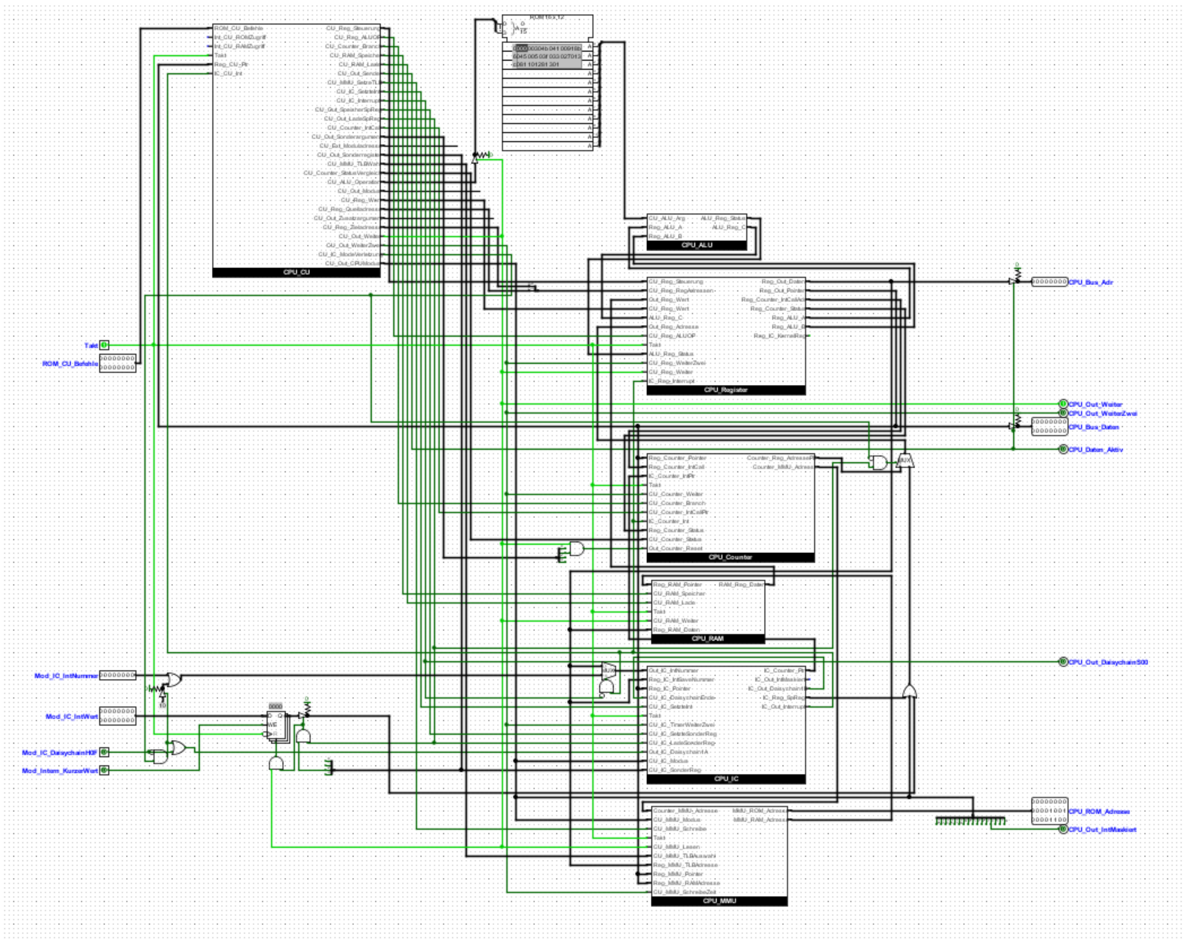
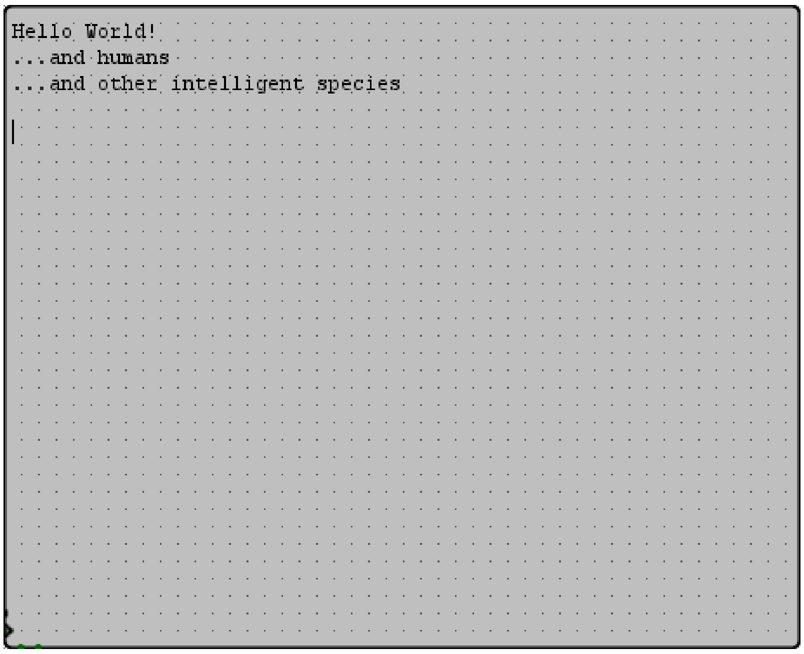
ERS3864

A complete guide through the wonders of ERS systems





```
Hello World!
...and humans
...and other intelligent species
```

By RascalFoxfire

1
28.07.2021

Index

Introduction

ERS 3864K is a virtual homebrew computer system build entirely in Logisim Evolution. Its name Einfaches RechenSystem 3864K (which translates to simple calculation system) is the third system in the ERS series and the first who was published on the internet. Although it is anything but simple anymore it has traded that aspect with more advanced features. This guide tries to compensate the first fact and gives anyone a detailed in view of the architecture and how to program the system. Starting by pure beginners who want to know how computers and especially how the CPU works to experienced digital circuit planner, builder and programmer who seek for a new challenge or free time project.

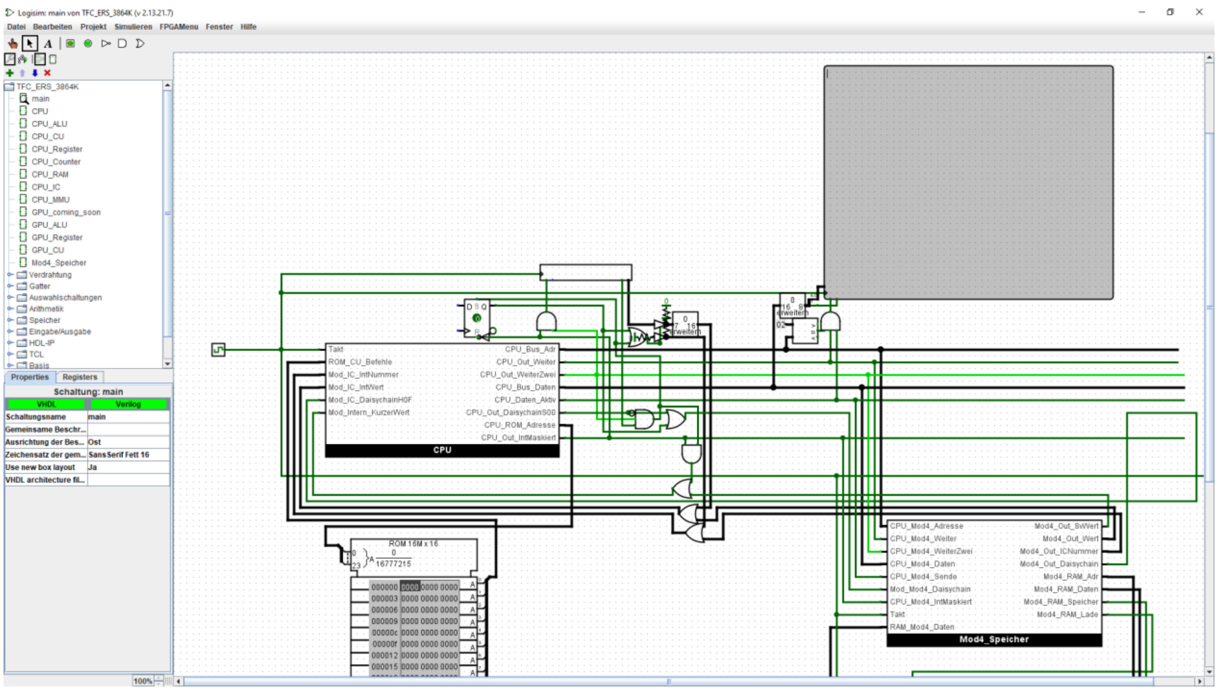
Technical specifications

- ERS 3864K computer system
- RX 8 Lepton CPU
- Harvard RISC like architecture (called XHA: experimental Harvard architecture)
- 2 cycles per instruction
- 16 ALU operations (excluding direct divide and multiplicate)
- 8 ALU flags (carry 8 bit, carry 4 bit, A <=> B, left shift bit, right shift bit, Overflow)
- 8 bit ALU width
- 16 CPU operations
- 16 CPU register separated into user and kernel mode register
- 32 interrupts (including one for each module, one timer, one mode violation interrupt, 10 for future hardware interrupts and 4 free for software)
- 4 CPU flags (user/kernel mode, virtual/direct addressing, timer interrupt activation and interrupt masking)
- 16 bit directly addressable ROM and separate RAM
- 24 bit indirectly addressable ROM and separate RAM over the integrated MMU
- Virtual addressing with 256 Byte fixed page size and separate ROM and RAM TLBs
- Bus system capable of handling up to 16 I/O modules
- Comes with integrated simple user I/O modules and a crude IDE-like controller for RAM modules

1. Getting started

1.1 Set up the computer

The ERS 3864K was built in Logisim Evolution (not the original old Logisim) version 2.13.21.7 but the computer will practically work on every version above that. You can download the actual version of Logisim on GitHub (https://github.com/logisim-evolution/logisim-evolution). Logisim Evolution needs Java RE to work. After you downloaded and started Logisim Evolution you just need to open the TFC_ERS_3864K.circ file in it.



Schaltungsname	main
Gemeinsame Beschr...	
Ausrichtung der Bes...	Ost
Zeichensatz der gem...	SansSerif Fett 16
Use new box layout	Ja
VHDL architecture fil...	

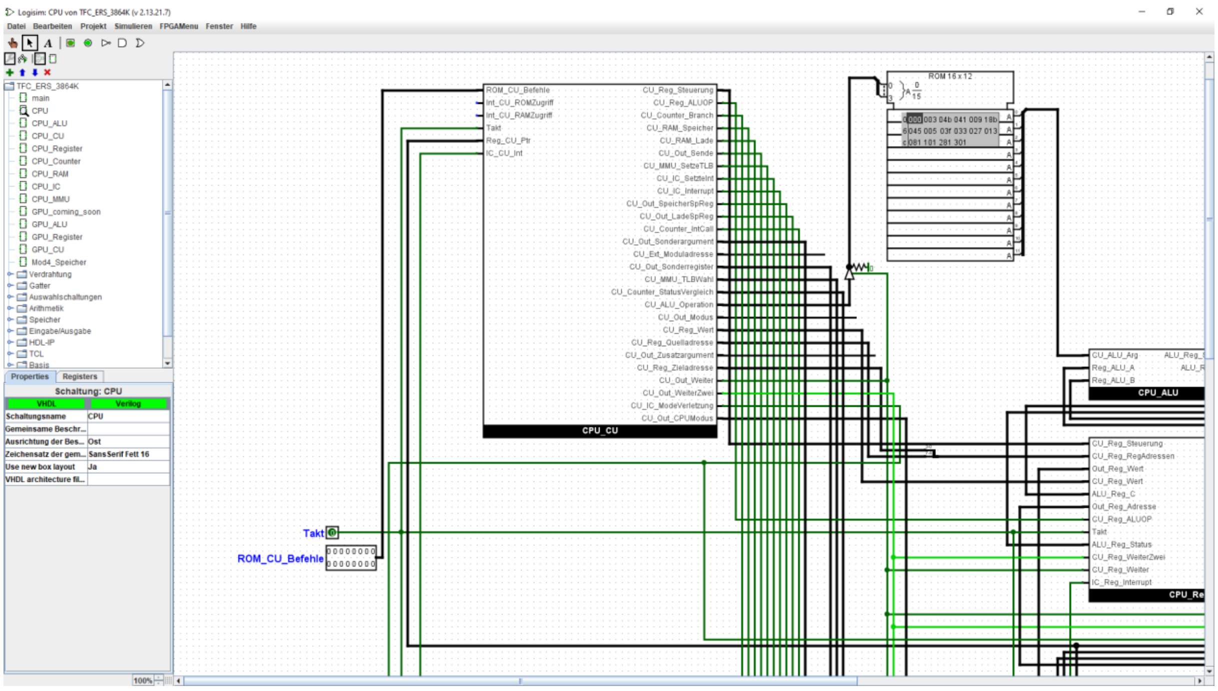
ERS3864K AFTER STARTING LOGISIM EVOLUTION

But the computer is not yet ready. You need to load two ROM files into the CPU. This is done by right clicking the big “CPU” module on the left side of the middle screen and clicking on the option “View CPU”.

After that you should be in the inside of the CPU. On the top middle should be an oddly shaped module with the title “ROM 16 x 12”. Like the CPU module you need to right click it and press on “load image”. Here you need to find the ALU_DecoderROM and load it. The second ROM needs to be loaded in the ROM module inside CPU_CU module with the CU_DecoderROM image.

When you are done with that your ERS 3864K is ready to work! Now you can go back to the main circuit (double click onto it in the explorer window left) to the option simulate => Tick Frequency => 4.1KHz and hit Ctrl + K (or click Ticks Enabled in the same drop-down menu)! And the ERS 3864K does… nothing. If you going to the main circuit, you see that the ROM beneath the CPU is filled with only zeros (which here means empty). A 0 on the left of one 4 number packet is interpreted as no operation by the CPU, so it does: nothing! And goes to the next instruction which is zero too.... For now, you can just hit Ctrl + K again and press Ctrl + R (which resets all dynamic memories like registers but not RAM or ROM modules). Now it is time to let the system do something useful.

Small hint: on the left side is a list with the main and subcircuits but you need to go manual into them (via right clicking and viewing in the middle screen section) at least once. But if you done that they will be actualized during runtime (which is handy for debugging).



Schaltung: CPU

VHDL	Verilog
Schaltungsname	CPU
Gemeinsame Beschr...	
Ausrichtung der Bes...	Ost
Zeichensatz der gem...	SansSerif Fett 16
Use new box layout	Ja
VHDL architecture fil...	

INSIDE OF THE RX 8 WITH LOADED ALU_DECODERROM

1.2 Make the CPU do something

The ERS 3864K is coming with a demonstration and test program called CPUTest.dat. All you need to do is to load it into the program memory. That memory is the ROM beneath the CPU. It contains all operations and data for the CPU like a drive in a real-life computer.

If you activate the clock (enable ticks) then you will see a lot of flickering in main and in the CPU itself but still nothing in the terminal. As the name suggests CPUTest is a program to test all CPU and module functions (except the external RAM module). The program has a documentation on when which component is tested, and which values should be in the registers if anything works well. After a short time, the CPU should do the same thing repeatedly. If you look closely at the program memory you will see that the one black box that marks the ROM values (and tells which values is send to the CPU now) jumps into the same values.

Now you can switch to the hand tool on the top left and click the small empty box top right of the CPU. This is a keyboard input module and can be used to input values to the circuitry with the real-life keyboard. If there is a pink marking around the keyboard module then you can tip some letter and numbers in it. These letters and numbers should now appear on the terminal. If some letters are not written to the terminal, then do not worry! It is just a hardware bug and will be corrected in the next time.

If some special symbols are not appearing on the terminal even after repeatedly pressing them or some other symbol is drawn: the keyboard module can only handle ASCI symbols. ASCI is a standard to encode regular symbols into binary values (do not forget normal computers work with zeros and ones and the ERS is not an exception). This standard can handle 128 different symbols and special commands like “new line”. 128 sounds much but it needs to address all capital and normal letters, numbers, symbols like “!” and some commands. But all is in this case not all and the fact that ASCI is old and replaced today with UTF-8/16 which can handle MUCH more symbols means that not all existing symbols are addressed by that. But that is not a problem because it contains all standard keys of a keyboard.

1.3 Hello World!

Now you know how to load programs into the ERS which is handy, but you probably want to program it and make some other stuff then what has been developed by others. It is time for your first own program! And what handbook would it be if it does not start with the classic “Hello World!”? Let us start.

First you need to go to the main circuit and right click the program memory. Now you click on “Edit Contents”. A new window should open now with a giant list of zeros packed into four number packets. That is where you program the computer (for the next time). But do not be scared, it is very simple to program the computer!

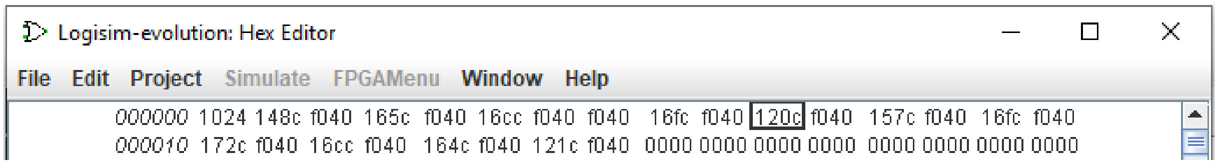
The numbers on the left on the start of each line are for orientation in the code and are standing for the address of the operation behind it. These addresses are hexadecimal numbers. We start with the operation “000000” on the top left side. Write “1024” into it.

Congratulations you write your first operation into the computer! But what does this operation? To understand this, you need to separate the four numbers: the one on the left is the operation, the zero is an argument, the two is and additional value and the same with the four on the right. In that case operation one means “load the value that stands in argument and the first additional values into the register with the name in the second additional value”. If you know the hexadecimal number system you should know that one hexadecimal number can be directly converted into a four-bit value. The ERS 3864K has an eight-bit CPU and with that in mind it can handle two hexadecimal numbers of data at the same time (which means it can handle numbers greater than decimal 15). For short: with that operation we load the value 02 into the CPU register 4.

Next, we need to load the value “48” into the special register C. You need to go to the next free operation (the right one) and write into it “148c”. 48 is the ASCI value for the letter “H” (you can google for an ASCI list for all symbols). Like the keyboard module the terminal uses the ASCI code to print symbols. The only thing we need to do now it to send the value to the terminal.

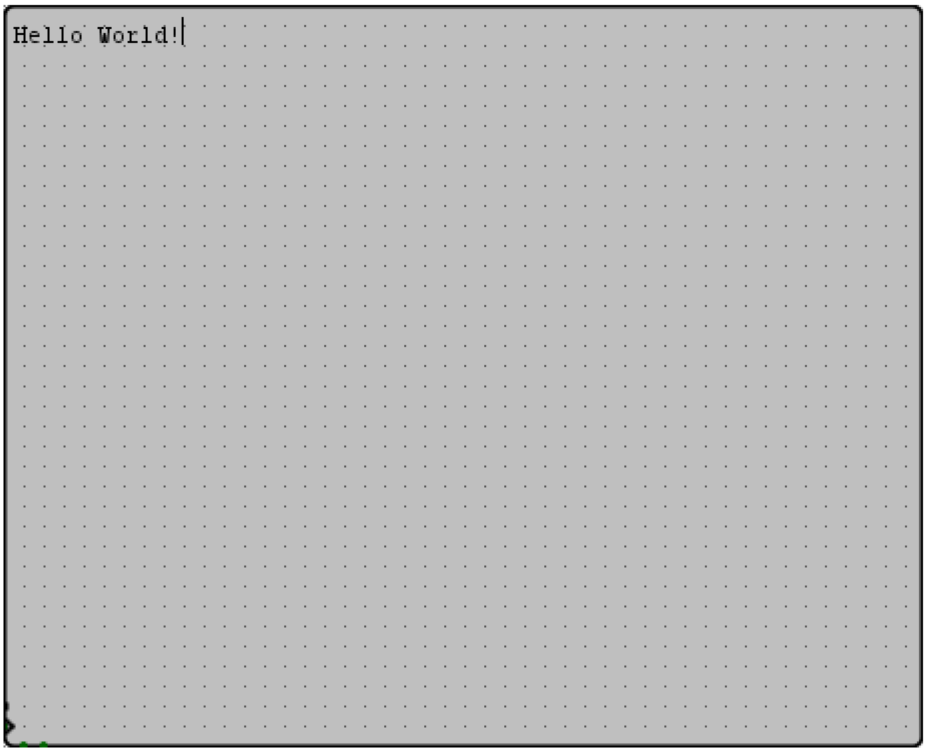
And that is where the operation “f” comes in: it sends the values in the register C and D to a module with the address inside a register. The terminal module address is 02, exactly the address we loaded into register 4 before. We only need to tell the CPU that it should send register C and D to the module address saved in register 4. And that is done with the operation “f040”. So, you write that into the third operation space in the first line. The zeros have no use in this operation and are practically blocked inside the CPU (not all numbers are used in all operations).

If you save the program, go back to Logisim, and enable the clock. A “H” should appear on the terminal. But now back to programming (do not forget to stop the clock and reset the computer): all values in the registers are saved over other operation if you did not overwrite them with another value. If we want to send a “e” we need to write a “65” into the C register and send it to the terminal. The same with the following values: 6c (“l”), 6f (“o”), 20 (space), 57 (“W”), “o” again, 72 (“r”), “l” again, 64 (“d”) and finally 21 (“!”). The program should look like this:


Logisim-evolution: Hex Editor
File Edit Project Simulate FPGAMenu Window Help

```
000000 1024 148c f040 165c  f040 16cc f040 f040  16fc f040 120c f040  157c f040 16fc f040
000010 172c f040 16cc f040  164c f040 121c f040  0000 0000 0000 0000  0000 0000 0000 0000
```

And you are done! Now start the clock watch the terminal and voilà! It prints “Hello World!”. That was a lot of work but do not be scared off, it is fun to write and debug programs and watch them working!



```
Hello World!
```

THE TERMINAL AFTER STARTING THE „HELLO WORLD” PROGRAM

1.4 What is next?

Now you know how to set up the computer, how to load programs into it and how to write simple text messages on the terminal. But this is only a tiny fraction of what the computer is capable of. You are free to go and visit the other chapters.

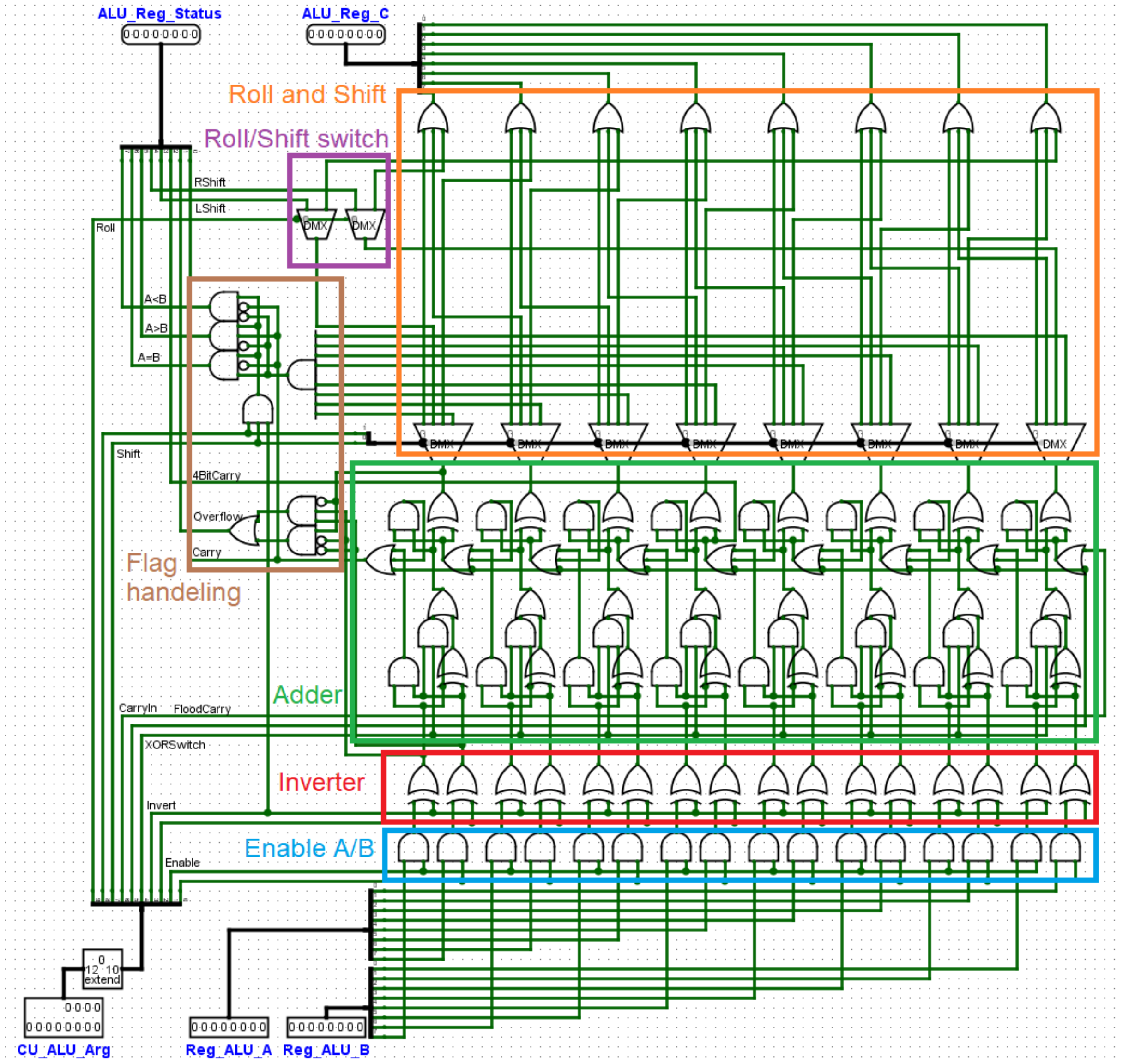
Do you want a complete tutorial into ERS programming with examples and projects? Or maybe you are experienced in programming Logisim computers and CPUs and just want a complete reference list of all operations? Or do you want to expand or modify the computer? Maybe you just want to look what the ERS 3864K is capable of? Whatever you want: this handbook has everything you need to do it!

2. Technical details

2.1 CPU RX 8 Lepton

2.1.1 The ALU

The ALU is based on a simple 8 bit adder with shift and roll unit. It consists of 6 different parts:



As you can see the ALU is not capable to directly divide or multiply, but it can perform all other arithmetic and logic functions. There would be two other designs: a look up table (a big ROM which contains all solutions) and a design with a de-multiplexer which leads A and B to the different functions (like an adder or AND gate). The first solution would mean that one must program a giant ROM with all possible solutions (or build a circuit to do that). The second is the most common type of ALU in Logisim CPUs but not a real challenge. That is why the RX 8 has a “switch logic” based ALU.

In the RX 8 the ALU can perform 16 operations including the NOP ALU operation which can be used to zeroth a register. But it can perform many more operation which are useful. If you want, you can reprogram the ALU decoder if you find more useful functions. More information to the supported ALU operations in the RX 8 is the references chapter.

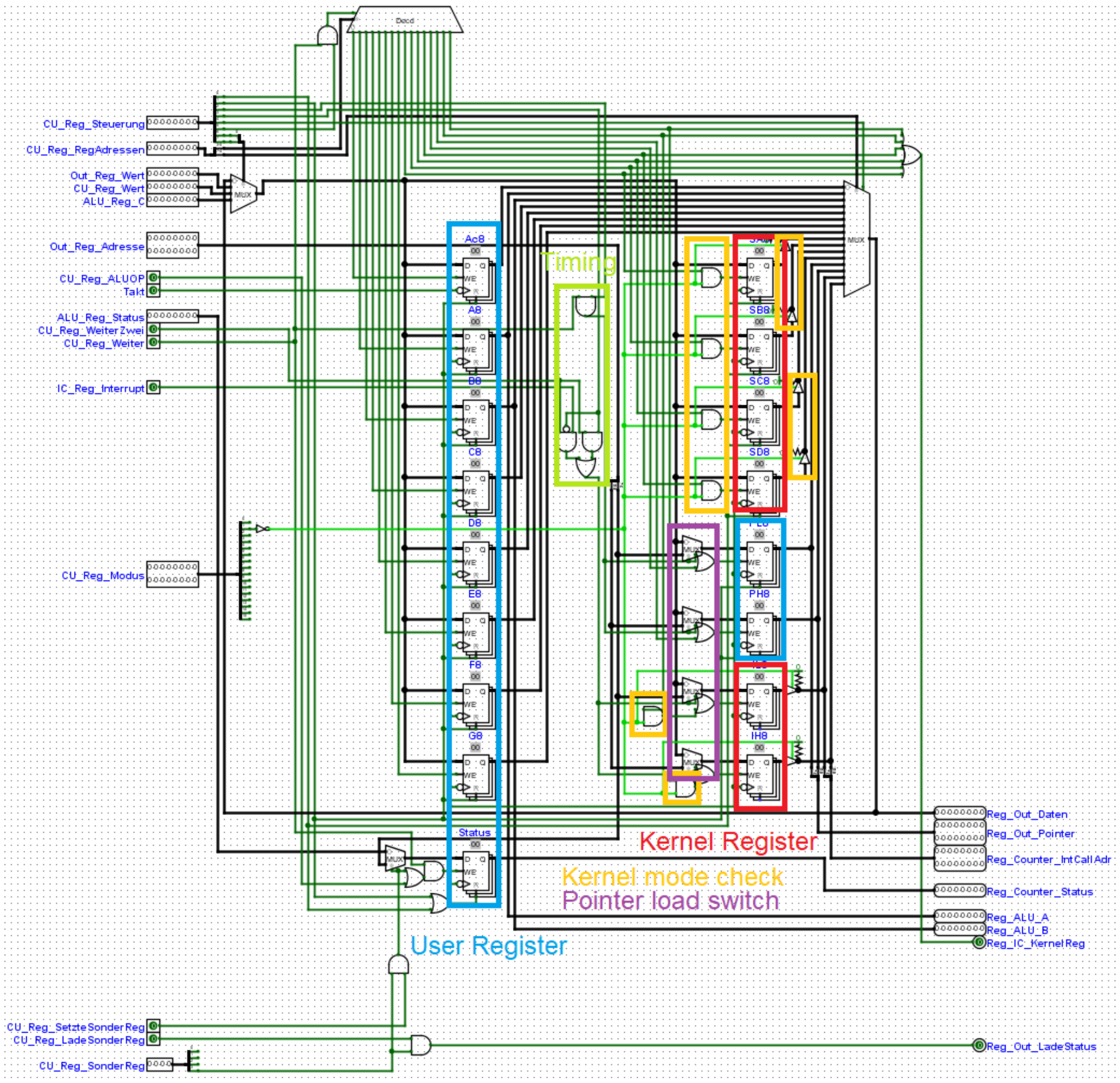
2.1.2 Registers

The register bank consists of 16 8 bit register. These are separated to user and kernel registers. User registers are:

- 1 Accumulator register (which is a universal register and only called accu because of historic reasons)
- 2 ALU source registers A8 and B8
- 5 universal registers
- 2 pointer registers
- And 1 status register

The kernel registers are:

- 4 kernel universal registers
- And 2 interrupt call pointers



The RX 8 Lepton blocks all kernel register operations if the CPU runs in user mode. All registers are usable in kernel mode. Aside from that all registers can be stored to/loaded from without limitations (except for the status register). The pointer register is used to address the RAM via the MMU and to jump/branch. The interrupt call pointer gets addresses during the start of an interrupt to jump back after the interrupt was handled. Since the RX 8 does not have a hardware stack nor functions to actively support a software stack it has 4 kernel universal register to make task switching possible.

2.1.3 Counter

The counter consists of a simple register which holds the address of the actual/next operation and an adder unit. It can be set during branches/jumps, interrupts, and returns from interrupts. The branch compare unit is in the counter included. It has a pointer bypass for the case if an interrupt occurs during a jump/branch operation.

2.1.4 CU

The CU consists of an operation register (which holds the actual operation), an operation decoder in form of an ROM module, a toggle register to generate two clock signals and a mode register unit. The mode registers have some weird logic on the left side which is used to automatically set the modes if an interrupt occurs.

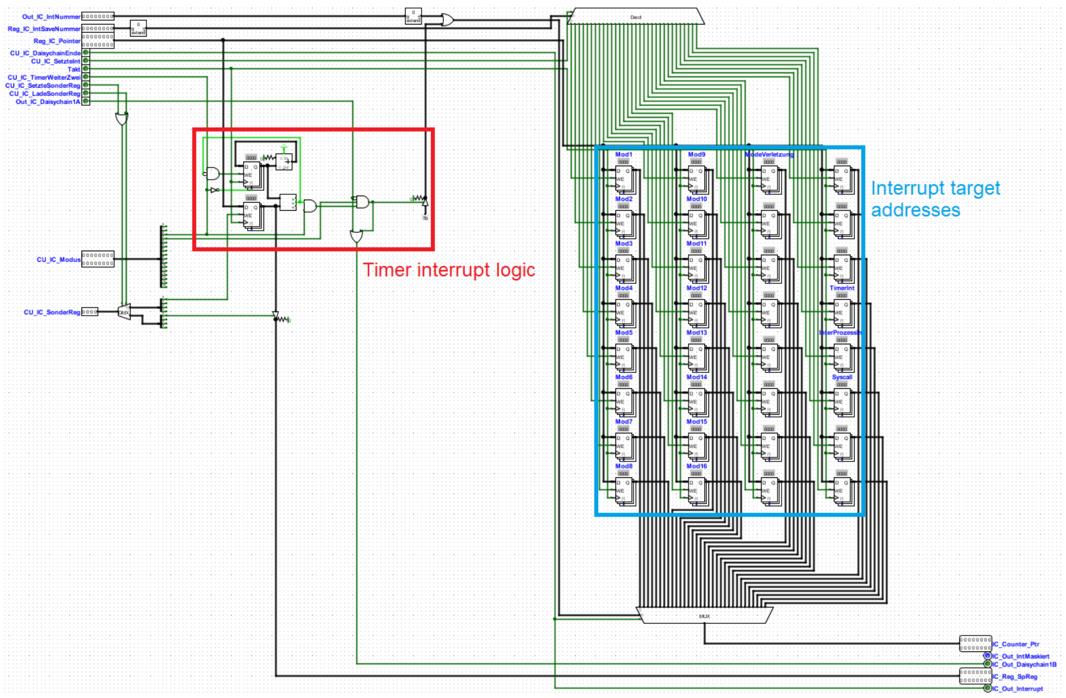
2.1.5 RAM

Not much to say here. It is a 16 Mbyte RAM module with separate busses for store and load. It gets the address from the MMU and get/send data from/to the register.

2.1.6 Interrupt controller

The IC is basically a register bank with 32 16 bit registers for 32 different interrupts. 16 are reserved for the 16 external modules and the other 16 are for internal purposes (timer and software) They hold an address with the interrupt handler in the ROM and force the register bank to store the actual counter value in the interrupt call pointer. If two interrupts are fired, then the one with the lowest priority/interrupt number will be processed. This is handled with an "daisy chain" which is set to high if a module wants to send an interrupt to CPU. The modules/interrupt sources with a lower priority must detect it and hold their interrupt number and additional value until the other interrupt is handled and the CPU can receive an interrupt again. All interrupts can be masked with a CPU mode flag (except the mode violation interrupt).

In the IC included is a timer interrupt which can be set/reset with the CPU mode flag. The timer compare value can be set with an operation in kernel mode.



2.1.7 Memory management unit

The MMU holds 2 RAM modules which are the TLBs for the ROM and RAM. They hold the translation values from virtual to physical addresses and can be bypassed with the TLB CPU mode flag. The offset is static at 8 bit which means that each page is 256 bytes large. Both TLBs have 256 entries for 16 bit “direct virtual” addressing, but they can output a 24 bit address thanks to 16 bit entries. Each entry can/must be set directly with the pointers and one register which holds the target entry address.

2.2 Bus and Modules

2.2.1 Bus

The external bus system is a not DMA ready event-based communication line. The CPU can directly send a 16 bit value to one of 16 different modules at a time. On the other side modules can send 16 bit data to the CPU via the interrupt system. For that the modules must be capable to hold their interrupt request if the CPU sends the “interrupt masked” signal. And they must use the daisy chain: if no other module/internal interrupt is fired then the module must send a signal to the daisy chain to inform and block interrupt requests from modules with lower priority.

2.2.2 Keyboard

The Keyboard is a read only module at the address 0x00 and with the interrupt number 0x00. If a character is typed into the keyboard, it will send an interrupt request with that character coded in ASCI to the CPU. It can be used for basic human interactions with the computer.

2.2.3 Terminal

The Terminal is a write only module with the address 0x02. It can output human readable ASCI characters.

2.2.4 The not-an-IDE-controller

The not-an-IDE-controller is an external memory module at address 0x04. It has a small register bank to set up the target address and the data. You can set these, store data into the RAM module and load data (via an operation that told the IDE to load the data at address X and send it with an interrupt). Its size is 16M words with 16 bit each. However, there are some problems:

- It can store ONLY 16 bit values
- It cannot be used as program/operation source memory (only as data source memory)
- It is extreme slow to set up, store and load data

The original thought behind the module was to enable self programing to some sort. Since it is too slow to write something more than a program adding some numbers this idea was dumped (sorry).

3. Basics of programming and operations

3.1 “Variables” and Values

If you have ever program with an “normal” programming language you should know that you can define variables, set them to a specific value and use them in calculations, if-then-else operations and more. A variable definition is just an abstraction layer above the physical address. The compiler handles the address for the variable when it should be loaded and when saved. In assembler you need to do it yourself: you cannot define variables or just let them handle by any sort of compiler. However, that is not a problem:

OP-Address	0x00	0x01	0x02	0x03
0x00	1ef0	1de1	1422	1763
0x04	18a4	1835	1f96	1c27

We have 8 universal register and 2 pointer registers. We only need to remember which register holds which variable. But there are some concessions: the first one is, that register 1 and 2 are ALU source registers and if we need to calculate something then we must store them somewhere else. The second is, that we can only store 6 variables effectively which is not enough for some more complex programs. But this is not a problem…

3.2 The RAM

We have MUCH more space to store values. We can send values to and from the RAM which can hold up to 2^24 values (which are 16.777.216 values)! More than enough even for a whole operation system and some processes. But how can we use this giant value storage? We need to address it. Each value in the RAM has one unique address with which the value can be load or stored. You can imagine it like a giant list: the first entry is called 0x0, the second 0x1, the third 0x2 and so on up to 16.777.216th in 0xFFFFFF (do not forget that we talk about the hexadecimal number system since it is simpler then to program as the binary system especially if we use big address numbers).

What we need to do is to set the address where we want to save our values. And for that we need to set the two pointer registers. They are merged and form one address for the RAM. If we want to clear our register 1 and 2, we need to choose two RAM addresses (in this example we use the address 0x0 and 0x1). The pointer registers are C and D with C being the one which holds the lower part of an address and D holding the higher one.

OP-Address	0x00	0x01	0x02	0x03
0x08	100c	100d	5010	101c
0x0c	5020			

Now we have copied the values in register 1 and 2 into the RAM at address 0x000000 and 0x000001. We can now just overwrite both registers with some values that needs the registers and we have still the values in the RAM. Just a small tip: if you write your own programs, you should create a text list with all values, there meanings and their address in the RAM. You can maybe remember 10 or 20 values with address but not 10.000 and more.

If we want to load the values back to the register, then we need the same procedure as above with only a minor change: operation 5 stores a value in the RAM but we need operation 6 which loads a value from the RAM. And do not forget to set the value address in the pointer registers.

OP-Address	0x00	0x01	0x02	0x03
0x10	100c	100d	6001	101c
0x14	6002			

And that is it! You now know how to load values, move them around into/from the RAM. But we cannot do much with them yet. And that means onward to calculation operations.

You maybe have seen that the position of the operations is handled in addresses too. For short: the operation addresses are handled like RAM addresses. But do not let you confuse by that; we will talk about operation addresses later.

If you know about addressing you may be wondering why we have 16 bit direct address space but 24 bit RAM address space. The reason is that we got 8 more bit thanks to the MMU but more about that in a while.

3.3 Time to calculate

It is time for operation 3: the ALU-operation-operation. With that we can load register 1 and 2 into the ALU, do an arithmetic (like add or subtract) logic (AND, XOR, …) or bit shift operation and save it to an register. The ALU can perform 16 different operations which are listed in the reference section. Now let us try it out! First, we need to store some values in register 1 and 2 and maybe some random values to the other regs.

OP-Address	0x00	0x01	0x02	0x03
0x00	1451	1a32	1945	1126

Now we can perform for example an addition (ALU operation number 1) with register 1 and 2. But we want to subtract (ALU operation number 2) register 6 from 6 too. We cannot define two source registers for the ALU with operation 3 (only the target register where the answer should be saved). We can only use 1 and 2 as source registers to calculate. That is why we first calculate the actual values from 1 and 2 together, save the answer in the free register 3 and then we move register 4 and 5 to 1 and 2 with operation 2, calculate with them and save the answer to register 4.

OP-Address	0x00	0x01	0x02	0x03
0x04	3103	2051	2062	3204

With operation 2 we can move one value from a register to another register. We do not need to save the values to the RAM if we have enough free registers (also we are much faster because we would need to set the pointer, save the values in the RAM, set the pointer again and load from the RAM which are 6 operations in the worst case!).

Now we know how to calculate and move registers around. We need only one more operation to make theoretical everything!

3.4 Jump, branches, and operation addresses

The branch operation! And here comes the fun: we have only programmed stuff that was executed linear. We could not go to another piece of code/operations somewhere in the operation memory, execute them and go somewhere else. You have maybe seen that each operation that we had written in the operation memory ROM has an address. We can use the same pointer that we have used to address the RAM to address the ROM! One small example:

OP-Address	0x00	0x01	0x02	0x03
0x00	100c	101d	9000	

If we execute the code, then it will jump to operation memory address 0x000100. The operation 9 is the jump/branch operation. If this operation is executed and all additional arguments are 0, then the “operation counter” jumps to the address which is in the pointer registers. The operation counter is the thing in the CPU that points to the next operation if the actual one is executed. The counter holds the address of the next operation in the operation memory. And with operation 9 we can set them to the pointer value.

But wait, there is more! It is cool that we can jump around in our code but what if we only want to jump if two values are equal? That is a branch: we only jump in a certain situation. These situations are ALU flags. The RX 8 can branch by 8 different situations: carry 8 bit (if the answer of an addition is larger than 8 bit), carry 4 bit, A <=> B (is set with the compare ALU function), left and right shift (a bit operation used to multiply and divide) and overflow (used when you calculate with negative binary numbers, it is a little bit buggy). These flags are stored during an ALU operation in a special register called flag register. It is only used for branch operations. The codes of the flags for operation 9 are described in the references.

Now to an example: we add two values and if the answer fired the carry flag (which it does if the answer is over decimal 255) then we jump to another piece of code. In that piece of code, we write another branch operation with another flag as condition which is not set in the flag register.

OP-Address	0x00	0x01	0x02	0x03
0x04	10cc	100d	1ff1	1ff2
0x08	3103	9010	0000	0000
0x0c	100c	101d	9200	

We see that the counter jumps during the first branch operation to 0x0c and ignores the second in 0x0e. The ALU operation has triggered the carry flag (0x01). The first branch had as jump condition the flag 0x01. The counter has detected that and jumps to the pointer. The second branch has the condition A > B which is not active in the flag register. It will be ignored, and the counter just goes to the next operation.

And that is it! These were all important CPU operations with examples and explanation. If you are still curios, then you can go to the next chapter and read about the more complex features of the computer or go to the chapter “Basic projects”.

4. Advanced operations

4.1 Output

If you read the last chapter, you know that we had only saw the results of our programs by checking the register and the RAM. This is good if we need to debug the programs, but a normal user cannot read that. As that we need to redesign the programs and a human readable output device. And the ERS has such a device: a terminal for ASCI characters on the external bus. We can load characters into the registers and send them to the module.

OP-Address	0x00	0x01	0x02	0x03
0x00	1021	148c	f010	169c
0x04	f010			

This program draws “Hi” on the terminal. First, we load the target module address of the terminal (which is 2) to register 1. Then we load a character to the pointer low register and send it with the operation “F” (send the pointer value to module address inside register x). The latter is repeated for the letter “I”. Why such a complicate “send to module” operation? Pretty simple: indirect module addressing is free real estate and make driver development much easier.

There is another module which can receive data from the CPU: the Not-an-IDE-controller with the address 4. You can use it as additional external memory (but not recommend since it is extreme slow) or for programming itself with an IDE (here I mean a development environment).

OP-Address	0x00	0x01	0x02	0x03
0x00	1041	1efc	113d	f010
0x04	1bec	114d	f010	102c
0x08	110d	f010	100c	120d
0x0c	f010	1ffc	113d	f010
0x10	1dec	114d	f010	105c
0x14	110d	f010	100c	120d
0x18	f010			

This program writes “beef” to external memory address 0x02 and “deff” to address 0x05. As before in the first step one register is set to hold the module address 4 (here it is register 1). Now follows a chain of operations that are send and executed by the Not-an-IDE-controller. The controller can be seen as an extreme simple CU with some registers striped to it and as that can be seen as an CPU. It is fed with operations which set the pointer for the memory and the register which holds the data.

As you see it is VERY slow and only for experimenting and development.

4.2 Interrupts and Input

If you have some experience with Logisim CPUs, you know that many have only a very crude I/O-system. One type of input in Logisim is the keyboard, which can be managed in two ways: polling or interrupts/event handling. The first means, that the CPU did not react directly on an input signal. Instead, it checks the input register every x cycles. This is simple but costs many clock cycles and CPU time.

The latter is the approach in the ERS 3864K. The RX 8 CPU can handle 32 different interrupts including 16 for external modules. If an interrupt is trigger (for example from a keyboard) the CPU directly stops the actual execution (but it finishes the actual active operation), saves the actual counter into the interrupt call pointer and jumps into the memory address with the interrupt handler. All of that is done in one operation cycle and with that much faster than polling.

That raised the question: where and what is the interrupt handler? It is just a piece of code in a program (obviously) which manages the value send by a module (even more obvious). We can freely decide where we want to write this routine, since we can set each interrupt target address individually. There is only one limitation: the interrupt controller is not connected with the MMU, which means that it can only point to the first 65536 words in the ROM. Although it is a little bit tricky to work with the interrupt controller it is a high pay giant gain feature of the CPU. But enough text walls, time for an example!

OP-Address	0x00	0x01	0x02	0x03
0x00	1001	110c	100d	7010
0x04	108c	100d	e200	107c
0x08	100d	9000	0000	0000
0x0c	0000	0000	0000	0000
0x10	1022	14fc	100d	f020
0x14	16fc	f020	166c	f020
0x18	10ac	f020	b800	

Onward to the description: First we need to set the interrupt for module 0 (which is the keyboard). This is done with one register which holds the interrupt number and the pointer which points to the handler routine. The operation “7” says: set interrupt number inside register x to pointer. Now we need to enable all interrupts which is done in a similar fashion like the operation “7”: operation “E” sets special registers (like the CPU mode register which we need to set) to the pointer value. The needed value to enable interrupts in the CPU mode register is 8. The final part of this code block is an endless loop.

The second code block, which is the interrupt handler routine (at address 0x10) draws the text “Oof” to the terminal with a line feed. The operation “B” jumps back from the interrupt and the 8 in the operation says that all interrupts are enabled after jumping back (interrupts are automatically disabled if an interrupt occur).

If we start the program nothing happened. But if we select the keyboard and type a random key, it will draw “Oof” to the terminal. This is repeatable thanks to the smart jump back from interrupt operation. This happened because the keyboard sends an interrupt request to the CPU and since there are no other waiting interrupt directly jumps to the interrupt pointer from module 0 (keyboard).

4.3 CPU modes

Now we can start with some comparisons to real life computers: imagine a computer with an operation system and some programs running on it. What happened if a malicious or faulty program is executed? Normally the operation system can handle the problem and shuts down the program. That is possible because the operation system has access to all functionalities of the CPU.

But how does it work? That is easy: we have a special register in the control unit called the CPU mode register. It holds the actual execution mode of the CPU. One of the bits in this register tells the CPU in which privilege mode it actual works. The RX 8 can differ between two privilege modes: Kernel mode (which means full privileges) and user mode (which means restricted access to CPU components). The magic comes with the fact that we can switch the modes during runtime.

With that we can capsule normal programs into the user mode while the operation system runs in kernel mode and can manage them. A program in user mode can never change to kernel mode by itself. But how can we get back to operation system and the kernel mode? With interrupts! If an interrupt is triggered, the CPU automatically changes to kernel mode. Normally the interrupt handling is completely under the control of the operation system which means from here we can just jump to OS routines. And if we want to go back to a program, we can just use the “jump back from interrupt” operation and set the additional argument to “2”. This sets the CPU mode back to user (the second bit in the CPU mode registers is used to indicate the actual execution mode).

Onward to the last two questions: what happened if a normal program in user mode tries to execute a kernel mode operation and how can a normal program use kernel mode features? First answer: it triggers interrupt number 0x10 “mode violation interrupt” which can be handled by the OS with a program shut down. If the program tries to access a forbidden register nothing happens. The actual internal architecture does not allow to trigger the MVI (it will be fixed in the future ERS systems). The second: the program can trigger a software interrupt called “syscall”. This interrupt is used to ask the operation system to do some stuff for the program like sending something to an output device or reserve more RAM for the program. The individual implementation and requestable functions of the syscall depends on the operation system.

OP-Address	0x00	0x01	0x02	0x03
0x00	1101	118c	100d	7010
0x04	110e	100f	b200	0000
0x08	0000	0000	0000	0000
0x0c	0000	0000	0000	0000
0x10	1011	3300	2001	2019
0x14	f000	111c	100d	9000
0x18	1028	20ca	20db	14ec
0x1c	f080	14fc	f080	10ac
0x20	f080	20bd	20ac	b200

First, we set up the mode violation interrupt. Then we use the interrupt call back operation to jump into a program (at address 0x10 - 0x17) and change the CPU mode to user. Do not be confused, a program is just a pile of code somewhere in the memory that do some stuff (office tools and video games included). The program continuously increments a value starting by 1. Shortly before the loop preparation and jump (with that I mean setting up the pointer and jump back to the loop start at 0x11) it tries to copy the actual value into a forbidden register and tries to send something to a module. The latter is a clear mode violation because the program needs kernel privilege to execute that. The interrupt is fired, and the interrupt handler writes “No” to the terminal. After that it jumps back to the program in user mode.

The register movements in the handler routine are to ensure that the pointer values from the program are saved before they get manipulated by the handler itself. They are moved back before the interrupt ends. With this the program is not affected by the interrupt in its execution flow. For more information about programs and program management read the chapter 6.

4.4 Virtual addressing

The end of the tunnel is near, onward to the last function that is provided by the ERS 3864K/RX 8 Lepton CPU: memory separation and virtual addressing. In the last chapter we have separate one program from some management routines via CPU modes. But how can we manage more than one program at the same time?

Yes, we could just pack some other programs in the memory and could change between them with the timer interrupt after x cycles in a simple management routine (more onto that in chapter 6). But this raises many more problems. How can we manage RAM accesses or more specific RAM spaces between the two programs? If our computer runs programs from many different developers, we need some sort of RAM space management between the two. If not one program could overwrite values in the RAM space which are later accessed by another program and cause havoc in both.

To solve that we have another encapsule trick in the sleeve: virtual address spaces. Every program starts at address 0 in the program memory and RAM. But what they do not know is, that this is only local (virtual) for their own program space. The program could be somewhere in the middle of the program memory, but it thinks it starts form zero in address space. In other words, we trick the programs into thinking that they have full control over the computer/are the only programs on the computer. To achieve this, we can use a translation table which translates the actual programs local addresses into real physical program memory and RAM addresses.

insert picture of how TLBs and address translation work here

The system behind that is simple: the lower pointer is directed directly to the ROM module (this part of the address is called offset) and the higher pointer goes into the translation table. This separates the memory into 256 Word pieces called “pages”. Each page can be reserved for exactly one program which means every program takes at least 256 Words of program memory space. But this is not a problem since we have more than enough program memory and RAM to compensate; we can just waste some words. Program pages do not need to be near each other in the memory. One page could be in the beginning of the memory and the other somewhere else and they will be seen as one code block from the view of the program. But this code fragmentation should be avoided (real HDDs can be slowed down significant when code is spread out in the memory).

The RX 8 has two separate translation tables for the program memory (ROM) and the RAM. These translation tables are just RAM modules which can be set with an operation. We can activate/deactivate local/virtual addressing via the CPU mode register. During an interrupt indirect addressing is deactivated which means, that the interrupt handler needs to stand at the first 64 Kbyte of the program memory space.

The last question: how can the operation system determine which pages belong to which program? This is a little bit tricky. During the installation the program tells the operation system how many pages of memory space it needs to operate. Then the operation system copies the program into some free memory space and saves the program pages to some internal RAM space (installing programs on the ERS 3864K is not possible since the program memory cannot be manipulated by the CPU itself, but you got the point. That is the reason why there are a program memory and a separate RAM). For more information about finding the right program pages read the FAT16/FAT32 memory management reference (sounds complicated but it is a simple standard).

The last major advantage we have with virtual addresses, is that we can dynamically set the physical address space. For example: the program itself needs 512 Words (2 pages) of program memory and 1 page of RAM. We cannot dynamically set the program memory space (it would mean that we move the program to a different physical address location), but we can dynamically set the RAM space needed by the program for saving variables. We can just allocate some random not actually used physical pages somewhere in the RAM. With that we can run programs with different RAM size and can reuse RAM space if a program is shut down. Only the operation system/the CPUs MMU needs to keep track which pages in the RAM are used and which not.

OP-Address	0x00	0x01	0x02	0x03
0x00	101c	100d	1013	c130
0x04	102c	100d	1023	c030
0x08	100e	100f	b300	

First, we set up both TLBs and then we jump into the program and enable virtual addressing and user mode. We load 0 into the interrupt call register to reset the counter for the program during the jump to the program. This is a good trick when you need to start programs outside the magic direct addressable 64KWords in the program memory as you only need to set the TLBs which are instant active after the interrupt call.

Now we can develop our program with user mode functions. But before we do this we need to know where to start in the memory. We have reserved one program memory page at address 0x000100 and one RAM page at address 0x000200. The local address space would be 0x00 – 0xFF for the program memory space and 0x00 – 0xFF for the RAM space. The physical addresses where we develop and execute the program would be 0x000100 – 0x0001FF in the program memory and 0x000200 – 0x0003FF in the RAM.

OP-Address	0x00	0x01	0x02	0x03
0x0100	1001	102c	100d	5010
0x0104	102c	100d	6001	3302
0x0108	5020	104c	100d	9000

This a simple program which increments a number and saves it to RAM address 0x02 local and 0x000202 physical. As you see we set the pointer to 0x0004 for the loop jump and not 0x0104. This can be a little bit confusing in future ERS programs but if you ever program with a high-level language it is a blessing because the MMU and operation system handles the virtual and physical addressing and the translation between them.

And this concludes the tutorial into all functions provided by the ERS 3864K and the RX 8 Lepton CPU. Now you can develop you own programs or even operation systems. If you want some other projects, then you can go to chapter 5 and 6.

5. Basic projects

5.1 Calculating the Fibonacci numbers

The Fibonacci numbers are a number chain which describe a perfect natural spiral. Snail shells follows this spiral like many other things in nature. The algorithm is simple: we add two numbers (we start with 0 and 1), save the answer into a third register, move one summand over another and finally move the answer into the register which holds the summand which was moved.

Iteration cycle	Register Accu	Register A	Register B
0	1	1	0
1	2	1	1
2	3	2	1
3	5	3	2

This is easily done with some register move operations

OP-Address	0x00	0x01	0x02	0x03
0x00	1000	1011	1002	3100
0x04	2012	2001	103c	9000

And that is it! If we going to the register, then we will see that the program behaves like in the table above. But there are two limitations: the bit width of the ALU and the lack of displaying the result. The first means that we can only calculate up to 255. For the algorithm it means that the highest number we can directly calculate is 233. The latter means no human readable numbers on the terminal. There is actual a third problem (which depends on the second): the result of the calculations is binary and not decimal.

It is a little bit complicated to solve these problems and that's why the continuation is in the Advanced projects chapter.

5.2 Input to Output

If you had tried the CPUTest program you maybe wondered how the Input/Output mirroring works. Well, we will discuss it here! It is a simple interrupt handler which copies an input value to the pointer to the terminal.

OP-Address	0x00	0x01	0x02	0x03
0x00	1020	10ac	100d	7020
0x04	108c	100d	e200	107c
0x08	100d	9000	20c3	20d4
0x0c	d400	f000	203c	204d
0x10	b800			

First, we set up the interrupt for the keyboard (address 0x02). Then we activate the interrupts and go into an endless loop to wait for user input. If one key is pressed, the interrupt handler is called which saves the actual pointer into some free registers, load the interrupt value into the pointer, sends it to the terminal module, restores the pointer back from the "cache register" and jumps back (with the argument to reactivate the interrupts).

Now we can start the program and it will copy our keyboard input to the terminal. Sometimes letters are “sucked” into nirvana. This Is a known bug with the interrupt controller.

5.3

6. Advanced projects

6.1 A more advanced Fibonacci program

Here we try to fix the problems that we had in the previous iteration of the program. First, we need to solve how we can use more than 8 bit to calculate. Simple answer, we cannot. But we can use the carry flag of the ALU to detect if the value gets too large. And here comes the principle of pointers: we reserve two registers for one value and if a lower part value calculation activates the carry flag, then the higher part value calculation is performed with + 1. We have only a small amount registers, but this is okay, since we have a giant RAM to store values. With that we create some sort of “array”.

The second and third problem are a little bit harder to solve. We need to convert the binary value to BCD which is a decimal representation in binary. It follows the binary representation up to value number 9. After that we need another 4 bit to represent a 10. All values between 0xA and 0xF are illegal. To display each digit, we just need to add 0x30 to it convert the BCD to an ASCI character of the same value (0x30 = ASCI 0, 0x31 = ASCI 1, 0x32 = ASCI 2, …). All we need to do now is to send all digits with the highest one first.

Yes, we can calculate in pure BCD, but we would waste the higher 4 bit which the ALU could calculate instant if we use the pure binary system. The ALU cannot directly calculate with two BCD numbers in parallel nor has the functionality to process even one BCD number natively. So, it would take many cycles to process one BCD calculation because the CPU needs to check itself if the result is over 0x9 and correct it. This is not a problem because we can just calculate in binary and convert the result into a BCD number to display it on the screen.

6.1.1 Core algorithm

We start to develop the algorithm with the “use more registers and more RAM to get more than 233” function.

OP-Address	0x00	0x01	0x02	0x03
0x00	4000	1010	5000	101d
0x04	5000	1008	100d	4000
0x08	207c	100d	6003	101d
0x0c	6004	1011	2062	3500
0x10	2031	2042	119c	100d
0x14	9400	3100	11ac	100d
0x18	9000	3b00	2034	2003
0x1c	1000	207c	100d	5030
0x20	101d	5040	1006	100c
0x24	101d	9010	2071	3300
0x28	2007	2071	2082	3500

0x2c	108c	100d	9400	9800
0x30	1007	100c	102d	9000

We initialize the registers with some index, reference and start values and write the number "1" into the first RAM address of the address space of variable "A" and reset the first RAM address of the address space of variable "B". Before we calculate we load the actual byte values from the RAM, add them together with a potential carry from a lower byte (which is saved in register 6, it is a simple if then else routine to switch between A + B and A + B + 1), save them back into the RAM and then go into the management routine. The register 7 is used to tell the program at which binary digit it actual works (we call it index value). The register 8 holds the reference index which means that it says how many bytes the actual numbers use. If the index value is over the reference value, we reset the index for the next calculation process and jump to the "convert to BCD" function (marked blue). This can be interpreted as an "for loop" from high level languages. Otherwise, it will increment the index and jump back to the loop start (the orange is the jump instruction, the yellow is the entry point of the loop). If the actual calculation throws a carry, then it will jump into the "create higher digit or add to higher digit" function.

As you can see it is hard to calculate values with more than 8 bit width with a CPU that does not actively support it. It makes the CPU slow and as such only for some small projects recommend.

6.1.2 Create higher digit or add to higher digit

In this function we need to compare the actual index with the reference value to find out if we just need to set register 6 to one (which leads to a simple carry) or if we need to create a new higher byte and update the reference value.

OP-Address	0x00	0x01	0x02	0x03
0x0100	2071	2082	3500	10dc
0x0104	101d	9400	1016	2071
0x0108	3307	108c	100d	9000
0x010c	2081	3308	2071	3307
0x0110	207c	100d	1000	5000
0x0114	101d	5000	1016	108c
0x0118	100d	9000		

We first compare the actual index with the reference. If they are the same, then the index and the reference are incremented, and the next RAM locations are cleared (a new higher byte is created). Otherwise only the index is incremented. In both ways the register 6 is set to 1 to inform the routine that it needs to add + 1 to the result in the next run.

6.1.3 The BCD converter

To convert the binary numbers into a BCD coded number, we use the double dabble algorithm. It shifts the binary number to the left, then it takes all outshifted bits and packs them into a new byte (array). If one of these new bytes contains a number over 5 the algorithm will add 3 to it before shifting again. These new bytes array is the final array containing the BCD numbers. To make things simpler we use one byte for one BCD number. To achieve this, we use the 4 bit carry of the ALU to detect if an addition triggers an 4 bit overflow and a simple comparison to shift the fifth bit to the next BCD byte.

OP-Address	0x00	0x01	0x02	0x03
0x0200	100c	100d	e800	1007
0x0204	2071	207c	100d	6003
0x0208	102d	5030	3307	2082
0x020c	2071	3500	115c	102d
0x0210	9200	2017	104c	102d
0x0214	9000	1007		

First, we copy the variable "A" to another memory location to manipulate it without disturbing the Fibonacci calculating routine (start of the function up to 0x0214).

6.2 Sorting algorithms

Sorting algorithms are developed to sort numbers in an array in a fast and simple way (obviously). Here we focus on algorithms sorting number dependent on their value (from the smallest to the largest).

6.2.1 Preparing the unsorted numbers

First, we need to write some unsorted numbers in the RAM. Since the ERS does not have any random number generator we need to write them manually. Either you create a RAM image by yourself by using an extra circuit, write a small external program to create an image, write some random values manually into the RAM or write an ERS program with some weird routines (for example with the timer interrupt) to generate the numbers. The choice is yours (I wrote some manually in the RAM and saved it because I am that kind of a mad man).

6.3 Task scheduling… and tasks

It is time to use the highlight functionality of the ERS 3864K in a project: the virtual address mode and timer interrupt. In this example we will write a simple task scheduler and some programs running in their own virtual address space.

6.3.1 The scheduler

We start with the task management routine and scheduler. This routine must switch between the tasks (context switching), set up the TLBs and interrupts, react to task triggered interrupts and start/stop them.

7. Adding modules

To add modules to the ERS 3864K you need to accomplish two things: they need to react on CPU commands and need to use the interrupt system to send data to the CPU. The first is simple as your module only needs to react to an address send from the CPU, which can be done with a comparator. The data must be stored during the two cycles of the sending operation, at best during the raising phase of the second cycle.

The latter one is a little bit trickier. The module needs to store and hold its interrupt request with data if either a higher priority module is sending a request and/or the interrupts are masked. It needs to handle higher priority requests instantly since it is possible that another module tries to send something at the same time. On the other hand, it needs to activate and hold its interrupt request for exactly two clock cycles. During that time the module sends the data, its address, the daisy chain signal, and a store external value signal to the CPU.

Since the interrupt trigger is highly dependent on the type of module you build, I cannot give you universal examples. But I can recommend you to take a closer look at the internals of the not-an-IDE controller.

8. References

Operation (pattern: 4 bit operation, 4 bit argument/operation extension, 4 bit source register address, 4 bit target register address)

Hex.	Structure	Mode	Description
0	0 0 0 0	User	No operation
1	1 V V T	User	Store Value to register Target
2	2 0 S T	User	Move value from register Source to Target
3	3 A 0 T	User	Perform ALU-Operation and save the result to register Target
4	4 0 0 0	User	Delete user register (except pointer)
5	5 0 S 0	User	Store register Source to RAM, address pointer
6	6 0 0 T	User	Load from RAM address pointer to register Source
7	7 0 S 0	Kernel	Set interrupt register Source to pointer
8	8 0 S 0	User	Execute interrupt register Source
9	9 S S 0	User	Branch if one Status in register flag is active
A	A 0 0 0	User	Load counter into the pointer
B	B A 0 0	Kernel	Jump back from kernel with Argument
C	C A S 0	Kernel	Set TLB Argument, address register to Source to pointer
D	D A 0 0	Kernel	Load special register Argument to pointer
E	E A 0 0	Kernel	Save from pointer in special register Argument
F	F 0 S 0	Kernel	Send pointer to module register Source

Registers

Adr.	Mode	Desciption
0	User	Universal register
1	User	ALU source register
2	User	ALU source register
3	User	Universal register
4	User	Universal register
5	User	Universal register
6	User	Universal register
7	User	Universal register
8	Kernel	Shadow/universal register
9	Kernel	Shadow/universal register
A	Kernel	Shadow/universal register
B	Kernel	Shadow/universal register
C	User	Pointer low
D	User	Pointer high
E	Kernel	Interrupt call low

F	Kernel	Interrupt call high

ALU-Operations

Hex.	Variables	Description
0	A	No operation, can be used to zeroth the target register
1	A, B	Add
2	A, B	Subtract
3	A	Increment
4	A	Decrement
5	A, B	Compare
6	A	Invert
7	A	Negate (NOT)
8	A, B	AND
9	A, B	NOR
A	A, B	XOR
B	A, B	Add with increment
C	A	Left shift
D	A	Right shift
E	A	Left roll
F	A	Right roll

Flags

Hex.	Description
1	Carry
2	Overflow
4	4 bit Carry
8	Left shift bit
10	Right shift bit
20	A > B
40	A = B
80	A < B

Special register

Adr.	Description
1	Timer interrupt reference holder
2	CPU mode register
4	External value register (for interrupt values)
8	Flag register

CPU modes

Hex.	Description
1	Activate virtual addressing
2	CPU mode (kernel/user)
4	Activate timer interrupt
8	Demask interrupts

Interrupts

Hex.	Interrupt
00	Module 0 (Keyboard)
01	Module 1
02	Module 2
03	Module 3
04	Module 4 (Not-an-IDE-controller)
05	Module 5
06	Module 6
07	Module 7
08	Module 8
09	Module 9
0A	Module A
0B	Module B
0C	Module C
0D	Module D
0E	Module E
0F	Module F
10	Mode violation
11	Free (for hardware)
12	Free (for hardware)
13	Free (for hardware)
14	Free (for hardware)
15	Free (for hardware)
16	Free (for hardware)
17	Free (for hardware)
18	Free (for hardware)
19	Free (for hardware)
1A	Free (for hardware)
1B	Timer
1C	Free (for software, e.g.: inter process call)
1D	Free (for software, e.g.: system call)
1E	Free (for software)
1F	Free (for software)

TLBs

Hex.	TLB	Address space
0	RAM	64Kbyte In/16Mbyte Out
1	ROM	64Kbyte In/16Mbyte Out

Modules

Adr.	Category	Description
0	In	Keyboard
1	-	Free
2	Out	Terminal
3	-	Free
4	In/Out	Not-an-IDE-controller
5	-	Free
6	-	Free
7	-	Free
8	-	Free
9	-	Free

A	-	Free
B	-	Free
C	-	Free
D	-	Free
E	-	Free
F	-	Free

Keyboard management

Does nothing with send data. Can only send data to the CPU via interrupt (ASCI values on the lower 8 bit)

Terminal management

Accepts ASCI data (only one at a time in the lower 8 bit) and prints it. Does not send data back to the CPU.

Not-an-IDE-controller management

Accepts Harvard-like “operations” to process them internal (full 16 bit, pattern: 4 bit operation, 4 bit target register address, 8 bit data). Can send data back to the CPU via interrupt (full 16 bit).

Not-an-IDE-controller Operations

Hex.	Structure	Description
0	0 0 0 0	No operation
1	1 T V V	Load Value into register Target
2	2 0 0 0	Save register data into external RAM, address register pointer
3	3 0 0 0	Load data from RAM address register pointer, send data via interrupt

Not-an-IDE-controller Registers

Hex.	Description
0	Pointer low
1	Pointer mid
2	Pointer high
3	Data low
4	Data high