Assignment #2:
A C++-Based Single-Cycle MIPS CPU

CAS3102-01 (Architecture of Computers), Spring 2025

Introduction

Welcome to the first programming assignment for CSI3102-02 @ Yonsei University for Spring 2024! This programming assignment will teach you how to leverage C++ to mimic hardware description language (HDL) style programming and implement a single-cycle MIPS CPU capable of executing a few instructions.

Learning HDL-style programming is essential for gaining a better understanding of how computers, which are essentially digital circuits driven by clock signals, work. There exist industry-standard HDL programming languages such as Verilog, VHDL, and SystemVerilog. However, given that this course is open to not only CS undergrads but also non-CS undergrads, we decided to refrain from using the HDL programming languages. Still, we highly recommend that you study and learn the HDL programming languages after taking this course if you are interested in the field of computer architecture. Essentially, all computers and their key components, such as Central Processing Units (CPUs), Graphics Processing Units (GPUs), and Neural Processing Units (NPUs), are written with the HDL programming languages and then fabricated with the use of low-level cell libraries.

The eventual goal of this programming assignment is to implement a **single-cycle** MIPS CPU using C++. This assignment requires you to implement various hardware components necessary for implementing a complete single-cycle processor. First, you will implement a few digital circuits which form the single-cycle CPU, such as the arithmetic logical unit, ALU control unit, Control unit, instruction/data memory, and register file. After that, you will utilize the digital circuits to complete the single-cycle CPU (i.e., faithfully implement the single-cycle behaviors of the single-cycle CPU).

Your implementation of the single-cycle MIPS CPU should support the following **ten** MIPS instructions:

- 32-bit signed integer addition/subtraction (`add`, `sub`, `addi`)
- Bitwise logical operations (`and`, `or`, `nor`)
- Set on less than (`slt`)
- Load/store word (`lw`, `sw`)
- Branch if equal (`beq`)

Downloading the Assignment

Please download this assignment and perform a sanity check by executing the following commands:

```
csi3102@csi3102:~$ wget \
            https://hpcp.yonsei.ac.kr/cas3102-2025sp/assn2-20250503-ArchIsFun.tar.gz
csi3102@csi3102:~$ tar zvxf ./assn2-20250503-ArchIsFun.tar.gz
csi3102@csi3102:~$ cd ./assn2
csi3102@csi3102:~/assn2$ ls -F
ALUControl.h   ALU.h   Control.h   DigitalCircuit.h   examples/   Makefile   Memory.h
RegisterFile.h  SingleCycleCPU.h  testAssn2.cc  testExamples.cc  tests/
```

The DigitalCircuit Class

The DigitalCircuit class, declared in the DigitalCircuit.h file, is the most fundamental C++ class you will be using throughout the semester. The class is quite simple; it has only one member function and one member. However, you should familiarize yourself with the DigitalCircuit::advanceCycle member function. Invoking this function mimics the actual hardware-level rising edge of a clock signal (i.e., the value of the clock signal changes from 0 to 1). All digital circuits use a rising edge as a trigger for their actions; CPUs and their primary components, such as registers, are no exception.

You should also note that the DigitalCircuit::advanceCycle method is declared as a purely virtual function having no definition. The DigitalCircuit serves as a highly abstract template for any digital circuit you will implement in C/C++ throughout the semester. Whenever you implement a digital circuit, you will be implementing the digital circuit as a class that inherits the DigitalCircuit class, and thus you need to implement the advanceCycle member function of the class appropriate for your class of interest.

N-Bit Wires as std::bitset<N>

Most digital circuits cannot operate independently; they need to retrieve input data from some other digital circuits or input devices, calculate their output using the input data, and propagate the output to other digital circuits and/or output devices. In other words, digital circuits should be able to transfer data to other digital circuits. Wires are what digital circuits use to retrieve/transmit data between them. An N-bit wire can transfer N bits of data. A bit can express either 0 or 1, but cannot express both values at the same time. Due to the characteristics of bits, an N-bit wire is capable of expressing and transferring N-bit binary numbers between digital circuits.

To utilize wires, we will abstract N-bit wires as std::bitset<N> throughout the assignments. std::bitset<N> is a templated C++ class that supports storing and altering an N-bit value. One thing to note is that once an std::bitset<N> object has been created, we cannot alter the bit width of the instance. For example, once a 4-bit wire has been created as an std::bitset<4> object, we cannot change the bit width of the wire; we must instantiate a new std::bitset<N> object using a desired value of N.

Due to the importance of the wires in digital circuits, we strongly recommend that you get familiar with std::bitset<N> by referring to public online resources (e.g., https://cplusplus.com/reference/bitset/bitset/) and examples/tutorials (e.g., https://www.geeksforgeeks.org/cpp-bitset-and-its-application/).

Examples

We provide two example digital circuit implementations through UIntAdder.h and Latch.h, which are located in the examples/ directory, so that you can better understand how to use the DigitalCircuit and the std::bitset<N> classes.

Example #1: An Adder for Unsigned Integers

The first example digital circuit is an adder, which adds two unsigned binary numbers. The adder takes as input two N-bit unsigned binary numbers and produces an (N+1)-bit unsigned binary number. The output is the summation of the two inputs. For example, the adder should produce an unsigned binary number 10000 when the two input binary numbers are 1111 and 0001. Since the output is (N+1) bits, the adder guarantees that no overflow occurs; an overflow occurs if a value does not fall into the range of the values that can be expressed by a specific data type.

The adder is implemented as the UIntAdder class and is declared and implemented in the UIntAdder.h file. The UIntAdder class inherits the DigitalCircuit class, so it implements its own advanceCycle member function. The class also specifies its two N-bit inputs and an (N+1)-bit output as Wires. The class takes advantage of C++ templates; it leaves the value of N as unknown but specifies its type as size_t. Using a C++ template allows the class to support any value of N. You can simply instantiate UIntAdder<4> and UIntAdder<32> instances to create 4-bit or 8-bit adders, respectively.

The adder is a combinational circuit whose output only depends on the values of the inputs for each clock cycle. As a result, the class declares its input and output signals as wires. The wires are not part of the adder; a larger digital circuit attaches the wires to the input and output ports of the adder. Therefore, the class stores the pointers to the input and output wires as its member attributes. You can see how such wiring can be implemented by inspecting the testUIntAdder function in the testExamples.cc file.

Example #2: An N-Bit Latch

The other example is an N-bit latch. Latches are used by digital circuits to store some temporary data for a short period. This characteristic makes latches sequential circuits. As opposed to combinational circuits, the output for a clock cycle of sequential circuits depends not only on the current clock cycle's input values, but also those of the previous clock cycles.

To keep some data over multiple clock cycles, the Latch class, defined and implemented in the Latch.h file, declares one Register as its member attribute. In HDLs, registers are used for temporarily storing a piece of data over time, so we borrow the name from HDLs. Similar to Wires, Registers are defined as std::bitset<N> for storing N-bit binary numbers. This allows us to use Wires and Registers interchangeably; however, you should know that in HDLs, wires and registers are separate and have different purposes.

The Latch class implements an N-bit latch using a C++ template. The value of N is not specified, but the implementation of the Latch class is generic and thus can support any value of N through a C++ template. In addition to an N-bit input wire, the Latch class also takes as input a 1-bit writeEnable wire. For each clock cycle, when the writeEnable wire is set to 1, the latch updates its register to have the value of the input wire. Otherwise (i.e., the writeEnable wire is set to 0), the latch ignores the input wire value in the clock cycle. In both cases, the latch sets the value of the output wire as the current value of its register.

Testing the Example Digital Circuits

You can compile and test the two example classes by typing the following command in the terminal:

```
~/assn2$ make clean && make testExamples && ./testExamples
rm -f testExamples testAssn2
g++ -o testExamples -std=c++11 -I. -Iexamples/ testExamples.cc


<<<<<<<<<<<<<<<<<<<< testing UIntAdder >>>>>>>>>>>>>>>>>>>>
[printName] this = 0x557c483d0eb0, this->_name = UInt32Adder
##### Cycle 1: 0x01010101 + 0x10101010 #####
[printWire] wireName = adderInput0, wireValue = 0x01010101
[printWire] wireName = adderInput1, wireValue = 0x10101010
[printWire] wireName = adderOutput, wireValue = 0x011111111
##### Cycle 2: 0x01010101 + 0x10101010 #####
[printWire] wireName = adderInput0, wireValue = 0x01010101
[printWire] wireName = adderInput1, wireValue = 0x10101010
[printWire] wireName = adderOutput, wireValue = 0x011111111
```

```
##### Cycle 3: 0xFFFFFFFF + 0x00000001 #####
[printWire] wireName = adderInput0, wireValue = 0xffffffff
[printWire] wireName = adderInput1, wireValue = 0x00000001
[printWire] wireName = adderOutput, wireValue = 0x100000000
##### Cycle 4: 0xFFFF0000 + 0x0000FFFF #####
[printWire] wireName = adderInput0, wireValue = 0xffff0000
[printWire] wireName = adderInput1, wireValue = 0x0000ffff
[printWire] wireName = adderOutput, wireValue = 0x0ffffffff


<<<<<<<<<<<<<<<<<<<< testing Latch >>>>>>>>>>>>>>>>>>>>
##### Cycle 1: writeEnable = 1, input = 0x12341234 #####
[printWire] wireName = input, wireValue = 0x12341234
[printWire] wireName = output, wireValue = 0x12341234
##### Cycle 2: writeEnable = 0, input = 0xABCDFFFF #####
[printWire] wireName = input, wireValue = 0xabcdffff
[printWire] wireName = output, wireValue = 0x12341234
##### Cycle 3: writeEnable = 1, input = 0xABCDFFFF #####
[printWire] wireName = input, wireValue = 0xabcdffff
[printWire] wireName = output, wireValue = 0xabcdffff
##### Cycle 4: writeEnable = 1, input = 0xDEADBEEF #####
[printWire] wireName = input, wireValue = 0xdeadbeef
[printWire] wireName = output, wireValue = 0xdeadbeef
```

Note that the prefix 0x denotes that the value is printed as hexadecimal numbers (i.e., the numbers with the base of 16). Also note that, for hexadecimal numbers, 0~9 correspond to decimal digits 0~9, respectively, and A/a~F/f correspond to decimal values of 10~15, respectively.

Problems

This assignment consists of six problems. The first five problems ask you to implement some digital circuits, which will later be used to implement the single-cycle CPU. The last problem asks you to complete implementing the single-cycle CPU using the digital circuits you wrote for the first five problems. Overall, all six problems lead to implementing the single-cycle CPU; however, we split the task into the six problems so that you can obtain appropriate partial scores for the assignment.

Q1: Arithmetic Logical Unit (ALU)

Implement a 32-bit Arithmetic Logical Unit (ALU) that can later become part of your single-cycle MIPS CPU implementation. You should complete the `ALU` class, defined in the `ALU.h` file, by completing the `advanceCycle` method of the `ALU` class.

The ALU takes as input two 32-bit values and one four-bit control signal. The ALU performs different operations on the two input values depending on the control signal, as shown in the table below (also available at page 259 of the textbook):

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Given the input values and the control signal, the ALU should produce a 32-bit output value and a one-bit zero signal. The zero signal gets set to 1 if the output value is zero. Otherwise, the zero signal gets set to 0.

Q2: ALU Control Unit

Implement the ALU control unit for the single-cycle MIPS CPU. You should complete the `ALUControl` class defined in the `ALUControl.h` file.

The ALU control unit takes as input a two-bit ALUOp signal and a six-bit Funct field. The ALU control unit then produces a four-bit operation signal, which later gets used as the control signal for the ALU, as follows:

ALUOp		Funct field						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

FIGURE 4.13 The truth table for the 4 ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Note that when the function field is used, the first 2 bits (F5 and F4) of these instructions are always 10, so they are don't-care terms and are replaced with XX in the truth table.

The Xs in the table denote that the corresponding input values "do not matter". In other words, the input values can be either 0 or 1.

Q3: Control Unit

Implement the Control unit for the single-cycle MIPS CPU by modifying the `Control.h` and completing the implementation of the `Control` class.

The Control unit takes as input the 6-bit opcode of the current instruction and determines the control signals for the various hardware components forming the single-cycle datapath. You should refer to the following table for implementing the Control unit:

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

FIGURE 4.18 The setting of the control lines is completely determined by the opcode fields of the instruction. The first row of the table corresponds to the R-format instructions (`add`, `sub`, `AND`, `OR`, and `slt`). For all these instructions, the source register fields are rs and rt, and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set. Furthermore, an R-type instruction writes a register (Reg-Write = 1), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high. The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field. The second and third rows of this table give the control signal settings for `lw` and `sw`. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register. The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU. The ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality. Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to RegDst when RegWrite is 0. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

Note that the table does not show how the control signals should be set for some instructions (e.g., `addi`). It is up to you to determine how to set the control signals for such instructions. In addition, for the control signals that do not matter, your Control unit should set such control signals as 0.

Q4: Register File

Implement the Register File for the single-cycle CPU by completing the `RegisterFile` class defined in the `RegisterFile.h` file. Specifically, the Register File supports the 32 general-purpose integer registers of the MIPS ISA (e.g., `$t0`, `$s0`, `$ra`). You will not be implementing floating-point instructions in this assignment, so implementing only the 32 integer registers suffices.

Upon invoking the `advanceCycle` method, your implementation of the `RegisterFile` class should behave as follows:

- For reading the values of the two source registers, the value of `RegWrite` is set to 0. The indices of `$rs` and `$rt` are given through `ReadRegister1` and `ReadRegister2`, respectively. Then, the values of `$rs` and `$rt` should be returned through `ReadData1` and `ReadData2`, respectively.

- When writing a value to a destination register, the value of `RegWrite` is set to 1. The index of the destination register gets passed through `WriteRegister`, and the value to be written to the destination register gets passed through `WriteData`. No return value exists for writes.

One thing to keep in mind is that the zero register (i.e., `$0`) should always have the value of zero. In other words, any writes to the zero register should be ignored.

Q5: Memory

Implement a Memory for the single-cycle CPU by extending the `Memory.h` file and completing the implementation of the `Memory` class. The Memory class gets used for both the instruction and the data memories by the single-cycle CPU.

Similar to the Register File, the Memory supports reading data from and writing data to it. For the purpose, the `Memory` class implements a byte-accessible memory by defining an array of `std::bitset<8>` objects.

Upon an invocation of the `advanceCycle` method, your Memory class should behave as follows:

- To read 32-bit data from the memory, the `MemRead` parameter is set to 1, and the 32-bit target address is given through the `Address` parameter. Then, the method should return the requested 32-bit data through the `ReadData` parameter.

- When writing some 32-bit data to the memory, the `MemWrite` parameter is set to one, and the target address is given through the `Address` parameter. The data to be written to the target address is given through the `WriteData` parameter. No value should be returned for memory writes.

The `Memory` class mostly resembles the `RegisterFile` class; however, it differs from the `RegisterFile` class in that one of its attributes denotes the *endianness* of the data stored in memory (i.e., the `_endianness` attribute). The 32-bit data stored in a register can be stored in memory in different byte orders: big-endian places the most-significant bit of the data at the lowest memory address, whereas little-endian places the most-significant bit at the highest memory address instead. Although we mostly focused on little-endian, your implementation of the `Memory` class should support both big- and little-endian. We recommend that you analyze the code of the `Memory::printMemory` method regarding the endianness.

Q6: Single-Cycle CPU

Using the digital circuits you wrote for the previous questions, you should now implement the single-cycle CPU by extending the `SingleCycleCPU` class in the `SingleCycleCPU.h` file. An invocation of the `SingleCycleCPU::advanceCycle` method simulates one clock cycle of the single-cycle CPU by modeling the single-cycle CPU shown below (Fig. 4.19 in the textbook).

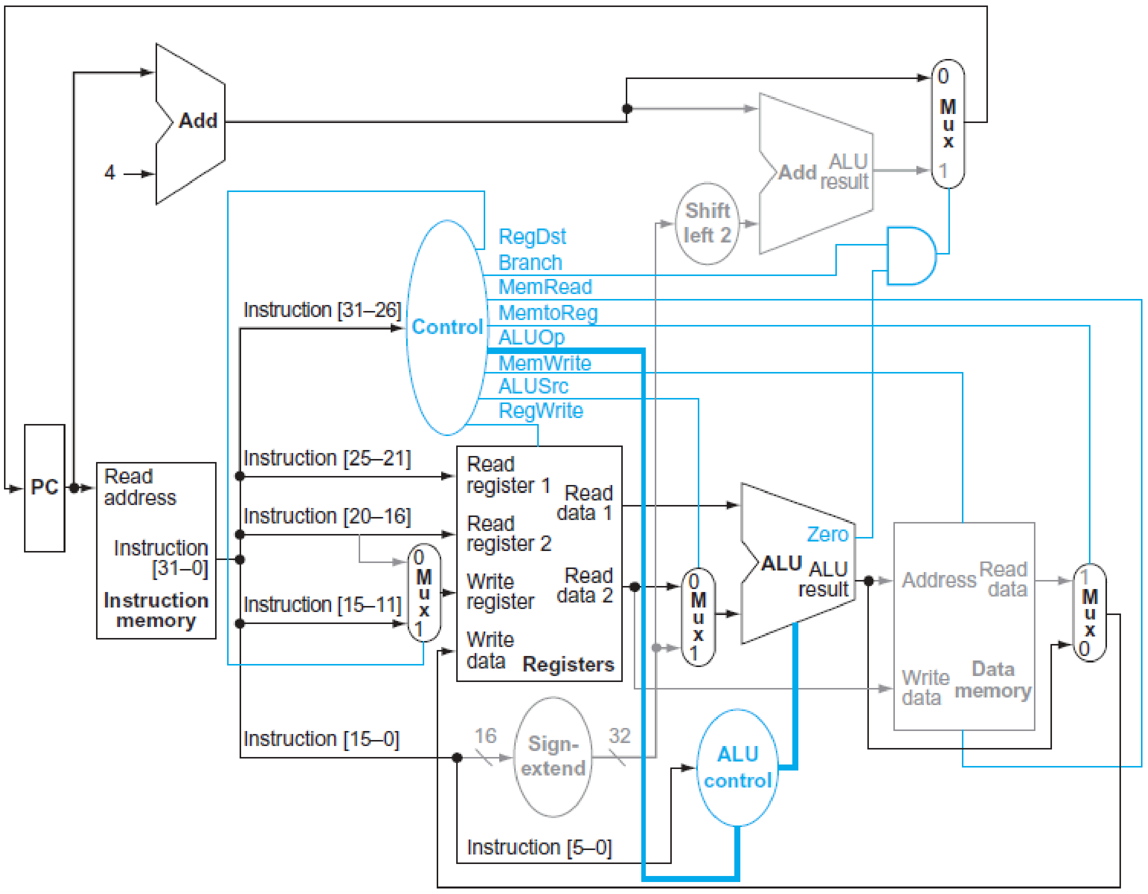

FIGURE 4.19 The datapath in operation for an R-type instruction, such as add $t1,$t2,$t3**.** The control lines, datapath units, and connections that are active are highlighted.

You will soon realize that, in addition to the digital circuits you wrote for the previous questions, you need to implement additional digital circuits (e.g., Sign-extend, Add) for completing the single-cycle CPU. For the necessary additional digital circuits, one thing to note is that you do not need to declare them as separate classes; combinational circuits can be abstracted as simple C/C++ functions, and such functions can be "inlined" into the `SingleCycleCPU::advanceCycle` method. For example, the Sign-extend unit can be replaced with a series of bit manipulation operations on a `Wire` object. As another example, the Shift-left-2 unit can be replaced with a single bitwise shift operation on a `Wire` object.

In addition, you will realize that you need various `Wire` objects for connecting the inputs and outputs of the digital circuits forming the single-cycle CPU. You should leverage your knowledge on the `Wire` class, which is essentially an alias to the `std::bitset<N>` class for `N`-bit wires, for implementing the necessary wires. Although defining and using custom `Wire` objects is one possible option, we highly recommend that you utilize the predefined `Wire`-typed attributes of the `SingleCycleCPU` class. All the attributes are associated with a short description of their purposes.

One difference between this question and the prior questions is that you also need to complete implementing the constructor for the `SingleCycleCPU` class. In the constructor, you should instantiate all the necessary digital circuits and connect the wires to the digital circuits. You can refer to the attributes of the `SingleCycleCPU` class, whose types are the sub-classes of the `DigitalCircuit` class, to get some hints on what components should be initialized in the constructor.

One obstacle you will encounter is: how you should invoke the `advanceCycle` methods of the digital circuits being used within the single-cycle CPU. One hint for solving the obstacle is to think of invoking the `advanceCycle` method of a digital circuit within the `SingleCycleCPU::advanceCycle` method as (1) creating a “virtual sub-cycle” within one cycle of the CPU, and (2) making a digital circuit perform the virtual sub-cycle to obtain the digital circuit’s output necessary for completing the implementation of the single-cycle CPU. One more hint: it is okay to create “multiple” virtual sub-cycles for some digital circuits **as long as the programmer-visible state of the single-cycle CPU correctly gets updated by the end of a clock cycle**. This second hint will especially be helpful when you deal with the RegisterFile :)

Testing Your Implementation

We provide the `testAssn2.cc` file so that you can perform a minimal test on your implementation of the digital circuits and the single-cycle CPU. The `testAssn2.cc` file includes the C/C++ code for instantiating and testing your implementation of the digital circuits and the single-cycle CPU. The digital circuits and the single-cycle CPU are tested in separate C/C++ functions (e.g., `testALU`, `testALUControl`, `testSingleCycleCPU`).

To test your implementations for the first five questions (i.e., the five digital circuits which become part of the single-cycle CPU), you should invoke the following command on the terminal:

```
csi3102@csi3102:~/assn2$ make clean && make testAssn2 && ./testAssn2
rm -f testExamples testAssn2
g++ -o testAssn2 -std=c++11 -I. testAssn2.cc
INFO: testing ALU...
INFO: 0x03478bcf AND 0x00000000 --> output = 0x00000000, zero = 0x1
[printWire] wireName = input0, wireValue = 0x03478bcf
[printWire] wireName = input1, wireValue = 0x00000000
[printWire] wireName = aluControl, wireValue = 0x0
[printWire] wireName = output, wireValue = 0x00000000
[printWire] wireName = zero, wireValue = 0x1
INFO: 0x03478bcf AND 0x12569ade --> output = 0x12569ade, zero = 0x0
[printWire] wireName = input0, wireValue = 0x03478bcf
[printWire] wireName = input1, wireValue = 0x12569ade
[printWire] wireName = aluControl, wireValue = 0x0
[printWire] wireName = output, wireValue = 0x02468ace
[printWire] wireName = zero, wireValue = 0x0
...
```

As shown above, executing `testAssn2` will show the expected output and the output of your implementation. You can compare the two outputs to check whether your implementation works correctly.

For testing the single-cycle CPU, you need to provide additional command-line arguments. We provide two examples and their reference outputs as follows.

Example #1: Executing Non-Branch MIPS Instructions

After implementing the `SingleCycleCPU` class, you can test your code using the example files (`regFile`, `instMemFile`, `dataMemFile`) located in the `tests` directory. You can run your code by:

```
csi3102@csi3102:~/assn2$ ./testAssn2 0 ./tests/ex1_regFile ./tests/ex1_instMemFile
./tests/ex1_dataMemFile 6
```

After initializing the registers, data memory, and instruction memory in cycle 0, the `advanceCycle` method executes one clock cycle of the single-cycle CPU. The following example shows a reference output for the six clock cycles by invoking the `advanceCycle` method six times. Note that the zero-valued registers are omitted for brevity, and that only the changes in the programmer-visible state are shown.

```
==================== Cycle 0 ====================
PC = 0x00000000
Registers:
  $12 = 0x00000001
Data Memory:
  memory[0x00000000..0x00000003] = 0x00000010
  memory[0x00000004..0x00000007] = 0x00000055
Instruction Memory:
  memory[0x00000000..0x00000003] = 0x8d090000
  memory[0x00000004..0x00000007] = 0x8d0a0004
  memory[0x00000008..0x0000000b] = 0x012a5820
  memory[0x0000000c..0x0000000f] = 0xad0b0008
  memory[0x00000010..0x00000013] = 0x016c6822
  memory[0x00000014..0x00000017] = 0xad0d000c

==================== Cycle 1 ====================
# $pc = 0, lw $t1, 0($t0)
PC = 0x00000004
Registers:
  $09 = 0x00000010

==================== Cycle 2 ====================
# $pc = 4, lw $t2, 4($t0)
PC = 0x00000008
Registers:
 $10 = 0x00000055

==================== Cycle 3 ====================
# $pc = 8, add $t3, $t1, $t2
PC = 0x0000000c
Registers:
  $11 = 0x00000065

==================== Cycle 4 ====================
# $pc = 12, sw $t3, 8($t0)
PC = 0x00000010
Data Memory:
  memory[0x00000008..0x0000000b] = 0x00000065

==================== Cycle 5 ====================
# $pc = 16, sub $t5, $t3, $t4
```

```
PC = 0x00000014
Registers:
  $13 = 0x00000064

==================== Cycle 6 ====================
# $pc = 16, sw $t5, 12($t0)
PC = 0x00000018
Data Memory:
  memory[0x0000000c..0x0000000f] = 0x00000064
```

Example #2: Executing MIPS Instructions Involving Taken Branches

Example #2 shows the use of the beq instruction and how it changes the control flow. The example assigns 0x00000711 to both $t0, $t1, which enables the branch to be taken. The offset specified in the branch instruction is 1, so the target register for the branch instruction would be calculated as ($pc + 4) + (offset << 2). Thus, the instruction in 0x00000008 would be executed instead of 0x00000004.

You can run the example by:

```
csi3102@csi3102:~/assn2$ ./testAssn2 0 ./tests/ex2_regFile ./tests/ex2_instMemFile
./tests/ex2_dataMemFile 4
```

The results for the example are shown below:

```
==================== Cycle 0 ====================
PC = 0x00000000
Registers:
  $08 = 0x00000711
  $09 = 0x00000711
  $10 = 0x00001030
  $11 = 0x00000703
Data Memory:
  memory[0x00000000..0x00000003] = 0x00000010
  memory[0x00000004..0x00000007] = 0x00000055
Instruction Memory:
  memory[0x00000000..0x00000003] = 0x11090001
  memory[0x00000004..0x00000007] = 0x01097020
  memory[0x00000008..0x0000000b] = 0x01686025
  memory[0x0000000c..0x0000000f] = 0x110a0002
  memory[0x00000010..0x00000013] = 0x01686824
  memory[0x00000018..0x0000001b] = 0x01097020

==================== Cycle 1 ====================
# $pc = 0, beq $t0, $t1, 1
PC = 0x00000008

==================== Cycle 2 ====================
# $pc = 8, or $t4, $t3, $t0
PC = 0x0000000c
Registers:
  $12 = 0x00000713
```

```
==================== Cycle 3 ====================
# $pc = 12, beq $t0, $t2, 2
PC = 0x00000010

==================== Cycle 4 ====================
# $pc = 16, and $t5, $t3, $t0
PC = 0x00000014
Registers:
  $13 = 0x00000701
```

Submitting Your Code

You should upload **only the six .h files you modified** to LearnUs (i.e., `ALU.h`, `ALUControl.h`, `Control.h`, `RegisterFile.h`, `Memory.h`, and `SingleCycleCPU.h`). If you solved only a part of the assignment, then upload only the .h files you modified to LearnUs.

The deadline for uploading the .h files to the LearnUs system is **11:59 pm on May 16, 2025 (Fri)**. If you need more time, you may upload the .h files to the LearnUs system by **11:59 pm on May 18, 2025 (Sun)**; however, uploading the files after May 16 will result in **a 50% reduction in your score**. For example, if your file gets 80 points, your score will be (1) 80 points if you uploaded your file by May 16, or (2) 40 points if you uploaded your file by May 18.

Please be careful that if your file results in a compilation error, you will receive no score (i.e., 0 points). Therefore, please double-check that your files successfully get compiled before submitting them.

<<<<<<<<<<<<<<<<<<<<<<<<<<<<< END OF ASSIGNMENT #2 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>