Assignment #2: A Single-Cycle MIPS CPU

CSI3102-02 (Architecture of Computers), Spring 2023

Welcome to the second programming assignment for CSI3102-02! You are given **12 days** to complete this assignment. Please carefully read the instructions below, and make sure you upload your assignment to the LearnUs system (https://ys.learnus.org) by the deadline.

In this programming assignment, you will implement a **single-cycle** MIPS CPU. This assignment requires you to implement various hardware components necessary for implementing a complete single-cycle processor. First, you will implement a single-cycle datapath involving 32 general-purpose registers, a program counter (PC) register, instruction/data memory, 2-to-1 multiplexers, an arithmetic-logic unit (ALU), a sign-extender, and so on. Second, you will implement the control and the ALU control units for configuring the behaviors of the datapath with respect to the MIPS instructions fetched from the instruction memory.

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

- Integer addition/subtraction (`add`/`sub`/`addi`), 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/files/csi3102/2023sp/assn2-1a2a3a4.tar.gz
csi3102@csi3102:~$ tar zvxf ./assn2-1a2a3a4.tar.gz
csi3102@csi3102:~$ cd ./assn2
csi3102@csi3102:~/assn2$ ls
Makefile   Memory.hpp   RegisterFile.hpp   SingleCycleCPU.hpp   testSingleCycleCPU.cpp
Memory.cpp RegisterFile.cpp SingleCycleCPU.cpp tests
```

The `SingleCycleCPU` Class

As the first step of this assignment, you should open and inspect the `SingleCycleCPU.hpp` header file. The header file defines the `SingleCycleCPU` class which you will ultimately be implementing in the assignment. While inspecting the header file, you will soon realize the following characteristics of the class:

- The `SingleCycleCPU` class heavily utilizes the `std::bitset<N>` class defined by the C++ standard template library and provided through the `<bitset>` header.

- The *combinational* circuits of the single-cycle datapath are declared as the *method*s of the `SingleCycleCPU` class. The combinational circuits refer to the digital circuits whose output(s) only depend on the provided input(s). Examples of the combinational circuits include the multiplexers, adders, sign-extender, and ALU.

- The *sequential* circuits of the single-cycle datapath, on the other hand, are declared as separate *class*es. Specifically, the registers and instruction/data memory are defined with the `RegisterFile`

and `Memory` classes, respectively. As the output(s) of a sequential circuit depend on not only the current input(s), but also the prior history, implementing sequential circuits as classes which can sustain some states over time using their attributes is more appropriate.

The `std::bitset<N>` Class

In this assignment, the `std::bitset<N>` class is heavily used for representing *any N-bit binary data*. Given that we are dealing with digital circuits which can interpret and produce only 0s and 1s, employing the `std::bitset<N>` class to represent the low-level data for the digital circuits is a sensible choice. For example, we can express a 32-bit wire (e.g., the 32-bit wire connected between the PC register and the `Address' input port of the instruction memory) using an instance of the `std::bitset<32>` class. As another example, we can express the PC register using an instance of the `std::bitset<32>` class as the PC register is 32-bit wide. As can be seen, the use of the `std::bitset<N>` allows us to accurately express the bit widths of various registers, wires, and thus low-level binary operations of any digital circuits.

Due to its heavy use in the assignment, we recommend you to get familiar with the `std::bitset<N>` class and its methods before deep-diving into the assignment by referring to online documentations such as https://en.cppreference.com/w/cpp/utility/bitset.

Combinational Circuits as Methods

You will be implementing the combinational circuits as the methods of the `SingleCycleCPU` class. This is possible as the output(s) of a combinational circuit only depend on the current input(s). Specifically, you should be implementing the following methods of the `SingleCycleCPU` class:

- `SingleCycleCPU::AND<N>` performs a logical AND operation on two `N`-bit binary values.
  - `output = input0 & input1`
- `SingleCycleCPU::Add<N>` adds two `N`-bit signed integers.
  - `output = input0 + input1`
- `SingleCycleCPU::Mux<N>` corresponds to a 2-to-1 multiplexer.
  - If `select = 0`, then `output = input0`. Otherwise, `output = input1`.
- `SingleCycleCPU::SignExtend<N, M>` sign-extends an `N`-bit signed integer into `M` bits.
  - `output[(M-1):N] = input[N-1]` and `output[(N-1):0] = input[(N-1):0]`

    ex)(N=8, M=16) 10000001 ->1111111110000001(9~16자리는 8번째 비트로, 1~8자리는 똑같이)
- `SingleCycleCPU::ShiftLeft2<N>` left-shifts an `N`-bit signed integer to the left by two bits.
  - `output = input << 2`
- `SingleCycleCPU::ALU` serves as the ALU of the single-cycle MIPS CPU.
  - This method takes as input a four-bit control signal through the `control` parameter. The method should perform the arithmetic/logic operation specified by the `control` parameter. The table below (also available at page 259 of the textbook) shows the mapping between the acceptable values of the `control` parameter and the arithmetic/logic operations:

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

- `SingleCycleCPU::Control` and `SingleCycleCPU::ALUControl` produce the appropriate control signals for the other datapath with respect to a given MIPS instruction.
    - As discussed in the prior lectures, our single-cycle MIPS CPU utilizes two control units, namely the *Control* and *ALUControl* units. 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. The ALUControl unit then further interprets the 6-bit funct field of the R-type instructions to pinpoint the arithmetic/logic operation the ALU should perform. You should refer to the following two tables below (Fig. 4.18 and Fig. 4.13 of the textbook) for implementing the Control and ALUControl units:

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.

	ALUOp		Funct field						
	ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation
lw, sw	0	0	X	X	X	X	X	X	0010
beq	X	1	X	X	X	X	X	X	0110
R-Type instruction	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.

This assignment utilizes the `std::bitset<N>` class to express low-level binary values, so all the inputs and outputs of the combinational circuits also utilize the `std::bitset<N>` class. Accordingly, your implementation of the methods should operate on the objects of the `std::bitset<N>` class to achieve the desired behaviors.

Sequential Circuits as Classes

Unlike combinational circuits, sequential circuits must maintain some internal state over time as their output(s) depend on not only the current input(s), but also the internal state. One way to track the internal state would be to utilize static variables within a function/method; however, the static variables do not allow a single function to be used for multiple hardware components as they get associated with their functions. For example, employing the static variables prevents us from using the same function for implementing both the instruction and data memories.

Alternatively, you will be implementing the sequential circuits as classes rather than methods/functions. A class can maintain a time-varying internal state using its attribute(s), and the same class can be used for multiple hardware components by simply creating an object of the class for each of the hardware components. In particular, you will implement the registers and the instruction/data memories as classes.

The `RegisterFile` Class

Open and inspect the `RegisterFile.hpp` header file. It defines the `RegisterFile` class which serves as the register file of our single-cycle MIPS CPU. You will quickly notice that the `RegisterFile` class maintains the 32-bit values of the 32 integer registers by defining the registers as its attribute. Again, the `std::bitset<N>` class is used to store the values of the registers.

You will also notice that the `RegisterFile` class defines the `RegisterFile::access` method. The method should be invoked to read the source registers and write a value to the destination register. The parameters of the method model the register file of the single-cycle CPU we discussed during the lectures. The `RegisterFile::access` method takes as input seven parameters and should behave as follows:

- To read the values of the two source registers, you should invoke the `RegisterFile::access` method by setting the value of `regWrite` as 0 and by providing the indices of `$rs` and `$rt` through `readRegister1` and `readRegister2`, respectively. Then, the values of `$rs` and `$rt` get returned through `readData1` and `readData2`, respectively.

- When writing a value to the destination register, you should invoke the `RegisterFile::access` method by setting the value of `regWrite` as 1 and by passing the index and value of `$rd` through `writeRegister` and `writeData`, respectively. The method does not return any return value for writes.

The implementation of the method is available within the `RegisterFile.cpp` source file; however, the implementation is currently empty as implementing the method is part of this assignment.

The `Memory` Class

Now, open and inspect the `Memory.hpp` header file. It defines the `Memory` class which can be used for implementing the instruction and data memories of the single-cycle MIPS CPU (i.e., create/use one `Memory` object as the instruction memory and another `Memory` object for the data memory). The class implements a byte-accessible memory by defining an array of `std::bitset<8>` objects. Similar to the `RegisterFile` class, you should read/write data from/to the memory using the `Memory::access` method.

- To read 32-bit data from the memory, you should invoke the `Memory::access` method by setting the value of the `memRead` parameter to 1 and assigning a valid 32-bit memory address to the `address` parameter. Then, the method should return the 32-bit data through the `readData` parameter.

- When writing some 32-bit data to the memory, the `Memory::access` method expects the value passed through `memWrite` parameter to be one and retrieves the target memory address through the `address` parameter. The data to be written to the target address should be passed through the `writeData` parameter. The method does not return any value 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 `m_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 to the lowest memory address, whereas little-endian places the most-significant bit to the highest memory address instead. Although we mostly focused on little-endian, your implementation of the `Memory::access` method should support both big- and little-endian. We recommend you to analyze the code of the `Memory::printMemory` method on the endianness.

Your Task: Implement the Single-Cycle MIPS CPU

Using the methods and classes which correspond to the combinational and sequential circuits, respectively, you should implement the `SingleCycleCPU::advanceCycle` method. An invocation of the method simulates one clock cycle of the single-cycle CPU. You should implement the method in `SingleCycleCPU.cpp` so that the method can model the single-cycle CPU shown below (Fig. 4.19 in the textbook).

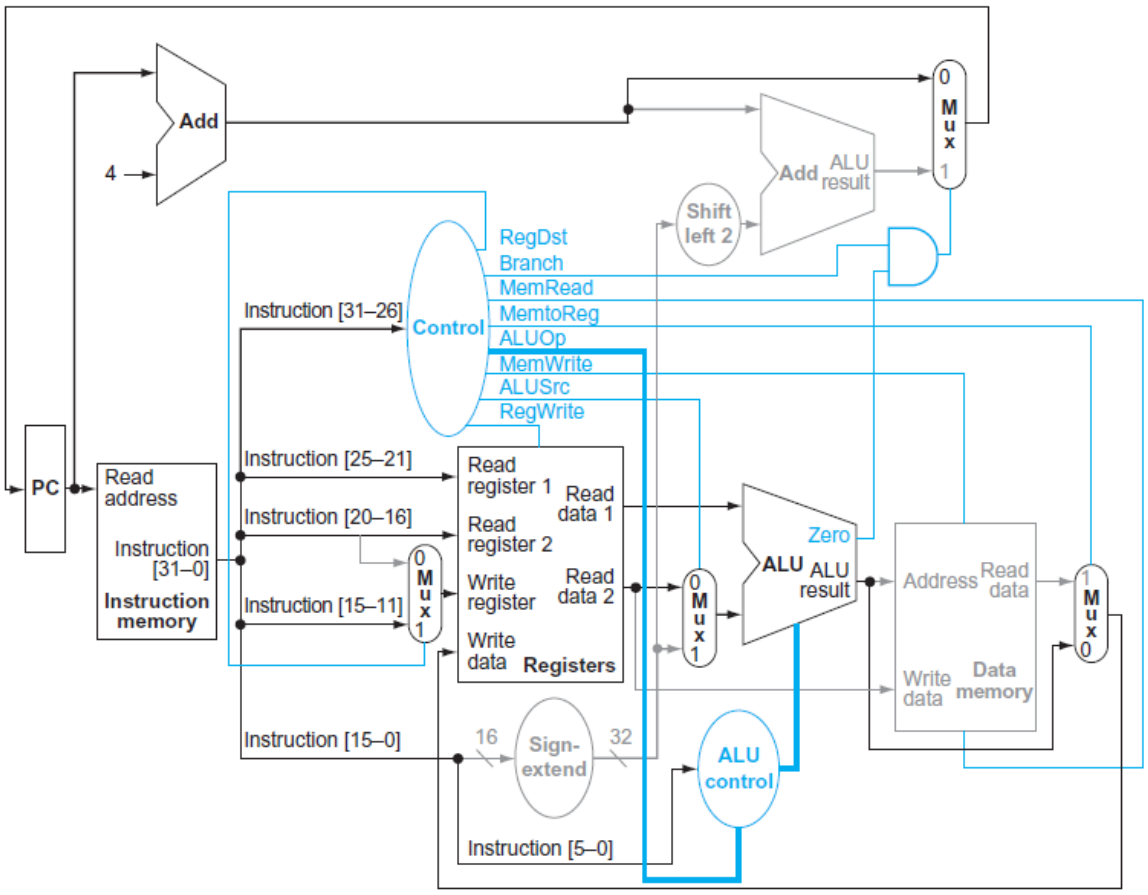

FIGURE 4.19 The datapath in operation for an R-type instruction, such as add $t1,$t2,$t3. The control lines, datapath units, and connections that are active are highlighted.

You will soon realize that implementing the aforementioned combinational and sequential circuits is insufficient to fully implement the single-cycle CPU due to the wires connected between the hardware components. The wires are essentially the paths which the input and output data get transferred between the hardware components. As the methods and classes you implement utilize the `std::bitset<N>` class to express N-bit binary values, we employ the `std::bitset<N>` class to represent a wire whose bit width is N.

One thing to keep in mind is that, although the wires can transfer binary values between the hardware components, they do not assign their values. That is, the binary value a wire transfers between the hardware components is always set by the output port of a hardware component. For example, the value of the wire between the PC register and the Read Address input port of the instruction memory gets set by the PC register during the instruction fetch stage, and the value of the wire will be consumed by the instruction memory through its Read Address input port. The distributed `SingleCycleCPU.hpp`, `Memory.hpp`, and `RegisterFile.hpp` aim to model the low-level behaviors of the hardware components as accurate as possible, so their methods take as input a pointer to an `std::bitset<N>` object rather than the object itself. In this sense, for implementing the wire between the PC register and the instruction memory, the value of the PC register should be assigned to the `std::bitset<N>` object corresponding to the wire and a pointer to the `std::bitset<N>` object should be fed to the `Memory::access` method. Note that the described implementation of the wires and input/output ports is just an example implementation methodology. You are free to implement the wires and ports in whichever way you prefer. However, modifying the `.hpp` header files is not allowed in this assignment.

The `testSingleCycleCPU.cpp` Source File

While you implement the methods for the combinational circuits, the classes for the sequential circuits, and the `SingleCycleCPU::advanceCycle` method for the single-cycle CPU, you can only modify the provided `.cpp` source files (i.e., `SingleCycleCPU.cpp`, `Memory.cpp`, `RegisterFile.cpp`).

You may freely modify the `testSingleCycleCPU.cpp` source file to test your implementations of the methods and classes. The source file takes as input some arguments which allow you to specify the initial value of the PC register, the initial data stored in the instruction and data memories, and the number of clock cycles you wish to simulate/test your single-cycle CPU implementation.

In addition, you can test the individual methods and classes by extending the `main` function of the `testSingleCycleCPU.cpp` source file. For example, if you completed implementing the `SingleCycleCPU::Mux` method and wish to test its correctness, you may add some code to the `main` function similar to:

```
int main(int argc, char **argv) {
    ...
    SingleCycleCPU *cpu = new SingleCycleCPU(...);
    ...
    // test SingleCycleCPU::Mux
    std::bitset<32> input0(0x11111111), input1(0x10101010), output0, output1;
    std::bitset<1> selector0(0), selector1(1);
    cpu->Mux<32>(&input0, &input1, &selector0, &output0);
    cpu->Mux<32>(&input0, &input1, &selector1, &output1);
    assert(output0.to_ulong() == 0x11111111 && output1.to_ulong() == 0x10101010);
    ...
}
```

Example #1: Executing Non-Branch MIPS Instructions

After implementing the `SingleCycleCPU::advanceCycle` method in `SingleCycleCPU.cpp`, check if the method is working well by running the `testSingleCycleCPU.cpp`. There are example files (`regFile`, `instMemFile`, `dataMemFile`) provided for each example in the `tests` folder. You can run your code by:

```
csi3102@csi3102:~/assn2$ make
g++ -std=c++11 -o testSingleCycleCPU testSingleCycleCPU.cpp RegisterFile.cpp
Memory.cpp SingleCycleCPU.cpp
csi3102@csi3102:~/assn2$ ./testSingleCycleCPU
# if you want to check arguments
Usage: ./testSingleCycleCPU initialPC regFileName instMemFileName dataMemFileName
numCycles
csi3102@csi3102:~/assn2$ ./testSingleCycleCPU 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 six clock cycles by invoking the `advanceCycle` method six times (the zero-valued registers are omitted for brevity):

```
==================== 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
```

오타?($pc = 20 아닌가..)

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

You can run your code by:

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

The following example shows the result 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 compress **only the .cpp files** and upload the compressed file to LearnUs. The deadline to submit your code is **11:59pm KST on May 31st, 2023 (Wed)**. **No late submissions will be accepted**.

One way to create a compressed file is shown below.

```
csi3102@csi3102:~/assn2$ tar -czvf assn2-submission.tar.gz *.cpp
```

After creating the compressed file (i.e., `assn2-submission.tar.gz`), upload it to LearnUs.