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

23.05.2022

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 23.05.2022

Unterschrift

Contents

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 [7].

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

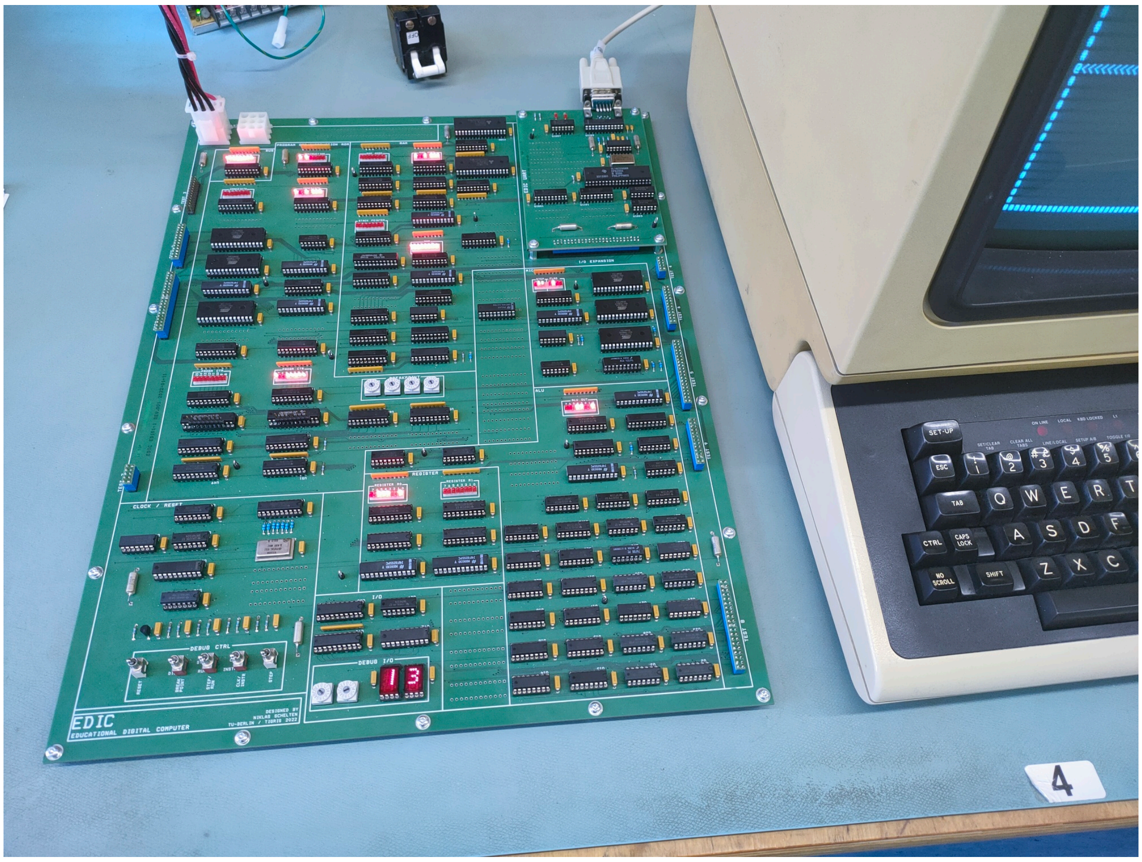


Figure 1.1: The final version of the EDiC playing Snake on a VT-100 over an RS-232 I/O card.

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

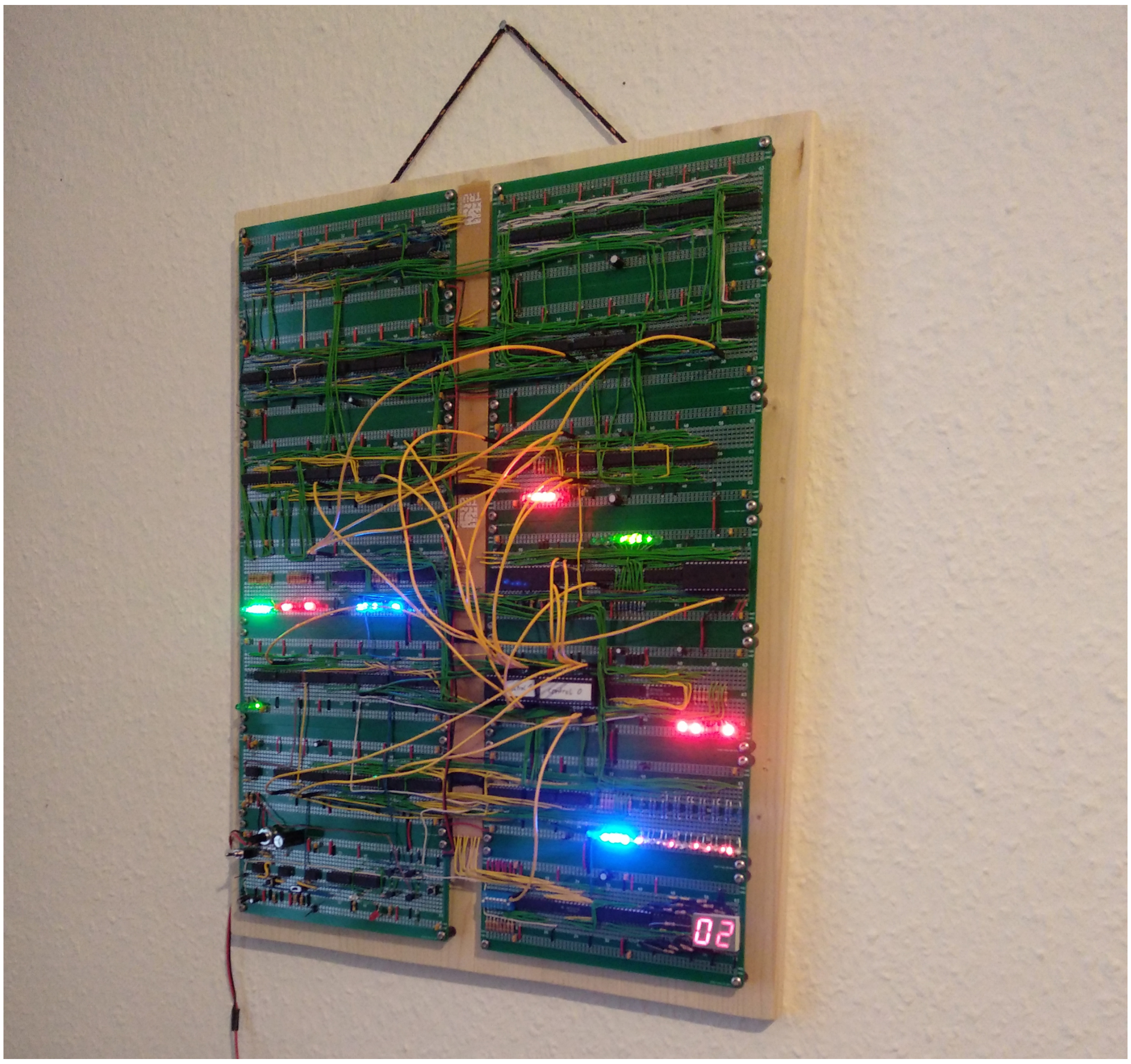


Figure 1.2: The first version of the CPU in its final state.

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 instructions[1]. 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* [16]. 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 [3]. 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 [16]:

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.

[1]The 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 instructions[2] 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 [5].

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

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

Table 2.1: Summary of the available alu operations.

aluOp[1]	aluOp[0]	aluSub	Resulting Operation
0	0	0	$(\mathsf{A} + \mathsf{B})$ Addition
0	0	1	$(\mathsf{A} - \mathsf{B})$ Subtraction
0	1	0	$(\mathsf{A} \wedge \mathsf{B})$ AND
0	1	1	$(\mathsf{A} \wedge \overline{\mathsf{B}})$
1	0	0	$(\mathsf{A} \veebar \mathsf{B})$ XOR
1	0	1	$(\overline{\mathsf{A} \veebar \mathsf{B}})$ XNOR
1	1	0	$(\mathsf{A} \gg \mathsf{B})$ logical shift right
1	1	1	$(\mathsf{A} \ll \mathsf{B})$ 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 essential subtracts the B input from the A input

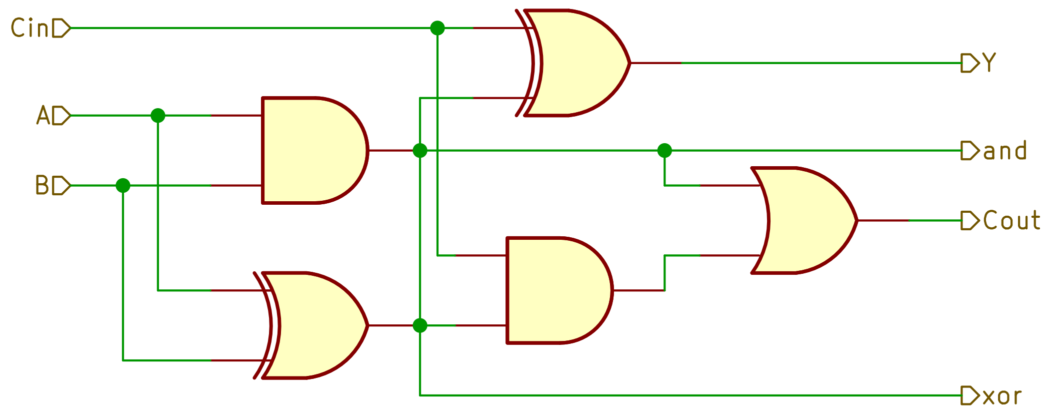


Figure 2.1: 1 bit full adder with the usual A, B and Carry inputs and Y and Carry outputs as well as the XOR and AND outputs.

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 desireable 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 width[3] which is especially important with only 8 data bits. It limits unsigned operations to only 0-127 even though the ALU would be capable of calculations with 0-255.

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

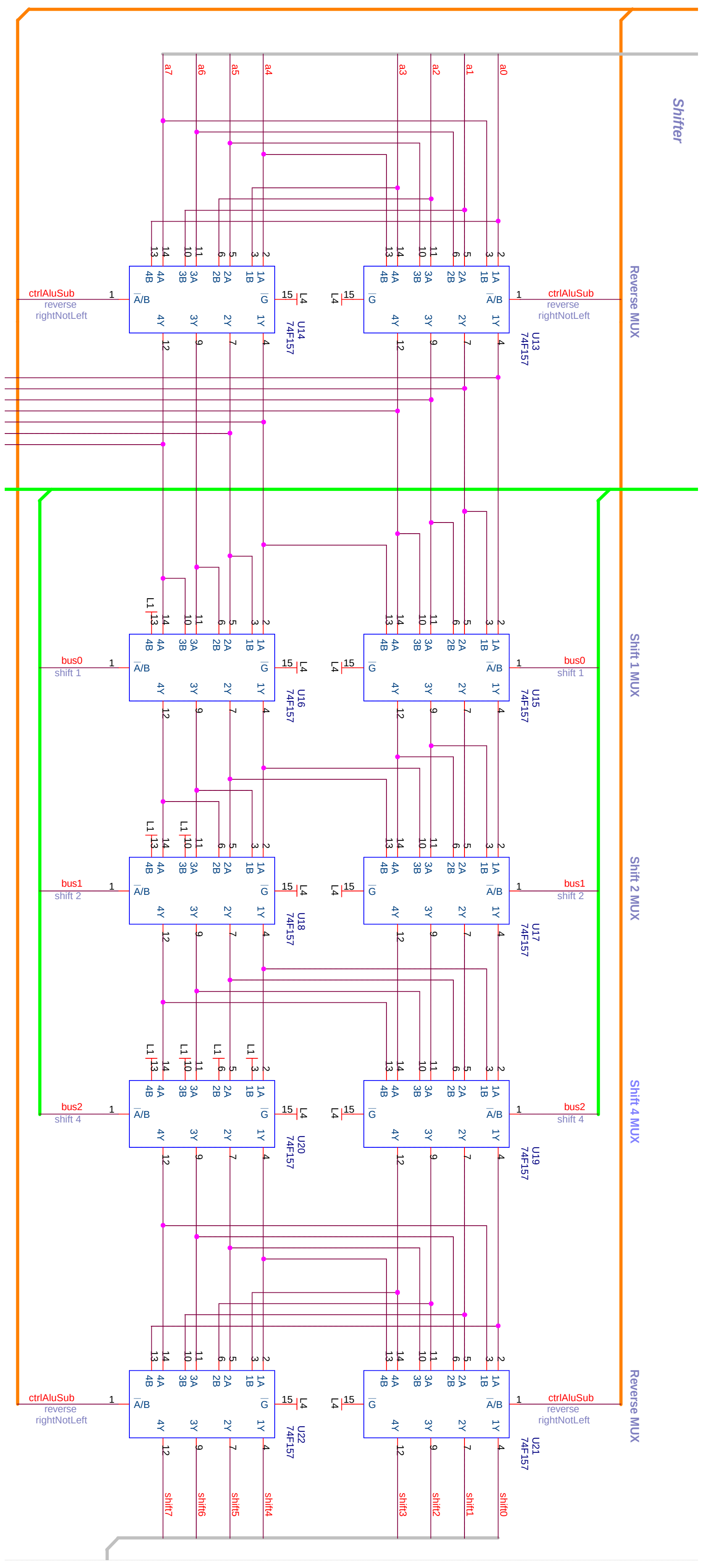


Figure 2.2: 8 bit bidirectional barrel shifter.

A lot of modern CPUs feature many different flags with the Intel 64® and IA-32 CPU having about 20 different flags [9, 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 be16 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 built 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 call[4]

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

[4]This 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.

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 MAR[5].

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

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:

  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) [10] 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

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

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

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

Code Example 3.1: Schema of the Microcode Definition CSON-File [2] as a TypeScript [12] 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:

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

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.

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

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

For example Code Example 3.3 he 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

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

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.

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

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

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

Code Example 3.6: 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'`.

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

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 $\geqslant$
`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 $\leqslant$
`1011`	`bge`	`N==V`	Signed $\geqslant$
`1100`	`blt`	`N!=V`	Signed $<$
`1101`	`bgt`	`Z==0` and `N==V`	Signed $>$
`1110`	`ble`	`Z==0` or `N!=V`	Signed $\leqslant$
`1111`	-	Any	Never (Not used)

on the current state of the ALU flags. For all other instructions, the flags are ignored and always executed[1]. For this special instruction, the last for 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[6] 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

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

```
PRNG_SEED = 0x0000
SIMPLE_IO = 0xfe00

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

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

Code Example 3.8: 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 EEP-ROM.

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

Code Example 3.9: 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 0xfffe (65534). The upper limit is not 0xffff because that address is reserved for the return address and should not be overwritten.

3.2.2.1 ALU Instructions

The following ALU instructions are available:

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

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

- 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: $r_0 := r_0 \veebar 15$

- 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 r0 $<$ 10 branches to instruction at address 66 and preserves the content of `r0`.

3.2.2.2 Memory Instructions

The following memory instructions are supported:

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

The two common instructions are `str` and `ldr` which are *store* and *load* operations. These two instructions take two parameters: Th 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 compile 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:

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

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

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

3.2.2.3 Miscellaneous Instructions

There are four more instructions that are essential:

- mov
- b
- call
- ret

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

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

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

Some examples:

`mov r0, 0xda`	Sets `r0` to `0xda`
`mov r1, r0`	Copies the value of `r0` to `r1`

`cmp r0, 10`
`blt 0x42`
	Branches to address (sets the PC to) `0x42` if the value of `r0` is smaller than 10

`call 0x100`	Calls a function at address `0x100`
`ret`	Returns from a function to the caller

3.2.3 Constants

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

3.2.3.1 Value constants

Value constants are the easiest kind of constants available. The first two lines of Code Examples 3.8 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 send to some I/O device (in this case the internal I/O register from section 2.2.6). It also prevents errors where a typo in an address causes unintended behavior of the code.

3.2.3.2 Labels

Instruction labels are often used in 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

```
PRNG_SEED = 0x0000             // no instruction
SIMPLE_IO = 0xfe00             // no instruction
                               b 0x0a // inserted by assembler
prng:                          // no instruction
  ldr r0, [PRNG_SEED]            ldr r0, [0x00]
  subs r0, 0                     subs r0, 0
  beq prngDoEor                  beq 0x07
  lsl r0, 1                      lsl r0, 1
  beq prngNoEor                  beq 0x08
  bcc prngNoEor                  bcc 0x08
prngDoEor:                     // no instruction
  xor r0, 0x1d                   xor r0, 0x1d
prngNoEor:                     // no instruction
  str r0, [PRNG_SEED]            str r0, [0x00]
ret                            ret

start:                         // no instruction
  mov r0, 0                      mov r0, 0
  str r0, [PRNG_SEED]            str r0, [0x00]
  prng_loop:                   // no instruction
    call prng                      call 0x01
    str r0, [SIMPLE_IO]            str r0, [0xfe00]
  b prng_loop                    b 0x0c
```

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 program[2] 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`

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

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

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

Code Example 3.11: Excerpts of the Snake assembler program used in the demo in figure 1.1.

```
mov r0, 0x59 // 'Y'
str r0, [0x2000]
mov r0, 0x6f // 'o'
str r0, [0x2001]
mov r0, 0x75 // 'u'
str r0, [0x2002]
mov r0, 0x20 // ' '
str r0, [0x2003]
mov r0, 0x6c // 'l'
str r0, [0x2004]
mov r0, 0x6f // 'o'
str r0, [0x2005]
mov r0, 0x73 // 's'
str r0, [0x2006]
mov r0, 0x74 // 't'
str r0, [0x2007]
mov r0, 0x21 // '!'
str r0, [0x2008]
mov r0, 0x21 // '!'
str r0, [0x2009]
mov r0, 0x21 // '!'
str r0, [0x200a]
mov r0, 0x20 // ' '
str r0, [0x200b]
mov r0, 0x53 // 'S'
str r0, [0x200c]
mov r0, 0x63 // 'c'
str r0, [0x200d]
mov r0, 0x6f // 'o'
str r0, [0x200e]
mov r0, 0x72 // 'r'
str r0, [0x200f]
mov r0, 0x65 // 'e'
str r0, [0x2010]
mov r0, 0x3a // ':'
str r0, [0x2011]
mov r0, 0x20 // ' '
str r0, [0x2012]
mov r0, 0x0 // NULL-byte
str r0, [0x2013]
mov r0, 0 // restore r0
```

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

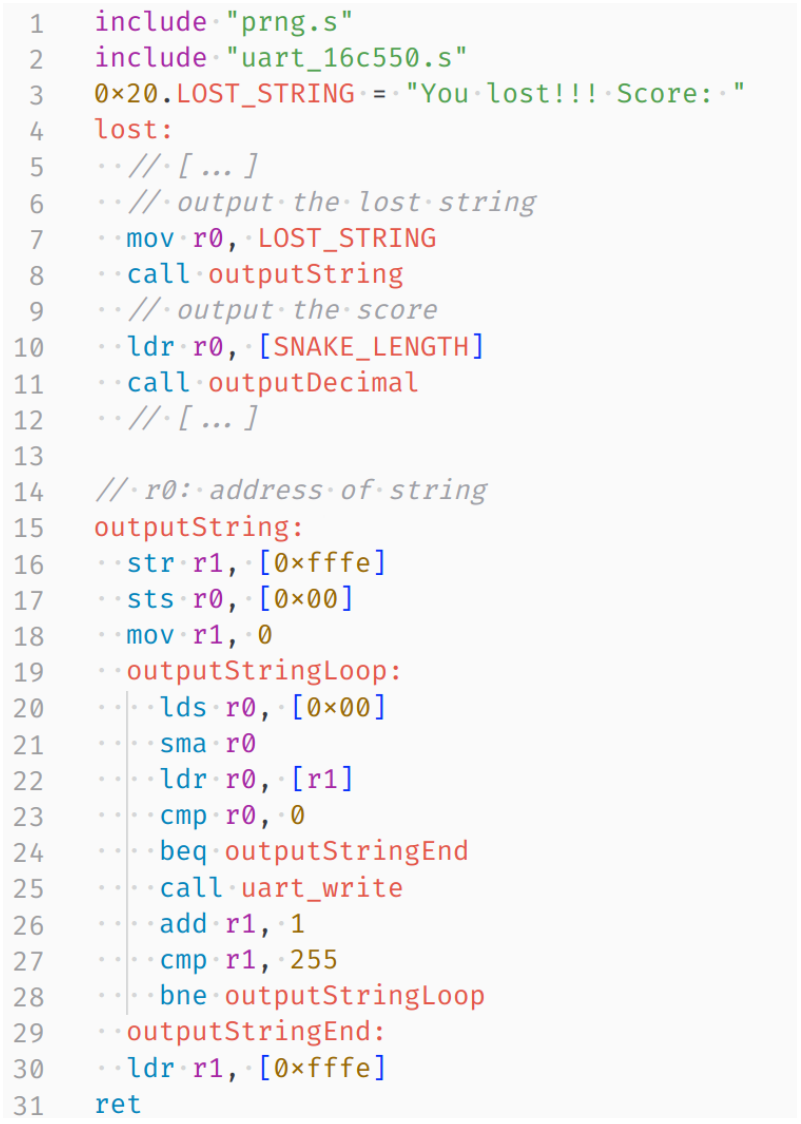
instruction is called with the MSBs of the address and then the `ldr` instruction with the loop register `r1` as an address argument is called. The character (in `r0`) is then passed as an argument to the `uart_write` function.

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



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

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

Figure 3.1: The syntax highlighting with the EDiC Visual Studio Code Extension and the Atom One Light Theme [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 [13] is one of the leading extensible code editors, an extension for EDiC assembler has been developed and published [17]. 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 [11] 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 simulation is to proof the general workings of the CPU architecture. There was no attempt made to provide a chip-level simulation of the hardware build but rather to provide a top-level behavioral model. The chosen development environment is the AMD - Xilinx Vivado [20] as it is freely available and provides an advanced simulation environment while providing the possibility to synthesize for relatively cheap 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. Most FPGAs do have bidirectional tri-state transceiver for I/O logic but not for internal logic routing. However, the Hardware Description Languages (HDLs) (both VHSIC Hardware Description Language (VHDL) and Verilog) support tri-state logic and the Xilinx Simulation tool also does. As the first CPU was only simulated, this was not a problem and the tri-state logic could be used the same way as in the hardware build. Chapter 4 describes how tri-state logic is solved for the synthesis of the EDiC.

4.0.1 Language Choice

There are two main HDLs: Verilog and VHDL. Both are widely supported and used and can also be used in parallel in the same design. At the Technical University Berlin (TUB) VHDL is taught and in Germany it also used more often. However, in general both are used about equally often [14].

As I only knew VHDL and very basic concepts of Verilog, I decided to start in Verilog to get to know the differences. Code Example 4.1 shows the Verilog Code for the ALU module (without the module definition to fit on one page). Lines 24-28 describe the synchronous, positive-edge-triggered alu output register with a write enable. The 8 XOR for the B input are described by lines 39-42 and lines 44-61 show an combinatorial process for the alu operation and multiplexing. The lines 50-58 describe the bidirectional barrel shifter with 3 shift steps (by 1, 2, and 4) and the reverse Multiplexers (MUXs) in front and at the end.

```
always @(posedge i_clk) begin
  if (i_aluWr) begin
    r_y <= s_y;
  end
end

transmitter inst_tx(
  .a(r_y),
  .b(o_y),
  .noe(i_noe)
);

assign o_negative = r_y[7];
assign o_nZero = | r_y;

genvar i;
for (i = 0; i < 8; i++) begin
  assign s_b[i] = i_b[i] ^ i_subShiftDir;
end

always @* begin
  case (i_aluOp)
    2'b00: s_y <= i_a + s_b + i_subShiftDir;
    2'b01: s_y <= i_a & s_b;
    2'b10: s_y <= i_a ^ s_b;
    2'b11: begin
      s_a = i_subShiftDir
        ? {i_a[0], i_a[1], i_a[2], i_a[3], i_a[4], i_a[5], i_a[6], i_a[7]}
        : i_a;
      s_shift1 = i_b[0] ? s_a >> 1 : s_a;
      s_shift2 = i_b[1] ? s_shift1 >> 2 : s_shift1;
      s_shift3 = i_b[2] ? s_shift2 >> 4 : s_shift2;
      s_y <= i_subShiftDir
        ? {s_shift3[0], s_shift3[1], s_shift3[2], s_shift3[3], s_shift3[4], s_shift3[5], s_shift3[6], s_shift3[7]}
        : s_shift3;
    end
  endcase
end
```

Code Example 4.1: (System)-Verilog Code for the ALU of the first CPU version.

4.1 Behavioral Simulation

4.2 Behavioral Implementation

4.3 Chip-level Implementation

4.3.1 Conversation Script

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 µA for high-level output and −200 µA for an low-level output [8]. 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 µA in both directions [18].

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 through[1]. When the $\overline{\text{EN}}$ input is high, on the other hand, the output does not change. This

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

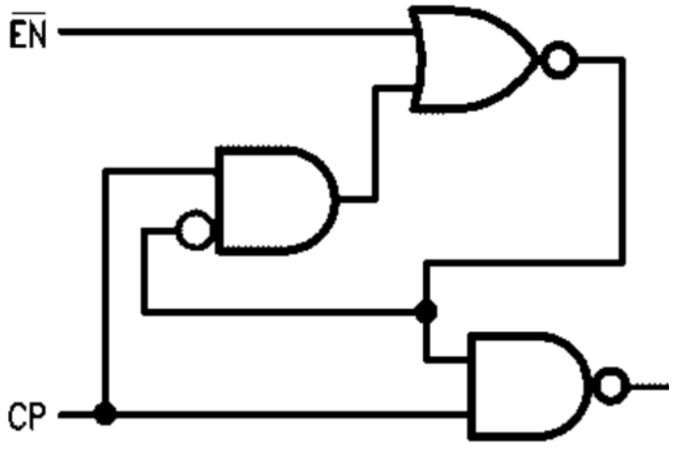


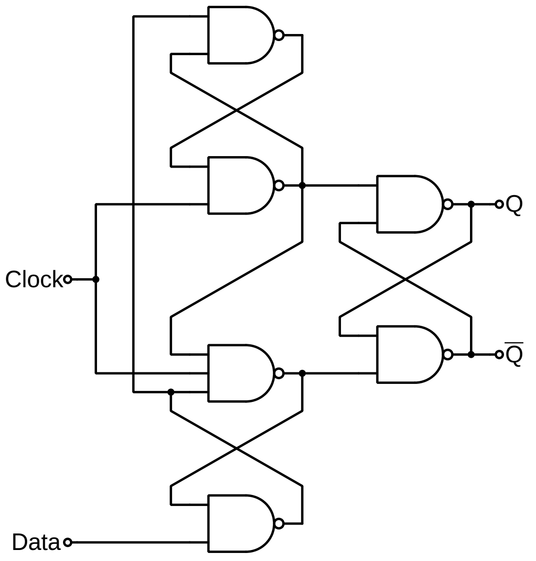
Figure 5.1: Clock Enable circuit of the *74F825* IC [4].

circuit prevents the $\overline{\text{EN}}$ to trigger a falling edge (which would trigger the flip-flops) on the CP output. However, when the $\overline{\text{EN}}$ goes high while the CP input is high, then the output also goes high. This is not directly a problem because the flip-flops only trigger on falling edges but is the reason for timing requirements on the $\overline{\text{EN}}$ input which are discussed in more detail in section 5.1. 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 $\overline{\text{SR}}$ NAND latches as shown in figure 5.2.

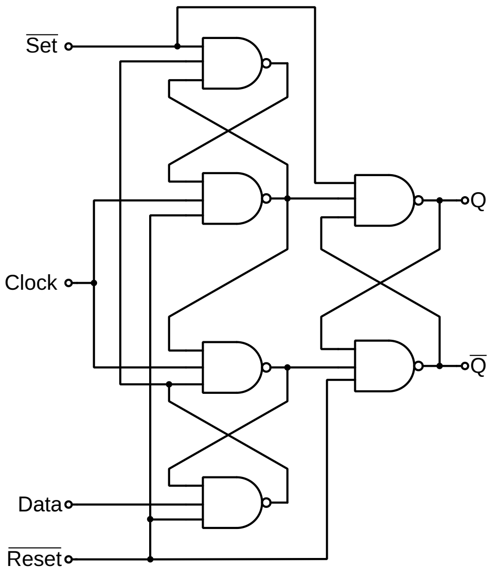
The third feature that may be important is a three-state output which allows the register to be directly connected to a bus. It is accomplished by adding a tri-state output driver to the outputs of the flip-flops.

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 $\overline{\text{SR}}$ NAND latches [15].



(b) D-type flip-flop modified to support $\overline{\text{Clear}}$ and $\overline{\text{Set}}$ [19].

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

6 Evaluation

6.0.1 Test Adapter

7 Conclusion and Future Work

A Collection of assembler programs for the EDiC

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

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

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

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

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


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

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

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

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

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

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

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

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

```

updateItem:
  str r1, [0xfffe]

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

  ldr r1, [0xfffe]
ret

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

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

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

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

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

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

headEnd:

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

updateTail:
  str r1, [0xfffe]

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

  tailUp:
  ldr r1, [SNAKE_TAIL_LINE]
  sub r1, 1
  str r1, [SNAKE_TAIL_LINE]
  b tailEnd

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

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

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

tailEnd:
  ldr r1, [0xfffe]
ret

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

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

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


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

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

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

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

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

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

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

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

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

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

```


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

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

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

  ldr r0, [0xfffe]
  call uart_write

  ldr r1, [0xfffd]

ret

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

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

  ldr r1, [0xfffe]
ret

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

    outputStringEnd:

    ldr r1, [0xfffe]
ret


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

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

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

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

    call uart_read
    cmp r0, 0
    beq readArrowNothing
```

```
  cmp r0, ESCAPE1
  bne readArrowLoop

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

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

  delay_ms_outer_loop:

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

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

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

```
PRNG_SEED = 0x0000
SIMPLE_IO = 0xfe00

```

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

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

Code Example A.3: The utility library for the Universal Asynchronous Receiver-Transmitter (UART) extension card of the EDiC with the 16c550 UART Transceiver.

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

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

uart_init:
  // line control register
  // 8bit, 2 stopbits, no parity, dlab active:
  // 0b10xx_0111
  // 8bit, 1 stopbit, no parity, dlab active:
  // 0b10xx_0011
  mov r0, 0x87
  str r0, [UART_LCR]

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

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

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

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

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

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

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

  str r0, [UART_DAT]

  lds r1, [0x00]
ret

uart_write:
  sts r1, [0x00]
  call uart_write_inner

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

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

uart_write_end:
  lds r1, [0x00]
ret

```

```

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

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


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

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

  ldr r0, [UART_DAT]

  lds r1, [0x00]
ret


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

Acronyms

Notation	**Description**
ALU	Arithmetic Logic Unit i, 7
CISC	Complex Instruction Set Computer 5
CPU	Central Processing Unit 1
CSON	CoffeeScript-Object-Notation 23
EDiC	Educational Digital Computer ii, 1
EEPROM	Electrically Erasable Programmable Read-Only Memory 6
FPGA	Field Programmable Gate Array 1
HDL	Hardware Description Language 41
IC	Integrated Circuit 1
IDE	Integrated Development Environment 39
JSON	JavaScript Object Notation 23
LED	Light-Emitting Diode 45
LSB	Least Significant Bit 12
MAR	Memory Address Register 14
MSB	Most Significant Bit 9
MUX	Multiplexer 41
NOP	No Operation 20
PC	Program Counter i, 6

Notation	**Description**
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 41
UART	Universal Asynchronous Receiver-Transmitter 62
VHDL	VHSIC Hardware Description Language 41

List of Figures

List of Tables

List of Code Examples

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

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

[4] 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.

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

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

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

[8] Texas Instruments. *SNx4LS245 Octal Bus Transceivers With 3-State Outputs.* 1976. URL: https://www.ti.com/lit/ds/sdls146b/sdls146b.pdf.

[9] 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.

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

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

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

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

[14] 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`.

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

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

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

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

[19] 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`.

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