JOHANNES KEPLER
UNIVERSITY LINZ

Name
Maximilian Zaglmayr

Entwurf Integrierter Schaltungen

4-Bit-CPU

Documentation

Contents

1 Introduction

The following documentation is about a small custom 4 bit CPU created for the TinyTapout workshop. The whole project can be found here.The source code is located in the subfolder `src`, the test-benches in `tb` and the simulation result vdc files in `sim`. The CPU utilizes a 4 bit accumulator register, a 4 bit ALU and a very small 16x4 bit memory block built out of registers, to store program code and data. It features a custom instruction set of 16 instructions and includes a programmer to enable the user to run different simple programs via UART. Additionally, the CPU implements input and output registers that are accessible through external pins, as explained later in the pinout section.
The CPU is designed with a very minimalistic approach (only 4 bits and a very small memory) in order to match the thigh space requirements of the TinyTapeout. A slightly different 5 Bit version was also tested, but did not fit on the space available.
The main goal of this project was, to learn more about CPU's and their architecture on a hands-on example. This CPU does not adhere to one specific type or standard nor does the instruction set.

2 Architecture Overview

Figure 1 illustrates a simplified architectural overview of all the components included in the project and their interconnections. The most important building blocks are

- **Control Unit:** Serves a brain of the CPU, controlling the data flow using control signals, decoding instructions and tracking the current program counter and program state.
- **Memory:** 16x4 bit memory for storing instructions and data.
- **Registers** Different registers for storing the current instruction (`Instr`), operand `OPND` and memory data (`MDR`). Additionally one working register (`A`) is also included.
- **ALU:** The `ALU` performs the 4 bit arithmetic calculations a+b and a-b as well as incrementing and decrementing by 1. Also included is a bitwise `AND`, `OR` and `XOR`.
- **Programmer (including UART RX):** The programmer can receive data via a UART interface (only RX) and can be used to program instructions codes into the memory.

Note that Figure 1 does not included clock and reset signals as well as the synchronization of external input signals.

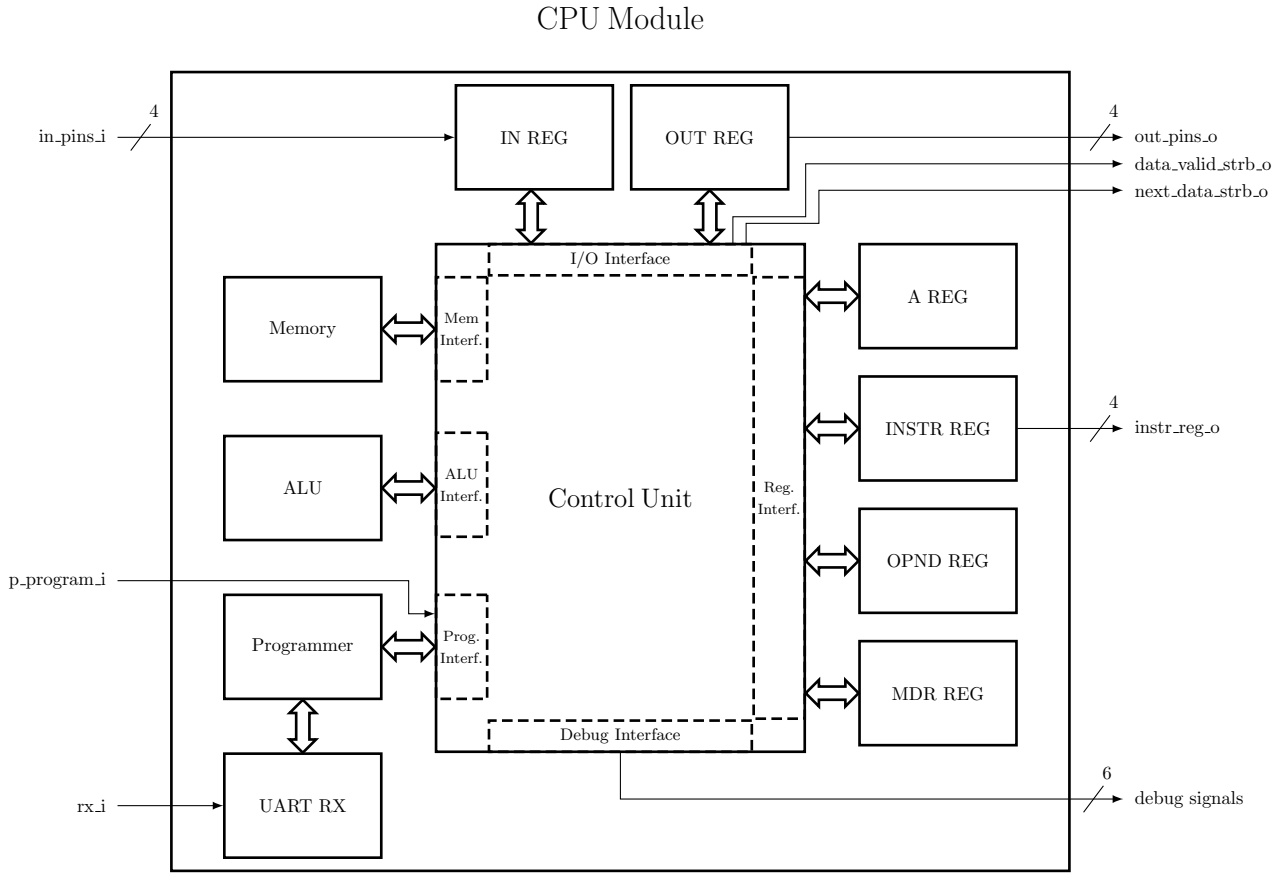

Figure 1: Simple architectural overview of the CPU

2.1 Operation Principle

The `Control Unit` starts reading the first data nibble from the memory (at address 0) and writes it into the instruction register. After decoding the instruction, the CPU starts executing the instruction in multiple steps depending on the active instruction. During those steps, the `OPND` and `MDR` registers are used to cache the instruction operand or memory data if needed. The `A` register serves as the main working register (accumulator), the input data and the result of an operation are always stored in this register. For example, the instruction ADD, adds the data in the `A` register and the data in the `MDR` register together using the `ALU` and stores the result in `A`. Some instructions relay on logical or

arithmetical operations performed by the `ALU`. After a instruction is done, the program counter is increased by one and the next instruction is read, starting the cycle from the beginning. The next few chapters will explain this flow in more detail. The `Control Unit` has one extra state (active when the `p_program_i` is pulled high externally) that enables the `Programmer` to write directly to the memory. In this mode, incoming data from the UART RX is streamed directly into the memory to enable loading different programs. To interact with the CPU, the project includes an `IN` and an `OUT` register with the according IN and OUT instructions. This enables external peripherals to communicate with the CPU.

2.2 Instruction Set Architecture

One instruction consists of a 4 bit op-code and an optional 4 bit operand, making them 4 or 8 bits long. The op-code and operand are stored directly in succession, with the opcode preceding the operand. All possible Instructions are listed in Table 1.
`A` is the `A` register, `MEM[]` represents the memory, `IN` and `OUT` are the in and out registers, `PC` is the program counter and `c` or `z` refer to the carry or zero flag of the last `ALU` result.
The op-codes are selected in a specific way, which makes decoding the instructions a simple as possible. For more detail see subsubsection 5.2.1 and 8.2.1.

Abbreviation	OP-Code	Operand	Operation	Bits
NOP	0000	-	No Operation	4
XOR	0001	addr	$A \leftarrow A \oplus MEM[addr]$	8
AND	0010	addr	$A \leftarrow A \wedge MEM[addr]$	8
OR	0011	addr	$A \leftarrow A \vee MEM[addr]$	8
ADD	0100	addr	$A \leftarrow A + MEM[addr]$	8
INC	0101	-	$A \leftarrow A + 1$	4
DEC	0110	-	$A \leftarrow A - 1$	4
SUB	0111	addr	$A \leftarrow A - MEM[addr]$	8
JMP	1000	addr	$PC \leftarrow addr$	8
JZ	1001	addr	if $z == 1$: $PC \leftarrow addr$	8
JC	1010	addr	if $c == 1$: $PC \leftarrow addr$	8
LD	1011	addr	$A \leftarrow MEM[addr]$	8
ST	1100	addr	$MEM[addr] \leftarrow A$	8
IN	1101	-	$A \leftarrow IN$	4
OUT	1110	-	$OUT \leftarrow A$	4
LDI	1111	imm	$A \leftarrow imm$	8

Table 1: Instruction Set Architecture

2.3 Pin Out

The `cpu_board` serves as the top level module for the TinyTapeout project. It sets all the parameters which define various bit-widths and constants throughout the project and includes the `cpu` module. It is also responsible to connect the in and output signals to the corresponding pins on the TineyTapeout board as listed in Table 2.

I/O	PIN	Signal	Name	Purpose
IN Pins				
IN	ui[0]	-	-	-
	ui[1]	–	-	-
	ui[2]	p_program_i	EN_PROG	enable programming mode
	ui[3]	rx_i	RX	UART receive input
	ui[4]	in_pins_i[0]	IN0	input signals
	ui[5]	in_pins_i[1]	IN1	
	ui[6]	in_pins_i[2]	IN2	
	ui[7]	in_pins_i[3]	IN3	
OUT Pins				
OUT	uo[0]	next_data_strb_o	next_data_strb	next data strobe for the input signals
	uo[1]	data_valid_strb_o	data_valid_strb	data valid strobe for the output signals
	uo[2]	program_o	st_program	debug signal for programming mode
	uo[3]	fetch_instr_o	st_fetch_i	debug signal for the fetch instruction state
	uo[4]	decode_o	st_decode	debug signal for the decode state
	uo[5]	fetcho_op_o	st_fetch_op	debug signal for the fetch operand state
	uo[6]	fetch_mdr_o	st_fetch_mdr	debug signal for the fetch mdr state
	uo[7]	execute_o	input st_ececute	debug signal for the execute state
Bidirectional Pins				
OUT	uio[0]	out_pins_o[0]	OUT0	output signals
	uio[1]	out_pins_o[1]	OUT1	
	uio[2]	out_pins_o[2]	OUT2	
	uio[3]	out_pins_o[3]	OUT3	
	uio[4]	instr_reg_o[0]	INSTR_REG0	instruction register values
	uio[5]	instr_reg_o[1]	INSTR_REG1	
	uio[6]	instr_reg_o[2]	INSTR_REG2	
	uio[7]	instr_reg_o[3]	INSTR_REG3	

Table 2: I/O of the top level `cpu_board` module

3 Registers

For all of the shown registers in Figure 1 the same Verilog module called `my_register` is reused (for the code, see here). It consists of a simple register process with a write enable signal included.

3.1 Operation Principle

All the input and output signals of the register module are listed in Table 3. The register supports asynchronous reading, which means that the data of the register can be accessed at any given time via the `out_o` signal. To set the value of the register, drive the `write_en_i` signal high and set the `in_i` to the desired input data. On the next positive edge of the clock, the register will update it's content. The default value applied during reset is 0.

I/O	Signal	Bit-width	Purpose
IN	clk_i	1	clock signal
	reset_i	1	asynchronous reset
	in_i	4 (variable)	input data
	write_en_i	1	write enable signal
OUT	out_o	4 (variable)	Current content

Table 3: I/O of the `my_register` module

3.2 Simulation

The test-bench for this module is called `my_register_tb` and located here. It simulates storing a few different values by setting the `in_i` and `write_en_i` accordingly. The simulation result (Figure 2) clearly shows, that the register works as expected. The input value is only considered at the next positive edge and when `write_en_i` is high. Afterwards the value is stored in the register (`reg_vals`) until it is overwritten again.

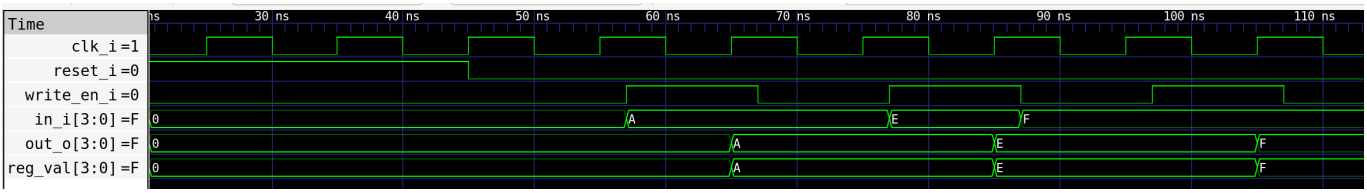

Figure 2: Simulation result of `my_register` module.

4 Register Memory

The next building block is the memory consisting of an array of registers. It acts as RAM for the CPU to store program code and data. The code for the `reg_memroy` module is located here.

4.1 Design Choice

The TinyTapeout website lists 3 options to implement memory:

- Registers
- DFF RAM (efficient array of registers)
- external SPI RAM

DFF RAM is a more space efficient arrangement of registers for memory use, but it only comes in very inflexible configurations. For example, the Ram32 macro configuration is a 32x32 bits DFF RAM register memory, which only fits on 2x3 tiles. Because of it's size and inflexibility, this was not an option.
Using the microcontroller on the TinyTapout board and simulating an external SPI RAM with it, was also ruled out as this is not really testable and rather complicated to implement.
Registers are the most space inefficient but the easiest to implement and the most flexible, therefore this option was chosen at the cost of a bigger memory.

4.2 Operation Principle

As explained earlier, the memory is only 16x4 bits small due to space limitation. 16x4 perfectly fits the 4 bit architecture of the project, utilizing a 4 bit addressing space. Each 4 bit nibble can be addressed separately. All the input and output signals of this block are listed in Table 4.

I/O	Signal	Bit-width	Purpose
IN	`clk_i`	1	Clock signal
	`reset_i`	1	Asynchronous reset
	`write_en_i`	1	Write enable signal
	`read_en_i`	1	Read enable signal
	`data_i`	4 (variable)	Input data
	`addr_i`	4 (variable)	address
OUT	`data_o`	4 (variable)	read data output

Table 4: I/O of the `reg_memory` module

Writing
To write to a specific address, set the `addr_i` and `data_i` accordingly and drive `write_en_i` to high. The data will be written to the memory at the next positive clock edge.

Reading
In order to read from a specific address, again set `addr_i` to the desired address and drive the `read_en_i` high. The data will be present on the `data_o` bus immediately, as this memory supports asynchronous reading.

Reset
A reset writes a simple standard program into the memory, see Codesnippet 1. This program counts up indefinitely and outputs every result to the output pins. It can later be overwritten using the programmer.

4.3 Simulation

To test the design, the test-bench (`reg_memory_tb`) writes the numbers from 0 to 15 in ascending order into the memory locations with the corresponding addresses. Afterwards the whole memory is read in the same order as it was written before. Figure 3 shows that the memory is working as expected. The correct values are stored in the `reg_vals` one at a time, matching the addresses. Afterwards the `data_out` shows, that the read functionality also works as expected.

```
if (reset) begin
    reg_vals[0]  <= OUT_INSTR;
    reg_vals[1]  <= INC_INSTR;
    reg_vals[2]  <= JMP_INSTR;
    reg_vals[3]  <= 4'b0000;
    reg_vals[4]  <= NOP_INSTR;
    reg_vals[5]  <= NOP_INSTR;
    reg_vals[6]  <= NOP_INSTR;
    reg_vals[7]  <= NOP_INSTR;
    reg_vals[8]  <= NOP_INSTR;

    reg_vals[9]  <= NOP_INSTR;
    reg_vals[10] <= NOP_INSTR;
    reg_vals[11] <= NOP_INSTR;
    reg_vals[12] <= NOP_INSTR;
    reg_vals[13] <= NOP_INSTR;
    reg_vals[14] <= NOP_INSTR;
    reg_vals[15] <= NOP_INSTR;
```

Codesnippet 1: Default memory configuration on reset

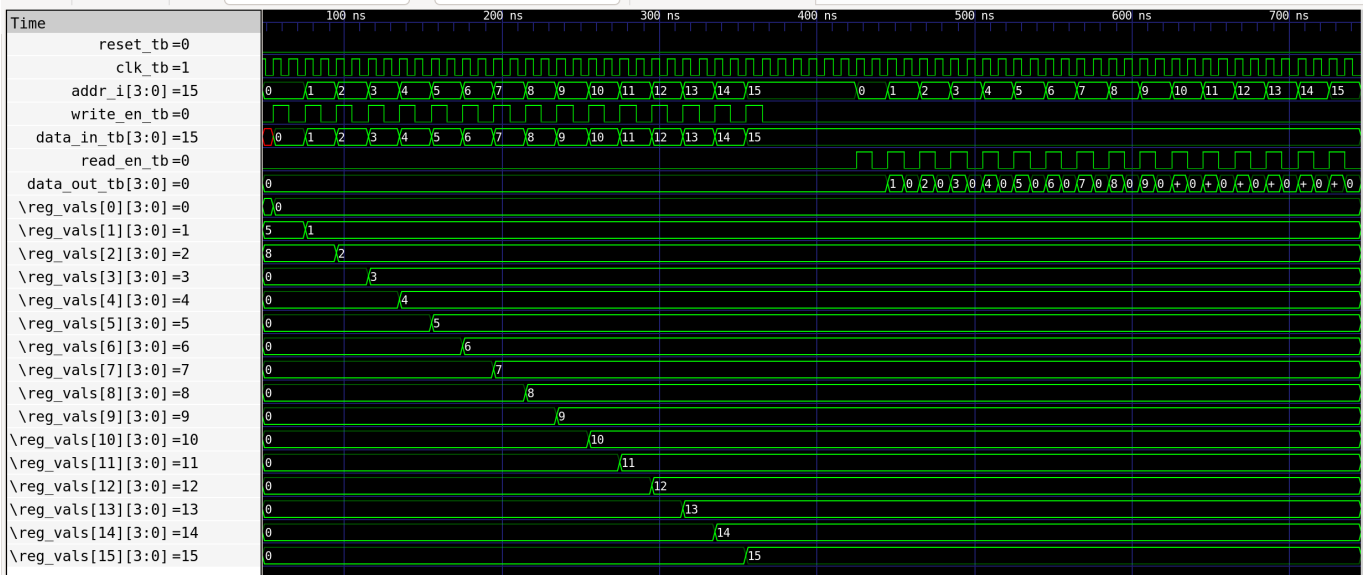

Figure 3: Simulation result of `reg_memory` module.

5 ALU

The ALU is the computational heart of the CPU, responsible for performing arithmetic and logic operations. It consists only of combinatorial logic and does not use any clock/reset signal or registers.

5.1 Internal Structure

The arithmetic part of the ALU is a 4 bit Carry-Ripple-Adder consisting of four Full-Adder, each based on two Half-Adder.

5.1.1 Half-Adder

On Half-Adder is a very simple building block that can add together two single bit numbers. It outputs the sum and a carry bit. This behaviour can be implemented with two logic statements:

$$\text{sum} = a \oplus b \qquad \text{carry} = a \wedge b$$

Figure 4 shows the schematic of the Half-Adder module with all the in and outputs.

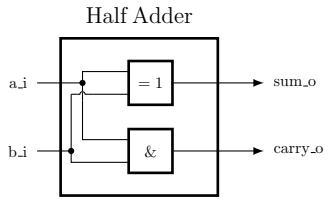

Figure 4: Schematic of a Half-Adder

5.1.2 Full-Adder

The Full-Adder combines two Half-Adders and introduces an input carry bit that is also taken into consideration. The truth-table of a Full-Adder is shown in Table 5. To achieve this behaviour, two Half-Adder have to be combined as shown in Figure 5.

carry_in	a	b	sum	carry_out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 5: Truth-table of a Full-Adder

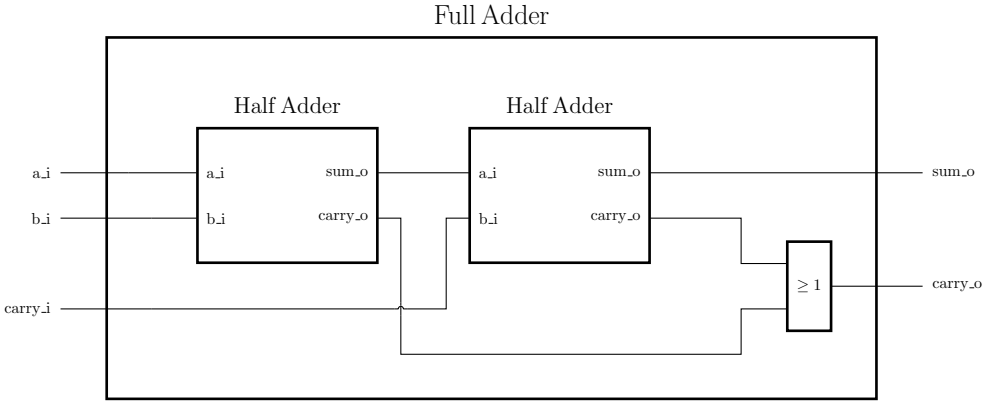

Figure 5: Schematic of a Full-Adder

5.1.3 Carry-Ripple-Adder

The Carry-Ripple-Adder now combines multiple Full-Adder by concatenating them to extend the bit-width. As the ALU should be able to add 4 bit numbers, four Full-Adders are required. The first Full-Adder uses the LSB bits of the two input signals (`a_0` and `b_0`) and the `carry_i`. The output sum of the first stage corresponds to the LSB of the output vector (`sum_0`). The output carry is passed on to the next Full-Adder stage as input next to the input bits (`a_1` and `b_1`). This pattern continuos to the last Full-Adder, as shown in Figure 6. The `carry_i` acts as a +1 to the total sum if set to 1:

$$\text{sum} = \text{a} + \text{b} + \text{carry\_in}$$

This is particularly useful for the two's compliment during subtraction. As a two's compliment requires the addition of 1, which can be implemented by setting the carry_i = 1.

5.2 Operation Principle

Table 7 lists all the input and output signals of the `alu` module. The `oc_i` (ALU op-code) signal determines the performed operation.

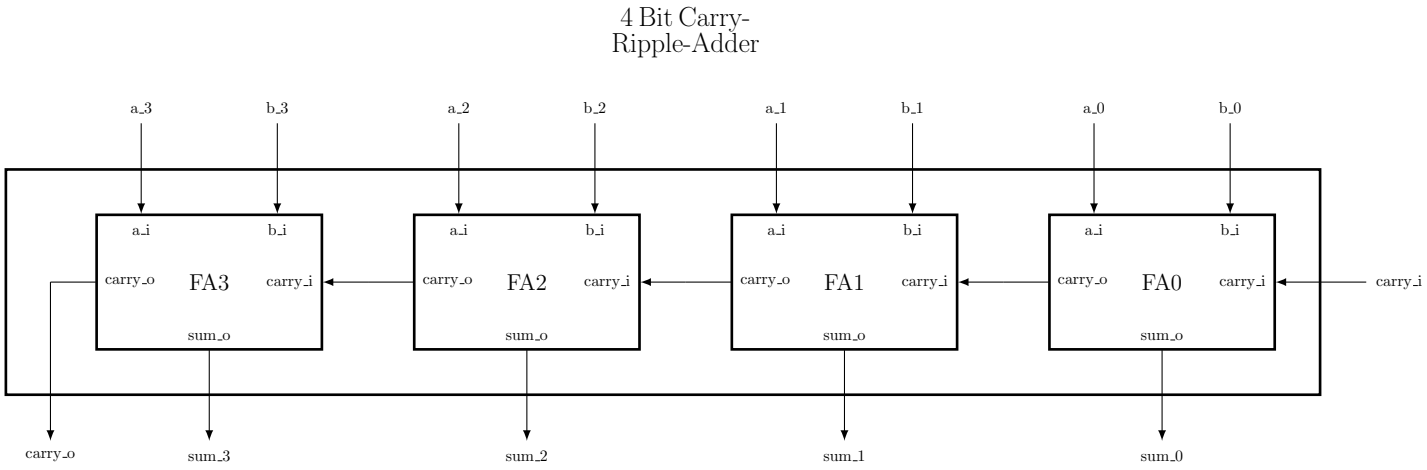

Figure 6: Schematic of a Carry-Ripple-Adder

5.2.1 OP-code decoding

Due to clever instruction design, the last 3 bits of the instructions operation code can be used as ALU operation code, which makes decoding for ALU-instructions easy. The ALU internally differentiates between logic and arithmetic operations based on the first bit of the ALU op-code.

- `oc_i[2] = 0`: logic operation
- `oc_i[2] = 1`: arithmetic operation

The exact logic operation is then determined using a simple 2 bit multiplexer (`oc_i[1]` and `oc_i[0]`. In the case of an arithmetic instruction, the ALU does not differentiate between `ADD` or `INC` and `SUB` or `DEC`, only the positive or negative sign is important. The middle bit of the op-code determines the sign:

- `op_i[1] = 0`: positive sign
- `op_i[1] = 1`: negative sing

I/O	Signal	Bit-width	Purpose
IN	`a_i`	4 (variable)	input number a
	`b_i`	4 (variable)	input number b
	`oc_i`	3 (variable)	code to determine desired operation
OUT	`result_o`	4 (variable)	result of the operation
	`carry_o`	1	carry bit of arithmetical operation

Table 6: I/O of the `alu` module

Operation	ALU OP-code
Logic Operations	
XOR	001
AND	010
OR	011
Arithmetic Operations	
ADD	100
INC	101
DEC	110
SUB	111

Table 7: I/O of the `alu` module

5.2.2 Subtraction

As already explained above, the `carry_i` bit can be used to create the two's compliment and therefore implement a subtraction. Whenever the ALU detects an arithmetic operation with a negative sign, the `b_i` input is inverted bitwise. Additionally the `carry_i` bit of the Carry-Ripple-Adder is set to 1, compensating for the +1 in the two's compliment.

5.3 Simulation

Every single component was tested and the GitHub repository includes a test-bench for each. In favour of a better readability, only the test-bench of the Carry-Ripple-Adder and the whole ALU are explained in more detail here.

5.3.1 Carry-Ripple-Adder

The test-bench (carry_ripple_adder_tb) first tests four scenarios:

- Scenario 1: a = 0001, b = 0010, carry_in = 0 ⇒ expected result: sum = 0011, carry_out = 0
  Tests basic addition.
- Scenario 2: a = 1111, b = 0001, carry_in = 0 ⇒ expected result: sum = 0000, carry_out = 1
  Tests if the overflow and `carry_out` bit works.
- Scenario 3: a = 0101, b = 1010, carry_in = 1 ⇒ expected result: sum = 0000, carry_out = 1
  Tests if the `carry_in` causes the `carry_out` to be 1.
- Scenario 4: a = 1111, b = 1111, carry_in = 1 ⇒ expected result: sum = 1111, carry_out = 1
  Tests maximal overflow.

Additionally 5 random number tests are performed. The results are shown in Figure 7 and Figure 8. The first four test-results do match the expected values. Also the five random test results do fit the a + b + carry_in expectation.

```
  a   |   b   | carry_in|  sum   | carry_out
-------------------------------------------------
0001 | 0010 |    0    |  0011  |    0
1111 | 0001 |    0    |  0000  |    1
0101 | 1010 |    1    |  0000  |    1
1111 | 1111 |    1    |  1111  |    1
-------------------------------------------------
0100 | 0001 |    1    |  0110  |    0
0011 | 1101 |    1    |  0001  |    1
0101 | 0010 |    1    |  1000  |    0
1101 | 0110 |    1    |  0100  |    1
1101 | 1100 |    1    |  1010  |    1
```

Figure 7: Result of the Carry-Ripple-Adder simulation.

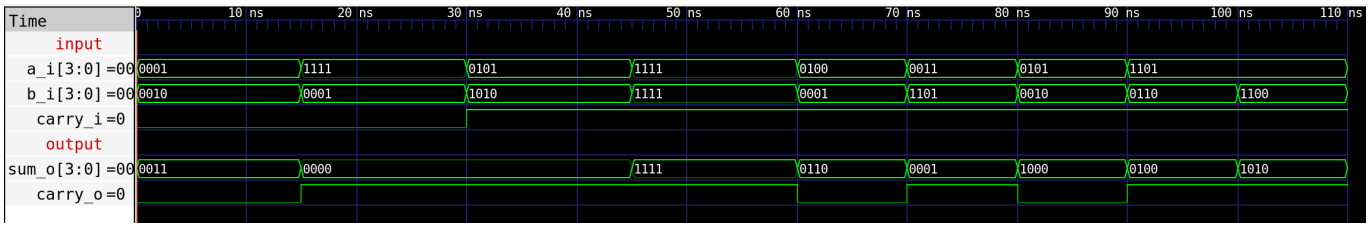

Figure 8: GTK-Viewer of the Carry-Ripple-Adder simulation.

5.3.2 ALU

Within the alu_tb every ALU operation is tested once.

- ADD: a = 0010, b = 1111, op_code = 100 ⇒ expected result: result = 0001, carry_out = 1
- INC: a = 0001, b = 0010, op_code = 101 ⇒ expected result: result = 0011, carry_out = 0

- SUB: a = 0001, b = 0010, op_code = 111 ⇒ expected result: result = 1111, carry_out = 0
- DEC: a = 0001, b = 0010, op_code = 110 ⇒ expected result: result = 1111, carry_out = 0
- XOR: a = 1001, b = 1010, op_code = 001 ⇒ expected result: result = 0011
- AND: a = 1001, b = 1010, op_code = 010 ⇒ expected result: result = 1000
- OR: a = 1001, b = 1010, op_code = 011 ⇒ expected result: result = 1011

Note that the ALU does not differentiate between `ADD/INC` and `SUB/DEC`, which means, that in both cases it just adds or subtracts `a` to/from `b`. The control unit is responsible to set the `b` input accordingly, when a `INC` or `DEC` op-code is applied.
The `carry_out` signal is ignored in the case of a logic operation, as it does not have any meaning in those cases. Due to the structure of the ALU, it is never the less possible, that the `carry_out` is high during a logic operation, but it has no further use.

Figure 9 and Figure 10 show the results of the simulation. It is easy to see, that all the resulting values match with the expected ones.

a	b	oc	result	carry
0010	1111	100	0001	1
0001	0010	101	0011	0
0001	0010	111	1111	0
0001	0010	110	1111	0
1001	1010	001	0011	1
1001	1010	010	1000	0
1001	1010	011	1011	0

Figure 9: Results of the ALU simulation.

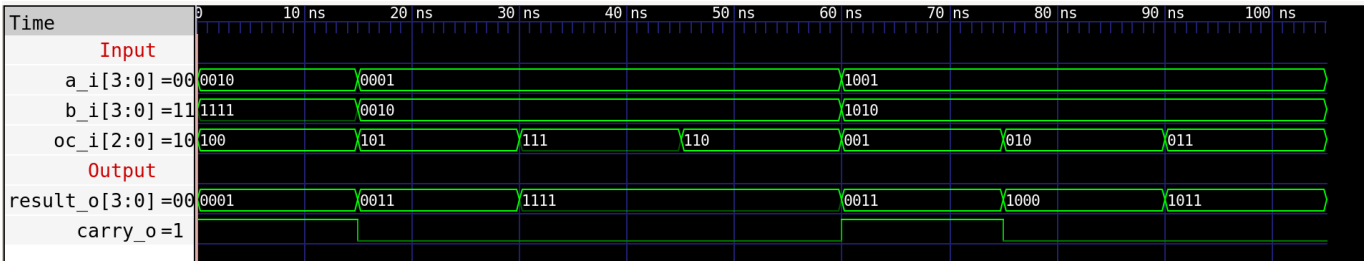

Figure 10: Wave-view of the ALU simulation

6 UART Receiver

The uart_rx module is a simple UART receiver block.

6.1 Internal Structure

The UART receiver can be conceptually divided into four main logical components, as shown in Figure 11. Although the `uart_rx` Verilog module is implemented as a single module and does not instantiate four separate submodules, its internal functionality can be logically partitioned into these four components. The inputs and outputs are listed in Table 8.

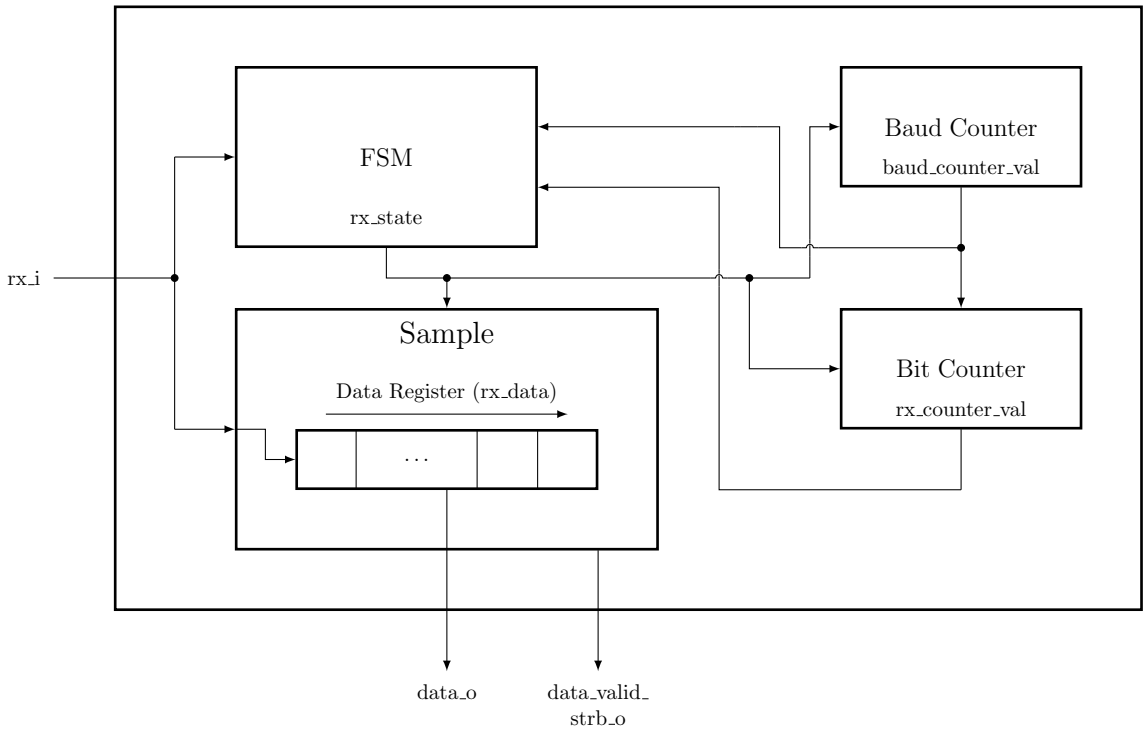

Figure 11: Schematic of the UART receiver module

I/O	Signal	Bit-width	Purpose
IN	`clk_i`	1	clock signal
	`reset_i`	1	asynchronous reset
	`rx_i`	1	receiving input line of UART protocol
OUT	`data_o`	8 (variable)	data of one UART packet
	`data_valid_strb_o`	1	valid strobe for data output

Table 8: I/O of the `uart_rx` module

6.1.1 FSM

The internal FSM features four states (Figure 12) corresponding to the possible states during a UART data transmission. Those states are the basis for a successful transmission and are mainly determined by the counter values and the input `rx_i`.

- **IDLE**
  The receiver waits until the input `rx_i` line is driven low indicating a new data transfer. During this state the counters are reset.
- **START BIT**
  During this state, the receiver waits until the start bit is over by counting the clock cycles using the baud-counter. The FSM transitions into the next state, whenever the baud-counter reaches its maximum value, which can be defined using the Verilog parameter `BAUD_COUNTS_PER_BIT`.

- **RECEIVING**
  In this state, the data bits are sampled. Once the rx-counter reaches its maximum (configured via the `UART_DATA_LENGTH` parameter), all the expected bits are sampled and the FSM continues to the next state. In addition, the baud-counter must also have reached its maximum before moving on to the next state. This ensures that the final bit is not lost.
- **STOP BIT**
  During this state, the FSM waits half a bit duration before continuing to the IDLE state.

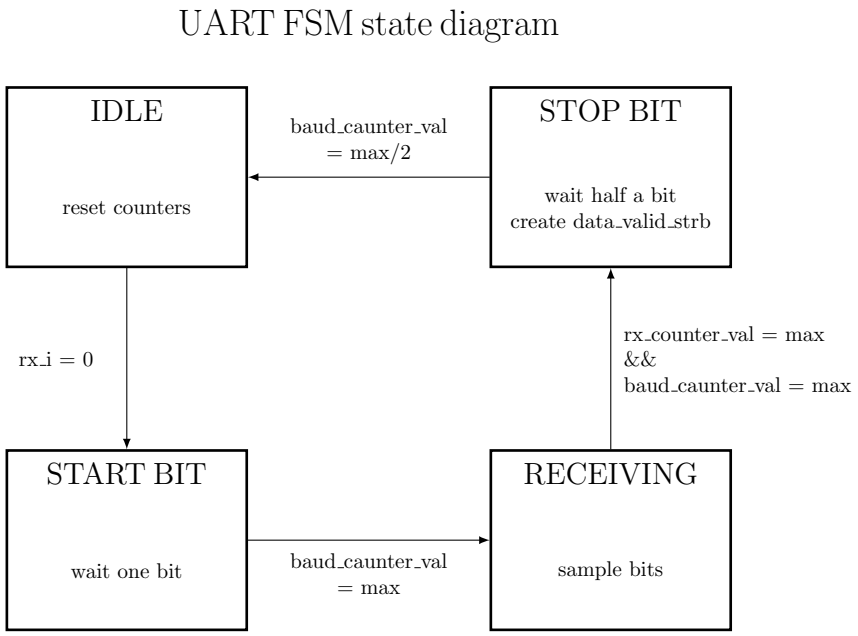

Figure 12: State diagram of the internal UART receiver FSM.

6.1.2 Baud Counter

The baud-counter counts the clock cycles during the START BIT, RECEIVING and STOP BIT state of the FSM. It simply counts up every positive clock edge until the `BAUD_COUNTS_PER_BIT` value is reached and then starts from 0. It should take the baud-counter exactly the duration of one bit to reach its maximum, therefore the `BAUD_COUNTS_PER_BIT` must be derived from the baud rate.

6.1.3 Bit Counter

While the baud-counter counts clock cycles, the bit-counter counts up every time the baud-counter overflows and starts again by 0. It therefore counts the incoming bits during the RECEIVING state. Outside the RECEIVING state the counter is reset to 0. The maximum value of this counter is defined by the parameter `UART_DATA_LENGTH`.

6.1.4 Sampler

The sampler is responsible for sampling the input data at the correct time. During the RECEIVING state, whenever the baud-counter is at `UART_DATA_LENGTH`/2 the `rx_i` is sampled and shifted into the `rx_data` register. When the bit-counter reaches its maximum, all the expected incoming values are stored inside the `rx_data` register in the correct order. During the transition between the STOP BIT and IDLE state, the sampler drives the `data_valid_strb_o` high for one clock cycle to indicate that the `data_o` is valid.

6.2 Operation Principle

Figure 13 shows how all the internal components work together to sample the incoming bits at the correct time. Note that the order of the sampled bits would get reversed due to the shifting nature of the `rx_data` register. Since the data is sent LSB first, the sent and received data (reversed) match.

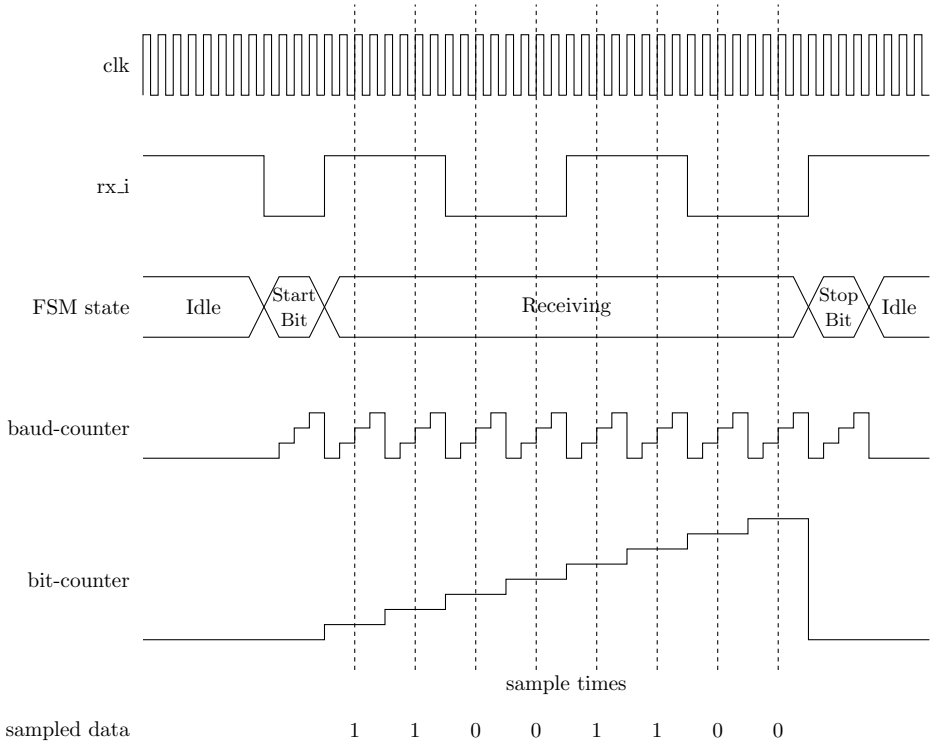

Figure 13: Example waveform of the `uart_rx` module combining all the internal components

6.3 Parameters

The `uart_rx` module is a generic module, where the baud rate can be set by defining the `BAUD_COUNTS_PER_BIT` parameter using following formula:

$$\text{BAUD\_COUNTS\_PER\_BIT} = \frac{\text{f}}{\text{baud rate}} \tag{1}$$

This project uses a baud rate of 19200 and a clock frequency of 10 MHz which results in:

$$\text{BAUD\_COUNTS\_PER\_BIT} = 520.8\dot{3}$$

As this is not an integer value, 521 is used instead. This results in a small timing error, which can be neglected as the effect resets every 8 bits.
The parameter `UART_DATA_LENGTH` is set to 8, as every UART packet includes 8 data bits in this case (no parity bit!).

6.4 Simulation

To simulate the `uart_rx` module, the project includes a `uart_rx_tb` test-bench file. It simulates the transmission of one UART packet containing following data: 1100 1100. The simulation result (Figure 14) shows, that the module is working as expected. The data is shifted into the register (see `data_o`), as soon as every bit is received, the `data_valid_strb_o` goes high for one clock cycle. During this time, the data matches the expected 1100 1100.

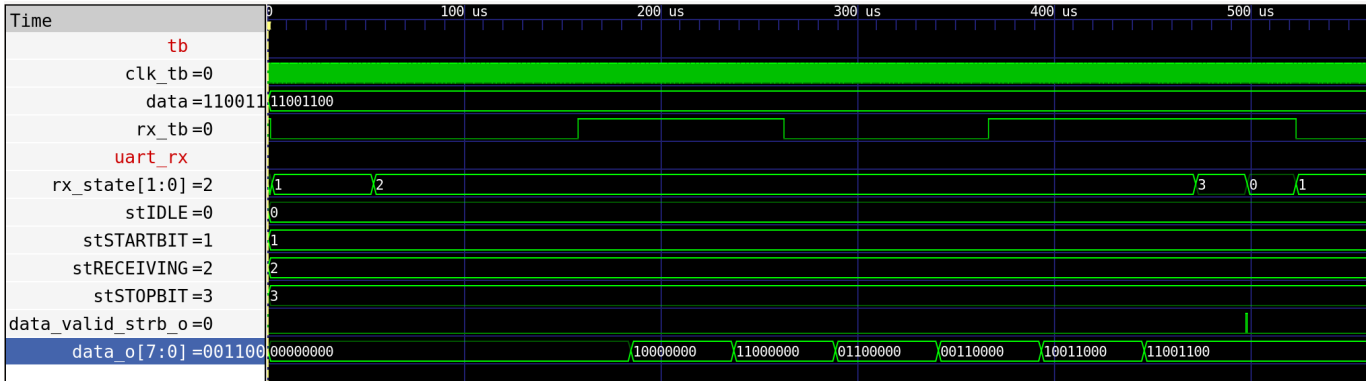

Figure 14: Simulation result of the `uart_rx` module.

7 Programmer

The programmer takes the 8 output bits from the UART receiver and converts them into two 4 bit blocks better suited for writing to the 4 bit memory.

7.1 Internal Structure

Table 9 lists all the in and outputs of the `programmer` module. Internally it utilizes a very minimalistic FSM with three states (see state diagram Figure 15):

- IDLE
- FIRST
- SECOND

The `programmer` includes an input register (8 bits) to cache the input data when `data_valid_strb_i` goes high for one clock period. Additionally, a small 4 bit counter counts up to create the `addr_o` signal.

I/O	Signal	Bit-width	Purpose
IN	`clk_i`	1	Clock signal
	`reset_i`	1	Asynchronous reset
	`active_i`	1	enable input
	`uart_data_i`	8 (variable)	input data from the UART receiver
	`data_valid_strb_i`	1	valid strobe for the input data
OUT	`data_o`	4 (variable)	output data to write to memory
	`addr_o`	4 (variable)	memory address
	`enable_write_memory_o`	1	memory write enable signal

Table 9: I/O of the `programmer` module

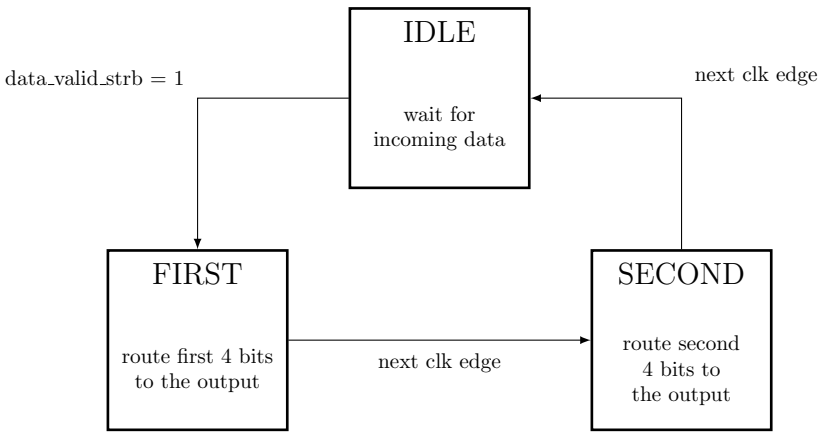

Figure 15: State diagram of the internal FSM of the `programmer`

7.2 Operation Principle

At startup, the FSM starts in the IDLE state and the addr-counter is set to 0. While the `active_i` signal stays low, the `programmer` keeps this IDLE state and ignores all incoming signals. When the `active_i` signal is asserted, the `programmer` operates as follows. As soon as the `data_valid_strb_i` goes high, the `programmer` caches the `uart_data_i` into the internal input register and advances into the FIRST state. During this state, the first 4 bits (MSB) of the input register are connected to the `data_o` output. In addition, the `enable_write_memory_o` is asserted and the `addr_o` is set to the addr-counter value, which is still 0 at this point. After one clock period, the FSM transitions into the SECOND state, which makes the addr-counter count up by one. Now, the last 4 bit (LSB) of the input register are connected to the `data_o` output and `addr_o` is set to the new addr-counter value of 1. Again the `enable_write_memory_o` gets asserted. At the following clock edge, the FSM loops back to IDLE state and the cycle starts from the beginning. Only the `reset_i` signal resets the addr-counter, not the IDLE state of the FSM. This causes the addr-counter to count up to 15 before it overflows and loops back to 0. Figure 16 shows the schematic of all the components of the `programmer` combined and their interconnections described above.

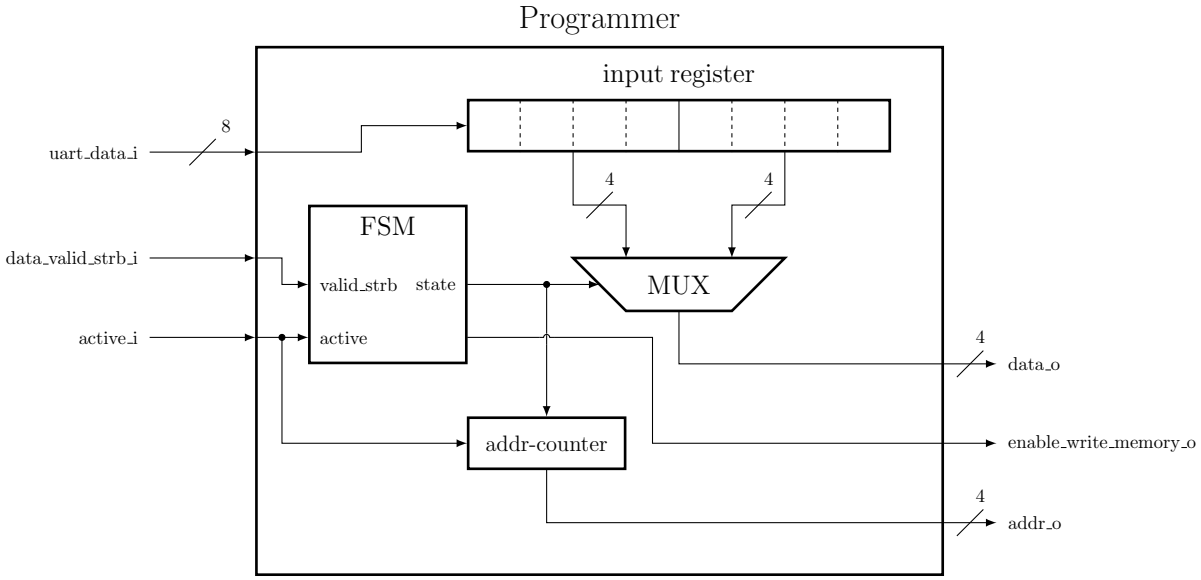

Figure 16: Schematic of the `programmer` module.

7.3 Simulation

The `programmer_tb` test-bench module simulates a few `data_valid_strb_i` cycles with following `uart_data_i`: 11010010. Figure 17 shows the simulation result. The output behaves as expected, dividing the input into two 4 bit output sections, first MSB then LSB. Also clear to see is, that the address is counting up and is not reset in the IDLE state. Although not pictured in Figure 17 (due to readability issues), the address resets after 8 input data blocks.

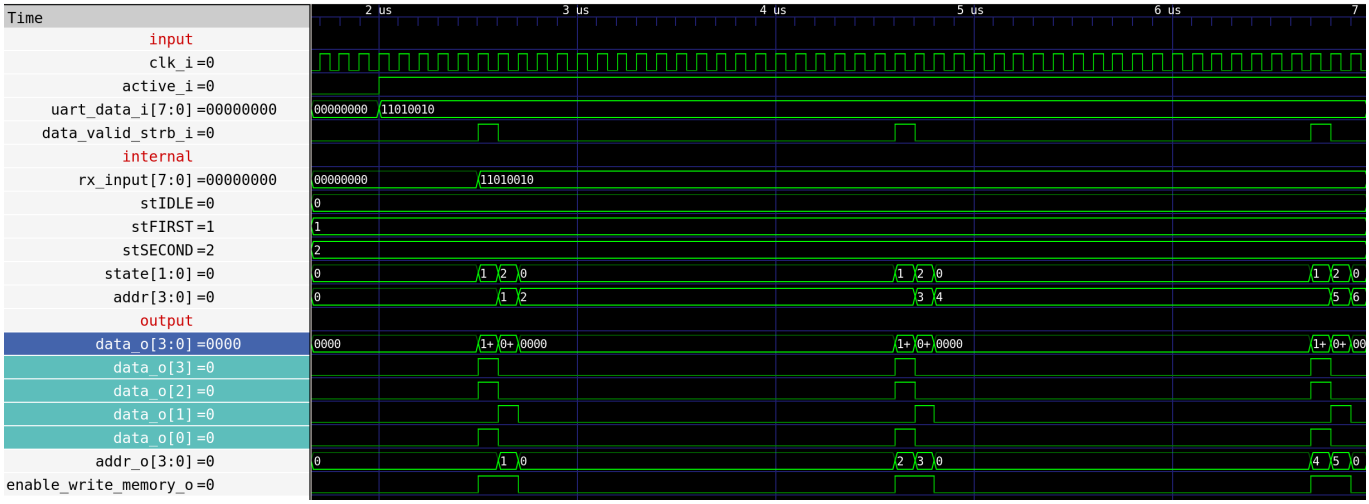

Figure 17: Simulation result of the `programmer` module.

8 Control Unit

The central and most important block is the `control unit`. It serves as the brain of the CPU, managing the data flow between all the registers and the memory, decoding the instructions, driving the control signals, controlling the programmer, and - most importantly - tracking of the current operation state using a finite state machine. It implements a multi-cycle design, which means that every instruction takes multiple clock cycles to perform.

8.1 External Interfaces

As the `control unit` is the central block, it is connected to all the components mentioned above. Those connections are grouped into interfaces for better structure and clarity.

8.1.1 I/O Interface

The I/O Interface hosts all the connections to the IN and the OUT register and some additional strobe signals. Those registers are the connection points to externally connected peripherals (via the pins on the TinyTapeout board, see subsection 2.3). Table 10 lists all the I/O interface signals.

I/O	Signal	Bit-width	Purpose
IN Register			
IN	`data_in_i`	4 (variable)	read data from the IN register
OUT	`write_en_in_o`	1	write enable for IN register
	`next_data_strb_o`	1	next data strobe for external peripherals
OUT Register			
OUT	`write_en_out_o`	1	write enable for OUT register
	`write_data_out_o`	4 (variable)	write data to OUT register
	`data_valid_strb_o`	1	data valid strobe for external peripherals

Table 10: Signals of the I/O Interface

8.1.2 Register Interface

The Register Interface features all the data and enable signals that enables the `control unit` to read and write data into the registers. As the connections for each register are the same, only a generic example is listed in Table 11. Use following `suffix` for the different registers:

- Instruction register: `ir`
- Accumulator register: `a`
- MDR register: `mdr`
- Operand register: `opnd`

I/O	Signal	Bit-width	Purpose
IN	data_<suffix>_i	4 (variable)	read data from the register
OUT	write_en_<suffix>_o	1	write enable for the register
	write_data_<suffix>_o	4 (variable)	data to write into the register

Table 11: Signals of the Register Interface

8.1.3 Memory Interface

The Memory Interface includes all signals used to communicate with the memory block (see Table 12).

I/O	Signal	Bit-width	Purpose
IN	read_data_mem_i	4 (variable)	read data from the memory
OUT	read_en_mem_o	1	read enable for the memory
	write_en_mem_o	1	write enable for the memory
	addr_mem_o	4 (variable)	memory address for read or write operations
	write_data_mem_o	4 (variable)	data to write into the memory

Table 12: Signals of the Memory interface

8.1.4 ALU Interface

The ALU is connected with the control unit via the ALU Interface, all connections are listed in Table 13.

I/O	Signal	Bit-width	Purpose
IN	result_alu_i	4 (variable)	result of the current ALU operation
	carry_alu_i	1	carry signal from the ALUs current operation
OUT	a_o	4 (variable)	first input data of the ALU
	b_o	4 (variable)	second input data of the ALU
	oc_o	3 (variable)	operation code to determine the current operation

Table 13: Signals of the ALU Interface

8.1.5 Programmer Interface

The Programmer Interface (Table 14) enables the `control unit` to establish a direct connection between `programmer` and `memory`. This way the `programmer` can write the incoming data to the `memory`.
Important to note is the difference between the two signals `p_programm_i` and `p_active_o`. The first of both is coming from an external peripheral and tells the `control unit` to enter the PROGRAM state and connect the `programmer` with the `memory`. Meanwhile the `p_active_o` is generated by the `control unit` once it enters the PROGRAM state and activates the `programmer`.

I/O	Signal	Bit-width	Purpose
IN	p_program_i	1	enable signal from external peripherals
	p_address_i	4 (variable)	memory address
	p_data_i	4 (variable)	data from the `programmer`
	p_write_eb_mem_i	1	write enable signal for the memory
OUT	p_active_o	1	activation signal for the `programmer`

Table 14: Signals of the Programmer Interface

8.1.6 Debug Interface

To make debugging easier from outside, the `control unit` generates 6 debug signals listed in Table 15 based on the state of the internal FSM (see 8.2.2). All of those signals are connected to pins on the TinyTapeout board to make them available from outside (see subsection 2.3).

I/O	Signal	Bit-width	Purpose
OUT	programm_o	1	high when control unit in PROGRAM state
	fetch_instr_o	1	high when control unit in FETCH_I state
	decode_o	1	high when control unit in DECODE state
	fetch_op_o	1	high when control unit in FETCH_O state
	fetch_mdr_o	1	high when control unit in FETCH_MDR state
	decode_o	1	high when control unit in EXEC or EXEC_ALU state

Table 15: Signals of the Debug Interface

8.2 Internal Structure

The control unit can be divided into 6 internal components displayed in Figure 18 (note that this image does not feature all connections, a lot of the I/O connections were omitted in favour of simplicity). The following subsections describe those 6 components in more detail.

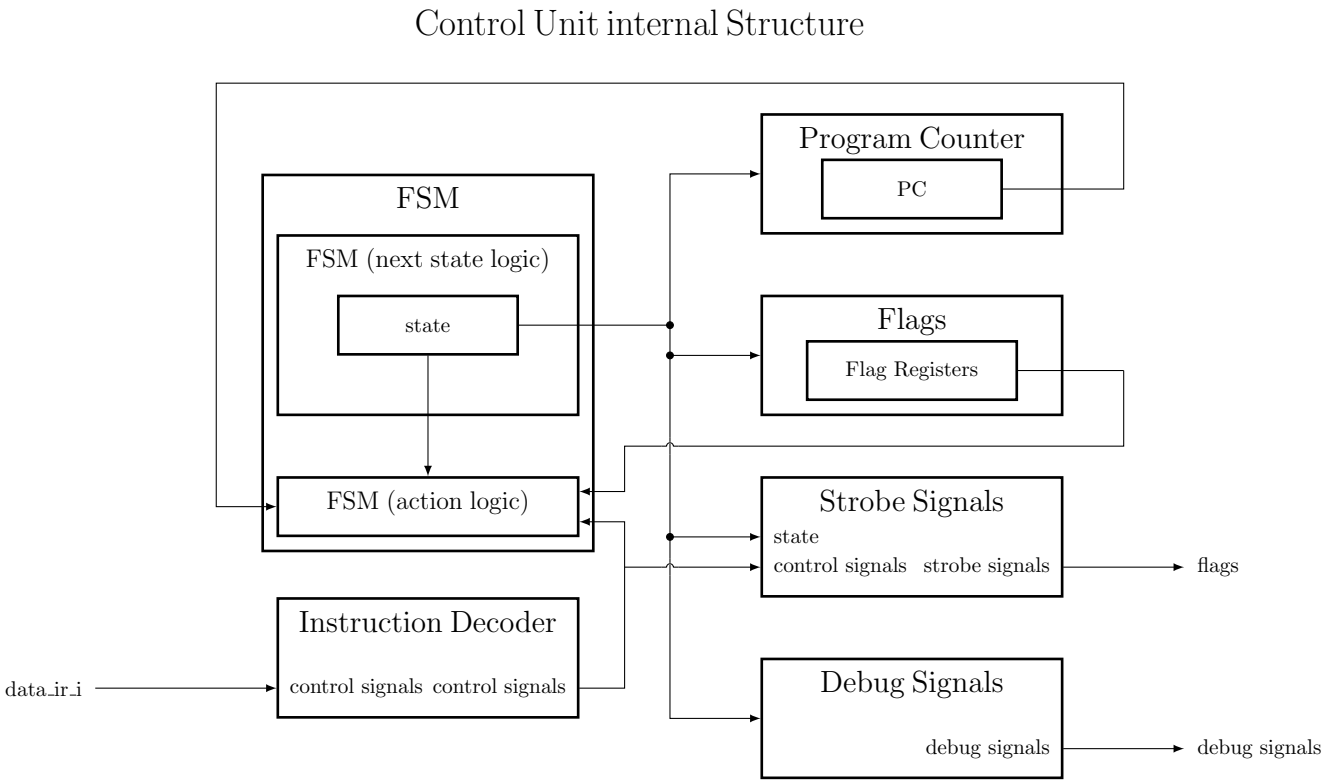

Figure 18: Internal structure of the control_unit

8.2.1 Instruction Decoder

The Instruction Decoder, as the name already suggests, decodes the currently active instruction, more explicitly, the operation code of the instruction stored in the INSTR register. It drives a set of control signals, on which all the other components relay on. For this purpose, some instructions can be grouped together to make the decoding simpler. Note that many instructions are part of multiple groups. For every group there is a separate control signal that the decoder asserts whenever the current instruction is part of this group.

ALU instructions: ADD, SUB, INC, DEC, AND, OR, XOR
Instructions that include the ALU for execution. The corresponding control signal is: alu_instr The ALU instructions are special, as the Instruction Decoder does not need to decode them further, this is already implemented in the ALU module itself (see 5.2.1). To detect an ALU instruction, the decoder has to check the first bit of the instruction op-code for a 0 and cancel out the possibility of a NOP instruction. This is done in Codesnippet 2 next to the // ALU_INSTR comment. In this case, the rest of the instruction op-code correspond to the ALU op-code and is therefore routed to the ALU (oc_o). For further processing later it is important to know if a INC or DEC instruction

is active. In this case the `inc_dec_instr` control signal is set to 1, see Codesnippet 2 next to `// INC or DEC INSTR` comment.

Operand instructions: `LDI, LD, ST, ADD, SUB, AND, OR, XOR, JMP, JZ, JC`
Instructions that include an operand and therefore must be treated specially. Following control signal is associated with this group: `operand_instr`. Operand instructions can be detected by ensuring that the last 3 bits of the instruction op-code do not match `101` or `110`. This leaves only operand instructions and the `NOP` instruction, which is handled separately and can be ignored here. The corresponding code is listed in Codesnippet 2 under the `// operand_instr` comment.

MDR instructions: `ADD, SUB, INC, DEC, AND, OR, XOR, LD`
Instructions that use the `MDR` register, again requiring special care using the control signal: `mdr_instr`. With the exception of the `LD` instruction, the set of MDR instructions coincides with the set of ALU instructions. Consequently, `mdr_instr` can be generated using a simple logical OR operation (see the top of Codesnippet 2), provided that the decoder correctly identifies when an `LD` instruction is active. `INC` and `DEC` are special cases, they require the `MDR` register to store 1 as the second summand.

The group control signals are used to determine which processing steps are required during instruction execution. In addition, all non-ALU instructions must be explicitly decoded. For this purpose, dedicated control signals are asserted whenever the corresponding instruction is detected (see at the bottom of Codesnippet 2).

8.2.2 FSM

The internal FSM is organized in two main logic parts: **next state logic** and **action logic**. The next state logic determines the subsequent state based on the current state and the control signals from the decoder. The action logic is responsible for driving all the interface signals and orchestrating the data flow based on the current state and control signals. Figure 19 ilustrates the state diagram of the FSM. Some transitions are associated with priority numbers. If the condition of multiple transitions are satisfied at the same time, the one with the lower priority value will be selected. This behaviour is represented with `if-else` statements in the Verilog code. With the exception of the PROGRAM and the RESET state, every state is only active for one clock period before the FSM transitions to the next one.

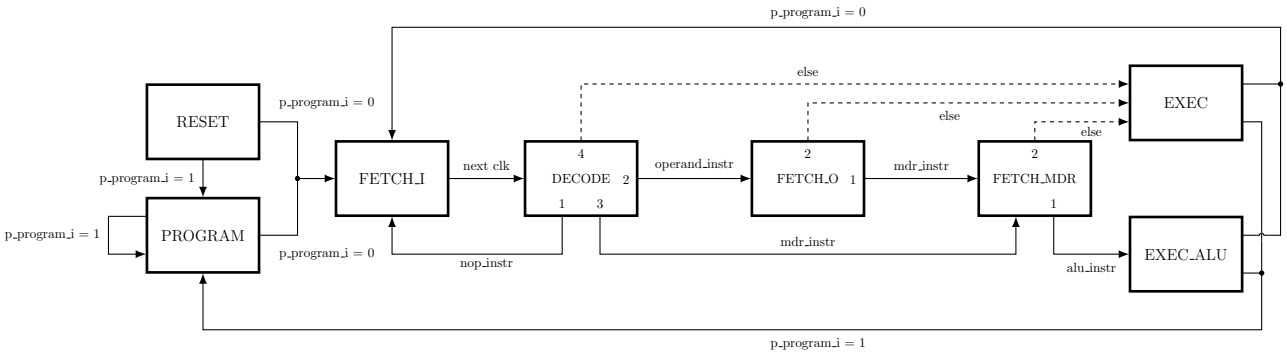

Figure 19: Control Unit FSM state diagram

RESET
The RESET state is the default state in which the FSM starts after the reset is released.

PROGRAM
If the `p_program_i` input is pulled high, the FSM enters the PROGRAM state. In this state the action logic drives the `p_active_o` high, activating the `programmer` as already explained in 8.1.5. The `programmer` is coupled to the `memory` by connecting the enable, address and data lines. Additionally the program counter and the `A` register are reset to 0. As soon as the `p_program_i` is released, the next state logic continues with the FETCH_I state.

FETCH_I
During this state, the action logic performs a read operation on the memory at the address specified by the current program counter and stores the data in the instruction register. The FETCH_I state is always followed by the DECODE state, during this transition, the program counter is increased by 1.

```
// alu and ld instruction use the MDR Register, therefore assign mdr_instr
assign mdr_instr = ld_instr || alu_instr;

   always @(*) begin

       // standard assignment
       oc_o = {OPERATION_CODE_WIDTH{1'b0}};
       nop_instr = 0;
       operand_instr = 0;
       alu_instr = 0;
       inc_dec_instr = 0;
       jmp_instr = 0;
       jz_instr = 0;
       jc_instr = 0;
       ld_instr = 0;
       st_instr = 0;
       in_instr = 0;
       out_instr = 0;
       ldi_instr = 0;

       // operand_instr
       if (!(data_ir_i[2:0] == 3'b101 || data_ir_i[2:0] == 3'b110)) begin
           operand_instr = 1; // instruction with operand
       end

       if (data_ir_i[3] == 1'b0) begin
           if (data_ir_i[2:0] == 3'b000) begin
               nop_instr = 1;              // NOP
           end else begin
               alu_instr = 1;              // ALU_INSTR
               oc_o = data_ir_i[2:0];
               if (data_ir_i[2:0] == 3'b110 || data_ir_i[2:0] == 3'b101) begin
                   inc_dec_instr = 1;      // INC or DEC INSTR
               end
           end
       end else begin
           case (opcode)
               3'b000: jmp_instr = 1;      // JMP_INSTR
               3'b001: jz_instr = 1;       // JZ_INSTR
               3'b010: jc_instr = 1;       // JC_INSTR
               3'b011: ld_instr = 1;       // LD_INSTR
               3'b100: st_instr = 1;       // ST_INSTR
               3'b101: in_instr = 1;       // IN_INSTR
               3'b110: out_instr = 1;      // OUT_INSTR
               3'b111: ldi_instr = 1;      // LDI_INSTR
               default: ;
           endcase
       end
   end
```

Codesnippet 2: Verilog code of the decoder

DECODE
This state gives the decoder combinatoric time to calculate all the control signals based on the content of the instruction register. Now the next state logic checks if the `nop_instr` control signal is high. In that case the FSM loops back to FETCH_I to begin reading the next instruction. If the first case does not apply, the next state logic checks whether the `operand_instr` signal is high. This causes the FSM to transition to the FETCH_O state to fetch the operand (for example during a `ADD` instruction). If this is also not the case, the `mdr_instr` signal determines if FSM switches to the FETCH_MDR state (only possible with `LD` instruction) or skip all the fetch steps and jump straight to the EXEC state (`OUT` or `IN` instruction for example).

FETCH_O
During the FETCH_O state, the action logic orchestrates the fetching of the operand into the `OPND` register. The

operand is stored inside the memory at the subsequent address of the associated instruction op-code. As the program counter has already been increased by one before (after FETCH_I), it now points to the operand. If the `mdr_instr` signal is high, the next state will be the FETCH_MDR, if not, the FSM jumps to the EXEC state. In both cases the program counter is increased by 1.

FETCH_MDR
This step fetches the data from the memory, stored at the address specified by the operand, in the `MDR` register. In the special case of a `INC` or `DEC` instruction, 1 is written to the `MDR` register. Depending on the execution nature of the current instruction (ALU instruction or not), the next state logic choses the EXEC or the EXEC_ALU.

EXEC_ALU
The output `a_o` is connected to the `A` register and the `b_o` to the `MDR` register. The result `result_alu_i` is written back into the `A` register. At this point, the FSM can jump back to the FETCH_I state, to continue with the next operation or enter the PROGRAM state, depending on the `p_program_i` signal.

EXEC
Based on the control signals associated with the non-ALU instructions, the action logic drives the necessary signals to execute the correct instruction.

- jump instructions
  The program counter is set to the data stored in the `OPND` register. For `JZ` and `JC` it checks the correct flags before.
- `IN` and `OUT`
  To execute the `IN` instruction, the data from the `IN` register is copied into the `A` register. The `OUT` instruction leads to a data flow from the `A` register to the `OUT` register.
- `LD` and `LDI`
  During a `LD` the data from the `MDR` register is copied into the `A` register. On the other hand, the `LDI` instruction loads the `OPND` data into to `A` register.

Again based on the `p_program_i` signal, the FSM continues with the FETCH_I or the PROGRAM state.

8.2.3 Program Counter

The program counter keeps track of the current memory address and is represented by a simple 4 bit register. It's value is increased by 1 after the FETCH_I and the FETCH_O state. Jump instruction can set the program counter to any arbitrary value.

8.2.4 Flags

The flags are used in the `JZ` and `JC` instructions. They indicate if the last ALU-Instruction resulted in an overflow (carry flag) or in all zeros (zero flag). Both are 1 bit registers controlled by the flag logic based on the EXEC_ALU state, the `carry_alu_i` signal and the `result_alu_i` signal.

8.2.5 Strobe Signals

The strobe signals should help external peripherals with the communication using the `IN` and `OUT` registers. Therefore both strobe signals are routed to external pins on the TinyTapeout board to make them available for external peripherals. The `next_data_strb` signals when the current data of the `IN` register is processed and new data can be applied on the input pins. Figure 20 illustrates the correct timing, note that the first strobe happens after the first `IN` instruction got executed. Make sure that the first data packet is present from the beginning.

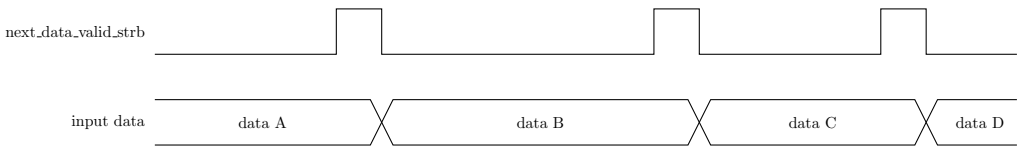

Figure 20: Timing diagram of the `next_data_strb` combined with the input data

To signal the external peripheral when the output data is valid, the `data_valid_strb` gets asserted for one clock period. During this time, the data can be used in any subsequent block.

9 CPU

The cpu module combines all the previously discussed components. It does not include any additional logic. Figure 21 illustrates the interconnections between the components. All the in and outputs are listed in Table 16.

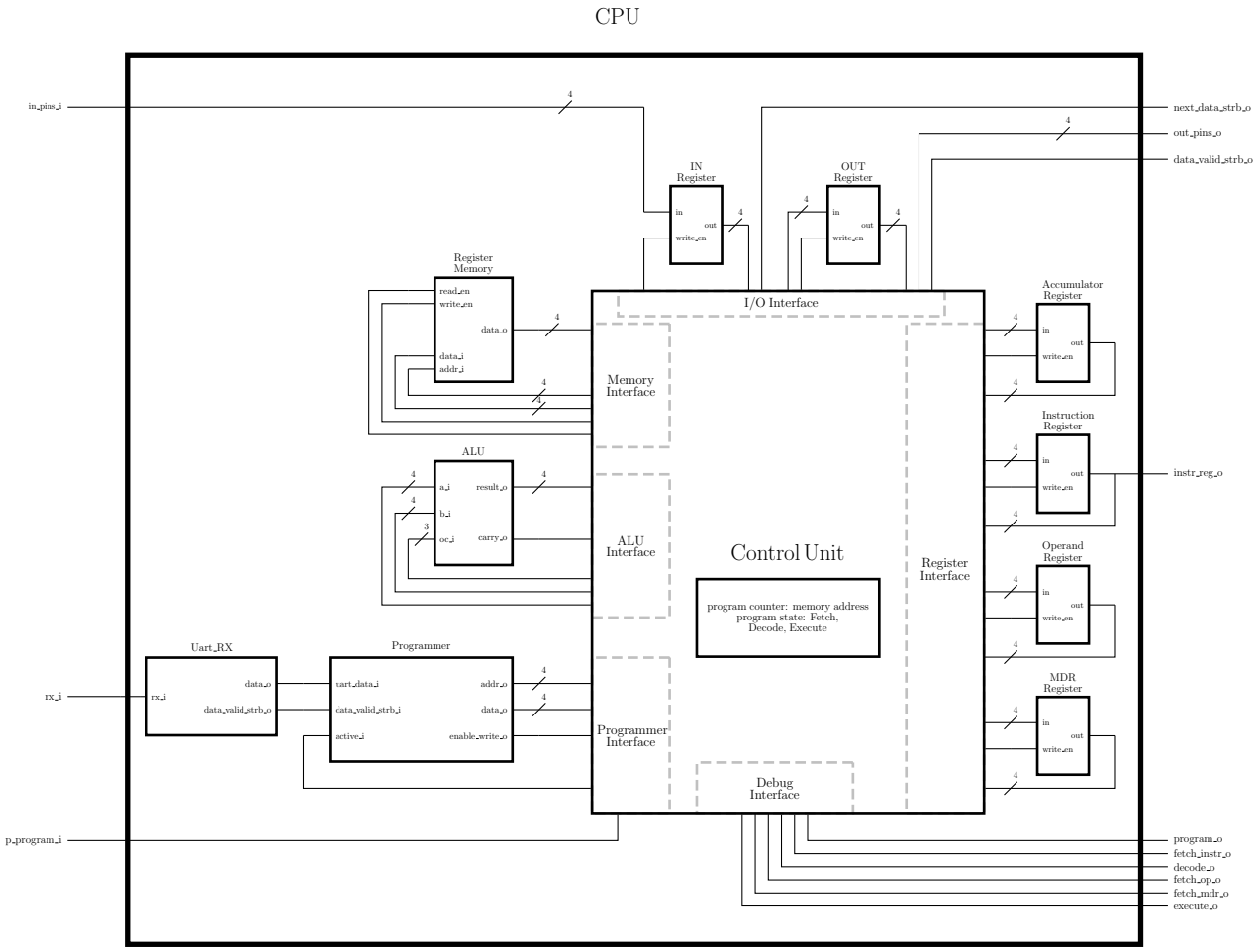

Figure 21: Architectural overview of the cpu module.

I/O	Signal	Bit-width	Purpose
IN	clk_i	1	global clock signal
	reset_i	1	global asynchronous reset signal
	in_pins_i	4 (variable)	external input data for the IN register
	p_program_i	1	external enable signal to enable programming mode
	rx_i	1	UART receive signal
OUT	next_data_strb_o	1	next data strobe for the in_pins_i
	out_pins_o	4 (variable)	output data to external peripherals from the OUT register
	data_valid_strb_o	1	data valid strobe for the out_pins_o
	instr_reg_o	4 (variable)	content of the INSTR register
	debug signals	6	debug signals, see 8.1.6

Table 16: I/O signals of the cpu module

9.1 Simulation

The cpu_tb module is the test-bench to test the cpu module. It lets the CPU run the default program for a while (adding 1 in a loop) and then enters the program mode by asserting the p_program_i. The default program is overwritten by the data show in Codesnippet 3. This program implements a circular left shift. The test-bench also includes a few other test programs to choose from.

```
        data_array = {
            IN_INSTR,
            OUT_INSTR,
            ST_INSTR,
            4'b1111,
            ADD_INSTR,
            4'b1111,
            JC_INSTR,
            4'b1010,

            JMP_INSTR,
            4'b0001,
            INC_INSTR,
            JMP_INSTR,
            4'b0001,
            NOP_INSTR,
            NOP_INSTR,
            4'b0001
        };
```

Codesnippet 3: Test-program for the `cpu` simulation.

Figure 22 shows the simulation result before overwriting the default program code. It is clear to see, that the output data is increased one by one as expected.

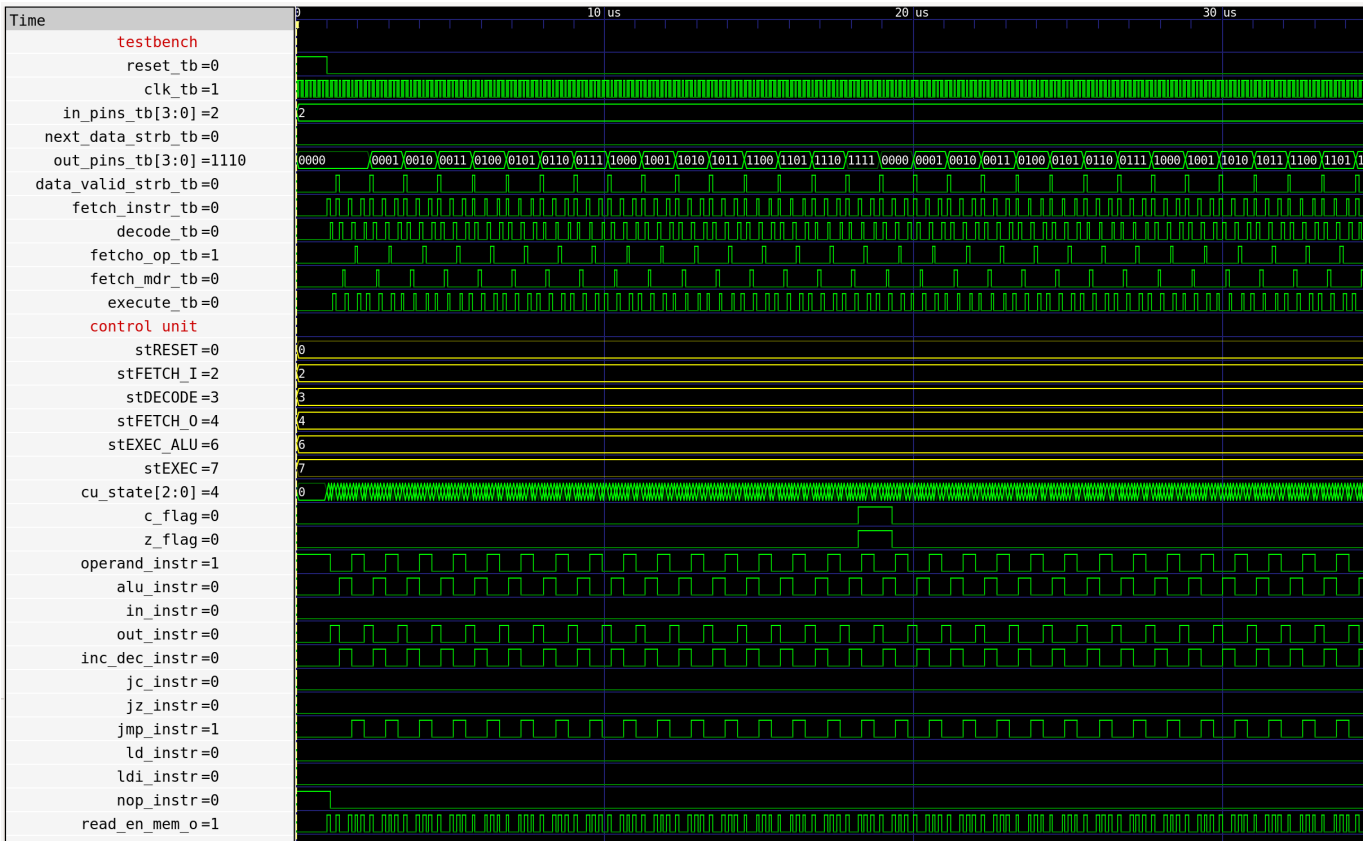

Figure 22: Simulation result of the `cpu` module running the default program code.

During the programming, the simulation shows clearly, that the memory values are overwritten successfully (Figure 23). Running the shift test program also creates the expected results (Figure 24), the output pins show the desired circular left shift.

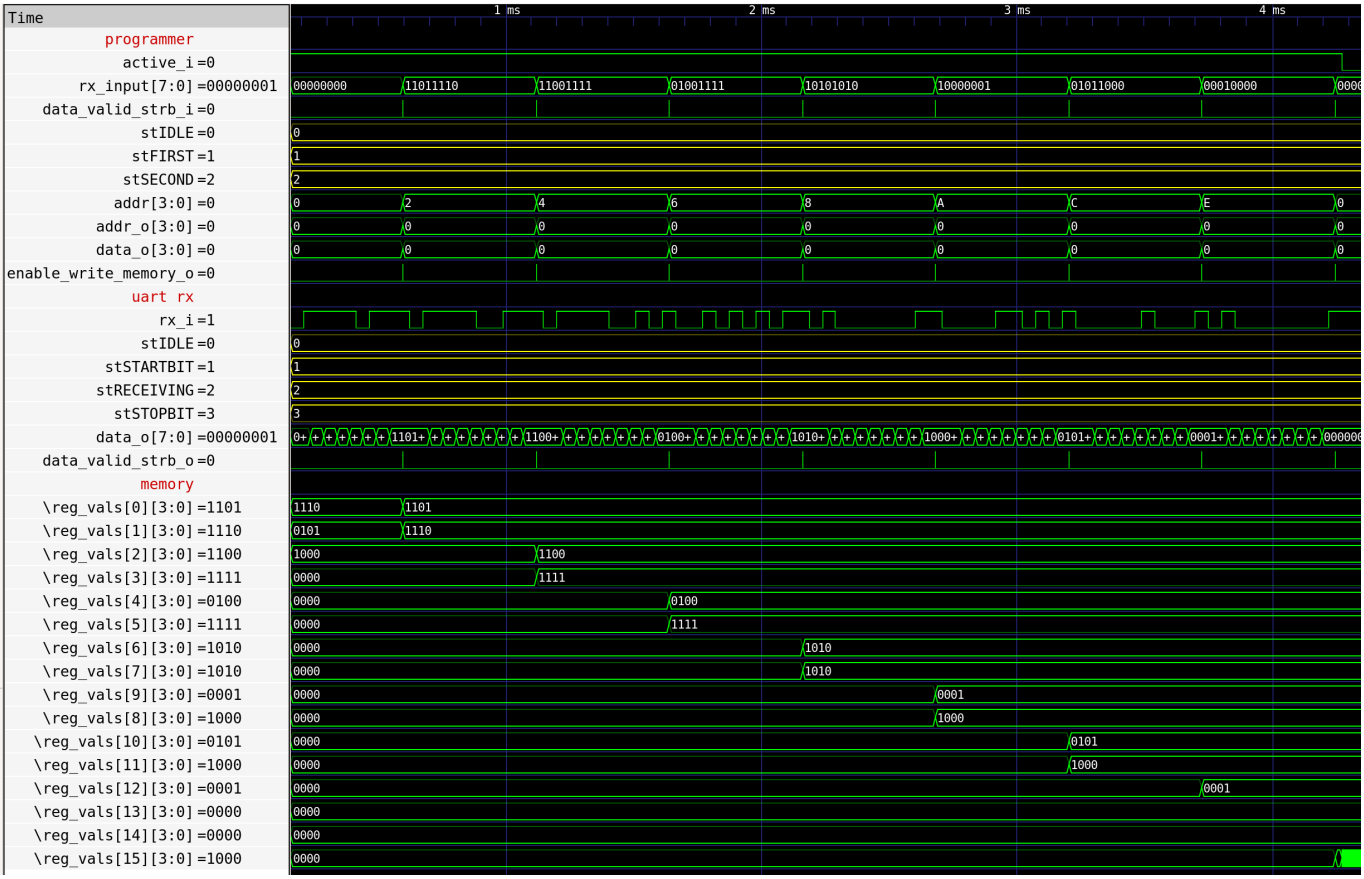

Figure 23: Simulation result of the `cpu` module during programming mode.

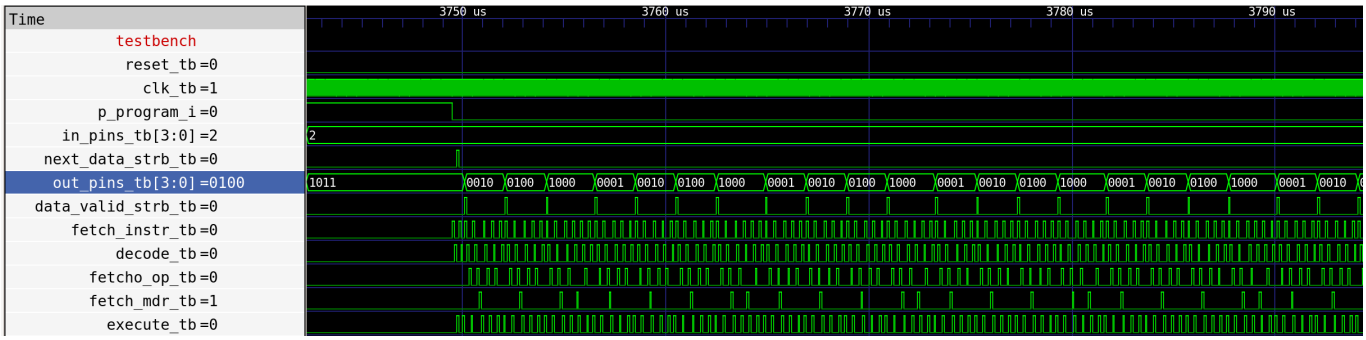

Figure 24: Simulation result of the `cpu` module running the shift test program.

9.2 Testing on FPGA

The design has ben tested on an Nexys Video Board with a Artix-7 FPGA (XC7A200T-1SBG484C). The CPU worked fine and the UART including the programming mode also work. Sadly, I forgot to take pictures to prove it...

9.3 GitHub actions

Figure 25 proves that the project did pass all the GitHub actions.

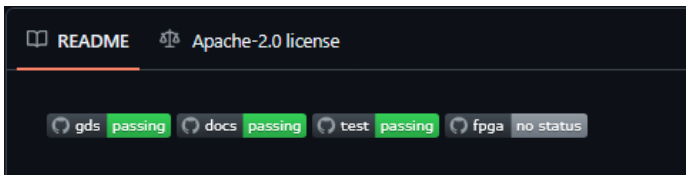

Figure 25: GitHub actions

9.4 Synthesis and Linter

9.4.1 Synthesis warnings

In the GitHub actions, one synthesis warning shows up:
`Warning: Replacing memory \reg_vals with list of registers.`
`See /home/runner/work/Custom-4Bit-CPU/Custom-4Bit-CPU/src/reg_memory.v:44`
This warning appears, because the Yosys detected something that looks like a memory array but it can't (or chooses not to) infer an actual memory. In this case, the memory array is most likely the registers of the custom register memory. It can't infer an actual memory block, because the TineyTapeout does not support it (as far as I know). This warning will not cause any harm and can therefore be ignored.

9.4.2 Linter warnings

Warnings like:
`Warning-UNUSEDPARAM: /home/runner/work/Custom-4Bit-CPU/Custom-4Bit-CPU/src/reg_memory.v:23:16: Parameter is not used: 'ADD_INSTR'`
can be ignored. The `reg_memory` has some local parameters that resemble the instructions, to make it easier to create the default program. A few instructions do not appear in the default program and are therefore not used.

All the other warnings are either because a few inputs are not used, which is totally fine, or because the length of the parameters does not fit the length of the signals they are compared with. As the simulation and the FPGA design work, this should cause any problems.