Homework #2: MIPS Single-cycle CPU Implementation

Due: Nov 26, 11:59 PM
Responsible TA: Minseok Kim (minseok1335@unist.ac.kr)

1 Homework Description

In this homework, you will implement a MIPS single-cycle CPU. The goal is to build a working model of a MIPS processor that can simulate a 5-stage in a single clock cycle. The simulator loads an MIPS binary into a simulated instruction memory and executes the instructions. Instruction execution will change the states of registers and memory. This homework emphasizes understanding of the MIPS datapath and control, single-cycle architecture, and how individual components work together to form a fully operational CPU.

2 Submission

- Late submission will be assessed a penalty of 10% per day (We will only accept late submissions of up to 3 days).
- You should complete all sections of the provided skeleton code [2] and submit the completed files as a zip archive named in the format `Your_ID-hw2.zip`. For example, if your ID is 20231234, then you should submit a file named `20231234-hw2.zip`. Upload this zip file to Blackboard.
- You should include comments explaining each part of your implemented logic within the code. Comments can be written in either English or Korean.

3 Execution Environment

You are required to fulfill this homework using Python 3.9, utilizing only the Python standard library [5]. This means we do not assume any additional libraries downloaded via `pip`. Note that we will evaluate your code in a Python 3.9.20 on Ubuntu 20.04. If your code does not run correctly in our environment, you will receive zero points for this homework. No exceptions will be made.

To help you ensure that your code runs correctly in our grading environment, we provide a Docker image [3]. You can set up our grading environment by running the following two commands in your code directory.

```
$ docker pull cse261/hw
$ docker run --rm -it -v .:/app cse261/hw /bin/bash
```

Using the `-v` option, the working directory inside the grading environment will be linked to the working directory in your host environment. By running your code in this grading environment, you can ensure that it runs properly before submission.

4 Instruction Set

Your CPU must support the following subset of the MIPS instructions:

`lw`, `sw` , `beq`, `add`, `sub`, `and`, `or`, `slt`, `nor`, `addi`

Detailed information regarding the instructions can be found on the MIPS green sheet page [1].

5 CPU Details

Download the skeleton code from [2]. In this homework, your task is to complete the skeleton code so that your CPU code produces the correct standard output when executed based on the given input files.

5.1 CPU Execution

`main.py` will read the provided input files (§5.2), initialize the MIPS CPU state elements, including registers and memory, execute the instructions, and display the corresponding execution outputs (§5.3). Here is an example of how to use it:

```
python3 main.py --inst-file testcase_add/instMem.txt --data-file testcase_add/dataMem.txt
                --reg-file testcase_add/register.txt
```

We provide multiple `testcase_*` directories within the skeleton code directory. You can use these `testcase_*` directories to verify that your CPU simulator code works correctly. During our evaluation, we will test your code not only with the `testcase_*` test cases but also with various other test cases to ensure it operates properly.

5.2 Input Files and Formats

There are three input files: `instMem.txt`, `dataMem.txt`, and `register.txt`. We describe the meaning and format of each file.

Instruction memory initialization file (`instMem.txt`). This file specifies the instructions (in hexadecimal format) to be loaded into instruction memory. Below is an example of the `instMem.txt` file:

```
00c90020 # add $0, $6, $9
aca00002 # sw $0, 2($5)
```

In the example, hexadecimal values were listed with comments following the # symbol indicating the actual instruction. Keep in mind that during evaluation, we will provide test input files where no comments are included. Note that you can easily convert instructions into hexadecimal format using this website [4].

Data memory initialization file (`dataMem.txt`). This file contains the initial data memory values that will be loaded into the data memory. These values will be used during the execution of load/store instructions. Below is an example of the `dataMem.txt` file:

```
00 00000010
04 00000055
08 00000000
12 0010ffff
16 00000000
20 00000000
24 00000000
28 00000000
```

This file consists of two columns. The first column specifies the word addresses (in decimal format) sequentially, and the second column lists the 32-bit data to be stored at each corresponding address (in hexadecimal format). In this homework, we will only consider addresses in the range of 0 to 32. Note that we only assume a word-aligned structure, meaning only word addresses are used for access.

Register initialization file. (`register.txt`). This file specifies the initial values of the registers. It defines the initial state of the registers before the CPU begins execution. Below is an example of the `register.txt` file:

```
$0 00000000
$1 00000000
$2 00000000
$3 00000000
$4 00000000
$5 0000000a
$6 00000010
```

```
$7 00000000
$8 00000000
$9 00000301
```

This file consists of two columns. The first column specifies the register numbers, denoted as `$` followed by the register number, and the second column lists the 32-bit data stored in that register in hexadecimal format. We will only consider registers `$0` through `$9`.

5.3 Output format

When you run `main.py`, you must display (1) the current cycle number, (2) the current instruction being fetched in hexadecimal format, and (3) the state of the registers and memory *after each cycle*, all in the standard output. The expected output for the input files in §5.2 is demonstrated in Appendix 8. Additionally, we provide an `expected_result.txt` file in each `testcase_*` folder, which contains the expected output for the corresponding directory's input files.

5.4 Skeleton Code: CPU (`cpu.py`)

This file contains the CPU execution logic. You need to implement instruction fetching, decoding, execution, memory access, and register write-back by using `Control` (§5.5), `ALU` (§5.6), `RegisterFile` (§5.7), and `Memory` (§5.8) classes.

- **Implement `run_cycle`.** The `run_cycle` function is where the entire instruction processing cycle takes place, and therefore, you should complete this function. `run_cycle`, you will implement the cycle steps by using functions from the following objects: `self.pc`, `self.alu`, `self.control`, `self.memory`, and `self.register_file`. Additionally, in this function, you need to print the fetched instruction. To do so, use the `print_instruction_fetch` function from `utils.py`.

5.5 Skeleton Code: Control Unit (`control.py`)

This file manages the CPU's control unit. The `set_control_signals` function sets the main control signals based on the `opcode`, while the `set_alu_signal` function determines the 4-bit control signal for the ALU operation. Refer to the comments in the respective functions for more details.

- **Implement `set_control_signals`.** Based on the `opcode`, this function determines the appropriate values for various control signals, including `RegDst`, `ALUSrc`, `MemtoReg`, `RegWrite`, `MemRead`, `MemWrite`, `Branch`, and `ALUOp`. The resulting control signals are stored in the `self.signals` dictionary. If the value of a control signal is 'don't care', assign it as `None`.
- **Implement `set_alu_signal`.** This function analyzes the `aluop` and `funct` to determine the exact ALU operation signal. The selected 4-bit ALU operation code will be stored in `self.alu_signal`.

5.6 Skeleton Code: ALU (`alu.py`)

ALU (Arithmetic Logic Unit) class for performing arithmetic and logical operations. Refer to the comments in the respective functions for more details.

- **Implement `operate`.** This function performs ALU operations based on the operation argument (4-bit ALU control signal) and the `operand1` and `operand2` arguments. It returns the result of the operation along with a `zero` flag.

5.7 Skeleton Code: RegisterFile (`register_file.py`)

This class uses the `load_registers` function to read the contents of `register.txt`, loading them into the `registers` variable. The class also allows reading from and writing to registers. Refer to the comments in the respective functions for more details.

- **Implement `read`.** This function retrieves the value stored in the specified register. It takes the register number as an argument and returns the corresponding register's value.
- **Implement `write`.** This function stores a value in the specified register. It takes the register number and the value to be written as arguments and writes the value to the corresponding register.

5.8 Skeleton Code: Memory (`memory.py`)

This class uses the `load_instructions` and `load_data` functions to read the contents of `instMem.txt` and `dataMem.txt`, respectively, loading them into the `instruction_memory` and `data_memory` variable. The instruction memory is filled sequentially starting from address 0. The class also allows reading from and writing to data memory and also supports reading instructions from instruction memory. Refer to the comments in the respective functions for more details.

- **Implement `read_instruction`.** This function retrieves an instruction from the instruction memory. It takes an address as an argument and returns the instruction stored at that address.
- **Implement `read_data`.** This function retrieves data from the data memory. It takes an address as an argument and returns the data stored at that address.
- **Implement `write_data`.** This function writes a value to the data memory at a specified address.

5.9 `utils.py`

This file contains functions for printing the fetched instruction and handling errors. You may add additional functions here if needed.

5.10 Error Handling

In this homework, we will check whether your code properly handles five specific errors: integer overflow, invalid `opcode`, invalid `funct`, invalid data memory access, and invalid register access. Each error-handling process should (1) print the relevant error log and (2) immediatly terminate the program. Specifically, we have prepared five error-handling functions in `utils.py` to facilitate error handling. Use these functions appropriately to handle errors.

- **Handle integer overflow** (using `handle_overflow` function): Call this function in `alu.py` if an integer overflow occurs during an operation.
- **Handle invalid opcode** (using `handle_invalid_opcode` function): Call this function within the `set_control_signals` function in `control.py` if an invalid `opcode` is detected. Specifically, we consider any `opcode` not specified in §4 as an invalid `opcode`.
- **Handle invalid funct** (using `handle_invalid_funct` function): Call this function within the `set_alu_signal` function in `control.py` if an invalid `funct` code is detected. Specifically, we consider any `funct` not specified in §4 as an invalid `funct`.
- **Handle invalid data memory access** (using `handle_invalid_memory_access` function): Call this function in `memory.py` if an attempt is made to access an invalid data memory address. Specifically, we consider any data memory address not specified in §5.2 as an invalid data memory. Note that we only assume word addresses. Therefore, any address that is not a 'word address' within the range of 0–32 (e.g., address 2) is also treated as an invalid address.
- **Handle invalid register access** (using `handle_invalid_register_access` function): Call this function in `register.py` if an attempt is made to access an invalid register number. Specifically, we consider any register number not specified in §5.2 as an invalid register number.

6 Test Your Code

As mentioned in §5.1, there are multiple `testcase_*` directories within the skeleton code directory. Each `testcase_*` directory includes input files (`dataMem.txt`, `instMem.txt`, `register.txt`) as well as an `expected_result.txt` file, which contains the expected output corresponding to those input files (the content that should appear in the standard output). You can use these to check if your code produces the expected output for the given input files. To facilitate comparison, we provide a tool called `test_cpu.py`. You can run `test_cpu.py` with the following command:

```
python3 test_cpu.py <test_case_dir>
```

Using this command, you can verify whether your CPU simulator works correctly for the specified `<test_case_dir>`. If it operates correctly, the standard output will display `"The output matches the expected output."` If it does not, it will show `"The output does NOT match the expected output."`, along with details of any discrepancies. Note that we will evaluate your submitted CPU program by using `test_cpu.py`.

7 Note

- **Binary as int type.** You should use the `int` type to process and store binary data. For example, to store the 4-bit ALU operation `0111` in `self.alu_signal` in `control.py`, you can set `self.alu_signal = 0b0111` (which is equivalent to `self.alu_signal = 7`). Although the variable is of type `int`, process it as if it only represents 4 bits. All functions should assume that both argument values and return values are of type `int`, even if they represent binary data.
- **`addi` instruction.** For the `addi` instruction, set the `ALUOp` control signal to `00` in the `set_control_signals` function and so that `alu_signal` is set to `0010` in `set_alu_signal`.
- **Be careful about plagiarism!** We will conduct strict cross-plagiarism detection for evaluation, including a series of answer generated by ChatGPT, code found online, and your submissions. Therefore, we do not recommend consulting ChatGPT or online solutions. Last semester, we identified several cases of plagiarism using an automated tool. If you are found to be engaged in "deep collaboration" with other students, both the provider and the recipient will receive penalties, including, in the worst case, receiving an F grade.
- **Grading policy.** We will grade strictly based on the accuracy of the output relative to the input. No excuses will be accepted for discrepancies. **Note that we will evaluate not only the entire code but also the functionality of each class individually**. Therefore, ensure that each function is implemented according to its given specifications.
- **Questions.** If you have any requests or questions (technical difficulties, late submission due to inevitable circumstances, etc.), please ask the TAs on Blackboard. We generally encourage the use of Blackboard for discussions. However, for urgent issues or secret issues, you can send an email to the responsible TA, Minseok Kim.

References

[1] 2009. MIPS Reference Data. https://websec-lab.github.io/courses/2024f-cse261/metarials/MIPS_Green_Sheet.pdf.

[2] 2024. CSE261 HW2 skeleton code. https://websec-lab.github.io/courses/2024f-cse261/hw/hw2-skeleton.zip.

[3] 2024. Docker Hub Image: cse261/hw. https://hub.docker.com/repository/docker/cse261/hw.

[4] 2024. MIPS converter. https://mips-converter.jeffsieu.com/.

[5] 2024. The Python Standard Library. https://docs.python.org/3.9/library/.

8 Appendix: Output Example

```
[*] Initial states
-----------------------------------
Current register states:
-----------------------------------
$0: 0x00000000
$1: 0x00000000
$2: 0x00000000
$3: 0x00000000
$4: 0x00000000
$5: 0x0000000a
$6: 0x00000010
$7: 0x00000000
$8: 0x00000000
$9: 0x00000301
-----------------------------------
-----------------------------------
Current memory states:
-----------------------------------
Address 00: 0x00000010
Address 04: 0x00000055
Address 08: 0x00000000
Address 12: 0x0010ffff
Address 16: 0x00000000
Address 20: 0x00000000
Address 24: 0x00000000
Address 28: 0x00000000
-----------------------------------

[*] Current cycle: 1
Fetching instruction at PC=0: 0x00C90020
-----------------------------------
Current register states:
-----------------------------------
$0: 0x00000311
$1: 0x00000000
$2: 0x00000000
$3: 0x00000000
$4: 0x00000000
$5: 0x0000000a
$6: 0x00000010
$7: 0x00000000
$8: 0x00000000
$9: 0x00000301
-----------------------------------
-----------------------------------
Current memory states:
-----------------------------------
Address 00: 0x00000010
Address 04: 0x00000055
Address 08: 0x00000000
Address 12: 0x0010ffff
Address 16: 0x00000000
Address 20: 0x00000000
Address 24: 0x00000000
Address 28: 0x00000000
-----------------------------------
```

```
[*] Current cycle: 2
Fetching instruction at PC=4: 0xACA00002
-----------------------------------
Current register states:
-----------------------------------
$0: 0x00000311
$1: 0x00000000
$2: 0x00000000
$3: 0x00000000
$4: 0x00000000
$5: 0x0000000a
$6: 0x00000010
$7: 0x00000000
$8: 0x00000000
$9: 0x00000301
-----------------------------------
-----------------------------------
Current memory states:
-----------------------------------
Address 00: 0x00000010
Address 04: 0x00000055
Address 08: 0x00000000
Address 12: 0x00000311
Address 16: 0x00000000
Address 20: 0x00000000
Address 24: 0x00000000
Address 28: 0x00000000
-----------------------------------
```