Digital Design and Computer Architecture LU

Lab Exercise III

Florian Huemer, Florian Kriebel, Jürgen Maier
{fhuemer, fkriebel, jmaier}@ecs.tuwien.ac.at
Department of Computer Engineering
TU Wien

Vienna, June 3, 2020

Contents

1 Introduction

This document contains the assignment for Exercise III. The deadline for the exercise is:

- Exercise III: 01.07.2020, 23:55

The combined points achieved in Exercise III and Exercise IV count 25 % to the overall grade of the course. Please hand in your solutions via TUWEL. We would like to encourage you to fill out the feedback form in TUWEL after you submitted your solution. The feedback is anonymous and helps us to improve the course.

Please note that this document is not the complete assignment. View the protocol template for additional tasks. Make sure that all required details can be seen on the screenshots you put into your report, otherwise these screenshots will be graded with zero points.

1.1 Coding Style

Refer to the "VHDL Coding and Design Guidelines" document (see TUWEL) before starting the work on your solution. Moreover, we highly recommend to implement state machines with the 2 or 3-process method discussed in the lecture. If you use the 1-process method you can quite easily introduce very hard-to-find bugs in your code.

We further recommend to use the "named association" method for creating and connecting instances. For the instance naming, use the corresponding entity name followed by the suffix `_inst` (e.g., `alu_inst` for the ALU). In case multiple instances are required, use `_inst1`, `_inst2`, ...

1.2 Software

As discussed in more detail in the Design Flow Tutorial, we are using Quartus and QuestaSim (formerly ModelSim) in the lab. If you want to work on your own computer you can download a free version of Quartus (Quartus Prime Lite Edition) and Questa/Modelsim (ModelSim-Intel) from the Intel website.[1] However, note that the simulation performance with ModelSim-Intel might be reduced, when compared to the full version of Questa/Modelsim provided in the lab.

1.3 Submission

Please note that it is mandatory to keep the files exactly in the required folders as defined by the provided template. Do **not** add additional packages, source files, etc., but use the provided files and packages appropriately. Moreover, the interfaces and record/type definitions explained in this document (and to be found in the corresponding source files) must **not** be changed.

[1] `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/download.html`

2 Overview

This section provides an overview of the architecture to be implemented: MiRiV – a minimal RISC-V implementation. It implements the majority (but not all) instructions of the RV32I Base Integer Instruction Set [3] without any extensions and for the most part follows the implementation described in [2]. Note that RISC-V is a load-store architecture, where memory is addressed as 8-bit bytes using little-endian ordering.

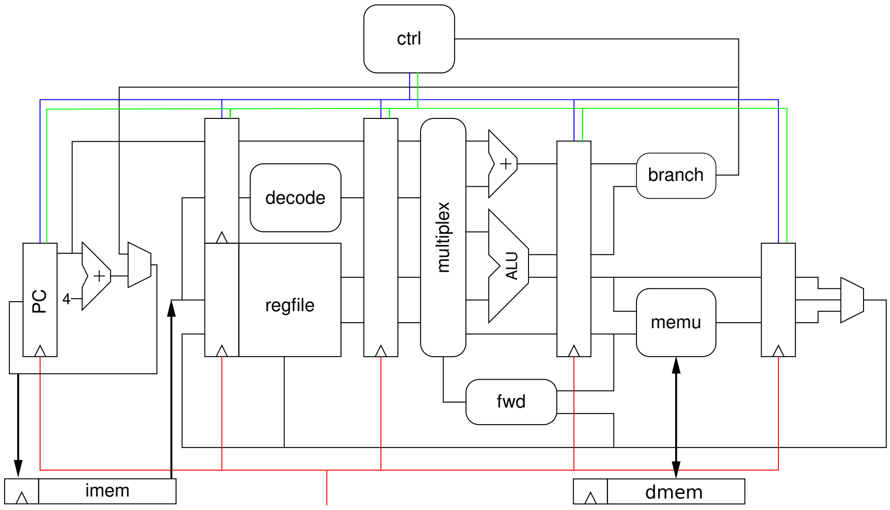


Figure 2.1: MiRiV pipeline

Figure 2.1 shows the 5-stage pipeline of the processor to be implemented. It comprises five pipeline stages: fetch, decode, execute, memory and write-back. The data path is drawn in black; the red signal is the clock, signals that flush a pipeline stage are blue and signals that stall the pipeline are green. Also take note of the pipeline registers and make sure to implement them at the correct place with an appropriate reset. The figure shows an abstract view of system to be implemented. In order to provide a quick overview, several details are not shown, but they are described in the following sections.

In the upcoming assignments, the parts to be implemented will be shown in light blue and entities to be instantiated will be shaded, to ease your navigation through the design.

3 Level 0: Basic Elements [4 Points]

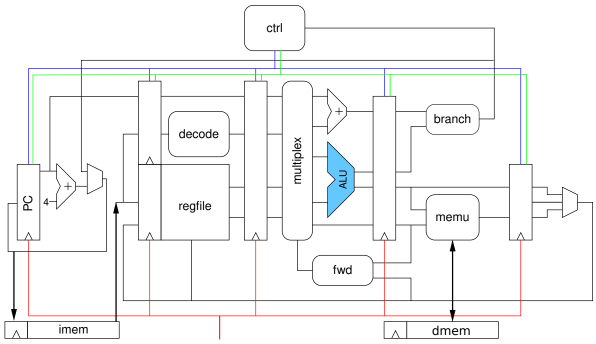
This assignment consists of three relatively simple hardware units. Implement the units described in this section, and write appropriate testbenches (store the testbenches in the location described in Section 5.3.1). Test the units thoroughly, as errors introduced at this stage might be very difficult to find in later stages.

Evaluation

The assignment will be evaluated with testbenches, which test the individual components. Points will be granted if the testbenches are passed successfully.

3.1 ALU

`alu.vhd`



Description

As the name suggests, the arithmetic logic unit (ALU) carries out arithmetic and logic operations. The interface of the ALU is described in Table 3.1; it shall implement the operations described in Table 3.2. The computation of the zero flag `Z` is shown in Table 3.3. Note that the shift operations can be implemented conveniently with the functions `shift_left()` and `shift_right()` from the package `numeric_std`.

Signal	Direction	Type	Description
`op`	in	`ALU_OP_TYPE`	Operation
`A`	in	`DATA_TYPE`	Operand A
`B`	in	`DATA_TYPE`	Operand B
`R`	out	`DATA_TYPE`	Result
`Z`	out	`std_logic`	Zero flag

Table 3.1: ALU interface

op	`R`
`ALU_NOP`	`B`
`ALU_SLT`	`A < B ? 1 : 0`, signed
`ALU_SLTU`	`A < B ? 1 : 0`, unsigned
`ALU_SLL`†	`A sll B(4 downto 0)`
`ALU_SRL`*	`A srl B(4 downto 0)`
`ALU_SRA`§	`A sra B(4 downto 0)`
`ALU_ADD`	`A + B`, signed
`ALU_SUB`	`A - B`, signed
`ALU_AND`	`A and B`
`ALU_OR`	`A or B`
`ALU_XOR`	`A xor B`

†shift left logical; *shift right logical;
§shift right arithmetic

Table 3.2: ALU result computation

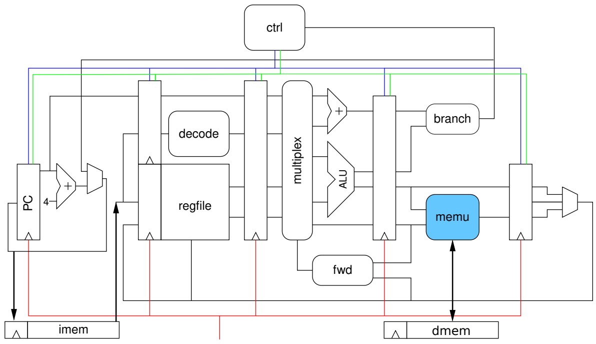
op	Z
`ALU_SUB`	`if A = B then Z <= '1'; else Z <= '0'; end if;`
`ALU_SLT`	`not R(0)`
`ALU_SLTU`	`not R(0)`
otherwise	`'-'`*

* use `'-'`, not an arbitrary value

Table 3.3: ALU zero-flag computation

3.2 Memory Unit

memu.vhd



Description

The memory unit is responsible for issuing memory access commands to the external interface, which connects the pipeline to the data memory (dmem). As the external interface is word-based (a word being 32 bits wide), the memory unit must translate sub-word accesses. The interface of the memory unit is described in Table 3.4. `MEMU_OP_TYPE`, `MEM_IN_TYPE` and `MEM_OUT_TYPE` are record types; their fields are described in Tables 3.5, 3.6 and 3.7.

The basic types for memory addressing and data are shown in Table 3.8. For the pipeline, `data_type` is used for both data and addresses. Note the difference between `mem_data_type` and `data_type`. While the former type is used by the memory, the latter type is used by the pipeline. In general, the available memory and the addressable memory by the pipeline might differ, similarly the expected data access granularities might be different. To partially avoid this complication, the pipeline data type is identical to the memory data type here. Note, however, that the pipeline operates on byte addresses while the interface to the memory operates on word addresses.

Table 3.9 shows how `M.byteena` and `M.wrdata` are computed. For this table, it is assumed that `W` consists of four bytes $b_3b_2b_1b_0$, with b_3 being the most significant byte and b_0 the least significant byte. A value b_0`XXX` in the last column states that the most significant byte of `M.wrdata` is the least significant byte from `W`, and other bytes are irrelevant and may contain arbitrary values. The value of `M.address` is the word address of `A`; other outputs must be set as described below.

How values from the external interface are translated is shown in Table 3.10. Here it is assumed that `D.rddata` consists of four bytes $b_3b_2b_1b_0$, with b_3 being the most significant byte and b_0 the least significant byte. Furthermore, `0` signifies that the byte is set to zero, and `S` that the value is sign-extended. For example, the value `SSS`b_3 means `R` is the sign-extended most significant byte of `D.rddata`.

Assert `B` if a valid memory read access is starting or ongoing. An ongoing read access is indicated by `D.busy`.

Tables 3.11 and 3.12 show how the load exception signal `XL` and the store exception signal `XS` are computed. Note that usually `M.rd` is assigned the value of `op.memread`, and `M.wr` the value of `op.memwrite`. However, if `XL` or `XS` are asserted, `M.rd` and `M.wr` must be zero, i.e., the processor must not issue a memory access that raises an exception.

Note that RISC-V uses little-endian ordering, i.e., the least significant byte is stored at the lowest memory address (e.g., the hexadecimal number 0x1234 is stored as 0x34 0x12).

Signal	Direction	Type	Description
op †	in	MEMU_OP_TYPE	Access type
A †	in	DATA_TYPE	Address
W †	in	DATA_TYPE	Write data
R †	out	DATA_TYPE	Result of memory load
B †	out	std_logic	Memory busy
XL †	out	std_logic	Load exception
XS †	out	std_logic	Store exception
D ⋆	in	MEM_IN_TYPE	Interface from memory
M ⋆	out	MEM_OUT_TYPE	Interface to memory

†to be connected to memory stage; ⋆to be connected to memory interface

Table 3.4: Memory Unit interface

Field	Type	Description
memread	std_logic	Read from memory
memwrite	std_logic	Write to memory
memtype	MEMTYPE_TYPE	Word, half-word or byte access

Table 3.5: MEMU_OP_TYPE fields

Field	Type	Description
busy	std_logic	Memory busy
rddata	MEM_DATA_TYPE	Actual data read from memory

Table 3.6: MEM_IN_TYPE fields

Field	Type	Description
address	MEM_ADDRESS_TYPE	Address to read from or write to
rd	std_logic	Asserted for reads
wr	std_logic	Asserted for writes
byteena	MEM_BYTEENA_TYPE	Byte-enable signal for sub-word writes
wrdata	MEM_DATA_TYPE	Data to be written

Table 3.7: MEM_OUT_TYPE fields

Type	Width	Description
mem_address_type	ADDR_WIDTH	Type for memory addresses
mem_data_type	DATA_WIDTH	Type for actual data transferred to/from memory
mem_byteena_type	BYTEEN_WIDTH	Type for byte enable

Table 3.8: Basic types w.r.t. memory

Operation	A(1 downto 0)	M.byteena	M.wrdata
MEM_B | MEM_BU	"00"	"1000"	b_0XXX
	"01"	"0100"	Xb_0XX
	"10"	"0010"	XXb_0X
	"11"	"0001"	XXXb_0
MEM_H | MEM_HU	"00"	"1100"	b_0b_1XX
	"01"	"1100"	b_0b_1XX
	"10"	"0011"	XXb_0b_1
	"11"	"0011"	XXb_0b_1
MEM_W	"00"	"1111"	$b_0b_1b_2b_3$
	"01"	"1111"	$b_0b_1b_2b_3$
	"10"	"1111"	$b_0b_1b_2b_3$
	"11"	"1111"	$b_0b_1b_2b_3$

Use '-' for 'X', not an arbitrary value

Table 3.9: Computation of M.byteena and M.wrdata, W = $b_3b_2b_1b_0$

Operation	A(1 downto 0)	R
MEM_B	"00"	SSSb_3
	"01"	SSSb_2
	"10"	SSSb_1
	"11"	SSSb_0
MEM_BU	"00"	000b_3
	"01"	000b_2
	"10"	000b_1
	"11"	000b_0
MEM_H	"00"	SSb_2b_3
	"01"	SSb_2b_3
	"10"	SSb_0b_1
	"11"	SSb_0b_1
MEM_HU	"00"	00b_2b_3
	"01"	00b_2b_3
	"10"	00b_0b_1
	"11"	00b_0b_1
MEM_W	"00"	$b_0b_1b_2b_3$
	"01"	$b_0b_1b_2b_3$
	"10"	$b_0b_1b_2b_3$
	"11"	$b_0b_1b_2b_3$

Table 3.10: Computation of R, D.rddata = $b_3b_2b_1b_0$

op.memread	op.memtype	A(1 downto 0)	XL
'1'	MEM_H	"01"	'1'
'1'	MEM_H	"11"	'1'
'1'	MEM_HU	"01"	'1'
'1'	MEM_HU	"11"	'1'
'1'	MEM_W	"01"	'1'
'1'	MEM_W	"10"	'1'
'1'	MEM_W	"11"	'1'
otherwise			'0'

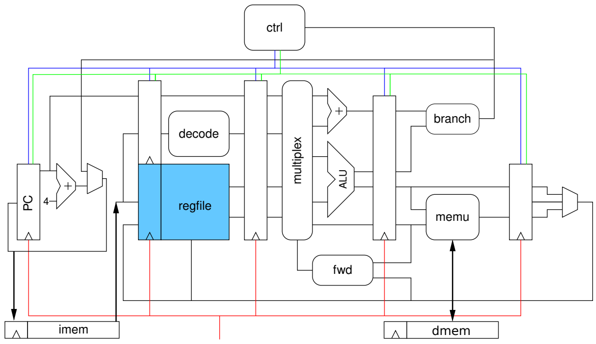
Table 3.11: Memory load exception computation

op.memwrite	op.memtype	A(1 downto 0)	XS
'1'	MEM_H	"01"	'1'
'1'	MEM_H	"11"	'1'
'1'	MEM_HU	"01"	'1'
'1'	MEM_HU	"11"	'1'
'1'	MEM_W	"01"	'1'
'1'	MEM_W	"10"	'1'
'1'	MEM_W	"11"	'1'
otherwise			'0'

Table 3.12: Memory store exception computation

3.3 Register File

`regfile.vhd`



Description

The register file is a memory with two read ports and one write port, with `2**REG_BITS` words that are `DATA_WIDTH` bits wide. The clock signal `clk` has the usual meaning and causes the circuit to latch the read and write addresses. The reset signal `reset` is active low and resets internal registers, but not necessarily the contents of the register file (initializing all registers of the register file with 0 might help avoiding problems, though). The signal `stall` causes the circuit not to latch input values such that old values are kept in all registers. Reads from address 0 must always return 0, which may be achieved by an appropriate power-up value and ignoring writes to that location or by intercepting reads from that location. When reading from a register that is written in the same cycle, the new value shall be returned.

As explained in [2], for many implementations of register files it is assumed that writing takes place in the first half of the clock cycle while reading is performed in the second half. This way writes to the register file are guaranteed to be finished, before the reads take place, ensuring that the most up-to-date values are being read. However, this approach does not work in the FPGAs used in this lab course. Therefore, the required behavior has to be implemented differently: If the internal register for a read address matches `wraddr` and `regwrite = '1'`, the register file shall return `wrdata` (i.e., you have to add an appropriate pass-through logic).

(Hint: Refer to [1] for implementation guidelines)

Signal	Direction	Type
`clk`	in	`std_logic`
`reset`	in	`std_logic`
`stall`	in	`std_logic`
`rdaddr1`	in	`REG_ADR_TYPE`
`rdaddr2`	in	`REG_ADR_TYPE`
`rddata1`	out	`DATA_TYPE`
`rddata2`	out	`DATA_TYPE`
`wraddr`	in	`REG_ADR_TYPE`
`wrdata`	in	`DATA_TYPE`
`regwrite`	in	`std_logic`

Table 3.13: Register file interface

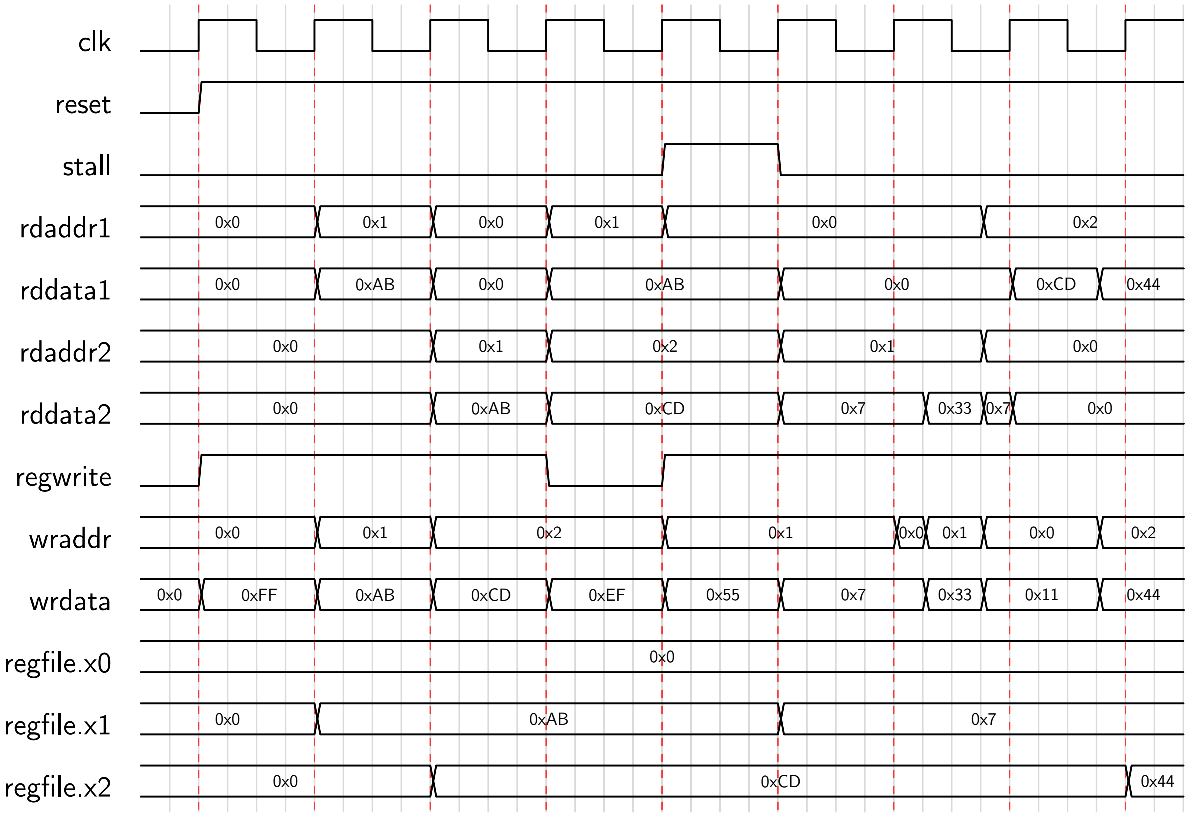


Figure 3.1: Examples for the required behavior of the register file

4 Level 1: Pipeline [8 Points]

In this assignment, the first version of the pipeline shall be implemented. The pipeline shall be able to execute code, though without resolving any hazards in the pipeline. This means that the results of operations are not available until two cycles later, and that branches have a three-cycle branch delay. This means that the three instructions following the branch instruction are executed, regardless of whether the branch is taken or not.

The pipeline is a classic 5-stage pipeline design, consisting of fetch, decode, execute, memory, and write-back stages.

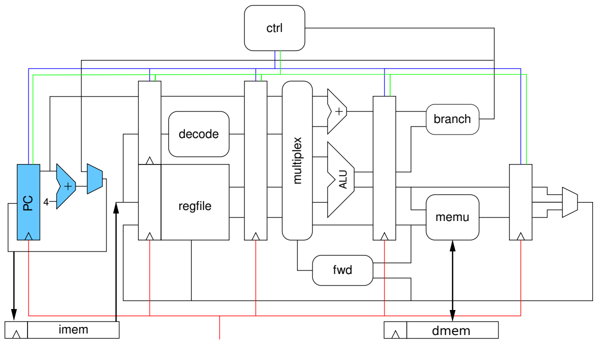
Hint: When implementing the (pipeline) registers, refer to the illustration at the start of each subsection to add them at the correct place.

Evaluation

The assignment will be tested with testbenches, which check the correctness of the behavior at the memory interface for a given content of the instruction memory. Note that this means that testing is only possible if memory operations are implemented. Points will be granted if the design passes the test suites.

4.1 Fetch

`fetch.vhd`



Description

In the fetch stage, the instruction memory is read, and the next value of the program counter is computed. Table 4.1 shows the interface of the fetch stage. `clk` and `reset` have their usual meaning, `reset` is active low. After a reset, the fetch stage shall return the instruction located at address 0 in the instruction memory (Hint: Think about an appropriate reset value for the internal PC register to achieve that). In this regard, make sure that after a reset the correct instruction is fetched and no unwanted instructions enter pipeline. Additionally, be careful that no instruction is unintentionally skipped or executed multiple times.

In case `flush` is asserted, insert a `nop` instruction into the pipeline. `stall` causes the fetch stage not to change internal registers, i.e., the program counter must not change while `stall` is asserted. If the fetch stage is not stalled and `pcsrc` is asserted, the next program counter shall be `pc_in`, if `pcsrc` is zero, it shall be the current program counter incremented by 4.

Note that the read port of the instruction memory is registered, which entails that it must be connected to the *next* program counter in order to output the instruction that corresponds to the current program counter register. The program counter is also passed on to the decode stage (see Figure 2.1). Further note, that the program counter holds a byte address, while the instruction memory is accessed word-wise. The lowest two bits of the program counter – which are always zero anyways – are therefore not used to address the instruction memory. As RISC-V uses little-endian as standard byte ordering, make sure that the individual bytes of the instruction word are in the correct order when passing them to the decode stage.

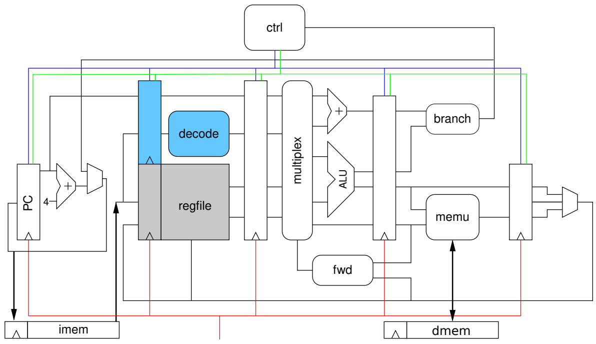
The interface used for the instruction memory is the same interface as for the data memory, however writing to the instruction memory is not required, thus some signals in the memory interface will not be used. Select appropriate default values for those signals. For this exercise it can be assumed that a read access to the instruction memory always returns the value in the next cycle and therefore, `mem_in.busy` will always be '0'. Nevertheless, connect `mem_in.busy` to `mem_busy` to be able to react to the busy signal later.

Signal	Dir.	Type	Description
clk	in	std_logic	Clock
reset	in	std_logic	Reset (low-active)
stall	in	std_logic	Stall
flush	in	std_logic	Flush
mem_busy	out	std_logic	Instruction Memory busy (towards control)
pcsrc	in	std_logic	Use pc_in or incremented program counter as new program counter
pc_in	in	PC_TYPE	New program counter
pc_out	out	PC_TYPE	Current program counter
instr	out	INSTR_TYPE	Fetched instruction
mem_out	out	MEM_OUT_TYPE	Output to memory controller
mem_in	in	MEM_IN_TYPE	Input from memory controller

Table 4.1: Fetch stage interface

4.2 Decode

`decode.vhd`



Description

The decode stage contains the register file and translates the raw instructions to signals that are used subsequently in the pipeline. More than one instruction may be mapped to an operation of a functional unit such as the ALU. For example, an addition of two registers, of a register and an immediate and calculations for memory accesses all make use of the ALU instruction `ALU_ADD`. Table 4.2 shows the interface of the decode stage. Definitions for the types `EXEC_OP_TYPE`, `MEM_OP_TYPE`, and `WB_OP_TYPE` are provided. The definitions for `EXEC_OP_TYPE`, `MEM_OP_TYPE` and `WB_OP_TYPE` are described in Tables 4.3, 4.4 and 4.5.

The signals `clk` and `reset` have their usual meaning, `reset` is active low. Asserting `stall` causes the stage not to latch inputs into its internal registers; asserting `flush` causes the unit to store a `nop` to its internal instruction register.

Figure 4.1 shows the RISC-V 32-bit instruction formats. The operations that the processor must support are shown in Table 4.7. The operation semantics in these tables are given in C-syntax. The decoding exception signal `exc_dec` shall be asserted if an instruction cannot be found in one of these tables.

The immediate calculation depending on the instruction type is shown in Figure 4.2. Although the immediate calculation seems awkward at first glance, it is designed to minimize the number of multiplexers for each bit.

Signal	Dir.	Type	Description
`clk`	in	`std_logic`	Clock
`reset`	in	`std_logic`	Reset (low-active)
`stall`	in	`std_logic`	Stall
`flush`	in	`std_logic`	Flush
`pc_in`	in	`PC_TYPE`	Program counter from fetch stage
`instr`	in	`INSTR_TYPE`	Instruction to be decoded
`reg_write`	in	`REG_WRITE_TYPE`	Information required for writing to register file
`pc_out`	out	`PC_TYPE`	Program counter for subsequent stages
`exec_op`	out	`EXEC_OP_TYPE`	Operation for execute stage
`mem_op`	out	`MEM_OP_TYPE`	Operation for memory stage
`wb_op`	out	`WB_OP_TYPE`	Operation for write-back stage
`exc_dec`	out	`std_logic`	Decoding exception

Table 4.2: Decode stage interface

Field	Type	Description
`aluop`	`ALU_OP_TYPE`	ALU operation
`alusrc1`	`std_logic`	Selecting ALU input
`alusrc2`	`std_logic`	Selecting ALU input
`alusrc3`	`std_logic`	Selecting new PC to be calculated for jmp/branch
`rs1`	`REG_ADDR_TYPE`	Specifies first register operand
`rs2`	`REG_ADDR_TYPE`	Specifies second register operand
`readdata1`	`DATA_TYPE`	Data from first register file read port
`readdata2`	`DATA_TYPE`	Data from second register file read port
`imm`	`DATA_TYPE`	Immediate value from instruction

Table 4.3: `EXEC_OP_TYPE` fields

Field	Type	Description
`branch`	`BRANCH_TYPE`	Branch operation
`mem`	`MEMU_OP_TYPE`	Operation for memory unit

Table 4.4: `MEM_OP_TYPE` fields

In Table 4.7, apart from C syntax, the following symbols are used:

$\emptyset$	Unsigned or zero-extended value
$\pm$	Signed or sign-extended value
$r_{a:b}$	Bits a to b of register r
DMEM[a]	Value at memory address a

The value `pc` corresponds to the value of the program counter as it is passed on from the fetch stage, i.e., it corresponds to the address of the currently executed instruction.

Field	Type	Description
`rd`	`REG_ADR_TYPE`	Address of register to be written to
`write`	`std_logic`	Write to register
`src`	`WBSRC_TYPE`	Source of data to be written to the register file

Table 4.5: `WB_OP_TYPE` fields

<table>
<tr><th></th><th>31</th><th>30 25</th><th>24 21</th><th>20</th><th>19 15</th><th>14 12</th><th>11 8</th><th>7</th><th>6 0</th></tr>
<tr><td>R</td><td colspan="2">funct7</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td colspan="2">rd</td><td>opcode</td></tr>
<tr><td>I</td><td colspan="4">imm[11:0]</td><td>rs1</td><td>funct3</td><td colspan="2">rd</td><td>opcode</td></tr>
<tr><td>S</td><td colspan="2">imm[11:5]</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td colspan="2">imm[4:0]</td><td>opcode</td></tr>
<tr><td>B</td><td>imm[12]</td><td>imm[10:5]</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td>imm[4:1]</td><td>imm[11]</td><td>opcode</td></tr>
<tr><td>U</td><td colspan="6">imm[31:12]</td><td colspan="2">rd</td><td>opcode</td></tr>
<tr><td>J</td><td>imm[20]</td><td colspan="2">imm[10:1]</td><td>imm[11]</td><td colspan="2">imm[19:12]</td><td colspan="2">rd</td><td>opcode</td></tr>
</table>

Figure 4.1: Instruction formats

<table>
<tr><th></th><th>31</th><th>30 20</th><th>19 12</th><th>11</th><th>10 5</th><th>4 1</th><th>0</th></tr>
<tr><td>I</td><td colspan="4">inst[31]</td><td>inst[30:25]</td><td>inst[24:21]</td><td>inst[20]</td></tr>
<tr><td>S</td><td colspan="4">inst[31]</td><td>inst[30:25]</td><td>inst[11:8]</td><td>inst[7]</td></tr>
<tr><td>B</td><td colspan="3">inst[31]</td><td>inst[7]</td><td>inst[30:25]</td><td>inst[11:8]</td><td>0</td></tr>
<tr><td>U</td><td>inst[31]</td><td>inst[30:20]</td><td>inst[19:12]</td><td colspan="4">0</td></tr>
<tr><td>J</td><td colspan="2">inst[31]</td><td>inst[19:12]</td><td>inst[20]</td><td>inst[30:25]</td><td>inst[24:21]</td><td>0</td></tr>
</table>

Figure 4.2: Types of immediates

opcode	Type
`0000011`	`OPC_LOAD`
`0100011`	`OPC_STORE`
`1100011`	`OPC_BRANCH`
`1100111`	`OPC_JALR`
`1101111`	`OPC_JAL`
`0010011`	`OPC_OP_IMM`
`0110011`	`OPC_OP`
`0010111`	`OPC_AUIPC`
`0110111`	`OPC_LUI`

Table 4.6: MiRiV base opcodes

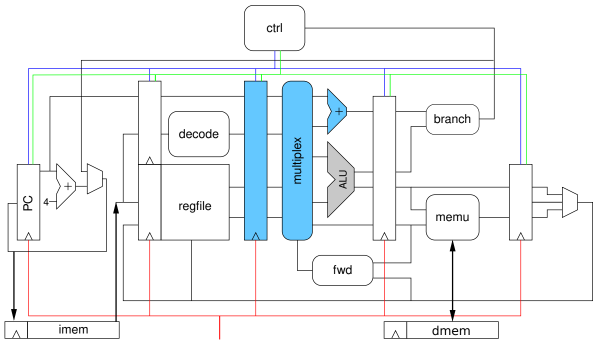
Opcode	Funct3	Funct7	Fmt	Syntax	Semantics
OPC_LUI	–	–	U	LUI rd, imm	rd=imm$^{\pm}$<<12
OPC_AUIPC	–	–	U	AUIPC rd, imm	rd=pc+(imm$^{\pm}$<<12)
OPC_JAL	–	–	J	JAL rd, imm	rd=pc+4; pc=pc+(imm$^{\pm}$<<1)
OPC_JALR	000	–	I	JALR rd,rs1,imm	rd=pc+4; pc=imm$^{\pm}$+rs1; pc[0]='0'
OPC_BRANCH	000	–	B	BEQ rs1,rs2,imm	if(rs1==rs2)pc=pc+(imm$^{\pm}$<<1)
OPC_BRANCH	001	–	B	BNE rs1,rs2,imm	if(rs1!=rs2)pc=pc+(imm$^{\pm}$<<1)
OPC_BRANCH	100	–	B	BLT rs1,rs2,imm	if(rs1$^{\pm}$<rs2$^{\pm}$)pc=pc+(imm$^{\pm}$<<1)
OPC_BRANCH	101	–	B	BGE rs1,rs2,imm	if(rs1$^{\pm}$>=rs2$^{\pm}$)pc=pc+(imm$^{\pm}$<<1)
OPC_BRANCH	110	–	B	BLTU rs1,rs2,imm	if(rs1$^{\emptyset}$<rs2$^{\emptyset}$)pc=pc+(imm$^{\pm}$<<1)
OPC_BRANCH	111	–	B	BGEU rs1,rs2,imm	if(rs1$^{\emptyset}$>=rs2$^{\emptyset}$)pc=pc+(imm$^{\pm}$<<1)
OPC_LOAD	000	–	I	LB rd,rs1,imm	rd=(int8_t)DMEM[rs1+imm$^{\pm}$]
OPC_LOAD	001	–	I	LH rd,rs1,imm	rd=(int16_t)DMEM[rs1+imm$^{\pm}$]
OPC_LOAD	010	–	I	LW rd,rs1,imm	rd=(int32_t)DMEM[rs1+imm$^{\pm}$]
OPC_LOAD	100	–	I	LBU rd,rs1,imm	rd=(uint8_t)DMEM[rs1+imm$^{\pm}$]
OPC_LOAD	101	–	I	LHU rd,rs1,imm	rd=(uint16_t)DMEM[rs1+imm$^{\pm}$]
OPC_STORE	000	–	S	SB rs1,rs2,imm	DMEM[rs1+imm$^{\pm}$]=rs2$_{7:0}$
OPC_STORE	001	–	S	SH rs1,rs2,imm	DMEM[rs1+imm$^{\pm}$]=rs2$_{15:0}$
OPC_STORE	010	–	S	SW rs1,rs2,imm	DMEM[rs1+imm$^{\pm}$]=rs2
OPC_OP_IMM	000	–	I	ADDI rd,rs1,imm	rd=rs1+imm$^{\pm}$
OPC_OP_IMM	010	–	I	SLTI rd,rs1,imm	rd=(rs1$^{\pm}$<imm$^{\pm}$)? 1 : 0
OPC_OP_IMM	011	–	I	SLTIU[¶]rd,rs1,imm	rd=(rs1$^{\emptyset}$<(imm$^{\pm}$)$^{\emptyset}$)? 1 : 0
OPC_OP_IMM	100	–	I	XORI rd,rs1,imm	rd=rs1^imm$^{\pm}$
OPC_OP_IMM	110	–	I	ORI rd,rs1,imm	rd=rs1|imm$^{\pm}$
OPC_OP_IMM	111	–	I	ANDI rd,rs1,imm	rd=rs1&imm$^{\pm}$
OPC_OP_IMM	001	–	I	SLLI[†]rd,rs1,shamt	rd=rs1<<shamt
OPC_OP_IMM	101	–	I	SRLI[⋆]rd,rs1,shamt	rd=rs1$^{\emptyset}$>>shamt
OPC_OP_IMM	101	–	I	SRAI[§]rs,rs1,shamt	rd=rs1$^{\pm}$>>shamt
OPC_OP	000	0000000	R	ADD rd,rs1,rs2	rd=rs1+rs2
OPC_OP	000	0100000	R	SUB rd,rs1,rs2	rd=rs1-rs2
OPC_OP	001	0000000	R	SLL rd,rs1,rs2	rd=rs1<<rs2$_{4:0}$
OPC_OP	010	0000000	R	SLT rd,rs1,rs2	rd=(rs1$^{\pm}$<rs2$^{\pm}$)? 1 : 0
OPC_OP	011	0000000	R	SLTU rd,rs1,rs2	rd=(rs1$^{\emptyset}$<rs2$^{\emptyset}$)? 1 : 0
OPC_OP	100	0000000	R	XOR rd,rs1,rs2	rd=rs1^rs2
OPC_OP	101	0000000	R	SRL rd,rs1,rs2	rd=rs1$^{\emptyset}$>>rs2$_{4:0}$
OPC_OP	101	0100000	R	SRA rd,rs1,rs2	rd=rs1$^{\pm}$>>rs2$_{4:0}$
OPC_OP	110	0000000	R	OR rd,rs1,rs2	rd=rs1|rs2
OPC_OP	111	0000000	R	AND rd,rs1,rs2	rd=rs1&rs2
0001111	000	–	I	FENCE	nop

[†⋆§] imm[4:0]=shamt \| [†⋆] imm[10]=0 \| [§] imm[10]=1
[¶] First sign-extend the immediate, then treat the resulting value as unsigned for the comparison

Table 4.7: MiRiV instructions

4.3 Execute

`exec.vhd`



Description

The execute stage contains the ALU, and therefore "executes" the arithmetic and logic instructions. Furthermore, the ALU is used to compute the addresses for memory accesses. Also, the addition for branches relative to the program counter is computed in this stage. Table 4.8 shows the interface of the execute stage.

The signals `clk` and `reset` have their usual meaning, `reset` is active low. Asserting `stall` causes the stage not to save inputs into its internal registers; asserting `flush` causes the unit to store a `nop` to the pipeline registers.

The information from `op` coming from the decode stage is meant to be used for controlling the ALU and feeding it with the correct input values in order to produce the required result. The ALU result is passed to the next pipeline stage via `aluresult`. Note that for some instructions using only the ALU is insufficient, since multiple operations have to be performed in parallel. One example are branch instructions, where the ALU can be used to perform the comparison (with the result being provided via the `zero` flag), while the branch target address has to be calculated by a separate component.

For regular operation, information in the signals suffixed `_in` and `_out` shall be passed on to subsequent pipeline stages without being modified. The only exception for this is `pc_new_out`, which is meant to carry the branch target address which is calculated in this stage. Finally, the content to be written to memory has to be made available to the memory stage using the `wrdata` signal.

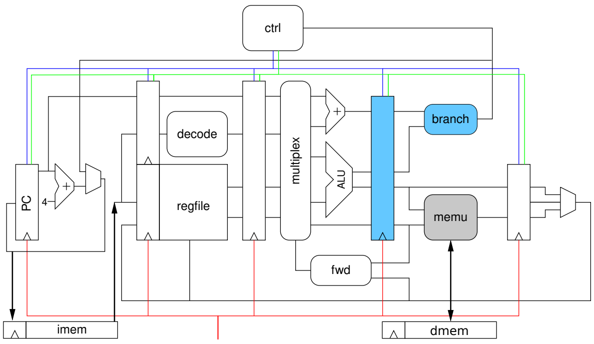
The signals `exec_op`, `reg_write_mem` and `reg_write_wr` are irrelevant for this assignment and can be ignored here. They will be used for forwarding the correct data to the ALU and for control purposes in Lab Exercise IV.

Signal	Dir.	Type	Description
`clk`	in	`std_logic`	Clock
`reset`	in	`std_logic`	Reset (low-active)
`stall`	in	`std_logic`	Stall
`flush`	in	`std_logic`	Flush
`op`	in	`EXEC_OP_TYPE`	Operation for this stage
`pc_in`	in	`PC_TYPE`	Program counter from decode stage
`pc_old_out`	in	`PC_TYPE`	Program counter for the memory stage
`pc_new_out`	in	`PC_TYPE`	Program counter (i.e., branch target) for the memory stage
`aluresult`	out	`DATA_TYPE`	Result from ALU
`wrdata`	out	`DATA_TYPE`	Value to be written to memory
`zero`	out	`std_logic`	Zero flag from ALU
`memop_in`	in	`MEM_OP_TYPE`	Memory operation from decode stage
`memop_out`	out	`MEM_OP_TYPE`	Memory operation to memory stage
`wbop_in`	in	`WB_OP_TYPE`	Write-back operation from decode stage
`wbop_out`	out	`WB_OP_TYPE`	Write-back operation to memory stage
`exec_op`	out	`EXEC_OP_TYPE`	Operation of this stage to `ctrl`
`reg_write_mem`	in	`REG_WRITE_TYPE`	Register to be written by current instr. in memory stage (for `fwd`)
`reg_write_wr`	in	`REG_WRITE_TYPE`	Register to be written by current instr. in writeback stage (for `fwd`)

Table 4.8: Execute stage interface

4.4 Memory

`mem.vhd`



Description

Most of the data memory-related functionality is already provided by the memory unit implemented earlier. Therefore, the further data memory-related implementation for this stage mainly consists of registering the inputs and passing them to the memory unit. The interface for this stage is shown in Table 4.9.

Despite its name, the memory stage does not only contain the memory unit, but is also used to evaluate and pass on the branch decision (taken/not taken via `pcsrc`) as well as the target address of the branch (via `pc_new_out`) to the fetch stage.

The signals `clk` and `reset` have their usual meaning, `reset` is active low. Asserting `flush` causes the unit to store nops to the pipeline registers. Asserting `stall` causes the stage not to latch inputs into its internal registers; additionally, neither `op.memread` nor `op.memwrite` of the memory unit may be asserted while the `stall` signal is asserted.

For regular operation, information in the signals suffixed `_in` and `_out` shall be passed on to subsequent pipeline stages without being modified.

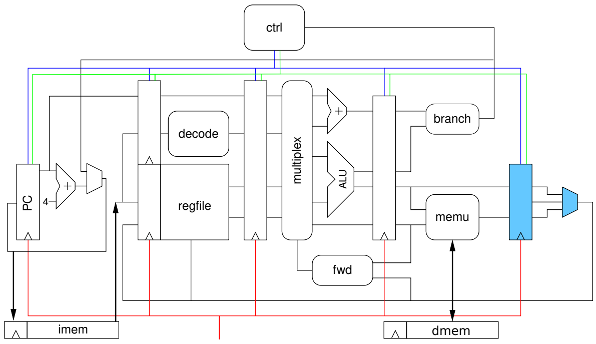
In this exercise it can be assumed that the memory read result is available at the next clock cycle. Therefore, memu's busy signal (`B`) is high for exactly on cycle per read access.

Signal	Dir.	Type	Description
`clk`	in	`std_logic`	Clock
`reset`	in	`std_logic`	Reset (low-active)
`stall`	in	`std_logic`	Stall
`flush`	in	`std_logic`	Flush
`mem_busy`	out	`std_logic`	Signaling to `ctrl` that data memory is busy
`mem_op`	in	`MEM_OP_TYPE`	Memory operation from execute stage
`wbop_in`	in	`WB_OP_TYPE`	Write-back operation from execute stage
`pc_new_in`	in	`PC_TYPE`	Program counter (i.e., branch target) from execute stage
`pc_old_in`	in	`PC_TYPE`	Program counter from execute stage
`aluresult_in`	in	`DATA_TYPE`	Result from ALU from execute stage
`wrdata`	in	`DATA_TYPE`	Data to be written to memory
`zero`	in	`std_logic`	Zero flag from ALU
`reg_write`	out	`REG_WRITE_TYPE`	Register to be written by current instruction (for `fwd`)
`pc_new_out`	out	`PC_TYPE`	Program counter (i.e., branch target) to fetch stage
`pcsrc`	out	`std_logic`	Asserted if a branch is to be executed; to fetch stage
`wbop_out`	out	`WB_OP_TYPE`	Write-back operation to writeback stage
`pc_old_out`	out	`PC_TYPE`	Program counter to writeback stage
`aluresult_out`	out	`DATA_TYPE`	Result from ALU to writeback stage
`memresult`	out	`DATA_TYPE`	Result of memory load to writeback stage
`mem_out`	out	`MEM_OUT_TYPE`	Memory operation sent to outside the pipeline
`mem_data`	in	`MEM_IN_TYPE`	Memory load result received from outside the pipeline
`exc_load`	out	`std_logic`	Load exception
`exc_store`	out	`std_logic`	Store exception

Table 4.9: Memory stage interface

4.5 Write-Back

`wb.vhd`



Description

The purpose of the write-back stage is to select between the result from the ALU, the result from a memory load or the PC and to relax the critical path(s) in the pipeline. Table 4.10 shows its interface.

Signal	Dir.	Type	Description
`clk`	in	`std_logic`	Clock
`reset`	in	`std_logic`	Reset (low-active)
`stall`	in	`std_logic`	Stall
`flush`	in	`std_logic`	Flush
`op`	in	`WB_OP_TYPE`	Write-back operation from memory stage
`aluresult`	in	`DATA_TYPE`	Result from ALU from memory stage
`memresult`	in	`DATA_TYPE`	Result from memory load for memory stage
`pc_old_in`	in	`PC_TYPE`	Program counter
`reg_write`	out	`REG_WRITE_TYPE`	Register to be written by current instruction (for decode stage and `fwd`)

Table 4.10: Write-back stage interface

4.6 Pipeline

`pipeline.vhd`

Description

The individual pipeline stages described above shall be connected to form a pipeline. The interface of the pipeline is shown in Table 4.11. The `clk` and `reset` signals have their usual meaning, `reset` is active low. If `mem_busy` from the fetch stage or memory stage is asserted, the pipeline shall be stalled. As the `ctrl` unit is not yet implemented, these two signals should be passed to all pipeline stages to stall them if required. As the pipeline in its current state does not resolve *any* hazards, the `flush` signal of the individual pipeline stages can be hardwired to `'0'`.

Signal	Dir.	Type	Description
`clk`	in	`std_logic`	Clock
`reset`	in	`std_logic`	Reset (low-active)
`mem_i_out`	out	`MEM_OUT_TYPE`	Interface from the pipeline to the instruction memory
`mem_i_in`	in	`MEM_IN_TYPE`	Interface from the instruction memory to the pipeline
`mem_d_out`	out	`MEM_OUT_TYPE`	Interface from the pipeline to the data memory
`mem_d_in`	in	`MEM_IN_TYPE`	Interface from the data memory to the pipeline

Table 4.11: Pipeline interface

The pipeline should now be able to execute sequences of assembly code. As hazards are not resolved, the results from operations only become available two instructions later. Also, branches require a three-cycle branch delay. The assembler code shown in Listing 1 shows an endless loop that stores the numbers 0, 1, 2, . . . to address 16. Note that after initializing or incrementing register `x5` two `nop` instructions are necessary for correct operation.

```
        addi x5, x0, 0
        nop
        nop
loop:
        addi x5, x5, 1
        nop
        nop
        sw x5, 16(x0)
        jal x0, loop
        nop
        nop
        nop
```

Listing 1: Assembler example without forwarding (see `submission.S`)

5 Template

5.1 Overview

The provided template gives you a starting point for your implementation and significantly reduces the effort for setting up your project. The template already provides all you need to compile software for your processor, simulate the processor and synthesize it for the FPGA. During this exercise you will need to develop your implementation as well as tests for (parts of) the processor.

The template contains 5 folders:

`quartus`	This folder contains the Quartus project. It contains all Quartus files as well as VHDL files that are required for synthesis only.
`sim`	This folder contains the simulation environment. It contains the testbench to simulate the processor as a whole as well as implementations of the required peripherals (UART and memory).
`software`	This folder contains the software build environment as well as some example software, which can be used to test the processor.
`test`	This folder contains test cases for the individual entities.
`vhdl`	This folder contains the code of your processor.

Note: It should not be required to change anything in the `quartus` and `sim` folder.

5.2 File Overview

In the `vhdl` folder all entities you need to implement your processor are already provided. Stick to this file structure, as otherwise submission tests will not work.

Consult the assignment description on how to implement the entities. Don't change the entity definitions as this might fail submission tests.

In addition to the entities, there are three VHDL packages provided for your convenience.

5.2.1 Memory Package

The package in file `mem_pkg.vhd` defines the memory interface. You are not supposed to change this file as your code must be compatible with this interface, otherwise simulation and synthesis will not work as expected.

5.2.2 Core Package

The package in file `core_pkg.vhd` contains definitions for fundamental types of the processor. You shall not change this definitions, but you are welcome to add further types and constants if required.

5.2.3 Operation Package

This package is in file `op_pkg.vhd`. It contains types and constants to specify the operations in various parts of the processor. Signals of those types are created in the decode stage and used in the appropriate entities later in the pipeline. Don't change existing types, as they are used in entity definitions. You are however welcome to add other types and helper functions when needed.

5.3 Template Usage

This section describes how to use the template during the development of your processor.

5.3.1 Run Tests

You are supposed to come up with your own tests and place them in the `test` folder. Create a sub-folder for each test and design the tests in a way that all tests can be run separately.

As an example on how a test can be created, one test is already provided. You are welcome to use this test as a template for further tests, but you are of course allowed to come up with your own ideas. However, each test has to contain a Makefile supporting the following two commands which are to be executed in the test's folder.

To compile all required VHDL files as well as the testbench itself, run

```
make compile
```

To execute the test after a successful compilation, run

```
make sim
```

Like the example test, your own tests are expected to print the results to the command line.

The sample test further provides a possibility to use the Questa/Modelsim GUI, which can be handy for debugging. Your tests are not required to be able to do this. To run the test in the graphical interface run:

```
make sim_gui
```

5.3.2 Software Compilation Process

The `software` folder contains two subfolders: one for assembler code and one for C code. The build process is slightly different between the two as the C compiler needs some initialization code, which is provided by the framework. For debugging your processor it is strongly recommended to write assembly code as this gives you (almost) full control on what is actually executed. Later in the development you can switch to C to try more elaborate programs.

Compilation

To compile a file called `test.S` (an upper case S is the recommended file extension for assembly code) or `test.c`, run the following commands in the appropriate folder:

```
make test.imem.mif
make test.dmem.mif
```

You can also build all files in the folder at once with

```
make all
```

For each program there are two files to be generated: one with extension `.imem.mif` and one with `.dmem.mif`. The first one contains the initialization of the instruction memory, thus your code. The second file is the initial content of the data section, which can also be empty. The generated files can both be used in the simulation environment as well as on the FPGA.

Library Functions

There are some library functions to be used in your C programs. They can be found in `util.h`. Use those functions to write and read from the terminal. Please note that the program is not linked against a full-featured libc. Therefore, not all standard functions are working.

Hints

- Don't use the C compiler until Exercise IV.
  The compiler generates code that is not compatible with the limitations of the first pipeline implementation as there are still hazards to be resolved in hardware.
- Use a lot of `nop` instructions.
  To debug your processor, it can help to insert 5 `nop` instructions between two meaningful instructions to add an artificial pipeline flush.
- If you are unsure how to write something in asm, let the compiler do it.
  There is a compiler option ('-S') to generate asm code instead of a binary. You can do this for short code snippets and copy the result in your asm program. There are also websites to do this.
- Don't try to access the null pointer.
  Although this might sound reasonable as memory location zero exists like any other location and there is no OS in place, according to the C standard accessing memory location zero is undefined behavior, which lets the compiler generate unexpected code.
- You can access any other memory location.
  There is no memory management and OS in place. If you want to access a memory location you can cast the address to a pointer: `(*((volatile unsigned int*)0x0000BEE0))`
- Take the memory alignment into account.
  It is illegal to perform non-aligned memory access[2]. This means if you are accessing a 2 or 4 byte word, your address must be a multiple of 2 or 4, respectively.
- RISC-V is little endian.
  If you have a look at some memory dump or compiler result, note that all 2 and 4 byte words are byte-wise reversed.
- When writing an asm program, consider adding an infinite loop at the end.
  Otherwise the processor will continue executing the instruction memory beyond your program. For C programs you can simply return from the main function, an infinite loop is added by the framework.

[2] A RISC-V implementation may handle misaligned access either in HW directly or in SW using an exception. If you are wondering what we are doing, have a look at your memu implementation.

5.3.3 Run Simulation

Once you are done with the first implementation of your processor, you can use the simulation environment to simulate the execution of a real program on the processor.

Note: This simulation environment is not a replacement for testing, as it is difficult to produce specific test vectors for individual components when simulating a real program.

Navigate inside the `sim` folder. To build your code for the simulation, run

```
make compile
```

To run a program (provided as `prog.imem.mif` and `prog.dmem.mif`) for 1 ms, execute the following command:

```
make sim IMEM=prog.imem.mif DMEM=prog.dmem.mif TIME=1ms
```

Note: You can use relative paths to the software folder to avoid copying around the files (e.g., `../software/c/foo.imem.mif`) .

The simulation environment will print all lines that are written to the UART.

5.3.4 Synthesis and Run

Although synthesis for MiRiV is rather fast, you should still get used to perform exhaustive simulations before starting an actual synthesis.

To run a synthesis go to the `quartus` folder and run

```
make all
```

At the end of the compile run there will be a report containing all warnings and errors.

During the synthesis, additional precompiled HW is added for memory access and printing. It is required not to change the PLL configuration and clock frequency to make the printing working.

If the synthesis run was successful, you can download your design using

```
make download
```

You can download a program (provided as `prog.imem.mif` and `prog.dmem.mif`) using

```
make run IMEM=prog.imem.mif DMEM=prog.dmem.mif
```

This command resets the processor, downloads the data and instruction memory and restarts the processor.

To start the Quartus GUI, you can run

```
make quartus_gui
```

5.3.5 Input and Output

The output of your processor is sent via a serial interface to the computer. To get the output on an Lab PC run

```
minicom -o -b 115200 -D /dev/ttyS0
```

6 Submission Requirements

6.1 Exercise III

The results have to be submitted via TUWEL (Deadline: 01.07.2020). Upload a **tar.gz** archive named **submission_ex3.tar.gz** containing the complete template including your additions. Further add the lab report to the archive. Make sure not to rename the 5 provided folders and clean (i.e., using `make clean`) your `quartus`, `test` and `sim` folders.

For your reference, the submitted archive should therefore have the following structure:

`submission_ex3.tar.gz`
- `report.pdf` . . . . . . Include your lab report here
- `quartus` . . . . . . The *complete* Quartus files.
- `sim` . . . . . . The *complete* files for running a simulation.
- `software` . The *complete* software folder. Additional programs for testing are recommended.
- `test` The *complete* test cases. A test for each individual component is highly recommended.
- `vhdl` . . . . . . The *complete* VHDL source code.

For a successful submission, your project must fulfill the following criteria:

- Quartus must successfully compile the project.
- Questa/Modelsim must successfully compile and simulate your processor.
- All submitted assembly programs must compile using the provided Makefiles.
- All tests must successfully compile and run without errors.

We will check these points with the provided Makefiles as described in the template description. This means that the following commands have to run without errors:

```
make -C quartus all
make -C software/asm all
make -C sim compile
make -C sim sim IMEM=../software/asm/submission.imem.mif DMEM=../
    software/asm/submission.dmem.mif
```

And for each of your tests:

```
make -C test/your_test compile
make -C test/your_test sim
```

References

[1] Altera Corporation. Quartus ii handbook version 13.1 - volume 1: Design and synthesis. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/qts/qts_qii51007.pdf`, 2013. [Online; accessed May-2020].

[2] David A. Patterson and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2017.

[3] Editors Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20190608-Base-Ratified.* RISC-V Foundation, March 2019.

Revision History

Revision	Date	Author(s)	Description
1.0	01.06.2020	TH, FK, FH, JM	Initial version

Author Abbreviations:

FH	Florian Huemer
TH	Thomas Hader
FK	Florian Kriebel
JM	Jürgen Maier

Acknowledgements

This lab assignment is based on an earlier assignment for the MIPS ISA, which was written by Wolfgang Puffitsch, with contributions of several other people, who have helped in improving it: Jomy Joseph Chelackal, Florian Ferdinand Huemer, Thomas Preindl, Jörg Rohringer, Markus Schütz and others.