University of Central Florida

Department of Computer Science

CDA3103 Computer Organization

Summer 2020

Programming Project 3 (Tiny MIPS)

DUE 7/17/2020

The Problem

Using C programming language write a program that implements a variant of MIPS. We will call this MIPS version the Tiny MIPS or TMIPS. In this implementation memory (RAM) is split into Instruction Memory (IM) and Data Memory (DM), and in the CPU there is a register file (RF) with 8 registers. The datapath is organized in a 5 stages pipeline. The names of the five stages are:

IF -- Instruction Fetch (Instruction Memory access)
ID/OF -- Instruction Decode/Operand Fetch (Register file read).
EX -- Execute.
DM -- Data Memory access.
WB -- Write Back (Register file write).

IF → ID/OF → EX → DM → WB

Each instruction must pass through each pipeline stage, although some of them just do not execute any action, just pass by the stage.

In between stages there are buffers. The names of these buffers are:

IFID: IF → [IFID] → OF

IDEX: OF→ [IDEX] →EX

EXDM:EX → [EXDM] →DM

DMWB:DM → [DMWB] → WB

All buffers can be implemented with integers arrays of size four or using a struct similar to the one used for the instruction register(IR). (the struct will be described shortly).

The instruction Format has four fields:

op	rx	ry	rz

IM must be implemented using an integer array (size 500)
In IM the instructions requires 4 array elements to be stored.
(We are simulating that a word has four bytes)

For example: The instructions
lw RF[5], 0, 8 // load the contents of memory location 8 into register 5.
lw RF[3], 0,10 // load the contents of memory location 10 into register 5.
add RF[7], RF[3],RF[5] //meaning RF[7] ← RF[3] + RF[5]

They will be stored in the instruction memory as:

	0	1	2	3	4	5	6	7	8	9	10	11	
IM	1	5	0	8	1	3	0	10	2	7	3	5	…..

Instruction 1 Instruction 2 Instruction 3

Therefore, in the IF the PC must be incremented by four to access the next instruction.
(PC = PC + 4)

DM must be implemented using an integer array (size 16). In DM each data values uses just one array element. This is done for simplicity and would be equivalent to say that the DM word is stored in one array element.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
DM	2	0	0	17	6	0	0	13	2	7	3	5	0	0	0	0

RF must be implemented using an integer array (size 8)

	0	1	2	3	4	5	6	7
RF	0	0	0	0	0	0	0	0

In RF, register RF[0] is the only register that can be used for I/O operations.

The Instruction Register (IR) is capable to hold one instruction.

Hint: Implementing a struct for your Instructions Register greatly simplifies this program.
Example:

```
typedef struct {
        int opcode, rx, ry, rz;
} Instruction;

Instruction IM[MAXPROGRAMSIZE];
```

You must implement the basic instruction set architecture (ISA) of the TMIPS:

Opcode	instruction	meaning	description	
1	lw	load word	lw Rx 0 addr	
2	add	add	add Rx Ry Rz	
3	sw	store word	sw Rx 0 addr	
4	sub	substract	sub Rx Ry Rz	
5	sio1	input	sio1 Rx 0 5	// Only work with R0.
6	sio2	output	sio2 Rx 0 9	// Only work with R0.
7	sio3	end of program	sio3 0 0 0	
8	jmp	jump uncondicional	jmp 0 0 addr	
9	beqz	If Rx = 0 {pc = pc + 4}	beq Rx 0 0	// Only work with R0.

Note: You must do the mapping between addr and the variable the memory location represents.

Each piece of the architecture must be accurately represented in your code (Instruction Register, Program Counter, Memory Address Registers, Instruction Memory, Data Memory, Memory Data Registers, Register, and buffers in between stages). Your Program Counter will begin pointing to the first instruction of the program (PC = 0).

Input Specifications

Your simulator must run from the command line with a single input file as a parameter to main. This file will contain a sequence of instructions for your simulator to store in "Instruction Memory" and then run via the IF-OF-EX-DM-WB. In the input file each instruction is represented with four integers: the first one represents the opcode and other 3 represent x, y z. Depending on the instructions, x, y, and z represents register indices, a memory address, a device number, and some times are not used when executing some instructions.

The input file for a program would be:

5 0 0 5 **6 0 0 9** 3 0 0 0 **5 0 0 5** 6 0 0 9 **3 0 0 3** 1 3 0 0 **1 4 0 1** 2 5 3 4 **3 5 0 2** 1 0 0 2 **7 0 0 0**

Your program must read in the file and store it in IM. Then your program must print out the program in assembly language using the ISA (print only the right hand side of the two columns). For example:

5 0 0 5	sio1 0 0 5
6 0 0 9	sio2 0 0 9
3 0 0 0	sw 0 0 0
5 0 0 5	sio1 0 0 5
6 0 0 9	sio2 0 0 9
3 0 0 3	sw 0 0 3
1 3 0 0	lw 3 0 0
1 4 0 1	lw 4 0 3
2 5 3 4	add 5 3 4
3 5 0 2	sw 5 0 2
1 0 0 2	lw 0 0 2
7 0 0 0	halt

Output Specifications

Your simulator should provide output according to the input file. Along with this output your program should provide status messages identifying details on the workings of your simulator. Output text does not have to reflect my example word-for-word, but please provide detail on the program as it runs in a readable format that does not conflict with the actual output of your simulator. After each instruction print the current state of the Program Counter, Register File, Data Memory and all in between buffers. The INPUT instruction is the only one that should prompt an interaction from the user.

Example:

Assembling Program…
Program Assembled. (print the program here)

Run.

PC = 0 | RF = [0, 0, 0, 0, 0, 0, 0, 0] | DM = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

IFID = [0, 0, 0, 0] | IDEX = [0, 0, 0, 0] | EXDM = [0, 0, 0, 0] | DMWB = [0, 0, 0, 0]

/* input value (sio1 0 0 5)*/
X

PC = 4 | RF = [X, 0, 0, 0, 0, 0, 0, 0] | DM = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

IFID = [5, 0, 0, 5] | IDEX = [5, 0, 0, 0] | EXDM = [5, 0, 0, X] | DMWB = [5, 0, 0, X]

/* outputting R0 to screen (sio 0 0 9) */
X

PC = 8 | RF = [X, 0, 0, 0, 0, 0, 0, 0] | DM = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

IFID = [6, 0, 0, 9] | IDEX = [6, X, 0, 0] | EXDM = [6, X, 0, 0] | DMWB = [6, X, 0, 0]

/* storing R0 to memory location 0 (sw 0 0 0) */

R0 = X.

PC = 12 | RF = [X, 0, 0, 0, 0, 0, 0, 0] | DM = [X, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

IFID = [3, 0, 0, 0] | IDEX = [3, X, 0, 0] | EXDM = [3, X, 0, 0] | DMWB = [3, X, 0, 0]

… etc

Program complete.

Grading

Your simulator will be graded on the above criteria. Your program should compile and run from the command line with one input file parameter. Please note that your program will not just be graded on whether or not it runs successfully; accurate simulation and a thorough demonstration of your understanding on the workings of this architecture will constitute a large portion of this grade. As that is the case it is in your best interest to comment your program in a concise and readable way. However, if your program does not run or compile the maximum points possible will be 30.

Note: you should use two MARs, MAR1 for IM and MAR2 for DM.

TMIPS DATAPATH for instruction add Rx Ry Rz:

INSTRUCTION FETCH

```
MAR1 ← PC
PC ← PC+4
IR.op  ←IM [MAR1]          // IM stands for Instruction Memory (program memory)
IR.x   ← IM [MAR1 + 1]     // IM stands for Instruction Memory (program memory)
IR.y   ← IM [MAR1 + 2]     // IM stands for Instruction Memory (program memory)
IR.z   ← IM [MAR1 + 3]     // IM stands for Instruction Memory (program memory)

IFID[0] = IR.op
IFID[1] = IR.x
IFID[2] = IR.y
IFID[3] = IR.z
```

INSTRUCTION DECODE/ OPERAND FETCH

```
idcu ← IFID[0]
idx  ← IFID[1]
idy  ← IFID[2]
idz  ← IFID[3]

if (idcu == add) {
  v1 = RF[idy]
  v2 = RF[idz]
  IDEX[0] ← cu
  IDEX[1] ← idx
  IDEX[2] ← v1
  IDEX[3] ← v2
}
```

EXECUTE

```
excu ← IDEX[0]
exx  ← IDEX[1]
exy  ← IDEX[2]   //value1
exz  ← IDEX[3]   //value2

if (excu == add) {
  value3 ← exy + exz            //value3 = value1 + value2

   EXDM[0] ← excu
   EXDM[1] ← exx
   EXDM[2] ← 0
   EXDM[3] ← value3
}
```

DATA MEMORY

```
   dmcu ← EXDM[0]
   dmx  ← EXDM[1]
   dmy  ← EXDM[2]
   dmz  ← EXDM[3]

if (dmcu == add) {
   DMWB[0] ← dmcu
   DMWB[1] ← dmx
   DMWB[2] ← dmy
   DMWB[3] ← dmz
}
```

WRITE BACK

```
   wbcu ← DMWB[0]
   wbx  ← DMWB[1]
   wby  ← DMWB[2]
   wbz  ← DMWB[3]

if (wbcu == add) {
   RF[wbx] ← wbz
 }
```

You have to implement all other TMIPS instructions. The instruction sw and lw were explained in class. These are the instructions you have to implement.

Opcode	Instruction	Description
1	lw Rx 0 addr	Rx ← DM[addr]
2	add Rx Ry Rz	Rx ← Ry + Rz
3	sw Rx 0 addr	DM[addr] ← Rx
4	sub Rx Ry Rz	Rx ← Ry - Rz
5	sio1 Rx 0 5	Rx ← Keybord
6	sio2 Rx 0 9	screen ← Rx
7	sio3 0 0 0	halt ← 0
8	jmp 0 0 addr	PC ← addr
9	beqz Rx 0 0	if Rx = 0 {PC ← PC + 4}

Submission

Your program must be submitted as a C file. For example: NameMyProgram.c
Please check and double check your submission.

Submit a readme document indicating how to run your program

You can test your implementation with the program described in the section input specifications.

Then, You have to write a program to multiply two numbers using successive additions.