

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

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 07.06.2022

Unterschrift

Contents

1	Introduction	1
1.1	The Beginning	1
2	Architecture	5
2.1	Design Decisions	5
2.1.1	8 bit bus width	5
2.1.2	Datapath Architecture - Multicycle CISC	6
2.1.3	Single-Bus Oriented	7
2.2	Modules	8
2.2.1	Arithmetic Logic Unit (ALU)	8
2.2.2	Register File	11
2.2.3	Program Counter (PC) & Instruction Register	11
2.2.4	Control Logic	12
2.2.5	Memory	13
2.2.5.1	Memory Mapped I/O	13
2.2.5.2	Stack Implementation	13
2.2.5.3	Addressing Logic	14
2.2.6	Input & Output	15
2.2.7	Clock, Reset & Debugging	15
2.3	Final Instruction Set	16
2.3.1	Arithmetic Logic Unit (ALU) operations	16
2.3.2	Memory operations	18
2.3.3	Miscellaneous operations	20
3	Software Development Environment	23
3.1	Microcode Generation	23
3.2	Assembler	28
3.2.1	Calling conventions	30
3.2.2	Available Instructions	31
3.2.2.1	ALU Instructions	31
3.2.2.2	Memory Instructions	32

3.2.2.3	Miscellaneous Instructions	33
3.2.3	Constants	34
3.2.3.1	Value constants	34
3.2.3.2	Labels	34
3.2.3.3	String constants	36
3.2.4	File imports	38
3.2.5	Syntax Definition for VS Code	39
4	Field Programmable Gate Array (FPGA) Model	41
4.1	FPGAs Background	41
4.2	FPGA choices	42
4.2.1	Language Choice	43
4.2.2	Tri-State Logic in FPGAs	43
4.3	Behavioral Implementation	44
4.4	Chip-level Implementation	46
4.4.1	Conversation Script	46
4.4.1.1	Electrically Erasable Programmable Read-Only Memorys (EEPROMs)	48
4.4.1.2	Tri-State Ports	49
5	Hardware Design	51
5.1	Timing Analysis	52
6	Initial Hardware Test & Component Verification	55
6.0.1	Test Adapter	55
7	Conclusion and Future Work	57
	Acronyms	59
	List of Figures	61
	List of Tables	63
	List of Code Examples	66
	Bibliography	67
A	Collection of assembler programs	69

Abstract

This thesis covers stuff.

Kurzfassung

Diese Arbeit umfasst Zeugs.

1 Introduction

This thesis covers the development and engineering process of the Educational Digital Computer (EDiC) pictured in figure 1.1. It is a completely novel Central Processing Unit (CPU) architecture built to visualize and show the fundamental workings of any CPU. The EDiC can execute over half a million instructions per second but also features step-by-step, instruction-by-instruction as well as breakpoint capabilities for better understanding of how CPUs work. All components can be tested individually with the help of dedicated test adapters and, therefore, Integrated Circuit (IC) failures can be tracked down and fixed easily. Additionally to the hardware built, the project includes an open source development environment including an assembler, tools to modify the micro-code and also Field Programmable Gate Array (FPGA) simulation and emulation of the hardware [9].

1.1 The Beginning

The foundation of this project started at the end of 2020 where I decided to design and build a CPU from scratch. In many university courses we would discuss some parts of a CPU like different approaches to binary adders or pipelining concepts but never would we build a complete CPU including the control logic. Due to a Covid-19 lockdown I had enough time at my hands and after 6 years of study, I felt like I had the expertise to complete this project.

At the end of January 2021 I succeeded with the actual hardware built and the CPU was able to execute a prime factorization of 7 bit numbers. Figure 1.2 depicts the final hardware build. Its design ideas, implementation and flaws are shown in ??.

TODO: rewrite to current chapters

Through the university module “Mixed-Signal-Baugruppen” I got to know Henry

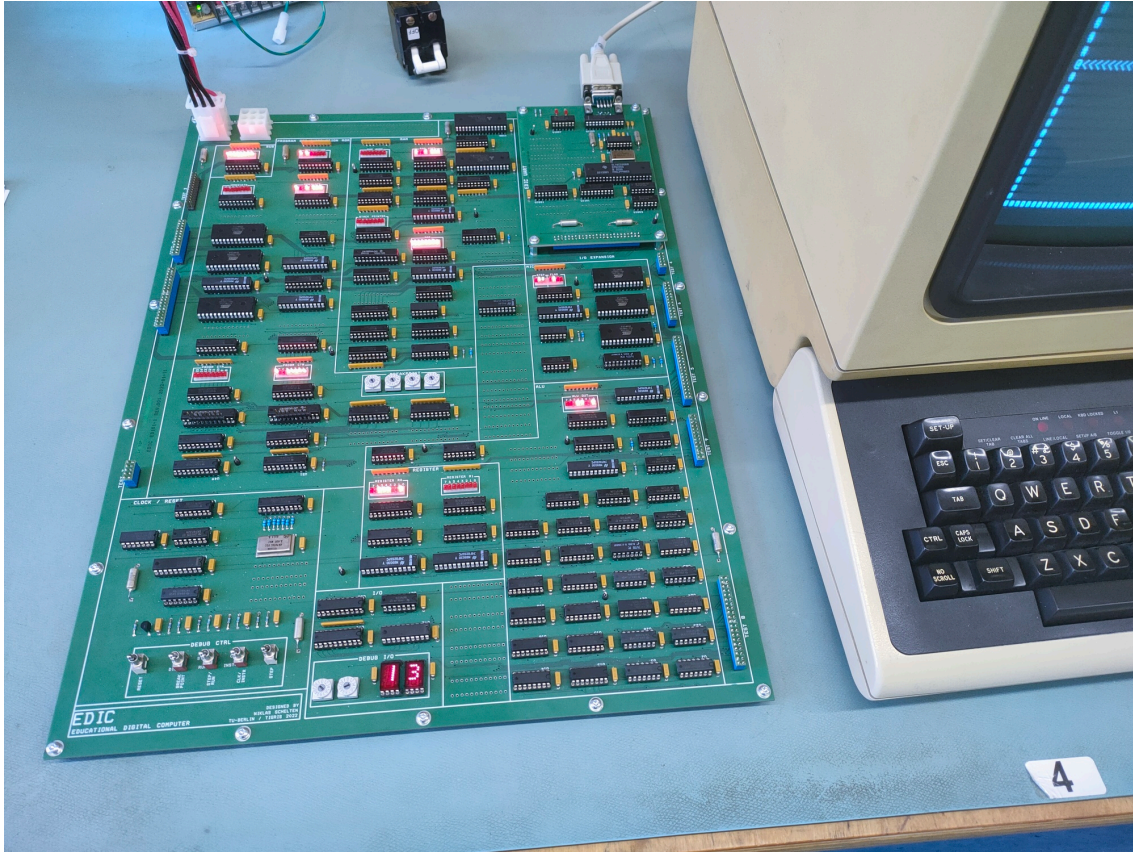


Figure 1.1: The final version of the EDiC playing Snake on a VT-100 over an RS-232 I/O card.

Westphal in summer 2021. He established a company that builds mixed-signal-electronics and, therefore, has a deep understanding of analog and digital circuitry. As he heard of my plans to build a future version of my CPU he was immediately interested and we wanted to rebuild a CPU with some changes:

- The general architecture should remain similar to the existing CPU with only changes where it was necessary.
- The objective was no longer only to create a functioning CPU, this was already accomplished, but the build should be such that it could be used for education.
- It should be more reliable, more capable and its components should be easily distinguishable. Therefore, it is to be build on a large Printed Circuit Board (PCB).
- There should be a generic interface for extension cards, i.e. IO Devices.

How the, now called, EDiC differs from its predecessor is presented in chapter 2.

To achieve the goal of the EDiC being educational it is important to not only build

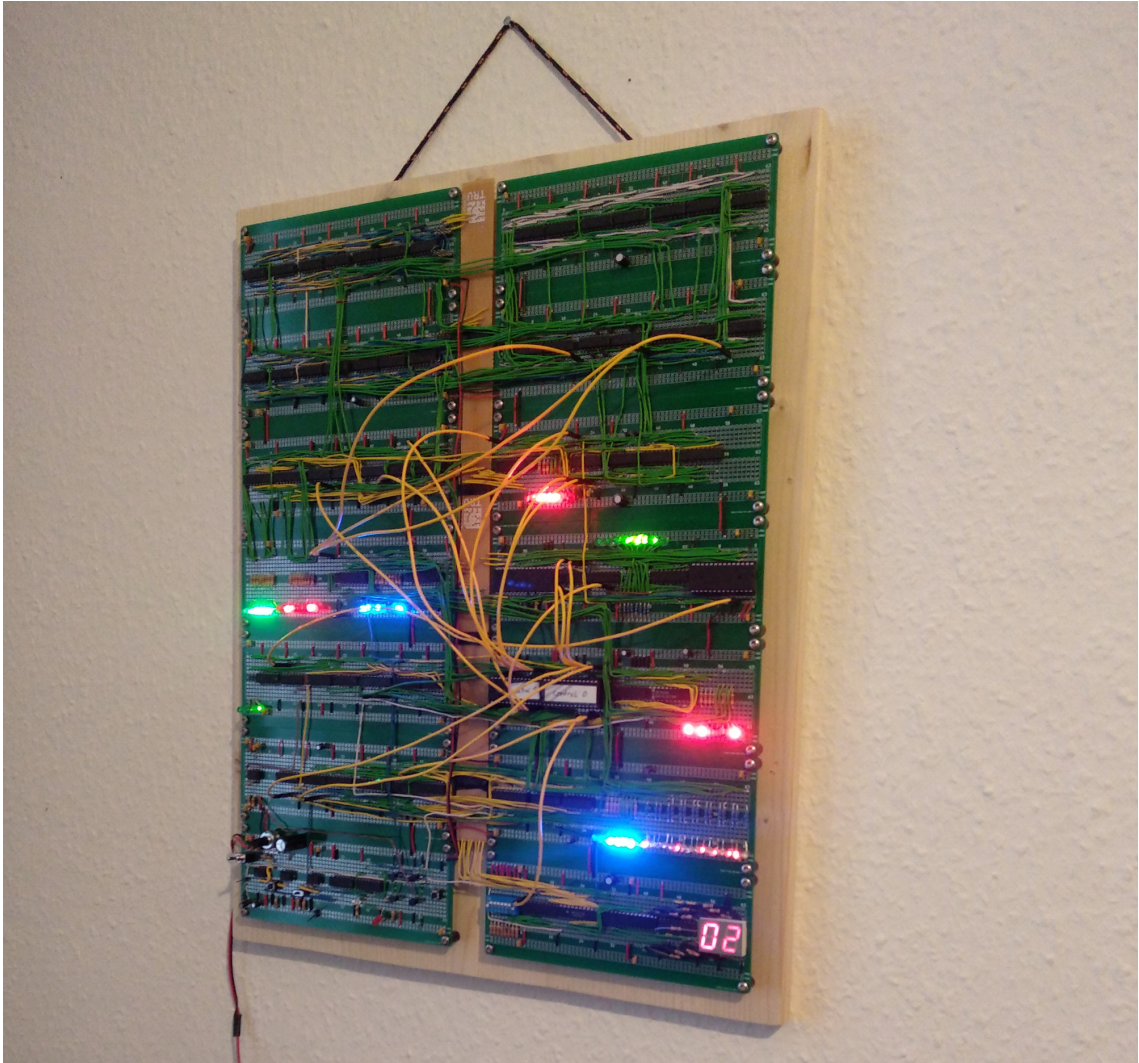


Figure 1.2: The first version of the CPU in its final state.

the hardware but to also provide a Software Environment to, for example, write applications. This is presented in chapter 3.

An important step in the design of the EDiC was to thoroughly simulate and implement the behavior on an FPGA. I firstly simulated the behavior and after the hardware schematic was finished, we built a script to convert the exported netlist to verilog to simulate the CPU on chip level. The process and differences between the FPGA design and the actual hardware are presented in chapter 4.

Chapter 5 describes the final hardware assembly, commissioning and timing analysis to determine the final clock frequency.

The final conclusion and future improvements are given in chapter 7.

2 Architecture

Designing and building a general purpose CPU includes a lot of architectural decisions which will decide how well the CPU performs, how complex it is and a lot more. The goal for the EDiC was to build a CPU that is capable of basic basic interaction with an I/O device such as the VT-100 but at the same time simple enough to easily understand its workings, such that it may 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 setup. Another important factor towards architectural structure is the fact that the final hardware build of the CPU will be based on the 74-series of 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 an easy to understand hardware build. Some early CPUs build with similar ICs worked with only 4 bits. This can work very well for specific applications but for the most arithmetics and data handling 8 bits are more practical. The EDiC will, therefore, use an 8 bit bus for data with a integer range of -128 to 127 or 0 to 255 for unsigned integers.

One of the major limitation 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 the first version of the CPU this limitation was extended a bit by providing 256 bytes of instruction memory besides 256 bytes of read only memory for instruction immediate values and 256 bytes of addressable Static Random-Access Memory (SRAM). However, especially with a Complex Instruction

Set Computer (CISC) architecture, 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 instructions¹. However, this raises problems of where the 16 bit addresses 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* [18]. 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 [4]. For example a add instruction in CISC could fetch operands from memory, execute the add and write the result back to memory. Reduced Instruction Set Computers (RISCs) on the other hand would need independent instruction to load operands from memory into registers, do the addition and write the result back to memory.

In Pipelining there a fixed steps each instruction goes through in a defined order and the intermediate results are stored in so called pipeline registers. Each pipeline step is constructed in such a way that it does not intervene with the others. Therefore, it is, in theory, possible to dispatch a new instruction each cycle even though the previous instruction is not yet finished. A typically 5-step pipeline would consist of the following steps [18]:

1. **Instruction Fetch:** The instruction is retrieved from memory and stored in a register.
2. **Instruction Decode:** The fetched instruction is decoded into control signals (and instruction specific data) for all the components of the CPU.
3. **Execute:** If arithmetic or logical operations are part of the instruction, they are performed.

¹The largest feasible Electrically Erasable Programmable Read-Only Memory (EEPROM) available has only 15 address bits and with that only 32768 words of data.

4. **Memory Access:** Results are written to the memory and/or data is read from memory.
5. **Writeback:** The results are written back to the registers.

However good the performance of a pipelined CPU is, it also comes with challenges. Those include a greater resource usage because all intermediate results need to be stored in pipeline registers. Additionally, branch instructions² pose a greater challenge because at the moment, the CPU execute the branch the next instructions have already been dispatched. This means that the pipeline needs to be flushed (i.e. cleared), performance is lost and more logic is required. It also noteworthy that branch prediction and pipeline flushes can be quite vulnerable as recently shown in CVE-2017-5753 with the Spectre bug [6].

Therefore, the EDiC is to 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’ or ‘1’ or high impedance which allows other tri-state drivers on the same bus to drive it. That way an instruction which fetches a word from the memory from an address stored in a register, 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
6. ALU operation
7. Data from ALU output over *bus* to register

²Branch 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.

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	$(A \vee B)$ XOR
1	0	1	$(\overline{A} \vee B)$ XNOR
1	1	0	$(A \gg B)$ logical shift right
1	1	1	$(A \ll B)$ logical shift left

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 7 rather independent modules of varying complexity which mainly interface with the bus and control signals.

2.2.1 Arithmetic Logic Unit (ALU)

An ALU is the operational core of any CPU as it performs the calculations. The ALU of the EDiC is by design simple with only 4 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, one input would have needed a register and every ALU instruction would have taken two instead of one cycle.

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 essentially subtracts the B input from the A input

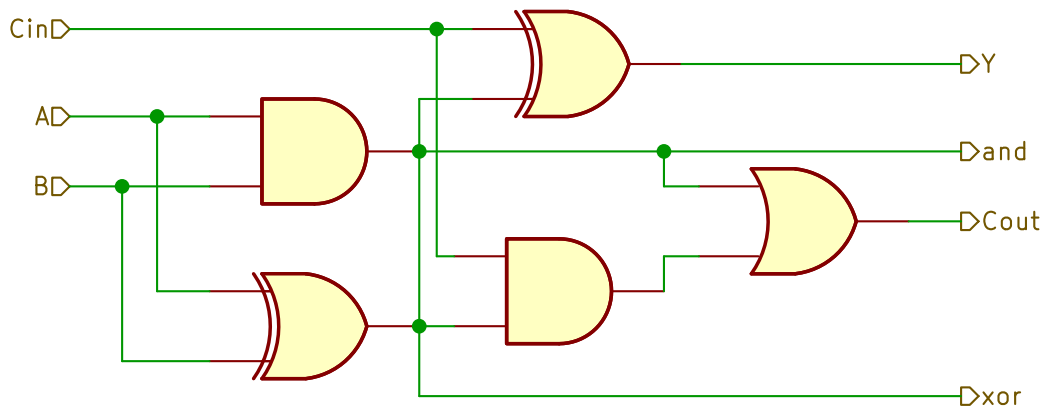


Figure 2.1: 1 bit full adder with the usual A, B and Carry inputs and Y and Carry outputs as well as the XOR and AND outputs.

in two's complement arithmetic. 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 3 consecutive multiplexers to shift by 1, 2 or 4 bit to the right that are controlled by the first 3 bit of the (not inverted) B input. To also allow shifting to the left there is one multiplexer before the three shift multiplexers to invert the order of bits and another one after the shifting to reorder the bits. In figure 2.2 a bidirectional barrel shifter implemented with the 74F157 is visualized. The 74F157 implements four 2 to 1 multiplexer and, therefore, 2 chips are needed for a full 8 bit 2 to 1 multiplexer.

The 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 first version of the CPU. However, the experience of programming for the CPU showed that it is desirable to 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 width³ 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.

³An overflow and with that a greater or less than comparison cannot be done

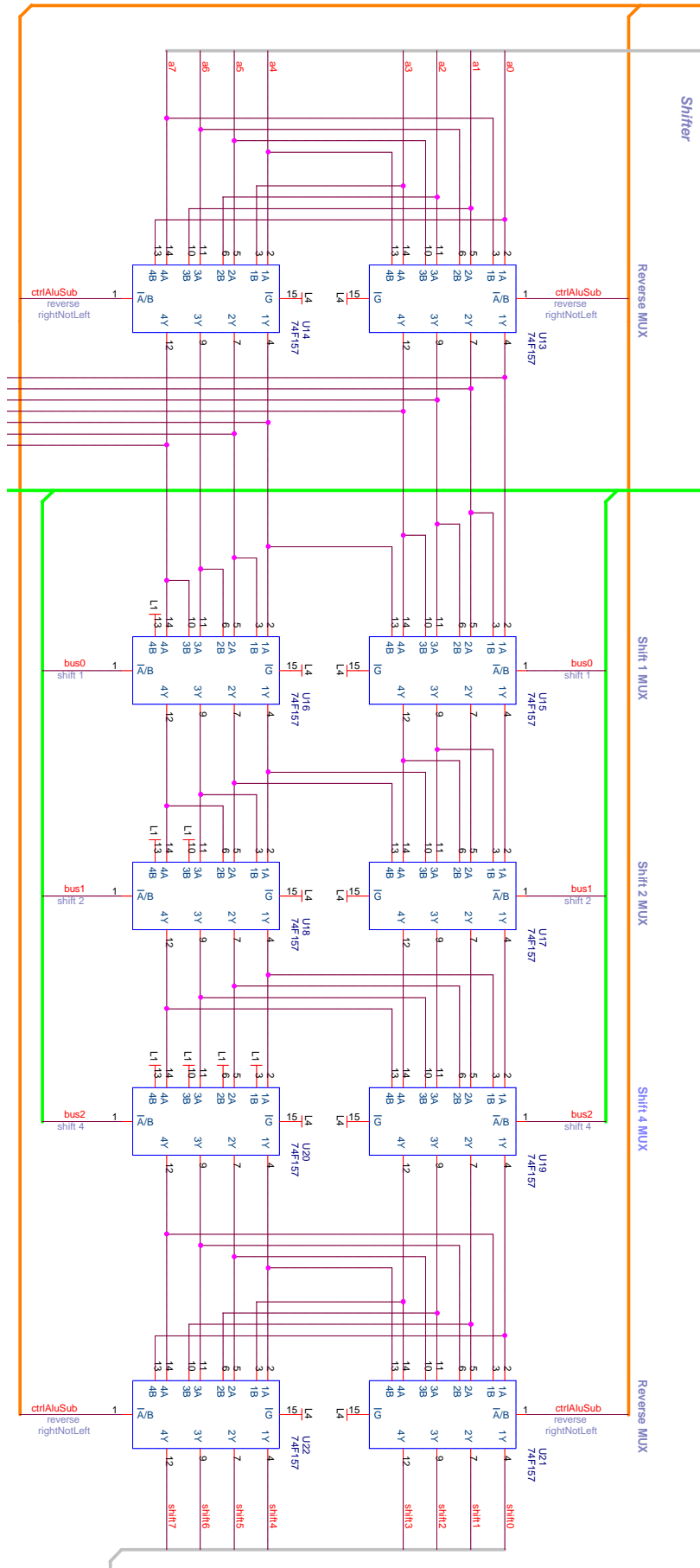


Figure 2.2: 8 bit bidirectional barrel shifter.

A lot of modern CPUs feature many different flags with the Intel 64[®] and IA-32 CPU having about 20 different flags [11, 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 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 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 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 address for the instruction EEPROMs and can also be driven to the bus (lower 8 bits) and a second 8 bit memory line (section 2.2.5) to store the PC 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 which 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. For keeping track which cycle of each instruction is currently executing a 3 bit synchronous counter is needed. Each control signal could be derived by a logical circuitry with 13 inputs: 8 bits instruction, 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. (For example when finding a bug in one instruction) 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 bits ALU opcodes. They are not decoded the usual way from the instruction but are directly take from the 3 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 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, because it is still being fetched and decoded. However, the instruction fetch and decode are always the same for each instruction, which means that all memory locations where the cycle counter is equal to 0 or 1 (the first two instructions) are filled with the control signals for an instruction fetch and decode.

2.2.5 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 address 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 result logic is described below.

2.2.5.1 Memory Mapped I/O

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

2.2.5.2 Stack Implementation

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

However, a typical stack implementation as in modern CPU architectures like ARM is rather complex. It requires a Stack Pointer (SP) register which usually is accessible like any other general purpose register and can be directly used as an address. This includes using it as 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 an unique approach to the stack:

Similarly to the memory mapped I/O it was decided to implement the stack as an 8 bit register which can be incremented and decremented. Every time a memory access is performed where the upper 8 bits of the address equal `0xff`, a 17th address bit is set and the upper 8 address bits are replaced by the current value of the SP. For example: The SP is currently `0x21` and a memory access to the address `0xff42` is performed. Then the actual address at the memory IC is `0x1_2142`.

This allows each function (which has a unique SP value on the current call stack) to have 256 bytes of function local memory. In the *call* instruction, the EDiC automatically stores the return address (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 2.3.

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 parameter before calling a function and it is also possible to retrieve modified values after the call⁴

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. 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 the 16 bit instruction immediate. As the bus is only 8 bits wide, there is a special instruction to write to the upper 8 bits of the MAR and the lower bits are written in the memory access instruction. This can be used when a memory address is stored in registers and is needed when looping through values in the memory like arrays. When accessing addresses known at compile time, the instruction immediate can be used as an address which has

⁴This is important when a function takes memory pointers as parameters and modifies them. 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.

been extended to support 16 bit. These two sources of addresses are then decoded to either select the stack (upper 8 bits equal `0xff`), memory mapped I/O (`0xfe`) or regular memory access. The chip enable of the main memory is only asserted when performing stack and regular memory accesses while the I/O chip enable is only asserted when the upper 8 bits are `0xfe`. Additionally, the 17th address bit is asserted when stack access is performed and the upper 8 bits of the address are replaced with the SP in this case.

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 values to be read can be selected by the user with a hexadecimal 8 bit switch and the values written to the address `0x00` are displayed with a 2 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 version could at least step the clock cycle by cycle. However, as programs get complexer this feature quickly becomes less useful as each instruction is made of several cycles and when a problem occurs after several hundred instructions it is infeasible to step through all cycles. Additionally, the usual application developer does not want to step through each cycle but rather step through each instruction, assuming that the instruction set works as intended. Another important debugging feature is the use of breakpoints where the CPU halts execution when the PC reaches a specific address.

In the EDiC halting was not realized by stopping the clock completely but rather by inhibiting the instruction step counter increment. This has the advantage that the clock is not abruptly pulled to 0 or 1 and, therefore, no spikes on the clock line can occur. To implement a cycle by cycle stepping mode, the halt signal is de-asserted for only one clock cycle, which in turn increments the step counter only once. To step whole instructions, the halt signal is de-asserted 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 those

two equal. As soon as the CPU halts, the user can then switch to stepping mode and debug the specific instruction of the program. The user can freely switch between these modes with switches and buttons.

TODO: explain control signals

2.3 Final Instruction Set

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

2.3.1 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.

- *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 MAR⁵.
 2. Address calculations.
 3. First register and memory content as A and B inputs, write enable of the result register.
 4. Write content of ALU result register into first parameter register.
- *Register x Memory (from immediate)*: Takes one register as ALU A input and a 16 bit value as immediate which is used as a memory address for the ALU B input. The result is stored in the first register.

Cycles:

1. Address calculations.
 2. First register and memory content as A and B inputs, write enable of the result register.
 3. Write content of ALU result register into first parameter register.
- *Register x Memory (from immediate, no write back)*: Takes one register as ALU A input and a 16 bit value as immediate which is used as a memory address for the ALU B input. The result is only calculated for the status flags.

Cycles:

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

⁵The upper 8 bits of the MAR should be set beforehand

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

2.3.2 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 subfunctions.

Cycles:

1. Increment Stack Pointer.
2. Memory address setup.

3. Memory read access and write back to first register.
 4. Memory address hold.
 5. Decrement Stack Pointer.
- *Store to register address*: Takes the second register parameter as the lower 8 bits of the memory address and writes the content of the first register to the memory.

Cycles:

1. Second register to lower MAR.
 2. Memory address and data setup.
 3. Memory write access.
 4. Memory address and data hold.
- *Store to immediate address*: Takes a 16 bit immediate as the memory address and writes the register content to memory.

Cycles:

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

Cycles:

1. Increment Stack Pointer.
 2. Memory address and data setup.
 3. Memory write access.
 4. Memory address and data hold.
 5. Decrement Stack Pointer.
- *Set upper 8 bits of MAR from register*: Sets the upper MAR register to the content of the register.

Cycles:

1. Register output enable and upper MAR write enable.
- *Set upper 8 bits of MAR from immediate*: Sets the upper MAR register to the 8 bit immediate value.

Cycles:

1. Immediate output enable and upper MAR write enable.

2.3.3 Miscellaneous operations

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

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

Cycles:

1. Second register output enable and first register write enable.
- *Move immediate to register*: Set the register to the value of the immediate.

Cycles:

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

Cycles:

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

Cycles:

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

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

Cycles:

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

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. Even more extreme is content of the EEPROMs holding the micro code, i.e. that decode the instruction depending on the instruction cycle and ALU flags.

Therefore, the EDiC comes with two main software utilities that form the Software Development Environment.

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 to easily make changes 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) [12] which is widely used in the computer industry. Besides basic types as strings and numbers, it allows basic 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

```
1 interface IMicrocodeFile {
2     signals: [
3         {
4             name: string;
5             noOp: 0 | 1;
6         }
7     ];
8
9     instructionFetch: [
10        {
11            [signalName: string]: 0 | 1;
12        }
13    ];
14
15    instructions: [
16        {
17            op: string;
18            cycles: [
19                {
20                    [signalName: string]: 0 | 1 | 'r' | 's' | '!r' | '!s';
21                }
22            ];
23        }
24    ];
25 };
```

Code Example 3.1: Schema of the Microcode Definition CSON-File [3] as a TypeScript [14] Type definition.

- Keys do not require string quotation marks

The schema for the file describing the microcode is shown in Code Example 3.1. The file is an object with three key, value pairs:

```
1 {
2     name: 'regONWE'
3     noOp: 1
4 }
```

Code Example 3.2: Example of a control signal definition for the microcode generation.

Signals The signals array consists of Objects that define the available control signals and the default value of the control signal. Code Example 3.2 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 it will stay high and, therefore, register 0 will not be written.

InstructionFetch This array defines the steps that are performed at the beginning of each instruction to fetch the new instruction and decode it. Each object represents one step and consists of key value pairs that define one control signal.

```

1      instructionFetch: [
2          { # write instruction
3              memInstrNWE: 0
4          }
5          { # increment PC
6              memPCNEn: 0
7              memPCLoadN: 1
8          }
9      ]

```

Code Example 3.3: Definition of the instruction fetch and decode steps for the microcode generation.

For example Code Example 3.3 the first step 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 specifies if register 0 or 1 is used to be set to 'r' or 's' and then multiple instructions are generated. Each instruction then The **cycles** array define the steps each instruction does in the same way as the **instructionFetch** array does. However, as the value of individual control signals may depend up on which register is specified in the op code, it is also possible to specify 'r', '!r', 's' or '!s'.

Code Example 3.4 defines the move immediate to register instruction for both register at the same time. The *instruction immediate not output enable* is low and

```
1  {
2    op: '1111100r' # r = imm
3    cycles: [
4      { # imm to bus to r
5        regONWE: 'r'
6        reg1NWE: '!r'
7        memInstrNOE: 0
8      }
9    ]
10 }
```

Code Example 3.4: Definition of the move immediate to register instruction for the microcode generation.

either register 0 or register 1 is written to. This definition would be equal to Code Example 3.5.

```
1  [
2    {
3      op: '11111000' # r0 = imm
4      cycles: [
5        { # imm to bus to r0
6          regONWE: 0
7          reg1NWE: 1
8          memInstrNOE: 0
9        }
10     ]
11   }
12   {
13     op: '11111001' # r1 = imm
14     cycles: [
15       { # imm to bus to r1
16         regONWE: 1
17         reg1NWE: 0
18         memInstrNOE: 0
19       }
20     ]
21   }
22 ]
```

Code Example 3.5: Definitions of the move immediate to register instruction for each register separately for the microcode generation.

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 operations are not generated by the microcode but are rather the 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 3.6 for example defines the alu operation

```

1  {
2      op: '000rsalu' # r = r x s (alu)
3      cycles: [
4          { # r x s into alu
5              aluYNWE: 0
6              reg0BusNOE: 's'
7              reg1BusNOE: '!s'
8              regAluSel: 'r'
9          }
10         { # alu into r
11             aluNOE: 0
12             reg0NWE: 'r'
13             reg1NWE: '!r'
14         }
15     ]
16 }
```

Code Example 3.6: Definition of the alu operation with two register arguments for the microcode generation.

with two registers and defines all 32 instructions with the op codes '00000000' to '00011111'.

```

1  {
2      op: '1010flag' # pc := imm
3      cycles: [
4          { # imm to pc
5              memPCNEn: 0
6              memPCLoadN: 0
7              memPCFromImm: 1
8          }
9      ]
10 }
```

Code Example 3.7: Definition of the branch instructions.

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

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)

on the current state of the ALU flags. For all other instructions, the flags are ignored and always executed¹. 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[7] 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 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 with it facilitate the application design.

3.2 Assembler

The second software that is probably even more important is the assembler. An assembler translates human readable instructions into the machine code, i.e. the

¹Meaning that all memory locations for the instruction and step counter, no matter the ALU flags, store the operation.

```

1  PRNG_SEED = 0x0000
2  SIMPLE_IO = 0xfe00
3
4  prng:
5      ldr r0, [PRNG_SEED]
6      subs r0, 0
7      beq prngDoEor
8      lsl r0, 1
9      beq prngNoEor
10     bcc prngNoEor
11 prngDoEor:
12     xor r0, 0x1d
13 prngNoEor:
14     str r0, [PRNG_SEED]
15     ret
16
17 start:
18     mov r0, 0
19     str r0, [PRNG_SEED]
20     prng_loop:
21         call prng
22         str r0, [SIMPLE_IO]
23     b prng_loop

```

Code Example 3.8: Pseudo Random Number Generator (PRNG) written in the EDiC Assembler.

bits that are stored in the instruction EEPROMs. For the EDiC each instruction is 24 bits wide, with 8 bits instruction op code and 8 or 16 bits immediate value. Even though assemblers usually only translate instructions one for 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 3.8 and 3.9 show the translation that the assembler does where Code Example 3.8 shows the assembler program that is written by any programmer and Code Example 3.9 summarizes what values are stored in the program EEPROM.

```
1 0x0000 - op: 10100000, imm: 0x000a - b 0x0a
2 0x0001 - op: 11110000, imm: 0x0000 - ldr r0, [0x00]
3 0x0002 - op: 10010001, imm: 0x0000 - subs r0, 0x00
4 0x0003 - op: 10100001, imm: 0x0007 - beq 0x07
5 0x0004 - op: 10000111, imm: 0x0001 - lsl r0, 0x01
6 0x0005 - op: 10100001, imm: 0x0008 - beq 0x08
7 0x0006 - op: 10100100, imm: 0x0008 - bcc 0x08
8 0x0007 - op: 10000100, imm: 0x001d - xor r0, 0x1d
9 0x0008 - op: 11110010, imm: 0x0000 - str r0, [0x00]
10 0x0009 - op: 10110001, imm: ----- - ret
11 0x000a - op: 11111000, imm: 0x0000 - mov r0, 0
12 0x000b - op: 11110010, imm: 0x0000 - str r0, [0x00]
13 0x000c - op: 10110000, imm: 0x0001 - call 0x01
14 0x000d - op: 11110010, imm: 0xfe00 - str r0, [0xfe00]
15 0x000e - op: 10100000, imm: 0x000c - b 0x0c
```

Code Example 3.9: The output of the PRNG of Code Example 3.8. The first 16 bits are the memory address, then 8 bits for the instruction op-code and 16 bits for the instruction immediate and for reference the original instruction with variables replaced.

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 with 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 Usually the first parameters from the caller to the callee are passed in registers, which avoids long memory operations for storing and loading the parameters. In the EDiC memory operations cannot stall and are, therefore, 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).

Return value The return value is to be place 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 to 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 them from the stack at the end of the function.

3.2.2 Available Instructions

First of all, this section summarizes all available instructions and which parameters they take. All instructions start with the operation and then up two parameters 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 0xffff (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`
- `and`
- `xor`
- `lsr`
- `sub`
- `eor`
- `xnor`
- `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.

- Two registers

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

- One register and one register as memory address

`lsr r1, [r0]` does: $r_1 := r_1 \gg \text{mem}[r_0]$ (section 3.2.2.2)

- One register and an immediate value

`and r0, 0x0f` does: $r_0 := r_0 \vee 15$

- One register and an immediate value as memory address

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

All of 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 is used later on. 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.

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

compares the `r0` register with the value 10 and if $r_0 < 10$ 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 | • sts | • stf | • sma |
| • ldr | • lds | • ldf | |

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 an 16bit 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 compiler makes sure that the addresses upper 8 bits are `0xff` to always access the stack.

The `ldf` and `stf` functions 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:

<code>ldr r0, [0xabba]</code>	Loads the value from address 0xabba into r0
<code>str r1, [0xc0de]</code>	Stores the value in r1 to address 0xc0de
<code>sma 0xca</code>	
<code>mov r0, 0xfe</code>	Loads the value from address 0xcafe into r0
<code>ldr r0, [r0]</code>	
<code>lds r1, [0x42]</code>	Loads the value from address 0xff42 which is translated into 0x{sp}42 into r1
<code>stf r0, [0xab]</code>	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:

<code>mov r0, 0xda</code>	Sets <code>r0</code> to <code>0xda</code>
<code>mov r1, r0</code>	Copies the value of <code>r0</code> to <code>r1</code>
<code>cmp r0, 10</code> <code>blt 0x42</code>	Branches to address (sets the PC to) <code>0x42</code> if the value of <code>r0</code> is smaller than 10
<code>call 0x100</code>	Calls a function at address <code>0x100</code>
<code>ret</code>	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 3.8 and 3.10 both declare a common constant that is used exactly like in higher level languages. Each instruction, taken an immediate value can instead specify the name of the constant and the value of the constant is then used instead. In Code Example 3.10 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 sent 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 assemblers are a huge convenience. 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

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

Code Example 3.10: The PRNG of Code Example 3.8 with the constants and labels resolved.

call instructions to jump to the instruction followed by the label declarations. As 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 3.9 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 later be used. Differently to the value constants of section 3.2.3.1 strings cannot be used as parameters to 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 3.11 line 4 with the syntax `<address>.<name> = "<value>"`. In the example a string constant with the name “LOST_STRING” is defined to have the content “You lost!!! Score: ” at the address `0x20`. The EDiC assembler treats a string as an NULL-terminated array of characters which are characters stored consecutively and after the last character a NULL-byte is stored to signal the end of the string. The address of a string constant 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 `0x2013` (18 characters plus 1 NULL-byte) and `mov r0, LOST_STRING` in line 9 is equivalent to `mov r0, 0x20`. As the assembler has no direct control over the memory contents as for example the ARM assembler, each string declarations results in two instructions per character that are inserted at the start of the program² as shown in Code Example 3.12.

Code Example 3.11 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`

²Before the `b start` instruction that is inserted when a start label exists.

```

1  include "prng.s"
2  include "uart_16c550.s"
3  0x20.LOST_STRING = "You lost!!! Score: "
4  lost:
5      // [...]
6      // output the lost string
7      mov r0, LOST_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 3.11: Excerpts of the Snake assembler program used in the demo in figure 1.1.

```
1  mov r0, 0x59 // 'Y'
2  str r0, [0x2000]
3  mov r0, 0x6f // 'o'
4  str r0, [0x2001]
5  mov r0, 0x75 // 'u'
6  str r0, [0x2002]
7  mov r0, 0x20 // ' '
8  str r0, [0x2003]
9  mov r0, 0x6c // 'l'
10 str r0, [0x2004]
11 mov r0, 0x6f // 'o'
12 str r0, [0x2005]
13 mov r0, 0x73 // 's'
14 str r0, [0x2006]
15 mov r0, 0x74 // 't'
16 str r0, [0x2007]
17 mov r0, 0x21 // '!'
18 str r0, [0x2008]
19 mov r0, 0x21 // '!'
20 str r0, [0x2009]
21 mov r0, 0x21 // '!'

22 str r0, [0x200a]
23 mov r0, 0x20 // ' '
24 str r0, [0x200b]
25 mov r0, 0x53 // 'S'
26 str r0, [0x200c]
27 mov r0, 0x63 // 'c'
28 str r0, [0x200d]
29 mov r0, 0x6f // 'o'
30 str r0, [0x200e]
31 mov r0, 0x72 // 'r'
32 str r0, [0x200f]
33 mov r0, 0x65 // 'e'
34 str r0, [0x2010]
35 mov r0, 0x3a // ':'
36 str r0, [0x2011]
37 mov r0, 0x20 // ' '
38 str r0, [0x2012]
39 mov r0, 0x0 // NULL-byte
40 str r0, [0x2013]
41 mov r0, 0 // restore r0
```

Code Example 3.12: The instructions resulting from the string definition of Code Example 3.11 line 4.

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.

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. This can for example be used to write a library utility 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 3.11 lines 1 and 2, the EDiC assembler supports the `include` keyword followed by a relative 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 used as is with some exceptions:

- The start label of all included files are discarded and the main file is required to

```

1  include "prng.s"
2  include "uart_16c550.s"
3  0x20.LOST_STRING = "You lost!!! Score: "
4  lost:
5  ..//[ ... ]
6  ..//output the lost string
7  ..mov r0, LOST_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

```

Figure 3.1: The syntax highlighting with the EDiC Visual Studio Code Extension and the Atom One Light Theme [1].

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.

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 correctly 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 3.11 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).

As Visual Studio Code [15] is one of the leading extensible code editors, an extension for EDiC assembler has been developed and published [19]. The code of the Code Example 3.11 is shown again in figure 3.1 as it is highlighting using the developed extension. The extension itself mainly consists of a TextMate language definition [13] 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 FPGAs 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 [2]. Today's FPGAs can have several million logic elements with several hundred MBs of Block RAM (BRAM), more than 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 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 different 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

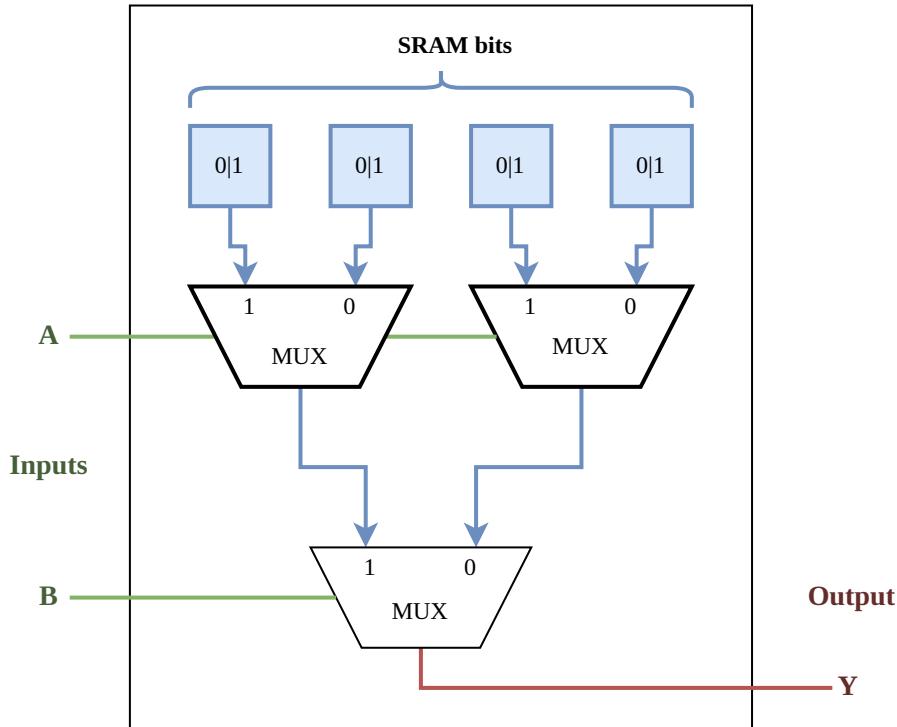


Figure 4.1: Internal structure of a 2-bit LUT

a NAND function, the 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 available that are very well suited for a prototype for the EDiC.

4.2 FPGA choices

For the EDiC the Nexys A7 development board [8] with the AMD-Xilinx Artix 7 XC7A100T-1CSG324C FPGA has been chosen. Its synthesis tool is the AMD-Xilinx Vivado [22] which is available as a free version and includes an advanced simulation environment.

```

65 | assign s_cin[0] = i_ctrlAluSub;
66 | for (i = 0; i < 8; i=i+1) begin
67 |     assign s_yXor[i] = i_a[i] ^ s_b[i];
68 |     assign s_yAnd[i] = i_a[i] & s_b[i];
69 |     assign s_yAdder[i] = s_cin[i] ^ s_yXor[i];
70 |     assign s_cin[i + 1] = s_yAnd[i] | (s_cin[i] & s_yXor[i]);
71 | end

```

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

4.2.1 Language Choice

There are two main Hardware Description Languages (HDLs): Verilog and VH-SIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL). Both are widely supported and used 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 [16]. 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 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 tristate module which then has only one output which represents the value of the net. If none of the driver have an active output enable, the output is 0xff; if one of the driver has an active output enable, the output represents its value and if more than one driver have an active output enable, an error is raised. The modules logic representation for a Tri-State net with two inputs is shown in figure 4.2. The o_noe

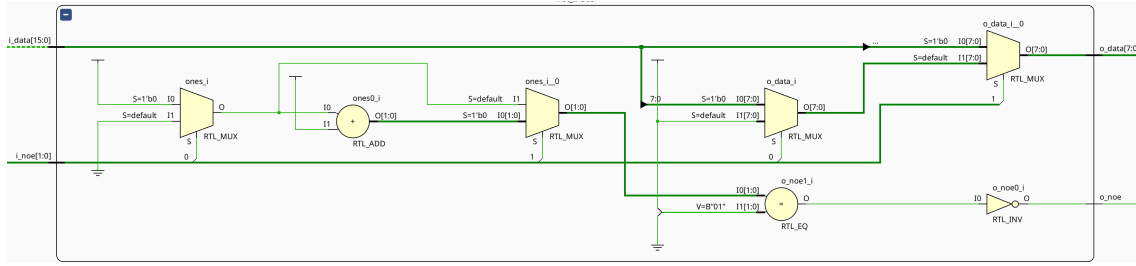


Figure 4.2: The elaborated Tri-State module with two 8 bit inputs.

```

1  mov r1, 0x12
2  add r1, 0x2f
3  end:
4  b end

```

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

is only active (low) if exactly one input `i_noe` is active (low) and depending it, the data output is selected. For this FPGA it is implemented with one LUT4 primitive per output data bit.

4.3 Behavioral Implementation

Two kinds of FPGA were developed in the process. 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 because 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 μ s 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 μ s 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 and then 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. Here, a tool was written which converts a given netlist into a Verilog file which can be compiled and synthesized by Vivado.

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 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, 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 `PCIN0` 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 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 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 is 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’) save

```

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

```

Code Example 4.3: An Electronic Design Interchange Format (EDIF) definition of an instance as exported by OrCAD/CAPTURE.

```

1
2  (net PCIN0
3    (joined
4      (portRef &18 (instanceRef U52))
5      (portRef &18 (instanceRef U51))
6      (portRef &3 (instanceRef U54)))
7    (property Name (string "PCIN0")))

```

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

```
1 // quad and https://www.ti.com/lit/ds/symlink/sn74ls08.pdf
2 module ic74x08(
3     input wire port1,
4     input wire port2,
5     output wire port3,
6     input wire port4,
7     input wire port5,
8     output wire port6,
9     input wire port7,
10    output wire port8,
11    input wire port9,
12    input wire port10,
13    output wire port11,
14    input wire port12,
15    input wire port13,
16    input wire port14
17 );
18
19 assign port3 = port1 & port2;
20 assign port6 = port4 & port5;
21 assign port8 = port9 & port10;
22 assign port11 = port12 & port13;
23
24 endmodule
```

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

for $0\ \Omega$ resistors. The schematic includes some $0\ \Omega$ 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 $0\ \Omega$ 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 EEPROMs

The 6 EEPROMs (3 for the instructionROM and 3 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 Xilinx ROM IP Cores which are configured with the respective initial values. The addresses for one ROM instantiations are used and then all 24 data ports from the


```

1364 microCodeRom inst_microCodeRom (
1365     .clka(i_asyncEEPROMSpecialClock),
1366     .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}),
1367     .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,
↪ CTRLMEMMAROWE_SRC, CTRLMEMINSTROE_SRC, CTRLMEMINSTRWE_SRC,
↪ CTRLMEMSPEN_SRC, CTRLMEMSPUP_SRC, CTRLMEMPCLOAD_SRC,
↪ CTRLREG1BUSOE_SRC, CTRLREGOBUSOE_SRC, CTRLREGALUSEL_SRC,
↪ CTRLREG1WE_SRC, CTRLREGOWE_SRC, CTRLALUOE_SRC,
↪ CTRLALUYWE_SRC})
1368 );

```

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

```

1794 tristatenet #(
1795     .INPUT_COUNT(2)
1796 ) inst_triBusPCINO (
1797     .i_data({PCINO_U51, PCINO_U52}),
1798     .i_noe({U51_b_noe, U52_b_noe}),
1799     .o_data(PCINO),
1800     .o_noe(PCINO_noe)
1801 );

```

Code Example 4.7: Verilog instantiation for the Tri-State Net PCINO.

3 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 tristatenet 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 an input to the tristatenet instance and a new net is created for each tristatenet which is the actual value of the net (the output of the tristatenet

module). The tristatenet for the `PCIN0` signal (Code Example 4.4) is represented by the instantiation shown in Code Example 4.7.

5 Hardware Design

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

Register IC The 74 series of logic ICs feature many different registers. The most basic register IC has n D-type flip-flops with respective data inputs and outputs plus one common clock input. On each rising edge of the clock the flip-flops capture the input values and hold them until the next rising edge of the clock. However, often it is required that a register does not capture on every rising edge of the clock. This is done with an additional input, called clock enable. In the first version of the CPU the clock inputs of the registers that needed clock enable were connected to the output of an AND gate of the clock and a control bit. This has the major drawback that glitches of the enable control signal can propagate to the clock input of the register when the clock is currently high. There are two widely used alternatives to the simple AND gate: The enable input can be used as the select input for an multiplexer to the data input of the flip flop, where it multiplexes between the actual input and the current output. This allows the flip-flop to always capture data but when the enable input is inactive, it recaptures the current output. The drawbacks are that each bit of the register needs a multiplexer at the input and secondly that the flip-flops draw power on every clock pulse, even though no data is captured. The *74F825* logic IC solves this with the circuit shown in figure 5.1. When the $\overline{\text{EN}}$ input is low, the CP input is NAND gate on the right passed the negated CP through¹. When the $\overline{\text{EN}}$ input is high, on the other hand, the output does not change. This

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

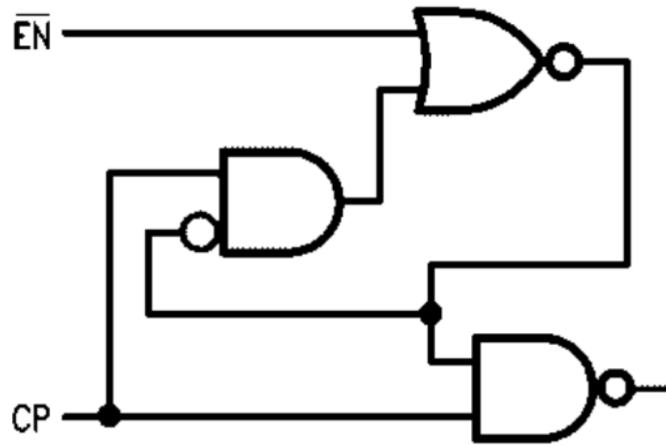


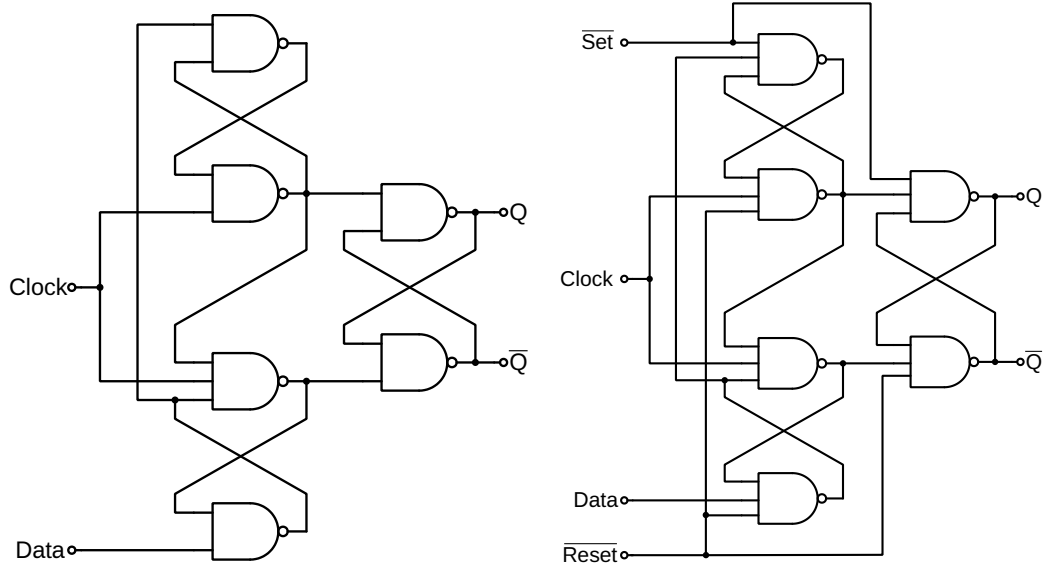
Figure 5.1: Clock Enable circuit of the *74F825* IC [5].

circuit prevents the \overline{EN} to trigger a falling edge (which would trigger the flip-flops) on the CP output. However, when the \overline{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{EN} input which are discussed in more detail in section 5.1. As the registers store the current state of execution, it is required that the registers start up to a known state. Therefore, some registers feature a clear input (or set input) which forces all flip-flops to 0 (or 1). This is usually accomplished by modifying the classical D-type flip-flop to allow for setting and resetting the internal \overline{SR} NAND latches as shown in figure 5.2.

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

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

5.1 Timing Analysis



(a) Classical D-type flip-flop built out of three \overline{SR} NAND latches [17].

(b) D-type flip-flop modified to support $\overline{\text{Clear}}$ and $\overline{\text{Set}}$ [21].

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

6 Initial Hardware Test & Component Verification

6.0.1 Test Adapter

7 Conclusion and Future Work

Acronyms

Notation	Description
ALU	Arithmetic Logic Unit i, 7
ASIC	Application-Specific Integrated Circuit 41
BRAM	Block RAM 41
CISC	Complex Instruction Set Computer 5
CLB	Configurable Logic Block 41
CPU	Central Processing Unit 1
CSON	CoffeeScript-Object-Notation 23
DSP	Digital Signal Processor 41
EDiC	Educational Digital Computer 1
EDIF	Electronic Design Interchange Format 47
EEPROM	Electrically Erasable Programmable Read-Only Memory ii, 6
EPROM	Erasable Programmable Read-Only Memory 41
FPGA	Field Programmable Gate Array ii, 1
HDL	Hardware Description Language 43
IC	Integrated Circuit 1
IDE	Integrated Development Environment 39
JSON	JavaScript Object Notation 23
LED	Light-Emitting Diode 51
LSB	Least Significant Bit 12

Notation	Description
LUT	Lookup Table 41
MAR	Memory Address Register 14
MSB	Most Significant Bit 9
MUX	Multiplexer 41
NOP	No Operation 20
PC	Program Counter i, 6
PCB	Printed Circuit Board 2
PRNG	Pseudo Random Number Generator 29
RISC	Reduced Instruction Set Computer 6
SP	Stack Pointer 13
SRAM	Static Random-Access Memory 5
TUB	Technical University Berlin 43
UART	Universal Asynchronous Receiver-Transmitter 66
VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language 43

List of Figures

1.1	The final version of the EDiC playing Snake on a VT-100 over an RS-232 I/O card.	2
1.2	The first version of the CPU in its final state.	3
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.	9
2.2	8 bit bidirectional barrel shifter.	10
3.1	The syntax highlighting with the EDiC Visual Studio Code Extension and the Atom One Light Theme [1].	39
4.1	Internal structure of a 2-bit LUT	42
4.2	The elaborated Tri-State module with two 8 bit inputs.	44
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). . .	45
5.1	Clock Enable circuit of the 74F825 IC [5].	52
5.2	Comparison of D-type flip-flops with and without $\overline{\text{Clear}}$ and $\overline{\text{Set}}$	53

List of Tables

2.1	Summary of the available alu operations.	8
3.1	All available branch instructions with their op-code and microcode translation based on the ALU flags explained in section 2.2.1.	28

List of Code Examples

3.1	Schema of the Microcode Definition CSON-File [3] as a TypeScript [14] Type definition.	24
3.2	Example of a control signal definition for the microcode generation. .	24
3.3	Definition of the instruction fetch and decode steps for the microcode generation.	25
3.4	Definition of the move immediate to register instruction for the microcode generation.	26
3.5	Definitions of the move immediate to register instruction for each register separately for the microcode generation.	26
3.6	Definition of the alu operation with two register arguments for the microcode generation.	27
3.7	Definition of the branch instructions.	27
3.8	PRNG written in the EDiC Assembler.	29
3.9	The output of the PRNG of Code Example 3.8. The first 16 bits are the memory address, then 8 bits for the instruction op-code and 16 bits for the instruction immediate and for reference the original instruction with variables replaced.	30
3.10	The PRNG of Code Example 3.8 with the constants and labels resolved.	35
3.11	Excerpts of the Snake assembler program used in the demo in figure 1.1.	37
3.12	The instructions resulting from the string definition of Code Example 3.11 line 4.	38
4.1	Behavioral Verilog Description of the Adder (including XOR and AND) of the ALU module.	43
4.2	The code for the waveform example of figure 4.3.	44
4.3	An EDIF definition of an instance as exported by OrCAD/CAPTURE.	47
4.4	An EDIF definition of a net as exported by OrCAD/CAPTURE. . .	47
4.5	Verilog implementation for the 74F08 IC.	48
4.6	Verilog instantiation of the microcode ROM generated out of three EEPROM instantiations.	49
4.7	Verilog instantiation for the Tri-State Net PCIN0.	49

A.1	The full snake assembler program.	69
A.2	The PRNG assembler program “prng.s” used in the snake program in Code Example A.1.	77
A.3	The utility library for the Universal Asynchronous Receiver- Transmitter (UART) extension card of the EDiC with the 16c550 UART Transceiver.	78

Bibliography

- [1] Mahmoud Ali. *Visual Studio Code - One Light Theme based on Atom*. 2022. URL: <https://marketplace.visualstudio.com/items?itemName=akamud.vscode-theme-onelight>.
- [2] Altera. *User-Configurable Logic Databook*. (WWW). 1988. URL: http://www.bitsavers.org/components/altera/_dataBooks/1988_Altera_Data_Book.pdf.
- [3] bevry. *CoffeeScript-Object-Notation*. 2022. URL: <https://github.com/bevry/cson>.
- [4] Crystal Chen, Greg Novick, and Kirk Shimano. *RISC Architecture*. 2000. URL: <https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/riscisc/>.
- [5] 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>.
- [6] *CVE-2017-5753*. 2018. URL: <https://www.cve.org/CVERecord?id=CVE-2017-5753>.
- [7] arm Developer. *ARM Developer Suite Assembler Guide - Conditional execution*. URL: <https://developer.arm.com/documentation/dui0068/b/ARM-Instruction-Reference/Conditional-execution>.
- [8] Digilent. *Nexys A7 Reference*. 2022. URL: <https://digilent.com/reference/programmable-logic/nexys-a7/start>.
- [9] *Git Repository of the EDiC development*. URL: <https://github.com/Nik-Sch/EDiC>.
- [10] Texas Instruments. *SNx4LS245 Octal Bus Transceivers With 3-State Outputs*. 1976. URL: <https://www.ti.com/lit/ds/sdls146b/sdls146b.pdf>.

- [11] 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>.
- [12] ECMA International. *The JSON Data Interchange Syntax*. 2017. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf.
- [13] MacroMates Ltd. *Language Grammars — TextMate 1.x Manual*. 2021. URL: https://macromates.com/manual/en/language_grammars.
- [14] Microsoft. *TypeScript: JavaScript With Syntax For Types*. 2022. URL: <https://www.typescriptlang.org/>.
- [15] Microsoft. *Visual Studio Code - Code Editing. Redefined*. 2022. URL: <https://code.visualstudio.com/>.
- [16] 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>.
- [17] Nolanjshettle. *File:Edge triggered D flip flop.svg*. 2013. URL: https://en.wikipedia.org/wiki/File:Edge_triggered_D_flip_flop.svg.
- [18] David Patterson and John LeRoy Hennessy. *Rechnerorganisation und Rechnerentwurf: Die Hardware/Software-Schnittstelle*. eng. De Gruyter Studium. De Gruyter, 2016. ISBN: 3110446057.
- [19] Niklas Schelten. *EDiC support for Visual Studio Code*. 2022. URL: <https://github.com/Nik-Sch/EDiC-vscode-syntax>.
- [20] Philips Semiconductors. *74ABT540 Octal buffer, inverting (3-State)*. 1998. URL: <https://www.mouser.com/datasheet/2/302/74ABT540-62406.pdf>.
- [21] 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.
- [22] AMD - Xilinx. *Vivado ML Editions - Free Vivado Standard Edition*. AMD - Xilinx. 2022. URL: <https://www.xilinx.com/products/design-tools/vivado.html>.

A Collection of assembler programs for the EDiC

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

```
1  include "prng.s"
2  include "uart_16c550.s"
3
4  SIMPLE_IO = 0xfe00
5  UART_RX_EMPTY = 0xfe09
6  UART_TX_FULL = 0xfe0a
7  UART_DATA = 0xfe0b
8  PAR1 = 0xff00
9  PAR2 = 0xff01
10 PAR3 = 0xff02
11 ESCAPE0 = 0x1b // \033
12 ESCAPE1 = 0x5b // '['
13 BORDER = 0x23 // '#'
14 SPACE = 0x20 // ' '
15 HEAD = 0x40 // '@'
16 LEFT = 0x3c // '<'
17 RIGHT = 0x3e // '>'
18 UP = 0x5e // '^'
19 DOWN = 0x76 // 'v'
20 ITEM = 0x58 // 'X'
21 ASCII_W = 0x77
22 ASCII_A = 0x61
23 ASCII_S = 0x73
24 ASCII_D = 0x64
25 ASCII_CAPITAL_W = 0x57
26 ASCII_CAPITAL_A = 0x41
27 ASCII_CAPITAL_S = 0x53
28 ASCII_CAPITAL_D = 0x44
29
30 // global variables
31 SNAKE_LENGTH = 0x0000
32 SNAKE_DIRECTION = 0x0001
33 SNAKE_HEAD_LINE = 0x0002
34 SNAKE_HEAD_COL = 0x0003
35 SNAKE_TAIL_LINE = 0x0004
36 SNAKE_TAIL_COL = 0x0005
37 SNAKE_LEFT_LINE = 0x0006
38 SNAKE_LEFT_COL = 0x0007
39 PRNG_SEED = 0x0008 //
   ↳ do not init for extra
   ↳ randomness
40
41 // local variables
42 LINE_COUNTER = 0xff00
43 COLUMN_COUNTER = 0xff01
44
45
46 // screen is in memory
   ↳ starting from 0x0100
```

```
47 // one line has 256 bytes for      83
   ↪ ease of access                  84
48 LINES = 24                         85 // move cursor to the top
49 COLUMNS = 80                     86 // mov r0, 1 // line
50 COLUMNS_1 = 79                   87 // stf r0, [PAR2]
51                                   88 // mov r0, 0 // col
52 0x20.LOST_STRING = "You           89 // stf r0, [PAR1]
   ↪ lost!!! Score: "              90 // mov r0, BORDER
53                                   91 // call setScreen
54 start:                             92 ldr r0, [SNAKE_LENGTH]
55     call uart_init                 93 str r0, [SIMPLE_IO]
56     // clear screen               94 // wait x ms
57     mov r0, ESCAPE0               95 // mov r0, 90
58     call uart_write               96 // call delay_ms
59     mov r0, ESCAPE1               97
60     call uart_write               98 call readArrow
61     mov r0, 0x32 // '2'           99 // change direction if !=
62     call uart_write              100 ↪ -1
63     mov r0, 0x4a // 'J'          101 cmp r0, -1
64     call uart_write              102 beq mainLoop
65                                   103 str r0, [SNAKE_DIRECTION]
66     call createBoard              104 b mainLoop
67     call updateItem              105
68 mainLoop:                         106 lost:
69     call updateHead              107 // set position to upper
70     cmp r0, -1                   108 ↪ center
71     beq lost                     109 mov r0, 6 // line
72     cmp r0, 1                   110 stf r0, [PAR2]
73     beq mainAteItem              111 mov r0, 27 // col
74     call updateTail              112 stf r0, [PAR1]
75     b mainUpdateBoard            113 mov r0, SPACE
76                                   114 call setScreen
77 mainAteItem:                     115 mov r0, LOST_STRING
78     ldr r0, [SNAKE_LENGTH]        116 call outputString
79     add r0, 1                    117 ldr r0, [SNAKE_LENGTH]
80     str r0, [SNAKE_LENGTH]        118 call outputDecimal
81     call updateItem
82 mainUpdateBoard:
    ldr r0, [SNAKE_LENGTH]
```

<pre> 119 120 updateItem: 121 str r1, [0xfffe] 122 123 itemColumn: 124 call prng 125 and r0, 0x7f // limit 126 ↳ columns 127 cmp r0, COLUMNS 128 bhs itemColumn // if out 129 ↳ of scope redo 130 mov r1, r0 131 itemLine: 132 call prng 133 and r0, 0x1f // limit 134 ↳ lines 135 cmp r0, LINES 136 bgt itemLine // if out of 137 ↳ scope redo 138 stf r0, [PAR2] 139 sma r0 // line 140 ldr r0, [r1] 141 cmp r0, SPACE 142 bne itemColumn // if there 143 ↳ is something at the new 144 ↳ item position find a 145 ↳ new one 146 // store new item 147 stf r1, [PAR1] 148 mov r0, ITEM 149 call setScreen 150 151 ldr r1, [0xfffe] 152 ret 153 154 // returns -1 if lost, 0 if 155 ↳ nothing happend and 1 if 156 ↳ ate item </pre>	<pre> 157 updateHead: 158 str r1, [0xfffe] 159 160 ldr r0, [SNAKE_HEAD_LINE] 161 stf r0, [PAR2] 162 sma r0 163 ldr r0, [SNAKE_HEAD_COL] 164 stf r0, [PAR1] 165 // load correct direction 166 ↳ char into r0 167 ldr r1, [SNAKE_DIRECTION] 168 cmp r1, 0 169 beq headUp 170 cmp r1, 1 171 beq headDown 172 cmp r1, 2 173 beq headRight 174 cmp r1, 3 175 beq headLeft 176 b headEnd // should not 177 ↳ happen 178 179 headUp: 180 mov r0, UP 181 call setScreen 182 ldr r0, [SNAKE_HEAD_LINE] 183 sub r0, 1 184 str r0, [SNAKE_HEAD_LINE] 185 b headEnd 186 187 headDown: 188 mov r0, DOWN 189 call setScreen 190 ldr r0, [SNAKE_HEAD_LINE] 191 add r0, 1 192 str r0, [SNAKE_HEAD_LINE] 193 b headEnd </pre>
---	--

184	headLeft:	219	ldr r1, [0xfffe]
185	mov r0, LEFT	220	ret
186	call setScreen	221	headSpace:
187	ldr r0, [SNAKE_HEAD_COL]	222	mov r0, 0
188	sub r0, 1	223	ldr r1, [0xfffe]
189	str r0, [SNAKE_HEAD_COL]	224	ret
190	b headEnd	225	headItem:
191		226	mov r0, 1
192	headRight:	227	ldr r1, [0xfffe]
193	mov r0, RIGHT	228	ret
194	call setScreen	229	
195	ldr r0, [SNAKE_HEAD_COL]	230	updateTail:
196	add r0, 1	231	str r1, [0xfffe]
197	str r0, [SNAKE_HEAD_COL]	232	
198	b headEnd	233	ldr r0, [SNAKE_TAIL_LINE]
199		234	str r0, [SNAKE_LEFT_LINE]
200	headEnd:	235	stf r0, [PAR2]
201		236	sma r0
202	ldr r1, [SNAKE_HEAD_LINE]	237	ldr r0, [SNAKE_TAIL_COL]
203	stf r1, [PAR2]	238	str r0, [SNAKE_LEFT_COL]
204	sma r1	239	stf r0, [PAR1]
205	ldr r1, [SNAKE_HEAD_COL]	240	// load direction char
206	stf r1, [PAR1]	241	ldr r1, [r0]
207	ldr r1, [r1] // load item	242	mov r0, SPACE
	↪ at new position	243	call setScreen
208	sts r1, [0x00]	244	cmp r1, UP
209	// store & show head	245	beq tailUp
210	mov r0, HEAD	246	cmp r1, DOWN
211	call setScreen	247	beq tailDown
212	// if new position is not	248	cmp r1, RIGHT
	↪ space or item -> lost	249	beq tailRight
213	lds r0, [0x00] // load	250	cmp r1, LEFT
	↪ saved item	251	beq tailLeft
214	cmp r0, SPACE	252	b tailEnd // should not
215	beq headSpace		↪ happen
216	cmp r0, ITEM	253	
217	beq headItem	254	tailUp:
218	mov r0, -1	255	ldr r1, [SNAKE_TAIL_LINE]

<pre> 256 sub r1, 1 257 str r1, [SNAKE_TAIL_LINE] 258 b tailEnd 259 260 tailDown: 261 ldr r1, [SNAKE_TAIL_LINE] 262 add r1, 1 263 str r1, [SNAKE_TAIL_LINE] 264 b tailEnd 265 266 tailLeft: 267 ldr r1, [SNAKE_TAIL_COL] 268 sub r1, 1 269 str r1, [SNAKE_TAIL_COL] 270 b tailEnd 271 272 tailRight: 273 ldr r1, [SNAKE_TAIL_COL] 274 add r1, 1 275 str r1, [SNAKE_TAIL_COL] 276 b tailEnd 277 278 tailEnd: 279 ldr r1, [0xfffe] 280 ret 281 282 createBoard: 283 str r0, [0xfffe] 284 str r1, [0xfffd] 285 286 // init snake 287 mov r0, 4 288 str r0, [SNAKE_LENGTH] 289 mov r0, 2 290 str r0, [SNAKE_DIRECTION] 291 mov r0, 12 // center 292 str r0, [SNAKE_HEAD_LINE] 293 mov r0, 40 #center </pre>	<pre> 294 str r0, [SNAKE_HEAD_COL] 295 mov r0, 12 296 str r0, [SNAKE_TAIL_LINE] 297 mov r0, 37 298 str r0, [SNAKE_TAIL_COL] 299 mov r0, 12 300 str r0, [SNAKE_LEFT_LINE] 301 mov r0, 36 302 str r0, [SNAKE_LEFT_COL] 303 304 // move to home position 305 mov r0, ESCAPE0 306 call uart_write 307 mov r0, ESCAPE1 308 call uart_write 309 mov r0, 0x48 // 'H' 310 call uart_write 311 312 313 // first and last line is 314 ↪ full border 315 mov r1, 0 316 createLine0Loop: 317 sma 1 318 mov r0, BORDER 319 str r0, [r1] 320 call uart_write 321 add r1, 1 322 cmp r1, COLUMNS 323 blt createLine0Loop 324 325 mov r0, 0x0a // LF 326 call uart_write 327 mov r0, 0x0d // CR 328 call uart_write 329 330 // line 2 to 23 have first 331 ↪ and last column border </pre>
---	---

```
330    mov r1, 2 // skip first          364    blt createLineLoop // skip
      ↪ line                          ↪ last line
331    str r1, [LINE_COUNTER]          365
332    createLineLoop:                 366    // draw last line
333    // load mar1 with line          367    mov r1, 0
      ↪ space                          368    createLineLastLoop:
334    sma r1                          369    sma LINES
335    mov r1, 0                       370    mov r0, BORDER
336    mov r0, BORDER                  371    str r0, [r1]
337    str r0, [r1]                    372    call uart_write
338    call uart_write                 373    add r1, 1
339    add r1, 1                       374    cmp r1, COLUMNS
340    // loop through line            375    blt createLineLastLoop
      ↪ (1-79) and store space         376
341    createColumnLoop:               377    // draw snake
342    ldr r0, [LINE_COUNTER]           378    ldr r0, [SNAKE_HEAD_LINE]
343    sma r0                           379    stf r0, [PAR2]
344    mov r0, SPACE                    380    ldr r0, [SNAKE_HEAD_COL]
345    str r0, [r1]                    381    stf r0, [PAR1]
346    call uart_write                 382    mov r0, HEAD
347    add r1, 1                       383    call setScreen
348    cmp r1, COLUMNS_1              384
349    blt createColumnLoop            385    mov r1, 1
350    // store end border              386    snakeBody:
351    mov r0, BORDER                  387    ldr r0, [SNAKE_HEAD_LINE]
352    str r0, [r1]                    388    stf r0, [PAR2]
353    call uart_write                 389    ldr r0, [SNAKE_HEAD_COL]
354                                    390    sub r0, r1
355    mov r0, 0x0a // LF               391    stf r0, [PAR1]
356    call uart_write                 392    mov r0, RIGHT
357    mov r0, 0x0d // CR              393    call setScreen
358    call uart_write                 394    add r1, 1
359                                    395    cmp r1, 3
360    ldr r1, [LINE_COUNTER]           396    ble snakeBody
361    add r1, 1                        397
362    str r1, [LINE_COUNTER]           398    ldr r0, [0xfffe]
363    cmp r1, LINES                    399    ldr r1, [0xfffd]
                                      400    ret
```

<pre> 401 402 403 // r0: char, PAR1: col, PAR2: ↳ line 404 setScreen: 405 str r0, [0xfffe] 406 str r1, [0xfffd] 407 408 // store 409 ldr r1, [PAR2] 410 sma r1 411 ldr r1, [PAR1] 412 str r0, [r1] 413 414 // decimal needs to be one ↳ based 415 mov r0, ESCAPE0 416 call uart_write 417 mov r0, ESCAPE1 418 call uart_write 419 ldr r0, [PAR2] // line is ↳ already one based 420 call outputDecimal 421 mov r0, 0x3b // ';' 422 call uart_write 423 ldr r0, [PAR1] 424 add r0, 1 // column is not ↳ one based 425 call outputDecimal 426 mov r0, 0x48 // 'H' 427 call uart_write 428 429 ldr r0, [0xfffe] 430 call uart_write 431 432 ldr r1, [0xfffd] 433 434 ret </pre>	<pre> 435 436 // r0 is parameter 437 outputDecimal: 438 str r1, [0xfffe] 439 440 mov r1, 100 441 stf r1, [PAR1] 442 call divMod // r0 / 100 443 ldf r1, [PAR1] // mod ↳ result 444 add r0, 0x30 // make to ↳ char 445 call uart_write 446 mov r0, r1 // remainder is ↳ parameter for next ↳ divMod 447 mov r1, 10 448 stf r1, [PAR1] 449 call divMod 450 ldf r1, [PAR1] 451 add r0, 0x30 // make to ↳ char 452 call uart_write 453 mov r0, r1 // last char to ↳ output 454 add r0, 0x30 // make to ↳ char 455 call uart_write 456 457 ldr r1, [0xfffe] 458 ret 459 460 // r0: address of string 461 outputString: 462 str r1, [0xfffe] 463 sts r0, [0x00] 464 mov r1, 0 465 outputStringLoop: </pre>
--	--

```

466     lds r0, [0x00]
467     sma r0
468     ldr r0, [r1]
469     cmp r0, 0
470     beq outputStringEnd
471     call uart_write
472     add r1, 1
473     cmp r1, 255
474     bne outputStringLoop
475
476 outputStringEnd:
477
478     ldr r1, [0xfffe]
479 ret
480
481 // r0 / PAR1
482 // result: r0 -> div, *PAR1 ->
483     ↪ mod
484 divMod:
485     str r1, [0xfffe]
486     mov r1, 0
487     divLoop:
488         add r1, 1
489         sub r0, [PAR1]
490         bpl divLoop // positive or
491             ↪ zero (N Clear)
492 // executing one step too
493     ↪ much, undo it
494
495     add r0, [PAR1]
496     sub r1, 1
497
498     str r0, [PAR1]
499     mov r0, r1
500     ldr r1, [0xfffe]
501
502 ret
503 // r0 is return value:
504
505 // -1 for nothing, 0 for up,
506     ↪ 1 for down, 2 for right,
507     ↪ 3 for left
508 readArrow:
509     str r1, [0xfffe]
510 readArrowLoop:
511     call uart_read
512     cmp r0, 0
513     beq readArrowNothing // no
514         ↪ char received
515 // up
516     cmp r0, ASCII_W
517     beq readArrowUp
518     cmp r0, ASCII_CAPITAL_W
519     beq readArrowUp
520 // left
521     cmp r0, ASCII_A
522     beq readArrowLeft
523     cmp r0, ASCII_CAPITAL_A
524     beq readArrowLeft
525 // down
526     cmp r0, ASCII_S
527     beq readArrowDown
528     cmp r0, ASCII_CAPITAL_S
529     beq readArrowDown
530 // right
531     cmp r0, ASCII_D
532     beq readArrowRight
533     cmp r0, ASCII_CAPITAL_D
534     beq readArrowRight
535
536     cmp r0, ESCAPE0
537     bne readArrowLoop // make
538         ↪ sure to empty the fifo
539
540     call uart_read
541     cmp r0, 0
542     beq readArrowNothing

```

<pre> 535 cmp r0, ESCAPE1 536 bne readArrowLoop 537 538 call uart_read 539 cmp r0, 0x41 // A 540 blt readArrowLoop 541 cmp r0, 0x44 // D 542 bgt readArrowLoop 543 sub r0, 0x41 // return 0-4 544 ret 545 // -1 for nothing, 0 for up, 546 ↪ 1 for down, 2 for right, 547 ↪ 3 for left 548 readArrowNothing: 549 ldr r1, [0xfffe] 550 mov r0, -1 551 ret 552 readArrowUp: 553 ldr r1, [0xfffe] 554 mov r0, 0 555 ret 556 readArrowLeft: 557 ldr r1, [0xfffe] 558 mov r0, 3 559 ret 560 readArrowDown: 561 ldr r1, [0xfffe] 562 mov r0, 1 563 ret 564 readArrowRight: 565 ldr r1, [0xfffe] </pre>	<pre> 564 mov r0, 2 565 ret 566 567 // r0: delay in ms 568 delay_ms: 569 sts r0, [0x00] 570 571 delay_ms_outer_loop: 572 573 // 2MHz clock -> 1ms is 574 ↪ 2000cycle 575 // per loop 4+4+3+3=14 576 ↪ cycles (below) 577 // -> 198.6 times 10 578 ↪ cycles per iteration 579 mov r0, 0 580 delay_ms_loop: 581 add r0, 1 // 4 cycles 582 cmp r0, 199 // 3 cycles 583 blo delay_ms_loop // 3 584 ↪ cycles 585 586 lds r0, [0x00] // 4 587 ↪ cycles 588 sub r0, 1 // 4 cycles 589 sts r0, [0x00] // 3 590 ↪ cycles 591 bhi delay_ms_outer_loop // 592 ↪ 3 cycles 593 ret </pre>
--	--

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

```

1  PRNG_SEED = 0x0000
2  SIMPLE_IO = 0xfe00
3

```

```
4 prng:
5     ldr r0, [PRNG_SEED]
6     subs r0, 0
7     beq prngDoEor
8     lsl r0, 1
9     beq prngNoEor
10    bcc prngNoEor
11 prngDoEor:
12     xor r0, 0x1d
13 prngNoEor:
14     str r0, [PRNG_SEED]
15     ret
16
17 start:
18     mov r0, 0
19     str r0, [PRNG_SEED]
20     prng_loop:
21         call prng
22         str r0, [SIMPLE_IO]
23     b prng_loop
```

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

```
1  UART_DAT = 0xfe08
2  UART_IER = 0xfe09
3  UART_IIR = 0xfe0a
4  UART_FCR = 0xfe0a
5  UART_LCR = 0xfe0b
6  UART_MCR = 0xfe0c
7  UART_LSR = 0xfe0d
8  UART_MSR = 0xfe0e
9  UART_SCR = 0xfe0f
10 UART_DLL_DLAB = 0xfe08
11 UART_DLM_DLAB = 0xfe09
12
13 UART_DIV = 10 // 19200 baud
14 // UART_DIV = 20 // 9600 baud
15 UART_FILL_AMOUNT = 60 //
16     ↪ 19200 baud
17 // UART_FILL_AMOUNT = 30 //
18     ↪ 9600 baud
19
20 uart_init:
21     // line control register
22     // 8bit, 2 stopbits, no
23     ↪ parity, dlab active:
24     // 0b10xx_0111
25     // 8bit, 1 stopbit, no
26     ↪ parity, dlab active:
```

<pre> 23 // 0b10xx_0011 24 mov r0, 0x87 25 str r0, [UART_LCR] 26 27 // divisor latch access 28 mov r0, 0x00 29 str r0, [UART_DLM_DLAB] 30 mov r0, UART_DIV 31 str r0, [UART_DLL_DLAB] 32 33 // lcr as above but dlab 34 ↳ inactive 35 mov r0, 0x07 36 str r0, [UART_LCR] 37 38 // fifo control register 39 // fifo enable, reset tx 40 ↳ and rx fifo 41 // 0b00xx_x111 42 mov r0, 0x07 43 str r0, [UART_FCR] 44 45 // interrupt enable register 46 // clear all interrupts -> 47 ↳ fifo polled mode 48 mov r0, 0x00 49 str r0, [UART_IER] 50 51 // modem control register 52 // assert dtr, deassert rts 53 ↳ (should be asserted?), 54 // 0bxxx0_xx01 55 mov r0, 0x01 56 str r0, [UART_MCR] 57 ret 58 59 // r0 is byte to write 60 uart_write_inner: </pre>	<pre> 57 sts r1, [0x00] 58 59 uart_write_loop: 60 ldr r1, [UART_LSR] 61 and r1, 0x20 // bit 5, 62 ↳ fifo empty (not full?) 63 ↳ -> if 1, can accept 64 ↳ new data 65 beq uart_write_loop 66 67 str r0, [UART_DAT] 68 69 lds r1, [0x00] 70 ret 71 72 uart_write: 73 sts r1, [0x00] 74 call uart_write_inner 75 76 cmp r0, 0x20 // if less 77 ↳ than 0x20 -> send fill 78 ↳ null bytes 79 bge uart_write_end 80 81 mov r0, 0x00 82 mov r1, UART_FILL_AMOUNT 83 uart_write_fill_loop: 84 call uart_write_inner 85 sub r1, 1 86 cmp r1, 0 87 bhi uart_write_fill_loop 88 89 uart_write_end: 90 lds r1, [0x00] 91 ret 92 93 // r0 is byte to write </pre>
---	--

<pre> 90 uart_read: 91 ldr r0, [UART_LSR] 92 and r0, 0x01 // bit 0, fifo ↳ not empty -> 1 if data ↳ exists 93 beq uart_read_0 94 95 ldr r0, [UART_DAT] 96 ret 97 uart_read_0: 98 mov r0, 0 99 ret 100 101 102 // r0 is byte to write 103 uart_read_busy: 104 sts r1, [0x00] 105 106 uart_read_busy_loop: 107 ldr r1, [UART_LSR] </pre>	<pre> 108 and r1, 0x01 // bit 0, ↳ fifo not empty -> 1 ↳ if data exists 109 beq uart_read_busy_loop 110 111 ldr r0, [UART_DAT] 112 113 lds r1, [0x00] 114 ret 115 116 117 start: 118 call uart_init 119 uart_loop: 120 call uart_read 121 str r0, [0xfe00] 122 cmp r0, 0 123 beq uart_loop 124 call uart_write 125 b uart_loop </pre>
---	--