Digital Design and Computer Architecture
MiMi— Minimal MIPS

Institute of Computer Engineering
Vienna University of Technology
{rnajvirt, tpolzer, lechner}@ecs.tuwien.ac.at

December 2, 2013

1. Introduction

This document describes MiMi, a minimal MIPS implementation. It is mostly binary compatible with the original MIPS implementation, as described in the seminal *Computer Architecture: A Quantitative Approach* [1] and *Computer Organization and Design* [2]. However, only a subset of the instruction set is implemented. Also, the design is a Harvard architecture, which entails that exception and interrupt handling is slightly different than in the original MIPS implementation.

Figure 1 shows the 5-stage pipeline of the processor to be implemented. It comprises five stages: fetch, decode, execute, memory and write-back. The data path is drawn in black; signals that flush a pipeline stage are blue, signals that stall the pipeline are green, and signals that refer to exceptions are red. 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.

Section 2 describes the implementation of the basic elements of the pipeline, such as the ALU and the register file. These elements are put together to form a pipeline in Section 3; Section 4 describes how hazards in the pipeline are to be resolved. Exception and interrupt handling is covered in Section 5.

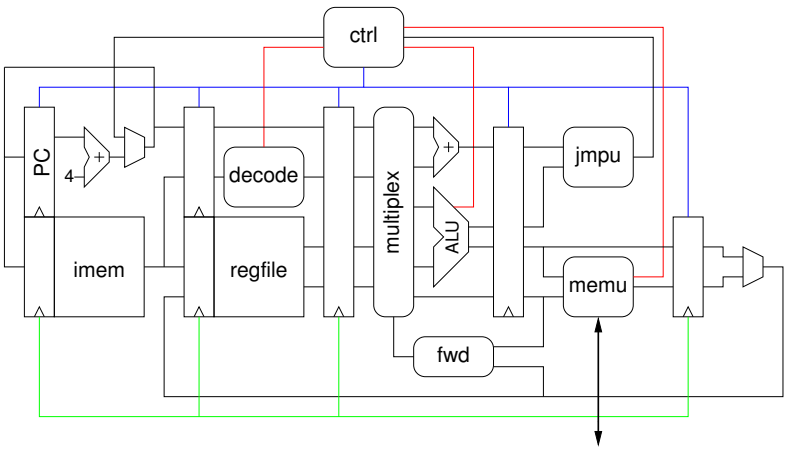


Figure 1: MiMi pipeline

The questions in the **Theory** sections do not need to be answered in order to achieve points for the practical part. However, they may be among the theoretical questions in the final exam.

Contents

List of Tables

List of Figures

Listings

2. Level 0: Basic Elements

This assignment consists of four relatively simple hardware units. Implement the units described in this section, and write appropriate test benches. Test the units thoroughly, as errors introduced at this stage might be very difficult to find in later stages.

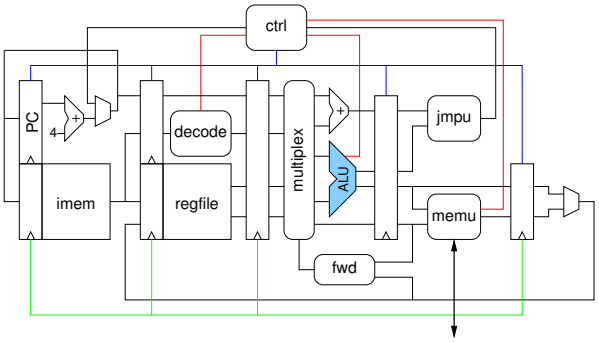
Points 4

Evaluation

The assignment will be evaluated with test benches, which thoroughly test the individual components. Points will be granted if the test benches are passed successfully. The assignment is part of lab exercise IV.

2.1. ALU

File `alu.vhd`



Description

The arithmetic logic unit (ALU) carries out – as it name suggests – arithmetic and logic operations. The interface of the ALU is described by Table 1; it shall implement the operations described in Table 2. The computation of the zero flag `Z` and the overflow flag `V` is shown in Tables 3 and 4, respectively. Note that the shift operations can be implemented conveniently with the functions `shift_left()` and `shift_right` from the package `numeric_std`.

Theory

A general comparison between two values for less-than/greater-than requires a subtraction and an appropriate evaluation of the most significant bits of the result. How many logic elements does the critical path for a comparison of two n-bit values contain asymptotically (i.e., $O(n^2)$, $O(n)$, $O(\log n)$, ...)? What about comparing for equality/inequality? Why is a comparison for less-than-zero cheap when using a two's complement representation?

Signal	Direction	Type	Width	Description
op	in	ALU_OP_TYPE	–	Operation
A	in	std_logic_vector	DATA_WIDTH	Operand A
B	in	std_logic_vector	DATA_WIDTH	Operand B
R	out	std_logic_vector	DATA_WIDTH	Result
Z	out	std_logic	–	Zero flag
V	out	std_logic	–	Overflow flag

Table 1: ALU interface

op	R
ALU_NOP	A
ALU_LUI	B **sll** 16
ALU_SLT	A < B ? 1 : 0, signed
ALU_SLTU	A < B ? 1 : 0, unsigned
ALU_SLL	B **sll** A(DATA_WIDTH_BITS-1 **downto** 0)
ALU_SRL	B **srl** A(DATA_WIDTH_BITS-1 **downto** 0)
ALU_SRA	B **sra** A(DATA_WIDTH_BITS-1 **downto** 0)
ALU_ADD	A + B
ALU_SUB	A - B
ALU_AND	A **and** B
ALU_OR	A **or** B
ALU_XOR	A **xor** B
ALU_NOR	**not** (A **or** B)

Table 2: ALU result computation

op	Z
ALU_SUB	**if** A = B **then** Z <= '1'; **else** Z <= '0'; **end if**;
otherwise	**if** A = 0 **then** Z <= '1'; **else** Z <= '0'; **end if**;

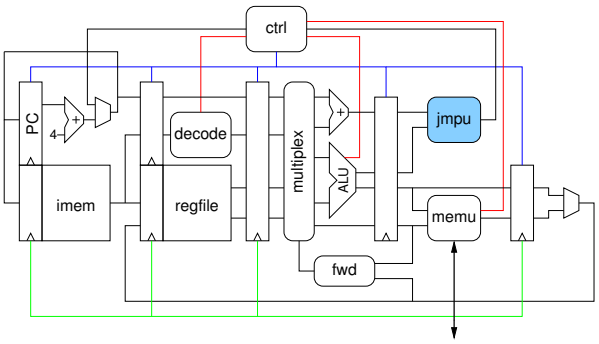
Table 3: ALU zero-flag computation

op	A	B	R	V
ALU_ADD	≥ 0	≥ 0	< 0	'1'
ALU_ADD	< 0	< 0	≥ 0	'1'
ALU_SUB	≥ 0	< 0	< 0	'1'
ALU_SUB	< 0	≥ 0	≥ 0	'1'
otherwise				'0'

Table 4: ALU overflow conditions

2.2. Jump Unit

File `jmpu.vhd`



Description

The interface of the jump unit is shown in Table 5. It shall implement the operations shown in Table 6. The zero flag `Z` corresponds to the zero flag of the ALU, while the negative flag `N` corresponds to a negative result from the ALU.

Theory

Table 6 does not contain operations for all boolean combinations of `N` and `Z`. Would an operation for `N and Z` make sense? If so, what would be the high-level comparison? If not, explain why.

Signal	Direction	Type	Description
`op`	in	`JMP_OP_TYPE`	Operation
`N`	in	`std_logic`	Negative flag
`Z`	in	`std_logic`	Zero flag
`J`	out	`std_logic`	Jump

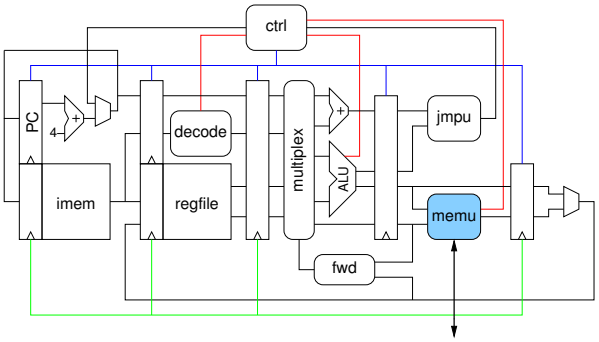
Table 5: Jump Unit interface

Operation	`J`
`JMP_NOP`	`'0'`
`JMP_JMP`	`'1'`
`JMP_BEQ`	`Z`
`JMP_BNE`	`not Z`
`JMP_BLEZ`	`N or Z`
`JMP_BGTZ`	`not (N or Z)`
`JMP_BLTZ`	`N`
`JMP_BGEZ`	`not N`

Table 6: Jump Unit operations

2.3. Memory Unit

File `memu.vhd`



Description

The memory unit is responsible for issuing memory access commands to the external interface. As the external interface is word-based, the memory unit must translate sub-word accesses. The interface of the memory unit is described in Table 7. `MEM_OP_TYPE` and `MEM_OUT_TYPE` are record types; their fields are described in Tables 8 and 9. The value of `M.address` is `A`; other outputs must be set as described below.

Table 10 shows how `M.byteena` and `M.wrdata` are computed. In that table, it is assumed that `W` consists of four bytes `DCBA`, with `D` being the most significant byte and `A` the least significant. A value `AXXX` in the last column states that the most significant `M.wrdata` is the least significant byte from `W`, and other bytes being are irrelevant and may contain arbitrary values.

How values from the external interface are translated is shown in Table 11. In that table, it is assumed that `D` consists of four bytes `DCBA`, with `D` being the most significant byte and `A` the least significant. Furthermore, `0` signifies that the byte is set to zero, and `S` that the value is sign-extended. For example, the value `SSSD` means `R` is the sign-extended most significant byte of `D`.

Tables 12 and 13 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.

Theory

The memory unit uses big-endian addressing, where the most significant byte of a word is stored at the lowest address. After storing the word `0x12345678` at address 4, what value should be returned when loading a byte from address 5? Which value should be returned for the half-word at address 6?

Signal	Direction	Type	Width	Description
`op`	in	`MEM_OP_TYPE`	–	Access type
`A`	in	`std_logic_vector`	`ADDR_WIDTH`	Address
`W`	in	`std_logic_vector`	`DATA_WIDTH`	Write data
`D`	in	`std_logic_vector`	`DATA_WIDTH`	Data from memory
`M`	out	`MEM_OUT_TYPE`	–	Interface to memory
`R`	out	`std_logic_vector`	`DATA_WIDTH`	Result of memory load
`XL`	out	`std_logic`	–	Load exception
`XS`	out	`std_logic`	–	Store exception

Table 7: 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 8: MEM_OP_TYPE fields

Field	Type	Width	Description
address	std_logic_vector	ADDR_WIDTH	Address to read from or write to
rd	std_logic	–	Asserted for reads
wr	std_logic	–	Asserted for writes
byteena	std_logic_vector	4	Byte-enable signal for sub-word writes
wrdata	std_logic_vector	DATA_WIDTH	Data to be written

Table 9: MEM_OUT_TYPE fields

Operation	A(1 **downto** 0)	M.byteena	M.wrdata
MEM_B | MEM_BU	"00"	"1000"	AXXX
	"01"	"0100"	XAXX
	"10"	"0010"	XXAX
	"11"	"0001"	XXXA
MEM_H | MEM_HU	"00"	"1100"	BAXX
	"01"	"1100"	BAXX
	"10"	"0011"	XXBA
	"11"	"0011"	XXBA
MEM_W	"00"	"1111"	DCBA
	"01"	"1111"	DCBA
	"10"	"1111"	DCBA
	"11"	"1111"	DCBA

Table 10: Computation of M.byteena and M.wrdata, W = DCBA

Operation	A(1 **downto** 0)	R
MEM_B	"00"	SSSD
	"01"	SSSC
	"10"	SSSB
	"11"	SSSA
MEM_BU	"00"	000D
	"01"	000C
	"10"	000B
	"11"	000A
MEM_H	"00"	SSDC
	"01"	SSDC
	"10"	SSBA
	"11"	SSBA
MEM_HU	"00"	00DC
	"01"	00DC
	"10"	00BA
	"11"	00BA
MEM_W	"00"	DCBA
	"01"	DCBA
	"10"	DCBA
	"11"	DCBA

Table 11: Computation of R, D = DCBA

op.memread	op.memtype	A(1 **downto** 0)	A(ADDR_WIDTH-1 **downto** 2)	XL
'1'	–	"00"	(**others** => '0')	'1'
'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'

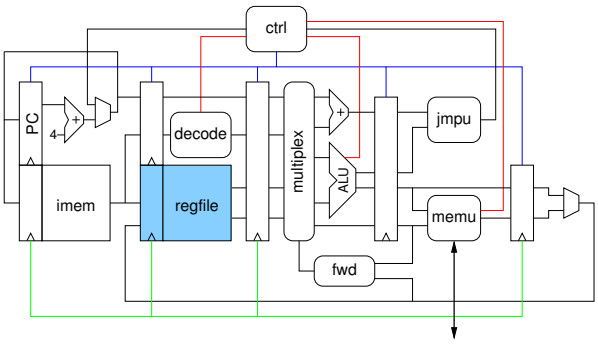
Table 12: Memory load exception computation

op.memwrite	op.memtype	A(1 **downto** 0)	A(ADDR_WIDTH-1 **downto** 2)	XS
'1'	–	"00"	(**others** => '0')	'1'
'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 13: Memory store exception computation

2.4. Register File

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. 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.

In the original MIPS implementation, reads took place on positive clock edges, while writes were performed on negative clock edges in order to forward new values through the register file. However, using both clock edges 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`.

Theory

Given memory blocks with one write- and one read-port, how can a memory with one write- and two read-ports be implemented efficiently? What is the overhead, compared to a memory with one write- and one read-port?

Signal	Direction	Type	Width
`clk`	in	`std_logic`	–
`reset`	in	`std_logic`	–
`stall`	in	`std_logic`	–
`rdaddr1`	in	`std_logic_vector`	`REG_BITS`
`rdaddr2`	in	`std_logic_vector`	`REG_BITS`
`wraddr`	in	`std_logic_vector`	`REG_BITS`
`wrdata`	in	`std_logic_vector`	`DATA_WIDTH`
`regwrite`	in	`std_logic`	–
`rddata1`	out	`std_logic_vector`	`DATA_WIDTH`
`rddata2`	out	`std_logic_vector`	`DATA_WIDTH`

Table 14: Register file interface

3. Level 1: Pipeline

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 three-cycle branch delay slots.

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

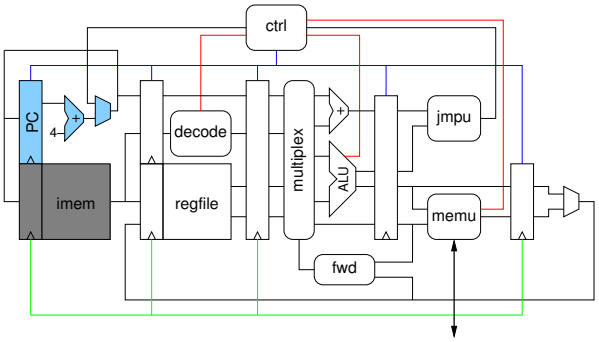
Points 5

Evaluation

The assignment will be tested with test benches, 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. The assignment is part of lab exercise IV.

3.1. Fetch

File `fetch.vhd`



Description

In the fetch stage, the instruction memory is read, and the next value of the program counter is computed. The instruction memory is located within this pipeline stage. Table 15 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. `stall` causes the fetch stage not to change internal registers, i.e., the program counter must not change while `stall` is asserted. Otherwise, if `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 *next* program counter is also passed on to the decode stage (see Figure 1). Therefore, the program counter in the decode stage does not match the address of the instruction to be decoded, but is usually already incremented by 4. Furthermore, the program counter holds a byte address, while the instruction memory is word-addressed. The lowest two bits of the program counter—which are always zero anyways—are therefore not used to address the instruction memory.

Theory

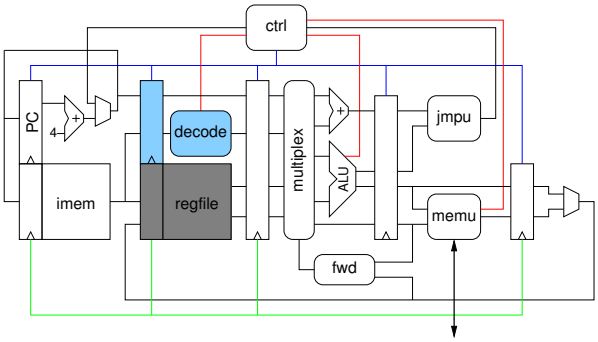
Sketch a fetch stage with variable-length instructions, where the value for the next program counter depends on the instruction that is currently fetched. Which sub-components would be on the critical path in such a fetch stage?

Signal	Dir.	Type	Width	Description
`clk`	in	`std_logic`	–	Clock
`reset`	in	`std_logic`	–	Reset
`stall`	in	`std_logic`	–	Stall
`pcsrc`	in	`std_logic`	–	Use `pc_in` or incremented program counter as new program counter
`pc_in`	in	`std_logic_vector`	`PC_WIDTH`	New program counter
`pc_out`	out	`std_logic_vector`	`PC_WIDTH`	Current program counter
`instr`	out	`std_logic_vector`	`INSTR_WIDTH`	Fetched instruction

Table 15: Fetch stage interface

3.2. Decode

File `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 function unit such as the ALU. For example, addition of two registers, of a register and an immediate and memory accesses are all mapped to the ALU instruction `ALU_ADD`. Table 16 shows the interface of the decode stage. We provide definitions for the types `EXEC_OP_TYPE`, `COP0_OP_TYPE`, `JMP_OP_TYPE`, `MEM_OP_TYPE`, and `WB_OP_TYPE`. You are however free to modify these types in order to optimize your design. The definitions for `JMP_OP_TYPE` and `MEM_OP_TYPE` are provided in Tables 6 and 8, respectively. `EXEC_OP_TYPE`, `COP0_OP_TYPE` and `WB_OP_TYPE` are described in Tables 17, 18 and 19.

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 no-op (all bits cleared) to its internal instruction register.

Figure 2 shows the MIPS instruction formats. The operations that the processor must support are shown in Tables 20, 21, 22 and 23. 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. As the coprocessor 0 is not implemented at this level, the instructions in Table 23 may be treated as no-ops.

Signal	Dir.	Type	Width	Description
`clk`	in	`std_logic`	–	Clock
`reset`	in	`std_logic`	–	Reset
`stall`	in	`std_logic`	–	Stall
`flush`	in	`std_logic`	–	Flush
`pc_in`	in	`std_logic_vector`	`PC_WIDTH`	Program counter from fetch stage
`instr`	in	`std_logic_vector`	`INSTR_WIDTH`	Instruction to be decoded
`wraddr`	in	`std_logic_vector`	`REG_BITS`	Address for writes to register file
`wrdata`	in	`std_logic_vector`	`DATA_WIDTH`	Data for writes to register file
`regwrite`	in	`std_logic`	–	Enable write to register file
`pc_out`	out	`std_logic_vector`	`PC_WIDTH`	Program counter for subsequent stages
`exec_op`	out	`EXEC_OP_TYPE`	–	Operation for execute stage
`cop0_op`	out	`COP0_OP_TYPE`	–	Operation for coprocessor 0, which handles exceptions and interrupts
`jmp_op`	out	`JMP_OP_TYPE`	–	Operation for jump unit
`mem_op`	out	`MEM_OP_TYPE`	–	Operation for memory unit
`wb_op`	out	`WB_OP_TYPE`	–	Operation for write-back stage
`exc_dec`	out	`std_logic`	–	Decoding exception

Table 16: Decode stage interface

Field	Type	Width	Description
alu_op	ALU_OP_TYPE	–	ALU operation
readdata1	std_logic_vector	DATA_WIDTH	Data from first register file read port
readdata2	std_logic_vector	DATA_WIDTH	Data from second register file read port
imm	std_logic_vector	DATA_WIDTH	Immediate value from instruction
rs	std_logic_vector	REG_BITS	Value of R-format field rs
rt	std_logic_vector	REG_BITS	Value of R-format field rt
rd	std_logic_vector	REG_BITS	Value of R-format field rd
useimm	std_logic	–	Use immediate value (for ALU or jumps)
useamt	std_logic	–	Use value of shamt field (for shifts only)
link	std_logic	–	Result is (adjusted) value of program counter
branch	std_logic	–	Branch relative to program counter
regdst	std_logic	–	Destination register is in R-format field rt or rd
cop0	std_logic	–	Result is value from coprocessor 0
ovf	std_logic	–	Pass on overflow signal from ALU

Table 17: EXEC_OP_TYPE fields

Field	Type	Width	Description
wr	std_logic	–	Write to coprocessor 0 register
addr	std_logic_vector	REG_BITS	Coprocessor 0 register to read from or write to

Table 18: COP0_OP_TYPE fields

Field	Type	Description
memtoreg	std_logic	Use ALU or memory result
regwrite	std_logic	Write to register

Table 19: WB_OP_TYPE fields

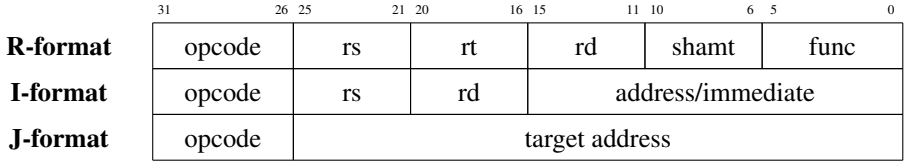


Figure 2: Instruction formats

Theory

Explain why it is beneficial to have source registers in the same position for all instruction formats, and why this is less of an issue for destination registers.

In Tables 20, 21, 22 and 23, 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
[a]	Value at memory address a

Please also note that in the column labelled "Syntax", imm18 denotes an 18-bit immediate value with its lowest two bit clear, which enables storing this value in the 16-bit field of the instruction. These values therefore have to be shifted by two bits before being used.

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 next program counter rather than the address of the currently executed instruction.

Opcode	Format	Syntax	Semantics
000000	R	–	see Table 21
000001	I	–	see Table 22
000010	J	J address	`pc = address`$^{\emptyset}$`<< 2`
000011	J	JAL address	`r31 = pc+4; pc = address`$^{\emptyset}$`<< 2`
000100	I	BEQ rd, rs, imm18	**`if`** `(rs == rd) pc += imm`$^{\pm}$`<< 2`
000101	I	BNE rd, rs, imm18	**`if`** `(rs != rd) pc += imm`$^{\pm}$`<< 2`
000110	I	BLEZ rs, imm18	**`if`** `(rs`$^{\pm}$`<= 0) pc += imm`$^{\pm}$`<< 2`
000111	I	BGTZ rs, imm18	**`if`** `(rs`$^{\pm}$`> 0) pc += imm`$^{\pm}$`<< 2`
001000	I	ADDI rd, rs, imm16	`rd = rs + imm`$^{\pm}$, overflow trap
001001	I	ADDIU rd, rs, imm16	`rd = rs + imm`$^{\pm}$
001010	I	SLTI rd, rs, imm16	`rd = (rs`$^{\pm}$`< imm`$^{\pm}$`) ? 1 : 0`
001011	I	SLTIU rd, rs, imm16	`rd = (rs`$^{\emptyset}$`< imm`$^{\emptyset}$`) ? 1 : 0`
001100	I	ANDI rd, rs, imm16	`rd = rs & imm`$^{\emptyset}$
001101	I	ORI rd, rs, imm16	`rd = rs | imm`$^{\emptyset}$
001110	I	XORI rd, rs, imm16	`rd = rs ^ imm`$^{\emptyset}$
001111	I	LUI rd, imm16	`rd = imm`$^{\emptyset}$`<< 16`
010000	R	–	see Table 23
100000	I	LB rd, imm16(rs)	`rd = (int8_t)[rs+imm`$^{\pm}$`]`
100001	I	LH rd, imm16(rs)	`rd = (int16_t)[rs+imm`$^{\pm}$`]`
100011	I	LW rd, imm16(rs)	`rd = (int32_t)[rs+imm`$^{\pm}$`]`
100100	I	LBU rd, imm16(rs)	`rd = (uint8_t)[rs+imm`$^{\pm}$`]`
100101	I	LHU rd, imm16(rs)	`rd = (uint16_t)[rs+imm`$^{\pm}$`]`
101000	I	SB rd, imm16(rs)	`(int8_t)[rs+imm`$^{\pm}$`] =` $rd_{7:0}$
101001	I	SH rd, imm16(rs)	`(int16_t)[rs+imm`$^{\pm}$`] =` $rd_{15:0}$
101011	I	SW rd, imm16(rs)	`(int32_t)[rs+imm`$^{\pm}$`] = rd`

Table 20: MiMi instructions

Func	Syntax	Semantics
000000	SLL rd, rt, shamt	rd = rt << shamt
000010	SRL rd, rt, shamt	rd = rt^{0}>> shamt
000011	SRA rd, rt, shamt	rd = $rt^{\pm}$>> shamt
000100	SLLV rd, rt, rs	rd = rt << $rs_{4:0}$
000110	SRLV rd, rt, rs	rd = rt^{0}>> $rs_{4:0}$
000111	SRAV rd, rt, rs	rd = $rt^{\pm}$>> $rs_{4:0}$
001000	JR rs	pc = rs
001001	JALR rd, rs	rd = pc+4; pc = rs
100000	ADD rd, rs, rt	rd = rs + rt, overflow trap
100001	ADDU rd, rs, rt	rd = rs + rt
100010	SUB rd, rs, rt	rd = rs - rt, overflow trap
100011	SUBU rd, rs, rt	rd = rs - rt
100100	AND rd, rs, rt	rd = rs & rt
100101	OR rd, rs, rt	rd = rs | rt
100110	XOR rd, rs, rt	rd = rs ^ rt
100111	NOR rd, rs, rt	rd = ~(rs | rt)
101010	SLT rd, rs, rt	rd = ($rs^{\pm}$< $rt^{\pm}$) ? 1 : 0
101011	SLTU rd, rs, rt	rd = (rs^{0}< rt^{0}) ? 1 : 0

Table 21: MiMi special instructions

rd	Syntax	Semantics
00000	BLTZ rs, imm18	**if** ($rs^{\pm}$< 0) pc += $imm^{\pm}$<< 2
00001	BGEZ rs, imm18	**if** ($rs^{\pm}$>= 0) pc += $imm^{\pm}$<< 2
10000	BLTZAL rs, imm18	r31 = pc+4; **if** ($rs^{\pm}$< 0) pc += $imm^{\pm}$<< 2
10001	BGEZAL rs, imm18	r31 = pc+4; **if** ($rs^{\pm}$>= 0) pc += $imm^{\pm}$<< 2

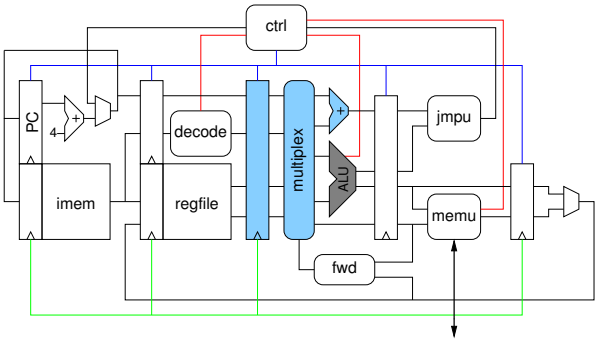
Table 22: MiMi regimm instructions

rs	Syntax	Semantics
00000	MFC0 rt, rd	rt = rd, rd register in coprocessor 0
00100	MTC0 rt, rd	rd = rt, rd register in coprocessor 0

Table 23: MiMi cop0 instructions

3.3. Execute

File `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 24 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 latch inputs into its internal registers; asserting `flush` causes the unit to store no-ops to the pipeline registers. The signal `exc_ovf` shall be asserted if the ALU asserts the overflow flag `V` and the current operation may trigger an overflow trap.

The signals `rs` and `rt` shall be instruction fields rs and rt, respectively. The signal `rd` shall be the destination register for the current operation, which may correspond to the field rd or rt, depending on the instruction format.

For most instructions, the `aluresult` signal is the result from the ALU. For the `mfc0` instruction, it shall hold the result from coprocessor 0. For instructions such as `jal` or `jalr`, `aluresult` shall contain the (adjusted) program counter. Please note that for the `bltzal` and `bgtzal` instructions, it is not possible to use the ALU to both compute the condition and adjust the program counter. Either the `zero` and `neg` flags have to be computed separately, or a separate adder to adjust the program counter has to be used.

Information in the signals suffixed `_in` and `_out` shall be passed on to subsequent pipeline stages without being modified.

The signals `forwardA`, `forwardB`, `mem_aluresult` and `wb_result` are irrelevant for this assignment and can be ignored. They will be used for forwarding the correct data to the ALU for the assignment in Section 4.

Theory

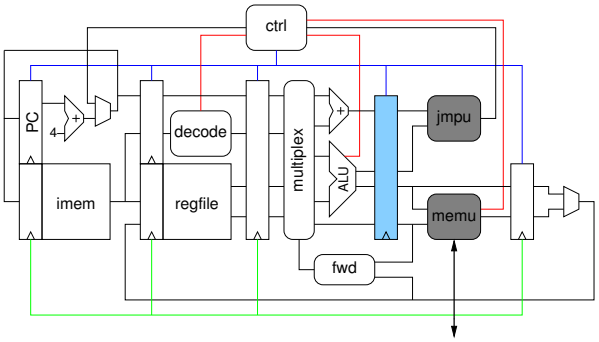
Explain why it is beneficial to multiplex the operands for a single adder over using several adders and multiplex their results. Does the benefit concern rather the performance or the size of the resulting hardware?

Signal	Dir.	Type	Width	Description
clk	in	std_logic	–	Clock
reset	in	std_logic	–	Reset
stall	in	std_logic	–	Stall
flush	in	std_logic	–	Flush
op	in	EXEC_OP_TYPE	–	Operation for this stage
rd	out	std_logic_vector	REG_BITS	Value of instruction's destination field
rs	out	std_logic_vector	REG_BITS	Value of instruction's rs field
rt	out	std_logic_vector	REG_BITS	Value of instruction's rt field
aluresult	out	std_logic_vector	DATA_WIDTH	Result from ALU or coprocessor 0, or adjusted PC
wrdata	out	std_logic_vector	DATA_WIDTH	Value to be written to memory
zero	out	std_logic	–	Zero flag from ALU
neg	out	std_logic	–	Negative result from ALU
new_pc	out	std_logic_vector	PC_WIDTH	Target address for branches
pc_in	in	std_logic_vector	PC_WIDTH	Program counter from decode stage
pc_out	out	std_logic_vector	PC_WIDTH	Program counter to memory stage
memop_in	in	MEM_OP_TYPE	–	Memory operation from decode stage
memop_out	out	MEM_OP_TYPE	–	Memory operation to memory stage
jmpop_in	in	JMP_OP_TYPE	–	Jump operation from decode stage
jmpop_out	out	JMP_OP_TYPE	–	Jump 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
forwardA	in	FWD_TYPE	–	Forwarding info for operand A
forwardB	in	FWD_TYPE	–	Forwarding info for operand B
cop0_rddata	in	std_logic_vector	DATA_WIDTH	Data from coprocessor 0
mem_aluresult	in	std_logic_vector	DATA_WIDTH	Result from ALU from previous cycle, from memory stage
wb_result	in	std_logic_vector	DATA_WIDTH	Result from ALU two cycles ago or from memory operation, from write-back stage
exc_ovf	out	std_logic	–	Overflow exception

Table 24: Execute stage interface

3.4. Memory

File `mem.vhd`



Description

Despite its name, the memory stage does not only contain the memory unit, but also the jump unit. Most of its functionality is already implemented in these two units. Therefore, the implementation for this stage mainly consists of registering the inputs and passing them on to the memory and jump unit. The interface for this stage is shown in Table 25.

The signals `clk` and `reset` have their usual meaning, `reset` is active low. Asserting `flush` causes the unit to store no-ops 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.

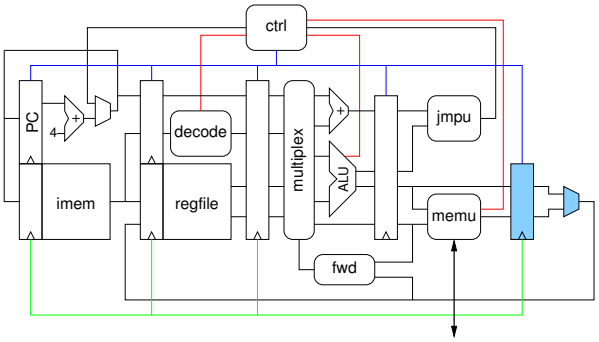
Information in the signals suffixed `_in` and `_out` shall be passed on to subsequent pipeline stages without being modified. Other signals shall be connected to the appropriate ports of the jump and memory units.

Signal	Dir.	Type	Width	Description
clk	in	std_logic	–	Clock
reset	in	std_logic	–	Reset
stall	in	std_logic	–	Stall
flush	in	std_logic	–	Flush
mem_op	in	MEM_OP_TYPE	–	Memory operation from execute stage
jmp_op	in	JMP_OP_TYPE	–	Jump operation from execute stage
wrdata	in	std_logic_vector	DATA_WIDTH	Data to be written to memory
memresult	out	std_logic_vector	DATA_WIDTH	Result of memory load
zero	in	std_logic	–	Zero flag from ALU
neg	in	std_logic	–	Negative result from ALU
pcsrc	out	std_logic	–	Asserted if a jump is to be executed
new_pc_in	in	std_logic_vector	PC_WIDTH	Jump target from execute stage
new_pc_out	out	std_logic_vector	PC_WIDTH	Jump target to fetch stage
pc_in	in	std_logic_vector	PC_WIDTH	Program counter from execute stage
pc_out	out	std_logic_vector	PC_WIDTH	Program counter to write-back stage
rd_in	in	std_logic_vector	REG_BITS	Destination register from execute stage
rd_out	out	std_logic_vector	REG_BITS	Destination register to write-back stage
aluresult_in	in	std_logic_vector	DATA_WIDTH	Result from ALU from execute stage
aluresult_out	out	std_logic_vector	DATA_WIDTH	Result from ALU to write-back stage
wbop_in	in	WB_OP_TYPE	–	Write-back operation from execute stage
wbop_out	out	WB_OP_TYPE	–	Write-back operation to write-back stage
mem_out	out	MEM_OUT_TYPE	–	Memory operation to outside the pipeline
mem_data	in	std_logic_vector	DATA_WIDTH	Memory load result from outside the pipeline
exc_load	out	std_logic	–	Load exception
exc_store	out	std_logic	–	Store exception

Table 25: Memory stage interface

3.5. Write-Back

File `wb.vhd`



Description

The purpose of the write-back stage is to select between the result from the ALU and from memory loads and to relax the critical paths in the pipeline. Table 26 shows its interface.

Signal	Dir.	Type	Width	Description
`clk`	in	`std_logic`	–	Clock
`reset`	in	`std_logic`	–	Reset
`stall`	in	`std_logic`	–	Stall
`flush`	in	`std_logic`	–	Flush
`op`	in	`WB_OP_TYPE`	–	Write-back operation from memory stage
`aluresult`	in	`std_logic_vector`	`DATA_WIDTH`	Result from ALU
`memresult`	in	`std_logic_vector`	`DATA_WIDTH`	Result from memory load
`result`	out	`std_logic_vector`	`DATA_WIDTH`	Result to register file
`regwrite`	out	`std_logic`	–	Write enable to register file
`rd_in`	in	`std_logic_vector`	`REG_BITS`	Destination register from memory stage
`rd_out`	out	`std_logic_vector`	`REG_BITS`	Destination register to register file

Table 26: Write-back stage interface

3.6. Pipeline

File `pipeline.vhd`

Description

The pipeline stages described above shall be connected to form a pipeline. The interface of the pipeline is shown in Table 27. The `clk` and `reset` signals have their usual meaning, `reset` is active low. If the field `busy` in the input signal `mem_in` is asserted, the pipeline shall be stalled. As the `ctrl` unit is not yet implemented, the appropriate signals from the memory stage shall be passed on to the fetch stage without modifying them. 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'`. The signal `intr` can be ignored for this assignment, but will be used in Section 5 to trigger external interrupts.

Signal	Dir.	Type	Width	Description
`clk`	in	`std_logic`	–	Clock
`reset`	in	`std_logic`	–	Reset
`mem_in`	in	`MEM_IN_TYPE`	–	Interface from memory to the pipeline
`mem_out`	out	`MEM_OUT_TYPE`	–	Interface from the pipeline to the memory
`intr`	in	`std_logic_vector`	`INTR_COUNT`	External interrupt lines

Table 27: 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 slot. 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 `$1` two `nop` operations are necessary for correct operation.

Listing 1: Assembler example without forwarding

```
        addi $1, $0, 0
        nop
        nop
loop:
        addi $1, $1, 1
        nop
        nop
        sw $1, 16($0)
        j loop
        nop
        nop
        nop
```

Theory

Listing 1 contains seven `nop`-instructions. How many of these instructions can be removed by reordering instructions, without changing the semantics of the program?

4. Level 2: Hazards

In this assignment, the data and control hazards shall be resolved. The pipeline should be able to execute compiler-generated MIPS code, as long as it does not contain instructions lacking from the MiMi implementation. You can test the processor by writing normal C programs and check whether they execute correctly.

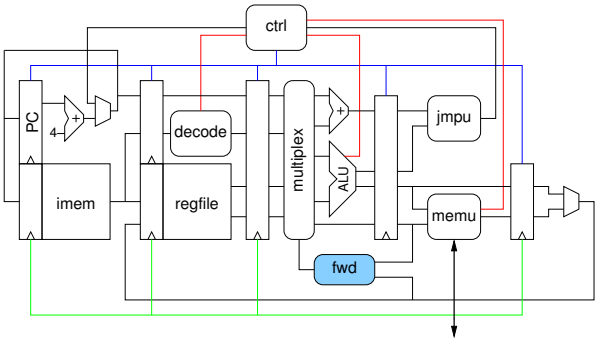
Points 6

Evaluation

The correctness of the design will be assessed with test benches, which check the correct behavior of the pipeline for given instruction memory contents. Furthermore, the design will be checked by the tutors with test programs, to ensure that the design can be correctly synthesized and runs at a frequency of at least 50 MHz. Points will be awarded if the design passes the test suites and operates correctly in hardware. The assignment is part of lab exercise V.

4.1. Forwarding

File `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 `$1` is still in the write-back stage, while the value of register `$2` 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.

Write a forwarding unit that compares information from the execute stage, the memory stage, and the write-back stage and decides whether forwarding is necessary. It shall return two signals `forwardA` and `forwardB`, which are of type `FWD_TYPE` and may have the values `FWD_NONE`, `FWD_ALU` or `FWD_WB`. `FWD_ALU` signifies that the result from the ALU that is currently stored in the memory stage shall be forwarded, `FWB_WB` signifies that the result from the write-back stage shall be forwarded. Extend the execute stage to make use of `forwardA` and `forwardB` and forward the appropriate value to the ALU. When operating correctly, the assembly code in Listing 2 must store the value 5 in register `$1`.

Listing 2: Assembler example with forwarding

```
addi $1, $0, 7
addi $2, $0, 5
and $1, $2, $1
nop
nop
```

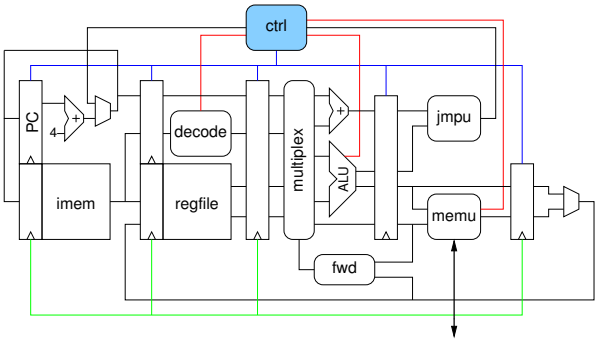
Note that it would not be possible to forward the result from a memory load to an instruction executed immediately after the load. While some MIPS incarnations stall the pipeline in such situations, MiMi, just as the original MIPS implementation, requires the compiler to avoid such situations. If rescheduling the instructions appropriately is not possible, the compiler must insert a `nop`. The instruction immediately after a load, where the result is not yet available is called *load delay slot*.

Theory

Explain why forwarding to an instruction immediately after a memory load is infeasible. Where would the critical path be if one would try to forward the result of a memory load to the ALU?

4.2. Branch Hazards

File `ctrl.vhd`



Description

When performing a branch in the memory stage, the fetch, decode and execute branch already hold instructions that follow the branch. In order to reduce the performance costs of branches, the MIPS ISA defines that the instruction immediately after the branch is executed, independent from whether the branch is taken or not. In the case of the pipeline described so far, this means that the instruction in the execution stage may finish execution, but the instructions in the fetch and decode stages need to be flushed. Implement a control unit that flushes the appropriate pipeline stages when branching.[1] When operating correctly, the assembly code in Listing 3 must increment register `$1`, but not register `$2`.

Listing 3: Assembler example with branch delay slot

```
loop:   j loop
        addi $1, $1, 1
        addi $2, $2, 1
        nop
```

Theory

A branch delay slot is a means to keep the hardware simple while reducing the cost of branches, but may increase the code size. How much is the code size increased with one-cycle branch delay slots, if 15% of the instructions are branches, and 30% of the branch delay slots can be filled by the compiler with useful instructions?

[1] The `flush` signals for some pipeline stages may not be necessary for this assignment, but will be so for the implementation of Level 3.

4.3. Integration

The processor is now almost ready to be tested in hardware. What is still missing are I/O modules in order to communicate with the outside world. The entity provided in file `mimi.vhd` wraps the processor core. It synchronizes external reset and interrupt signals and includes a PLL to adjust the external clock frequency appropriately. The entity `core` integrates the pipeline, on-chip data memory and a serial port. Use the serial port module you developed in lab exercise III for the implementation of the serial port. Synthesize the processor and test it in hardware.

The timing analysis for your design must yield a maximum frequency f_{max} of at least 50 MHz. A lower f_{max} hints at serious flaws in your design, such as not being properly pipelined. Adapt the default frequency of 75 MHz to fit your needs in `mimi.vhd`. Make sure that the entity `core` uses the generics `clk_freq` and `baud_rate` for the instantiation of the serial port. Your design will fail the automated test benches if it specifies the clock frequency or baud rate by any other means.

You are welcome to integrate I/O devices from previous assignments when synthesizing the processor. However, for the test benches, the interface of the `core` entity must not include any additional ports.

5. Level 3: Exceptions and Interrupts

In Section 4.2, the `ctrl` unit has been implemented to correctly handle branch hazards. This unit shall now be extended to implement the *coprocessor 0*, which handles exceptions and interrupts. An *exception* is a synchronous transfer of control that is triggered from within the pipeline, e.g., when trying to decode an unimplemented instruction. An *interrupt* is an asynchronous transfer of control that is triggered from outside the pipeline, e.g., when pressing a button. When resuming execution, there are in principle two possibilities: either the processor resumes at the instruction that was interrupted, or immediately after that instruction. The former alternative is mandatory for interrupts, where the execution must continue as if the interrupt had not happened. The latter alternative is useful when the instruction that caused the exception can be emulated by an exception handler.

Points 8

Evaluation

You are assigned two of the sub-assignments described in Sections 5.3, 5.4, 5.5 or 5.6. The mapping between your group number and the assignments is shown in Table 28. You can receive up to 4 points for each sub-assignment, i.e., at most 8 points in total. Solving this assignment is *optional*, but provides the opportunity to receive bonus points and improve your grade.

The correctness of the design will be assessed with test benches, which check the correct behavior of the pipeline for given instruction memory contents. Furthermore, the design will be checked by the tutors with test programs, to ensure that the design can be correctly synthesized and runs at a frequency of at least 50 MHz. Points will be awarded if the design passes the test suites and operates correctly in hardware. The assignment is part of lab exercise V.

Group	Assignment 1	Assignment 2
1	5.3	5.4
2	5.4	5.5
3	5.5	5.6
5	5.6	5.3
6	5.3	5.5
7	5.4	5.6
8	5.5	5.3
9	5.6	5.4
10	5.3	5.6
11	5.4	5.3

Table 28: Level 3 Assignments

5.1. Coprocessor 0

In order to interface the coprocessor 0, the instructions shown in Table 23 shall be implemented. The registers that shall be supported and their addresses are shown in Table 29. Note that the execution stage already contains an input port from the coprocessor 0, which can be used to move data from the coprocessor registers to the normal registers.

Splitting the `epc` and `npc` register is necessary, because exceptions may occur within a branch delay slot. Then, it would not be possible to determine whether the branch was actually taken or not, e.g. whether the next instruction is from the next higher address or the branch target. This computation has to be done within the pipeline and provided to the software via the `npc` register.

The `cause` register is described in detail in Figure 3. The bit labelled *B* shall be set to `'1'` if the exception or interrupt occurred in a branch delay slot and set to `'0'` otherwise. The field labelled *pen* represents the pending interrupts, with bit *n* in that field set if interrupt *n* is pending. The field *exc* holds the cause of the exception or interrupt. Table 30 details the exception codes to be used for that field.

The status register is shown in Figure 4. The flag labelled *I* is the interrupt enable flag. Note that in the original MIPS specification, the status register provides support for masking interrupts and the distinction of kernel- and user-mode interrupts. For the sake of simplicity, this register is reduced to a single interrupt enable flag in MiMi. The interrupts enable flag enables/disables only interrupts; exceptions may be raised even if that bit is cleared.

Address	Register	Description
`01100`	`status`	Status register
`01101`	`cause`	Cause of the exception or interrupt
`01110`	`epc`	Program counter of the instruction that caused the exception or was interrupted
`01111`	`npc`	Program counter of the next instruction

Table 29: Coprocessor 0 registers

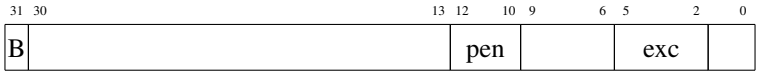


Figure 3: `cause` register

Value	Exception
`0000`	Interrupt
`0100`	Load exception
`0101`	Store exception
`1010`	Decoding exception
`1100`	Overflow exception

Table 30: Exception codes

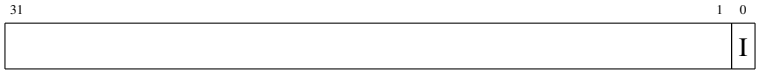


Figure 4: `status` register

5.2. General Operation

In case of an exception or interrupt, execution shall proceed at address `EXCEPTION_PC`. The exception handler located at this address saves the processor state and calls an exception handling routine. Depending on the return value of this routine, execution resumes at the interrupted instruction or the instruction to be executed after that instruction. In the latter case, this is the instruction at the address stored in the `npc` register. In the former case, the return address depends on the bit "B" in the `cause` register. If it is set (i.e., if the exception/interrupt was triggered in a branch delay slot), execution resumes at `epc-4`; otherwise, execution resumes at address `epc`.

5.3. Decode Exception

The decode exception shall be triggered if the decode stage detects an unimplemented instruction, and the respective instruction would be actually executed. The exception should therefore be suppressed if a branch is taken and the respective instruction would be flushed from the pipeline. Furthermore, if the invalid instruction is in a branch delay slot, triggering of the exception must be delayed until the correct value for the `npc` register becomes available.

5.4. Overflow Exception

If the ALU signals an overflow for an ADDI, ADD, or SUB instruction, an overflow exception shall be triggered. A combinatorial path from the adder/subtracter in the ALU to the fetch stage would result in a long critical path. Inserting registers can cut this critical path and help in achieving a reasonable maximum frequency of the overall design.

5.5. Memory Exception

A memory exception shall be triggered if the memory stage asserts the `exc_load` or `exc_store` signal. If the exception is triggered in a branch delay slot, the branch already has been executed in the previous cycle. The coprocessor therefore has to "remember" if a branch has occurred in order to compute the correct value of the `npc` register.

5.6. Interrupts

An interrupt shall be triggered if an external interrupt signal is asserted and bit 0 of the `status` register is set. Upon triggering an interrupt, this bit shall be cleared. As the interrupted instruction must always be re-executed, both the `npc` and `epc` register shall point to this instruction.

Whenever a bit in the input `intr` of the pipeline is `'1'`, the corresponding bit in the field *pen* of the `cause` register shall be set to `'1'`. This *pending* flag must remain `'1'` until overwritten by a write to the `cause` register. An interrupt is triggered if any bit in the field *pen* is set and the interrupt enable flag in the `status` register is `'1'`. It is acceptable to delay interrupts to avoid conflicts with branches.

A. Tools

Listing 4 shows a basic Makefile to compile programs for MiMi in the lab environment. The variable `CC` denotes the C compiler, `LD` the linker, `AR` the program to create library files and `OBJCOPY` the program to convert between binary representations of a program. The default flags for the compiler indicate that MiMi is compatible to the MIPS 1 ISA and that it does not contain a floating-point unit. The flags for `LD` determine the memory layout and must not be changed.

The target `lib` builds the program preamble `crt0.o` and a minimal version of the C library, `libc.a`. This target must be built explicitly before building programs. Note that it is necessary to provide an explicit rule for linking in order to link the right files (the `libc.a` in the current directory) in the right order (`crt0.o` before everything else).

The build process for MiMi is slightly more complex than for more common architectures. After having linked the program in the ELF file format, the binary needs further processing. First, the data for the instruction and data memories have to be extracted to files which end in `.imem.hex` and `.dmem.hex`, respectively. Then, these files have to be converted from the Intel HEX format to the MIF file format understood by Quartus, which is done with the `hex2mif.pl` script. The `.mif` files then need to be copied to the appropriate directory such that the correct program is synthesized into the processor.

Listing 4: Makefile fragment

```
PREFIX=/usr/mips

CC=${PREFIX}/bin/mips-elf-gcc -mips1 -msoft-float
LD=${PREFIX}/bin/mips-elf-ld -N -Ttext=0x40000000 --section-start .rodata=4
AR=${PREFIX}/bin/mips-elf-ar
OBJCOPY=${PREFIX}/bin/mips-elf-objcopy

CFLAGS=-O2 -DARCH_IS_BIG_ENDIAN=1

test.elf: test.o
        ${LD} -o $@ crt0.o $^ -L. -lc

lib: crt0.o libc.a

libc.a: exceptions.o util.o
        ${AR} rc $@ $^

%.imem.hex : %.elf
        ${OBJCOPY} -j .text -O ihex $< $@

%.dmem.hex : %.elf
        ${OBJCOPY} -R .text -O ihex $< $@

%.mif : %.hex
        ./hex2mif.pl < $< > $@
```

B. Automated Test Environment

In order to submit your source code to the automated test system, copy your design to the directory `/ddcanightly/ddcagrp<grpnr>/level<lvlnr>/`. The VHDL files should reside in the subdirectory `src`. For example, group 99 should copy their implementation of the decode stage to `/ddcanightly/ddcagrp99/level1/src/decode.vhd` to submit it to the level 1 tests.

The source code for each level must be complete; higher levels must include the source code from lower levels. The files `imem_altera.vhd`, `ocram_altera.vhd` and `serial_port_wrapper.vhd` must be identical to the provided files. Only the files that were provided to you are taken into account by the test suite. Your design will therefore not pass the test benches if it requires any additional source files.

The automated test benches create a snapshot of the `/ddcanightly` directory daily at 1:00 AM; please do not submit files around that time to avoid an inconsistent state of your source code. The test bench results are reported to the e-mail addresses stated in `/ddcanightly/ddcagrp<grpnr>/recipients`. You will receive the reports only if you enter your e-mail address in that file.

The test system is not intended to replace your own testing. In order to avoid abuse of the test system for debugging, the test reports provide only minimal information on the failed test cases. For test cases that are based on assembler or C code, you will however be provided with the source code.

In order to receive points for the exercises, the source code must be uploaded to the MyTI system before the deadline. Points will only be awarded if the source code submitted to the MyTI system passes the test suites.

C. Submission Requirements

C.1. Exercise IV

The results have to be submitted via MyTI. The deadline is December 20th, 2013, 23:59. Upload a `zip` or `tar.gz` archive containing the following items:

- Your lab protocol as PDF file.
- The complete VHDL source code for the assignments detailed in Section 2 and 3, including entities and packages provided to you.

Source code must extract to a directory named `src`. It is permissible to place the source code for Level 0 in a directory `level0/src` and the source code for Level 1 in a directory `level1/src`.

C.2. Exercise V

The results have to be submitted via MyTI. The deadline is January 23rd, 2014, 23:59. Upload a `zip` or `tar.gz` archive containing the following items:

- Your lab protocol as PDF file.
- The complete VHDL source code and the Quartus project for the assignments detailed in Section 4 and 5, including entities and packages provided to you.

Source code must extract to a directory named `src`. It is permissible to place the source code for Level 2 in a directory `level2/src` and the source code for Level 3 in a directory `level3/src`.

Acknowledgements

This document was written by Wolfgang Puffitsch. Other people, who have helped in improving it: Jomy Joseph Chelackal, Florian Ferdinand Huemer, Thomas Preindl, Jörg Rohringer, Markus Schütz and others.

References

[1] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[2] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

Revision Log

- 2011-11-29: First public version.
- 2011-12-07: Minimum f_{max} requirement, `clk_freq` and `baud_rate` generics.
  Group assignments for level 3.
  Fix typos.
- 2011-12-13: Definite version of Levels 2 and 3.
- 2011-12-16: Updated points, Level 3 is now optional.
- 2011-12-19: Clarifications for value of program counter from fetch unit.
  Fix description of branches.
  Explain why almost all stages have a `flush` signal.
- 2012-01-09: Fix description of `mfc0` and `mtc0`.
- 2012-01-13: Clarify description of pending flags and interrupts.