Tel Aviv University

Faculty Of Engineering – School of Electrical Engineering

Final Project in Computer Structure course 0512.4400

Fall 2022-23

Last Updated: 21/12/2022 (Non-Bonus version)

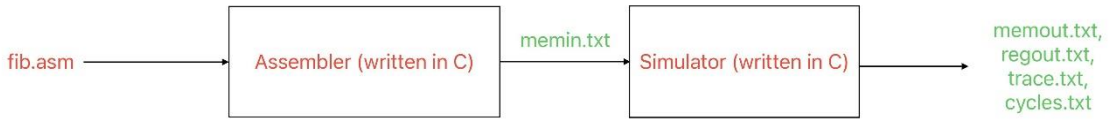
In modern processor development the first stage of processor design is the construction of a simulator in software which simulates the functionality of a processor to test the functionality and viability of a proposed design.

In this project we will develop a simulator for a RISC type processor named SIMP which is like a MIPS processor but far less complex.

The operational clock frequency of the processor is 512 Hz.

This simulator simulates a "slow" processor operating at 512 Hz however there is no need to slow down the speed of the simulator which runs on your computer to the speed of the SIMP processor being simulated. The simulation runs on your personal computer which have a clock speed in the gigahertz range and obviously runs faster than the SIMP.

The diagram below illustrates the project:



The parts of the project which are illustrated in red are what you need to prepare. The output files are shown in green. These output files will be generated automatically by the software.

Registers

The SIMP processor contains 16 registers. The width of each register is 32 bits. The names and the purpose of these registers is shown in the following table:

Register Number	Register Name	Purpose
0	$zero	Constant zero
1	$imm	Sign extended imm
2	$v0	Result value

3	$a0	Argument register
4	$a1	Argument register
5	$a2	Argument register
6	$a3	Argument register
7	$t0	Temporary register
8	$t1	Temporary register
9	$t2	Temporary register
10	$s0	Saved register
11	$s1	Saved register
12	$s2	Saved register
13	$gp	Global pointer (static data)
14	$sp	Stack pointer
15	$ra	Return address

The names of the registers and their function is like what we saw in the lectures and recitations in typical MIPS processors with the following exceptions: The register $imm and $zero are special registers that cannot be written to but can be used as a source operand. The register $zero is zero. The register $imm contains the fixed field $imm as coded in the assembly after performing sign extension.

Instructions that write to these registers do not change their value.

Main Memory, Instruction set architecture

The main memory is 4096 lines each containing 20 bits. In the SIMP processor there are two instruction formats: R format and I format. The R format does not use an immediate value in its instruction so its length is one line. The length of an I format instruction is two lines since the second line is the immediate operand. The first line of an I format instruction is identical to the single line of the R format instruction. Below are diagrams of the R format and I format instructions.

R format			
19:12	11:8	7:4	3:0
opcode	Rd	rs	rt

I format			
19:12	11:8	7:4	3:0

opcode	rd	rs	rt
imm			

The program counter (PC) resister is 12 bits wide. After the fetching of a R format instruction the PC will advance by one. After the fetching of an I type instruction the PC will increase by two.

The supported opcodes for this processor and their meaning are shown in the table below:

Opcode Number	Name	Meaning
0	add	R[rd] = R[rs] + R[rt]
1	sub	R[rd] = R[rs] – R[rt]
2	mul	R[rd] = R[rs] * R[rt]
3	and	R[rd] = R[rs] & R[rt]
4	or	R[rd] = R[rs] | R[rt]
5	xor	R[rd] = R[rs] ^ R[rt]
6	sll	R[rd] = R[rs] << R[rt]
7	sra	R[rd] = R[rs] >> R[rt], arithmetic shift with sign extension
8	srl	R[rd] = R[rs] >> R[rt], logical shift
9	beq	if (R[rs] == R[rt]) pc = R[rd]
10	bne	if (R[rs] != R[rt]) pc = R[rd]
11	blt	if (R[rs] < R[rt]) pc = R[rd]
12	bgt	if (R[rs] > R[rt]) pc = R[rd]
13	ble	if (R[rs] <= R[rt]) pc = R[rd]
14	bge	if (R[rs] >= R[rt]) pc = R[rd]
15	jal	R[rd] = next instruction address, pc = R[rs]
16	lw	R[rd] = MEM[R[rs]+R[rt]], with sign extension
17	sw	MEM[R[rs]+R[rt]] = R[rd] (bits 19:0)
18	halt	Halt execution, exit simulator

Cycle execution time

Every access to main memory takes one clock cycle. The number of cycles that it takes to execute an instruction is the number of accesses to memory and can be 1,2 or three clock cycles.

An R format instruction and except sw and lw will take one clock cycle. (the time it takes to bring the instruction from memory).

An I format instruction and not lw and sw and use $imm as one of the source operands will take two clock cycles. In the first clock cycles the opcode is loaded and in the second cycle the constant is loaded. In the event that the instruction is lw or sw it takes an additional clock cycle to read or write the data to/from memory (meaning a total of 3).

The Simulator

The simulator simulates the **fetch-decode-execute loop**. At the beginning of the run the register "PC"=0. At the beginning of every iteration of this loop the next instruction is fetched, decoded and executed according to the instructions' contents. At the end of the execution the PC is set to the next instruction in sequence (either 1 or two are added in accordance to the type of instruction fetched as described earlier) unless a jump is executed and the PC is set to the target of the jump instruction. **The run concludes when the instruction HALT is executed.**

The simulator is written in the C programming language and will be run from a command line application that receives 5 command line parameters as written in the following execution line.

sim.exe memin.txt memout.txt regout.txt trace.txt cycles.txt

Input Files

memin.txt is an input text file that contains the main memory (RAM) at the beginning of the run. Each row in the file contains one memory location staring from location zero represented as 5 hexadecimal digits(equivalent to 20 bits). In the event that the file size is less than 4096 rows, the remaining memory locations are assumed to be zero. One can assume that the file is valid.

Output Files

memout.txt is an output file in identical format to memin.txt that contains the contents of the memory at the end of the run.

regout.txt is an output file that contains the contents of the registers R2-R15 at the end of the run (you must not print out registers R0 and R1). Each row will contain 8 hexadecimal digits representing a 32-bit number (the contents of those respective registers).

trace.txt is an output file that contains a row of text for each instruction that was executed by the processor in the following format:

PC INST R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15

This row is printed in hexadecimal digits. The PC (current program counter) contains 3 such digits. The INST field contains the hexadecimal representation of the current instruction (5 digits) and the contents of the registers, as 8 hexadecimal digits, **prior to execution** of the instruction which means that the result of the execution of a particular instruction is seen on the next line.

The R0 field contains 8 hexadecimal digits equal to zero always. The R1 field contains 8 hexadecimal digits equal to zero (0) if the instruction is in R format OR the value of the **immediate field after sign extension to 32 bits** if the instruction is in I format.

cycles.txt is an output file which contains the number of clock cycles that transpired during the run(in decimal).

Assembler

In order to make it easier to program the processor and create the memory image in the input file memin.txt we will write in this project the assembler program. The assembler will be written in the C programming language and will translate the assembler code written in text in assembler format to machine code. One can assume that the input file is valid.

Just like the simulator the assembler is executed using the command line as shown below:

asm.exe program.asm memin.txt

The input file **program.asm** contains the assembly program, the output file file **memin.txt** contains the memory image (as described above) and is input to the simulator.

Each code statement in the assembly file contains all 5 parameters in the instruction that the first parameter is the opcode, and the parameters are separated by commas. After the last parameter one may add the sign # followed by a comment. An example is shown below.

```
# opcode, rd, rs, rt, imm
add $t3, $t2, $t1, 0          # $t3 = $t2 + $t1
add $t1, $t1, $imm, 2         # $t1 = $t1 + 2
add $t1, $imm, $imm, 2        # $t1 = 2 + 2 = 4
```

In each instruction there are three options for the immediate field imm.:

- A decimal number either positive or negative
- A hexadecimal number which starts with 0x followed by the hexadecimal digits
- As a symbol (starts with a letter). This indicated that this is a label. The label in the code is indicated by its name followed by a semicolon

To support labels the assembler goes through the code in two passes. In the first pass the address of each label is stored. In the second pass in any location that there is a label in the immediate field it is replaced by an address of the label computed during the first pass. Notice that the register $imm in the various instructions such as the beq instruction when the condition is always satisfied then use and unconditional jump.

Example:

```
        bne $imm, $t0, $t1, L1      # if ($t0 != $t1) goto L1
                                    # (reg1 = address of L1)
        beq $imm, $zero, $zero, L2  # jump to L2 (reg1 = address L2)
L1:     sub $t2, $t2, $imm, 1       # $t2 = $t2 – 1 (reg1 = 1)
L2:     add $t1, $zero, $imm, L3    # $t1 = address of L3
        beq $t1, $zero, $zero, 0    # jump to the address specified in reg $t1
L3:     jal $ra, $imm, $zero, L4    # function call L4, save return addr in $ra
        halt $zero, $zero, $zero, 0 # halt execution
```

In addition to the instructions the assembler supports another instruction that allows to set the value of a row directly in the memory image in this format:

.word address data

where **address** is the address of the word and the **data** is the data to be written. These fields can be written in decimal OR hexadecimal preceded by the 0x prefix.

Example:

```
.word 256 1              # set MEM[256] = 1
.word 0x100 0x1234A      # MEM[0x100] = MEM[256] = 0x1234A
```

Assumptions

1. One may assume the maximum line size of the input file is 300.
2. The maximum label size is 50.
3. Labels must begin with a letter followed by either letters or numbers.
4. Ignore whitespace (spaces,tabs). There may be spaces or tabs and the input is valid.
5. Support hexadecimal digits in lower case and upper case.
6. Follow the forum on the moodle for updates and to get answer to questions.

Submission Instructions:

1. The assignment is to be submitted in pairs in the submission box that will be provided on the moodle. **The deadline for submission is 15.1.23 at 23:59 (15th January, 2023).**
2. Please enter the ID numbers of you and your partner, in the excel file shared on the moodle.
3. We reserve the right to detect and punish plagiarism. Your programs may be automatically checked.
4. You must submit a documentation file in pdf format. Please name this file id1_id2.pdf. the id's are your id number. Please include in the documentation file your names and id numbers. In this file you may provide details about how to run your code, what works and what doesn't, etc.
5. The project will be written in the C programming language. The assembler and simulator are different programs each in a different library that compiles and runs separately.
6. The code must contain comments that explains its operation.
7. You may use whatever IDE you prefer. **We will be using Visual Studio 2017, Community Edition to test your codes.** So before you submit you project, please make sure that it compiles and runs correctly in the Visual Studio 2017, Community Edition. During

submission, please submit your source code along with your compiled executables for each library you built **separately.**

8. We will be providing installation instructions specifically for Visual Studio 2017 on the moodle. Feel free to use them.
9. Your project will tested using programs that are not available to you. The project will be also tested using test programs that you must prepare. You must place comments in the assembly code. Please also submit a program named **fibo.asm** that stores starting at memory location 0x100 the Fibonacci series. When an overflow occurs, stop the program. The test program should be placed in a folder titled fibo.

Each of these above libraries will contain a copy of the executable files **sim.exe** and **asm.exe** and the input and output files of the test programs using the assembler and simulator. For example, in the fibo directory there will be the following files:

sim.exe, asm.exe, fibo.asm, memin.txt, memout.txt, trace.txt, cycles.txt

It is important to verify that the assembler and the simulator runs in the cmd window and not just from visual studio. Also, one must verify that you are using the proper file names in your command line. We will test your code using batch files that can detect incorrect command line arguments.