Tel Aviv University

Faculty Of Engineering – School of Electrical Engineering

Final Project in Computer Structure course 0512.4400

Fall 2022-23

Last Updated: 20/12/2022

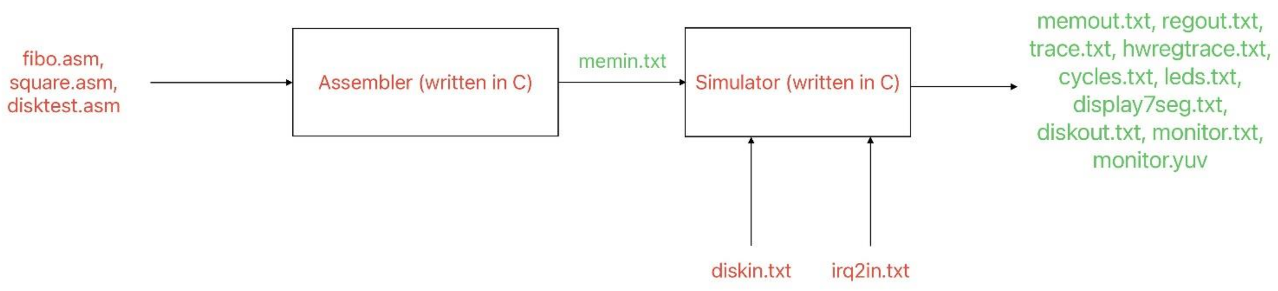
In modern processor development the first stage of processor design is the construction of a simulator in software which simulates the functionality of a processor to test the functionality and viability of a proposed design.

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.

This simulator will simulate a SIMP processor and several input/output types such as lights, 7 segment number display, monochrome monitor with a 256X256 resolution, hard disk. 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:


fibo.asm, square.asm, disktest.asm
Assembler (written in C)
memin.txt
Simulator (written in C)
memout.txt, regout.txt, trace.txt, hwregtrace.txt, cycles.txt, leds.txt, display7seg.txt, diskout.txt, monitor.txt, monitor.yuv
diskin.txt
irq2in.txt

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	reti	PC = IORegister[7]
19	in	R[rd] = IORegister[R[rs] + R[rt]]
20	out	IORegister [R[rs]+R[rt]] = R[rd]

21	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).

Input/Output

The processor supports input/output using the instructions in and out that access an array of hardware registers and is shown in the table below. The initial values of these registers after reset is zero.

IORegister Number	Name	number bits	Meaning
0	irq0enable	1	IRQ 0 enabled if set to 1, otherwise disabled.
1	irq1enable	1	IRQ 1 enabled if set to 1, otherwise disabled.
2	irq2enable	1	IRQ 2 enabled if set to 1, otherwise disabled.
3	irq0status	1	IRQ 0 status. Set to 1 when irq 0 is triggered.
4	irq1status	1	IRQ 1 status. Set to 1 when irq 1 is triggered.
5	irq2status	1	IRQ 2 status. Set to 1 when irq 2 is triggered.
6	irqhandler	12	PC of interrupt handler
7	irqreturn	12	PC of interrupt return address
8	clks	32	cyclic clock counter. Starts from 0 and increments every clock. After reaching 0xffffffff, the counter rolls back to 0.
9	leds	32	Connected to 32 output pins driving 32 leds. Led number i is on when leds[i] == 1, otherwise its off.

10	display7seg	32	Connected to 7-segment display of 8 letters. Each 4 bits displays one digit from 0 – F, where bits 3:0 control the rightmost digit, and bits 31:28 the leftmost digit.
11	timerenable	1	1: timer enabled 0: timer disabled
12	timercurrent	32	current timer counter
13	timermax	32	max timer value
14	diskcmd	2	0 = no command 1 = read sector 2 = write sector
15	disksector	7	sector number, starting from 0.
16	diskbuffer	12	Memory address of a buffer containing the sector being read or written. Each sector will be read/written using DMA in 128 words
17	diskstatus	1	0 = free to receive new command 1 = busy handling a read/write commad
18-19	reserved		Reserved for future use
20	monitoraddr	16	Pixel address in frame buffer
21	monitordata	8	Pixel luminance (gray) value (0 – 255)
22	monitorcmd	1	0 = no command 1 = write pixel to monitor

Interrupts

The SIMP processor supports 3 interrupts: irq0, irq1, irq2.

Interrupt 0(irq0) is connected to the timer and the assembly code can control how often an interrupt can occur.

Interrupt 1(irq1) is connected to the simulated hard disk. This interrupt informs the processor when it completes a read or a write instruction.

Interrupt2(irq2) is connected to an external line attached to the processor. An input file to the simulator determines when this interrupt is enabled.

In the clock cycle when the interrupt is received, one of the registers irq0status, irq1status, irq2status is turned on. If during the same clock cycle more than one interrupt is received, the corresponding status are enabled during that clock cycle.

During each clock when an instruction is executed the processor checks the signal:

irq = (irq0enable & irq0status) | (irq1enable & irq1status) | (irq2enable & irq2status)

Instructions that take multiple clock cycles to execute, no instruction is halted upon receipt of an interrupt. The irq is checked only prior to execution of a new instruction.

In the event that an irq is equal to 1 and the processor is not currently running an interrrupt service routine(ISR), the processor will go to an interrupt service routine which is located at an address whose value is in the register “irqhandler”. This address is placed in the PC after the previous PC is stored in the register “irqreturn”.

In the event that upon receipt of an interrupt the processor is executing an ISR the processor will ignore the request until completion of the ISR. After completion of the ISR, the irq will be checked again and if neccesary the ISR will, at that time run again, to service the new interrrupt.

The code of the ISR will test the bits of irq status and upon start of the ISR will turn these bits off. At the conclusion of execution of the ISR the instruction “reti” is executed which sets the PC to “irqreturn” which is the return address. This is the final address that the software executed prior to receipt of the interrupt.

Timer

The SIMP processor supports a 32 bit timer connected to the irq interrupt. This interrupt is enabled when timerenable=1

The current value of the timer is stored in a hardware register called timercurrent. This register is incremented every cycle by one.

When timecurrent = timermax (the maximum value of the timer) the “irqstatus” bit is turned on. Instead of increasing the timer count, this value is reset to zero.

LED

The SIMP processor is connected to 32 lights. The assembly code turns the light on or off by writing a 32 bit value to hardware register leds where the bit 0 controls LED0 (rightmost one) and bit 31 the LED31, respectively.

Monitor

The SIMP processor is connected to a monochrome monitor with a resolution of 256X256 pixels. Each pixel is represented by 8 bits which represents the grayscale luminance of the pixel. The value zero(0) is black and the value 255 is white. The remaining values go from black to white in a proportional manner.

In the monitor there is frame buffer array of size 256X256 which contains the values of the pixels that are currently shown on the screen. At the commencement of the system all the values of this array are zero. Each row of this array corresponds to a row on a monitor. Row 0 is at the top and column 0 in the row is the leftmost pixel on the monitor. The search of pixels are from left to right.

The register “monitoraddr” contains the offset in the buffer of the pixel to which the processor can write to. The register “monitordata” contains the value to write to that pixel. To write a pixel: use an “out” instruction to set this register to 1 which writes the data to the monitor. Reading the “monitorcmd” register with the “in” instruction will return the value of 0.

Hard Disk

The SIMP processor is connected to a hard disk containing 128 sectors where each sector contains 128 lines of 20 bits. The disk is connected to irq1 and uses DMA to copy a sector to/from memory.

The initial values of the disk are given in the input file diskin.txt. The contents of the disk at the end of the run is in the file diskout.txt.

Prior to issue of a write or read instruction the assembly code checks whether the disk is free to handle a new request by checking the hardware register diskstatus.

In the event that the disk is available, one writes to the hardware register “disksector”, the sector number that one wishes to read from or write to. In hardware register “diskbuffer” we place the memory address. Afterwards in the hardware register “diskcmd” contains the write or read command.

The read or write time to the disk is 1024 clock cycles. During this time the register “diskstatus” will indicate if the disk is busy.

After 1024 clock cycles the hardware registers “diskstatus” and “diskcmd” will be set to 0, simultaneously, and the disk will set “irq1” to notify that the operation has finished.

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 13 command line parameters as written in the following execution line.

sim.exe memin.txt diskin.txt irq2in.txt memout.txt regout.txt trace.txt hwregtrace.txt cycles.txt leds.txt display7seg.txt diskout.txt monitor.txt monitor.yuv

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.

diskin.txt is an input file that contains the contents of the hard disk at the beginning of the run. Where each row contains 5 hexadecimal digits. If the file size is less than the size of the disk assume that the remainder of the disk is zero.

irq2in.txt is an input file that contains the a list the of clock cycles(in decimal) in ascending order when the external interrupt irq2 is enabled (equal to 1). This interrupt is enabled for one clock cycle and subsequently disabled, unless two interrupts occur sequentially.

These three input files must be present even if they are not used in the code. (ie. If there is no use for the disk, then leave the relevant disk input file empty).

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.

hwregtrace.txt is an output file that includes rows of text for each read or write to a hardware register (using in or out instructions) in the following format:

CYCLE READ/WRITE NAME DATA

CYCLE is the clock cycle number in decimal format.

The next field contains the word READ or WRITE depending on instruction type.

The name field contains the name of the register as shown in the table

The DATA field contains the value written or read in 8 hexadecimal digits.

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

leds.txt is an output file which contains the status of the 32 led's. Whenever the status of any led changes one writes two columns to this file with two numbers separated by a space. The leftmost number is the cycle number followed by the led status represented by 8 hexadecimal digits.

display7seg.txt is an output file that contains the 7-segment display at every clock cycle when the display changes. There are two numbers separated by a space on each line. The numbers are the clock cycle and the value of the display represented by 8 hexadecimal digits.

diskout.txt is an output file in the same format as diskin.txt and contains the contents of the hard disk at the end of the run.

monitor.txt is an output file that contains the values of the pixels on the screen at the end of the run. Each row contains a value of a single pixel (0-255 using 2 hexadecimal digits), when the scan of the screen is from the top to the bottom and left to right. The first row of the file contains the value of the leftmost pixel on the top row. In the event that the number of rows in the file are less than the number of pixels, the remaining pixels are zero.

monitor.yuv is a binary file that contains the same data as in monitor.txt and can be displayed on the monitor using "yuvplayer". This software can be found at this link

https://github.com/Tee0125/yuvplayer

The parameters for this program are size = 256 X 256 and color – Y.

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 submit 3 test programs as listed below.

    a. A program named **fibo.asm** that stores starting at memory location 0x100 the Fibonacci series. When an overflow occurs, stop the program.

    b. Program **square.asm** that draws a square. At memory location 0x100 is the location on the screen of the upper left-hand corner of the square. At location 0x101 is the length of the all the squares edges. The interior of the square is white. The area outside the square is black. You must check that the square can be drawn without going out of the boundaries of the monitor.

    c. A program named **disktest.asm** takes as an input 2 sector numbers and sums the first 8 words (word 0 to word 7) in each sector number. At memory location 0x100 is the result of the first sector and at 0x101 is the result of the second sector. At memory location 0x102 is the larger number of the two.

The test programs are to be submitted in 3 different libraries with the names fibo, square, disktest.

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, diskin.txt, irq2in.txt, memout.txt, regout.txt, trace.txt, hwregtrace.txt, cycles.txt, leds.txt, display7seg.txt, diskout.txt, monitor.txt, monitor.yuv

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.