INF-2200: Computer architecture and organization

Assignment 2

Adrian Løberg Moen

UiT id: amo172@uit.no

GitHub user: AdrianMoen

GitHub Classroom: assignment-2-adrian

October 18, 2022

1 Introduction

MIPS is a microprocessor and computer architecture using RISC instruction set architecture (ISA). This report details the design and implementation of a python simulating a single cycle MIPS microprocessor without forwarding or pipelining. Such a finnished implementation should be able to parse and run so called "memfiles" containing both instruction and data memory in the form of machine code.

It should be noted that I am referencing and using a modified older version of this assignment from the INF2200 fall 2021 course, of which, aside from the precode and help, I(Adrian Løberg Moen) am the sole author.

2 Methods

2.1 Design

As for the design of the single cycle mips implementation, we are handed precode which abstracts a good portion of the simulation. Provided is a simluation engine, pytests, memoryfiles, and a cpu element class which includes ways of connecting elements togeher. The precode heavily implies that we implement each of the processor parts as a separate python file and/or class, that inherits from the cpu element class. Connecting all of these together in the right way and order should yield a functioning microprocessor.

These cpu elements have input signals, output signals, input control and output control which using polymorphism can vary from class to class. For each element, we have n inputs and m outputs of different sorts. these are indexed by strings corresponding to whence it came from. and similarily stored strings corresponding to what we want out. This way we abstract and simplify the connection process, making the process rather uniform. In addittion the information we pass around these addition individual components should mimic the information that the MIPS's components passes around. The design is largely based around what datapath below displays, which would be a single cycle setup. This simply means that each instruction is carried out in its entirety before fetching a new one.

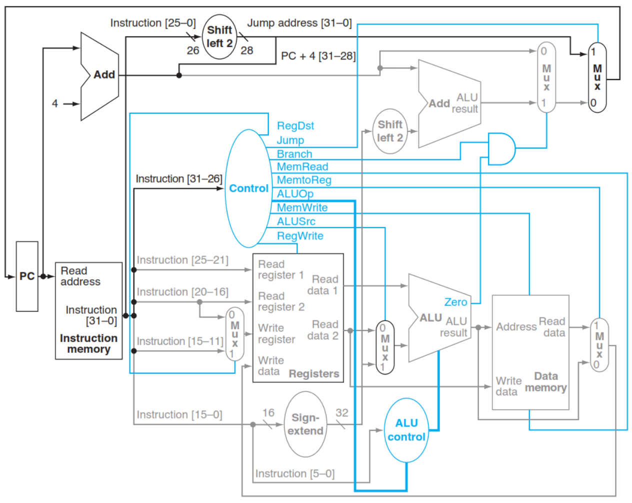


Figure 1: MIPS architecture datapath[2]

2.2 Implementation

One thing to be noted is that every time i refer to or speak of my implementation/code/project, it refers to a implementation derived from where I (Adrian Løberg Moen) left of on the 'home exam 1' in INF-2200 fall 2021. Almost all of the implementation choices, except for changes made in light of new problems, where made then. Despite this, I shall refer to this as one continous project and make no further destinctions.

When it comes to implementing the actuall design, a few key choices have to be made. One of them being what values you wish to store and pass around between the components. For this I decided to store the instruction and data memory using integer values, the other popular option being strings. The reason for choosing integers is due to the fact that the binary signals that are passed around can simply be represented by a number instead. However, as we will see further on, there are significant downsides.

Due to the nature of python, a lack of proper sources, and other issues, there are a multitude of different workarounds that have been used which deviate from the original design[2].

2.2.1 Control

Under the control section I've made a few changes that may or may not deviate from the intended MIPS design. Going by the sources found [3], another bit has been added to the aluOP control signal, making it 3 bits instead of 2, and another control signal called BNE. Due to pythons handling of signed vs unsigned integers and their int vs bit reprensentation, it was necessary to distinguish between Lui and other immediate type operations, for reasons that shall be covered soon. The BNE control signal will also be discussed further on.

2.2.2 Sign Extend

Despite sign extends simple task, work arounds had to be made. Since ints are passed around, the simple act of extending the most significant bit of the immedi value will not give the correct result. A "masking" has to be done using the FromUnsignedWordToSignedWord function from the supplied common.py file. If we where to perform a lui operation on this masked integer, the resulting 16 left shift will leave us with an incorrect result. Thus we connect the aluOP control from the control and into the sign extend so that we can skip the masking functions when needed.

2.2.3 Alu

In the alu, when dealing with integers, one might stumple upon some issues regarding incoming values from registers not addhearing to 2's complement overflowing rules. We can only represent values between $2^{31} - 1$

and -2^{31}. These incoming values need to be masked upon entry. However, bitwise operations do not like this. and we need to create a set of secondary variables that are only used for bitwise operations in the alu, like 'lui' or 'nor'.

2.2.4 BNE

Some changes and additions had to be made in order to get a functioning BNE(branch not equal). Since the alu at this current stage, always outputs a zero signal should the incoming values be equal, we use our BNE signal to revert this zero signal in a sepcial added component called bne.py. The zero signal is only reverted if the bne control signal is on, otherwise it passes on unchanged.

2.2.5 Datamemory

One prevalent issues is the registry writeback in the event of a load word operation. the datamemory is the last component in the list to generate its output. Thusly, it is neccessary to add another readinput and writeoutput for the registryile at the end of the tick method in the simulaton engine. This guarantees that the data memory output is ready for a subsequent instruction should it depend on it.

2.2.6 Misc.

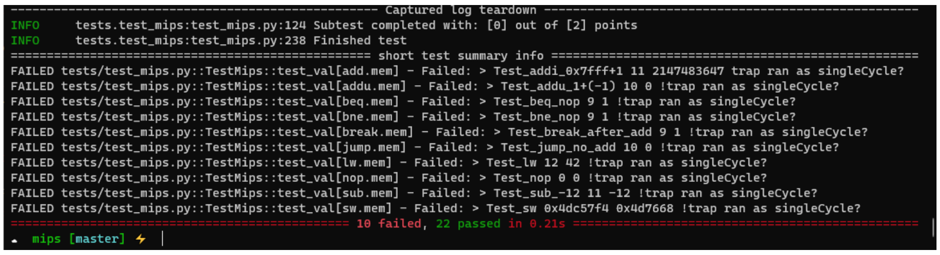
Other interesting implementations are f. ex the pc adder that join the upper 4 bits of the next pc address with the lower 28 jump address bits. This happends in its own file pcjump.py. In order to test the memfiles inside memfiles, a break in the memfiles parsing had to be made. Using the breakinmemoryfile variable inside the memory parent class, one can choose which part of the memfile to store in memory. A value of 0 equals a deactivation of this function. There are a myriad of other interesting ways certain situations have been solved considering all the problems that have rose along the way.

2.3 Experiments

A part of the precode consisted of testing equipment for the simulations. The testing aspect was not a big part of this task, and testing refers more to weather or not the implementation works as intended. Since this project stems from a repository and precode that is over a year old, it is unclear if there have been any changes to the tests. The tests require forwarding and pipelining to get all 32 marks.

3 Results

If you look at figure **??** we can see that all the remaining test that fail are traps that require forwarding and pipelining to complete properly. A single cycle implementation can not get a better score than this. This shows that the current implementation is working. Add.mem and fibonacci.mem are also successfull.



```
------------------------------------------------ Captured log teardown ------------------------------------------------
INFO     tests.test_mips:test_mips.py:124 Subtest completed with: [0] out of [2] points
INFO     tests.test_mips:test_mips.py:238 Finished test
================================================= short test summary info =================================================
FAILED tests/test_mips.py::TestMips::test_val[add.mem] - Failed: > Test_addi_0x7fff+1 11 2147483647 trap ran as singleCycle?
FAILED tests/test_mips.py::TestMips::test_val[addu.mem] - Failed: > Test_addu_1+(-1) 10 0 !trap ran as singleCycle?
FAILED tests/test_mips.py::TestMips::test_val[beq.mem] - Failed: > Test_beq_nop 9 1 !trap ran as singleCycle?
FAILED tests/test_mips.py::TestMips::test_val[bne.mem] - Failed: > Test_bne_nop 9 1 !trap ran as singleCycle?
FAILED tests/test_mips.py::TestMips::test_val[break.mem] - Failed: > Test_break_after_add 9 1 !trap ran as singleCycle?
FAILED tests/test_mips.py::TestMips::test_val[jump.mem] - Failed: > Test_jump_no_add 10 0 !trap ran as singleCycle?
FAILED tests/test_mips.py::TestMips::test_val[lw.mem] - Failed: > Test_lw 12 42 !trap ran as singleCycle?
FAILED tests/test_mips.py::TestMips::test_val[nop.mem] - Failed: > Test_nop 0 0 !trap ran as singleCycle?
FAILED tests/test_mips.py::TestMips::test_val[sub.mem] - Failed: > Test_sub_-12 11 -12 !trap ran as singleCycle?
FAILED tests/test_mips.py::TestMips::test_val[sw.mem] - Failed: > Test_sw 0x4dc57f4 0x4d7668 !trap ran as singleCycle?
============================================== 10 failed, 22 passed in 0.21s ==============================================
 mips [master] ⚡ |
```

Figure 2: Result of the pytest

4 Discussion

4.1 Lack of control signal 'sources'

One thing that struck me both last and this year, is the lack of sources regarding correct opcode and control signal output. Sure after some time going through the project I could reason which control signals would need to be activated for each opcode. In a similar way, the alucontrol and aluop control signals are relatively arbitrary, aslong as the recieving end understands what needs to be done, it should not really matter. However for students who are attempting this challenge head on. The lack of sources, and clear sources at that, is a thing that still puzzles me. The book contains the opcode for r-format instructions, load word, store word and beq. There are far more instructions than this, and the complete picture is not likely to be understood until the very end, if at all.

4.2 Signed vs unsigned

It comes to ones attention along the path, hopefully, that the u in Addiu does not actually mean unsigned in the way we understand. Addiu can be used for both adding and subtracting, and will allow for overflows from both ends without problem. This is not emmidiately apparent unless one follows the book excactly, and forget about the u in the name of the operation. Same goes for the add, addi and sub, who will raise an overflow flag(exception) upon overflowing. These operations all overflow between $2^{31} - 1$ and -2^{31} and any number above or beyond can actually not be represented in MIPS. This raises the question, how will be access memory addresses above $2^{31} - 1$, since this will be an alu output. One idea could be to simply mask it back should it be a negative value once addressing takes place.

4.3 Conclusion

In this assignment, We have successfully implemented a single cycle mips simulation. All the memory files, except for the ones requiring forwarding and pipelining have ran correctly.

5 Sources

References

[1] David A. Patterson. John L. Hennessy. Computer Organization and Design - The Hardware/Software Interface, 6th edition. Chapters 4.

[2] Mustafa. (October 17th 2018). MIPS architecture perform an instruction called swap. Retrieved 22.29, October 17th, 2022 from`https://stackoverflow.com/questions/52860116/mips-architecture-perform-an-instruction-called-swap`

[3] Safaa Omran. (Jan 2016). FPGA Implementation of MIPS RISC Processor for Educational Purposes. Retrieved 23:42, October 17th, 2022, from `https://www.researchgate.net/figure/main-control-truth-table_tbl2_317490993`

[4] Wikipedia contributors. (2019, March 7). Page replacement algorithm. In Wikipedia, The Free Encyclopedia. Retrieved 01:17, October 18th, 2022, from `https://en.wikipedia.org/wiki/MIPS_architecture`