



<https://renerocksai.github.io/rrisc>

RRISC CPU

BUILDING A CPU OVER CHRISTMAS

RENE SCHALLNER



0

What and why?

WHAT IS A CPU AND WHY WOULD I WANT TO BUILD ONE ANYWAY?

What is a RRISC CPU and why would I want to build one anyway?

CPU

- > a CPU (Central Processing Unit) is a "microprocessor"
- > the heart of a computer

CISC, RISC, RRISC

- > early - mid 1970s : trend from simple register machines towards more and more complex architectures: **CISC**
(*Complex Instruction Set Computer*)
- > late 1970s - 1990s: counter trend after critical observation towards less, and simpler instructions: **RISC**
(*Reduced Instruction Set Computer*)
- > early 1990s: out of necessity, I come up with: **RRISC**
(*Radically Reduced Instruction Set Computer*)

Main Differences RISC -> RRISC

- > **No ALU** (arithmetic logical unit) inside the core of the CPU
- > **No stack**
- > 5 basic instructions, with variations (*addressing modes*)
 - > LD (load from memory)
 - > ST (store to memory)
 - > IN (read from port)
 - > OUT (write to port)
 - > JMP (jump to address)
- > ALL instructions are conditional
(=> Turing complete)
- > identical execution time of all instructions

What is a RRISC CPU and why would I want to build one anyway?

WHY?

- > Because!
- > I always wanted to know how CPUs work anyway
 - > best way to learn is to attempt to build one
 - > from the ground up
- > Original motivations:
 - > for learning and educating about CPU design
 - > De-mystifying CPUs

Might it even be relevant?

RISC is back in business!

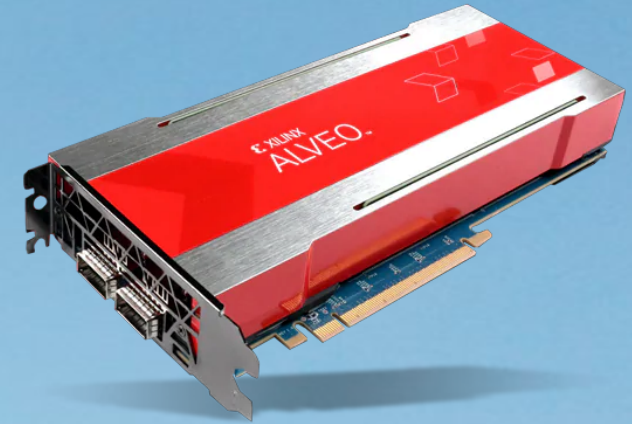
- > **ARM** (Acorn RISC Machine) cores power billions of smartphones today
- > **Apple's new** ARM based **M1 chip**
- > Open **RISC-V** architecture is aiming to become "the new ARM" - without license fees!

As we will see later, we will implement the CPU using an FPGA chip.

FPGAs are accelerating Artificial Intelligence!

- > **In the data center:**
 - > Xilinx Versal Adaptive Compute Acceleration Platform, Alevo FPGA boards
 - > Intel Deep Learning Acceleration, Arria 10 FPGAs -- **powering Bing search!**
- > **On the edge:** mobile, IOT, self-driving cars, robotics, ...
- > **FPGAs are parallel "by nature":**
 - > electronic circuits are parallel by nature
 - > they lend themselves to accelerating parallel algorithms:
 - > convolutions, matrix multiplications, etc.
- > **Dedicated silicon is power efficient:**
 - > 38..42W typically in Intel Vision Accelerator design with Arria 10 FPGA versus ~250W in NVIDIA V100 GPU

Image source: Xilinx



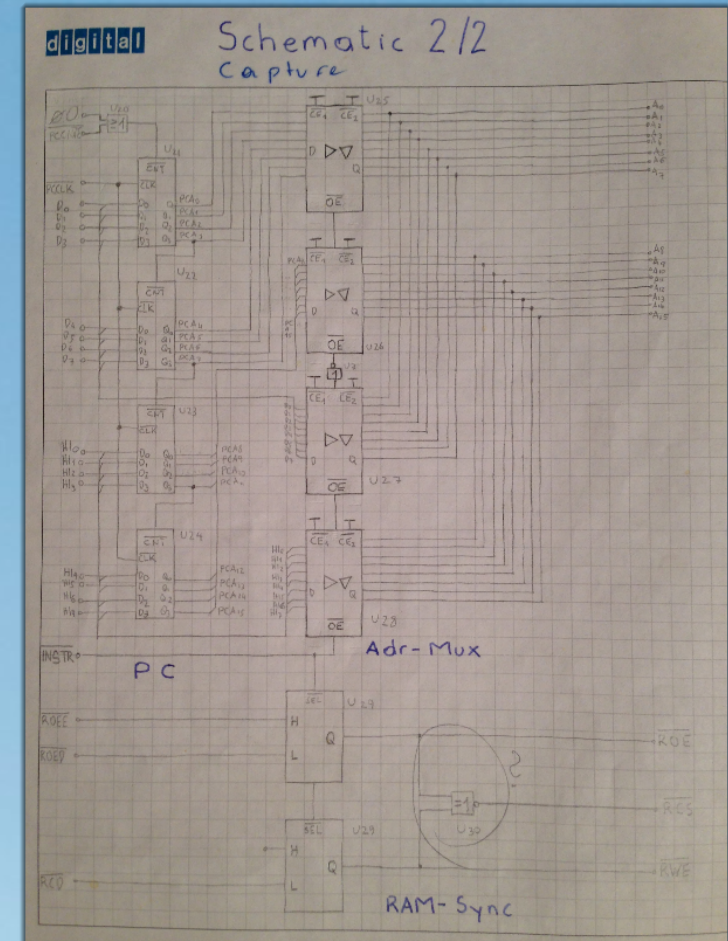
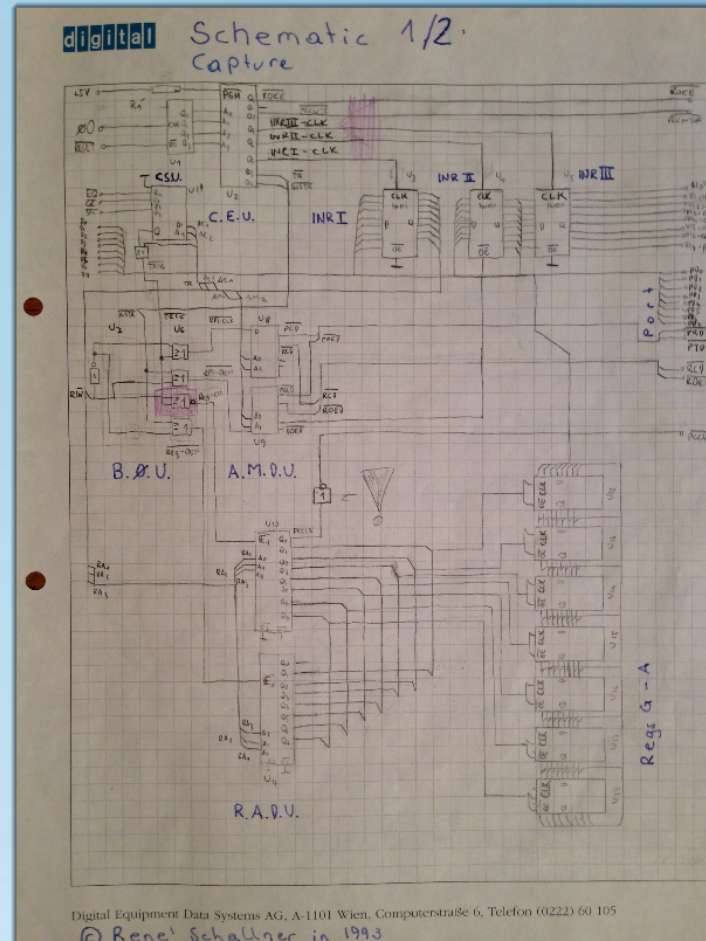


1

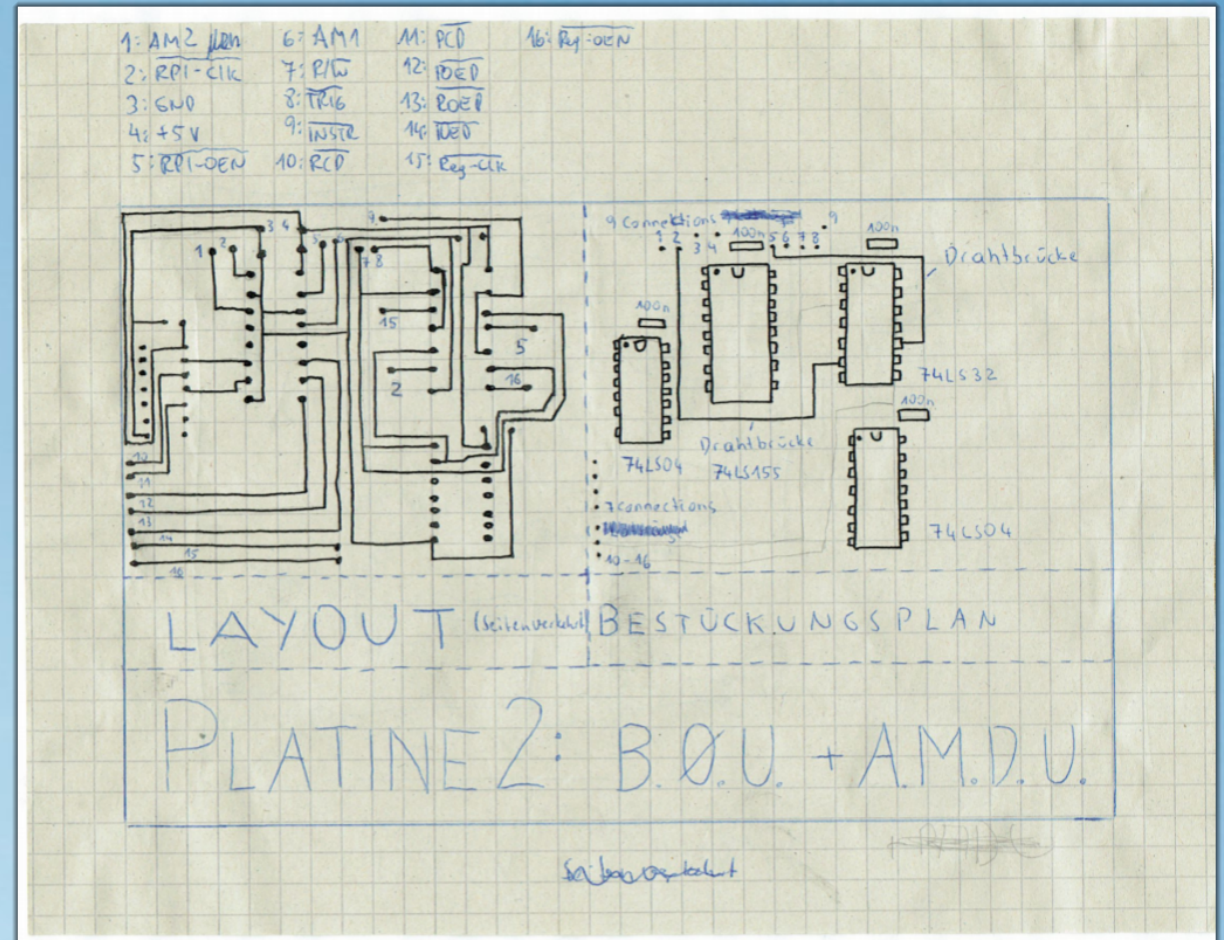
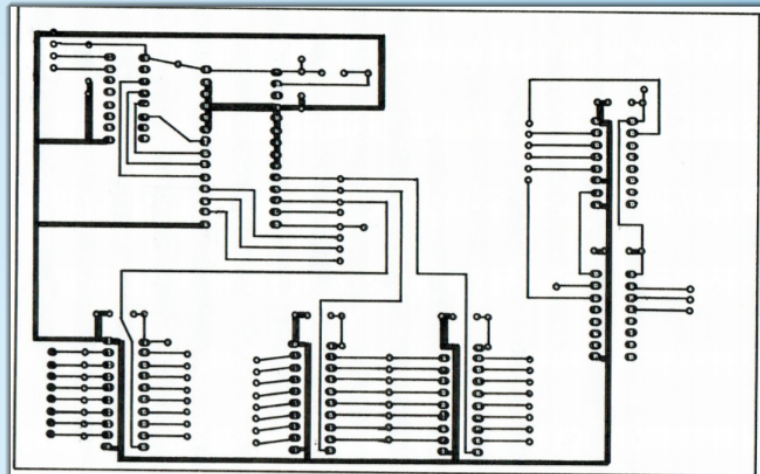
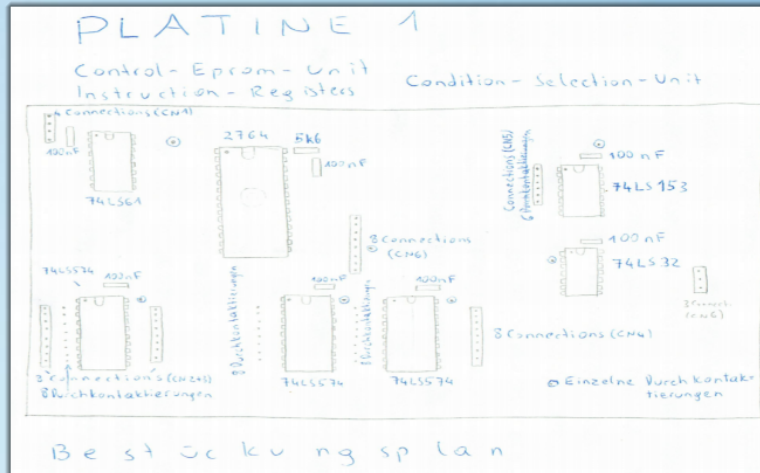
Days 1-3: learning

Starting off with some 30 years old schematics

- > I designed the RRISC CPU in the early nineties
- > It was intended to be implemented using standard TTL logic circuits (LS 74XX)
- > I never got around to actually build it



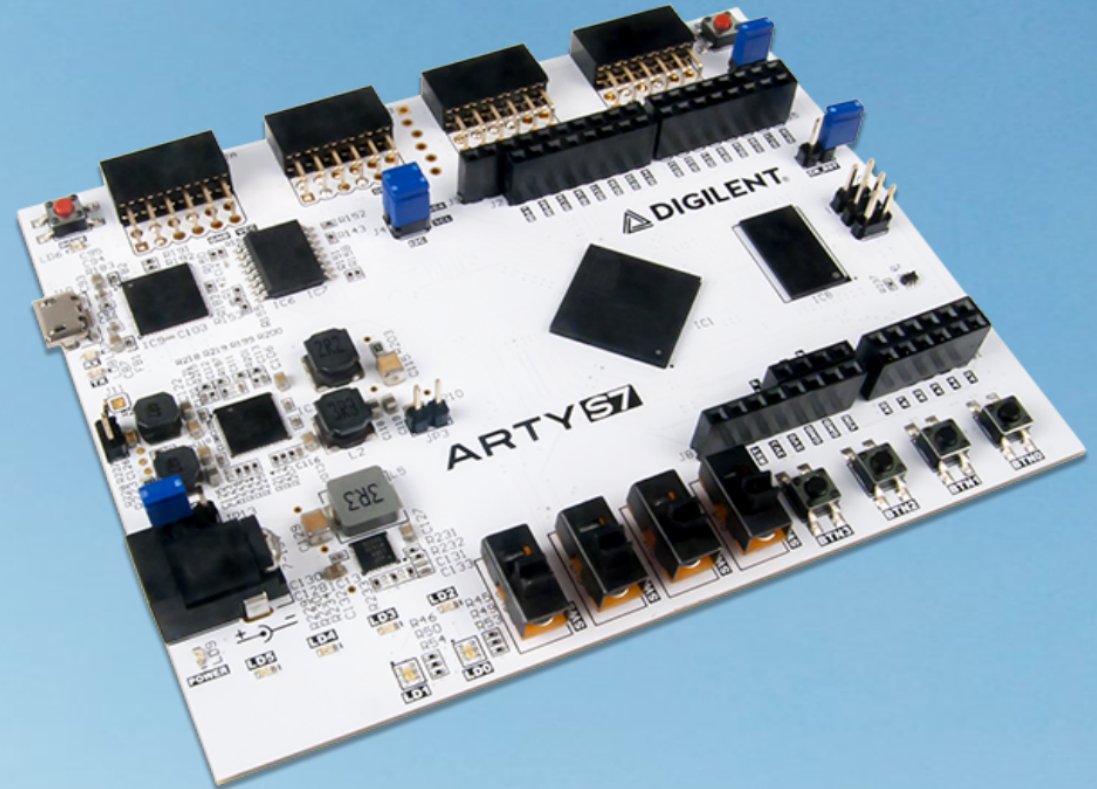
More old design stuff



Back to today: Let's order an FPGA development board

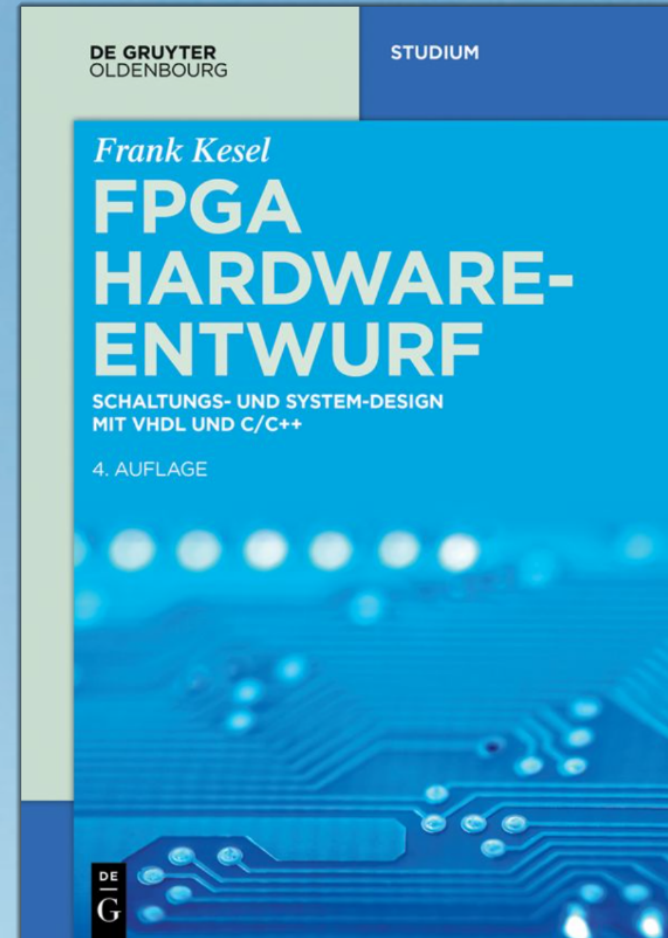
- > Intention: upgrading the implementation to present times technology
- > The educational requirement of using discrete logic ICs for the CPU is outdated
- > FPGA ... Field Programmable Logic Array
- > FPGAs enable us to load very complex digital circuits onto a programmable chip

GOAL: Have the CPU run a demo program here --->



Time to learn VHDL

- > We order an ebook about FPGAs and VHDL
 - > VHDL ... VHSIC-HDL
 - > VHSIC ... Very High Speed Integrated Circuit
 - > HDL ... Hardware Description Language
 - > **VHDL allows us to describe digital circuits in a textual representation** - a bit like programming
- > We spend the rest of the week reading and learning





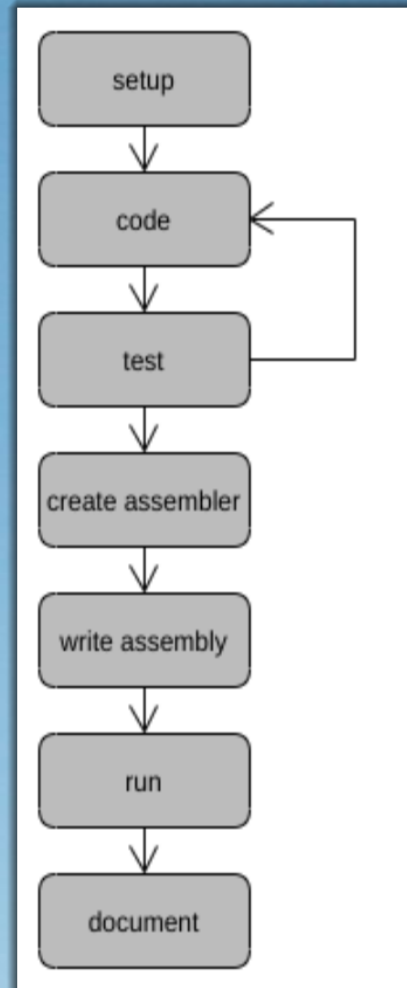
2

Days 4-13: doing

The plan

There was a rough plan of what needs to be done:

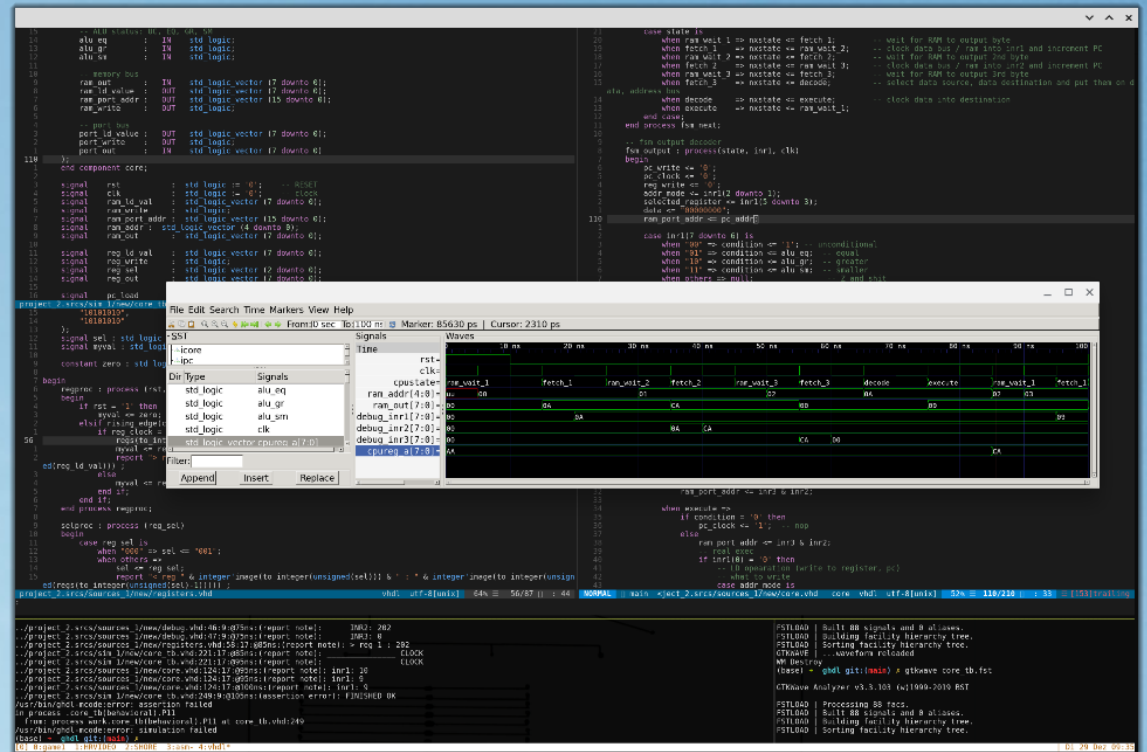
- > **Setup:** Install and familiarize with Xilinx Vivado FPGA design suite for programming the FPGA
- > **Code:** Write the VHDL files of the CPU core components
- > **Test:** Write VHDL "testbenches" to verify the components via simulation
- > **Repeat** code & test for peripheral components:
 - > RAM
 - > ALU (arithmetic logical unit)
 - > Arty development board (LEDs, buttons, switches)
- > **Create** a macro-assembler
 - > translates RRISC assembly language into binary machine code
- > **Write** assembly code for
 - > ALU verification
 - > assembler macro capability examples
 - > sample code
 - > the demo program running on the board
- > **Run:** get the code translated and programmed into the FPGA
- > **Document:** create a web page documenting the CPU and project



Software stack change

- > Turned out Vivado was bad for my working style
 - > slow, resource-hogging Java GUI
 - > (neo)vim is a far better editor
- > Surprise: High quality open source VHDL software exists!
 - > (neo)**vim** is my preferred editor
 - > **ghdl**
 - > compiles VHDL electronic designs to native code simulations
 - > **gtkwave** is a viewer for waveforms
 - > such as created by ghdl

- > Editing, simulating, and testing of VHDL designs
- > right in the terminal, no mouse needed (viewing waveforms requires a mouse)



The core CPU components

The core components of the CPU:

- > **Control Unit**

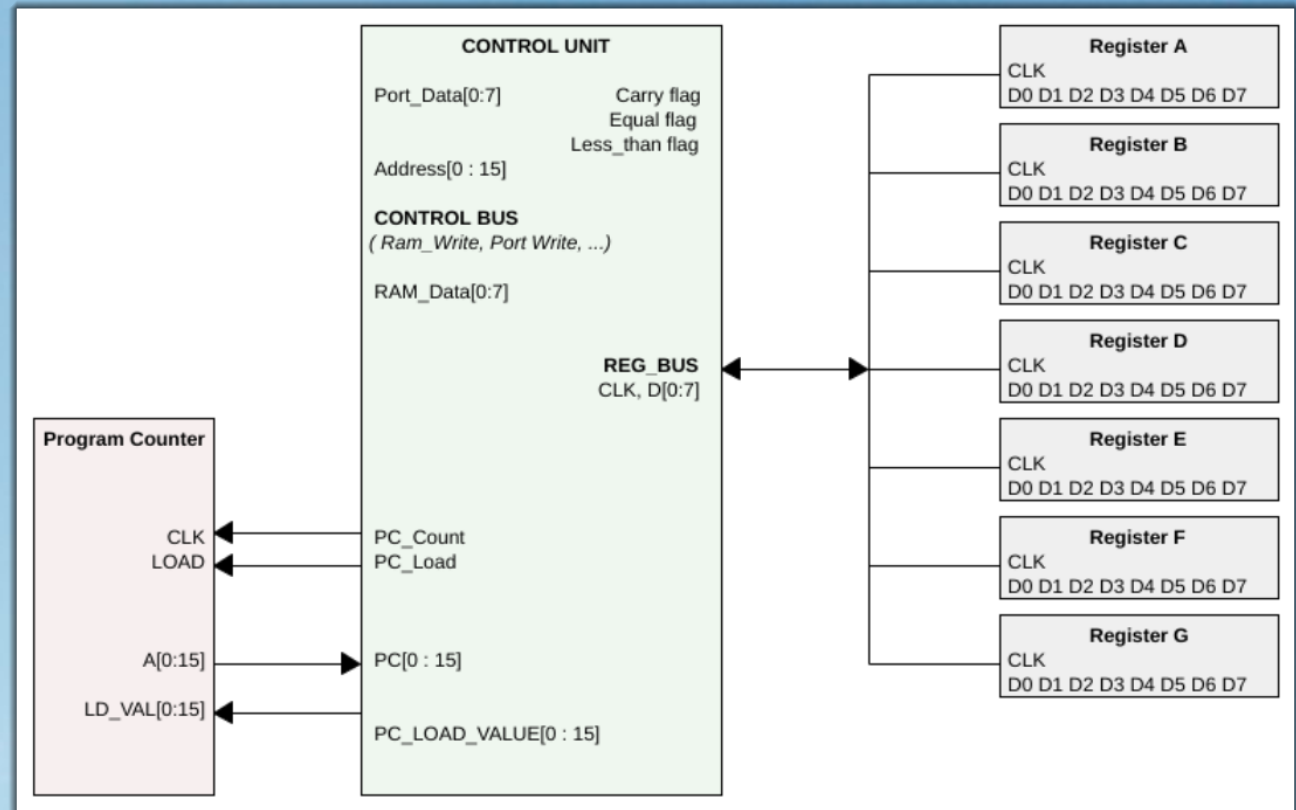
- > does all the work
- > *"the CPU inside the CPU"*

- > **7 registers:** A, B, C, D, E, F, G

- > for storing values

- > **Program Counter**

- > provides address ("location") of current instruction in memory



When we implement these, we have a CPU, legally speaking.

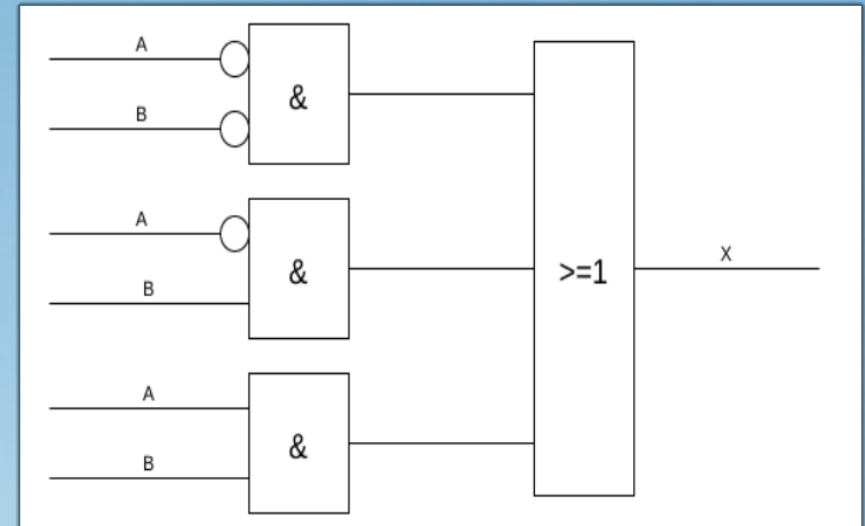
The CPU inside the CPU -- 1: combinatorial logic

- > Combinatorial logic implements a function:
 - > from the **combination** of binary inputs, a set of binary outputs is derived
 - > the function can be described as a **function table**
 - > combinatorial logic is built from basic logic gates:
 - > AND, OR, NOT
 - > they are easily translated into basic electronic circuits with transistors

Example:

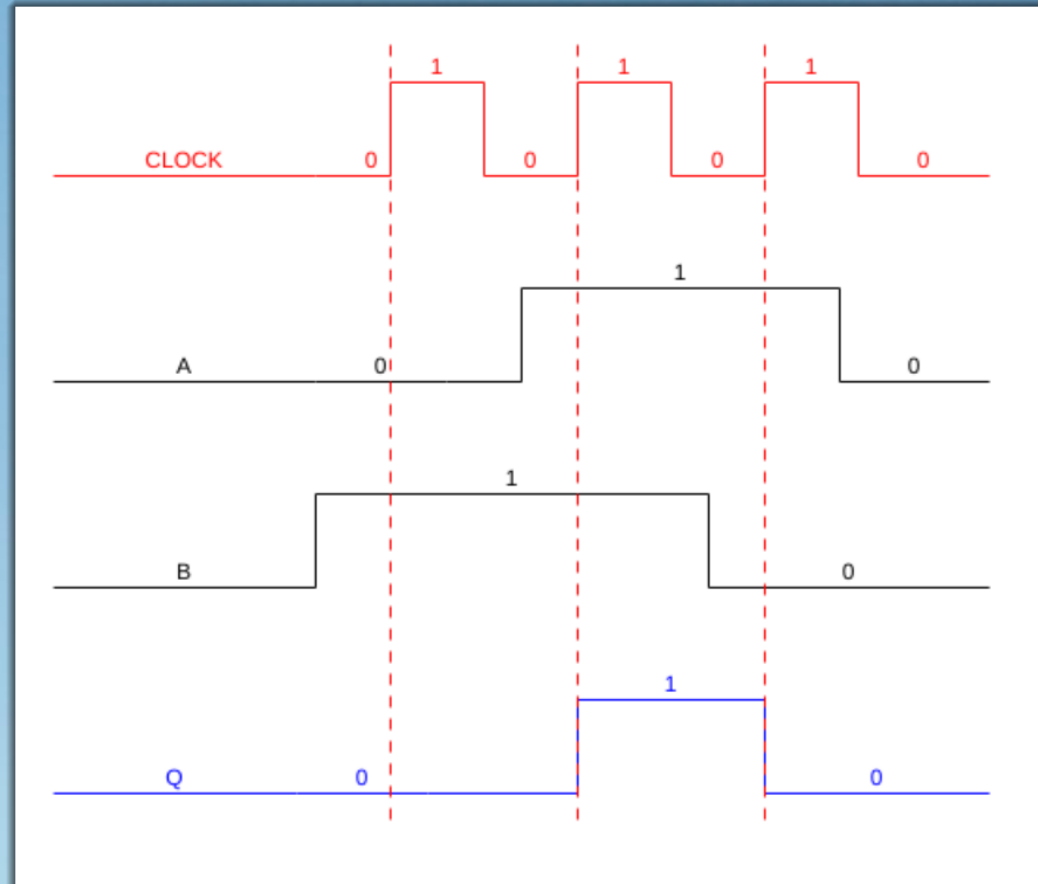
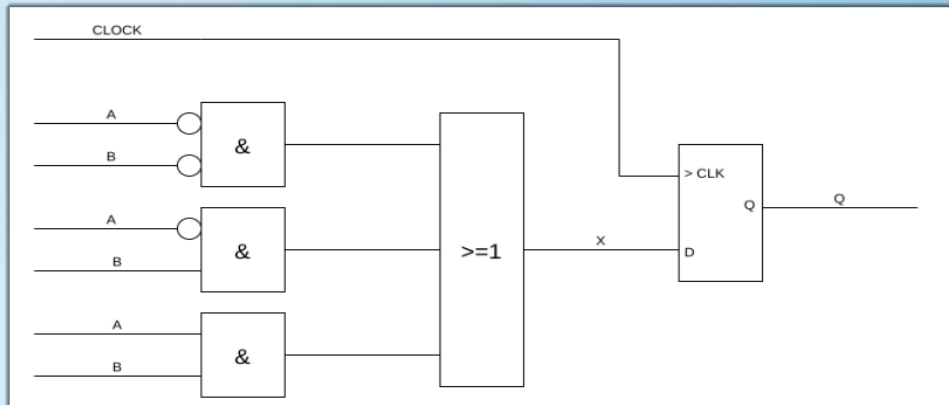
$X = (\{\text{NOT } A\} \text{ AND } \{\text{NOT } B\})$
 $\text{OR } (A \text{ AND } \{\text{NOT } B\})$
 $\text{OR } (A \text{ AND } B)$

Input A	Input B	Output X
0	0	1
0	1	0
1	0	0
1	1	1



The CPU inside the CPU -- 2: adding state

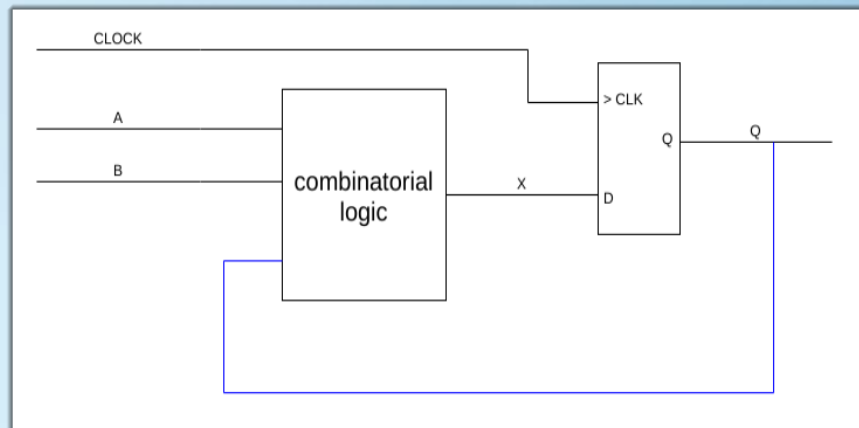
- > When we add a **D-flipflop** behind the combinatorial logic ==> the circuit becomes **stateful** ==> **time-dependent**
- > A **D-flipflop** is a 1-bit memory:
 - > when a CLOCK signal (rising edge*) arrives
 - > its output becomes the input at this very moment
 - > the output does not change even if the input changes
 - > this state is kept until the next CLOCK signal arrives
 - > *) *rising edge: level change from 0 to 1*
 - > such flipflops are easily built from basic logic gates



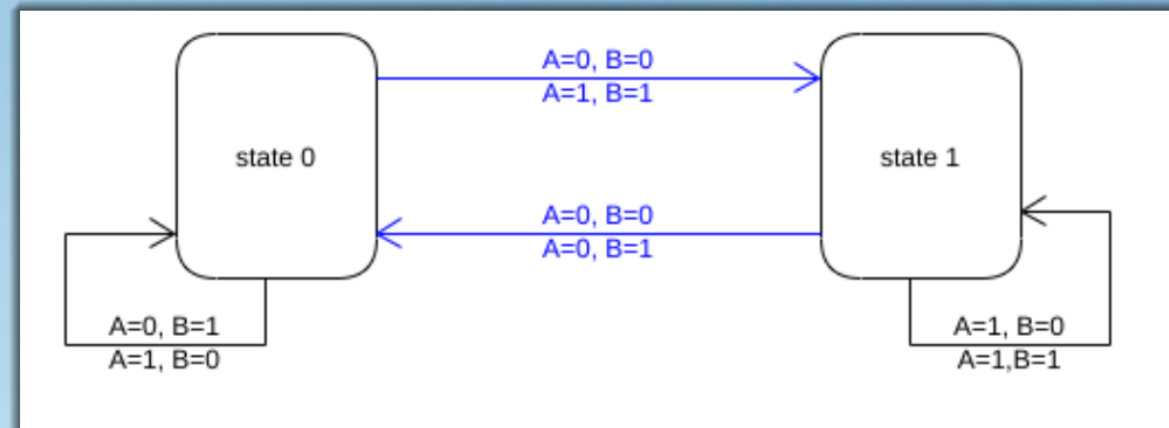
The CPU inside the CPU -- 3: Finite State Machines

- > When we **feed back the state**
==> we get a **finite state machine**
- > In a finite state machine
 - > the output state is dependent on:
 - > the *condition* (input signals A, B)
 - > the current state
 - > you can jump from state to state, given a condition

Example:



Input A	Input B	current state	next state Q
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

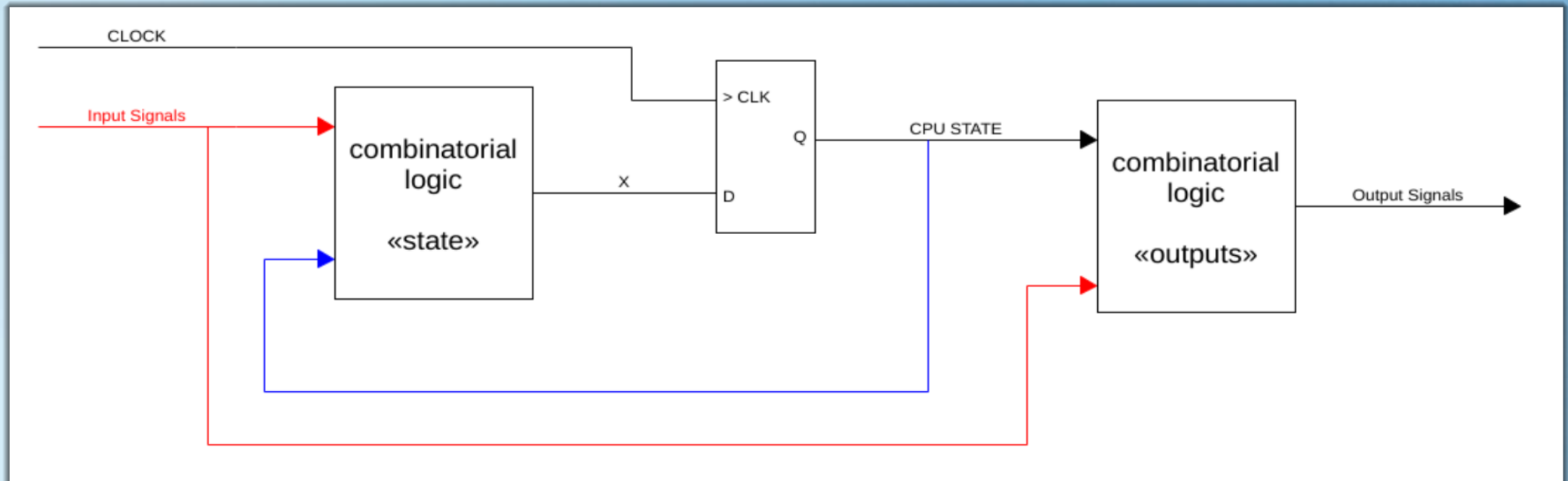


The CPU inside the CPU -- 4: Control Unit

In the final control unit of the CPU

- > we first use an FSM to define the CPU state
- > we use an additional combinatorial logic circuit to decode the combination of state and input signals into the desired output signals

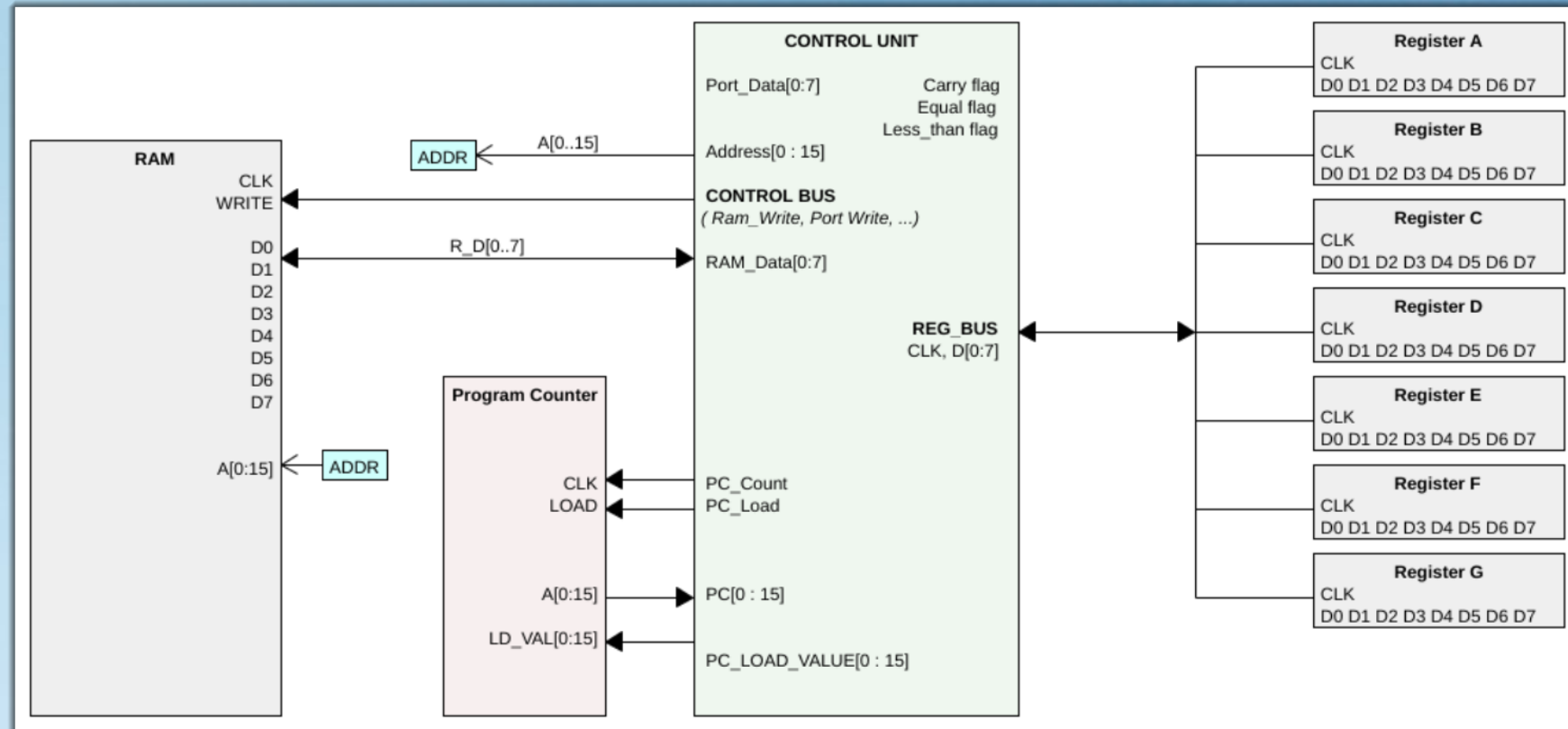
Separating CPU state and output signals makes for a cleaner, more easy to maintain design.



A minimalistic computer - for testing the CPU

Without any memory, there's nothing to execute.

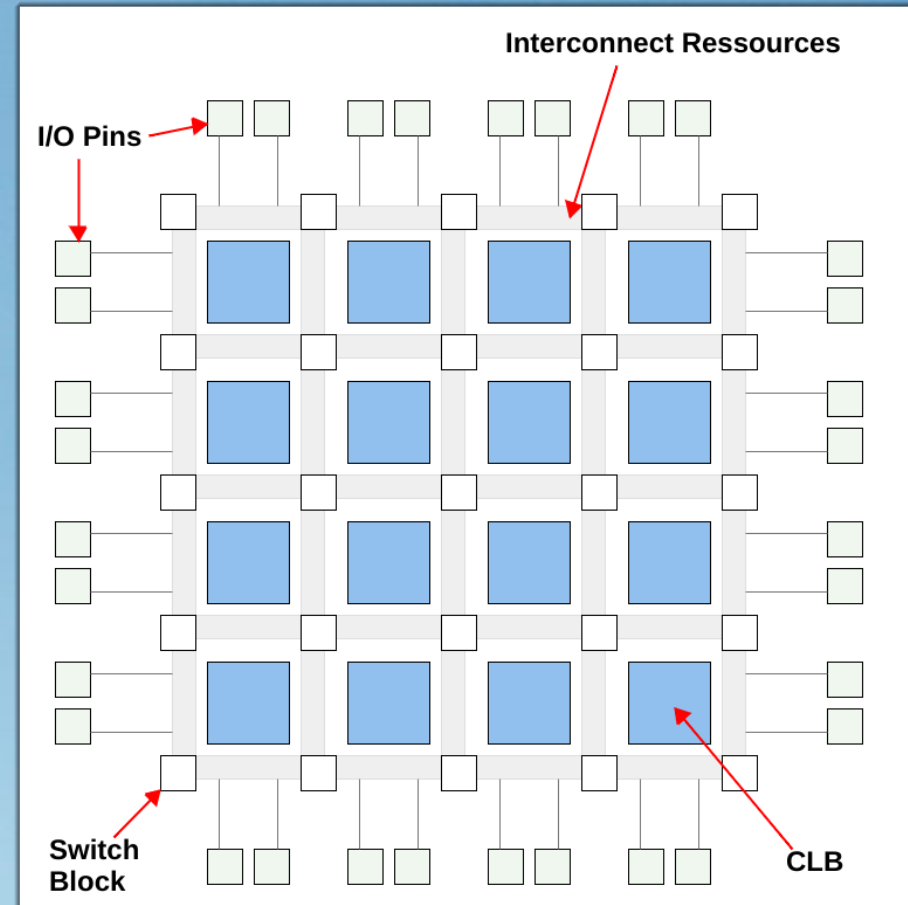
Hence, for testing **we have to add some RAM!**



How would we get this onto an FPGA?

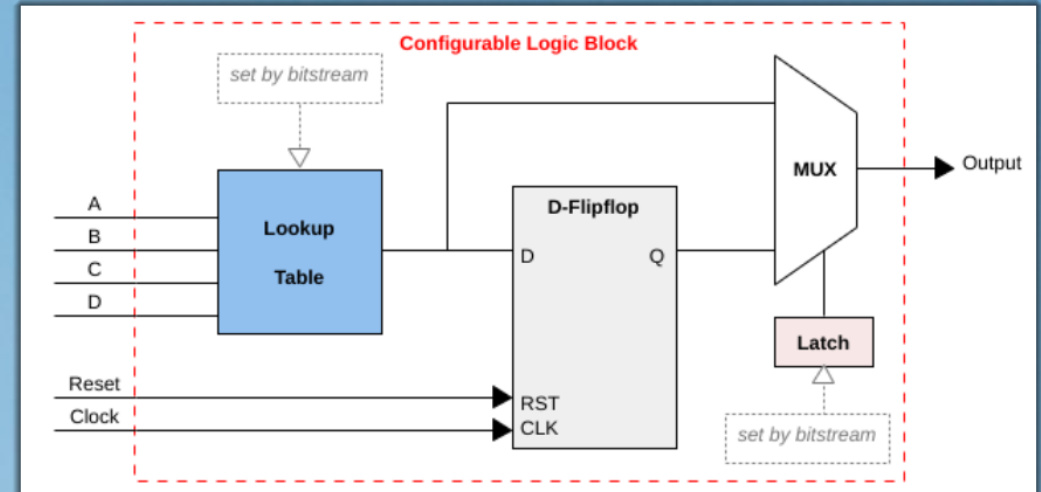
Intermezzo: FPGAs

- > Consist of a 2-dimensional array of **Configurable Logic Blocks** and interconnections between them
- > Both CLBs and interconnections are programmable
- > CLBs can be programmed to implement very small functions and are connected by programming the switch boxes
- > Electronic designs are broken down into very small functions which are then connected
- > FPGAs enable us to "load" very complex digital circuits onto a programmable chip
- > FPGAs are also used in prototyping ASICs (custom chips)



Configurable Logic Blocks

- > Consist of a lookup table (LUT) and an optional flipflop (1-bit memory)
- > LUTs are typically 4..6 bits wide and can implement any combinatorial logic function
- > To enable/disable the flipflop, a multiplexer (MUX) is used that selects either the LUT output or the flipflop output



- > During **synthesis**, the design described by our VHDL code is divided up into CLBs and other FPGA primitives, and during **implementation**, actual FPGA resources are allocated, and the actual **bitstream** for programming is created
- > Both the LUT and the MUX selector input are configured via the bitstream sent to the FPGA when programming, together with the interconnects, I/O pin configurations, etc.

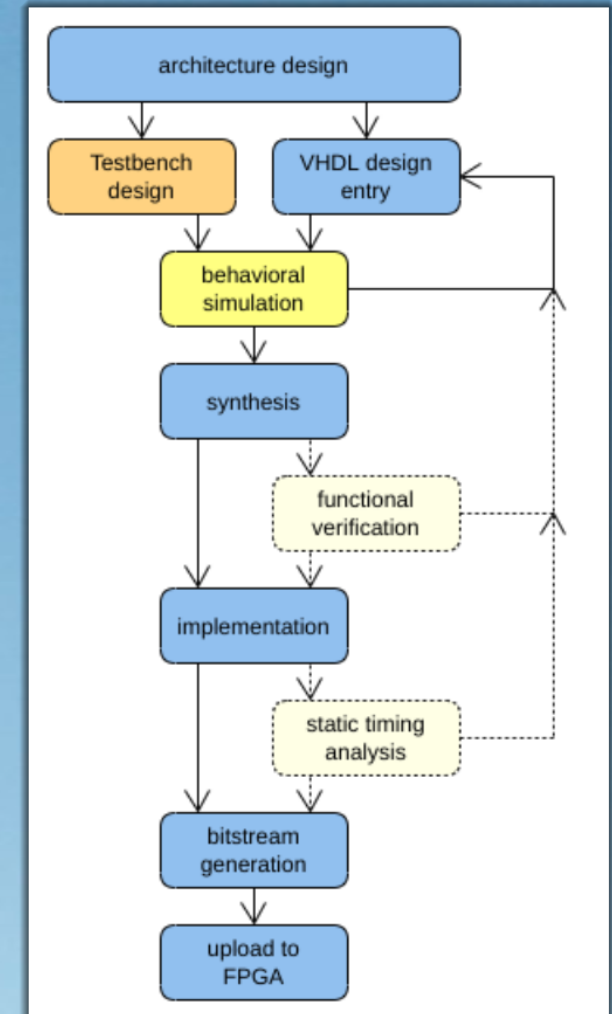
FPGA design flow

Actual, simplified design flow:

- > **VHDL design entry:** writing VHDL code
- > **Testbench design:** writing VHDL code for testing
- > **behavioral simulation:** simulating the testbench with ghdl tool

(until the whole CPU is verified, we will stop here)

- > **synthesis:** with Vivado (FPGA tool)
- > **implementation:** with Vivado
- > **bitstream generation:** with Vivado



Day 7: Breakthrough!!!!

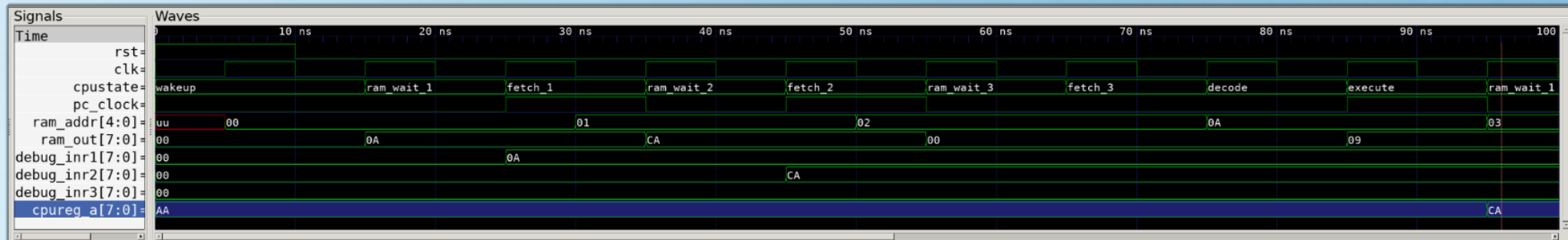
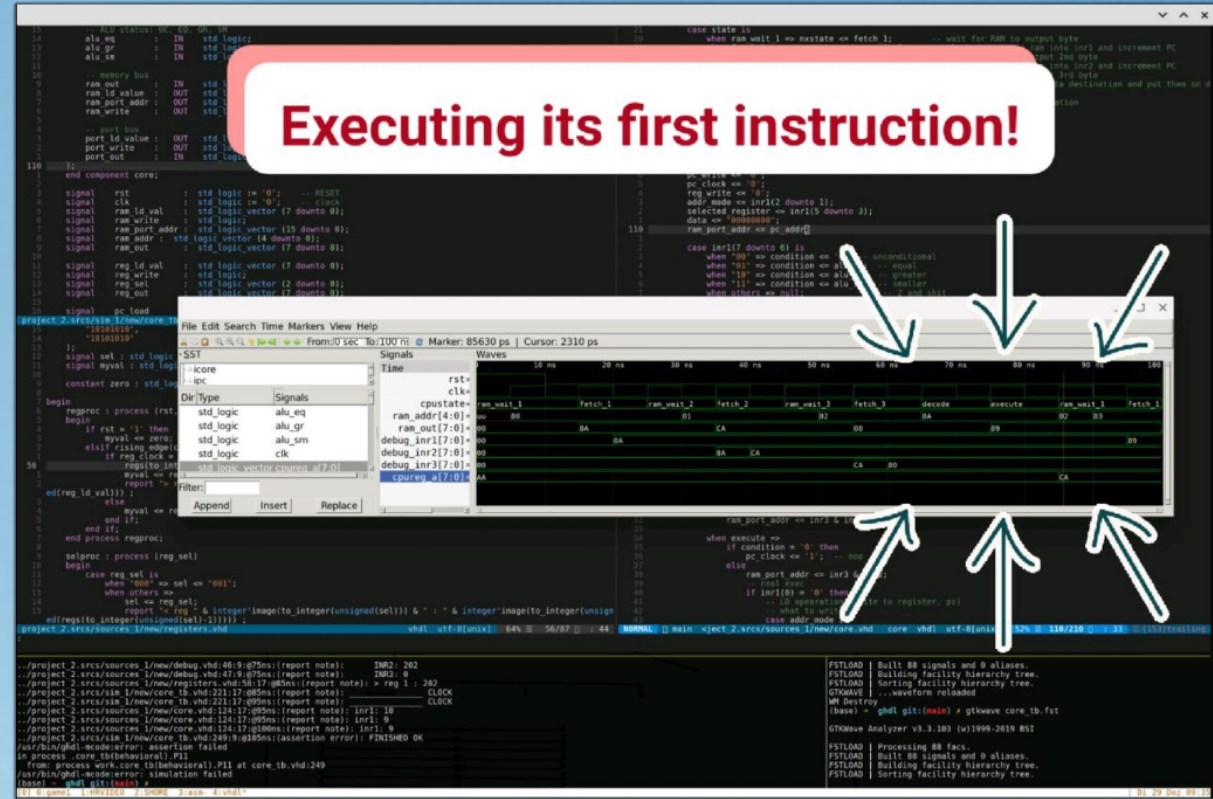
Eventually, on day 7, we can **run the simulation!**

First instruction ever executed:

ld a, # \$CA

- > Loads register A
- > with constant value \$CA
CA (hex) = 202 (dec)

Simulation waveform of relevant signals:



The macro assembler

- > Translates RRISC assembly to machine code
- > Supports
 - > constants (dec/hex)
 - > labels
 - > include files
 - > macros with parameters
- > Generates files:
 - > **.bin**: binary
 - > **.lst**: source listing annotated with generated code
 - > **.sym**: symbol table
 - > **.bit**: textual representation in binary notation for use in VHDL files
 - > **.coe**: coefficient file for use in BRAM cells

```
include simtest2.inc

org 0

macro testmacro $CA data
macro loop_forever

:data
db $ff
```

```
MACRODEF testmacro
lda # @1 ; load a with 1st passed in parameter
sta @2 ; store to addr defined by 2nd param
ldb @2
ENDMACRO

MACRODEF loop_forever
:@label
jmp @label ; jump to local label
ENDMACRO
```

```
(base) → renemann@penguin ..ub.com/renerocksai/rrisc/asm git:(main) ✕ python asm.py simtest.asm
```

```
Symbol Table:
loop_forever : 0009
data : 000c
```

```
Generating: simtest.lst
Generating: simtest.sym
Generating: simtest.coe
Generating: simtest.bin
Generating: simtest.bit
Program size: $000d bytes.
Done!
```

```
Set window scale
Set window scale
Set window scale
Set window scale
Set window scale
Done CanvasItem
Set window scale
Set window scale
Set window scale
Set window scale
Done CanvasItem
Done CanvasItem
```

```
org 0

lda # $CA ; load a with 1st passed in parameter
; > 0000: 0a ca 00

sta data ; store to addr defined by 2nd param
; > 0003: 09 0c 00

ldb data
; > 0006: 10 0c 00

:label_2
jmp label_2 ; jump to local label
; > 0009: 02 09 00

:data
db $ff
```


The ALU - arithmetic logical unit

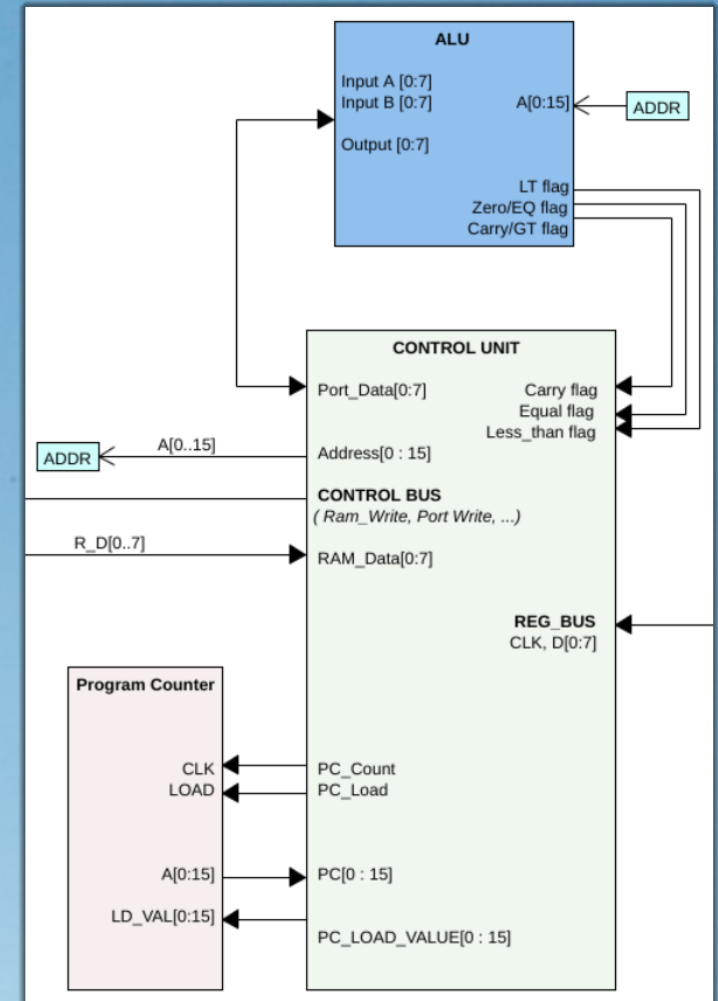
- > Necessary to perform calculations
- > **NOT** part of the CPU itself
- > Accessed via 4 external ports:
 - > Operand A
 - > Operand B
 - > Operation
 - > Result

Provides 3 flags to the CPU:

- > EQ/zero: last result was zero
or comparison result: equal
- > GT/carry:
 - > comparison result: 'greater than
 - > add/sub over/underflowed
 - > shift operation shifted a bit into it
- > LT: comparison result: less than

Operations:

- > add with carry
- > subtract with carry
- > shift left, MSB into carry
- > shift right, LSB into carry
- > bitwise rotate left
- > bitwise rotate right
- > bitwise or
- > bitwise and
- > bitwise nand
- > bitwise xor
- > bitwise compare
- > increment by 1
- > decrement by 1

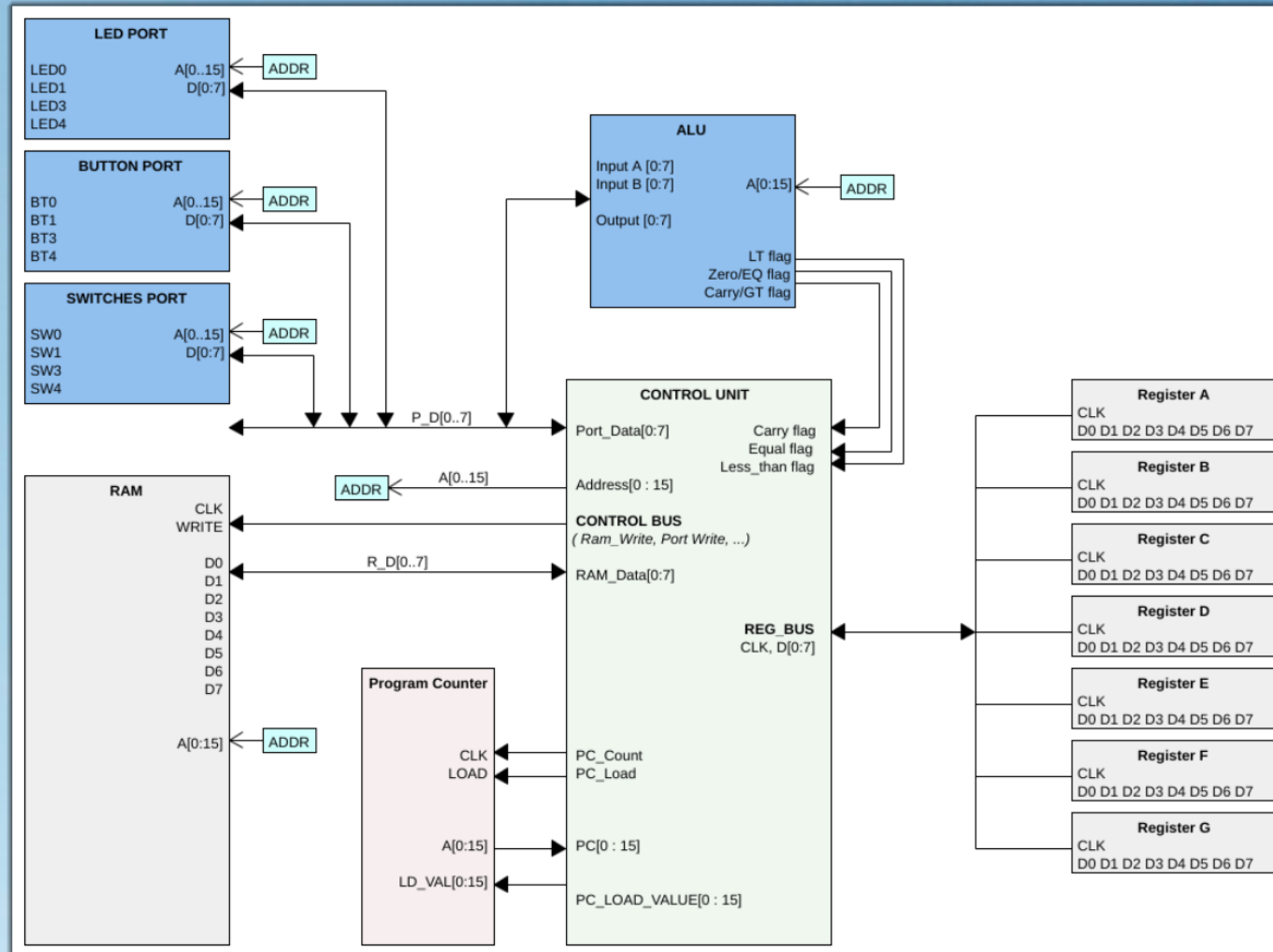


Preparing for the real-world test

- > For testing the CPU in the real world, we need to provide means for interaction with the board.
- > We add ports for LED lights and buttons.
- > The demo should show a running light when a button is pressed

The complete design consists of:

- > The RRISC CPU core
- > The RRISC ALU (arithmetic logical unit)
- > 1 kB of RAM for the demo program
- > 3 ports connected to the Arty board:
 - > 4 LEDs
 - > 4 switches
 - > 4 buttons



The running light demo program

```

;-----
; -- Arty S7 board test prog --
; --   for RRISC ALU on FPGA   --
;-----

```

```
include arty.inc
```

```
const my delay = $01ca : $0100 gives us 500ms, this gives us 395.595 ms
```

org @

```

out a, LED_PORT
out a, ALU_PORT_A
ldb # ALU_ROL
out b, ALU_PORT_INSTR
in a, ALU_PORT_RESULT
ENDMACRO

```

```
MACRODEF RIGHT : shifts led pattern to the right
```

```

out a, LED_PORT
out a, ALU_PORT_A
ldb # ALU_ROR
out b, ALU_PORT_INSTR
in a, ALU_PORT_RESULT
ENDMACRO

```

```
lda # $00
out a, LED_PORT ; initially, clear led pattern
```

```
in b, BIN PORT
out b, ALU PORT A
```

```

    ldb # $01
    out b, ALU_PORT_B
    ldb # ALU_AND
    out b, ALU_PORT_INSTR
    in b, ALU_PORT_RESULT
    jmp loop : F0

```

```
; running light
lda #$01                ; delay and shift
```

```
macro LEFT my_delay
macro LEFT
macro DELAY my_delay
macro LEFT
macro DELAY my_delay
macro LEFT
macro DELAY my_delay
```

```
macro RIGHT
macro RIGHT
macro RIGHT
macro DELAY my_delay
macro RIGHT
macro DELAY my_delay
macro RIGHT
jmp loop
```

```
include alu.inc
```

```
const BTN_PORT = $fff8;
const SW_PORT  = $fff9;
const LED_PORT = $fffa;
```

```
const arty_delay = $10da ; gives us 500ms
```

MACRODEF DELAY

```
ldf # < @1
lde # > @1
```

```
:@loop delay          ; loop over 16bit const
```

```

:loop_delay          ; loop over 20000 times
ldd # <arty_delay
ldc # >arty_delay
:@loop_low           ; loop over low byte of const

```

```

out d, ALU_PORT_A
ldg # ALU_DEC
out g, ALU_PORT_INSTR
in d, ALU_PORT_RESULT
jmp @break_low : EQ
jmp @loop_low
: @break_low ; loop over high byte of constant

```

```

out c, ALU_PORT_A
in c, ALU_PORT_RESULT
jmp @break_high : EQ
loop @loop_low
@break_high ; loop over low byte of param
out f, ALU_PORT_A
in f, ALU_PORT_RESULT
jmp @break_param_h1 : EQ
loop @loop_delay
@break_param_h1 ; loop over high byte of param

```

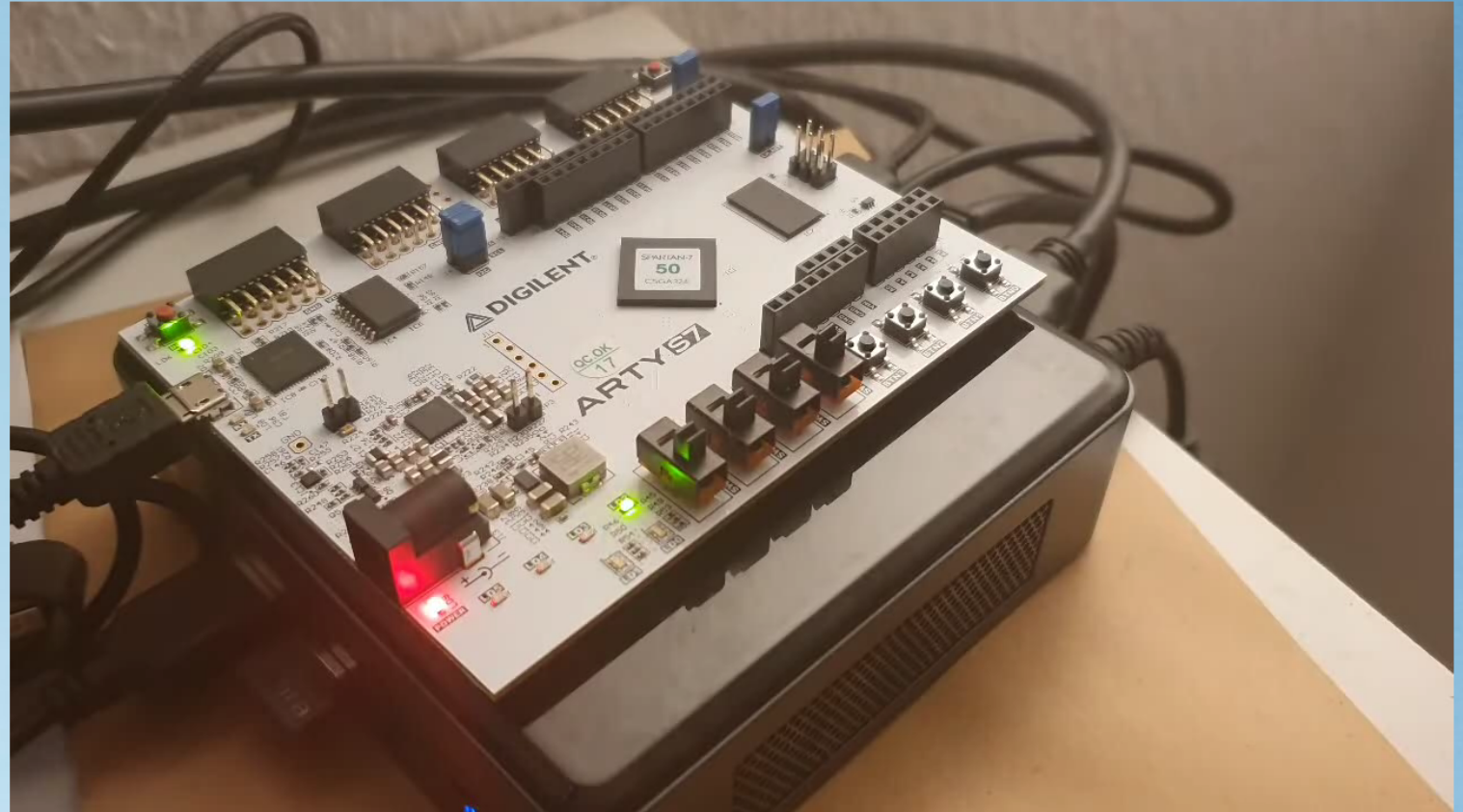
```
out e, ALU_PORT_A
in e, ALU_PORT_RESULT
jmp @end : EQ
jmp @loop_delay
:end
ENDMACRO
```



It **should** work...

Day 11: Breakthrough!!!! RRISC CPU in the real world

- > After successful
 - > synthesis
 - > implementation
- > The test program runs!
- > On the RRISC CPU!
- > On the FPGA board!



Play

The website

<https://renerocksai.github.io/rrisc>

- > static HTML website
- > authored in Markdown
- > HTML generated by the Jekyll static site generator
- > hosted on Github pages

./ VHDL implementation of the RRISC CPU

A small CPU with a radically reduced instruction set. Hand-crafted. Implemented in VHDL, for use in an FPGA.

[Download as .zip](#) [Download as .tar.gz](#) [View on GitHub](#)

rrisc

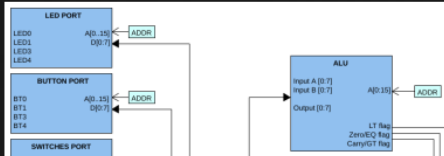
At Christmas 2020, I decided to hack on my [RRISC](#) CPU. Read all about it [here](#).

In a hurry? See it executing a single instruction [here](#), a complete program [here](#), and in physical form [here](#).

1. [Background and why I built the RRISC CPU](#)
2. [Radical RISC from the early nineties](#)
3. [What's unique about the RRISC CPU](#)
4. [It's executing its first instruction](#)
5. [RRISC Assembly - introduction](#)
6. [RRISC Assembler - writing programs](#)
7. [It runs the whole test program](#)
8. [We have an ALU!](#)
9. [Playing with the ALU](#)
10. [Open source, text-based VHDL design: vim, tmux, ghdl, gtkwave](#)
11. [The FPGA](#)
12. [NEW Becoming real: The CPU in action on an FPGA board](#)

This is a work in progress. More info on the minimalistic [RRISC](#) CPU will follow as soon as I get to it.

Here is a block diagram of the CPU with periphery of the demo implementation to get you started:



The diagram shows a central ALU block with inputs A [0:7] and B [0:7], and output [0:7]. It also has a 16-bit program counter (PC), a 7-bit register file (Reg), and a 7-bit instruction register (Inst). The ALU is connected to the PC, Reg, and Inst. The LED PORT (LEDD0-LEDD3) is connected to the ALU output. The BUTTON PORT (BT0-BT3) is connected to the ALU input B. The SWITCHES PORT (BT0-BT3) is connected to the ALU input A.

./ VHDL implementation of the RRISC CPU

A small CPU with a radically reduced instruction set. Hand-crafted. Implemented in VHDL, for use in an FPGA.

[Download as .zip](#) [Download as .tar.gz](#) [View on GitHub](#)

RRISC Assembly - introduction

The RRISC CPU can address the following:

- >> the 16bit program counter
- >> 7 registers (A..G)
- >> 64k of RAM
- >> 64k port addresses

Basic commands

The RRISC CPU understands the following basic commands, that are all based on the read / write principle:

- >> LD register
- >> ST register
- >> JMP address

LD stands for *load* and is used for loading values into a register *reg*.

ST stands for *store* and is used for writing the value of a register to the RAM (or external ports).

Each of the above commands causes a transaction between elements of two groups: The first group consists of the 7 registers and the second group is comprised of RAM, ports and instruction register 2 ('operand' of the instruction).

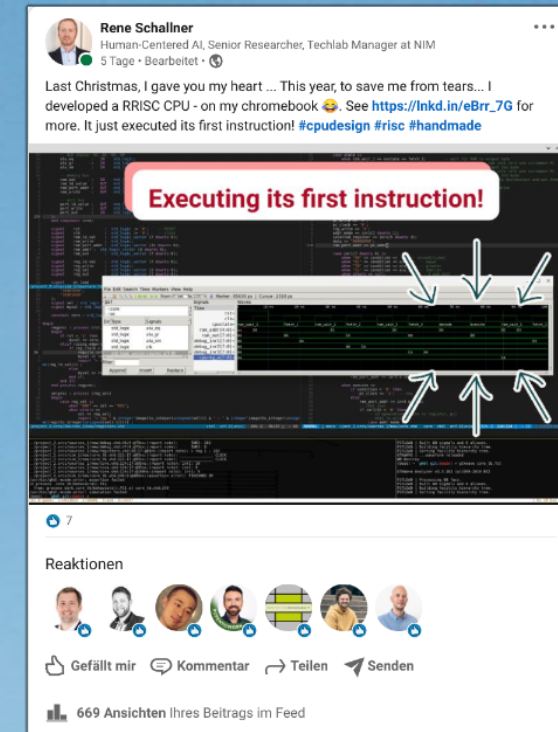
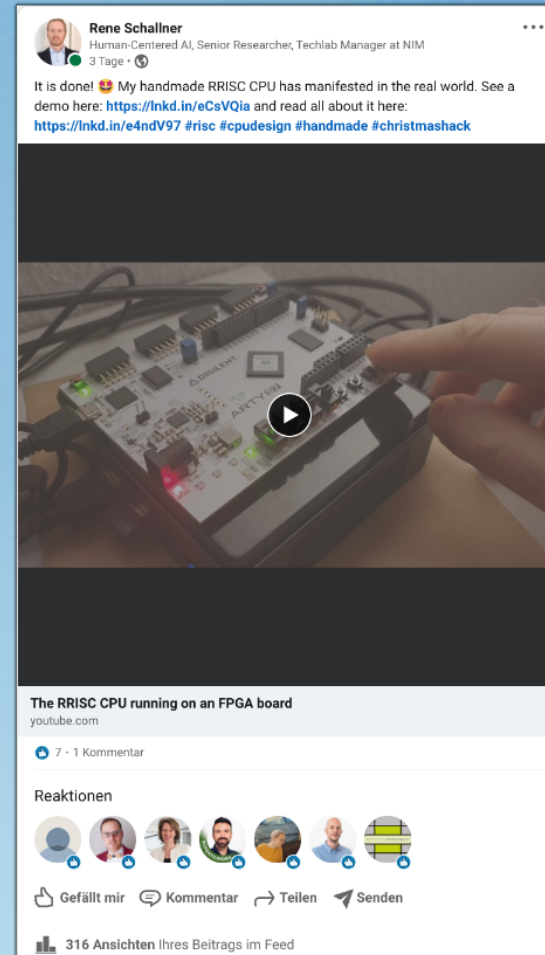
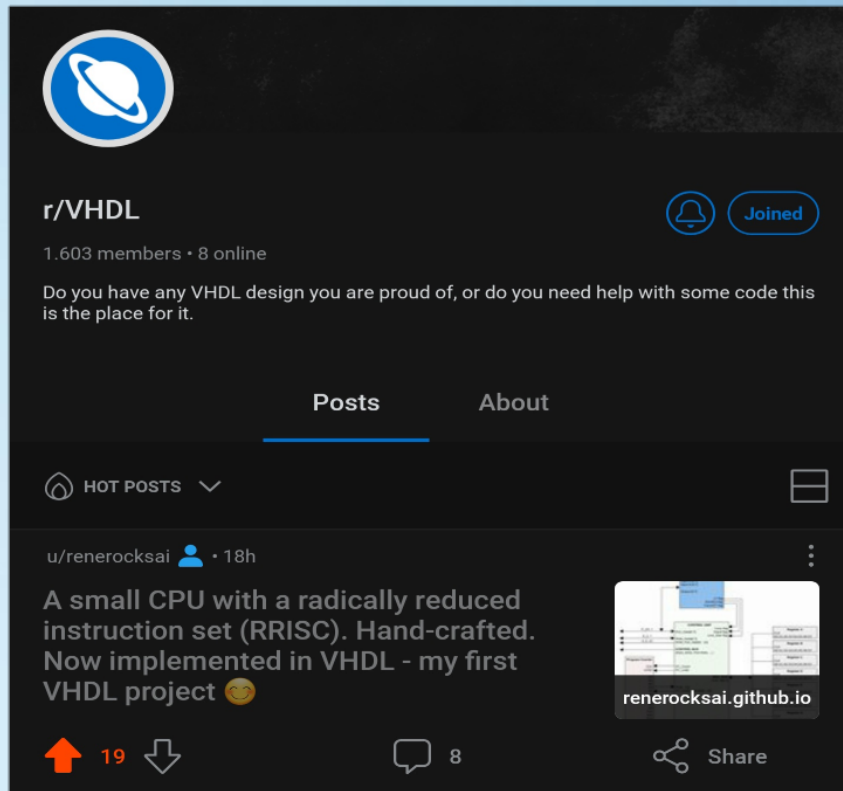
The following types of transactions are valid:

- >> register → RAM
- >> register → port
- >> register ← RAM
- >> register ← port
- >> register ← operand (instruction register 2)

JMP stands for *jump* and is used to continue program execution at a defined address. Strictly speaking, a jump is nothing else but an LD of the program

Reactions on social media

- > RRISC is actively being discussed on reddit
- > **Hottest topic** in r/VHDL on days 12 & 13 (2021-01-03 and 2021-01-04)



Takeaways

- > FPGAs are fascinating, extremely powerful chips
- > Designing electronics in **VHDL adds substantially to reusability and testability** of electronic circuit designs
- > **Electronic circuits** are parallel by nature
 - > **lend themselves to speeding up algorithms**
 - > Current trend of accelerating AI with FPGAs
- > Lightweight, free, commandline beats tens of gigabytes download, heavy GUI all the time
- > Creating toy CPUs is fun
 - > and there's always someone on reddit ready to comment
- > **Long, uninterrupted stretches of time** for concentration and **deep work** prove to be **extremely productive**



THANK YOU FOR YOUR ATTENTION

RRISC CPU

BUILDING A CPU OVER CHRISTMAS

RENE SCHALLNER