MIPS 1 in VHDL

HDL Lab - SS 2015

Bahri Enis Demirtel, Carlos Minamisava Faria, Lukas Jäger, Patrick Appenheimer

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Elektrotechnik und Informationstechnik
Fachgebiet Integrated Electronic Systems Lab

Contents

1. Introduction **1**
1.1. Task 1

2. Design **2**
2.1. ALU 3
2.2. Datapath 3
2.2.1. Instruction fetch 4
2.2.2. Instruction decode 4
2.2.3. Execution 6
2.2.4. Memory Stage 6
2.2.5. Write-back 6
2.3. Controlpath 6
2.3.1. Decoder for the instruction-fetch- and instruction-decode-stage 7
2.3.2. Decoder for the execution-stage 7
2.3.3. Decoder for the memory stage 8
2.3.4. Decoder for the writeback stage 8

3. Evaluation **9**

4. Synthesis **10**

5. Conclusion **11**

6. Bibliography **12**

A. Code **13**

1 Introduction

In the HDL Lab is a practical exercise of a hardware description language implementation. This semester the task is the implementation of a MIPS I microcontroller in vhdl. A requirement to this laboratory is the lecture HDL: Verilog and VHDL by Prof. Dr.-Ing. Klaus Hofmann.
MIPS is an acronym for Microprocessor without interlocked pipeline stages. The MIPS instruction set is a reduced instruction set computer (RISC). There are available references for 32 and 64-bit with many revisions.
MIPS was developed in the 80s with the intent to take fully advantage of pipelines. Nowadays this instruction set and structure is often used as an hdl first project. Commercially it is used embedded systems such as Windows CE devices, routers, residential gateways and video consoles such as Nintendo 64, Sony PlayStation, PlayStation 2 and PlayStation Portable.

1.1 Task

The objective is to design, implement, synthesize and test a MIPS-I specified processor core on FPGA. The hardware description language is VHDL and the target technology is a Virtex5 from Xilinx. The synthesized microcontroller must be able to run at 50 MHz with a desirable frequency of 200 MHz. The microcontroller must use a pipeline of a minimum of 2 and maximum of 6 stages. The following sub-components are mandatory: ALU, data-path and control-path.
A counter test program shall run on the synthesized microcontroller, outputting the counter value to eight LEDs on the FPGA board.

2 Design

This chapter describes the design of a MIPS 1 microcontroller. A microcontroller consists mostly of a processor core, memory and programmable inputs/outputs peripherals. This design implementation focus only on the processor core design.

This laboratory requires a microcontroller structure of at least a CPU containing a controlpath, datapath and an ALU. The created design uses this base with a 5-staged pipeline. The CPU interacts with two external memories and has also a clock and a reset input. All of which are considered external to this design. The CPU is divided in two base components: a control and a datapath block, as shown in Figure 2.1.

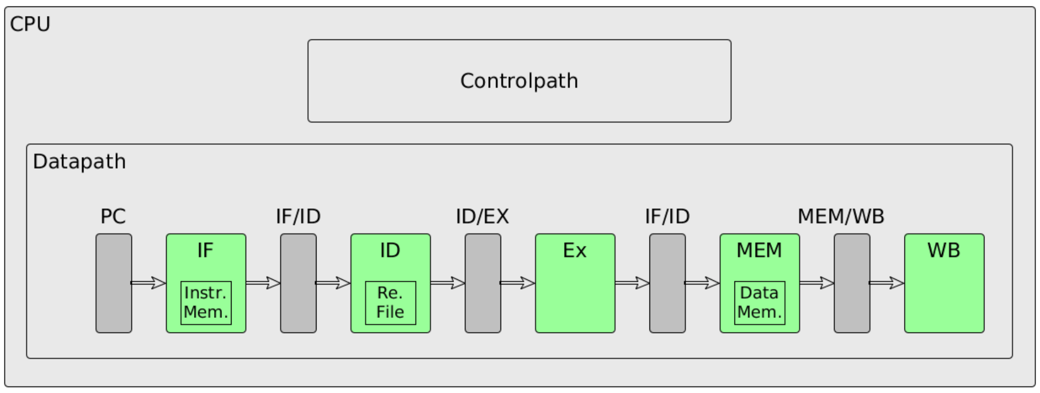

Figure 2.1.: Datapath pipeline

In the datapath there is the pipeline made of five blocks: instruction fetch (IF), instruction decode (ID), execution (EX), memory stage (Me) and writeback (WB).

The following sections describe the central component ALU and the two base components: datapath and controlpath. The cpu components structure is shown in Figure 2.2:

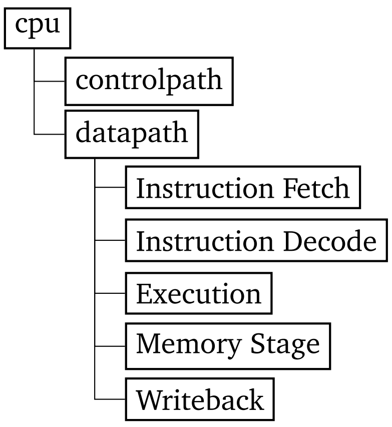

Figure 2.2.: CPU component structure

2.1 ALU

This section describes the ALU (arithmetic logic unit). The implementation of the ALU has three parallel data buses consisting of two 32bit input operands and a 32bit result output. Furthermore, there is an 6bit input for the opcode. The ALU can perform 7 operations: add, sub, and, or, sll, slt and interconnect one of the two inputs to the result output. Additionally, there is an zero output flag if the operation results in zero.

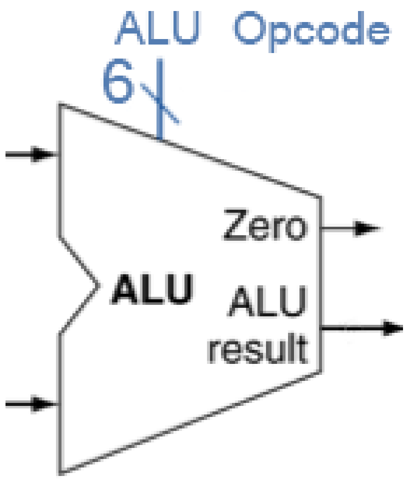

Figure 2.3.: ALU

2.2 Datapath

This section describes the data-path and its internal elements. The data-path is the component that connects the pipeline components within itself as well as with CPU inputs and outputs and the control-path.

This MIPS implementation works with a 5 stage pipeline in order to achieve a fast clock. The data-path consists of instruction fetch, instruction decode, execution, memory stage and write-back. The data-path controls the information flow from one pipeline stage to the next with registers. These writing process occur on the positive edge of the clock when the pipeline stage input from the control-path, so that the registers forwards information synchronously. The data-path forwards the control-path signals asynchronously, contrary to the pipeline to pipeline signals.

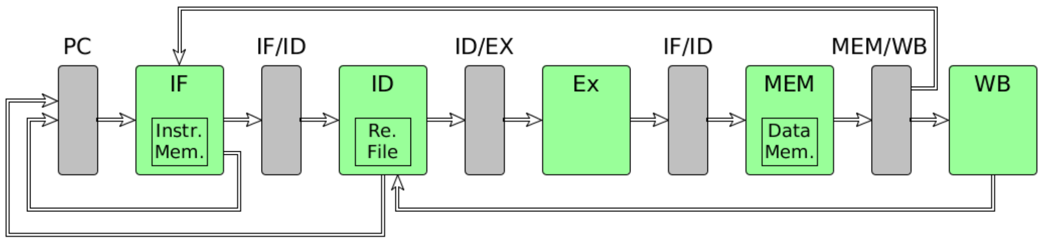

Figure 2.4.: Data-path pipeline

The program counter (PC) is programmed into the data-path. Its function is the store the current program address, which is mostly counted up. It has a multiplexer controlled by the control-path to choose the input. The two possible inputs are to count up (PC+4) from instruction fetch and the jump or branch from instruction decode. The control-path chooses always the instruction decode input in the case of jump or branch.

The following subsections describe the pipeline components as well as its functions and IOs.

2.2.1 Instruction fetch

This first block of the pipeline is the instruction fetch. The main task of this block is to fetch the next instruction and relay it to the pipeline.

The program counter is a 32-bit input, which is directly outputted as instruction address to fetch an instruction. The instruction fetch inputs the program memory's instruction data, with the 32-bit instruction. This value is directly forwarded to the pipeline. The memory word is one byte long, but each instruction read operation retrieves four bytes or the 32-bit word.

The program counter is also incremented by four, because the used memory is 8-bit long, pointing to the next valid instruction. This incremented value is given back to PC.

2.2.2 Instruction decode

The second block of the pipeline if the instruction decode. Its main tasks are to divide the instruction into its pieces, manage the register file and manage branches.

The main input is the instruction from the instruction fetch stage. This instruction is 32-bit long and can be of three types. These are shown in Table 2.1 [1].

Table 2.1.: MIPS instruction types

Type	format (bits)					
R	opcode (6)	rs (5)	rt (5)	rd (5)	shat (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

The **opcode** indicates the operation or arithmetic family of operations. Opcode equals zero are the R-type operations. The field **funct** provides an specific operation. **rs**, **rt** and **rd** provide sources or destinations register addresses. **shamt** indicates the shift amount for shift operations. **immediate** carries a relative address or constant, which is zero or sign extended to 32-bits. **address** is an absolute address.

The main outputs are register A, register B, shift, regdest, immediate and IP. Other than IP, all outputs depend on the instruction decoding. Register A contains always the value that is contained in the register given by the source register field (instruction's sub-vector 25-21). Register B likewise always contains the value of the register given by the R-type-instruction's field for the target register (instruction's sub-vector 20-16). Immediate always contains the lower half of the instruction with a signed extension.

Regdest has to be chosen by the control path. It can be either the target register field or the R-type's destination register field or 31, which was originally implemented for jump-and-link- or branch-and-link-instructions, but rendered pointless by the more complex branch logic and write-back functionality for register 31. Shift also must be chosen by the control path. It can be set to the R-type-instruction's shift-field, to 16 or to 0.

The execution stage chooses the signals necessary for an operation using two multiplexers so if any output yields a nonsensical value is is simply not used.

Register File

The register file is a set of 32 general purpose 32-bit registers. These have the advantage, comparing to the ram memory, that they can always be accessed within one clock cycle. The access to these registers is made with five bits, which allows multiple registers to be referenced per instruction. All loaded memory values are stored in a register for later use.

The registers are numbered from $0 through $31. There is also a convention for using these registers, which must be enforced by assembly language and follow Table 2.2 [2]:

Table 2.2.: MIPS registers

Register Number	Conventional Name	Usage
$0	$zero	Hard-wired to 0
$1	$at	Reserved for pseudo-instructions
$2 -$3	$v0, $v1	Return values from functions
$4 - $7	$a0 - $a3	Arguments for functions - not preserved by subprograms
$8 - $15	$t0 - $t7	Temporary data, not preserved by subprograms
$16 - $23	$s0 - $s7	Saved registers, preserved by subprograms
$24 - $25	$t8 - $t9	More temporary registers, nor preserved by subprograms
$26 - $27	$k0 - $k1	Reserved for kernel. Dot not use.
$28	$gp	Global Area Pointer (base of global data segment)
$29	$gp	Stack pointer
$30	$sp	Frame Pointer
$31	$ra	Return Address

This implementation of MIPS does not have a FPU. In case of FPUs another 32 32-bit register set is used.

The register file is written on the clock's negative edge with write-back information. The register file require a 5-bit destination register address and the 32-bit word to be written into the register. Additionally there is the possibility to write to register 31. On every rising clock cycle, an internal write-back flag is checked. If it was set by another process, the value of the internal write-back register is copied to register 31. This bypasses some pipeline stages, but makes control path design a little easier.

Branch Logic

The branch and jump instructions require just one clock between instruction fetch and the jump itself. Due to this constrain, there is the need of a branch logic inside the instruction decode part. The output of this operation is the next instruction address for PC.

On the jump command, PC will receive the jump value. In case of a branch, PC will receive either the branch value or PC+4, depending on the instruction decode decision. This behavior allows for the control-path always to activate the instruction decode input in cases of jumps and branches. If a Jump-and-link- or a branch-and-link-instruction is detected, the branch logic writes the last program counter to the internal write-back register and sets the internal write-back flag to 1 to signal a necessary copy operation to the register bank. It evaluates the flag and treats the internal write-back register as described above.

Forwarding

Often calculated or memory read values are used in the following instructions. Due to the pipeline, the values are not ready in the register file, causing a data hazard. In order to avoid this conflict a data forwarding system is integrated. The data forwarding provide separated inputs for the 5-bit destination register address and the 32-bit word for the ALU, memory stage and write-back. If the destination register address in one of this stages is equal to an address used in the current instruction decode phase, the value of the register bank is replaced by a forwarded value. The forwarding system takes care to always use the most recent value. For example, if both write-back and execution stage contain the same destination address, the execution stage's value is forwarded because the value that has to be written to that destination register was changed by the instruction in the execution stage after it was changed by the instruction in the write-back stage, so the value contained in the execution stage is the most recent.

2.2.3 Execution

This stage of the pipeline takes care of the actual mathematical operations. It provides two main multiplexers, one for each value input of the ALU. The inputs of the first multiplexer are the zero padded shift input, the number four (32-bit) and the register A from instruction decode. The second multiplexer provides register B, the immediate value and IP as inputs.

Both multiplexers are controlled by the control-path.

2.2.4 Memory Stage

The memory stage is the fourth block of the pipeline and has the main task of fetch or save in the memory.

For memory operations the execution stage outputs two 32-bit values: aluResult_in, which works as the memory address, and data_in, which is data to be written in the memory. On read operations, the data_to_cpu input delivers the 32-bit memory value.

This stage has one multiplexer choosing the pipeline stage output from aluResult_in or data_to_cpu.

2.2.5 Write-back

This write-back stage is the fifth and last stage of the pipeline. Its main task is just to hold the calculated values, as well as the values read from the memory so they can be written the register file.

2.3 Controlpath

The control path is not designed as a finite state machine. Due to its simplicity we chose an approach that is closer to the pipelined structure of the MIPS-Architecture. The control path is built around a 32-bit wide 4-deep shift register. When the memory returns the instruction to the instruction-decode-stage of the datapath, it is also fed into the shift register and propagates in the following clock cycles. To the stages of this shift register we attatched a decoder that produces control signals for one stage of our datapath. Each stage of the shift register matches exactly one stage of the datapath, except for the first controller stage which handles instruction fetch and instruction decode. The second stage returns the control signals for the execution stage, the third stage is mapped to the memory stage and the fourth stage controls the datapath's writeback stage. In case of any stalls the propagation of the instructions through the shift register is halted.

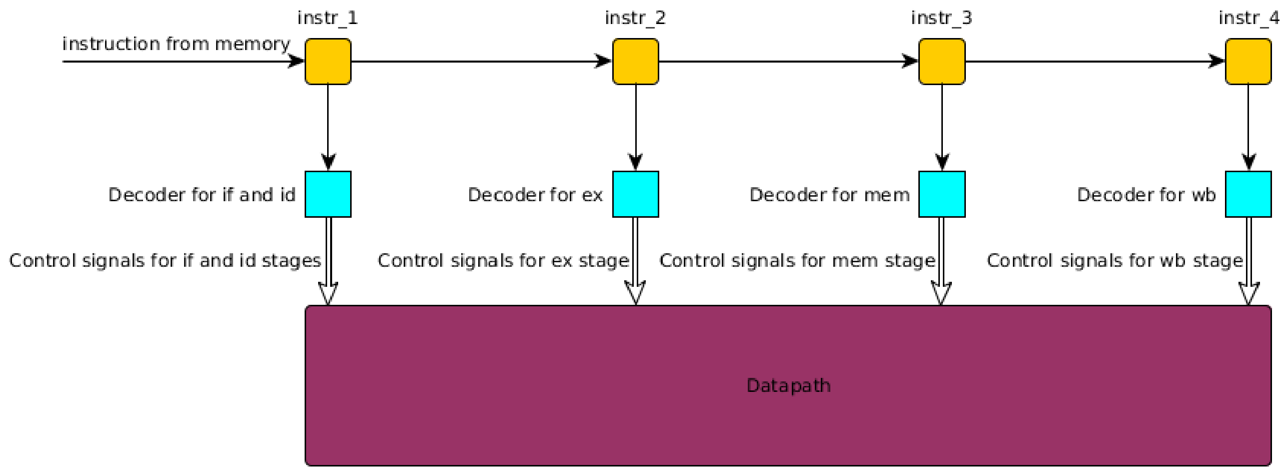

Figure 2.5.: Control path

2.3.1 Decoder for the instruction-fetch- and instruction-decode-stage

The decoder attatched to the first stage of the shift register distinguishes between the opcodes of the given instruction to determine the control signals. If the opcode is 000000, an R-type-instruction is assumed and the instruction-decode-stage's destination register control signal is set to 0 to use the R-type destination register. The shift multiplexor is set to 0 as well to use the R-type-instructions shift field. For the settings of the pc's multiplexor in the instruction-fetch-stage, the other bits of the instruction are evaluated. If they make the instruction a Jump-Register-instruction, the signal is set to 1 so that the new program counter calculated by the branch logic is used instead of the previous value incremented by 4. In every other case, this multiplexor's control signal is set to zero because all other R-type-instructions do not modify the program counter.
All non-R-type-instructions are treated as I-type-instructions by the decoder. This means, that the control signal for the shift-multiplexor is set to 0 for all instructions except the LUI-Instruction, which needs a shift of 16. Therefore, the signal is set to 1. A more complex decision has to be made for the program counter multiplexor. All instructions that influence the program's control flow (Branch- and Jump-Instructions) produce an output of 1, all other instructions return 0 and the program counter is incremented by 4. The multiplexor for the destination register is always set to 2 to use the I-type-instruction destination field. This is again ignored, when a J-type-instruction reaches the further stages, so no damage is done.

2.3.2 Decoder for the execution-stage

The decoder for the execution stage has to set the signals that select the operands of the operation to be executed and set the kind of operation the ALU has to perform. Since the instruction's opcode is in hardly any way connected to the executed operation, the decoder logic is a little more difficult. It groups all operations of the example code that perform an addition of a register's value to the immediate value of the instruction (ADDIU, LW, SW, SB, LBU) and sets the control signals to 2 for the ALU's a-operand, 1 to forward the immediate field to the ALU and the ALU's operation code itself is set to 20 for addition. For the LUI-instruction, the ALU's a-operand-multiplexor is set to 0 for a shift of 16, the b-operand-multiplexor is set to 1 for the immediate-field and the ALU itself is perfoming the shift operation. All other immediate-operations get the a-operand-multiplexor set to 2, b set to 1 and the ALU operation code set according to the instruction. The only supported R-type operation, SLT gets a set to 2, b set to 0 and the ALU set for set-less-than-operations. All other R-Type instructions are treated like NOOP-instructions:

A set to 2, b set to 0 and the ALU-operation set to a left-shift. Although the datapath would be able to support more operations, the datapath is limited to the described instructions and has to be expanded for a full MIPS instruction set.

2.3.3 Decoder for the memory stage

The memory stage's decoder is more simple than the execution stage's decoder. It just has to evaluate whether the instruction is a store-instruction, a load-instruction or any other instruction. In case of a load instruction the multiplexor signal for the memory access has to be set to 1 to let the result of the memory access get to the writeback stage. In all other cases, this signal is set to 0 to just forward the signal coming from the execution stage. The read or write masks are set according to the instruction stored in the shift register stage. A SW-instruction for example produces an output of F for the write mask and an LBU-instruction returns a read mask of 1. All other instructions have read and write masks of 0.

2.3.4 Decoder for the writeback stage

The decoder for the writeback stage is the simplest of the whole controller because it controls just one signal that enables the register bank. It distinguishes between the commands that write back to the register bank bank (currently supported: LUI, ADDIU, LW, LBU, SLTI, SLT, ANDI, ORI) and sets the *enable_ regs*-signal to for them, and the other instructions, where the register bank is disabled by setting the signal to zero.

3 Evaluation

The CPU evaluation is done with Modelsim from MentorGraphics. The simulations provide a timed analysis of the code. It provides information about the timing relations and allows for bug identification still in a simulation environment, without the need of a complete synthesis and programming of the FPGA. This is a powerful tool to speed up the development process evaluating the design in an early stage.

Individual test-benches test each separate CPU components on all hierarchical levels up to the complete CPU. All component's simulation passed, including the complete CPU with a simulated perfect memory. Furthermore the simulation with a simulated real memory passed. The tests prove the complete implementation up to real conflict cases of instruction and data access stalls.

For the implementation on the FPGA a hdllab code was prepared with the CPU, memory, UART and pll components as well as LEDs, clock and reset interface with the FPGA already integrated, as shown in Figure 3.1. The CPU passed this simulation with the counter program, outputting the counter value to the LEDs output.

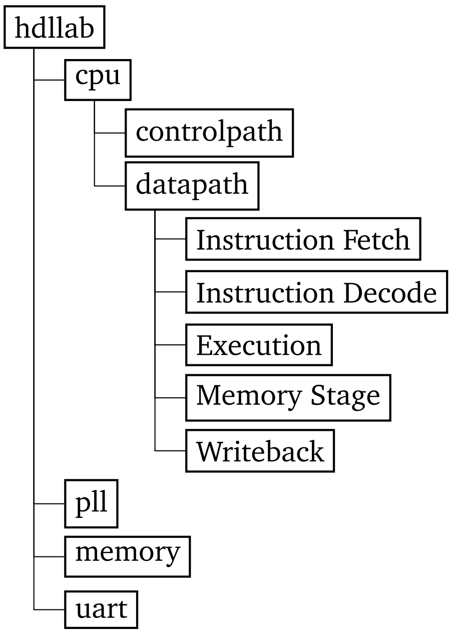

Figure 3.1.: CPU component structure

The evaluation phase was successful, proving exhaustively the correct behavior of each component separately and as a piece of the whole CPU. The functional test of a counter program serve also as a prove of concept.

4 Synthesis

The design was synthesized using the Xilinx ISE suite and the *hdllab* configuration. The target device for synthesis is a Xilinx Virtex-5 FPGA. Since MIPS is a rather compact architecture, the target FPGA provides a lot of programmable resources and a clock frequency of 50 MHz is a mandatory requirement, we opted for the design strategy that promised the highest maximum clock frequency and did not care much about the used space. However a comparison to the results of the balanced synthesis strategy showed no significant differences.
The synthesis result can be operated with a clock frequency up to 73.27 MHz and thus with a minimal clock period of 13.65 ns, so the frequency requirements to the project can be fulfilled. The maximum delay is caused by the instruction decode stage, which is the most complex part of the processor and it is not surprising that it is limiting the maximum frequency of the whole design. The minimum input arrival time of the design is 9.82 ns, the maximum output time required after clock cycle is 3.27 ns and the maximum combinatorial path delay is 1.15 ns.
Although the resource usage was not an optimization goal, the synthesis result is rather compact. It uses 3579 slice registers and 4090 slice look-up-tables, which is 7% of the FPGA's total available slice registers and 9% of the look-up-tables. Combined, 7281 logic slices are used by the design. 388 of these slices are fully used, 3702 of them contain an unused flip-flop and 3191 contain an unused look-up-table. This means that 95% of the used slices are not fully utilized. An area-optimizing synthesis strategy could improve this rather bad utilization percentage values but the design still takes only 16 % of the available logic slices on the FPGA so there is no urgent reason to shrink the size of the result at the cost of making it slower. In fact, our design could surely be used on a more low-end FPGA at about the same speed. On the other hand the free space on the Virtex-5 FPGA could easily be used to build a MIPS-based System-on-chip on top of our project. Of course for full MIPS compatibility the control path and the ALU have to be expanded, but there is plenty of space for it.
13 IO ports of the FPGA are used. That makes eight for the LEDs, two for UART, two more for the clock and one for the reset. Of the available 640 IOBs, the 13 used ones make only 2%.
The synthesis result was tested with the original counter assembler code, waiting 16 clock cycles before the memory cell is incremented by one. The UART-interface was compiled with the frequency parameter configured for 50 MHz to meet the actual board's clock frequency and the *hdllab*-project uses the given memory for FPGAs, not the memory used for simulation. The LEDs flashed as they were supposed to be but 16 clock cycles are to fast for a human eye to detect, so we used an attached oscilloscope to test for the correct blinking sequence. The counter version that should make the LEDs blink perceivably for the human eye however did not work. We suppose that this modified counter contains MIPS-instructions that are somehow not supported by our implementation. Although we searched for new instructions and added code to the controller (the datapath should be able to process any MIPS-instruction and needs no further modification) to process them, the example did not make the LEDs blink. But since the requirement to run the original counter was fulfilled and the time was pressing, no further efforts were made to make this functionality possible.

5 Conclusion

In this laboratory a 32-bit MIPS I CPU with restricted instruction set is successfully designed in vhdl, implemented, synthesized and test a MIPS-I specified processor core on FPGA. The required components of ALU, data-path and control-path are present. The implementation on the FPGA passed a functional test with the counter sample program, running at the desired speed of 50 MHz and blinking the LEDs with the 8-bit counter value.

The CPU design comprises a control-path and a data-path with five pipeline stages. Each component was designed and simulated separately and together up to as a complete CPU. The simulations in Modelsim included test cases for a perfect memory and a real one with instruction and data stalls. The simulation of a functional test with the program counter passed outputting the counter value as an 8-bit LED array.

The synthesis is done with Xilinx ISE for the FPGA Virtex5. With configurations for the fastest clock, the synthesis reports a maximum running frequency of over 70 MHz. The clock configuration for the on board test is set at 50 MHz. The synthesized code passes the functional test with the counter program on the Virtex5. This counter uses the planned instruction set.

This work shows an implementation of a restricted MIPS instruction set. The expansion of this instructions is expected to lead to a full MIPS 32-bit compliant microcontroller.

6 Bibliography

[1] MIPS Technologies Inc. *MIPS32TM Architecture For Programmers Volume I: Introduction to the MIPS32TM Architecture*. 2001.

[2] Jason W. Bacon. Computer science 315 lecture notes, 2011.

A Code

```
-- revision history:
-- 06.07.2015     Alex Schoenberger      created
-- 04.08.2015     Patrick Appenheimer   added entity and architecture behave
-- 05.08.2015     Patrick Appenheimer   bugfixes
-- 05.08.2015     Patrick Appenheimer   subu, and, or added
-- 10.08.2015     Patrick Appenheimer   shift, slt added

library IEEE;
  use IEEE.std_logic_1164.ALL;
  USE IEEE.numeric_std.ALL;

library WORK;
  use WORK.all;

---- ALU FUNCTION CODES: ----
-- ADD ==> 10_0000
-- SUB ==> 10_0010
-- AND ==> 10_0100
-- OR ==> 10_0101
-- result <= reg_a ==> 00_0001
-- result <= reg_b ==> 00_0010
-- shift ==> 00_0100
-- slt ==> 00_1000

entity alu is

    port(
      in_a                      : in   std_logic_vector(31 downto 0);
      in_b                      : in   std_logic_vector(31 downto 0);
      function_code             : in  std_logic_vector(5 downto 0);
      result                    : out std_logic_vector(31 downto 0);
      zero                      : out std_logic_vector(0 downto 0)
    );

end entity alu;

architecture behave of alu is
begin
  process(in_a, in_b, function_code)
  -- declaration of variables
  variable a_uns : std_logic_vector(31 downto 0);
  variable b_uns : std_logic_vector(31 downto 0);
  variable r_uns : std_logic_vector(31 downto 0);
  variable z_uns : std_logic_vector(0 downto 0);
  begin
    -- initialize values
    a_uns := in_a;
    b_uns := in_b;
    r_uns := (others => '0');
    z_uns := b"0";
    -- select operation
    case function_code is
    -- add
    when b"10_0000" =>
    r_uns := std_logic_vector(unsigned(a_uns) + unsigned(b_uns));
    -- sub
    when b"10_0010" =>
    r_uns := std_logic_vector(unsigned(a_uns) - unsigned(b_uns));
    -- and
    when b"10_0100" =>
    r_uns := in_a and in_b;--std_logic_vector(unsigned(a_uns) AND unsigned(b_uns));
    -- or
    when b"10_0101" =>
    r_uns := std_logic_vector(unsigned(a_uns) OR unsigned(b_uns));
    -- out <= a
    when b"00_0001" =>
    r_uns := a_uns;
     -- out <= b
    when b"00_0010" =>
    r_uns := b_uns;
    -- shift
    when b"00_0100" =>
    r_uns := std_logic_vector(SHIFT_LEFT(unsigned(b_uns),to_integer(unsigned(a_uns))));
    -- slt
    when b"00_1000" =>
```

```vhdl
    if (to_integer(signed(a_uns)) < to_integer(signed(b_uns))) then
      r_uns := x"0000_0001";
    end if;
    -- others
    when others => r_uns := (others => 'X');
    end case;
    if r_uns = x"0000_0000" then
    z_uns := b"1";
    else
    z_uns := b"0";
    end if;
    -- assign variables to output signals
    result <= r_uns;
    zero <= std_logic_vector(unsigned(z_uns));
end process;
end architecture behave;
```

../src/vhdl/alu.vhd

```vhdl
-- Revision history:
-- 2015-08-12        Lukas Jaeger        created

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library WORK;
use WORK.all;

entity control_pipeline is
    port(
      clk                    : in  std_logic;
      rst                    : in  std_logic;
      rd_mask             : out std_logic_vector(3  downto 0);
      wr_mask            : out std_logic_vector(3  downto 0);
      instr_stall         : in  std_logic;
      data_stall          : in  std_logic;
      instr_in               : in  std_logic_vector(31 downto 0);
      alu_op              : out std_logic_vector(5 downto 0);
      exc_mux1              : out std_logic_vector(1 downto 0);
      exc_mux2              : out std_logic_vector(1 downto 0);
      exc_alu_zero     : in  std_logic_vector(0 downto 0);
      memstg_mux          : out std_logic;
      id_regdest_mux      : out std_logic_vector (1 downto 0);
      id_regshift_mux        : out std_logic_vector (1 downto 0);
      id_enable_regs     : out std_logic;
      in_mux_pc              : out std_logic;
      stage_control    : out std_logic_vector (4 downto 0)
    );
end entity control_pipeline;

architecture behavioural of control_pipeline is
    signal instr_1, instr_2, instr_3, instr_4: std_logic_vector (31 downto 0);

    begin

    pipeline: process(clk, rst) is
    begin
            if (rst = '1') then
                    instr_1 <= x"00000000";
                    instr_2 <= x"00000000";
                    instr_3 <= x"00000000";
                    instr_4 <= x"00000000";
                    stage_control <= "11111";
            elsif (rising_edge(clk)) then
                    instr_1 <= instr_in;
                    instr_2 <= instr_1;
                    instr_3 <= instr_2;
                    instr_4 <= instr_3;
            end if;
    end process;

    id: process (instr_1) is
    begin
            if (instr_1(31 downto 26) = "000000") then -- R-type instructions
                    id_regdest_mux <= "00";
                    id_regshift_mux <= "00";
                    if (instr_1(20 downto 0) = "000000000000000001000") then -- JR-instruction
                            in_mux_pc <= '1';
                    else
                            in_mux_pc <= '0';
                    end if;
            else    -- I-Type- and J-Type instructions. They can go together, because nobody cares about
                    -- the alu-result of a J-Type, so it does not matter, which value is yielded to ex
                    id_regdest_mux <= "10";
                    if (instr_1(31 downto 26) = "001111") then  -- LUI needs a shift
                        id_regshift_mux <= "01";
                        in_mux_pc <= '0';
                    elsif ((instr_1(31 downto 26) = "000010") -- J
                        or (instr_1 (31 downto 26) = "000011") -- JAL
                        or (instr_1 (31 downto 26) = "011101") -- JALX
                        or (instr_1 (31 downto 26) = "000100") -- BEQ
                        or (instr_1 (31 downto 26) = "000001") -- BGEZ
                        or (instr_1 (31 downto 26) = "000111") -- BGTZ
                        or (instr_1 (31 downto 26) = "000110") -- BLEZ
                        or (instr_1 (31 downto 26) = "000101") -- BEQZ
                        ) then
                        id_regshift_mux <= "00";
                        in_mux_pc <= '1';
                    else
                        id_regshift_mux <= "00";
                        in_mux_pc <= '0';
                    end if;
            end if;

    end process;

    ex: process (instr_2) is
    begin
            if (instr_2 (31 downto 26) = "001111") then --LUI
                    exc_mux1 <= "00";
                    exc_mux2 <= "01";
                    alu_op <="000100";
            elsif ((instr_2 (31 downto 26) = "001001") --ADDIU
                or (instr_2 (31 downto 26) = "100011") --LW
                or (instr_2 (31 downto 26) = "101011") --SW
                or (instr_2 (31 downto 26) = "101000") --SB
                or (instr_2 (31 downto 26) = "100100") --LBU
                )then
                    exc_mux1 <="10";
                    exc_mux2 <="01";
                    alu_op <="100000";
            elsif (instr_2 (31 downto 26) = "001010") then --SLTI
                    exc_mux1 <="10";
                    exc_mux2 <="01";
                    alu_op <="001000";
            elsif (instr_2 (31 downto 26) = "001100") then --ANDI
                    exc_mux1 <="10";
                    exc_mux2 <="01";
                    alu_op <="100100";
            else --if (instr_2 (31 downto 26) = "000000") then -- NOP and other R-types and Ops, where the result does not matter
                    exc_mux1 <= "10";
                    exc_mux2 <= "00";
                    alu_op <= "000100";
            end if;
    end process;

    mem: process (instr_3) is
    begin
            if (instr_3 (31 downto 26) = "100011") then --LW
                    memstg_mux <= '1';
                    rd_mask <= "1111";
                    wr_mask <= "0000";
            elsif (instr_3 (31 downto 26) = "100100") then --LBU
                    memstg_mux <= '1';
                    rd_mask <= "0001";
                    wr_mask <= "0000";
            elsif (instr_3 (31 downto 26) = "101011") then --SW
                    memstg_mux <= '0';
                    rd_mask <= "0000";
                    wr_mask <= "1111";
            elsif (instr_3 (31 downto 26) = "101000") then --SB
                    memstg_mux <= '0';
                    rd_mask <= "0000";
                    wr_mask <= "0001";
            else
                    memstg_mux <= '0';
                    rd_mask <= "0000";
                    wr_mask <= "0000";
            end if;
    end process;

    wb: process (instr_4) is
    begin
            if ((instr_4 (31 downto 26) = "001111") or --LUI
            (instr_4 (31 downto 26) = "001001") or --ADDIU
            (instr_4 (31 downto 26) = "100011") or --LW
            (instr_4 (31 downto 26) = "100100") or --LBU
            (instr_4 (31 downto 26) = "001010") or --SLTI
            (instr_4 (31 downto 26) = "001100") --ANDI
            ) then
```

```
                    id_enable_regs <= '1';
                else
                    id_enable_regs <= '0';
                end if;
        end process;

        stall: process (data_stall, instr_stall) is
        begin
                if (data_stall = '1' or instr_stall = '1') then
                    stage_control <= "00000";
                else
                    stage_control <= "11111";
                end if;
        end process;

end architecture;
```

../src/vhdl/control_pipeline.vhd

```
-- revision history:
-- 06.07.2015      Alex Schoenberger      created
-- 10.08.2015      Patrick Appenheimer    entity
-- 11.08.2015      Patrick Appenheimer    main fsm
-- 11.08.2015      Patrick Appenheimer    5 instr fsm
-- 12.08.2015      Patrick Appenheimer    minor changes
-- 14.08.2015    Patrick Appenheimer    stall logic changed

library IEEE;
  use IEEE.std_logic_1164.ALL;
  USE IEEE.numeric_std.ALL;

library WORK;
  use WORK.all;

entity cpu_control is
  port(
      clk                    : in  std_logic;
      rst                    : in  std_logic;
      rd_mask            : out std_logic_vector(3  downto 0);
      wr_mask          : out std_logic_vector(3  downto 0);
      instr_stall        : in  std_logic;
      data_stall         : in  std_logic;
      instr_in               : in  std_logic_vector(31 downto 0);
      alu_op             : out std_logic_vector(5 downto 0);
      exc_mux1               : out std_logic_vector(1 downto 0);
      exc_mux2               : out std_logic_vector(1 downto 0);
      exc_alu_zero     : in  std_logic_vector(0 downto 0);
      memstg_mux         : out std_logic;
      id_regdest_mux     : out std_logic_vector (1 downto 0);
      id_regshift_mux        : out std_logic_vector (1 downto 0);
      id_enable_regs   : out std_logic;
      in_mux_pc              : out std_logic;
      stage_control    : out std_logic_vector (4 downto 0)
    );

end entity cpu_control;

architecture structure_cpu_control of cpu_control is

  constant s0    : std_logic_vector(4 downto 0) := b"00000";
  constant s1    : std_logic_vector(4 downto 0) := b"00001";
  constant s2    : std_logic_vector(4 downto 0) := b"00010";
  constant s3    : std_logic_vector(4 downto 0) := b"00011";
  constant s4    : std_logic_vector(4 downto 0) := b"00100";
  constant sX    : std_logic_vector(4 downto 0) := b"11111";

  signal instr1         : std_logic_vector(31 downto 0);
  signal instr2         : std_logic_vector(31 downto 0);
  signal instr3         : std_logic_vector(31 downto 0);
  signal instr4         : std_logic_vector(31 downto 0);
  signal instr5         : std_logic_vector(31 downto 0);

  signal currentstate1         : std_logic_vector(4 downto 0);
  signal currentstate2         : std_logic_vector(4 downto 0);
  signal currentstate3         : std_logic_vector(4 downto 0);
  signal currentstate4         : std_logic_vector(4 downto 0);
  signal currentstate5         : std_logic_vector(4 downto 0);

  signal nextstate1         : std_logic_vector(4 downto 0);
  signal nextstate2         : std_logic_vector(4 downto 0);
  signal nextstate3         : std_logic_vector(4 downto 0);
  signal nextstate4         : std_logic_vector(4 downto 0);
  signal nextstate5         : std_logic_vector(4 downto 0);

  signal output_buffer1 : std_logic_vector(29 downto 0);
  signal output_buffer2 : std_logic_vector(29 downto 0);
  signal output_buffer3 : std_logic_vector(29 downto 0);
  signal output_buffer4 : std_logic_vector(29 downto 0);
  signal output_buffer5 : std_logic_vector(29 downto 0);

  signal busy1      : std_logic;
  signal busy2      : std_logic;
  signal busy3      : std_logic;
  signal busy4      : std_logic;
  signal busy5      : std_logic;

  signal go1      : std_logic;
  signal go2      : std_logic;
  signal go3      : std_logic;
  signal go4      : std_logic;
  signal go5      : std_logic;

  signal currentstate  : std_logic_vector(4 downto 0);
  signal nextstate     : std_logic_vector(4 downto 0);

  signal stall     : std_logic := '0';

begin

  fsm1: entity work.fsm(behavioral) port map(clk, rst, instr1, stall,
  currentstate1, nextstate1, output_buffer1, busy1, go1);
  fsm2: entity work.fsm(behavioral) port map(clk, rst, instr2, stall,
  currentstate2, nextstate2, output_buffer2, busy2, go2);
  fsm3: entity work.fsm(behavioral) port map(clk, rst, instr3, stall,
  currentstate3, nextstate3, output_buffer3, busy3, go3);
  fsm4: entity work.fsm(behavioral) port map(clk, rst, instr4, stall,
  currentstate4, nextstate4, output_buffer4, busy4, go4);
  fsm5: entity work.fsm(behavioral) port map(clk, rst, instr5, stall,
  currentstate5, nextstate5, output_buffer5, busy5, go5);

  --stage_control (1 downto 0) <= b"11";

stall_ctrl: process(instr_stall, data_stall)
begin
  if ((instr_stall = '0') and (data_stall = '0')) then
    stall <= '0';
  else
    stall <= '1';
  end if;
end process;

state_encode: process(currentstate, busy1, busy2, busy3, busy4, busy5)
  begin
    case currentstate is
      when sX =>
      if (busy1 = '0') then
          nextstate <= s0;
      else
          nextstate <= sX;
      end if;
        when s0 =>
        if (busy2 = '0') then
            nextstate <= s1;
        else
            nextstate <= s0;
        end if;
        when s1 =>
        if (busy3 = '0') then
            nextstate <= s2;
        else
            nextstate <= s1;
        end if;
        when s2 =>
        if (busy4 = '0') then
            nextstate <= s3;
        else
            nextstate <= s2;
        end if;
        when s3 =>
        if (busy5 = '0') then
            nextstate <= s4;
        else
            nextstate <= s3;
        end if;
        when s4 =>
        if (busy1 = '0') then
            nextstate <= s0;
        else
            nextstate <= sX;
        end if;
        when others => nextstate <= sX;
```

```vhdl
    end case;
  end process state_encode;

  state_register: process(rst, clk)
  begin
    if (rst = '1') then
      currentstate <= sX;
    elsif (clk'event and clk = '1') then
      currentstate <= nextstate;
    end if;
  end process state_register;

  state_decode: process(currentstate)
  begin
    case currentstate is
      when sX =>
        go1 <= '0';
        go2 <= '0';
        go3 <= '0';
        go4 <= '0';
        go5 <= '0';
      when s0 =>
        instr1 <= instr_in;
        go1 <= '1';
      when s1 =>
        instr2 <= instr_in;
        go2 <= '1';
      when s2 =>
        instr3 <= instr_in;
        go3 <= '1';
      when s3 =>
        instr4 <= instr_in;
        go4 <= '1';
      when s4 =>
        instr5 <= instr_in;
        go5 <= '1';
      when others =>
        -- do something
    end case;
  end process state_decode;

  fsm_ctrl: process(currentstate1, currentstate2, currentstate3, currentstate4, currentstate5)
  begin
    stage_control <= b"11111";
    case currentstate1 is
    when s0 =>
      id_regdest_mux <= output_buffer1 (28 downto 27);
      id_regshift_mux <= output_buffer1 (26 downto 25);


    when s1 =>
      exc_mux1 <= output_buffer1 (23 downto 22);
      exc_mux2 <= output_buffer1 (21 downto 20);
      alu_op <= output_buffer1 (19 downto 14);

      stage_control (2) <= output_buffer1 (2);
    when s2 =>
      memstg_mux <= output_buffer1 (13);
      rd_mask <= output_buffer1 (12 downto 9);
      wr_mask <= output_buffer1 (8 downto 5);
      stage_control (3) <= output_buffer1 (3);
    when sX =>
      --in_mux_pc <= output_buffer1 (29);
    when s3 =>
      id_enable_regs <= output_buffer1 (24);
    when others =>
      --do nothing
  end case;

  case currentstate2 is
    when s0 =>
      id_regdest_mux <= output_buffer2 (28 downto 27);
      id_regshift_mux <= output_buffer2 (26 downto 25);

    when s1 =>
      exc_mux1 <= output_buffer2 (23 downto 22);
      exc_mux2 <= output_buffer2 (21 downto 20);
      alu_op <= output_buffer2 (19 downto 14);

      stage_control (2) <= output_buffer2 (2);
    when s2 =>
      memstg_mux <= output_buffer2(13);
      rd_mask <= output_buffer2 (12 downto 9);
      wr_mask <= output_buffer2 (8 downto 5);
      stage_control (3) <= output_buffer2 (3);
    when sX =>
      --in_mux_pc <= output_buffer2 (29);
    when s3 =>
      id_enable_regs <= output_buffer2 (24);
    when others =>
      --do nothing
  end case;

  case currentstate3 is
    when s0 =>
      id_regdest_mux <= output_buffer3 (28 downto 27);
      id_regshift_mux <= output_buffer3 (26 downto 25);

    when s1 =>
      exc_mux1 <= output_buffer3 (23 downto 22);
      exc_mux2 <= output_buffer3 (21 downto 20);
      alu_op <= output_buffer3 (19 downto 14);

      stage_control (2) <= output_buffer3 (2);
    when s2 =>
      memstg_mux <= output_buffer3 (13);
      rd_mask <= output_buffer3 (12 downto 9);
      wr_mask <= output_buffer3 (8 downto 5);
      stage_control (3) <= output_buffer3 (3);
    when sX =>
      --in_mux_pc <= output_buffer3 (29);
    when s3 =>
      id_enable_regs <= output_buffer3 (24);
    when others =>
      --do nothing
  end case;

  case currentstate4 is
    when s0 =>
      id_regdest_mux <= output_buffer4 (28 downto 27);
      id_regshift_mux <= output_buffer4 (26 downto 25);

    when s1 =>
      exc_mux1 <= output_buffer4 (23 downto 22);
      exc_mux2 <= output_buffer4 (21 downto 20);
      alu_op <= output_buffer4 (19 downto 14);

      stage_control (2) <= output_buffer4 (2);
    when s2 =>
      memstg_mux <= output_buffer4 (13);
      rd_mask <= output_buffer4 (12 downto 9);
      wr_mask <= output_buffer4 (8 downto 5);
      stage_control (3) <= output_buffer4 (3);
    when sX =>
      --in_mux_pc <= output_buffer4 (29);
    when s3 =>
      id_enable_regs <= output_buffer4 (24);
    when others =>
      --do nothing
  end case;

  case currentstate5 is
    when s0 =>
      id_regdest_mux <= output_buffer5 (28 downto 27);
      id_regshift_mux <= output_buffer5 (26 downto 25);

    when s1 =>
      exc_mux1 <= output_buffer5 (23 downto 22);
      exc_mux2 <= output_buffer5 (21 downto 20);
      alu_op <= output_buffer5 (19 downto 14);

      stage_control (2) <= output_buffer5 (2);
    when s2 =>
      memstg_mux <= output_buffer5 (13);
      rd_mask <= output_buffer5 (12 downto 9);
      wr_mask <= output_buffer5 (8 downto 5);
      stage_control (3) <= output_buffer5 (3);
    when sX =>
      --in_mux_pc <= output_buffer5 (29);
    when s3 =>
      id_enable_regs <= output_buffer5 (24);
    when others =>
      --do nothing
  end case;

  end process fsm_ctrl;

  mux_pc_ctrl: process(clk, output_buffer1, output_buffer2, output_buffer3, output_buffer4, output_buffer5)
  begin
  if (clk'event and clk = '1') then
    if (currentstate1 = sX) then
      in_mux_pc <= output_buffer1 (29);
    elsif (currentstate2 = sX) then
      in_mux_pc <= output_buffer2 (29);
    elsif (currentstate3 = sX) then
      in_mux_pc <= output_buffer3 (29);
    elsif (currentstate4 = sX) then
      in_mux_pc <= output_buffer4 (29);
    elsif (currentstate5 = sX) then
      in_mux_pc <= output_buffer5 (29);
    else
      in_mux_pc <= '0';
    end if;
  end if;
  end process;

end architecture structure_cpu_control;
```

../src/vhdl/cpu_control_fsm.vhd

```
-- Revision history:
-- 2015-08-12        Lukas Jaeger        created
-- 2015-08-16     Lukas Jaeger  fixed all bugs and made it working with the cpu

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library WORK;
use WORK.all;

entity cpu_control is
    port(
      clk                        : in  std_logic;
      rst                        : in  std_logic;
      rd_mask               : out std_logic_vector(3  downto 0);
      wr_mask             : out std_logic_vector(3  downto 0);
      instr_stall          : in  std_logic;
      data_stall           : in  std_logic;
      instr_in                   : in  std_logic_vector(31 downto 0);
      alu_op                : out std_logic_vector(5 downto 0);
      exc_mux1                   : out std_logic_vector(1 downto 0);
      exc_mux2                   : out std_logic_vector(1 downto 0);
      exc_alu_zero     : in  std_logic_vector(0 downto 0);
      memstg_mux            : out std_logic;
      id_regdest_mux       : out std_logic_vector (1 downto 0);
      id_regshift_mux            : out std_logic_vector (1 downto 0);
      id_enable_regs     : out std_logic;
      in_mux_pc                  : out std_logic;
      stage_control    : out std_logic_vector (4 downto 0)
    );
end entity cpu_control;

architecture structure_cpu_control of cpu_control is
    signal instr_1, instr_2, instr_3, instr_4: std_logic_vector (31 downto 0);

    begin

    pipeline: process(clk, rst) is
    begin
            if (rst = '1') then
                    instr_1 <= x"00000000";
                    instr_2 <= x"00000000";
                    instr_3 <= x"00000000";
                    instr_4 <= x"00000000";
            elsif (rising_edge(clk) and instr_stall /= '1' and data_stall /= '1') then
                    instr_1 <= instr_in;
                    instr_2 <= instr_1;
                    instr_3 <= instr_2;
                    instr_4 <= instr_3;
            end if;
    end process;

    id: process (instr_1) is
    begin
            if (instr_1(31 downto 26) = "000000") then -- R-type instructions
                    id_regdest_mux <= "00";
                    id_regshift_mux <= "00";
                    if (instr_1(20 downto 0) = "000000000000000001000") then -- JR-instruction
                            in_mux_pc <= '1';
                    else
                            in_mux_pc <= '0';
                    end if;
            else    -- I-Type- and J-Type instructions. They can go together, because nobody cares about
                    -- the alu-result of a J-Type, so it does not matter, which value is yielded to ex
                    id_regdest_mux <= "10";
                    if (instr_1(31 downto 26) = "001111") then  -- LUI needs a shift
                    id_regshift_mux <= "01";
                elsif ((instr_1(31 downto 26) = "000010") -- J
                    or (instr_1 (31 downto 26) = "000011") -- JAL
                    or (instr_1 (31 downto 26) = "011101") -- JALX
                    or (instr_1 (31 downto 26) = "000100") -- BEQ
                    or (instr_1 (31 downto 26) = "000001") -- BGEZ
                    or (instr_1 (31 downto 26) = "000111") -- BGTZ
                    or (instr_1 (31 downto 26) = "000110") -- BLEZ
                    or (instr_1 (31 downto 26) = "000101") -- BEQZ
                    ) then
                    id_regshift_mux <= "00";
                    in_mux_pc <= '1';
                    else
                    id_regshift_mux <= "00";
                    in_mux_pc <= '0';
                end if;
            end if;

    end process;

    ex: process (instr_2) is
    begin
            if (instr_2 (31 downto 26) = "001111") then --LUI
                    exc_mux1 <= "00";
                    exc_mux2 <= "01";
                    alu_op <="000100";
            elsif ((instr_2 (31 downto 26) = "001001") --ADDIU
                or (instr_2 (31 downto 26) = "100011") --LW
                or (instr_2 (31 downto 26) = "101011") --SW
                or (instr_2 (31 downto 26) = "101000") --SB
                or (instr_2 (31 downto 26) = "100100") --LBU
                )then
                    exc_mux1 <="10";
                    exc_mux2 <="01";
                    alu_op <="100000";
            elsif (instr_2 (31 downto 26) = "001010") then --SLTI
                    exc_mux1 <="10";
                    exc_mux2 <="01";
                    alu_op <="001000";
            elsif (instr_2 (31 downto 26) = "001100") then --ANDI
                    exc_mux1 <="10";
                    exc_mux2 <="01";
                    alu_op <="100100";
        elsif (instr_2 (31 downto 26) = "001101") then --ORI
                    exc_mux1 <="10";
                    exc_mux2 <="01";
                    alu_op <="100101";
  elsif ((instr_2 (31 downto 26) = "000000") and (instr_2(10 downto 0) = "00000101010")) then
        exc_mux1 <= "10";
        exc_mux2 <= "00";
        alu_op <= "001000";
            else --if (instr_2 (31 downto 26) = "000000") then -- NOP and other R-types and Ops, where the result does not matter
                    exc_mux1 <= "10";
                    exc_mux2 <= "00";
                    alu_op <= "000100";
            end if;
    end process;

    mem: process (instr_3) is
    begin
            if (instr_3 (31 downto 26) = "100011") then --LW
                    memstg_mux <= '1';
                    rd_mask <= "1111";
                    wr_mask <= "0000";
            elsif (instr_3 (31 downto 26) = "100100") then --LBU
                    memstg_mux <= '1';
                    rd_mask <= "0001";
                    wr_mask <= "0000";
            elsif (instr_3 (31 downto 26) = "101011") then --SW
                    memstg_mux <= '0';
                    rd_mask <= "0000";
                    wr_mask <= "1111";
            elsif (instr_3 (31 downto 26) = "101000") then --SB
                    memstg_mux <= '0';
                    rd_mask <= "0000";
                    wr_mask <= "0001";
            else
                    memstg_mux <= '0';
                    rd_mask <= "0000";
                    wr_mask <= "0000";
            end if;
    end process;

    wb: process (instr_4) is
    begin
            if ((instr_4 (31 downto 26) = "001111") or --LUI
            (instr_4 (31 downto 26) = "001001") or --ADDIU
            (instr_4 (31 downto 26) = "100011") or --LW
            (instr_4 (31 downto 26) = "100100") or --LBU
            (instr_4 (31 downto 26) = "001010") or --SLTI
            (instr_4 (31 downto 26) = "001100") or --ANDI
            (instr_4 (31 downto 26) = "001101") or --ORI
            (instr_4 (31 downto 26) = "000000") and (instr_4(10 downto 0) = "00000101010")) --SLT
            ) then
                id_enable_regs <= '1';
            else
                id_enable_regs <= '0';
            end if;
    end process;

    stall: process (data_stall, instr_stall) is
    begin
            if (data_stall = '1' or instr_stall = '1') then
                stage_control <= "00000";
            else
                stage_control <= "11111";
            end if;
    end process;

end architecture;
```

../src/vhdl/cpu_control.vhd

```
-- revision history:
-- 06.07.2015     Alex Schoenberger    created
-- 05.08.2015     Patrick Appenheimer    first try
-- 06.08.2015     Patrick Appenheimer    ports and entities added
-- 10.08.2015     Patrick Appenheimer    minor changes
-- 12.08.2015     Patrick Appenheimer    changed rising_edge to falling_edge
-- 14.08.2015     Patrick Appenheimer    changed pc_mux control

library IEEE;
  use IEEE.std_logic_1164.ALL;
  USE IEEE.numeric_std.ALL;

library WORK;
  use WORK.all;

  -- -- stage_control: --
  -- to activate registers, set signal stage_control as follows:
  -- stage0->stage1: xxxx1
  -- stage1->stage2: xxx1x
  -- stage2->stage3: xx1xx
  -- stage3->stage4: x1xxx
  -- stage4->stage5: 1xxxx

entity cpu_datapath is
  port(
      clk                       : in  std_logic;
      rst                       : in  std_logic;
      instr_addr                : out std_logic_vector(31 downto 0);
      data_addr                 : out std_logic_vector(31 downto 0);
      instr_in                  : in  std_logic_vector(31 downto 0);
      data_to_cpu               : in  std_logic_vector(31 downto 0);
      data_from_cpu             : out std_logic_vector(31 downto 0);
      alu_op                : in  std_logic_vector(5 downto 0);
      exc_mux1                  : in  std_logic_vector(1 downto 0);
      exc_mux2                  : in  std_logic_vector(1 downto 0);
      exc_alu_zero     : out std_logic_vector(0 downto 0);
      memstg_mux            : in  std_logic;
      id_regdest_mux        : in std_logic_vector (1 downto 0);
      id_regshift_mux           : in std_logic_vector (1 downto 0);
      id_enable_regs     : in std_logic;
      in_mux_pc                 : in std_logic;
      stage_control    : in std_logic_vector (4 downto 0)

    );

end entity cpu_datapath;


architecture structure_cpu_datapath of cpu_datapath is

  -- -------- PC ==> Instr. Fetch ----------------
  signal mux_out_0        : std_logic_vector(31 downto 0);

  -- -------- Instr. Fetch ==> Instr. Decode -----------------
  signal instr_1          : std_logic_vector(31 downto 0);
  signal ip_1             : std_logic_vector(31 downto 0);

  -- -------- Instr. Decode ==> Execution ----------------
  signal shift_2          : std_logic_vector(4 downto 0);
  signal reg_a_2          : std_logic_vector(31 downto 0);
  signal reg_b_2          : std_logic_vector(31 downto 0);
  signal regdest_2        : std_logic_vector(4 downto 0);
  signal imm_2            : std_logic_vector(31 downto 0);
  signal ip_2             : std_logic_vector(31 downto 0);

  -- -------- Execution ==> Memory Stage ----------------
  signal alu_result_3     : std_logic_vector(31 downto 0);
  signal data_3           : std_logic_vector(31 downto 0);
  signal regdest_3        : std_logic_vector(4 downto 0);

  -- -------- Memory Stage ==> Write Back ----------------
  signal writeback_4      : std_logic_vector(31 downto 0);
  signal regdest_4        : std_logic_vector(4 downto 0);

-- IP:
  signal mux_pc_out      : std_logic_vector(31 downto 0);

-- Instr. Fetch:
  signal if_ip      : std_logic_vector(31 downto 0);
  signal if_instr   : std_logic_vector(31 downto 0);

-- Instr. Decode:
  signal id_a       : std_logic_vector(31 downto 0);
  signal id_b       : std_logic_vector(31 downto 0);
  signal id_imm     : std_logic_vector(31 downto 0);
  signal id_ip      : std_logic_vector(31 downto 0);
  signal id_regdest : std_logic_vector(4 downto 0);
  signal id_shift   : std_logic_vector(4 downto 0);

-- Execution:
  signal alu_result       : std_logic_vector(31 downto 0);
  signal data_out         : std_logic_vector(31 downto 0);
  signal exc_destreg_out  : std_logic_vector(4  downto 0);

-- Memory Stage:
  signal memstg_writeback_out : std_logic_vector(31 downto 0);
  signal memstg_destreg_out   : std_logic_vector(4  downto 0);

-- Write Back:
  signal wb_writeback_out : std_logic_vector(31 downto 0);
  signal wb_destreg_out   : std_logic_vector(4  downto 0);

  signal last_instruction : std_logic_vector( 31 downto 0);


begin

-- INSTRUCTION FETCH:
  instruction_fetch:    entity work.instruction_fetch(behavioral) port map(clk, rst, mux_out_0, instr_in, if_ip,
                                                                            instr_addr, if_instr);

-- INSTRUCTION DECODE:
  instruction_decode:   entity work.instruction_decode(behavioral) port map(instr_1, ip_1, wb_writeback_out, alu_result,
                                                                            memstg_writeback_out, regdest_4, exc_destreg_out,
                                                                            memstg_destreg_out, id_regdest_mux,
                                                                            id_regshift_mux, clk, rst, id_enable_regs,
                                                                            id_a, id_b, id_imm, id_ip, id_regdest, id_shift);

-- EXECUTION:
  execution:            entity work.execution(behave) port map(clk, rst, alu_result, data_out, exc_destreg_out,
                                                          exc_alu_zero, reg_a_2, reg_b_2, regdest_2, imm_2,
                                                          ip_2, shift_2, exc_mux1, exc_mux2,alu_op);

-- MEMORY STAGE:
  memory_stage:         entity work.MemoryStage(behavioral) port map(clk, rst, alu_result_3, data_3, data_addr,
                                                                data_from_cpu, data_to_cpu, memstg_mux,
                                                                memstg_writeback_out, regdest_3, memstg_destreg_out);

-- WRITE BACK:
  write_back:           entity work.write_back(behavioral) port map(clk, rst, writeback_4, regdest_4,
                                                              wb_writeback_out, wb_destreg_out);


stage0: process(clk, rst)
begin
  if (rst = '1') then
    mux_out_0 <=  (others => '0');
  elsif ((rising_edge(clk)) and (stage_control (0 downto 0) = "1")) then
    mux_out_0 <= mux_pc_out;
  end if;
end process;

stage1: process(clk, rst)
begin
```

```vhdl
  if (rst = '1') then
    instr_1 <= (others => '0');
    ip_1 <= (others => '0');
  elsif ((rising_edge(clk)) and (stage_control (1 downto 1) = "1")) then
    instr_1 <= if_instr;
    ip_1 <= if_ip;
  end if;
end process;

stage2: process(clk, rst)
begin
  if (rst = '1') then
    shift_2 <= (others => '0');
    reg_a_2 <= (others => '0');
    reg_b_2 <= (others => '0');
    regdest_2 <= (others => '0');
    imm_2 <= (others => '0');
    ip_2 <= (others => '0');
  elsif ((rising_edge(clk)) and (stage_control (2 downto 2) = "1")) then
    shift_2 <= id_shift;
    reg_a_2 <= id_a;
    reg_b_2 <= id_b;
    regdest_2 <= id_regdest;
    imm_2 <= id_imm;
    ip_2 <= ip_1;
  end if;
end process;

stage3: process(clk, rst)
begin
  if (rst = '1') then
    alu_result_3 <= (others => '0');
    data_3 <= (others => '0');
    regdest_3 <= (others => '0');
  elsif ((rising_edge(clk)) and (stage_control (3 downto 3) = "1")) then
    alu_result_3 <= alu_result;
    data_3 <= data_out;
    regdest_3 <= exc_destreg_out;
  end if;
end process;

stage4: process(clk, rst)
begin
  if (rst = '1') then
    writeback_4 <= (others => '0');
    regdest_4 <= (others => '0');
  elsif ((rising_edge(clk)) and (stage_control (4 downto 4) = "1")) then
    writeback_4 <= memstg_writeback_out;
    regdest_4 <= memstg_destreg_out;
  end if;
end process;

mux: process(in_mux_pc, id_ip, if_ip)
  begin
  if (in_mux_pc = '1')then
    mux_pc_out <= id_ip;
  else
    mux_pc_out <= if_ip;
  end if;
  end process;

--mux: process(instr_in, id_ip, if_ip, clk)
--  begin
--    if (clk'event and clk = '1') then
--    if ((instr_in(31 downto 26) = "000100") or (instr_in(31 downto 26) = "000010") or (instr_in(31 downto 26) = "000101") or (
    instr_in(31 downto 26) = "011101"))then
--      mux_pc_out <= id_ip;
--    else
--      mux_pc_out <= if_ip;
--    end if;
--  end if;
--  end process;

--last_instruction_proc: process(clk) is
--begin
--        if (clk'event and clk = '1') then
--              last_instruction <= instr_in;
--         end if;
--end process;


end architecture structure_cpu_datapath;
```

../src/vhdl/cpu_datapath.vhd

```
-- revision history:
-- 06.07.2015      Alex Schoenberger     created
-- 07.08.2015      Patrick Appenheimer   cpu_datapath instanciated
-- 10.08.2015      Bahri Enis Demirtel cpu_control added

library IEEE;
  use IEEE.std_logic_1164.ALL;

library WORK;
  use WORK.cpu_pack.all;

entity cpu is
    port(
      clk                    : in  std_logic;
      rst                    : in  std_logic;
      instr_addr             : out std_logic_vector(31 downto 0);
      data_addr              : out std_logic_vector(31 downto 0);
      rd_mask                : out std_logic_vector(3  downto 0);
      wr_mask                : out std_logic_vector(3  downto 0);
      instr_stall            : in  std_logic;
      data_stall             : in  std_logic;
      instr_in               : in  std_logic_vector(31 downto 0);
      data_to_cpu            : in  std_logic_vector(31 downto 0);
      data_from_cpu          : out std_logic_vector(31 downto 0)
    );
end entity cpu;

architecture structure_cpu of cpu is

signal alu_op              : std_logic_vector(5 downto 0);
signal exc_mux1            : std_logic_vector(1 downto 0);
signal exc_mux2            : std_logic_vector(1 downto 0);
signal exc_alu_zero        : std_logic_vector(0 downto 0);
signal memstg_mux          : std_logic;
signal id_regdest_mux      : std_logic_vector (1 downto 0);
signal id_regshift_mux     : std_logic_vector (1 downto 0);
signal id_enable_regs      : std_logic;
signal in_mux_pc           : std_logic;
signal stage_control       : std_logic_vector (4 downto 0);




begin

  -- control logic
   u1_control: entity work.cpu_control(structure_cpu_control)
  PORT MAP(clk, rst, rd_mask, wr_mask, instr_stall, data_stall, instr_in, alu_op, exc_mux1, exc_mux2,
  exc_alu_zero, memstg_mux, id_regdest_mux, id_regshift_mux, id_enable_regs, in_mux_pc, stage_control
  );




  -- datapath
   u2_datapath: entity work.cpu_datapath(structure_cpu_datapath)
     PORT MAP(clk, rst, instr_addr, data_addr, instr_in, data_to_cpu, data_from_cpu, alu_op, exc_mux1, exc_mux2,
              exc_alu_zero, memstg_mux, id_regdest_mux, id_regshift_mux, id_enable_regs, in_mux_pc, stage_control
     );

end architecture structure_cpu;
```

../src/vhdl/cpu.vhd

```
-- revision history:
-- 03.08.2015      Patrick Appenheimer    created
-- 04.08.2015      Patrick Appenheimer    added alu1
-- 04.08.2015      Patrick Appenheimer    mux1, mux2
-- 05.08.2015      Patrick Appenheimer    bugfixes

library IEEE;
  use IEEE.std_logic_1164.ALL;

library WORK;
  use WORK.all;

-- -- MUX Steuereingaenge: -- --
-- mux1:
-- 00: out <= Zero Extend
-- 01: out <= 4
-- 10: out <= Reg A
-- mux2:
-- 00: out <= Reg B
-- 01: out <= Imm
-- 10: out <= IP

-- -- ALU FUNCTION CODES: -- --
-- ADD ==> 10_0000
-- SUB ==> 10_0010
-- AND ==> 10_0100
-- OR ==> 10_0101
-- result <= reg_a ==> 00_0001
-- result <= reg_b ==> 00_0010

entity execution is
    port(
      clk                 : in  std_logic;
      rst                 : in  std_logic;
      ex_alu              : out std_logic_vector(31 downto 0);
      ex_data             : out std_logic_vector(31 downto 0);
      ex_destreg          : out std_logic_vector(4  downto 0);
      ex_alu_zero         : out std_logic_vector(0  downto 0);
      in_a                : in  std_logic_vector(31 downto 0);
      in_b                : in  std_logic_vector(31 downto 0);
      in_destreg          : in  std_logic_vector(4 downto 0);
      in_imm              : in  std_logic_vector(31 downto 0);
      in_ip               : in  std_logic_vector(31 downto 0);
      in_shift            : in  std_logic_vector(4 downto 0);
      in_mux1             : in  std_logic_vector(1 downto 0);
      in_mux2             : in  std_logic_vector(1 downto 0);
      in_alu_instruction  : in  std_logic_vector(5 downto 0)

    );
end entity execution;

architecture behave of execution is
signal mux1_out, mux2_out: std_logic_vector(31 downto 0);
begin



  process(in_destreg, in_b, in_mux1, in_shift, in_a, in_mux2, in_imm, in_ip)
  begin
    ex_destreg <= in_destreg;
    ex_data <= in_b;

    case in_mux1 is
      -- 00
      when "00" =>
      mux1_out <= b"000_0000_0000_0000_0000_0000_0000" & in_shift;
      -- 01
      when "01" =>
      mux1_out <= x"0000_0004";
      -- 10
      when "10" =>
      mux1_out <= in_a;
      -- others
      when others => mux1_out <= (others => 'X');
    end case;

    case in_mux2 is
      -- 00
      when "00" =>
      mux2_out <= in_b;
      -- 01
      when "01" =>
      mux2_out <= in_imm;
      -- 10
      when "10" =>
      mux2_out <= in_ip;
      -- others
      when others => mux2_out <= (others => 'X');
    end case;

  end process;

  alu1: entity work.alu(behave) port map(mux1_out, mux2_out,
  in_alu_instruction, ex_alu, ex_alu_zero);

end architecture behave;
```

../src/vhdl/execution.vhd

```
-- Revision history:
-- 10.08.2015   Patrick Appenheimer        created
-- 10.08.2015   Carlos Minamisava Faria    moore state machine states definition
-- 10.08.2015   Carlos Minamisava Faria & Patrick Appenheimer    Instructions added
-- 11.08.2015   Patrick Appenheimer        added state_register and state_decode
```

```vhdl
-- 12.08.2015    Patrick Appenheimer    minor changes
-- 14.08.2015    Patrick Appenheimer    stall logic changed

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;


library WORK;
use WORK.all;

entity FSM is
  port (
    clk                     : in std_logic;
    rst                     : in std_logic;
    instr_in                : in  std_logic_vector(31 downto 0);
    stall                   : in  std_logic;
    out_currentstate        : out std_logic_vector(4 downto 0);
    out_nextstate           : out std_logic_vector(4 downto 0);
    out_buffer              : out std_logic_vector(29 downto 0);
    out_busy                : out std_logic;
    in_go                   : in  std_logic
    );
end entity FSM;

architecture behavioral of FSM is

--      State Machine    --
  constant s0     : std_logic_vector(4 downto 0) := b"00000";
  constant s1     : std_logic_vector(4 downto 0) := b"00001";
  constant s2     : std_logic_vector(4 downto 0) := b"00010";
  constant s3     : std_logic_vector(4 downto 0) := b"00011";
  constant s4     : std_logic_vector(4 downto 0) := b"00100";
  constant sX     : std_logic_vector(4 downto 0) := b"11111";

--      Arithmetic      --
  constant addiu : std_logic_vector(5 downto 0) := b"0010_01";  -- Type I
--      Data Transfer    --
  constant lui    : std_logic_vector(5 downto 0) := b"0011_11";  -- Type I       --Register access
  constant lbu    : std_logic_vector(5 downto 0) := b"1001_00";  -- Type I       --Memory access
  constant lw     : std_logic_vector(5 downto 0) := b"1000_11";  -- Type I       --Memory access
  constant sb     : std_logic_vector(5 downto 0) := b"101000";  -- Type I       --Memory access
  constant sw     : std_logic_vector(5 downto 0) := b"101011";  -- Type I       --Memory access
--      Logical --
  constant slti   : std_logic_vector(5 downto 0) := b"001010";  -- Type I
  constant andi   : std_logic_vector(5 downto 0) := b"0011_00";  -- Type I
  constant shift  : std_logic_vector(5 downto 0) := b"0000_00";  -- Type R       --NOP is read as sll $0,$0,0
--      Conditional branch      --
  constant beqz   : std_logic_vector(5 downto 0) := b"000100";  -- Type I
  constant bnez   : std_logic_vector(5 downto 0) := b"000101";  -- Type I
--      Unconditional jump      --
  constant j      : std_logic_vector(5 downto 0) := b"0000_10";  -- Type J
  constant jalx   : std_logic_vector(5 downto 0) := b"0011_01";  -- Type J

  constant r_type : std_logic_vector(5 downto 0) := b"0000_00";  -- Type R


-- output_buffer is a register with all control outputs of the state machine:
-- output_buffer (29 downto 29): pc_mux              : out std_logic;
-- output_buffer (28 downto 27): id_regdest_mux      : out std_logic_vector (1 downto 0);
-- output_buffer (26 downto 25): id_regshift_mux     : out std_logic_vector (1 downto 0);
-- output_buffer (24 downto 24): id_enable_regs      : out std_logic;
-- output_buffer (23 downto 22): exc_mux1            : out  std_logic_vector(1 downto 0);
-- output_buffer (21 downto 20): exc_mux2            : out  std_logic_vector(1 downto 0);
-- output_buffer (19 downto 14): alu_instruction     : out  std_logic_vector(5 downto 0);
-- output_buffer (13 downto 13): mem_mux_decision    : out std_logic;
-- output_buffer (12 downto 9):  rd_mask             : out std_logic_vector(3  downto 0);
-- output_buffer (8 downto 5):   wr_mask             : out std_logic_vector(3  downto 0);
-- output_buffer (4 downto 0):   stage_control       : out std_logic_vector(4  downto 0);
  signal output_buffer : std_logic_vector(29 downto 0);

  signal currentstate  : std_logic_vector(4 downto 0);
  signal nextstate     : std_logic_vector(4 downto 0);



begin

  state_encode: process(currentstate, stall, in_go)
  begin
    case currentstate is
        when sX =>
        if (in_go = '1') then
          nextstate <= s0;
        else
          nextstate <= sX;
```

```
   end if;
   when s0 =>
   if (stall = '0') then
     nextstate <= s1;
   else
     nextstate <= s0;
   end if;
   when s1 =>
   if (stall = '0') then
     nextstate <= s2;
   else
     nextstate <= s1;
   end if;
   when s2 =>
   if (stall = '0') then
     nextstate <= s3;
   else
     nextstate <= s2;
   end if;
   when s3 =>
   if (stall = '0') then
     nextstate <= sX;
   else
     nextstate <= s3;
   end if;
   when s4 =>
   if (stall = '0') then
     nextstate <= sX;
   else
     nextstate <= s4;
   end if;
   when others => nextstate <= sX;
 end case;
end process state_encode;

state_register: process(rst, clk)
begin
 if (rst = '1') then
   currentstate <= sX;
 elsif (clk'event and clk = '1') then
  currentstate <= nextstate;
 end if;
end process state_register;

out_buffer_ctrl: process(clk)
begin
 if(clk'event and clk = '0') then
  out_buffer <= output_buffer;
 end if;
end process;


state_decode: process(currentstate)
begin
 out_currentstate <= currentstate;
 out_nextstate <= nextstate;

 case currentstate is
  when sX =>
   out_busy <= '0';
  when s0 =>
   out_busy <= '1';
  when s3 =>
   out_busy <= '0';
  when others =>
   -- do something
 end case;
end process state_decode;

out_buff_ctr: process(instr_in)
begin
case instr_in (31 downto 26) is
      when lui    => output_buffer <= b"0_10_01_1_00_01_000100_0_0000_0000_11111";
      when addiu  => output_buffer <= b"0_10_00_1_10_01_100000_0_0000_0000_11111";
      when lbu    => output_buffer <= b"0_10_00_1_10_01_100000_1_0001_0000_11111";
      when lw     => output_buffer <= b"0_10_00_1_10_01_100000_1_1111_0000_11111";
      when sb     => output_buffer <= b"0_10_00_0_10_01_100000_0_0000_0001_11111";
      when sw     => output_buffer <= b"0_10_00_0_10_01_100000_0_0000_1111_11111";
      when slti   => output_buffer <= b"0_10_00_1_10_01_001000_0_0000_0000_11111";
      when andi   => output_buffer <= b"0_10_01_1_00_01_100100_0_0000_0000_11111";
      when shift  => output_buffer <= b"0_00_00_1_00_00_001000_0_0000_0000_11111";
      when beqz   => output_buffer <= b"1_10_00_0_00_00_000001_0_0000_0000_11111";
      when bnez   => output_buffer <= b"1_10_00_0_00_00_000001_0_0000_0000_11111";
      when j      => output_buffer <= b"1_10_00_0_00_00_000001_0_0000_0000_11111";
      when jalx   => output_buffer <= b"1_10_00_0_00_00_000001_0_0000_0000_11111";
      --when r_type => output_buffer <= b"0_00_00_1_10_00_000000_0_0000_0000_11111";
```

```
              when others => output_buffer <= b"0_00_00_0_00_00_000000_0_0000_0000_11111";
        end case;
    end process;

end architecture behavioral;
```

../src/vhdl/fsm.vhd

```
-- Implementation of a 5-stage pipelined MIPS processor's instruction decode stage
-- 2015-08-03   Lukas Jaeger      created
-- 2015-08-04   Lukas Jaeger      added architecture and started to implement both processes
-- 2015-08-04   Lukas Jaeger      added asynchronous reset
-- 2015-08-05 Lukas Jaeger   fixed bugs that resulted from me not knowing any VHDL
-- 2015-08-05 Lukas Jaeger   added functionality for branch logic
-- 2015-08-06 Lukas, Carlos     fixed bug in JAL-instruction-decode
-- 2015-08-06 Lukas     added signed/unsigned logic for immediate-output
-- 2015-08-07 Lukas     added signed/unsigned exceptions for LW-instructions
-- 2015-08-11   Lukas              fixed some bugs in forwarding
-- 2015-08-11 Bahri Enis Demirtel added BLEZ, BLTZ, BLTZAL, BNE
-- 2015-08-12   Lukas             fixed bug in immediate expansion and made it falling clock edge sensitive
library IEEE;
    use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;

entity instruction_decode is
    port(instr,ip_in, writeback, alu_result, mem_result : in std_logic_vector (31 downto 0);
        writeback_reg, regdest_ex, regdest_mem : in std_logic_vector (4 downto 0);
        regdest_mux, regshift_mux: in std_logic_vector (1 downto 0);
        clk, reset, enable_regs: in std_logic;
        reg_a, reg_b, imm, ip_out : out std_logic_vector (31 downto 0);
        reg_dest, shift_out : out std_logic_vector (4 downto 0)
        );
end entity;

architecture behavioral of instruction_decode is
  type regfile is array (31 downto 0) of std_logic_vector (31 downto 0);
      signal register_file : regfile;
      signal imm_internal : std_logic_vector(31 downto 0) := x"00000000";
  signal internal_writeback : std_logic_vector(31 downto 0);
      signal pc_imm : std_logic_vector (31 downto 0);
      signal imm_16 : std_logic_vector (15 downto 0);
  signal internal_wb_flag : std_logic := '0';
begin
  imm_16  <= instr (15 downto 0);

  -- Splitting registers for R-type-instructions
      pc_imm <= imm_internal (31 downto 2) & "00";
  -- Defines the instruction decode logic
      logic : process (instr, ip_in, writeback, alu_result, mem_result, writeback_reg, regdest_ex, regdest_mem, regdest_mux,
     regshift_mux)  is
        begin

    -- Forwarding logic for reg_a
    -- If the destination register is still used by the writeback-phase, the writeback-output is forwarded
                if ((instr (25 downto 21) = regdest_ex) and (instr (25 downto 21) /= "00000")) then
      reg_a <= alu_result;
    -- If the destination register is still used by the memory-phase, the alu-result is forwarded
    elsif ((instr (25 downto 21) = regdest_mem) and (instr (25 downto 21) /= "00000")) then
      reg_a <= mem_result;
                elsif ((instr (25 downto 21) = writeback_reg)and (instr (25 downto 21) /= "00000")) then
                        reg_a <= writeback;
    -- Otherwise, no forwarding is required and the register specified by rs is read
    else
      reg_a <= register_file(to_integer(unsigned (instr (25 downto 21))));
    end if;


    --Forwarding logic for reg_b. Works analogously to the reg_a block above
    if ((instr (20 downto 16) = regdest_ex) and (instr (20 downto 16) /= "00000")) then
      reg_b <= alu_result;
    elsif ((instr (20 downto 16) = regdest_mem) and (instr (20 downto 16) /= "00000")) then
      reg_b <= mem_result;
                elsif ((instr (20 downto 16) = writeback_reg) and (instr (20 downto 16) /= "00000")) then
                        reg_b <= writeback;
    else
      reg_b <= register_file(to_integer(unsigned (instr (20 downto 16))));
    end if;

          case regshift_mux is    -- Determines the output at shift_out
                when "00" => shift_out <= instr(10 downto 6);
                when "01" => shift_out <= "10000";
                when others => shift_out <= "00000";
          end case;

          case regdest_mux is     -- Determines the output at reg_dest
```

```
              when "00" => reg_dest <= instr (15 downto 11);
              when "01" => reg_dest <= "11111";
              when "10" => reg_dest <= instr (20 downto 16);
              when others => reg_dest <= "00000";
        end case;
    end process;

-- Process for clocked writebacks to the register file and the asynchronous reset
register_file_write : process (clk,reset,writeback_reg) is
    begin
      if (reset= '1') then    -- asynchronous reset
        for i in 0 to 31 loop
                register_file(i) <= x"00000000";
            end loop;
        elsif (clk'event and clk = '0') then
      if (enable_regs = '1') then  -- If register file is enabled, write back result
        if (to_integer(unsigned (writeback_reg)) > 0) then
                register_file(to_integer(unsigned (writeback_reg))) <= writeback;
        end if;
      elsif (internal_wb_flag = '1') then
        register_file (31) <= internal_writeback;
      end if;
    end if;
    end process;
-- Process that defines the branch logic
branch_logic : process (instr, ip_in, writeback, alu_result, mem_result, writeback_reg, regdest_ex, regdest_mem, regdest_mux,
   regshift_mux) is
variable offset : integer;
variable a, b : integer;
begin
  -- Prepares values of reg_a and reg_b for comparison
  if ((instr (25 downto 21) = regdest_ex) and (instr (25 downto 21) /= "00000")) then
    a := to_integer(signed(alu_result));
  elsif ((instr (25 downto 21) = regdest_mem) and (instr (25 downto 21) /= "00000")) then
    a := to_integer(signed(mem_result));
              elsif ((instr (25 downto 21) = writeback_reg) and (instr (25 downto 21) /= "00000")) then
                      a := to_integer(signed(writeback));
  else
    a := to_integer(signed(register_file(to_integer(unsigned (instr (25 downto 21))))));
  end if;


  if ((instr (20 downto 16) = regdest_ex) and (instr (20 downto 16) /= "00000")) then
    b:= to_integer(signed(alu_result));
  elsif ((instr (20 downto 16) = regdest_mem) and (instr (20 downto 16) /= "00000")) then
    b := to_integer(signed(mem_result));
              elsif ((instr (20 downto 16) = writeback_reg) and (instr (20 downto 16) /= "00000")) then
                      b := to_integer(signed(writeback));
  else
    b := to_integer(signed(register_file(to_integer(unsigned (instr (20 downto 16))))));
  end if;


  -- Annoyingly lengthy list of if-statements for calculation of branch logic
  if (instr (31 downto 26) = "000010") then -- Jump instruction
    internal_wb_flag <= '0';
    offset := to_integer(signed(instr(25 downto 0)));
    offset := (offset * 4);
    ip_out <= ip_in (31 downto 28) & std_logic_vector(to_signed(offset,28));
  elsif ((instr (31 downto 26) = "000011") or (instr (31 downto 26) = "011101")) then --JAL(X) instruction
    internal_writeback <= std_logic_vector(to_unsigned(to_integer(unsigned(ip_in)) + 4,32));
    internal_wb_flag <= '1';
    offset := to_integer(signed(instr(25 downto 0)));
    offset := offset * 4;
    ip_out <= ip_in (31 downto 28) & std_logic_vector(to_signed(offset,28));
  elsif ((instr(31 downto 26) = "000000") and (instr (20 downto 0) ="000000000000000001000")) then --JR instruction
    internal_wb_flag <= '0';
    offset := to_integer(signed(instr(25 downto 0)));
    offset := offset * 4;
    -- VHDL code de ja-vu?
    -- This is the same forwarding logic as above for reg_a
                      if ((instr (25 downto 21) = regdest_ex) and (instr (25 downto 21) /= "00000")) then
      ip_out <= alu_result;
    elsif ((instr (25 downto 21) = regdest_mem) and (instr (25 downto 21) /= "00000")) then
      ip_out <= mem_result;
                      elsif ((instr (25 downto 21) = writeback_reg) and (instr (25 downto 21) /= "00000")) then
                              ip_out <= writeback;
    else
      ip_out <= register_file(to_integer(unsigned (instr (25 downto 21))));
    end if;
              elsif (instr (31 downto 26) = "000100") then        --BEQ
                      internal_wb_flag <= '0';
                      if (a = b) then
                              offset := to_integer(signed(instr(15 downto 0)));
                              offset := offset * 4;
                              offset := offset + to_integer (signed(ip_in));
```

```
                                ip_out <= std_logic_vector(to_signed(offset,32));
                        else
                                ip_out <= std_logic_vector(to_unsigned(to_integer(unsigned(ip_in)) + 4, 32));
                        end if;
    elsif ((instr (31 downto 26) = "000001") and (instr (20 downto 16) = "00001")) then --BGEZ instruction
      internal_wb_flag <= '0';
      b :=0;
                        if (a >= b) then
                                offset := to_integer(signed(instr(15 downto 0)));
                                offset := offset * 4;
                                offset := offset + to_integer (signed(ip_in));
                                ip_out <= std_logic_vector(to_signed(offset,32));
                        else
                                ip_out <= std_logic_vector(to_unsigned(to_integer(unsigned(ip_in)) + 4, 32));
                        end if;
    elsif ((instr (31 downto 26) = "000001") and (instr (20 downto 16) = "10001")) then --BGEZAL instruction
      internal_wb_flag <= '1';
      internal_writeback <= std_logic_vector(to_unsigned(to_integer(unsigned(ip_in)) + 4,32));
      b :=0;
                        if (a >= b) then
                                offset := to_integer(signed(instr(15 downto 0)));
                                offset := offset * 4;
                                offset := offset + to_integer (signed(ip_in));
                                ip_out <= std_logic_vector(to_signed(offset,32));
                        else
                                ip_out <= std_logic_vector(to_unsigned(to_integer(unsigned(ip_in)) + 4, 32));
                        end if;
    elsif ((instr (31 downto 26) = "000111")and(instr (20 downto 16)="00000")) then -- BGTZ
      internal_wb_flag <= '0';
      b :=0;
      report "The value of 'a' is " & integer'image(a);
      if (a > b) then
                                offset := to_integer(signed(instr(15 downto 0)));
                                offset := offset * 4;
                                offset := offset + to_integer (signed(ip_in));
                                ip_out <= std_logic_vector(to_signed(offset,32));
                        else
                                ip_out <= std_logic_vector(to_unsigned(to_integer(unsigned(ip_in)) + 4, 32));
                        end if;
    elsif ((instr (31 downto 26) = "000110")and(instr (20 downto 16)="00000")) then -- BLEZ
      internal_wb_flag <= '0';
      b :=0;
      if (a <= b) then
                                offset := to_integer(signed(instr(15 downto 0)));
                                offset := offset * 4;
                                offset := offset + to_integer (signed(ip_in));
                                ip_out <= std_logic_vector(to_signed(offset,32));
                        else
                                ip_out <= std_logic_vector(to_unsigned(to_integer(unsigned(ip_in)) + 4, 32));
                        end if;
    elsif ((instr (31 downto 26) = "000001")and(instr (20 downto 16)="00000")) then -- BLTZ
      internal_wb_flag <= '0';
      b :=0;
      if (a < b) then
                                offset := to_integer(signed(instr(15 downto 0)));
                                offset := offset * 4;
                                offset := offset + to_integer (signed(ip_in));
                                ip_out <= std_logic_vector(to_signed(offset,32));
                        else
                                ip_out <= std_logic_vector(to_unsigned(to_integer(unsigned(ip_in)) + 4, 32));
                        end if;
    elsif ((instr (31 downto 26) = "000001")and(instr (20 downto 16)="10000")) then -- BLTZAL
      internal_wb_flag <= '1';
      internal_writeback <= std_logic_vector(to_unsigned(to_integer(unsigned(ip_in)) + 4,32));
      b :=0;
      if (a < b) then
                                offset := to_integer(signed(instr(15 downto 0)));
                                offset := offset * 4;
                                offset := offset + to_integer (signed(ip_in));
                                ip_out <= std_logic_vector(to_signed(offset,32));
                        else
                                ip_out <= std_logic_vector(to_unsigned(to_integer(unsigned(ip_in)) + 4, 32));
                        end if;
    elsif (instr (31 downto 26) = "000101") then -- BNE
      internal_wb_flag <= '0';
      if (a /= b) then
                                offset := to_integer(signed(instr(15 downto 0)));
                                offset := offset * 4;
                                offset := offset + to_integer (signed(ip_in));
                                ip_out <= std_logic_vector(to_signed(offset,32));
                        else
                                ip_out <= std_logic_vector(to_unsigned(to_integer(unsigned(ip_in)) + 4, 32));
                        end if;
    else
      ip_out <= std_logic_vector(to_unsigned(to_integer(unsigned(ip_in)) + 4, 32));
    end if;
```

```
  end process;

  imm_expand : process (instr) is
  begin
              imm <= std_logic_vector(to_signed(to_integer(signed (instr (15 downto 0))),32));
  end process;
end architecture;

-- FSM-signal-Howto:
--
-- regdest_mux:
-- 00: if instruction is of R-type
-- 01: if regdest must be set to 31 (JAL?)
-- 10: if instruction is of I-type
-- 11: NEVER EVER EVER!!!
--
-- regshift_mux:
-- 00: if instruction is of R-type
-- 01: if shift must be 16 (No idea which instruction uses that...)
-- 10: if you like non-deterministic behaviour
-- 11: if you love non-deterministic behaviour
--
-- enable_regs:
-- 1: if the writeback-stage just finished an R-type- or I-type-instruction (except for JR)
-- 0: if the writeback-stage just finished a J-type-instruction or JR
```

../src/vhdl/instruction_decode.vhd

```
-- 03.08.2015      Bahri Enis Demirtel     created


library IEEE;
  use IEEE.std_logic_1164.ALL;

library IEEE;
  use IEEE.STD_LOGIC_UNSIGNED.ALL;

library WORK;
  use WORK.all;


entity instruction_fetch is
port(

clk : in std_logic;
rst : in std_logic;

PC : in std_logic_vector(31 downto 0);      --PC : in std_logic_vector(CPU_ADDR_WIDTH-1 downto 0);                    --PC
     (32 bit)
InstrData : in std_logic_vector(31 downto 0);     --InstrData : in std_logic_vector(CPU_DATA_WIDTH-1 downto 0);           --
     InstrData, Adress information comes from Memory(32 bit)

--StallData : in std_logic;

IR : out std_logic_vector(31 downto 0);     --IR : out std_logic_vector(CPU_ADDR_WIDTH-1 downto 0);                  --IR, Next PC
     goes to Execution Stage(32 bit)
InstrAddr: out std_logic_vector(31 downto 0); --InstrAddr: out std_logic_vector(CPU_ADDR_WIDTH-1 downto 0);           --InstrAddr
     , PC goes to Memory(32 bit)
Instr : out std_logic_vector(31 downto 0) --Instr : out std_logic_vector(CPU_DATA_WIDTH-1 downto 0);                 --Instr, Adress
      information from Memory goes to Instruction Decoder(32 bit)


);

end entity instruction_fetch;




architecture behavioral of instruction_fetch is


  begin

  process (rst, clk, PC, InstrData) is
  begin

    if(rst = '1') then
    InstrAddr <= X"0000_0000";      --If reset comes PC goes to the beginning, its value is 0000_0000
    IR <= X"0000_0000";      --If reset all coming signals are 0000_0000
    Instr <= X"0000_0000";      --If reset all coming signals are 0000_0000


      else
             --elsif (rising_edge(clk)) then
```

```
            --if(StallData='0') then

        InstrAddr <= PC;         --We can the value of PC to the memory adress
        IR <= PC + X"0000_0004";  --IR value is always PC+4;
        Instr <= InstrData;      --Instr value is always equal to InstrData value

    end if;
  end process ;

end architecture behavioral;
```

../src/vhdl/instruction_fetch.vhd

```
-- Revision history:
-- 03.08.2015 Carlos Minamisava Faria created
-- 03.08.2015 Carlos Minamisava Faria entity MemoryStage
-- 04.08.2015 Carlos Minamisava Faria architecture MemoryStage
-- 05.08.2015 Carlos Minamisava Faria first working version
-- 11.08.2015   Lukas Jaeger            fixed a bug in memory access
library IEEE;
  use IEEE.std_logic_1164.ALL;
  USE IEEE.numeric_std.ALL;

library WORK;
  use WORK.all;

entity MemoryStage is
port(
  clk: in std_logic;
  rst: in std_logic;

  aluResult_in: in std_logic_vector(31 downto 0); --CPU_DATA_WIDTH-1 downto 0); -- ALU results from Execution Stage

  data_in: in std_logic_vector(31 downto 0);--CPU_DATA_WIDTH-1 downto 0); -- Data from execution stage
                  -- Memory Read/Write decision comes from the FSS
  data_addr: out std_logic_vector(31 downto 0);--CPU_ADDR_WIDTH-1 downto 0);  -- Memory address output for memory r/w
  data_from_cpu: out std_logic_vector(31 downto 0);--CPU_DATA_WIDTH-1 downto 0);  -- Memory data out.
  data_to_cpu: in std_logic_vector(31 downto 0);--CPU_DATA_WIDTH-1 downto 0); -- Read data from memory.

--not needed--     data_stall              : in  std_logic;        -- data stall - cpu input

  mux_decision: in std_logic;         -- FSS decision for writeback output. ALU results or memory data can be forwarded to
     writeback

  writeback: out std_logic_vector( 31 downto 0);--CPU_DATA_WIDTH-1 downto 0); -- Data to send to next stage: Writeback

  reg_dest_in: in std_logic_vector(4 downto 0);--CPU_REG_ADDR_WIDTH-1 downto 0);        -- k.A.
  reg_dest_out: out std_logic_vector(4 downto 0));--CPU_REG_ADDR_WIDTH-1 downto 0));     -- k.A.
end entity MemoryStage;

architecture behavioral of MemoryStage is
--  signal memory_buffer: std_logic_vector(31 downto 0);--CPU_DATA_WIDTH-1 downto 0);
  begin

    -- Data address and data are always routed out.
    data_addr <= aluResult_in;
    data_from_cpu <= data_in;

    -- reg_dest is forwarded
    reg_dest_out <= reg_dest_in;

    output: process (rst, aluResult_in,data_in, data_to_cpu, mux_decision, reg_dest_in)is
    begin
      if (rst='1') then                                    -- reset condition
        writeback <= x"00_00_00_00";
      else
        if (mux_decision ='0') then   -- mux_decision choses between the two possible outputs: the result from ALU of the read
     memory
          writeback <= aluResult_in;  -- output is the aluResult_in
        else
          writeback <= data_to_cpu; -- output is the memory_buffer, which carries the memory read value.
        end if;
      end if;
    end process output;

end architecture behavioral;


-- FSM-signal-Howto:
--
-- mux_decision:
```

```
-- 0: to forward aluResult
-- 1: to forward memory data
```

../src/vhdl/mem_stage.vhd

```
-- 03.08.2015      Bahri Enis Demirtel     created


library IEEE;
  use IEEE.std_logic_1164.ALL;


entity write_back is
port(

clk : in std_logic;
rst : in std_logic;
writeback_in : in  std_logic_vector(31 downto 0); -- : in  std_logic_vector(CPU_ADDR_WIDTH-1 downto 0);
regdest_in : in  std_logic_vector(4 downto 0);     -- : in  std_logic_vector(CPU_REG_ADDR_WIDTH-1 downto 0);
writeback_out : out std_logic_vector(31 downto 0);  -- :out  std_logic_vector(CPU_ADDR_WIDTH-1 downto 0);
regdest_out : out std_logic_vector(4 downto 0)     -- : out std_logic_vector(CPU_REG_ADDR_WIDTH-1 downto 0)
);


end entity write_back;


architecture behavioral of write_back is

begin

process (rst, clk, writeback_in, regdest_in) is
  begin

    if(rst = '1') then
    writeback_out <= x"0000_0000";
    regdest_out <= b"00000";



    else

      writeback_out <= writeback_in;
      regdest_out <= regdest_in;

    end if;
  end process ;
end architecture behavioral;
```

../src/vhdl/write_back.vhd