Digital Design and Computer Architecture LU

Lab Exercises III and IV

Florian Huemer, Florian Kriebel
fhuemer@ecs.tuwien.ac.at, florian.kriebel@tuwien.ac.at
Department of Computer Engineering
TU Wien

Vienna, May 31, 2023

Contents

1 Introduction

This document contains the assignment for Exercise III (Level 0 and Level 1) and Exercise IV (Level 2 and Level 3). The deadlines for the exercises are:

- Exercise III: 05.06.2023, 23:55
- Exercise IV: 23.06.2023, 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 also 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 only one part of the assignment. Take a look at the protocol template for all required measurements, screenshots and questions to be answered. Make sure that all necessary details can be seen in the figures you put into your report, otherwise they will be graded with zero points.

1.1 Coding Style

Again the "VHDL Coding and Design Guidelines" apply. 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 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. The submission script will assist you to avoid mistakes. Moreover, the interfaces and record/type definitions, which are explained in this document (and can be found in the corresponding source files), must **not** be changed.

2 Overview

This section provides an overview of the architecture to be implemented: MiRiV – a minimal RISC-V implementation. It covers 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.

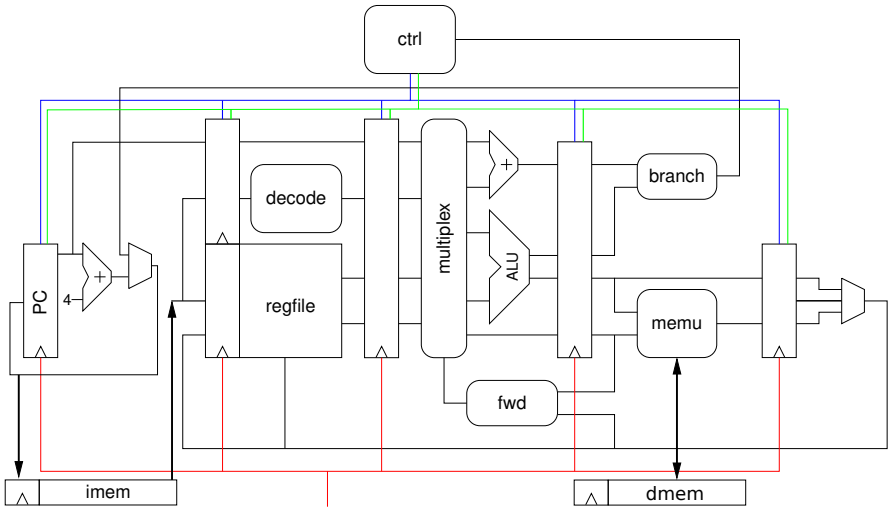


Figure 2.1: MiRiV pipeline

Figure 2.1 shows the pipeline of the processor to be implemented. It comprises 5 pipeline stages: fetch, decode, execute, memory and write-back. The data path is marked 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 the MiRiV pipeline. 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.

Hints

In the following some hints are given, which you might consider to get started:

- Read the complete assignment before starting the discussions and implementation.
- Make a plan for breaking down the work and define responsibilities and deadlines within the group (note that you also have to provide corresponding information in the report). Regular meetings might also help to discuss problems and to ensure constant progress.
- Consult the figures at the start of the description of each element to make sure the corresponding functionality is implemented in the right place.
- Start with the basic elements described in Level 0 and test them thoroughly before integrating them into the Level 1 elements.
- For the Level 1 elements, it might be useful to start with implementing the pipeline registers and to ensure that they are complete, at the right place and are working properly before adding the other logic (records might be very useful here).

- Test the individual pipeline stages before integrating them to form the complete pipeline. Finding issues in this stage of the implementation might save significant effort compared to finding those problems after the integration.
- It might be useful for all group members to come up with and contribute test cases in all stages of the implementation (even if they are not mainly responsible for a certain element).
- Once the integration is complete, test with simple assembly examples first (e.g., R-format arithmetic/logic instructions) and after ensuring their correct execution, keep adding more instruction types to the tests one at a time (e.g., I-format arithmetic/logic instructions, load instructions, store instructions, branches, etc.).
- For your tests, keep in mind that hazards are only resolved in Exercise IV (for details please refer to the corresponding lecture slides).
- In case conflicts arise within a group, contact the teaching staff as early as possible.

3 Level 0: Basic Elements [30 Points]

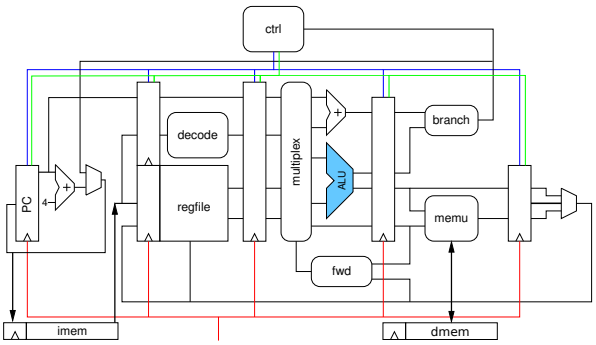
Level 0 consits of 3 tasks that implement three relatively simple hardware units. Implement the units described in this section, and write appropriate testbenches (store the testbenches in the location described in Section 7.3.1). Test the units thoroughly, as errors introduced at this stage might be 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

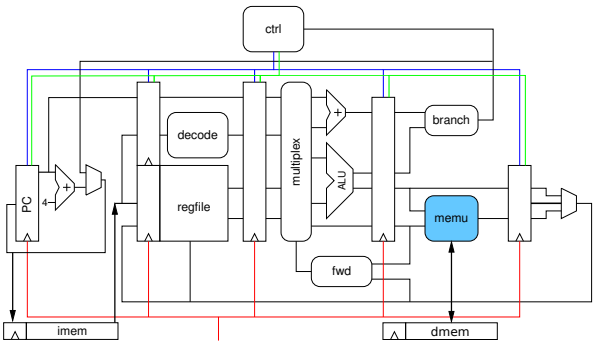
op	Z
ALU_SUB	`if A = B then Z <= '1'; else Z <= '0'; end if;`
ALU_SLT	`not R(0)`
ALU_SLTU	`not R(0)`
otherwise	`'-'`*

* 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'

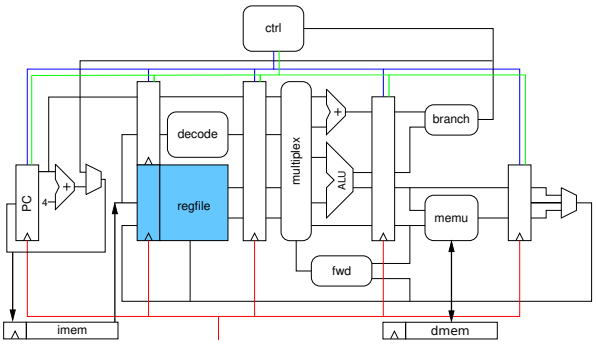
Table 3.11: Memory load exception computation

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 record the read and write addresses. The reset signal `res_n` 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 preserve 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 on memories.

Signal	Direction	Type
`clk`	in	`std_logic`
`res_n`	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

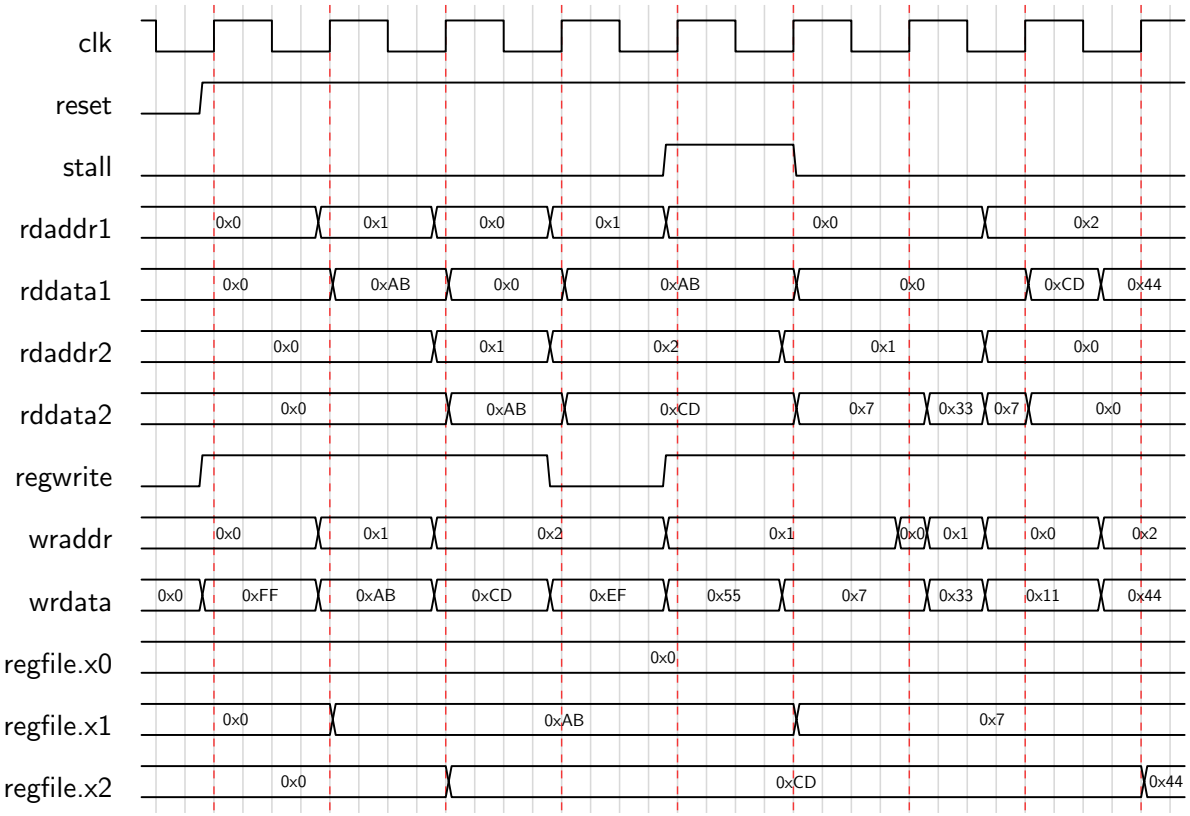


Figure 3.1: Examples for the required behavior of the register file

4 Level 1: Pipeline [70 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.

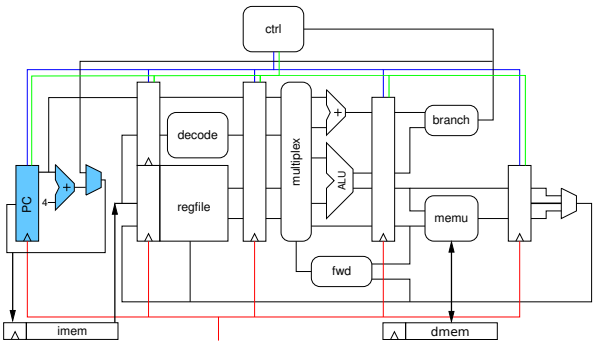
Hint: When implementing the (pipeline) registers, refer to the illustration at the start of each subsection to add them at the correct place.

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 `res_n` have their usual meaning, `res_n` 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.

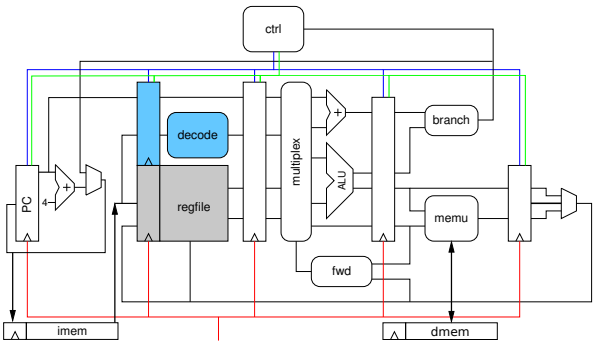
The interface used for the instruction memory is the same interface as for the data memory, however writing to the instruction memory is not required, thus some signals in the memory interface will not be used. Select appropriate default values for those signals. For this exercise it can be assumed that 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
`res_n`	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 this 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 `res_n` have their usual meaning, `res_n` is active low. Asserting `stall` causes the stage not to transfer 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
res_n	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

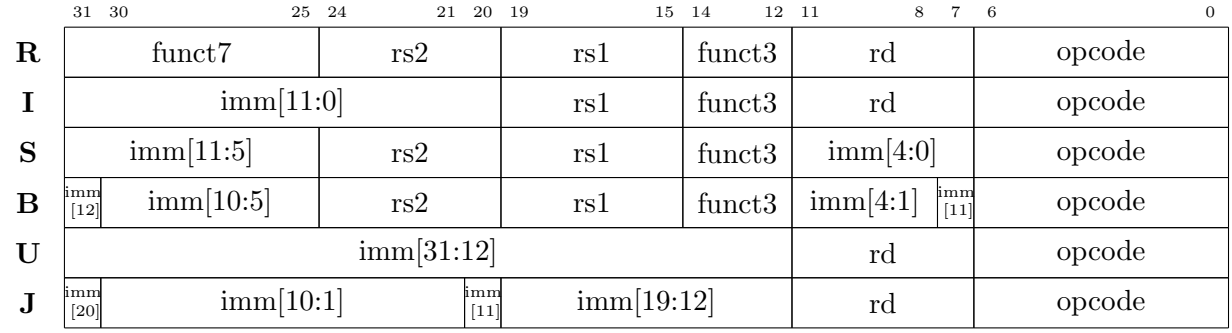


Figure 4.1: Instruction formats

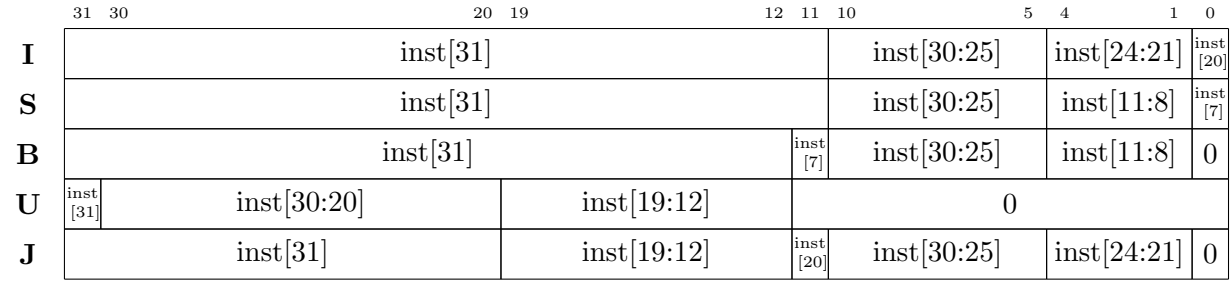


Figure 4.2: Types of immediates

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±<<12`
OPC_AUIPC	–	–	U	`AUIPC rd, imm`	`rd=pc+(imm±<<12)`
OPC_JAL	–	–	J	`JAL rd, imm`	`rd=pc+4; pc=pc+(imm±<<1)`
OPC_JALR	000	–	I	`JALR rd,rs1,imm`	`rd=pc+4; pc=imm±+rs1; pc[0]='0'`
OPC_BRANCH	000	–	B	`BEQ rs1,rs2,imm`	`if(rs1==rs2) pc=pc+(imm±<<1)`
OPC_BRANCH	001	–	B	`BNE rs1,rs2,imm`	`if(rs1!=rs2) pc=pc+(imm±<<1)`
OPC_BRANCH	100	–	B	`BLT rs1,rs2,imm`	`if(rs1±<rs2±) pc=pc+(imm±<<1)`
OPC_BRANCH	101	–	B	`BGE rs1,rs2,imm`	`if(rs1±>=rs2±) pc=pc+(imm±<<1)`
OPC_BRANCH	110	–	B	`BLTU rs1,rs2,imm`	`if(rs1∅<rs2∅) pc=pc+(imm±<<1)`
OPC_BRANCH	111	–	B	`BGEU rs1,rs2,imm`	`if(rs1∅>=rs2∅) pc=pc+(imm±<<1)`
OPC_LOAD	000	–	I	`LB rd,rs1,imm`	`rd=(int8_t) DMEM[rs1+imm±]`
OPC_LOAD	001	–	I	`LH rd,rs1,imm`	`rd=(int16_t) DMEM[rs1+imm±]`
OPC_LOAD	010	–	I	`LW rd,rs1,imm`	`rd=(int32_t) DMEM[rs1+imm±]`
OPC_LOAD	100	–	I	`LBU rd,rs1,imm`	`rd=(uint8_t) DMEM[rs1+imm±]`
OPC_LOAD	101	–	I	`LHU rd,rs1,imm`	`rd=(uint16_t) DMEM[rs1+imm±]`
OPC_STORE	000	–	S	`SB rs1,rs2,imm`	`DMEM[rs1+imm±]=rs2`$_{7:0}$
OPC_STORE	001	–	S	`SH rs1,rs2,imm`	`DMEM[rs1+imm±]=rs2`$_{15:0}$
OPC_STORE	010	–	S	`SW rs1,rs2,imm`	`DMEM[rs1+imm±]=rs2`
OPC_OP_IMM	000	–	I	`ADDI rd,rs1,imm`	`rd=rs1+imm±`
OPC_OP_IMM	010	–	I	`SLTI rd,rs1,imm`	`rd=(rs1±<imm±) ? 1 : 0`
OPC_OP_IMM	011	–	I	`SLTIU`¶ `rd,rs1,imm`	`rd=(rs1∅<(imm±)∅) ? 1 : 0`
OPC_OP_IMM	100	–	I	`XORI rd,rs1,imm`	`rd=rs1^imm±`
OPC_OP_IMM	110	–	I	`ORI rd,rs1,imm`	`rd=rs1|imm±`
OPC_OP_IMM	111	–	I	`ANDI rd,rs1,imm`	`rd=rs1&imm±`
OPC_OP_IMM	001	–	I	`SLLI`† `rd,rs1,shamt`	`rd=rs1<<shamt`
OPC_OP_IMM	101	–	I	`SRLI`⋆ `rd,rs1,shamt`	`rd=rs1∅>>shamt`
OPC_OP_IMM	101	–	I	`SRAI`§ `rs,rs1,shamt`	`rd=rs1±>>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±<rs2±) ? 1 : 0`
OPC_OP	011	0000000	R	`SLTU rd,rs1,rs2`	`rd=(rs1∅<rs2∅) ? 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∅>>rs2`$_{4:0}$
OPC_OP	101	0100000	R	`SRA rd,rs1,rs2`	`rd=rs1±>>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`

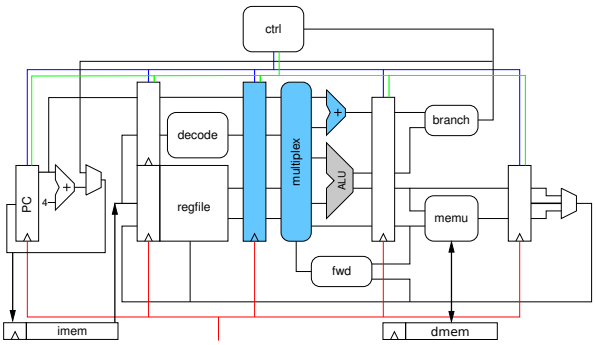
†⋆§ imm[4:0]=shamt — †⋆ imm[10]=0 — § imm[10]=1

¶ 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 `res_n` have their usual meaning, `res_n` 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.

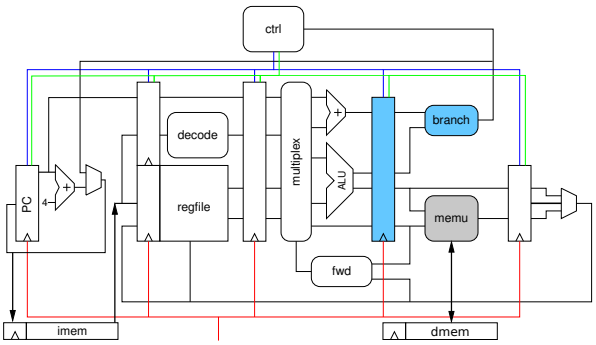
The signals `exec_op`, `reg_write_mem` and `reg_write_wr` are irrelevant for this assignment and can be ignored here. They will be used for forwarding the correct data to the ALU and for control purposes in Lab Exercise IV.

Signal	Dir.	Type	Description
clk	in	std_logic	Clock
res_n	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	out	PC_TYPE	Program counter for the memory stage
pc_new_out	out	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 `res_n` have their usual meaning, `res_n` is active low. Asserting `flush` causes the unit to store a `nop` to the pipeline registers. Asserting `stall` causes the stage not to transfer 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.

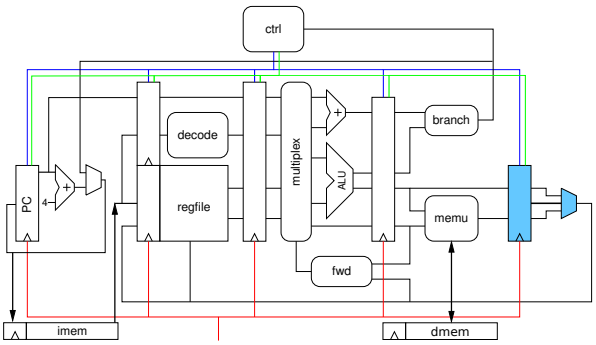
In this exercise it can be assumed that the memory read result is available at the next clock cycle. Therefore, memu's busy signal (`B`) is high for exactly one cycle per read access.

Signal	Dir.	Type	Description
clk	in	std_logic	Clock
res_n	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_in	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
`res_n`	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 `res_n` signals have their usual meaning, `res_n` 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
`res_n`	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 1, 2, ... to address 16. Note that after initializing or incrementing register `x5` two `nop` instructions are necessary for correct operation.

```
1          addi x5, x0, 0
2          nop
3          nop
4  loop:
5          addi x5, x5, 1
6          nop
7          nop
8          sw x5, 16(x0)
9          jal x0, loop
10         nop
11         nop
12         nop
```

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

5 Level 2: Hazards [50 Points]

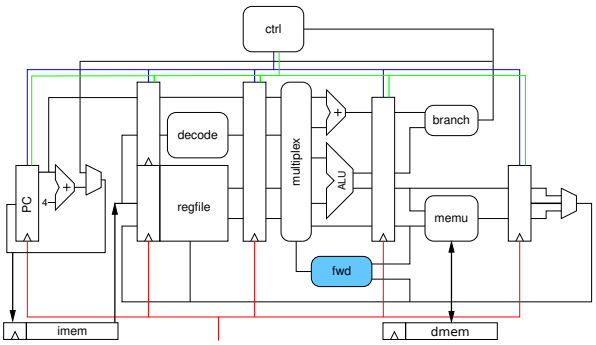
In this assignment, the data and control hazards shall be resolved. After this step your pipeline is finally ready to execute compiler-generated RISC-V code, as long as the instructions do not exceed the implemented RISC-V feature set. Thus you are then able to test the processor by compiling regular C programs and check their correct execution.

Evaluation

The correctness of the design will be assessed with testbenches, which check the correct behavior of the pipeline for given instruction memory contents. Furthermore, the design will be verified with test programs, to ensure that the design can be correctly synthesized and runs at a frequency of at least 75 MHz. Points will be awarded if the design passes the test suites and operates correctly in hardware (a correct simulation alone is not enough for all points to be awarded).

5.1 Forwarding

`fwd.vhd`



Description

When executing the sequence of instructions in Listing 2, the `and` instruction uses the results of the two preceding instructions. However, these results are not available from the register file when the `and` reaches the execute stage. The value of register `x1` is still in the write-back stage, while the value of register `x2` is in the memory stage. For correct operation, these values must be *forwarded* to the execute stage. While forwarding increases the complexity of a pipeline, it is usually more efficient to resolve this hazard in hardware than by having the compiler reorder code and insert `nop` instructions where necessary.

Implement a forwarding unit, which compares information from the execute stage, the memory stage, and the write-back stage and decides for a single input read register address whether forwarding is necessary (Note that therefore the component has to be instantiated twice). The inputs `reg_write_mem` and `reg_write_wb` provide (a) the information if the instruction in the memory and write-back stages write to a register and if they do (b) also the respective data and (c) the write register address. In addition, the read register address (`reg`) also has to be provided to the forwarding unit. Based on this information the output `do_fwd` indicates if forwarding is required and provides the respective data on output `val`. Finally, extend the execute stage to properly handle these signals, e.g., to forward the required value(s) to the ALU.[1]

Hint: handle writes to the `x0` register appropriately.

```
1        addi x1, x0, 7
2        addi x2, x0, 5
3        and x1, x2, x1
4        nop
5        nop
```

Listing 2: Assembly example with forwarding

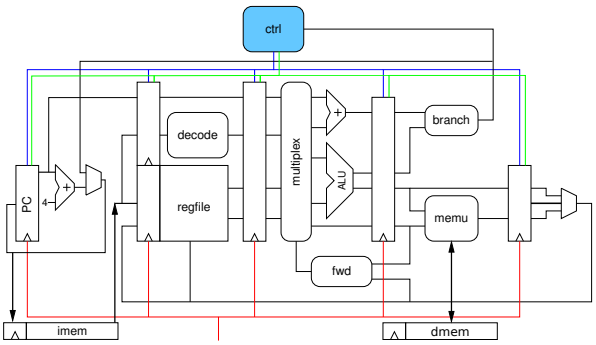
Note that forwarding a result of a memory load to an instruction executed immediately after the load is not possible. Therefore, in such situations the pipeline has to be stalled[2] until forwarding the correct data is possible. The pipeline should only be stalled if it is necessary, i.e., in case the loaded value is actually used when executing the next instruction. Not every load instruction causes a hazard.

[1] Note that it is permitted to relax the earlier constraint that updating the register file during a stall is not allowed, if required.

[2] It is recommended to implement this functionality in `ctrl.vhd`.

5.2 Branch Hazards

`ctrl.vhd`



Description

When performing a branch in the memory stage, the fetch, decode and execute stages already hold instructions that follow the branch. This means that the instructions in the fetch, decode and execute stages need to be flushed in case a branch is taken. Implement a control unit that flushes the appropriate pipeline stages when branching. When operating correctly, the assembly code in Listing 3 must not increment registers `x1`, `x2` and `x3`.

```
1 loop:   j loop
2         addi x1, x1, 1
3         addi x2, x2, 1
4         addi x3, x3, 1
```

Listing 3: Assembly example for branches

5.3 Integration

The processor is now ready to be tested in hardware. The corresponding I/O modules for communication with the outside world are already integrated. See Section 7 for further details.

Note that the timing analysis for your design must yield a maximum frequency f_{max} of at least 75 MHz.

6 Level 3: Cache [50 Points]

In modern processor implementations, CPU cores are usually much faster than the corresponding main memory. This requires a processor to stall for many cycles for each main memory access. A common strategy to cope with this performance gap is caching. Caching provides a fast but small memory that is used to facilitate copies of memory areas of the main memory that are often accessed by the processor. This enables the processor to perform fast accesses on heavily used locations while having a huge amount of cheap main memory available.

Evaluation

The design will be verified with test programs, to ensure that it can be correctly synthesized and runs at a frequency of at least 75 MHz. Points will be awarded if the design operates correctly in hardware when being integrated into your existing processor implementation.

Description

In this assignment, a cache for the data memory should be added to the processor implemented so far. It is recommended to ensure that your design is able to execute the provided applications successfully before starting with this part of the lab task.

The cache to be implemented should have the following properties:

- **Direct Mapped Cache:** Every memory location has one unique cache location.
- **Write-around On Miss:** If there is a write access to a location that is currently not in the cache, the cache should be bypassed and the write operation should be performed on the main memory directly (i.e., without transferring the data to the cache).
- **Write-back On Hit:** If there is a write operation on a cached memory location, only update the cache (and do not update the main memory) until the cache block is evicted from the cache.

If you want to implement a more advanced version of the data cache, bonus points can be earned by implementing a 2-way and 4-way set-associative cache (in addition to the direct mapped one). The details are explained in Section 6.7.

6.1 Overview

Description

The cache entity is instantiated in the `core.vhd` file. It intercepts the signals of the memory interface from the core to the devices.

The cache entity should be implemented in `cache.vhd`, which is placed in the subdirectory `cache` in the `vhdl` directory. A template for this file along with additional sub-components are provided. The corresponding interfaces are described in the following sections. The interface defined in `cache.vhd` must be adhered to, while the interface(s) of the sub-components of the cache are flexible and can be changed. In case you decide to define your own interface(s), add a description in the corresponding file(s).

Initially, `cache.vhd` only contains a "bypass implementation" (simply connecting the resepective incoming and outgoing signals without any interference) named `bypass`. The actual implementation has to be added as `impl`. In this way, testing the design with and without your cache implementation should be simplified. Note that the intended configuration (i.e., either `bypass` or `impl`) needs to be selected in `core.vhd`.

Memory interface

So far, the read access to the data memory was quite fast (i.e., the memory read result was available at the next clock cycle). As explained above this is oftentimes not the case for real processors, which are usually much faster than their corresponding main memory. Therefore, to simulate a slow memory, the `busy` signal of the memory interface is used to indicate that a memory operation is ongoing. To make the implementation easier, only read accesses are slowed down, while write accesses are handled immediately.[3] This can be tested by adapting `DMEM_DELAY` in `sim/tb/tb.vhd`.

The memory access follows the protocol shown in Figure 6.1. The read result is valid for one cycle after the `busy` signal is zero. This protocol applies for the interface between memory and cache, as well as between cache and processor.

To make your cache implementation easier, it can be assumed that the processor adheres to the protocol, so make sure your processor implementation actually does!

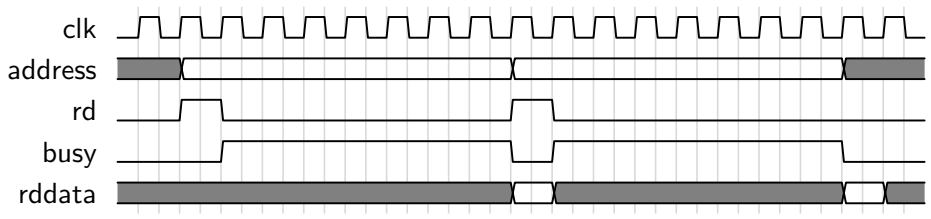


Figure 6.1: Read accesses

Note that in this lab task a cache is only implemented for the data memory.

[3] In practice there are other techniques (e.g., a write queue) in place trying to achieve immediate write accesses. Therefore, the assumption of immediate write access is not that far-fetched.

6.2 Cache

`cache.vhd`

Description

The cache entity should be implemented in `cache.vhd`. The generics and the interface are described in Table 6.1 and Table 6.2.

Generic	Type	Description
`SETS_LD`	`natural`	The ld (binary logarithm) of the number of sets (i.e., the number of sets is $2^{\texttt{SETS\_LD}}$)
`WAYS_LD`	`natural`	The ld (binary logarithm) of the number of ways (i.e., the number of ways is $2^{\texttt{WAYS\_LD}}$)
`ADDR_MASK`	`mem_address_type`	Mask for the address line to bypass the cache on device access.

Table 6.1: Cache interface generics

Signal	Direction	Type	Description
`clk`	in	`std_logic`	Clock
`res_n`	in	`std_logic`	Reset
`mem_out_cpu`	in	`mem_out_type`	Memory requests from the processor
`mem_in_cpu`	out	`mem_in_type`	Memory results to the processor
`mem_out_mem`	out	`mem_out_type`	Memory requests to the memory
`mem_in_mem`	in	`mem_in_type`	Memory results from the memory

Table 6.2: Cache interface signals

The number of cache sets is given by the parameter defined in the `generic` of the entity. Subdivide the address accordingly into tag and index. As the memory interface operates on 32 bit words, which is the same width as the processor, the size of the address offset is zero.

When a read request is issued by the processor, check if the corresponding data can be found in the cache. In case of a hit (read hit), return the cached data. In case the corresponding data are currently not cached (read miss), fetch the data from the memory, add it into the corresponding location in the cache and provide the required data to the processor.

On a write access to a cached location (write hit), update the location in the cache instead of the main memory and mark the location as being dirty. If the write destination is not cached (write miss), perform a write to the main memory directly and ignore the cache (i.e., no fetch on write).

Design a state machine that handles the different cases appropriately. The state machine could have the following states:

- `IDLE` No memory operation ongoing (i.e., no memory request from the processor).
- `READ_CACHE` Read access to the cache.
- `READ_MEM_START` First cycle of the memory read (assert the read line to the memory).

- `READ_MEM` Waiting for the memory request to finish and write the result in the cache.
- `WRITE_BACK_START` First cycle of the memory write if the evicted cache location was dirty.
- `WRITE_BACK` Finishing the write operation.

Note that the state machine does not have a state corresponding to a write hit. In this situation, writing to the cache must be handled immediately (as it would also happen in case of a write miss), i.e., it is not allowed to raise `busy` in this case. For a read hit, `busy` can be high for one clock cycle at most.

The cache must be bypassed if an access to a device (e.g., the UART) is performed. To identify such an access, the `ADDR_MASK` generic parameter is used. If an address bit is high, which is *not* set in the `ADDR_MASK`, the access shall bypass the cache (i.e., no caching of such addresses).

When implementing the reset functionality, ensure to clear only the necessary parts of the cache's management information instead of the whole cache. This is important to enable Quartus to use on-chip memory blocks during synthesis which improves performance and fitting time drastically. On-chip memory does not provide a clear (i.e., reset) functionality, therefore, all storage with clear functionality will be implemented using the flip-flop in the normal logic cells.

Note that the generic parameter `WAYS_LD` can be ignored for the implementation of the direct mapped cache.

6.3 Cache Management Information

`mgmt_st.vhd`

Description

This entity contains the complete management information required for keeping track of the cache entries and their status.

The interface is shown in Table 6.3, the generics should be used as described in Table 6.1.

Signal	Direction	Type	Description
`clk`	in	`std_logic`	Clock
`res_n`	in	`std_logic`	Reset
`index`	in	`c_index_type`	Index, i.e., the set to be accessed
`wr`	in	`std_logic`	Control updating the management information
`rd`	in	`std_logic`	Control reading the management information
`valid_in`	in	`std_logic`	Validity information of entry to be written
`dirty_in`	in	`std_logic`	Dirty information of entry to be written
`tag_in`	in	`c_tag_type`	Tag of entry to be written
`way_out`	out	`c_way_type`	Way where a hit occurred or where data has to be updated (for the advanced implementation)
`valid_out`	out	`std_logic`	Validity information of the accessed entry
`dirty_out`	out	`std_logic`	Dirty information of the accessed entry
`tag_out`	out	`c_tag_type`	Tag of the accessed entry
`hit_out`	out	`std_logic`	Hit

Table 6.3: Management interface signals

This entity should encapsulate the handling of management information, this means (among others):

- Accessing the correct set
- Deciding whether an access is a hit (tag comparison)
- Handling updates for the management information (e.g., valid, dirty, ...)
- Searching for entries in all ways (for the advanced implementation)
- Keeping track of the replacement information (for the advanced implementation)

Note that the meaning of reading (i.e., `rd`) and writing (i.e., `wr`) the management information differs from a read and write access to the memory system. For example, when writing to the memory system (store), it is first required to check if the corresponding entry is present and valid in the cache (i.e., read access to management information). Afterwards, in case of a write hit, the management information might have to be updated (dirty flag). Similarly, a read to the memory system (load) requires finding out if the corresponding entry is present and valid in the cache and might afterwards require a write to the management information in case an entry has to be evicted.

The constants, type definitions, etc. can be found in `cache_pkg.vhd`.

6.4 Cache Management Information Per Way

`mgmt_st_1w.vhd`

Description

This entity contains the management information required for one way. It is mainly a storage for saving/accessing the management information. Implement this storage using registers. It is required to be able to reset the management information stored, as otherwise old values might be accessed from the cache although the processor was reset.

The interface is shown in Table 6.4, the generics should be used as described in Table 6.1.

Signal	Direction	Type	Description
`clk`	in	`std_logic`	Clock
`res_n`	in	`std_logic`	Reset
`index`	in	`c_index_type`	Index, i.e., the set to be accessed
`we`	in	`std_logic`	Control updating the management information
`we_repl`	in	`std_logic`	Control updating the replacement information
`mgmt_info_in`	in	`c_mgmt_info`	For updating the management information
`mgmt_info_out`	out	`c_mgmt_info`	For receiving the stored management information

Table 6.4: Management interface signals per way

6.5 Cache Data

`data_st.vhd`

Description

This entity contains the data storage required for all cache entries.

The interface is shown in Table 6.5, the generics should be used as described in Table 6.1.

Signal	Direction	Type	Description
`clk`	in	`std_logic`	Clock
`we`	in	`std_logic`	Control updating the stored data
`rd`	in	`std_logic`	Control reading the stored data
`way`	in	`c_way_type`	Way to be accessed (for the advanced implementation)
`index`	in	`c_index_type`	Index, i.e., the set to be accessed
`byteena`	in	`mem_byteena_type`	Byte-enable signal for sub-word writes
`data_in`	in	`mem_data_type`	For updating the stored data
`data_out`	out	`mem_data_type`	For receiving the stored data

Table 6.5: Data storage interface signals

As the storage required for the data is typically larger than the one required for the management information (especially when having larger block sizes), instantiate memory blocks for the data storage. A RAM implementation (single-clock, dual-port, synchronous, new data read during write) is provided in `single_clock_rw_ram.vhd`.

Note that even when accessing only e.g., one byte and a read miss occurs, still the complete block must be brought to the cache. Similarly, when evicting a dirty cache entry, the complete block needs to be written to the data memory.

6.6 Cache Data For One Way

`data_st_1w.vhd`

Description

This entity contains the data storage required for the cache entries of one way.

The interface is shown in Table 6.6, the generics should be used as described in Table 6.1.

Signal	Direction	Type	Description
`clk`	in	`std_logic`	Clock
`we`	in	`std_logic`	Control updating the stored data
`rd`	in	`std_logic`	Control reading the stored data
`index`	in	`c_index_type`	Index, i.e., the set to be accessed
`byteena`	in	`mem_byteena_type`	Byte-enable signal for sub-word writes
`data_in`	in	`mem_data_type`	For updating the stored data
`data_out`	out	`mem_data_type`	For receiving the stored data

Table 6.6: Data storage interface signals per way

Note that it is required to be able to write individual bytes of the data to the storage, while leaving other bytes in the same block untouched. Therefore, make sure the data storage you implement supports this.

6.7 2&4-way Set Associative Cache (Advanced Implementation)

This part of the assignment is an optional bonus task. If you implement this part correctly, up to **16 bonus points** will be awarded.

If you plan to implement the advanced cache, we recommend to start with a direct mapped cache, but use or design the interfaces and types in a fashion such that you can increase the number of ways later.

Hint: Use `for generate` constructs for instantiations or to perform certain operations (e.g., checking for a hit) on all ways without code duplication. You can further use `or_reduce` to combine signals you have to generate for every way (e.g., to check whether there was a hit in *any* way). Also note that multiple instances of `mgmt_st_1w` and `data_st_1w` are required when implementing the 2-way and 4-way set associative cache.

Depending on the `WAYS_LD` generic parameter generate a 1-way (i.e., direct-mapped), 2-way, or 4-way set associative cache. Any value of `WAYS_LD` beyond 2 (i.e., 4-way) is invalid and you may rise an error using an assertion.

Replacement Policy

`repl.vhd`

Description

This entity contains the replacement policy. The interface is shown in Table 6.7, use the `WAYS` generic parameter for defining the number of ways.

Signal	Direction	Type	Description
`valid_in`	in	`std_logic_vector`	Valid information of all ways of the current set
`dirty_in`	in	`std_logic_vector`	Dirty information of all ways of the current set
`replace_in`	in	`std_logic_vector`	Replacement information of all ways of the current set
`replace_out`	out	`std_logic_vector`	Updated replacement information for all ways of the current set

Table 6.7: Replacement policy interface

For a cache with multiple ways, a strategy needs to be found to decide which entry is removed from the cache, if there is no free entry for the requested index. As a replacement policy, least-recently-used (LRU) has proven to be very efficient in practical applications.

For a 2-way set associative cache LRU should be implemented. One potential implementation could be setting the replacement bit accordingly when accessing an entry. Start by using invalid entries (use lowest way ID in case multiple invalid entries are present) to avoid eviction of valid entries. If only valid entries are present, evict the one which has not been used for the longest time.

Implementing LRU for more than two ways is complicated and requires more management information. Therefore, a simplified strategy should be used for the 4-way cache. For this, the four ways are partitioned into two groups of two (grouping way0&1 and way2&3). As a first priority, invalid entries should be used irrespective of the group to avoid eviction of valid entries. If only valid entries are present, evict an entry from the way group which has not been used for the longest time. Within this group evicting dirty entries should be avoided.

The entity for the replacement policy should be embedded in the `mgmt_st` module.

7 Template

7.1 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 everything you need to compile software for your processor, simulate the processor and synthesize it for the target 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, i.e., 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.

7.2 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.

7.2.1 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.

7.2.2 Core Package

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

7.2.3 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.

7.3 Template Usage

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

7.3.1 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 (note that this test is meant for demonstration purposes and does not necessarily cover all required test cases). You are welcome to use this test as a template for further ones, but you are of course allowed to come up with your own ideas. However, each test has to contain a Makefile supporting the targets `compile`, `clean` and `sim`, which have the usual meaning.

The sample test further provides the possibility to use the Questa/Modelsim GUI via the `sim_gui` taget, which can be handy for debugging.

7.3.2 Software Compilation Process

The `software` folder contains two sub-folders: one for assembly 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, i.e., your compiled 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 to (and potentially read from) the UART interface. 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 a non-aligned memory access[4]. 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 assembly 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.

- To communicate with the outside world, MiRiV uses a UART, which is accessed as a memory mapped device. This interface is used by the provided example programs (i.e., `helloworld` and `md5`) to send data from the board via a serial interface to the PC. The UART uses two addresses: `0xFFFC`, which can be used to access the data to be sent or received and `0xFFF8`, which provides information about its status (e.g., if it is ready for writing or reading data). There are corresponding library functions (see `util.h` and `util.c`) to simplify sending data via the UART and to prevent you from having to deal with the corresponding details. However, the UART can also be used for testing and debugging purposes, where the code in `send_uart.S` might get you started (note: make sure to send a newline, i.e., '\n', to ensure that an output is shown).

[4] 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.

Note: In order to compile any software in the VM, you first have to copy the appropriate RISC-V compiler from the lab. Execute the following commands inside the VM.

```
1 scp -r USERNAME@ssh.tilab.tuwien.ac.at:/opt/ddca/riscv .
2 sudo mkdir -p /opt/ddca
3 sudo mv riscv /opt/ddca
```

Building the C Compiler This section is only relevant to people who want to build the C compiler used in the lab exercise by themselves. For the CentOS 7 systems used in the lab (and the VM) the following commands can be used to build the compiler.

```
1 sudo yum install autoconf automake python3 libmpc-devel mpfr-devel gmp-devel gawk bison
      ↪ flex texinfo patchutils gcc gcc-c++ zlib-devel expat-devel
2 git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
3 cd riscv-gnu-toolchain/
4 mkdir build
5 cd build
6 ../configure --prefix=/opt/ddca/riscv --with-arch=rv32i --with-abi=ilp32
7 make
```

For other Linux distributions the packages that have to be installed (line 1) may have different names, but the rest of the script should work the same.

7.3.3 Run Simulations

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 to the `sim` folder. To build your (VHDL) code and the testbench (`sim/tb/tb.vhd`) for the simulation, run

```
1 make compile
```

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

```
1 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.

7.3.4 Synthesis and Run Programs

Although synthesis for MiRiV is rather fast, you should still get used to perform extensive simulations before starting an actual synthesis. Debugging problems in hardware can be very challenging.

To run a synthesis, go to the `quartus` folder, where a preconfigured Quartus project resides and execute

```
1 make all
```

To start the Quartus GUI, you can use the `quartus_gui` target. At the end of the synthesis run there will be a report containing all warnings and errors.

During synthesis, additional precompiled hardware is added to your design, which handles memory access and printing. Do not to change the PLL configuration and clock frequency as this will break the UART connection used for printing.

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

```
1 make download
```

To run a program (provided as `prog.imem.mif` and `prog.dmem.mif`) use

```
1 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.

Finally, to receive data printed by the processor (over the UART interface) use

```
1 make minicom
```

All these commands assume that the board is directly connected to the computer the commands are executed on. In order to simplify working with the remote environment in the lab, the Makefile also provides the targets `remote_download`, `remote_run` and `remote_minicom`. To implement those targets, the Makefile makes use of the `rpa_shell.py` script. Note that the targets `remote_download` and `remote_run` should only be executed when you have already checked out (i.e., acquired) a host in the lab (because they exit after performing their task). You can do this by either running `rpa_shell.py` separately, or by using the `remote_minicom` target first. However, keep in mind that if you do so and you exit `minicom`, the connection will be terminated (i.e., you give up your host). The `remote_run` target works exactly the same as the `run` target. It automatically copies the provided `mif` files to the lab computer.

8 Submission Requirements

8.1 Exercise III

To create an archive for submission in TUWEL (Deadline: 05.06.2023) execute the `submission_exercise3` makefile target of the template we provided you with.

```
cd /path/to/ddca_ss2023/ca
make submission_exercise3
```

The makefile creates a file named `submission.tar.gz` which should contain the following information.

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

```
submission.tar.gz
├── report.pdf ..... Your lab report.
├── quartus ..... The (clean) Quartus project.
├── sim ..... All files required for running a simulation.
├── software ..... All test programs. Additional programs for testing are recommended.
├── test ..... All 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
```

8.2 Exercise IV

To create an archive for submission in TUWEL (Deadline: 23.06.2023) execute the `submission_exercise4` makefile target of the template we provided you with.

```
cd /path/to/ddca_ss2023/ca
make submission_exercise4
```

The makefile creates a file named `submission.tar.gz` which should basically have the same overall structure as for the Exercise III submission.

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 and C-programs must compile using the provided Makefiles.
- All tests must successfully compile and run without errors.
- The synthesized design must operate correctly in hardware.

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
```

Additionally, we will test the design in hardware using:

```
make -C quartus all
make -C quartus download
make -C quartus run IMEM=prog.imem.mif DMEM=prog.dmem.mif
```

References

[1] Altera Corporation. Intel Quartus Prime Pro Edition User Guide - Design Recommendations. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qpp-design-recommendations.pdf`, 2022. [Online; accessed May-2023].

[2] David A. Patterson and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann Publishers Inc., 2nd edition, 2021.

[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
2.0	30.05.2023	FK	Added Exercise IV
1.0	06.05.2023	FK	Initial version of Exercise III

Author Abbreviations:

FH Florian Huemer
FK Florian Kriebel

Acknowledgements

The main contributors to the MiRiV implementation and this exercise are Thomas Hader and Florian Kriebel. Thanks to Florian Huemer and Jürgen Maier for their input and comments.

The 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 Chelackal, Florian Huemer, Thomas Preindl, Jörg Rohringer, Markus Schütz, Thomas Polzer, Robert Najvirt and others.