Faculty of Science
Department of Computer Science
Chair for Embedded Systems
Prof. Dr. O. Bringmann

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Digital Design and Synthesis of Embedded Systems

Summer Term 2025

Exercise 4 : Deadline 01.06.2025 - 23:59 o'clock

The solutions are submitted via ILIAS as a packed archive (ZIP/TAR.GZ). This ZIP contains the written answers to questions, the source code (not as a listing in the PDF), the simulation results as VCD files, and the Makefile. Each of your programming tasks **must** be executable with your Makefile or with a standard python interpreter. If your python solution uses non-standard packages you have to provide a requiretmens.txt file. If you do not follow this format, your handing may be graded with 0 points!

The solutions are submitted via ILIAS as a packed archive (ZIP/TAR.GZ). This ZIP contains the written answers to questions, the source code (not as a listing in the PDF), the simulation results as VCD files, and the Makefile. Each of your programming tasks **must** be executable with your Makefile or with a standard python interpreter. If your python solution uses non-standard packages you have to provide a requiretmens.txt file. If you do not follow this format, your handing may be graded with 0 points!

Exercise 1 Tiny5 - RISC-V CPU [20 Points]

In this task you will implement a functionally reduced single-cycle RISC-V CPU (Tiny5), as shown in Figure 2, and test it with smaller assembler programs. The reduced instruction set consists of:

1. ADD
2. MUL
3. ADDI
4. LW
5. SW
6. JAL
7. BNE

The schematic drawing shown here contains architectural elements that are implemented differently in a normal RISC-V CPU. For example, usually it is not sufficient to use only the `opcode` to control the multiplexers/flags. With the reduced instruction set, however, there is almost no overlap in the `opcodes`, which makes it possible to simplify the generation of control signals. To be able to execute programs, you also need an instruction memory and a data memory, as shown in Figure 1.

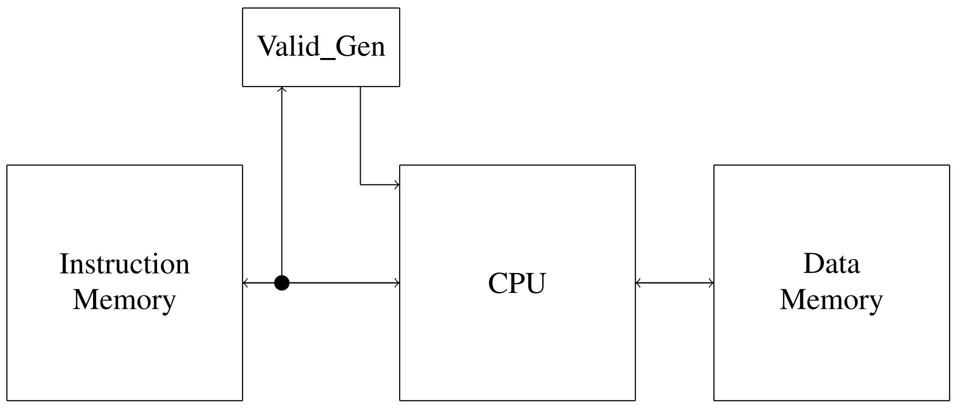

Figure 1: Block Diagram

Below you will find some implementation features of the Tiny5 CPU:

1. The memory models used require a clock cycle to load a date from the memory: In tact n the address is sampled and the data is loaded. In tact $n + 1$ the data is available at the output of the memory. As the *LW* command requires a clock delay, a `Valid_Gen` block is used, which delays the `mem_data_valid_in` signal. In this case, the clock of the Tiny5 CPU is connected to logical 0 (clock gating).
2. The `mem_read_write_out` signal (de)activates writing to the data memory (high-active).
3. The CPU can be stopped with the `halt_in` signal.
4. If an illegal instruction is found, the CPU sets the `error_out` signal.
5. The reset is low-active, i.e. a logical 0 triggers the reset of the processes.
6. The program counter is output via port `pc_out` and connected to the instruction memory. The instructions are sent back to the CPU via the `instruction_in` port. After the reset, the 0-th instruction is present at the instruction memory output. For this reason, the `pc_out` output is connected to the next PC address.

The provided model for the instruction memory allows you to load instructions from a text file **`instr.mem`** into the memory. You must specify the instructions as a binary string without formatting with a length of 32 bits, e.g:

00000000000000000000000010010011

Each instruction is in one line. If your program is smaller than the size of the instruction memory, you must fill the remaining lines with 32-bit long 0 lines. The size of the instruction memory can be found in the variable `INSTR_MEM_SIZE`. You may need to adjust the size. Similarly, you can find the size of the data memory in the variable `DATA_MEM_SIZE`.

We will provide you with the basic structure of the CPU. Familiarize yourself with the files provided!

We provide you with the following files:

1. `instructions.md` (TODO)
2. `rriscv_pkg.sv`
3. `instr.mem`
4. `alu.sv` (TODO)
5. `cpu.sv` (TODO)
6. `instr_mem.sv`
7. `instruction_decoder.sv`
8. `mem.sv`
9. `register_file.sv`
10. `tb_cpu.sv`

In the Markdown file you will find further information on the instructions and the architecture. In the package provided you will find important data structures which are used in the design. You can use the following online assembler for your test programs: `https://luplab.gitlab.io/rvcodecjs/`.

Hint:

1. For better orientation, the signals within the top module of the CPU are marked with a prefix: `id_` = Instruction Decoder, `rf_` = Register File, `alu_` = ALU.
2. The first register (*x0*) always has the value 0 according to the RISC-V specification. You cannot change this value.
3. You can use the testbench provided by us. You are free to modify it as you wish.

Tasks:

(a) Make yourself familiar with the instruction decoder and describe how it works in your own words.

(b) Make yourself familiar with the top module of the CPU (`cpu.sv`) and describe how the calculation of the next program counter works.

(c) **Instruction decoder:** (`cpu.sv`)

  (a) Instantiate the instruction decoder and connect it to the appropriate signals as shown in the block diagram.

(d) **Register file:** (`cpu.sv`)

  (a) Create the multiplexer, which is connected to the `data_in` port of the register file (see figure 2). The output of the multiplexer should be connected to the `rf_data` signal. Tip: You can control your multiplexer using the `opcodes` of the *LW* instruction.

  (b) Instantiate the register file and connect the inputs and outputs to the appropriate signals.

(e) **ALU:** (`alu.sv`)

  (a) Implement the zero flag of the ALU.

  (b) Instantiate the ALU and connect the inputs and outputs to the appropriate signals.

(f) **Test programs:** (`instr.mem` oder `tb_alu.sv`)

  (a) Test each instruction.

  (b) Write a small program and describe its function. Your program must accept at least one immediate value. *Hint:* You can load immediate values into a register of your choice with the following command, e.g: `addi x1, x0, 5`.

Note: We provide you with a program that can calculate the Fibonacci number of a position n. The result should be saved in register 16 (*x15*) of the register file. The program also tests whether the access to the memory works correctly by writing the result to memory address 0 and then loading it into register 17 (*x16*). With the first instruction in the file `instr.mem` you can select your own n. It is saved in register 7 (*x6*).

Exercise 2 Bonus task [9 Points]

Extend the instruction set with the following instructions (each instruction gives 4.5 points) and write a test case. Instructions without test case aren't graded.

1. BEQ: Branch Equal

2. XOR: XOR Operation

You can find the types of instruction and immediate in the linked architecture specification.

You can actively use the forum if you have questions about the assignment. If you have questions about your solutions, you can write an e-mail to the tutor or the supervisor of the lecture at any time.

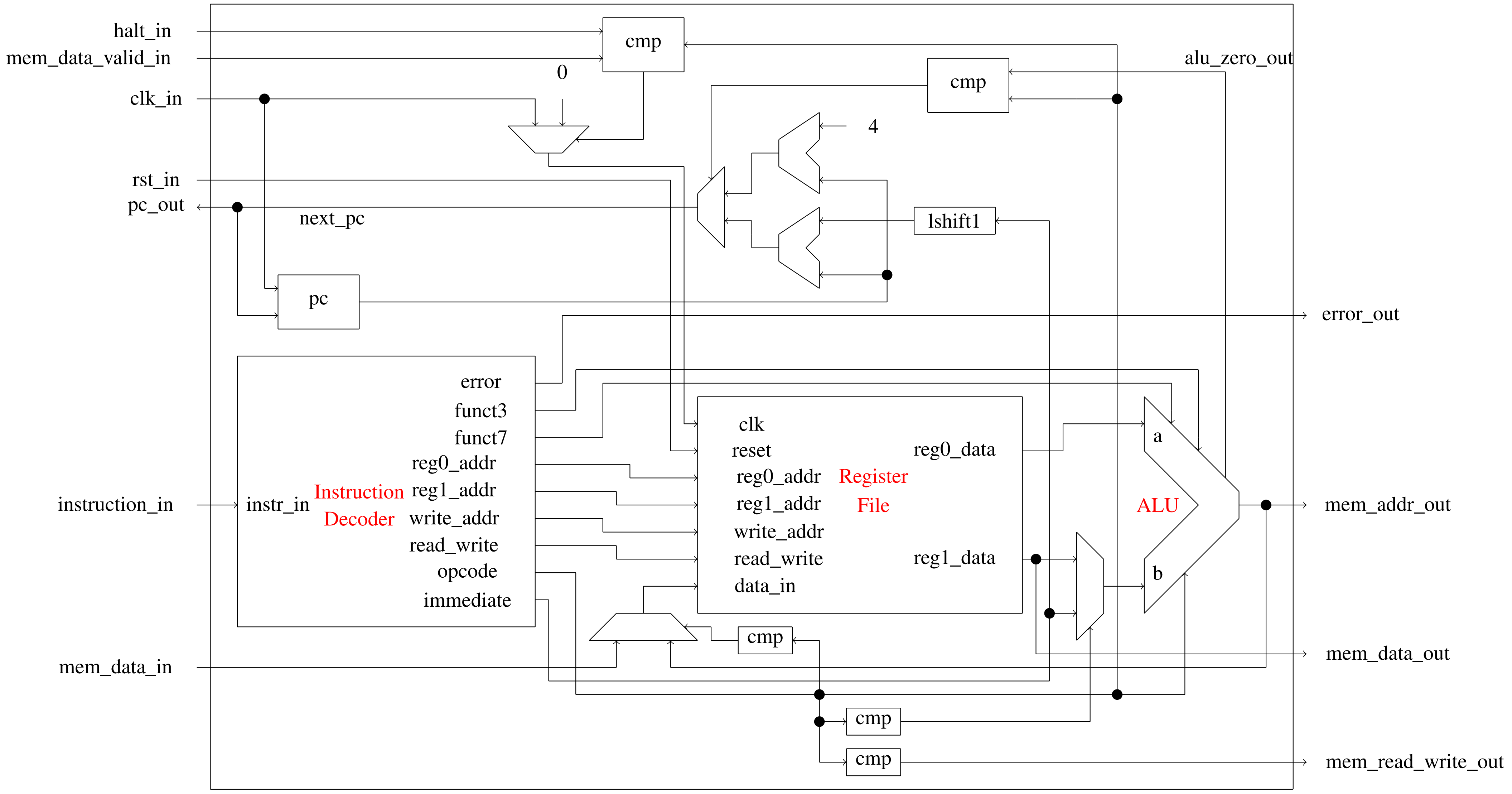

Figure 2: Tiny5 CPU