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

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.

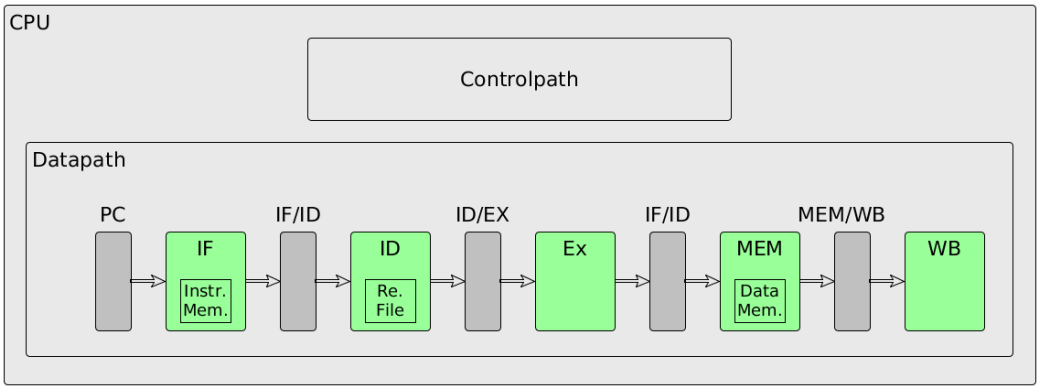

Figure 2.1: Datapath pipeline

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:

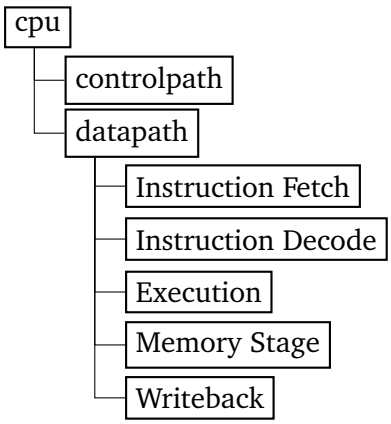

Figure 2.2: CPU component structure

2.1 ALU

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.

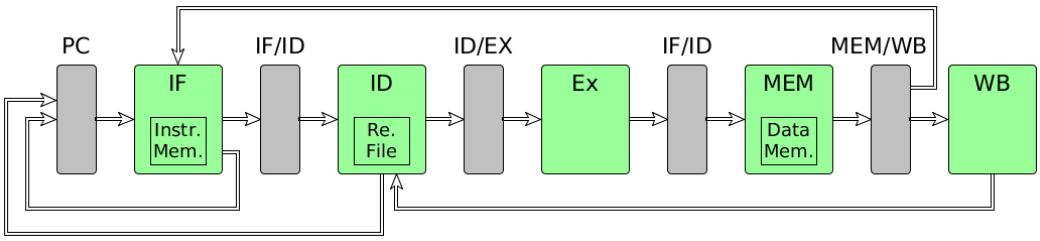

Figure 2.3: Data-path pipeline

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].

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.

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 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.

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.

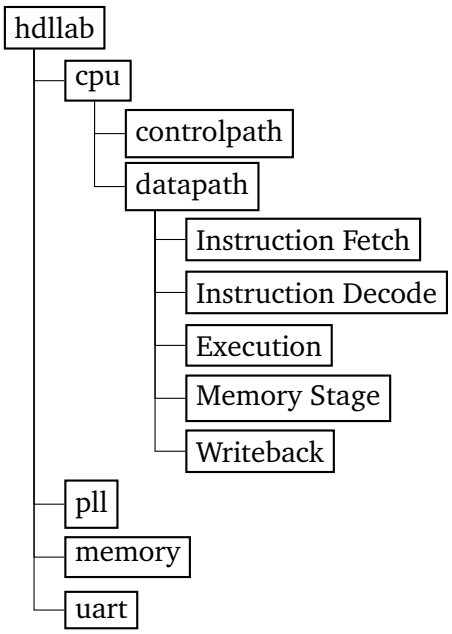

Figure 3.1: CPU component structure

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

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.