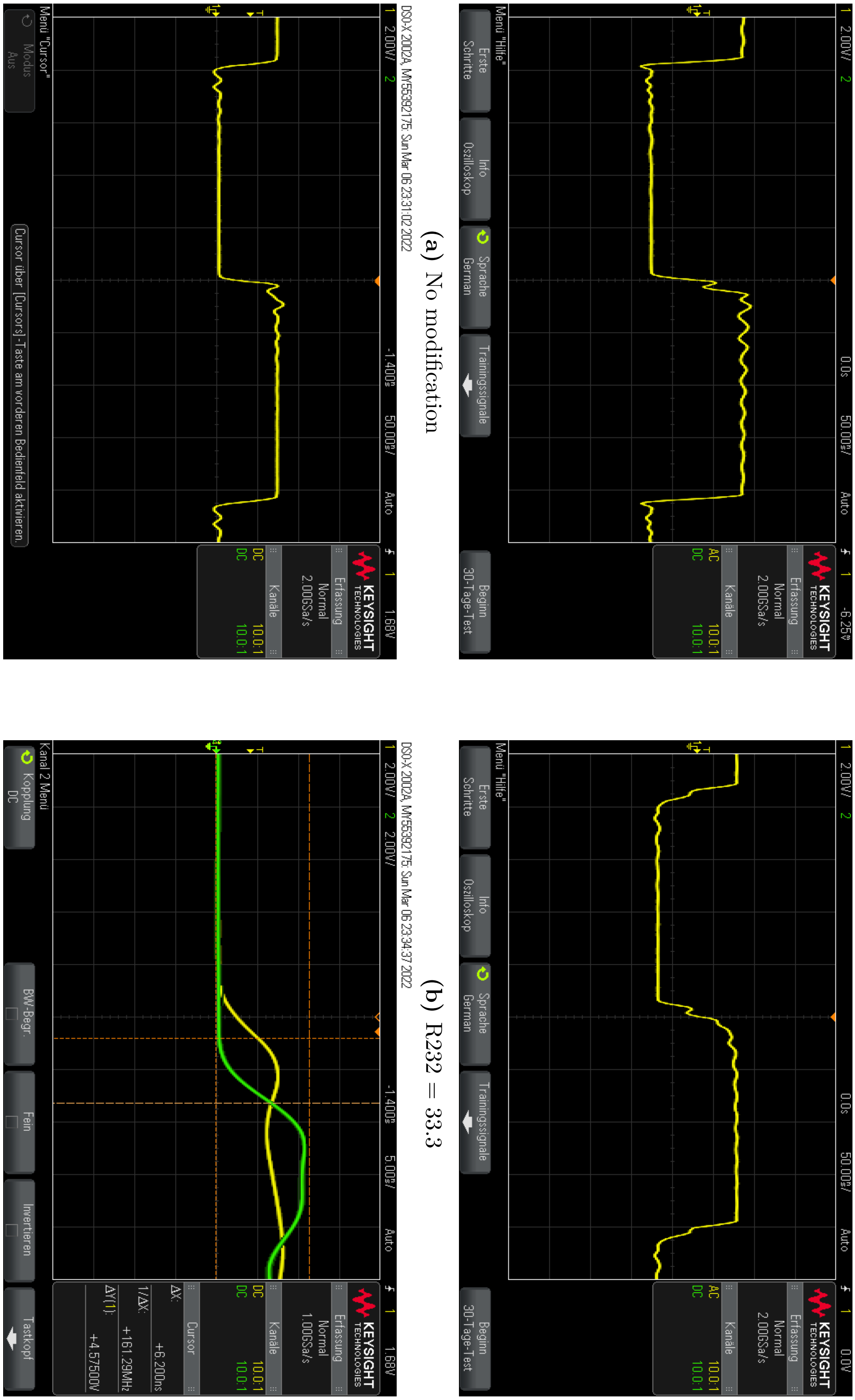
Especially with larger PCBs a good clock distribution is a must-have. Long clock lanes with a large load (i.e. many connected components) may induce several unwanted effects:

- Non-monotony in the rising edge due to reflection from the ends of clock traces.
- A less steep rising edge due to an implicit RC-low pass filter with capacitive loads from the clock inputs and wire resistor.
- Over and undershoot after the edges exceeding the maximum rated voltage.
- Clock latency between ICs reducing the time between two registers.

The effects must all be checked and kept under control. The clock lane for the EDiC was not routed as one continues trace and rather similar to a clock tree split in two. This reduces the maximum distance one clock input is away from the clock source (U95 in figure A.4). In the schematic design, additional clock buffers were implemented to allow further splitting of the clock tree to help with clock distribution.

Without any modifications, the clock looked like shown in figure 6.2a[1]. It can be seen that there is only a little bit of overshoot but at about the middle of the rising edge there is a dip of about 500 mV. This could lead to a double trigger where the rising edge is detected as two individual rising edges in a register and, therefore, a counter could increment by two instead of by one. Therefore, an attempt was made to circumvent this by changing R232 (a resistor in series after the clock buffer) from 0 to a larger value. In figure 6.2b a 33.3 resistor was added. It becomes obvious that the time constant of the implicit low-pass filter increased and with it the edge becomes less steep, but the dip also becomes less of an impact. Even though this is a decent improvement, it is not perfect. The next attempt was to add a line termination of 100 at the end of both clock lines instead of the 33.3 resistor in series. The result can be seen in figure 6.2c. It shows that the dip in the rising edge is no longer there and the edge is also as steep as without any modifications. Even though the overshoot changed a bit, it is by no means a problem and, therefore, this solution looked promising.

[1] figures 6.2a and 6.2b have AC coupling enabled which is why the y scaling is off.



(a) No modification

(b) R232 = 33.3

(c) R232 = 0 and 100 termination

(d) Latency

Figure 6.2: Comparison of the rising edge of the clock in different configurations. Measured close to the clock buffer (yellow) and in figure 6.2d at the end of a clock lane (U204 pin 8) (green).

```
SIMPLE_IO = 0xfe00
mov r0, 0x42
loop:
str r0, [SIMPLE_IO]
b loop
```

(a) First test program.

```
SIMPLE_IO = 0xfe00
UART_SCR = 0xfe0f

ldr r0, [SIMPLE_IO]
str r0, [UART_SCR]
loop:
ldr r1, [UART_SCR]
str r1, [SIMPLE_IO]
b loop
```

(b) Second test program.

```
s:
call function
b s

function:
  add r0, 1
ret
```

(c) Third test program.

Code Example 6.1: Test programs for integration testing.

Figure 6.2d zooms into the rising edge and shows the edge at U204 pin 8 (one end of the clock tree) in green. It can be seen that the edge looks a bit different which may be explained by the different behavior of different probes in the small time scale. Additionally, the latency of the clock signal can be observed which is about 6 ns. The clock frequency was chosen in section 5.3 to have a safety margin of about 30% and, therefore, a latency of 6 ns is not a problem with a clock period of 416.7 ns (2.4 MHz).

6.2.3 Driving Bus High

Until now, no program was programmed into the EEPROMs, and it was time for the first real integration test of the EDiC. The first program used for the integration test in code example 6.1a was a basic test to see if basic instructions get executed and if the built-in I/O works. After it ran successfully and displayed `0x42` at the displays, the second testing program from code example 6.1b included the RS232 I/O extension card and its scratch register (at address `0xfe0f`). When this also ran successfully, a more complex Universal Asynchronous Receiver-Transmitter (UART) echo program (code example C.3) was programmed into the EEPROMs. It finally had problems and did not work as expected. It could be observed that the PC would randomly be set to an unreasonably high value and after that NOPs were executed until the PC overflowed to 0 and the program started again. However, when turning

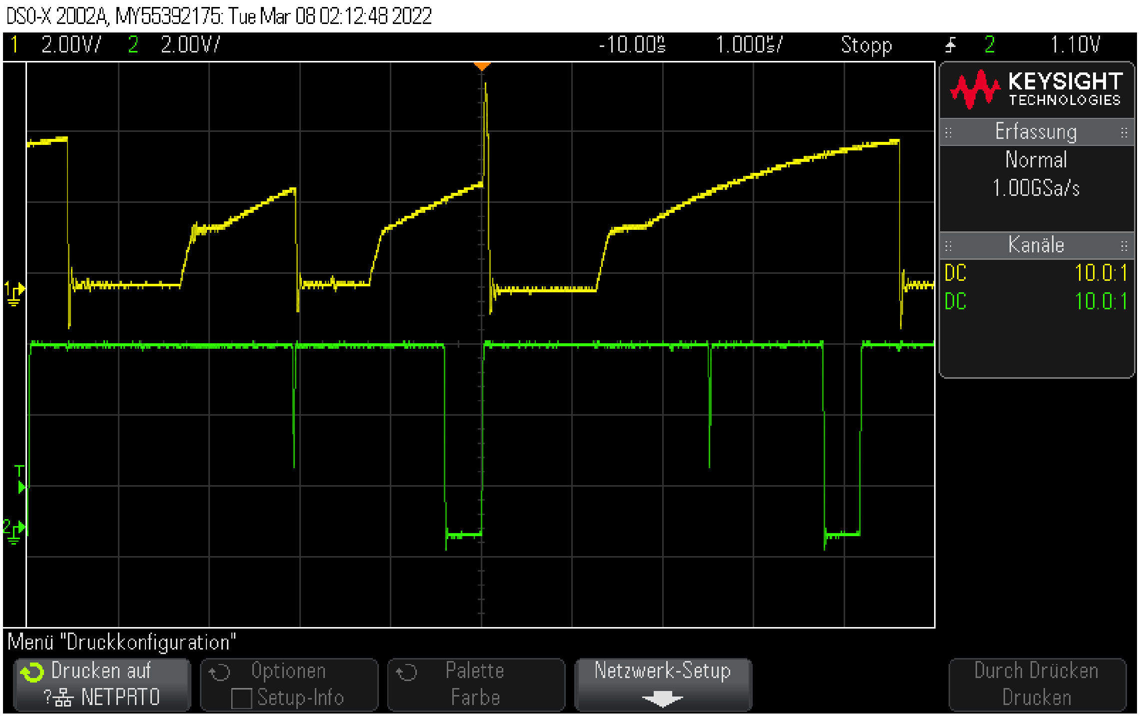


Figure 6.3: Write Enable of MAR register (green) and one bus lane without `0xff` driver (yellow).

on the cycle by cycle debugger and stepping through all cycles, the program worked perfectly and bytes got read correctly from the RS232 extension card and were also sent back correctly. After debugging for a long time, the bug was tracked down to the return instruction which sometimes (about 1 in 100 times) would return to a random instruction and not return to after the call instruction. It was further debugged with the third test program (code example 6.1c). With an oscilloscope it was finally possible to detect the problem which is shown in figure 6.3. It is actually a bug in both, the call and return instruction, which results in the same misbehavior of return to a wrong location: Both instruction load `0xff` into the MAR by not driving the bus with a specific value and relying on the pull-up resistors to pull the lines high. In figure 6.3 two MAR write enable pulses can be seen and, especially, in the first pulse, the problem becomes apparent. If the bus was pulled low in the cycle before the MAR is written, the pull-up resistors take some time to pull the voltage to 5 V which leaves the level at about 2 V at the time of the write pulse. This is right at the required minimum voltage to be detected as a high signal by the *74F825*. Therefore, most of the time, the register detects the bus input as a 1, but sometimes it is detected as a low signal.

The fix is quite easy as soon as the problem is detected: A new bus driver (*74F245*) is added in one of the spare slots which A input is connected to `H1`, the B input to the bus and the output enable signal is connected to a new control signal. This fix would

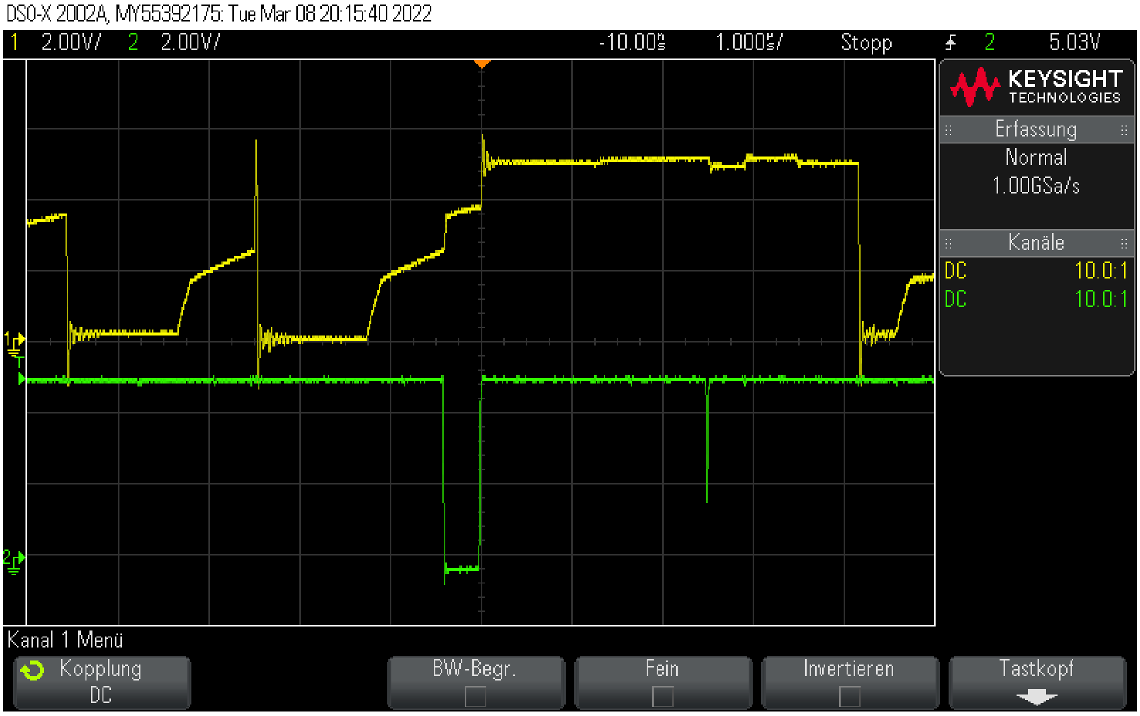


Figure 6.4: Write Enable of MAR register (green) and one bus lane with `0xff` driver (yellow).

have been very difficult if no output of the microcode EEPROMs would have been free to use or if no place for spare ICs was left on the PCB. Therefore, it is always important to design everything with enough resources left.

The result of the fix can be seen in figure 6.4 where the bus line is raised to about 3.8 V as soon as the write-enable pulse starts[2].

6.2.4 UART Transceiver lost data

The final bug was only observed with the test adapter and never while running the EDiC on its own. One of the integration tests was to manually write a value from the test adapter to the scratch register of the UART IC (TL16C550AN) on the RS232 extension card and then read it out repeatedly. It could be observed that the data was read back successfully for a couple of times but often the data would be read back incorrectly after several reads. However, in the automated test with the program from code example 6.1b, the data was still correctly displayed at the built-in I/O after half an hour which results in about 300 million reads without

[2]The even higher level on the bus line after the write-enable pulse is driven by the SRAM driver (outputs the return address in the return instruction) which is an *ACT* type for compliance with the SRAM specification. In figure 6.3 the SRAM probably drove a '0' to the bus line.

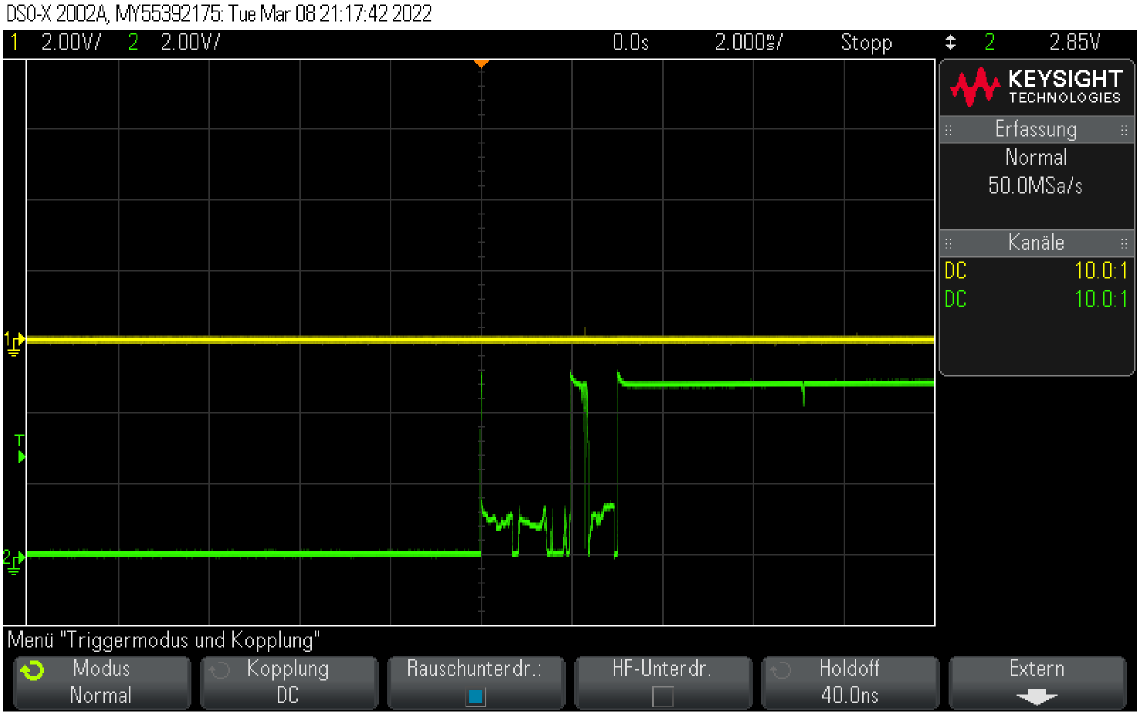


Figure 6.5: DIP Switch output on switching (yellow).

errors[3]:

$$\frac{2.4\,\text{MHz}}{14} \cdot 1800\,\text{s} \approx 309 \cdot 10^6 \tag{6.1}$$

The only notable difference between the two tests is that in the manual test, the output enable signal to the UART IC was set by hand with a DIP Switch and in the automated test, it was controlled by the output of the EEPROM. When looking at the waveforms of the signal coming from the DIP switch, a bouncing of the trigger can sometimes be observed as shown in figure 6.5. In theory this should not have an effect on the content of the scratch register of the UART IC as the glitch happens on the output enable input of the IC. However, the datasheet explicitly states a minimum time for a read strobe pulse duration of 80 ns. For this reason, the test adapter was altered to include a low pass filter and Schmitt trigger on the `memRamNOE` and `memRamNWE` control signals because those are the only control signals which are used asynchronously. All other control signal are only used in components which do not state a minimum pulse duration for it (only setup and hold times in relation to the rising edge of the clock). After implementing this fix, the manual test worked perfectly for many read cycles which means that the minimum read pulse duration for the UART IC needs to be respected.

Even though, the problem never occurred with the control signals coming from

[3] 2.4 MHz, 14 cycles per loop iteration and 1800 seconds run time

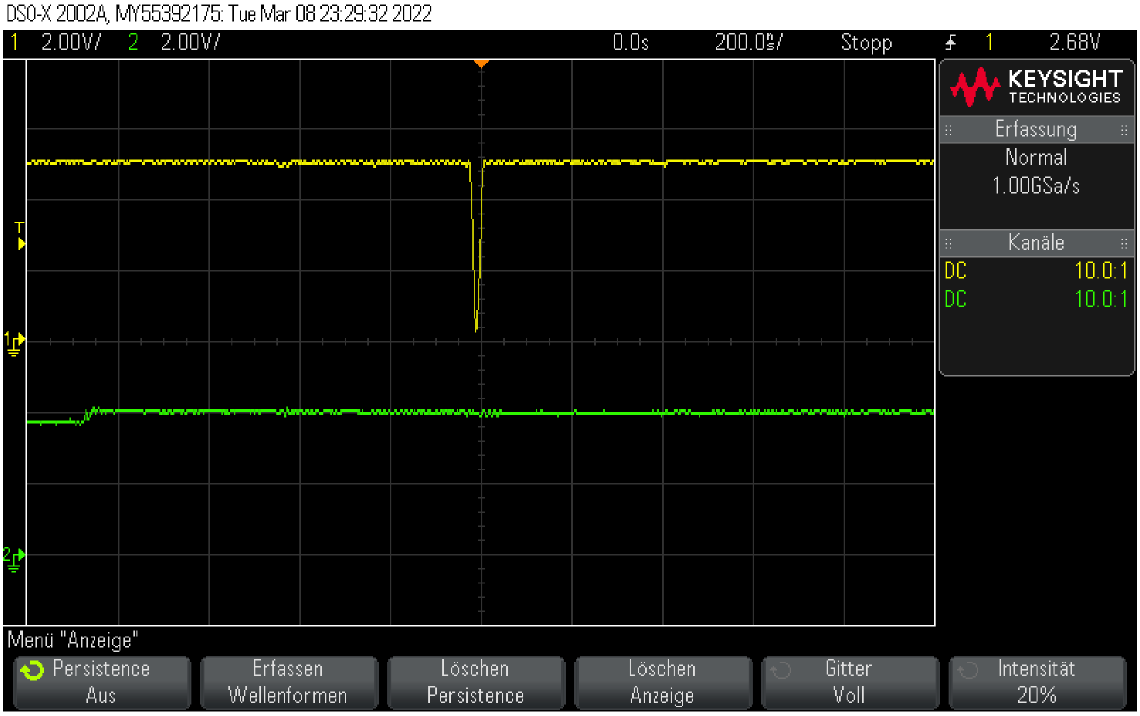


Figure 6.6: Output of the `memRamNOE` control signal from the EEPROM (yellow).

the microcode EEPROM, we looked into the specifications for the EEPROM and found it also has a period after the address inputs change where the data output is undefined. When observing the output with the oscilloscope, it was observed that there sometimes are glitches on the `memRamNOE` control signal as shown in figure 6.6. Therefore, it was decided to add a register to the `memRamNOE` and `memRamNWE` control signals. This results in the microcode needing adjustment to assert these two signals one cycle earlier which was easy to implement and prevented the problem completely.

7 Conclusion and Future Work

This thesis presented the challenges of and solutions for designing and building a model CPU for educational purposes. It was shown that a simple and yet powerful CPU can be developed using the easy-to-understand TTL ICs of the 74 family. The most suitable architecture for this purpose has to be modularized and, therefore, a multicycle and single-bus oriented architecture was chosen. By adding a more complex address logic for the memory, it was possible to extend the address space from 8 to 16 bits while maintaining a simpler 8 bit data bus. The address logic also enabled versatile memory mapped I/O for arbitrary extension cards as shown with the RS-232 UART extension card used in the demonstration in figure 1.1. Moreover, with a custom stack implementation the EDiC is also able to implement function calls and function local variables.

One major contribution to ease the educational use of the EDiC is the comprehensive software development environment which supports the custom EDiC Instruction Set Architecture (ISA). It consists of an assembler which is able to translate advanced human-readable assembler code to the 24 bit instructions used by the EDiC. Features such as value and string constants, file imports and label definitions support the programmer in creating software for the EDiC. Additionally, a tool to generate microcode for the EDiC is provided. Creating the memory contents for the microcode EEPROMs of the EDiC is a task that is very time-consuming and error-prone if done manually. Therefore, a tool is provided which reads a human-readable file which describes all the instructions and what control signals are to be asserted in which cycle of the instruction. A second tool converts the file to the memory contents for the EEPROMs.

For quicker design iterations and also for verification of the finalized design, two FPGA implementations were created in the process. The first implementation is a behavioral implementation which was used to efficiently make alterations to the architecture in the design phase. With the behavioral implementation, it was easy to design the CPU on the logical level before diving into the details and mapping the logic onto discrete logic ICs. The second FPGA implementation was used as a verification of the hardware schematic after it was created. A specifically written software converted the netlist that was exported from the schematic tool to a HDL description of the exact schematic. All the logic ICs were implemented as individual modules and an FPGA design which is logically equivalent to the final hardware build could be simulated. With the help of a small adapter from I/O of the FPGA evaluation board to the extension board connector, it was possible to also test the extension card before ordering the large PCB for the hardware build of the EDiC.

When designing the final hardware build, the extensive simulation and testing of the FPGA design simplified the required verification. Additionally, a comprehensive timing analysis was performed to choose the correct clock frequency for the best performance while still ensuring that timing bugs do not occur at any time. The hardware build includes custom designed test adapters which ease the initial hardware tests enormously. With the test adapters and by using sockets for all logic ICs it was possible to incrementally test the PCB and discover some possible problems or bugs which slipped through the extensive logical simulation.

7.1 Future Work

Even though the EDiC is a complex and yet simple to understand model CPU with an extensive development environment, there are some more ideas that would enhance the EDiC as a whole.

Power Supply At the moment, the EDiC is supplied by an industry standard 5V power supply. However, with a custom-built CPU made of discrete logic ICs it would only be appropriate to also power the EDiC with its own custom power supply. Therefore, it is planned to design and build a power supply before the EDiC will be presented at the Vintage Computing Festival Berlin (VCFB). [31]

Extension Cards The EDiC currently only has the RS-232 UART extension card. This is one of the most versatile extensions as it can be connected to a lot of different devices and thus providing a communication interface to the outside world via a standardized serial protocol. However, a lot more possible extensions could be used such as an extension card to provide a persistent storage or the capabilities of the ALU could be enhanced by providing a multiply extension or other computational hardware in the form of extension cards.

High Level Language Compiler The most complex software which currently exists for the EDiC is the snake program. However, writing more complex software in assembler is of course possible but becomes increasingly hard and takes a lot of programming effort. Therefore, a possible addition is to implement a compiler which could translate a high level programming language like C to EDiC assembler or machine code. With modular compiler infrastructure as it is provided by the LLVM it may be possible to create a compiler backend for the EDiC.

Acronyms

Notation	Description
ALU	Arithmetic Logic Unit v–vii, 9–11, 13–15, 19–22, 24, 29–32, 35, 47, 49, 64–66, 71–74, 84, 92, 93, 95, 117, 118, 120
ASIC	Application-Specific Integrated Circuit 45
BRAM	Block RAM 45
CE	chip enable 64, 65
CISC	Complex Instruction Set Computer i, ii, 8, 9, 13
CLB	Configurable Logic Block 45
CMOS	Complementary metal-oxide-semiconductor 3
CPU	Central Processing Unit i–iii, 1, 3, 4, 6–11, 13, 15, 16, 18, 27, 30, 31, 45, 49, 55, 57, 61, 72, 75, 83, 84
CSON	CoffeeScript-Object-Notation 27
DIP	dual in-line package 3, 4
DSP	Digital Signal Processor 45, 46
EDiC	Educational Digital Computer i–iii, v, vii, 1–4, 6–11, 13–20, 22, 27, 30–33, 37–40, 42, 43, 45, 46, 49, 59–62, 65–67, 69, 71–75, 77, 79, 83–85, 91, 93, 95, 96, 101, 102, 104, 106, 108, 110, 112, 114, 123, 132
EDIF	Electronic Design Interchange Format 51, 52, 95
EEPROM	Electrically Erasable Programmable Read-Only Memory vi, 8, 14, 27, 30–32, 52–54, 56, 62, 64, 67, 71, 74, 77, 79–81, 83, 91, 92, 95
EPROM	Erasable Programmable Read-Only Memory 45

Notation	**Description**
FPGA	Field Programmable Gate Array i, ii, vi, 1, 4, 6, 45–50, 52–59, 73, 83, 84, 91
HDL	Hardware Description Language 46, 47, 84
I/O	Input / Output i, vi, 3, 7, 15–18, 20, 22, 47, 55, 57, 65, 83, 84, 91
IC	Integrated Circuit i, ii, 1–5, 7, 16, 51–53, 55, 59–62, 64, 66–69, 71, 73–75, 79, 80, 83, 84, 91, 93, 95
IDE	Integrated Development Environment 42
ISA	Instruction Set Architecture i, ii, 83
JSON	JavaScript Object Notation 27
LED	Light-Emitting Diode 61–63, 66, 67, 73, 91
LSB	least significant bit 14
LUT	Lookup Table 45, 46, 91
MAR	Memory Address Register 17, 20–25, 36, 63, 64, 78, 79, 92
MSB	most significant bit 11, 13, 41, 66
MUX	Multiplexer 45
NOP	No Operation 25, 77
PC	Program Counter v, 8, 9, 13, 14, 16, 18, 19, 25, 29, 37, 38, 52, 61–64, 77, 91
PCB	Printed Circuit Board 4, 6, 18, 45, 49, 52, 55, 57, 66, 69, 73, 75, 79, 84
PRNG	Pseudo Random Number Generator 33, 34, 39, 95, 96, 131
RAM	Random-Access Memory vii, 20, 54, 57, 117, 118, 120
RISC	Reduced Instruction Set Computer 8
ROM	Read-Only Memory 7, 54, 56, 91
RTL	Register-transistor logic i, ii, 3, 6

Notation	**Description**
SMD	surface-mounted device 3
SP	Stack Pointer 16, 17, 19, 23–25, 36, 37, 61, 64
SRAM	Static Random-Access Memory 7, 8, 15, 16, 20, 45, 46, 54, 62, 64, 67, 71, 72, 74, 79
TTL	Transistor-transistor logic i, ii, v, 3–5, 7, 83, 91
TUB	Technical University Berlin 47
UART	Universal Asynchronous Receiver-Transmitter vi, 77, 79, 80, 83, 84, 96, 132
VCFB	Vintage Computing Festival Berlin 84
VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language 46, 47

List of Figures

List of Tables

List of Code Examples

Bibliography

[1] Manfred Albach. *Grundlagen der Elektrotechnik 1 Erfahrungssätze, Bauelemente, Gleichstromschaltungen*. German. Pearson Deutschland, 2011, p. 352. ISBN: 9783868940794. URL: `https://elibrary.pearson.de/book/99.150005/9783863266042`.

[2] Manfred Albach. *Grundlagen der Elektrotechnik 2 Periodische und nicht periodische Signalformen*. Pearson Deutschland, 2011, p. 304. ISBN: 9783868940800. URL: `https://elibrary.pearson.de/book/99.150005/9783863266059`.

[3] Mahmoud Ali. *Visual Studio Code - One Light Theme based on Atom*. 2022. URL: `https://marketplace.visualstudio.com/items?itemName=akamud.vscode-theme-onelight`.

[4] Altera. *User-Configurable Logic Databook*. (WWW). 1988. URL: `http://www.bitsavers.org/components/altera/_dataBooks/1988_Altera_Data_Book.pdf`.

[5] bevry. *CoffeeScript-Object-Notation*. 2022. URL: `https://github.com/bevry/cson`.

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

[7] B. Jack Copeland. “The Modern History of Computing”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2020. Metaphysics Research Lab, Stanford University, 2020.

[8] 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`.

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

[10] arm Developer. *ARM Developer Suite Assembler Guide - Conditional execution*. URL: `https://developer.arm.com/documentation/dui0068/b/ARM-Instruction-Reference/Conditional-execution`.

[11] Digilent. *Nexys A7 Reference*. 2022. URL: https://digilent.com/reference/programmable-logic/nexys-a7/start.

[12] Digilent. *Nexys A7 Reference - Common Anode Circuit Node*. 2022. URL: https://digilent.com/reference/_detail/reference/programmable-logic/nexys-a7/n4t.png?id=programmable-logic%3Anexys-a7%3Areference-manual.

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

[14] Alliance Memory Inc. *512K X 8 BIT LOW POWER CMOS SRAM*. 2009. URL: https://www.mouser.de/datasheet/2/12/AS6C4008-1265427.pdf.

[15] Texas Instruments. *CDx4ACT245 Octal-Bus Transceiver, Three-State, Non-Inverting*. 1998. URL: https://www.ti.com/lit/ds/symlink/cd74act245.pdf.

[16] Texas Instruments. *SNx4F245 Octal Bus Transceivers With 3-State Outputs*. 1987. URL: https://www.ti.com/lit/ds/sdfs010a/sdfs010a.pdf.

[17] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture*. 2016. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf.

[18] ECMA International. *The JSON Data Interchange Syntax*. 2017. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf.

[19] William Irwin. *THE DIFFERENTIAL ANALYSER EXPLAINED*. 2009. URL: http://amg.nzfmm.co.nz/differential_analyser_explained.html.

[20] MacroMates Ltd. *Language Grammars — TextMate 1.x Manual*. 2021. URL: https://macromates.com/manual/en/language_grammars.

[21] Microsoft. *TypeScript: JavaScript With Syntax For Types*. 2022. URL: https://www.typescriptlang.org/.

[22] Microsoft. *Visual Studio Code - Code Editing. Redefined*. 2022. URL: https://code.visualstudio.com/.

[23] MovGP0. *TTL NAND with a "totem-pole" output stage, one of four in 7400*. 2006. URL: https://en.wikipedia.org/wiki/File:7400_Circuit.svg.

[24] 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.

[25] Nolanjshettle. *File:Edge triggered D flip flop.svg*. 2013. URL: https://en.wikipedia.org/wiki/File:Edge_triggered_D_flip_flop.svg.

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

[27] Niklas Schelten. *EDiC support for Visual Studio Code*. 2022. URL: https://github.com/Nik-Sch/EDiC-vscode-syntax.

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

[29] Stunts1990. *File:Edge triggered D flip flop with set and reset.svg*. 2020. URL: https://en.wikipedia.org/wiki/File:Edge_triggered_D_flip_flop_with_set_and_reset.svg.

[30] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (Jan. 1937), pp. 230–265. ISSN: 0024-6115. DOI: 10.1112/plms/s2-42.1.230. eprint: https://academic.oup.com/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf. URL: https://doi.org/10.1112/plms/s2-42.1.230.

[31] VCFB. *Vintage Computing Festival Berlin (VCFB)*. VCFB. 2022. URL: https://vcfb.de/2022/.

[32] B. Winston and ProQuest (Firm). *Media Technology and Society: A History : from the Telegraph to the Internet*. Communication studies. Routledge, 1998. ISBN: 9780415142304. URL: https://books.google.de/books?id=gfeCXlElJTwC.

[33] Andrew Wylie. *The first monolithic integrated circuits*. 2013. URL: https://web.archive.org/web/20180504074623/http://homepages.nildram.co.uk/~wylie/ICs/monolith.htm.

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

A Full Schematics of the EDiC

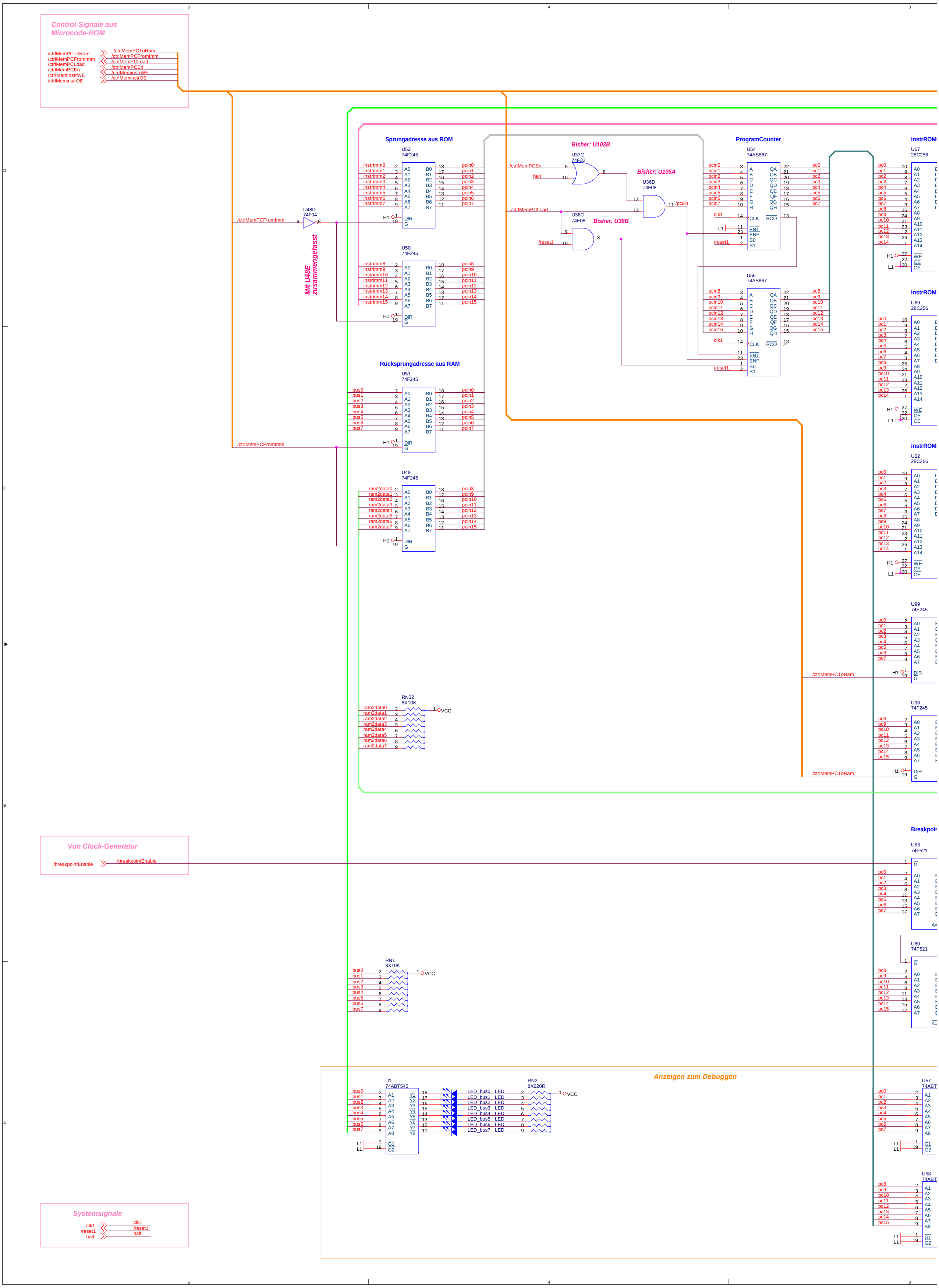


Figure A.1: Schematic: Program Counter / Instruction ROM.


Instruction Register
Konstante zu Bus
Datenbus
Immediate
Instruction
Von / zu RAM 2
Zu Clock-Generator
Bedienelemente zum Debuggen
Test
TIGRIS Elektronik GmbH
Teltowkanalstr.2
12247 Berlin
Tel: +49 (30) 7688 083-0
Web: http://www.tigris.eu
Mail: info@tigris.eu
Title
EDIC
Program Counter / Instruction ROM
Size A1
Document Number
TIGRIS Artikel-Nummer: ED101-1
Rev see title
Date: Tuesday, January 11, 2022
Sheet 2 of 10

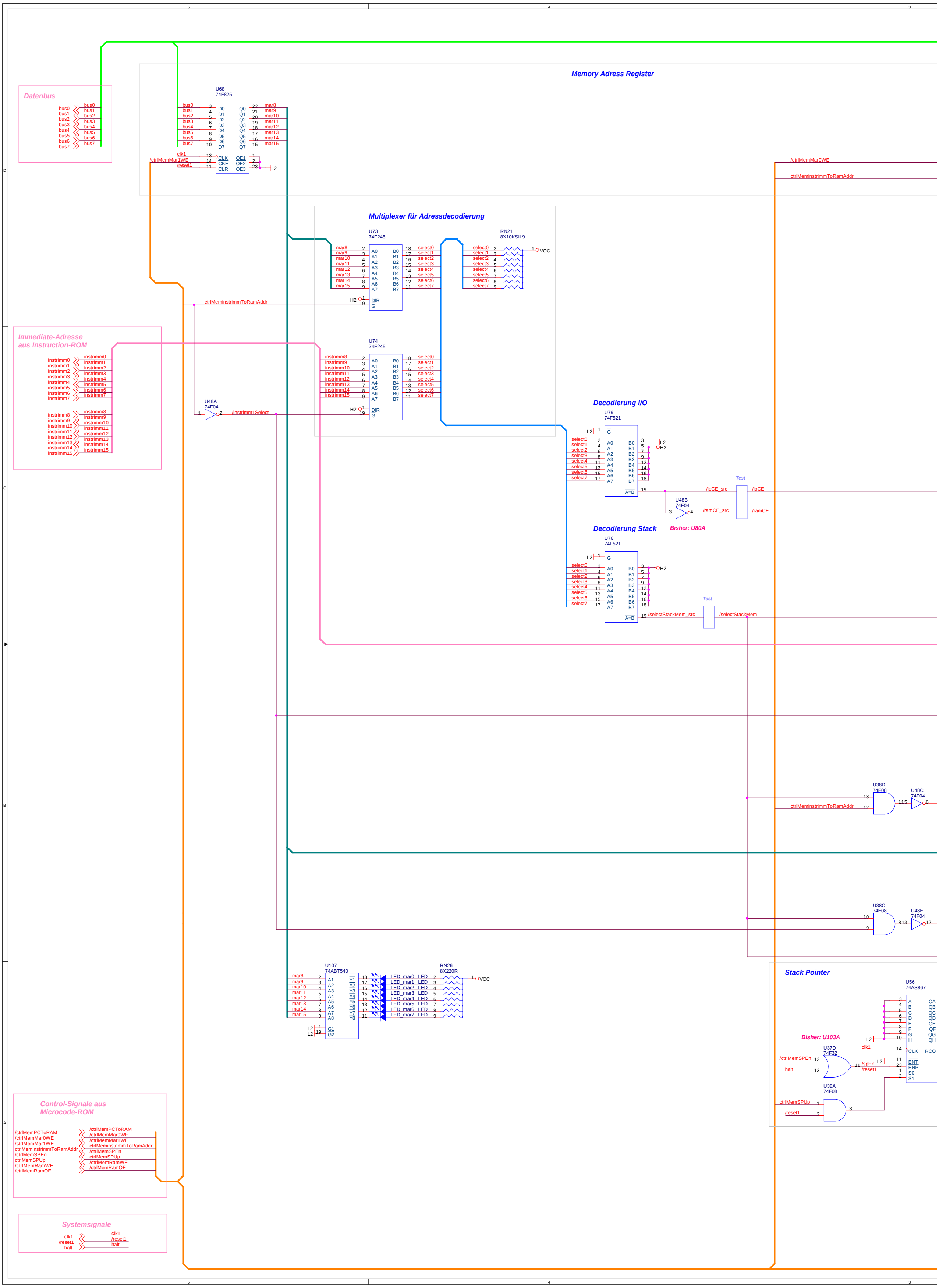
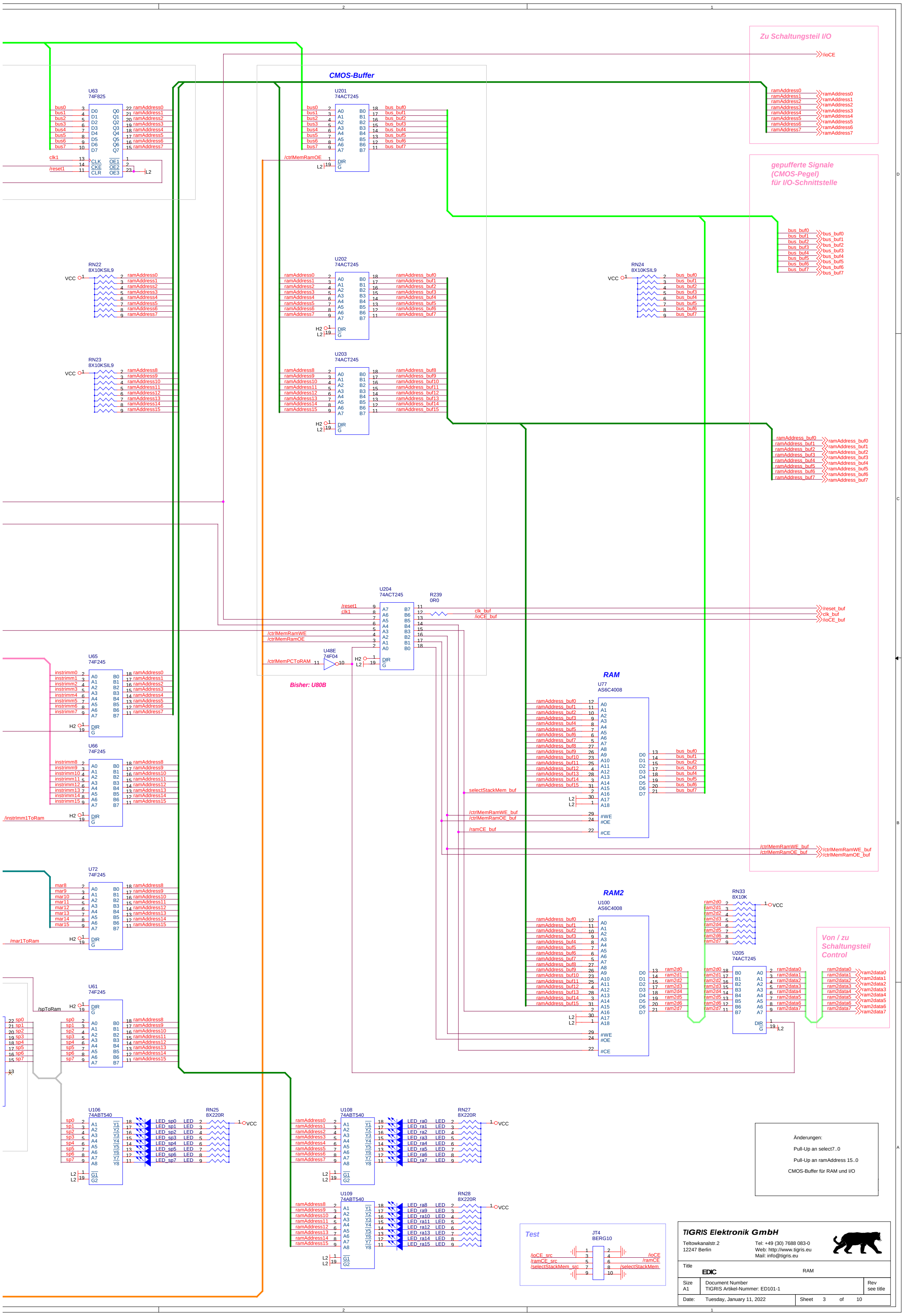


Figure A.2: Schematic: RAM.


Zu Schaltungsteil I/O
CMOS-Buffer
gepufferte Signale
(CMOS-Pegel)
für I/O-Schnittstelle
U63
74F825
U201
74ACT245
U202
74ACT245
U203
74ACT245
U204
74ACT245
RN22
8X10KSIL9
RN23
8X10KSIL9
RN24
8X10KSIL9
U48E
74F04
Bisher: U80B
U65
74F245
U66
74F245
U72
74F245
U61
74F245
RAM
U77
AS6C4008
RAM2
U100
AS6C4008
RN33
8X10K
U205
74ACT245
Von / zu
Schaltungsteil
Control
U106
74ABT540
U108
74ABT540
U109
74ABT540
RN25
8X220R
RN27
8X220R
RN28
8X220R
Test
JT4
BERG10
Änderungen:
Pull-Up an select7..0
Pull-Up an ramAddress 15..0
CMOS-Buffer für RAM und I/O
TIGRIS Elektronik GmbH
Teltowkanalstr.2
12247 Berlin
Tel: +49 (30) 7688 083-0
Web: http://www.tigris.eu
Mail: info@tigris.eu
Title
EDIC
RAM
Size
A1
Document Number
TIGRIS Artikel-Nummer: ED101-1
Rev
see title
Date: Tuesday, January 11, 2022
Sheet 3 of 10

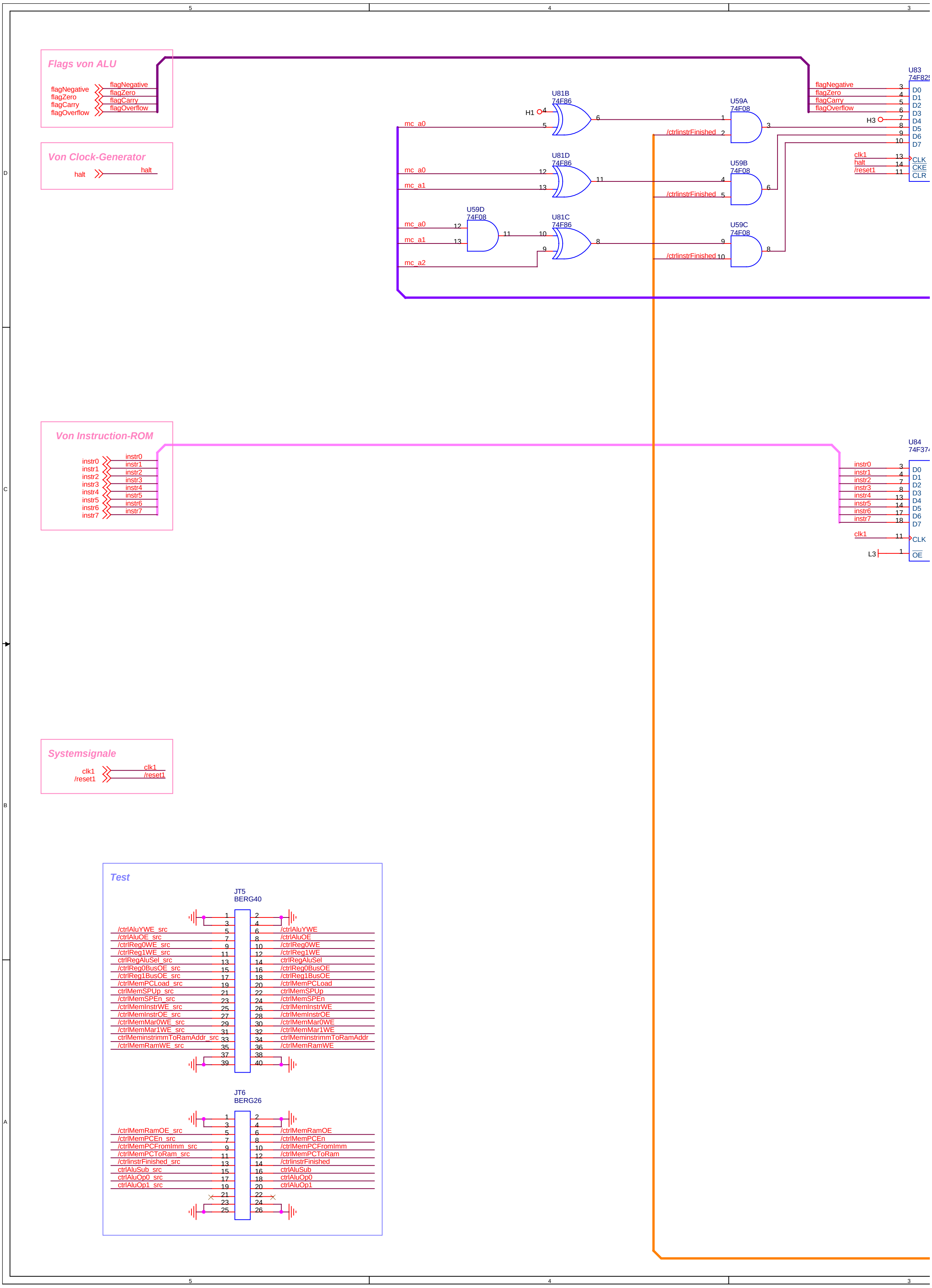
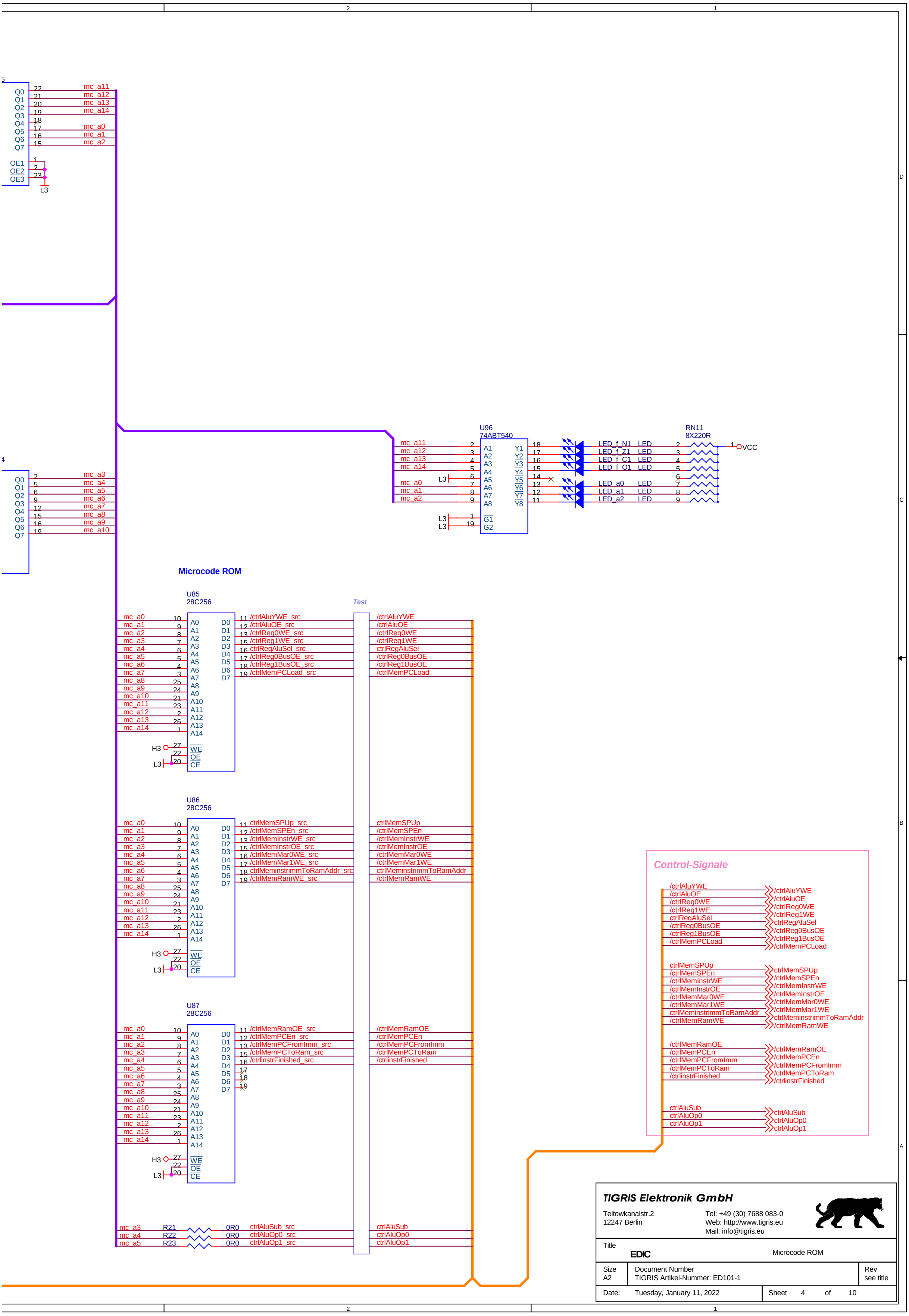


Figure A.3: Schematic: Microcode.


Microcode ROM
U85
28C256
U86
28C256
U87
28C256
U96
74ABT540
RN11
8X220R
VCC
Test
Control-Signale
TIGRIS Elektronik GmbH
Teltowkanalstr.2
12247 Berlin
Tel: +49 (30) 7688 083-0
Web: http://www.tigris.eu
Mail: info@tigris.eu
Title
EDiC
Microcode ROM
Size
A2
Document Number
TIGRIS Artikel-Nummer: ED101-1
Rev
see title
Date: Tuesday, January 11, 2022
Sheet 4 of 10

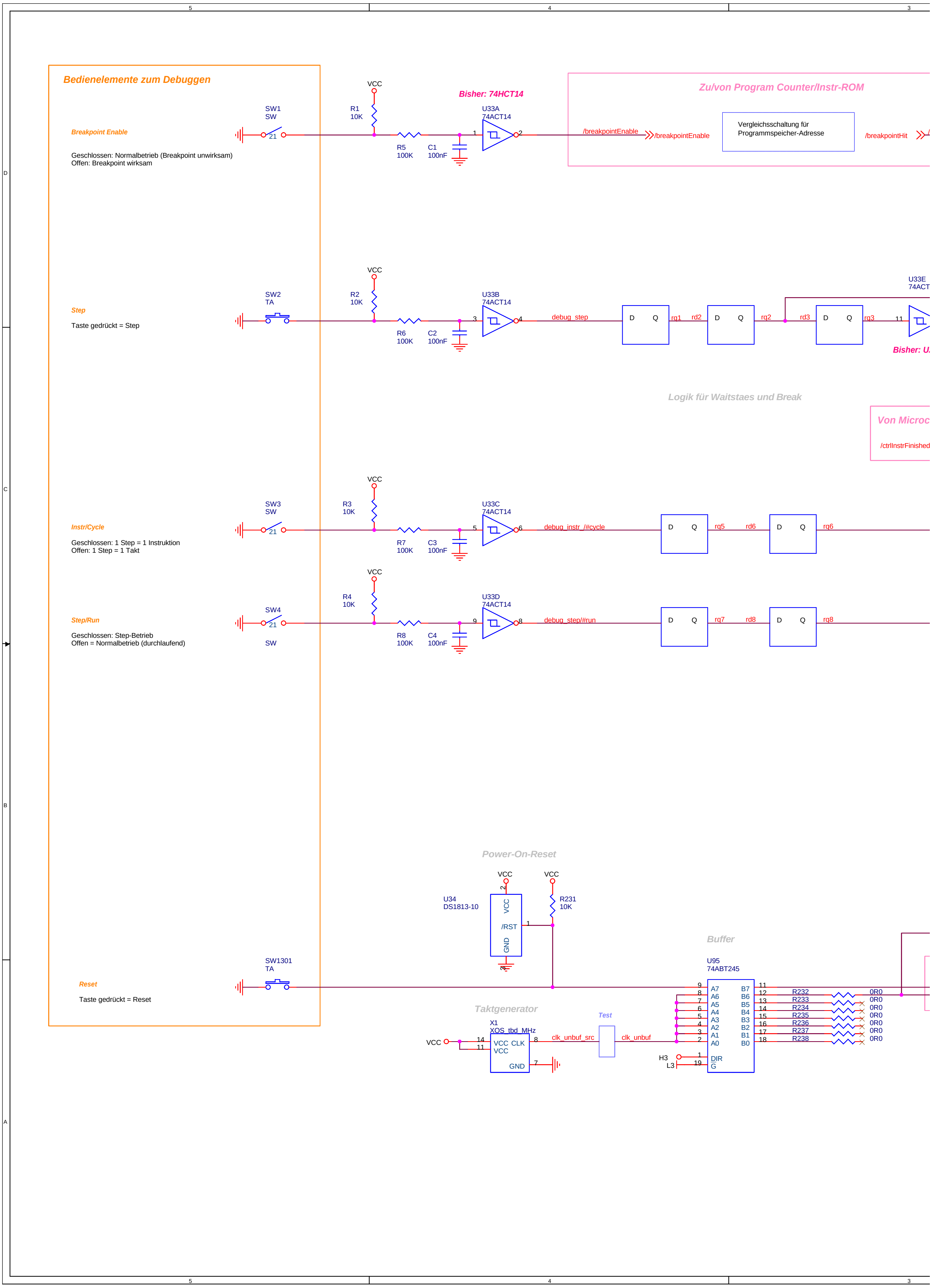
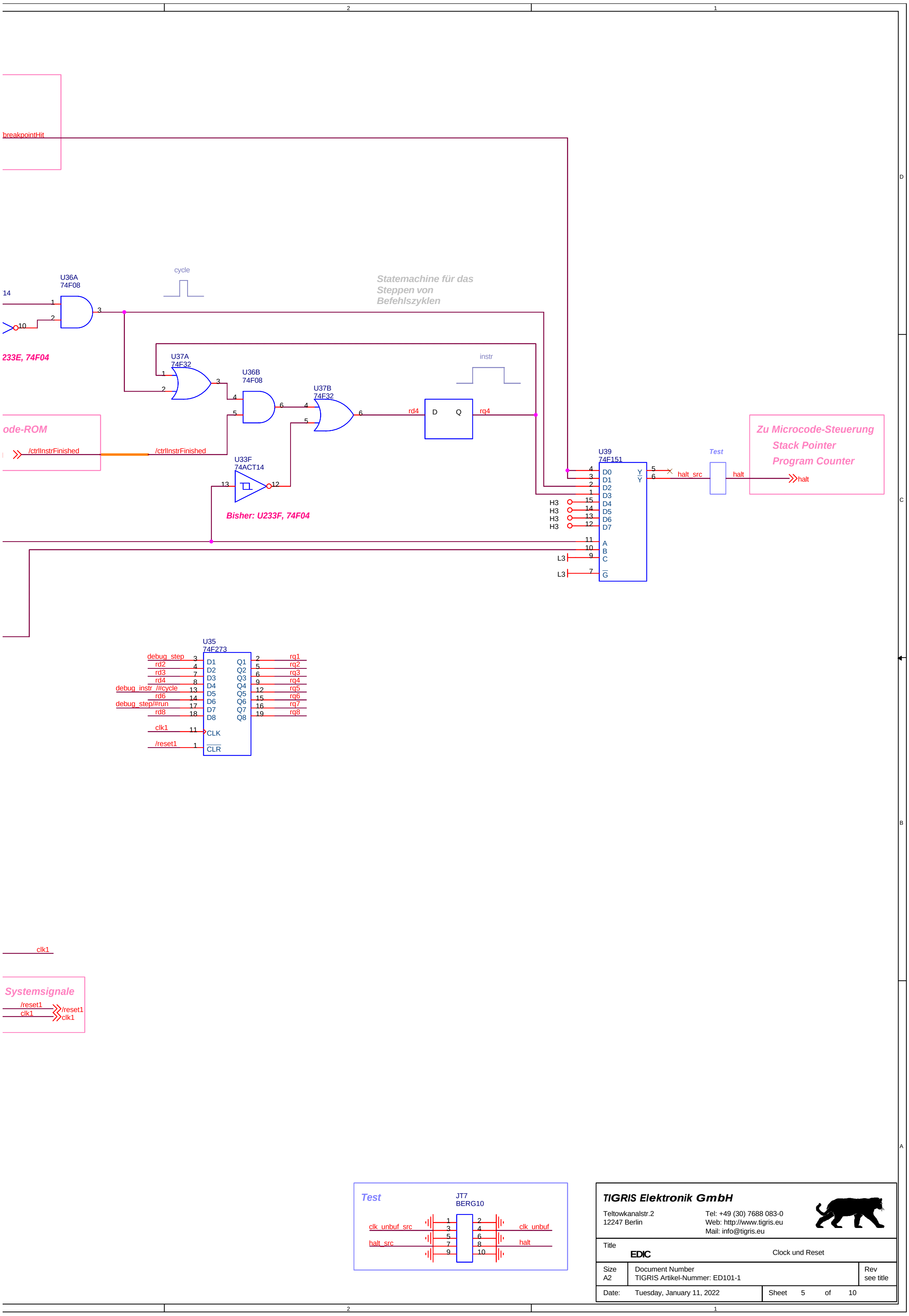


Figure A.4: Schematic: Clock and Reset.


breakpointHit
U36A
74F08
cycle
Statemachine für das
Steppen von
Befehlszyklen
233E, 74F04
U37A
74F32
U36B
74F08
U37B
74F32
instr
rd4
D
Q
rq4
ode-ROM
/ctrlInstrFinished
/ctrlInstrFinished
U33F
74ACT14
Bisher: U233F, 74F04
U39
74F151
halt_src
Test
halt
halt
Zu Microcode-Steuerung
Stack Pointer
Program Counter
U35
74F273
debug_step
debug_instr_/#cycle
debug_step/#run
clk1
/reset1
CLK
CLR
clk1
Systemsignale
/reset1
clk1
Test
JT7
BERG10
clk_unbuf_src
halt_src
clk_unbuf
halt
TIGRIS Elektronik GmbH
Teltowkanalstr.2
12247 Berlin
Tel: +49 (30) 7688 083-0
Web: http://www.tigris.eu
Mail: info@tigris.eu
Title
EDIC
Clock und Reset
Size
A2
Document Number
TIGRIS Artikel-Nummer: ED101-1
Rev
see title
Date: Tuesday, January 11, 2022
Sheet 5 of 10

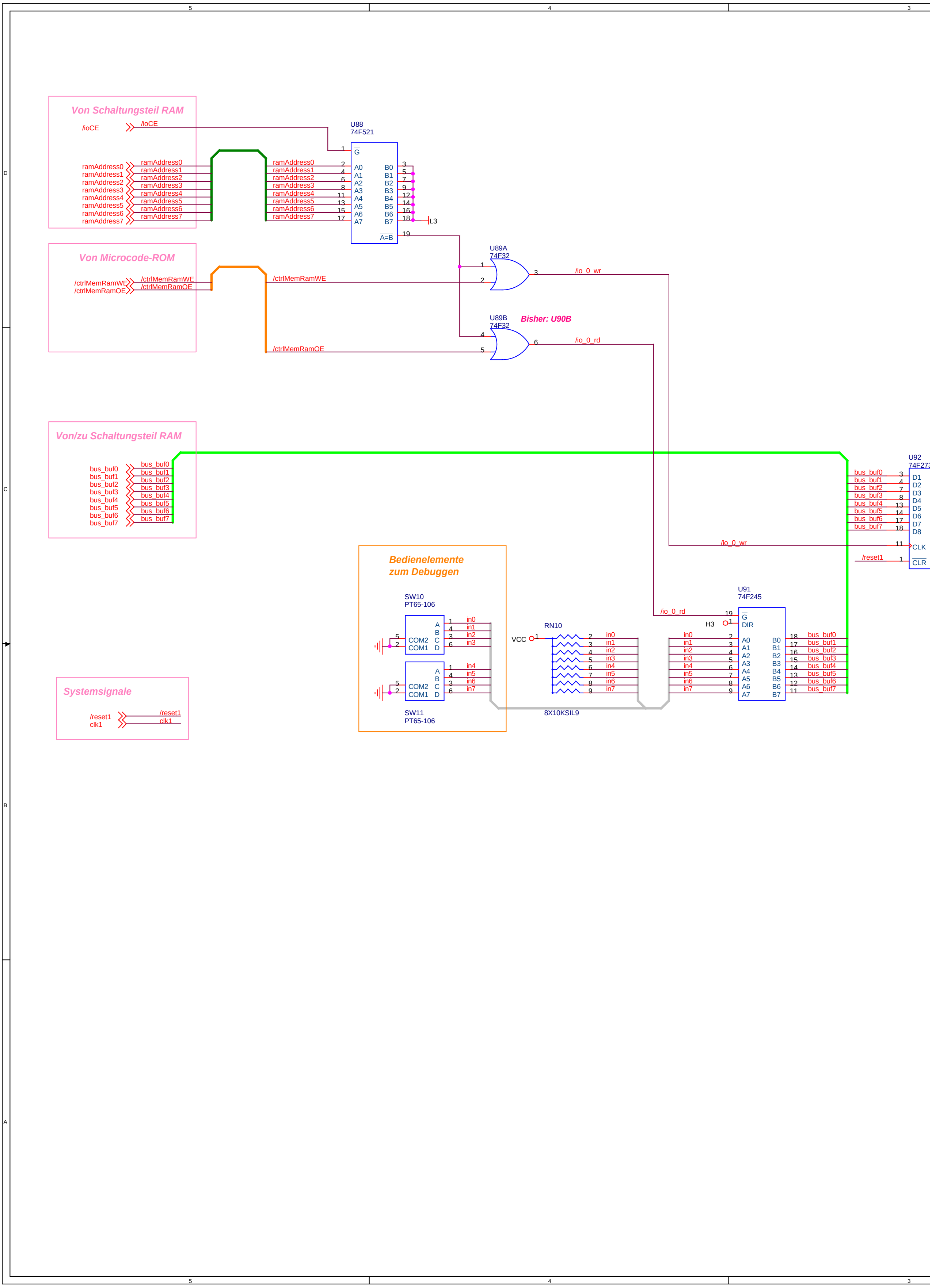
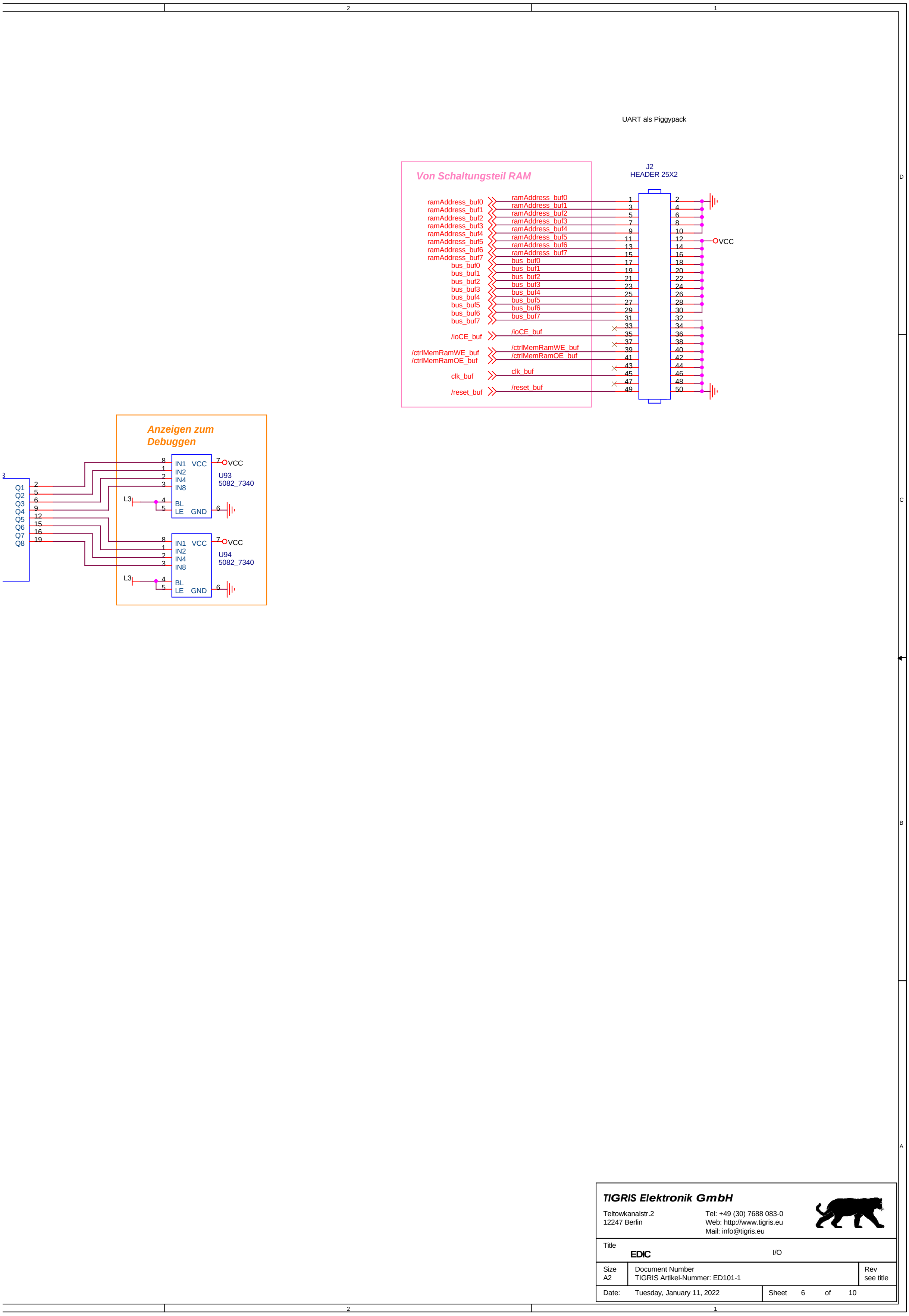


Figure A.5: Schematic: Built-In I/O.



UART als Piggypack
Von Schaltungsteil RAM
J2
HEADER 25X2
ramAddress_buf0
ramAddress_buf1
ramAddress_buf2
ramAddress_buf3
ramAddress_buf4
ramAddress_buf5
ramAddress_buf6
ramAddress_buf7
bus_buf0
bus_buf1
bus_buf2
bus_buf3
bus_buf4
bus_buf5
bus_buf6
bus_buf7
/ioCE_buf
/ctrlMemRamWE_buf
/ctrlMemRamOE_buf
clk_buf
/reset_buf
VCC
Anzeigen zum Debuggen
U93
5082_7340
U94
5082_7340
IN1 IN2 IN4 IN8 BL LE VCC GND
Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8
L3
TIGRIS Elektronik GmbH
Teltowkanalstr.2
12247 Berlin
Tel: +49 (30) 7688 083-0
Web: http://www.tigris.eu
Mail: info@tigris.eu
Title
EDIC
I/O
Size
A2
Document Number
TIGRIS Artikel-Nummer: ED101-1
Rev
see title
Date: Tuesday, January 11, 2022
Sheet 6 of 10

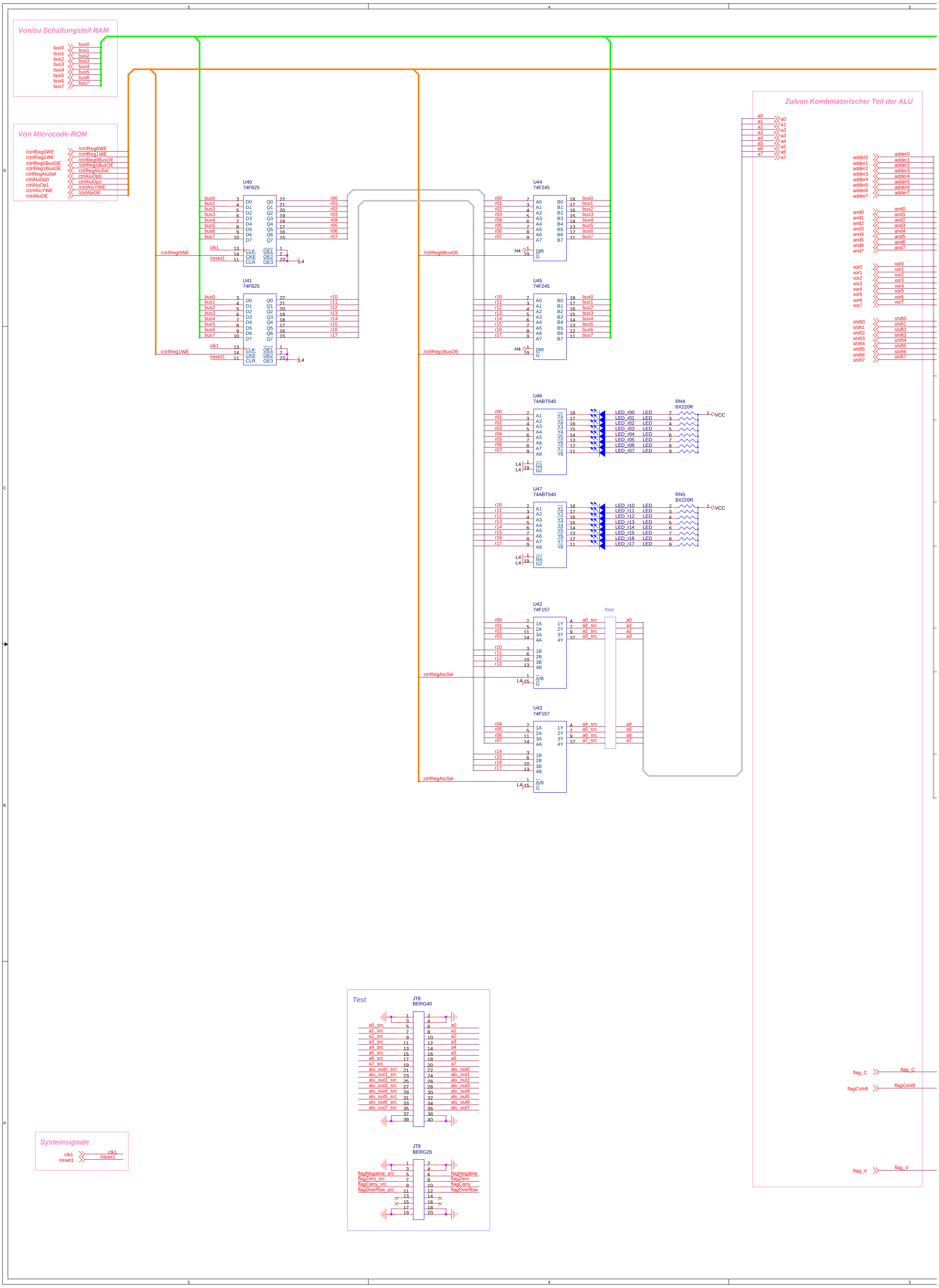
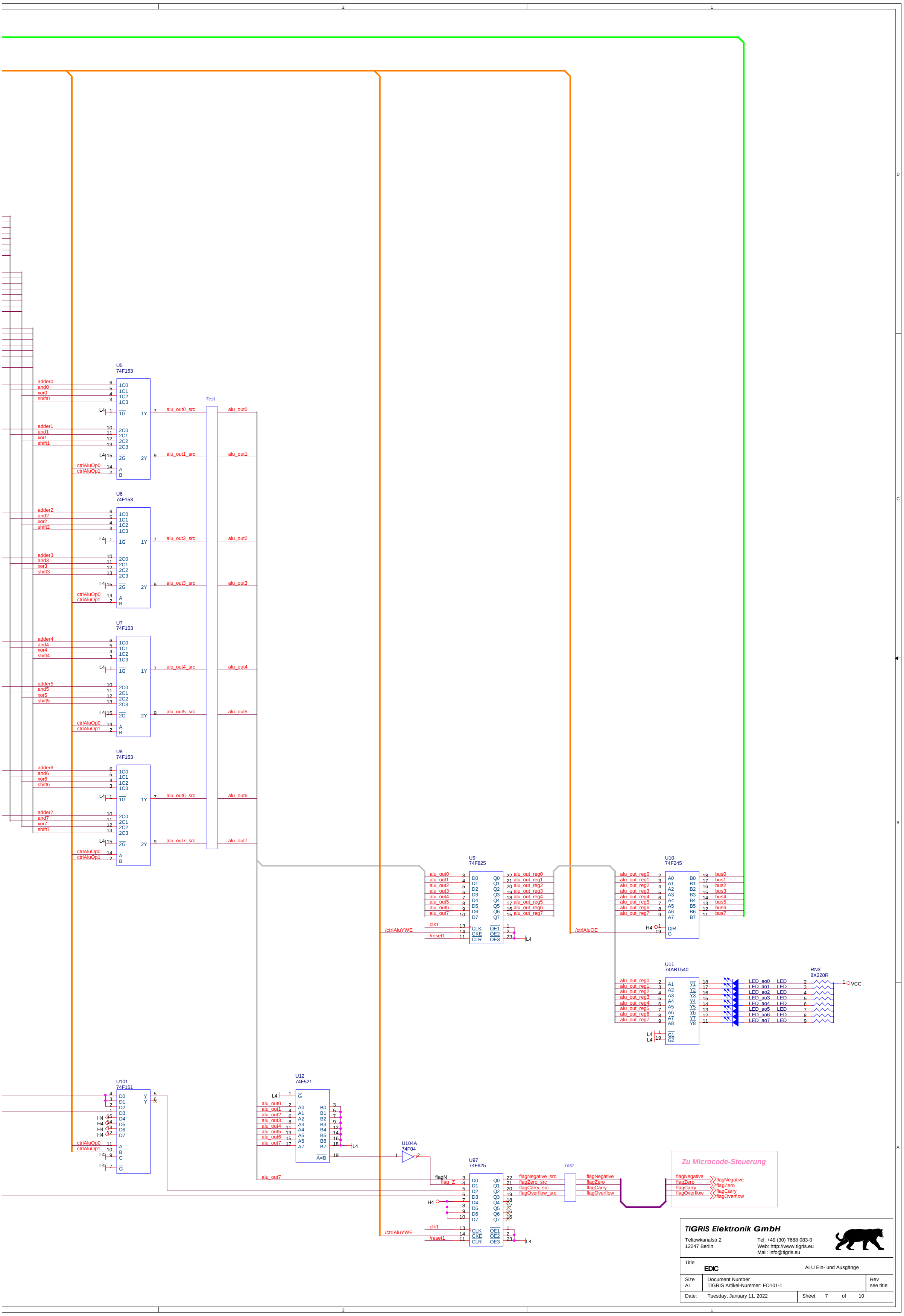


Figure A.6: Schematic: Register Set + ALU output.


Zu Microcode-Steuerung
TIGRIS Elektronik GmbH
Teltowkanalstr.2
12247 Berlin
Tel: +49 (30) 7688 083-0
Web: http://www.tigris.eu
Mail: info@tigris.eu
Title
EDIC
ALU Ein- und Ausgänge
Size
A1
Document Number
TIGRIS Artikel-Nummer: ED101-1
Rev
see title
Date: Tuesday, January 11, 2022
Sheet 7 of 10

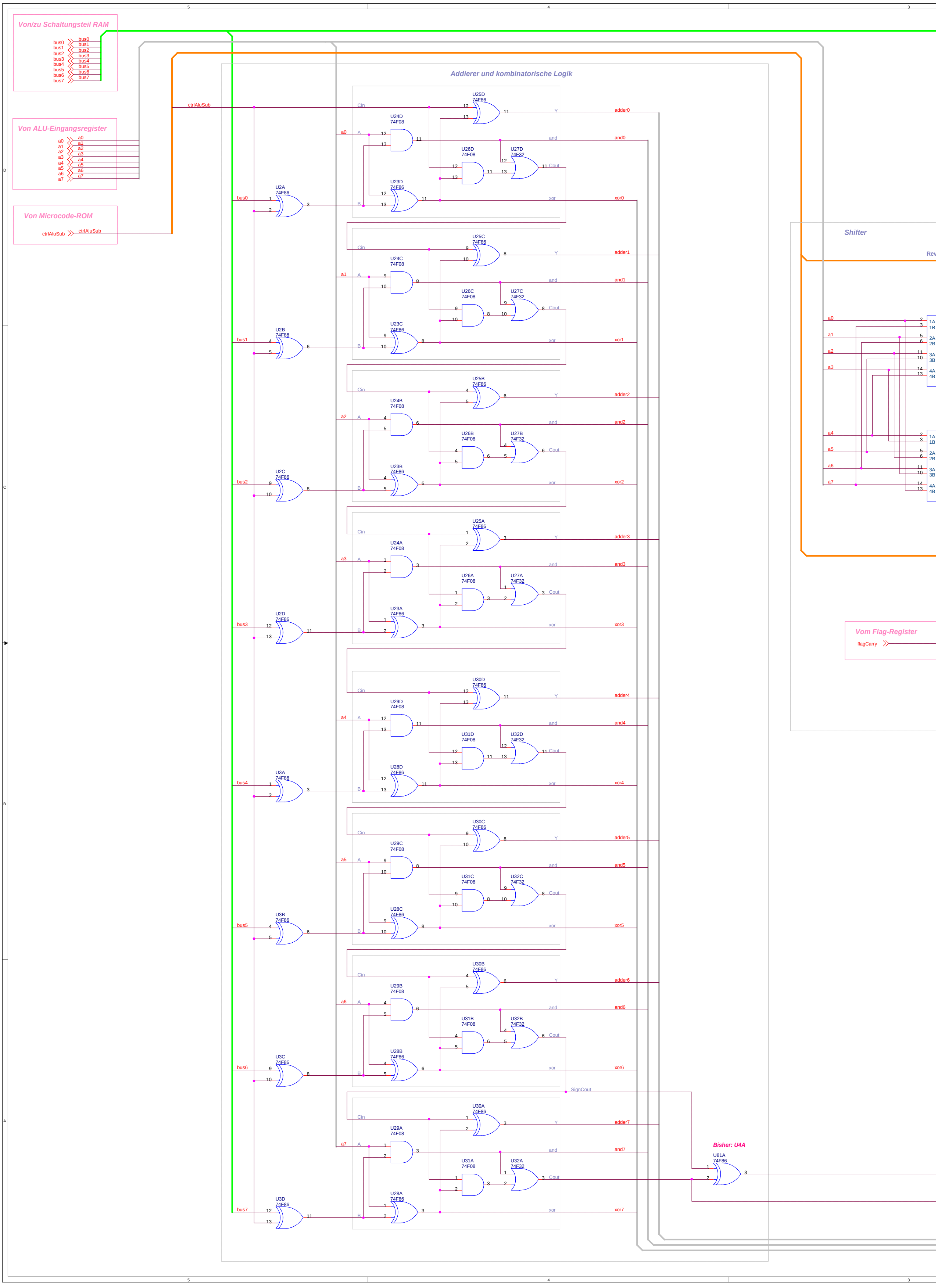
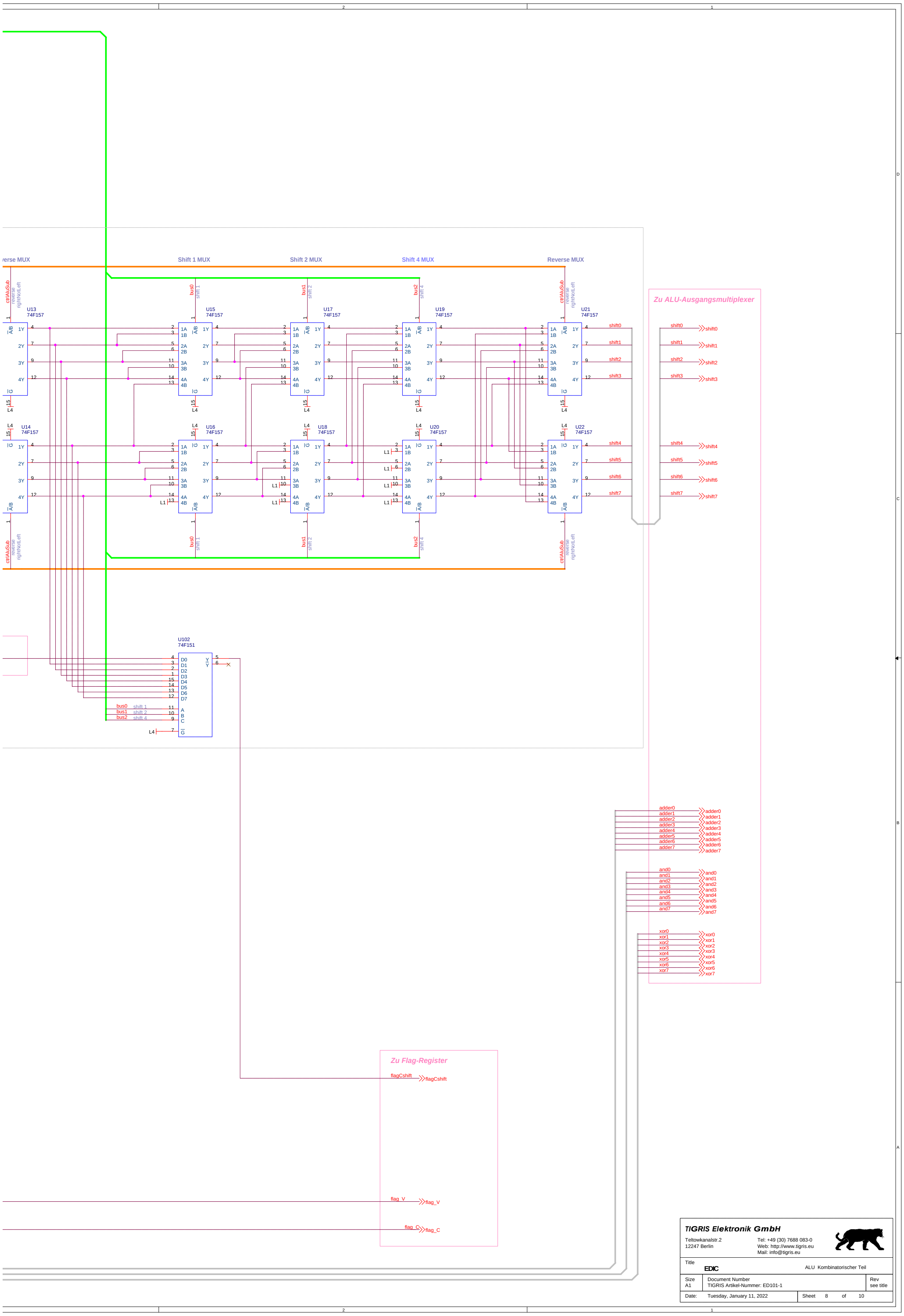


Figure A.7: Schematic: combinatorial ALU.


Shift 1 MUX
Shift 2 MUX
Shift 4 MUX
Reverse MUX
U13 74F157
U14 74F157
U15 74F157
U16 74F157
U17 74F157
U18 74F157
U19 74F157
U20 74F157
U21 74F157
U22 74F157
U102 74F151
Zu ALU-Ausgangsmultiplexer
Zu Flag-Register
flagCshift
flag_V
flag_C
TIGRIS Elektronik GmbH
Teltowkanalstr.2
12247 Berlin
Tel: +49 (30) 7688 083-0
Web: http://www.tigris.eu
Mail: info@tigris.eu
Title
ALU_Kombinatorischer Teil
Size
A1
Document Number
TIGRIS Artikel-Nummer: ED101-1
Rev
see title
Date: Tuesday, January 11, 2022
Sheet 8 of 10

B Timing analysis of the RAM and ALU

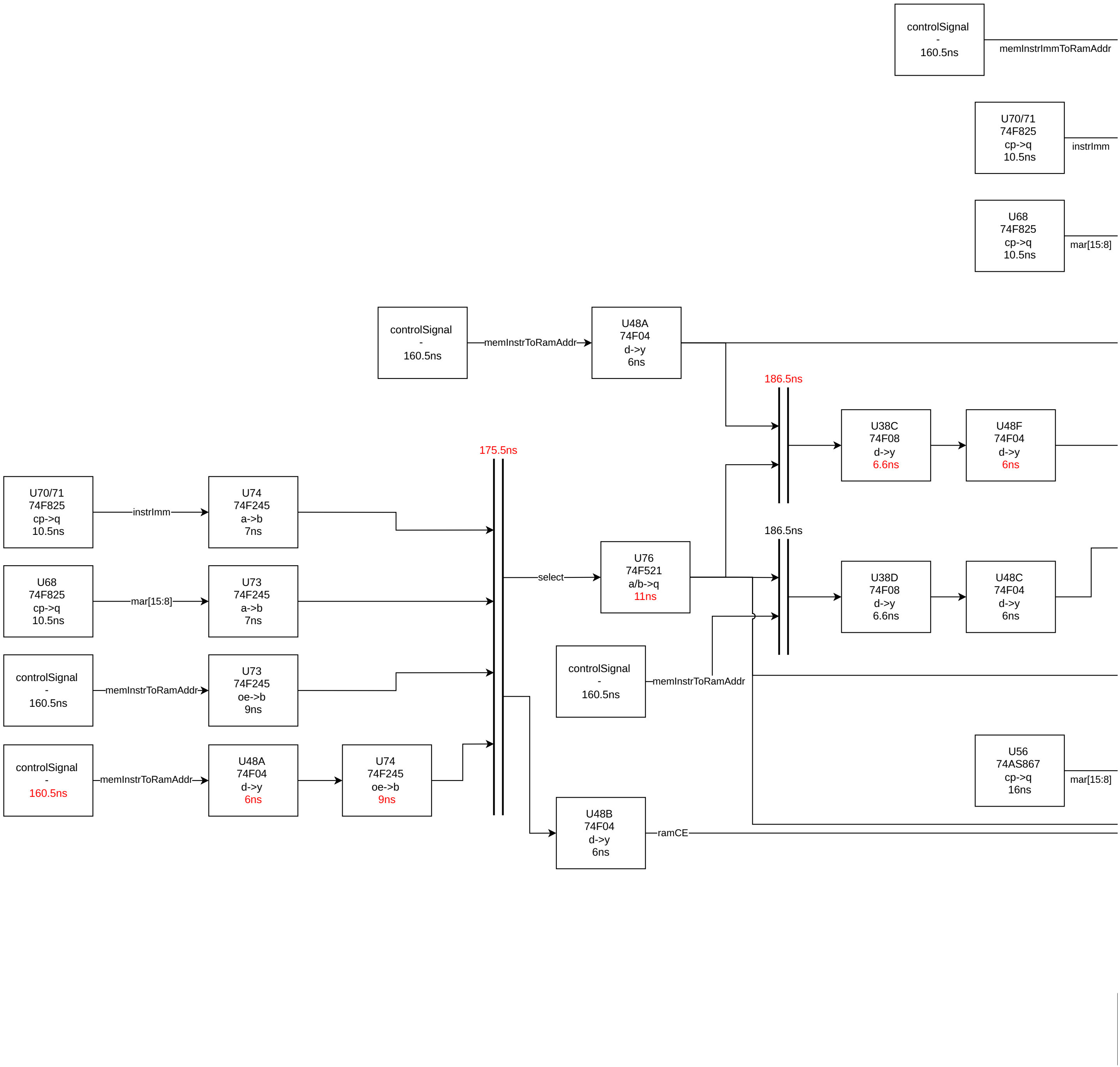
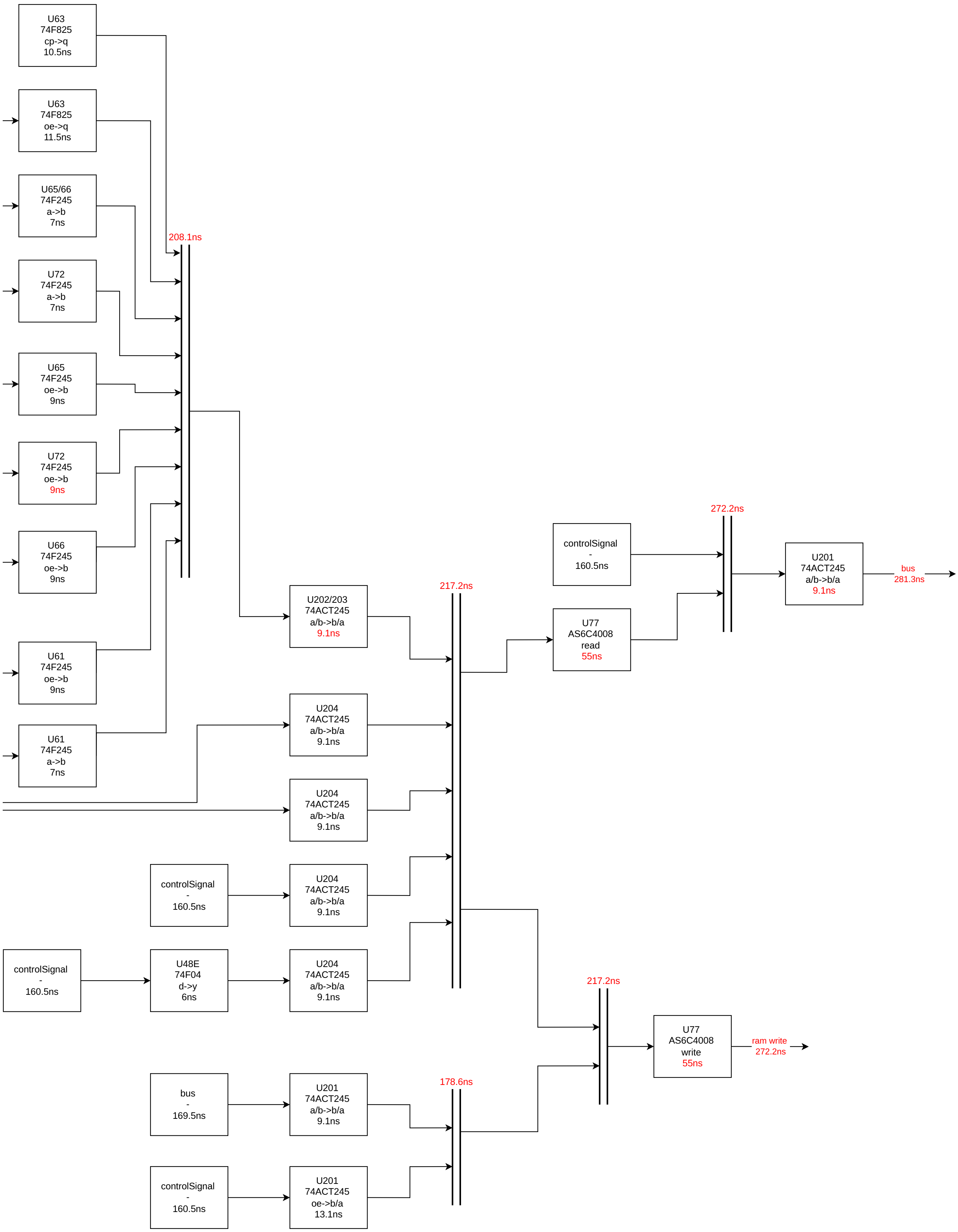


Figure B.1: Timing analysis for the memory latency.


U63
74F825
cp->q
10.5ns
U63
74F825
oe->q
11.5ns
U65/66
74F245
a->b
7ns
U72
74F245
a->b
7ns
U65
74F245
oe->b
9ns
U72
74F245
oe->b
9ns
U66
74F245
oe->b
9ns
U61
74F245
oe->b
9ns
U61
74F245
a->b
7ns
208.1ns
U202/203
74ACT245
a/b->b/a
9.1ns
U204
74ACT245
a/b->b/a
9.1ns
U204
74ACT245
a/b->b/a
9.1ns
controlSignal
-
160.5ns
U204
74ACT245
a/b->b/a
9.1ns
controlSignal
-
160.5ns
U48E
74F04
d->y
6ns
U204
74ACT245
a/b->b/a
9.1ns
217.2ns
controlSignal
-
160.5ns
U77
AS6C4008
read
55ns
272.2ns
U201
74ACT245
a/b->b/a
9.1ns
bus
281.3ns
217.2ns
U77
AS6C4008
write
55ns
ram write
272.2ns
bus
-
169.5ns
U201
74ACT245
a/b->b/a
9.1ns
178.6ns
controlSignal
-
160.5ns
U201
74ACT245
oe->b/a
13.1ns

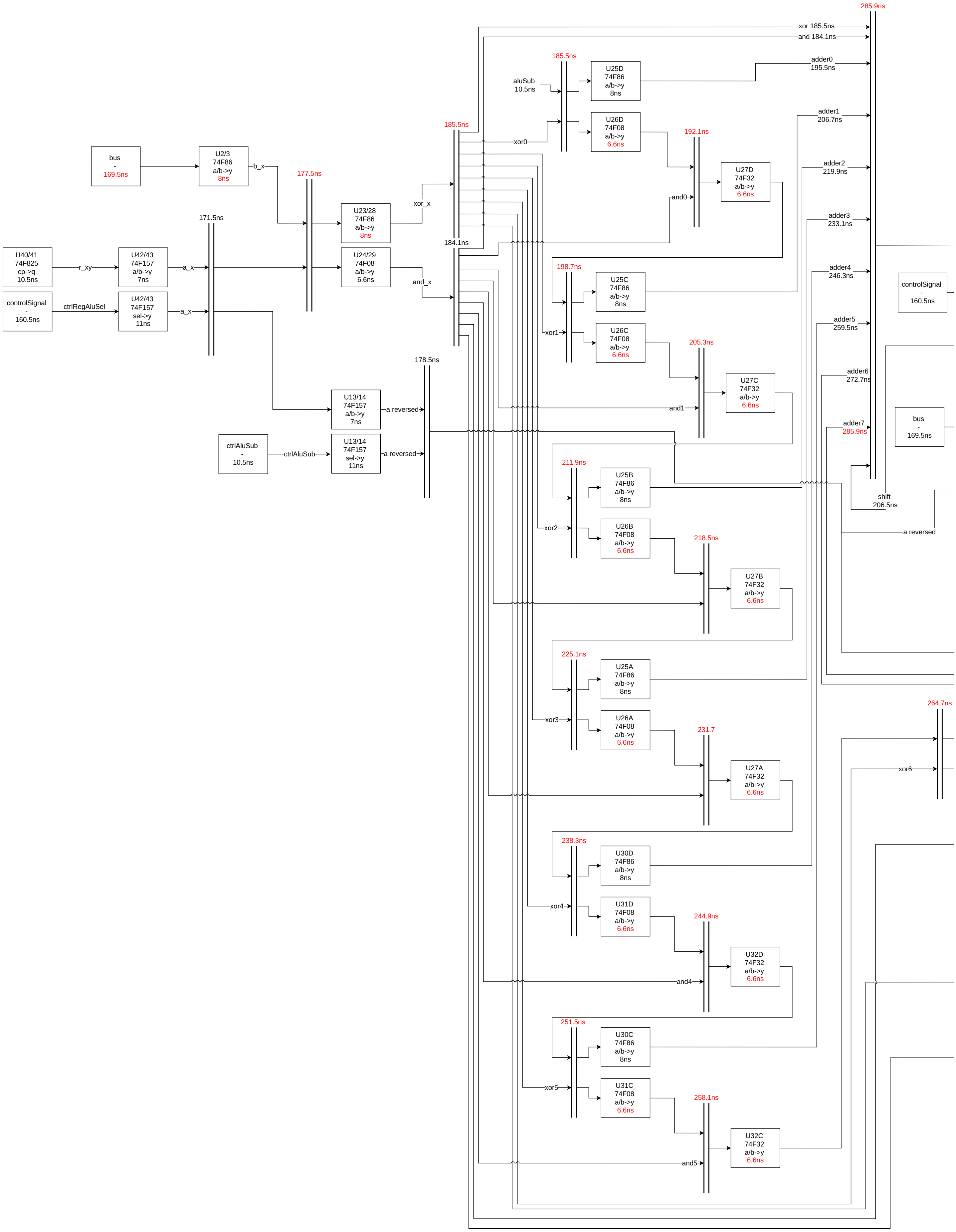
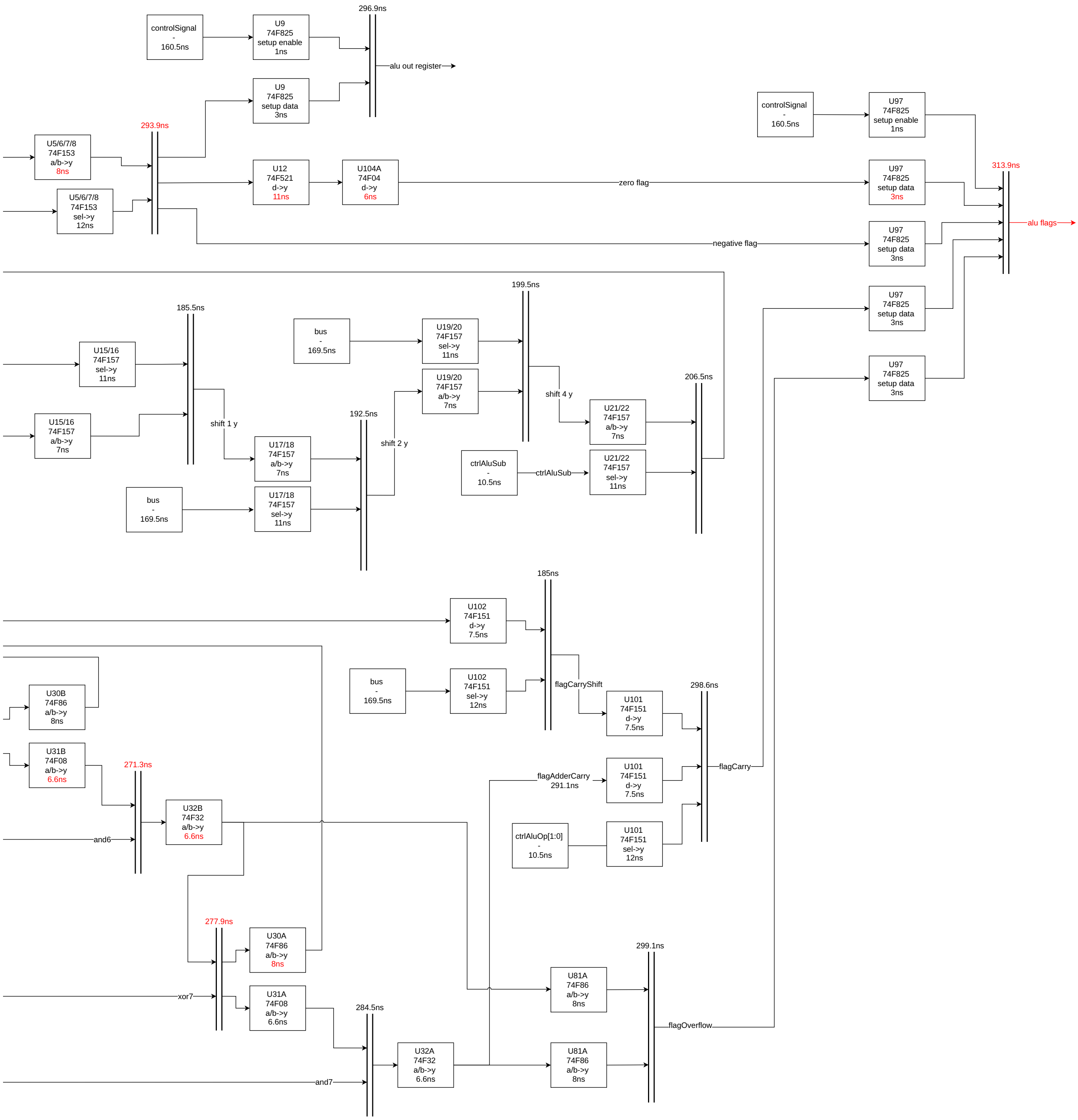


Figure B.2: Timing analysis for the ALU latency.


controlSignal
-
160.5ns
U9
74F825
setup enable
1ns
296.9ns
alu out register
U9
74F825
setup data
3ns
293.9ns
U5/6/7/8
74F153
a/b->y
8ns
U5/6/7/8
74F153
sel->y
12ns
U12
74F521
d->y
11ns
U104A
74F04
d->y
6ns
zero flag
negative flag
controlSignal
-
160.5ns
U97
74F825
setup enable
1ns
U97
74F825
setup data
3ns
313.9ns
alu flags
185.5ns
U15/16
74F157
sel->y
11ns
U15/16
74F157
a/b->y
7ns
shift 1 y
bus
-
169.5ns
U19/20
74F157
sel->y
11ns
U19/20
74F157
a/b->y
7ns
199.5ns
shift 4 y
192.5ns
shift 2 y
U17/18
74F157
a/b->y
7ns
U17/18
74F157
sel->y
11ns
ctrlAluSub
-
10.5ns
ctrlAluSub
U21/22
74F157
a/b->y
7ns
U21/22
74F157
sel->y
11ns
206.5ns
185ns
U102
74F151
d->y
7.5ns
U102
74F151
sel->y
12ns
flagCarryShift
U101
74F151
d->y
7.5ns
298.6ns
flagCarry
flagAdderCarry
291.1ns
ctrlAluOp[1:0]
-
10.5ns
U101
74F151
sel->y
12ns
U30B
74F86
a/b->y
8ns
U31B
74F08
a/b->y
6.6ns
271.3ns
and6
U32B
74F32
a/b->y
6.6ns
277.9ns
U30A
74F86
a/b->y
8ns
xor7
U31A
74F08
a/b->y
6.6ns
284.5ns
and7
U32A
74F32
a/b->y
6.6ns
U81A
74F86
a/b->y
8ns
299.1ns
flagOverflow

C Collection of assembler programs for the EDiC

Code Example C.1: The full snake assembler program.

```
include "prng.s"
include "uart_16c550.s"

SIMPLE_IO = 0xfe00
UART_RX_EMPTY = 0xfe09
UART_TX_FULL = 0xfe0a
UART_DATA = 0xfe0b
PAR1 = 0xff00
PAR2 = 0xff01
PAR3 = 0xff02
ESCAPE0 = 0x1b // \033
ESCAPE1 = 0x5b // '['
BORDER = 0x23 // '#'
SPACE = 0x20 // ' '
HEAD = 0x40 // '@'
LEFT = 0x3c // '<'
RIGHT = 0x3e // '>'
UP = 0x5e // '^'
DOWN = 0x76 // 'v'
ITEM = 0x58 // 'X'
ASCII_W = 0x77
ASCII_A = 0x61
ASCII_S = 0x73
ASCII_D = 0x64
ASCII_CAPITAL_W = 0x57
ASCII_CAPITAL_A = 0x41
ASCII_CAPITAL_S = 0x53
ASCII_CAPITAL_D = 0x44

// global variables
SNAKE_LENGTH     = 0x0000
SNAKE_DIRECTION = 0x0001
SNAKE_HEAD_LINE = 0x0002
SNAKE_HEAD_COL  = 0x0003
SNAKE_TAIL_LINE = 0x0004
SNAKE_TAIL_COL  = 0x0005
SNAKE_LEFT_LINE = 0x0006
SNAKE_LEFT_COL  = 0x0007
PRNG_SEED       = 0x0008 // do not init for extra randomness

// local variables
LINE_COUNTER = 0xff00
COLUMN_COUNTER = 0xff01


// screen is in memory starting from 0x0100
```

```
// one line has 256 bytes for ease of access
LINES = 24
COLUMNS = 80
COLUMNS_1 = 79

0x20.LOST_STRING = "You lost!!! Score: "

start:
  call uart_init
  // clear screen
  mov r0, ESCAPE0
  call uart_write
  mov r0, ESCAPE1
  call uart_write
  mov r0, 0x32 // '2'
  call uart_write
  mov r0, 0x4a // 'J'
  call uart_write

  call createBoard
  call updateItem
  mainLoop:
    call updateHead
    cmp r0, -1
    beq lost
    cmp r0, 1
    beq mainAteItem
    call updateTail
    b mainUpdateBoard
  mainAteItem:
    ldr r0, [SNAKE_LENGTH]
    add r0, 1
    str r0, [SNAKE_LENGTH]
    call updateItem
  mainUpdateBoard:
    ldr r0, [SNAKE_LENGTH]

    // move cursor to the top
    // mov r0, 1 // line
    // stf r0, [PAR2]
    // mov r0, 0 // col
    // stf r0, [PAR1]
    // mov r0, BORDER
    // call setScreen
    ldr r0, [SNAKE_LENGTH]
    str r0, [SIMPLE_IO]
    // wait x ms
    // mov r0, 90
    // call delay_ms

    call readArrow
    // change direction if != -1
    cmp r0, -1
    beq mainLoop
    str r0, [SNAKE_DIRECTION]
  b mainLoop

lost:
// set position to upper center
mov r0, 6 // line
stf r0, [PAR2]
mov r0, 27 // col
stf r0, [PAR1]
mov r0, SPACE
call setScreen
mov r0, LOST_STRING
call outputString
ldr r0, [SNAKE_LENGTH]
call outputDecimal
lostLoop:
b lostLoop
```

```
updateItem:
  str r1, [0xfffe]

  itemColumn:
    call prng
    and r0, 0x7f // limit columns
    cmp r0, COLUMNS
    bhs itemColumn // if out of scope redo
    mov r1, r0
  itemLine:
    call prng
    and r0, 0x1f // limit lines
    cmp r0, LINES
    bgt itemLine // if out of scope redo
    stf r0, [PAR2]
    sma r0 // line
    ldr r0, [r1]
    cmp r0, SPACE
  bne itemColumn // if there is something at the new item position find a new one
  // store new item
  stf r1, [PAR1]
  mov r0, ITEM
  call setScreen

  ldr r1, [0xfffe]
ret

// returns -1 if lost, 0 if nothing happend and 1 if ate item
updateHead:
  str r1, [0xfffe]

  ldr r0, [SNAKE_HEAD_LINE]
  stf r0, [PAR2]
  sma r0
  ldr r0, [SNAKE_HEAD_COL]
  stf r0, [PAR1]
  // load correct direction char into r0
  ldr r1, [SNAKE_DIRECTION]
  cmp r1, 0
  beq headUp
  cmp r1, 1
  beq headDown
  cmp r1, 2
  beq headRight
  cmp r1, 3
  beq headLeft
  b headEnd // should not happen

  headUp:
    mov r0, UP
    call setScreen
    ldr r0, [SNAKE_HEAD_LINE]
    sub r0, 1
    str r0, [SNAKE_HEAD_LINE]
  b headEnd

  headDown:
    mov r0, DOWN
    call setScreen
    ldr r0, [SNAKE_HEAD_LINE]
    add r0, 1
    str r0, [SNAKE_HEAD_LINE]
  b headEnd
```

```
 headLeft:
   mov r0, LEFT
   call setScreen
   ldr r0, [SNAKE_HEAD_COL]
   sub r0, 1
   str r0, [SNAKE_HEAD_COL]
 b headEnd

 headRight:
   mov r0, RIGHT
   call setScreen
   ldr r0, [SNAKE_HEAD_COL]
   add r0, 1
   str r0, [SNAKE_HEAD_COL]
 b headEnd

headEnd:

 ldr r1, [SNAKE_HEAD_LINE]
 stf r1, [PAR2]
 sma r1
 ldr r1, [SNAKE_HEAD_COL]
 stf r1, [PAR1]
 ldr r1, [r1] // load item at new position
 sts r1, [0x00]
 // store & show head
 mov r0, HEAD
 call setScreen
 // if new position is not space or item -> lost
 lds r0, [0x00] // load saved item
 cmp r0, SPACE
 beq headSpace
 cmp r0, ITEM
 beq headItem
 mov r0, -1
 ldr r1, [0xfffe]
ret
headSpace:
 mov r0, 0
 ldr r1, [0xfffe]
ret
headItem:
 mov r0, 1
 ldr r1, [0xfffe]
ret

updateTail:
 str r1, [0xfffe]

 ldr r0, [SNAKE_TAIL_LINE]
 str r0, [SNAKE_LEFT_LINE]
 stf r0, [PAR2]
 sma r0
 ldr r0, [SNAKE_TAIL_COL]
 str r0, [SNAKE_LEFT_COL]
 stf r0, [PAR1]
 // load direction char
 ldr r1, [r0]
 mov r0, SPACE
 call setScreen
 cmp r1, UP
 beq tailUp
 cmp r1, DOWN
 beq tailDown
 cmp r1, RIGHT
 beq tailRight
 cmp r1, LEFT
 beq tailLeft
 b tailEnd // should not happen

 tailUp:
 ldr r1, [SNAKE_TAIL_LINE]
```

```
  sub r1, 1
  str r1, [SNAKE_TAIL_LINE]
  b tailEnd

  tailDown:
  ldr r1, [SNAKE_TAIL_LINE]
  add r1, 1
  str r1, [SNAKE_TAIL_LINE]
  b tailEnd

  tailLeft:
  ldr r1, [SNAKE_TAIL_COL]
  sub r1, 1
  str r1, [SNAKE_TAIL_COL]
  b tailEnd

  tailRight:
  ldr r1, [SNAKE_TAIL_COL]
  add r1, 1
  str r1, [SNAKE_TAIL_COL]
  b tailEnd

tailEnd:
  ldr r1, [0xfffe]
ret

createBoard:
  str r0, [0xfffe]
  str r1, [0xfffd]

  // init snake
  mov r0, 4
  str r0, [SNAKE_LENGTH]
  mov r0, 2
  str r0, [SNAKE_DIRECTION]
  mov r0, 12 // center
  str r0, [SNAKE_HEAD_LINE]
  mov r0, 40 #center
  str r0, [SNAKE_HEAD_COL]
  mov r0, 12
  str r0, [SNAKE_TAIL_LINE]
  mov r0, 37
  str r0, [SNAKE_TAIL_COL]
  mov r0, 12
  str r0, [SNAKE_LEFT_LINE]
  mov r0, 36
  str r0, [SNAKE_LEFT_COL]

  // move to home position
  mov r0, ESCAPE0
  call uart_write
  mov r0, ESCAPE1
  call uart_write
  mov r0, 0x48 // 'H'
  call uart_write


  // first and last line is full border
  mov r1, 0
  createLine0Loop:
    sma 1
    mov r0, BORDER
    str r0, [r1]
    call uart_write
    add r1, 1
    cmp r1, COLUMNS
  blt createLine0Loop

  mov r0, 0x0a // LF
  call uart_write
  mov r0, 0x0d // CR
  call uart_write

  // line 2 to 23 have first and last column border
```

```
  mov r1, 2 // skip first line
  str r1, [LINE_COUNTER]
  createLineLoop:
    // load mar1 with line space
    sma r1
    mov r1, 0
    mov r0, BORDER
    str r0, [r1]
    call uart_write
    add r1, 1
    // loop through line (1-79) and store space
    createColumnLoop:
      ldr r0, [LINE_COUNTER]
      sma r0
      mov r0, SPACE
      str r0, [r1]
      call uart_write
      add r1, 1
      cmp r1, COLUMNS_1
    blt createColumnLoop
    // store end border
    mov r0, BORDER
    str r0, [r1]
    call uart_write

    mov r0, 0x0a // LF
    call uart_write
    mov r0, 0x0d // CR
    call uart_write

    ldr r1, [LINE_COUNTER]
    add r1, 1
    str r1, [LINE_COUNTER]
    cmp r1, LINES
  blt createLineLoop // skip last line

  // draw last line
  mov r1, 0
  createLineLastLoop:
    sma LINES
    mov r0, BORDER
    str r0, [r1]
    call uart_write
    add r1, 1
    cmp r1, COLUMNS
  blt createLineLastLoop

  // draw snake
  ldr r0, [SNAKE_HEAD_LINE]
  stf r0, [PAR2]
  ldr r0, [SNAKE_HEAD_COL]
  stf r0, [PAR1]
  mov r0, HEAD
  call setScreen

  mov r1, 1
  snakeBody:
    ldr r0, [SNAKE_HEAD_LINE]
    stf r0, [PAR2]
    ldr r0, [SNAKE_HEAD_COL]
    sub r0, r1
    stf r0, [PAR1]
    mov r0, RIGHT
    call setScreen
    add r1, 1
    cmp r1, 3
  ble snakeBody

  ldr r0, [0xfffe]
  ldr r1, [0xfffd]
ret
```

```


// r0: char, PAR1: col, PAR2: line
setScreen:
  str r0, [0xfffe]
  str r1, [0xfffd]

  // store
  ldr r1, [PAR2]
  sma r1
  ldr r1, [PAR1]
  str r0, [r1]

  // decimal needs to be one based
  mov r0, ESCAPE0
  call uart_write
  mov r0, ESCAPE1
  call uart_write
  ldr r0, [PAR2] // line is already one based
  call outputDecimal
  mov r0, 0x3b // ';'
  call uart_write
  ldr r0, [PAR1]
  add r0, 1 // column is not one based
  call outputDecimal
  mov r0, 0x48 // 'H'
  call uart_write

  ldr r0, [0xfffe]
  call uart_write

  ldr r1, [0xfffd]

ret

// r0 is parameter
outputDecimal:
  str r1, [0xfffe]

  mov r1, 100
  stf r1, [PAR1]
  call divMod // r0 / 100
  ldf r1, [PAR1] // mod result
  add r0, 0x30 // make to char
  call uart_write
  mov r0, r1 // remainder is parameter for next divMod
  mov r1, 10
  stf r1, [PAR1]
  call divMod
  ldf r1, [PAR1]
  add r0, 0x30 // make to char
  call uart_write
  mov r0, r1 // last char to output
  add r0, 0x30 // make to char
  call uart_write

  ldr r1, [0xfffe]
ret

// r0: address of string
outputString:
  str r1, [0xfffe]
  sts r0, [0x00]
  mov r1, 0
  outputStringLoop:
```

```
      lds r0, [0x00]
      sma r0
      ldr r0, [r1]
      cmp r0, 0
      beq outputStringEnd
      call uart_write
      add r1, 1
      cmp r1, 255
      bne outputStringLoop

    outputStringEnd:

    ldr r1, [0xfffe]
ret


// r0 / PAR1
// result: r0 -> div, *PAR1 -> mod
divMod:
    str r1, [0xfffe]
    mov r1, 0
    divLoop:
      add r1, 1
      sub r0, [PAR1]
    bpl divLoop // positive or zero (N Clear)
    // executing one step too much, undo it
    add r0, [PAR1]
    sub r1, 1

    str r0, [PAR1]
    mov r0, r1
    ldr r1, [0xfffe]
ret

// r0 is return value:
// -1 for nothing, 0 for up, 1 for down, 2 for right, 3 for left
readArrow:
  str r1, [0xfffe]
readArrowLoop:
  call uart_read
  cmp r0, 0
  beq readArrowNothing // no char received
  // up
  cmp r0, ASCII_W
  beq readArrowUp
  cmp r0, ASCII_CAPITAL_W
  beq readArrowUp
  // left
  cmp r0, ASCII_A
  beq readArrowLeft
  cmp r0, ASCII_CAPITAL_A
  beq readArrowLeft
  // down
  cmp r0, ASCII_S
  beq readArrowDown
  cmp r0, ASCII_CAPITAL_S
  beq readArrowDown
  // right
  cmp r0, ASCII_D
  beq readArrowRight
  cmp r0, ASCII_CAPITAL_D
  beq readArrowRight

  cmp r0, ESCAPE0
  bne readArrowLoop // make sure to empty the fifo

  call uart_read
  cmp r0, 0
  beq readArrowNothing
```

```
  cmp r0, ESCAPE1
  bne readArrowLoop

  call uart_read
  cmp r0, 0x41 // A
  blt readArrowLoop
  cmp r0, 0x44 // D
  bgt readArrowLoop
  sub r0, 0x41 // return 0-4
ret
// -1 for nothing, 0 for up, 1 for down, 2 for right, 3 for left
  readArrowNothing:
  ldr r1, [0xfffe]
  mov r0, -1
ret
  readArrowUp:
  ldr r1, [0xfffe]
  mov r0, 0
ret
  readArrowLeft:
  ldr r1, [0xfffe]
  mov r0, 3
ret
  readArrowDown:
  ldr r1, [0xfffe]
  mov r0, 1
ret
  readArrowRight:
  ldr r1, [0xfffe]
  mov r0, 2
ret

// r0: delay in ms
delay_ms:
  sts r0, [0x00]

  delay_ms_outer_loop:

    // 2MHz clock -> 1ms is 2000cycle
    // per loop 4+4+3+3=14 cycles (below)
    // -> 198.6 times 10 cycles per iteration
    mov r0, 0
    delay_ms_loop:
      add r0, 1 // 4 cycles
      cmp r0, 199 // 3 cycles
    blo delay_ms_loop // 3 cycles

    lds r0, [0x00] // 4 cycles
    sub r0, 1 // 4 cycles
    sts r0, [0x00] // 3 cycles
  bhi delay_ms_outer_loop // 3 cycles
ret
```

Code Example C.2: The PRNG assembler program “prng.s” used in the snake program in code example C.1.

```
PRNG_SEED = 0x0000
SIMPLE_IO = 0xfe00

```

```
prng:
  ldr r0, [PRNG_SEED]
  subs r0, 0
  beq prngDoEor
  lsl r0, 1
  beq prngNoEor
  bcc prngNoEor
prngDoEor:
  xor r0, 0x1d
prngNoEor:
  str r0, [PRNG_SEED]
ret

start:
  mov r0, 0
  str r0, [PRNG_SEED]
  prng_loop:
    call prng
    str r0, [SIMPLE_IO]
  b prng_loop
```

Code Example C.3: The utility library for the UART extension card of the EDiC with the `16c550` UART Transceiver.

```
UART_DAT = 0xfe08
UART_IER = 0xfe09
UART_IIR = 0xfe0a
UART_FCR = 0xfe0a
UART_LCR = 0xfe0b
UART_MCR = 0xfe0c
UART_LSR = 0xfe0d
UART_MSR = 0xfe0e
UART_SCR = 0xfe0f
UART_DLL_DLAB = 0xfe08
UART_DLM_DLAB = 0xfe09

UART_DIV = 10 // 19200 baud
// UART_DIV = 20 // 9600 baud
UART_FILL_AMOUNT = 60 // 19200 baud
// UART_FILL_AMOUNT = 30 // 9600 baud

uart_init:
  // line control register
  // 8bit, 2 stopbits, no parity, dlab active:
  // 0b10xx_0111
  // 8bit, 1 stopbit, no parity, dlab active:
```

```
    // 0b10xx_0011
    mov r0, 0x87
    str r0, [UART_LCR]

    // divisor latch access
    mov r0, 0x00
    str r0, [UART_DLM_DLAB]
    mov r0, UART_DIV
    str r0, [UART_DLL_DLAB]

    // lcr as above but dlab inactive
    mov r0, 0x07
    str r0, [UART_LCR]

    // fifo control register
    // fifo enable, reset tx and rx fifo
    // 0b00xx_x111
    mov r0, 0x07
    str r0, [UART_FCR]

    // interupt enable register
    // clear all interupts -> fifo polled mode
    mov r0, 0x00
    str r0, [UART_IER]

    // modem control register
    // assert dtr, deassert rts (should be asserted?),
    // 0bxxx0_xx01
    mov r0, 0x01
    str r0, [UART_MCR]
ret

// r0 is byte to write
uart_write_inner:
  sts r1, [0x00]

  uart_write_loop:
    ldr r1, [UART_LSR]
    and r1, 0x20 // bit 5, fifo empty (not full?) -> if 1, can accept new data
  beq uart_write_loop

  str r0, [UART_DAT]

  lds r1, [0x00]
ret

uart_write:
  sts r1, [0x00]
  call uart_write_inner

  cmp r0, 0x20 // if less than 0x20 -> send fill null bytes
  bge uart_write_end

  mov r0, 0x00
  mov r1, UART_FILL_AMOUNT
  uart_write_fill_loop:
    call uart_write_inner
    sub r1, 1
    cmp r1, 0
  bhi uart_write_fill_loop

uart_write_end:
  lds r1, [0x00]
ret


// r0 is byte to write
```

```
uart_read:
  ldr r0, [UART_LSR]
  and r0, 0x01 // bit 0, fifo not empty -> 1 if data exists
  beq uart_read_0

  ldr r0, [UART_DAT]
ret
uart_read_0:
  mov r0, 0
ret


// r0 is byte to write
uart_read_busy:
  sts r1, [0x00]

  uart_read_busy_loop:
    ldr r1, [UART_LSR]
    and r1, 0x01 // bit 0, fifo not empty -> 1 if data exists
  beq uart_read_busy_loop

  ldr r0, [UART_DAT]

  lds r1, [0x00]
ret


start:
  call uart_init
  uart_loop:
    call uart_read
    str r0, [0xfe00]
    cmp r0, 0
    beq uart_loop
    call uart_write
  b uart_loop
```