Assignment #2: MiniMIPS

CSI3102-02 (Architecture of Computers), Spring 2022

Welcome to the second programming assignment for CSI3102-02 @ Yonsei University for Spring 2022! You are given 9 days to complete this assignment. Please carefully read the instructions below, and make sure you upload your assignment to the LearnUs system (https://learnus.yonsei.ac.kr) by the designated deadline.

In this programming assignment, you will parse 32-bit MIPS instructions and execute the parsed MIPS instructions. The 32-bit MIPS instructions are encoded as binary machine instructions which consist of 0s and 1s, and it is up to you to correctly parse the given instructions. After parsing the instructions, you will write some C++ code to properly update the Programmer-Visible State (PVS) of a MIPS CPU core. In short, you will write a simplified version of SPIM, called `MiniMIPS` in this assignment, which can execute a few representative MIPS instructions.

Preparing C++ Compiler

In order to prepare the programming environment, please use the VMware image from the previous assignment. The ID and password you should use is **csi3102**. Log into the csi3102 account by typing csi3102 as the password.

To make sure that you have a proper version of g++, a GNU C++ compiler, launch a terminal after logging in. Then, type the following commands to the terminal:

```
csi3102@csi3102:~$ which g++
/usr/bin/g++
csi3102@csi3102:~$ g++ --version
g++ (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Downloading the Assignment

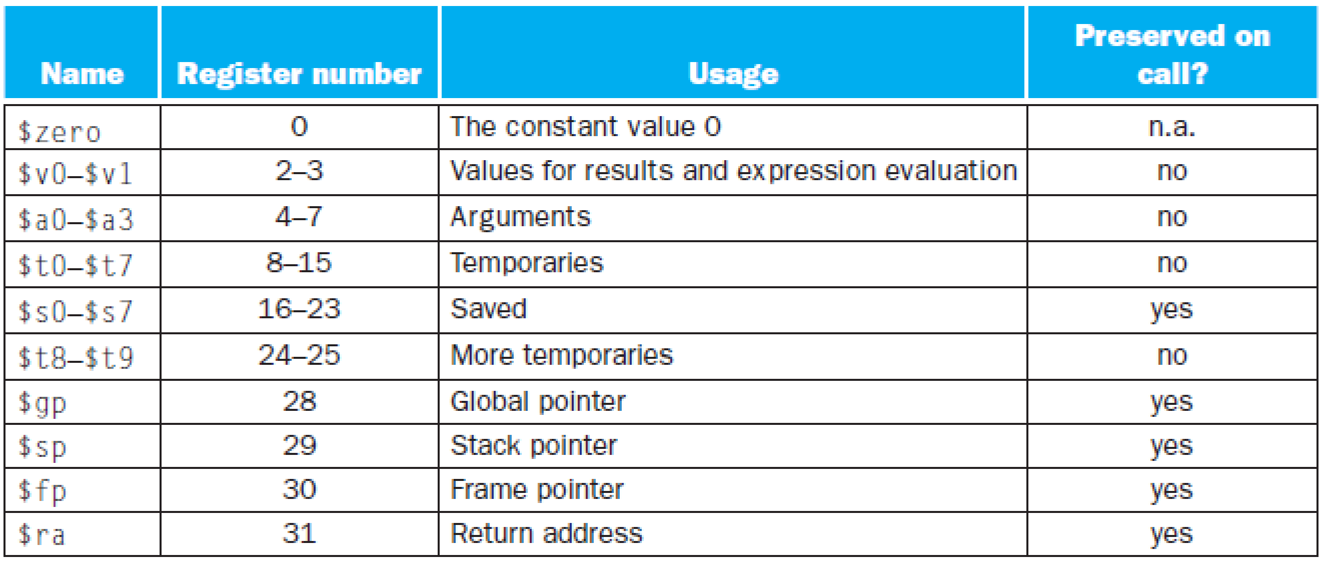
Please download this assignment and perform a sanity check by executing the following commands:

```
csi3102@csi3102:~$ wget https://hpcp.yonsei.ac.kr/files/csi3102/assn2.tar.gz
csi3102@csi3102:~$ tar zvxf ./assn2.tar.gz
csi3102@csi3102:~$ cd ./assn2
csi3102@csi3102:~/assn2$ ls
data.0  data.1  Makefile  MiniMIPS.cc  MiniMIPS.h  testMiniMIPS.cc
```

The `MiniMIPS` Class

Open and inspect `MiniMIPS.h` file. The file defines the MiniMIPS class whose instance consists of 32 general-purpose registers, a program counter (PC) register, and a byte-addressable memory. The registers and memory get initialized within the `MiniMIPS` constructor. Each register is 32-bit wide (i.e., its type is `unsigned int`), and the memory is defined as an `unsigned char` array.

When accessing the 32 registers, you should follow the MIPS register convention which maps the registers to their corresponding register numbers.



Name	Register number	Usage	Preserved on call?
`$zero`	0	The constant value 0	n.a.
`$v0–$v1`	2–3	Values for results and expression evaluation	no
`$a0–$a3`	4–7	Arguments	no
`$t0–$t7`	8–15	Temporaries	no
`$s0–$s7`	16–23	Saved	yes
`$t8–$t9`	24–25	More temporaries	no
`$gp`	28	Global pointer	yes
`$sp`	29	Stack pointer	yes
`$fp`	30	Frame pointer	yes
`$ra`	31	Return address	yes

FIGURE 2.14 MIPS register conventions. Register 1, called `$at`, is reserved for the assembler (see Section 2.12), and registers 26–27, called `$k0–$k1`, are reserved for the operating system. This information is also found in Column 2 of the MIPS Reference Data Card at the front of this book.

Note that the MIPS ISA uses big-endian as the default endianness; both the registers and memory should assume big-endian.

Your Task: The `MiniMIPS::execInstr` Method

Your task is to **implement the `MiniMIPS::execInstr` method within `MiniMIPS.cc` which updates the PVS of a `MiniMIPS` instance with respect to the 32-bit instruction stored in the memory starting from the current value of the PC register**. Your method should fetch the instruction from the memory and update the PVS with respect to the fetched instruction.

Your method should support the following ten MIPS instructions:

- **Two data transfer instructions**: `lw`, `sw`
- **Six arithmetic instructions**: `add`, `sub`, `addi`, `and`, `or`, `slt`
- **Two control instructions**: `j`, `beq`

Each instruction will be stored in the memory as 32-bit binary values with respect to its corresponding MIPS instruction format. Some of the instructions and the MIPS instruction formats are shown in the tables below:

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add `$s1,$s2,$s3`
sub	R	0	18	19	17	0	34	sub `$s1,$s2,$s3`
addi	I	8	18	17	100			addi `$s1,$s2,100`
lw	I	35	18	17	100			lw `$s1,100($s2)`
sw	I	43	18	17	100			sw `$s1,100($s2)`
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

FIGURE 2.6 MIPS architecture revealed through Section 2.5. The two MIPS instruction formats so far are R and I. The first 16 bits are the same: both contain an *op* field, giving the base operation; an *rs* field, giving one of the sources; and the *rt* field, which specifies the other source operand, except for load word, where it specifies the destination register. R-format divides the last 16 bits into an *rd* field, specifying the destination register; the *shamt* field, which Section 2.6 explains; and the *funct* field, which specifies the specific operation of R-format instructions. I-format combines the last 16 bits into a single *address* field.

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

FIGURE 2.20 MIPS instruction formats.

Now, have a look at `testMiniMIPS.cc` which creates a `MiniMIPS` instance, initializes the memory contents, and executes a given number of MIPS instructions. The `MiniMIPS` instance is generated by invoking the `MiniMIPS` constructor with `dataFile` and `initialPC` as input arguments. The two input arguments are given as the command-line arguments to `testMiniMIPS` binary file which you can obtain by typing `make` in the command-line. For example, the following table shows how to generate and invoke the binary file:

```
csi3102@csi3102:~/assn2$ make
g++ -o testMiniMIPS testMiniMIPS.cc MiniMIPS.cc
csi3102@csi3102:~/assn2$ ls
data.0  data.1  Makefile  MiniMIPS.cc  MiniMIPS.h  testMiniMIPS
testMiniMIPS.cc
csi3102@csi3102:~/assn2$ ./testMiniMIPS ./data.0 1000 4
```

Notice that the `testMiniMIPS` binary file takes as input three arguments: `dataFile`, `initialPC`, and `numInstrs`. `dataFile` is the path to the file which you can use to set the initial values of registers and/or memory locations. `initialPC` is the initial value to be used for the PC register. `numInstrs` denotes the number of instructions your `execInstr` method should be invoked. In the example above, `dataFile` is `./data.0`, `initialPC` is `1000`, and `numInstr` is 4.

`dataFile` consists of pairs whose first element is either a register or a memory address, and the second element is the 32-bit hexadecimal value to be assigned to the register or the memory address. If the first element begins with a '$' character, the pair denotes a register initialization. Otherwise, the first element is a decimal value of the starting memory address of the following 4-byte value. The table below shows the contents of `./data.0` which consists of one register initialization and 6 32-bit memory initializations. Note that all the registers and memory addresses which do not appear in the provided `dataFile` get initialized to zero.

```
$8   00000100   # $8 = $t0 = 0x100 = 256
256  0123abcd   # memory[259:256] = 0x0123ABCD
260  abcd0123   # memory[263:260] = 0xABCD0123
1000 8d090000   # memory[1003:1000] = lw $t1, 0x0($t0)
1004 8d0a0004   # memory[1007:1004] = lw $t2, 0x4($t0)
1008 012a5820   # memory[1011:1008] = add $t3, $t1, $t2
1012 ad0b0008   # memory[1015:1012] = sw $t3, 0x8($t0)
```

Now, the code executes the four MIPS instructions which perform `memory[267:264] = memory[259:256] + memory[263:260]` as `initialPC` is set to 1000 and `numInstrs` is set to 4. After you complete implementing the `execInstr` method in `MiniMIPS.cc`, the output should look like:

```
csi3102@csi3102:~/assn2$ make && ./testMiniMIPS ./data.0 1000 4
g++ -o testMiniMIPS testMiniMIPS.cc MiniMIPS.cc
[parseDataFile] $08 = 0x00000100
[parseDataFile] memory[259:256] = 0x0123abcd
[parseDataFile] memory[263:260] = 0xabcd0123
[parseDataFile] memory[1003:1000] = 0x8d090000
[parseDataFile] memory[1007:1004] = 0x8d0a0004
[parseDataFile] memory[1011:1008] = 0x012a5820
[parseDataFile] memory[1015:1012] = 0xad0b0008
[execInstr] PC = 1000, instr = 0x8d090000
[execInstr] PC = 1004, instr = 0x8d0a0004
[execInstr] PC = 1008, instr = 0x012a5820
[execInstr] PC = 1012, instr = 0xad0b0008
[printPC] PC = 1016
[printRegister] $08 = 0x00000100
[printRegister] $09 = 0x0123abcd
[printRegister] $10 = 0xabcd0123
[printRegister] $11 = 0xacf0acf0
[printMemory] memory[267:264] = 0xacf0acf0
```

You may assume that all the instructions which get stored in the memory and get executed by your `execInstr` method are valid MIPS instructions. One exception is the MIPS instructions which write a value to `$zero`. In such a case, your implementation should prevent `$zero` from being overwritten as the MIPS ISA designates `$zero` to have a constant value of 0 at any time.

We strongly recommend you to try implementing various `dataFiles` and test your code with them. Note that you will be submitting only the `MiniMIPS.cc` file, so you may freely modify `testMiniMIPS.cc` depending on your `dataFiles` of interest.

As another example, using the same `dataFile` and `initialPC`, if we change numInstrs from 4 to 2, the output should look like the following:

```
csi3102@csi3102:~/assn2$ make && ./testMiniMIPS ./data.0 1000 2
g++ -o testMiniMIPS testMiniMIPS.cc MiniMIPS.cc
[parseDataFile] $08 = 0x00000100
[parseDataFile] memory[259:256] = 0x0123abcd
[parseDataFile] memory[263:260] = 0xabcd0123
[parseDataFile] memory[1003:1000] = 0x8d090000
```

```
[parseDataFile] memory[1007:1004] = 0x8d0a0004
[parseDataFile] memory[1011:1008] = 0x012a5820
[parseDataFile] memory[1015:1012] = 0xad0b0008
[execInstr] PC = 1000, instr = 0x8d090000
[execInstr] PC = 1004, instr = 0x8d0a0004
[printPC] PC = 1008
[printRegister] $08 = 0x00000100
[printRegister] $09 = 0x0123abcd
[printRegister] $10 = 0xabcd0123
[printRegister] $11 = 0x00000000
[printMemory] memory[267:264] = 0x00000000
```

In particular, notice the following three lines: `[printPC] PC = 1008` as we executed only two MIPS instructions, and `[printRegister] $11 = 0x00000000` and `[printMemory] memory[267:264] = 0x00000000` as the third and fourth instructions are not executed.

The example shown below calculates the 3rd fibonacci number where the 1st and 2nd fibonacci numbers are both 1. `data.1` can support calculating later fibonacci numbers as well, so we recommend you test your code by altering the initial value of $9 and `numInstr`.

```
csi3102@csi3102:~/assn2$ make && ./testMiniMIPS ./data.1 1024 9
g++ -o testMiniMIPS testMiniMIPS.cc MiniMIPS.cc
[parseDataFile] memory[259:256] = 0x00000001
[parseDataFile] memory[263:260] = 0x00000001
[parseDataFile] $08 = 0x00000002
[parseDataFile] $09 = 0x00000003
[parseDataFile] $10 = 0x00000100
[parseDataFile] memory[1027:1024] = 0x11090008
[parseDataFile] memory[1031:1028] = 0x8d4b0000
[parseDataFile] memory[1035:1032] = 0x8d4c0004
[parseDataFile] memory[1039:1036] = 0x016c6820
[parseDataFile] memory[1043:1040] = 0xad4d0008
[parseDataFile] memory[1047:1044] = 0x21080001
[parseDataFile] memory[1051:1048] = 0x214a0004
[parseDataFile] memory[1055:1052] = 0x1000fff9
[execInstr] PC = 1024, instr = 0x11090008
[execInstr] PC = 1028, instr = 0x8d4b0000
[execInstr] PC = 1032, instr = 0x8d4c0004
[execInstr] PC = 1036, instr = 0x016c6820
[execInstr] PC = 1040, instr = 0xad4d0008
[execInstr] PC = 1044, instr = 0x21080001
[execInstr] PC = 1048, instr = 0x214a0004
[execInstr] PC = 1052, instr = 0x1000fff9
[execInstr] PC = 1024, instr = 0x11090008
[printPC] PC = 1056
[printRegister] $08 = 0x00000003
[printRegister] $09 = 0x00000003
[printRegister] $10 = 0x00000104
[printRegister] $11 = 0x00000001
[printMemory] memory[267:264] = 0x00000002
```

Submitting Your Code

After completing your assignment, you should submit your **`MiniMIPS.cc`** to the LearnUs system. The deadline to submit your code is **11:59pm KST on May 19th, 2021 (Thu)**. **No late submissions will be accepted**.