Digital Design and
Computer Architecture LU

Digital Design and Computer Architecture
MiMi – Minimal MIPS
Exercise IV

{fhuemer, fkriebel, jmaier}@ecs.tuwien.ac.at
Department of Computer Engineering
University of Technology Vienna

Vienna, June 1, 2019

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 1.1 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.

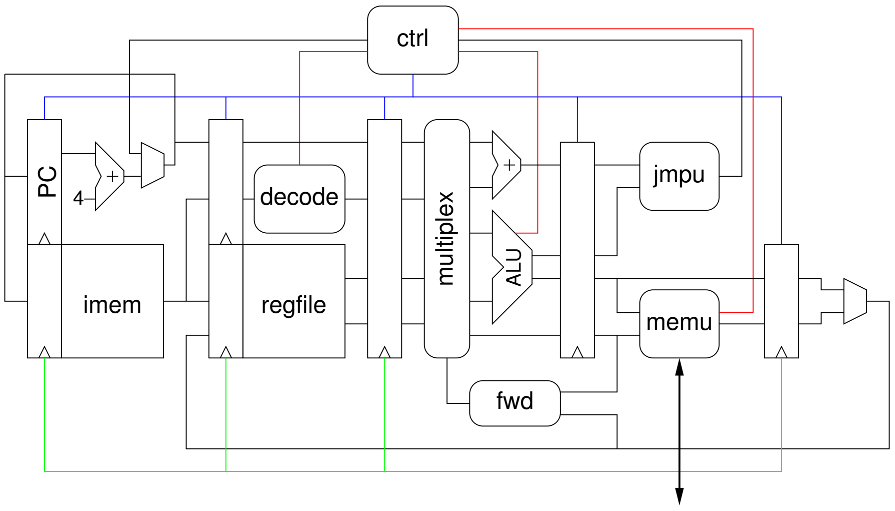


Figure 1.1: MiMi pipeline

Section 2 describes how hazards in the pipeline are to be resolved. Exception and interrupt handling is covered in Section 3.

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 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.

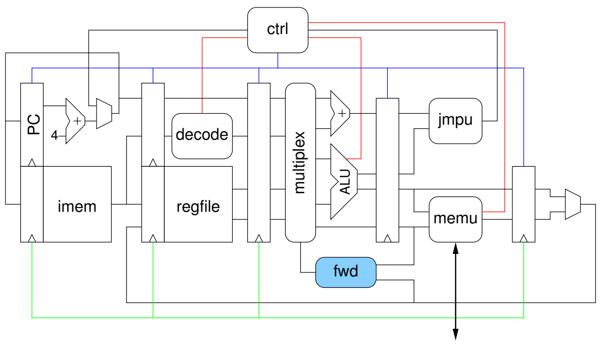
Points 6

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 IV.

2.1 Forwarding

`fwd.vhd`



Description

When executing the sequence of instructions in Listing 1, 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 1 must store the value 5 in register `$1`.

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

Listing 1: Assembler example with forwarding

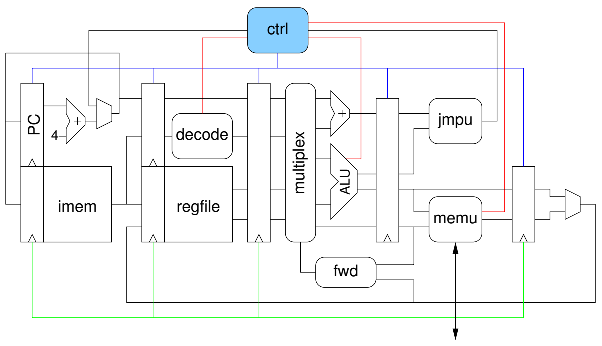
Note that it would not be possible to forward the result from a memory load to an instruction executed immediately after the load. While some MIPS incarnations stall the pipeline in such situations, MiMi, just as the original MIPS implementation, requires the compiler to avoid such situations. If rescheduling 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?

2.2 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. 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 2 must increment register `$1`, but not register `$2`.

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

Listing 2: Assembler example with branch delay slot

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.

2.3 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. 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.

3 Level 3: Exceptions and Interrupts

In Section 2.2, 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 3.3, 3.4, 3.5 or 3.6. The mapping between your group number and the assignments is shown in Table 3.1.

In case additional sub-assignments beyond the ones shown in Table 3.1 are implemented, two additional bonus points can be achieved, respectively. However, the bonus points are only awarded if the assignments shown in Table 3.1 have been implemented.

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 IV.

Group	Assignment 1	Assignment 2
2	3.3	3.4
3	3.4	3.5
5	3.5	3.6
6	3.4	3.5
7	3.6	3.3
8	3.3	3.4
10	3.4	3.6
11	3.5	3.3
12	3.6	3.4
15	3.3	3.6
18	3.3	3.5
20	3.3	3.4
21	3.5	3.6
25	3.6	3.3
42	3.6	3.4

Table 3.1: Level 3 Assignments

3.1 Coprocessor 0

In order to interface the coprocessor 0, the instructions shown in Table 3.2 shall be implemented. The registers that shall be supported and their addresses are shown in Table 3.3. 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.

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 3.2: MiMi cop0 instructions

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 3.3: Coprocessor 0 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 3.1. 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 3.4 details the exception codes to be used for that field.

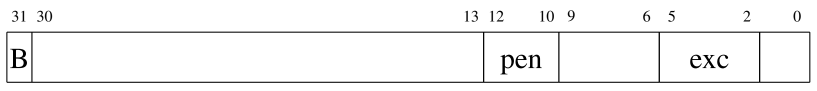


Figure 3.1: `cause` register

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

Table 3.4: Exception codes

The status register is shown in Figure 3.2. 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.

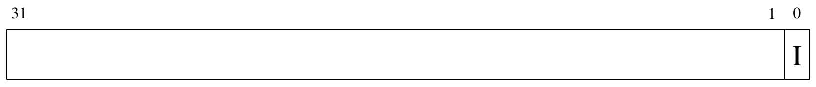


Figure 3.2: `status` register

3.2 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`.

3.3 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.

3.4 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.

3.5 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.

3.6 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 3 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.

To avoid problems with the compilation process and the generated files, it is recommended to use the setup in the lab environment.

```
PREFIX=/opt/ddca/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 < $< > $@
```

Listing 3: Makefile fragment

B Automated Test Environment

During this part of the lab course an automated test system can be used. It is offered as a service tool to guide you during the implementation process and point out certain issues in your design to help you to detect and avoid problems for higher levels.

Each group has a weekly "token budget" for 7 test runs of the automated test system. Test runs can be triggered individually (i.e., per group) upon request at any time, given the token budget has not been exceeded yet. The budget is reset at the beginning of a week, where potential tokens left over from the previous week(s) will be lost.

To make sure that a test run is still counted for the budget of the current week, corresponding requests have to be made on Sunday before 8 p.m. (as a higher number of requests might be sent at that time). Otherwise, it is not guaranteed whether the token budget of the current or the next week will be used (since test runs might be delayed until Monday). New test runs can be requested on Monday starting from 1 a.m.

Procedure:

1. 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 testbenches if it requires any additional source files.
2. To request a test run by the automated test system, an (empty) file named *TESTME* has to be created (case sensitive; create it e.g., via the command '*touch TESTME*').
3. It will be checked regularly, whether the group directories contain such a *TESTME* file. If this is the case the following steps will be executed automatically:
   - Your design will be copied to a separate test directory.
   - The automated test will be started.
   - The *TESTME* file will be removed automatically and a *TESTME-TESTS-RUNNING* file will be created. This file indicates that the automated test started to run.
4. After the test finished, the following data will be created in `ddcanightly/results/ddcagrp<ddcagrpnr>/Week<weeknr>`:
   - `budget_used.txt`: This file contains information about the token budget already used in the current week (used token budget/total token budget).
   - `failed.txt`: This file will only be created in case the token limit is exceeded and another test run was requested.
5. After the test finished, the following data will be created in `ddcanightly/results/ddcagrp<ddcagrpnr>/Week<weeknr>/<date_time>/`:
   - `ddcagrp<ddcagrpnr>_date_time.tar.gz`: A copy of the submitted design.
   - `report_level<levelnr>.txt`: A report for each test level which is currently activated.

Hints:

- Please avoid changes in the submitted files after a test run has been requested. Otherwise this might lead to an inconsistent state.
- Before requesting a test run, discuss with the other group members whether they agree in order to avoid conflicts.
- In case you have difficulties in interpreting the output of the automated test system for your group, write an e-mail to *ddcatut@ecs.tuwien.ac.at*.

IMPORTANT: 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. Therefore and to save system resources, remove `report` statements or similar code used for your own debugging purposes before submitting your implementation to the automated test system. 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 TUWEL system before the deadline. Points will only be awarded if the source code submitted to the TUWEL system passes the test suites.

C Submission Requirements

C.1 Exercise IV

The results have to be submitted via TUWEL (Deadline: 26.06.2019, 23:55). Upload a **tar.gz** archive named **submission_ex4.tar.gz** 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 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 2 in a directory **level2/src** and the source code for Level 3 in a directory **level3/src**.

Make sure the submitted Quartus project is compilable. All submissions which can not be compiled will be graded with zero points! The submitted archive should have the following structure:

```
submission_ex4.tar.gz
├── report.pdf ........................ Include your lab report here.
├── src ............................... The complete VHDL source code.
│   ├── alu.vhd
│   ├── core.vhd
│   └── [...  all other source files]
└── quartus ........................... The quartus project and associated files.
    └── ...
```

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: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.