Aviv University, Faculty of Engineering

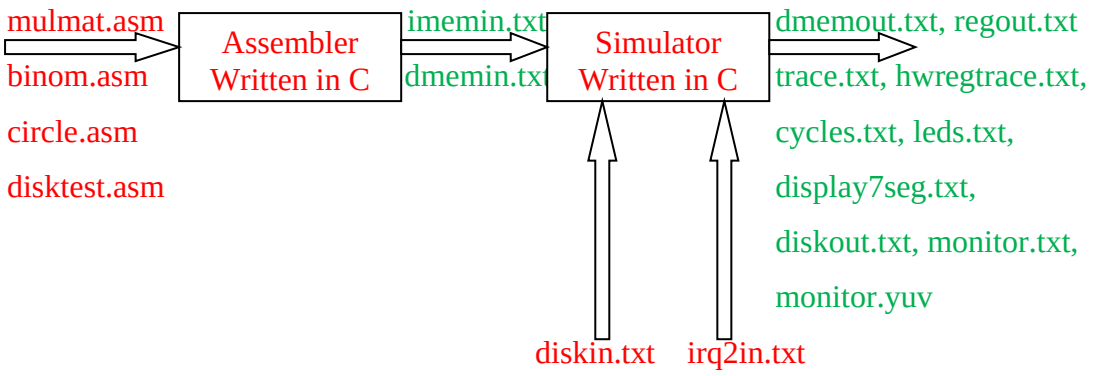
Project in the course: Computer Organization 0512.4400

Winter 2023

In this project we'll exercise the subjects of ISA, Input/Output, as well as our skills in the C language. We'll implement an assembler and a simulator (separate programs), and write programs in assembly language for a RISC processor named SIMP, which is similar to the MIPS processor but simpler.

The simulator will simulate the SIMP processor, as well as several input/output devices: leds, 7-segment-display, monochromatic monitor with 256x256 resolution, and a disk drive. Every instruction in the processor is executed in one clock cycle.

The following diagram illustrates the project:



The parts that you'll write manually are marked in red color, while output files that will be created automatically by the assembler and simulator that you'll write are marked in green color.

Registers

The SIMP processor includes 16 registers, each 32 bits wide. The name, number and role of each register according to the calling conventions is given in the following table:

Register Number	Register Name	Purpose
0	$zero	Constant zero
1	$imm1	Sign extended immediate 1
2	$imm2	Sign extended immediate 2
3	$v0	Result value
4	$a0	Argument register
5	$a1	Argument register
6	$a2	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 register names and roles are similar to what we have seen in the lectures and recitations for the MIPS processor, with one difference: the two registers $imm1, $imm2 are special registers that can't be written, and always contain the immediate ields immediate1, immediate2 (respectively), after performing sign extension, as was coded in the assembly instruction. Register 0 ($zero) is by definition identical to 0. Instructions that write to $zero, $imm1, $imm2 are legal, but don't change their values.

Instruction and Data memories

The instruction memory has a width of 48 bits and a depth of 4096 lines. The PC register is therefore 12 bits, and consecutive instructions are separated by a difference of 1 in the PC (not 4 like in MIPS).

The data memory has a width of 32 bits and a depth of 4096 lines. The address of the data memory is therefore 12 bits wide. In contrast to the MIPS processor, the SIMP

processor does not support byte or short. Every access to the data memory reads or writes a 32-bit wide word.

Intruction set and encoding

The SIMP processor has a single instruction format used to encode all instructions. Every instruction is 48 bits wide, where the bit numbers of every field are given in the following table:

47:40	39:36	35:32	31:28	27:24	23:12	11:0
opcode	rd	rs	rt	rm	immediate1	immediate2

The opcodes supported by the processor and the meaning of each instruction are given in the following table:

Opcode Number	Name	Meaning
0	add	R[rd] = R[rs] + R[rt] + R[rm]
1	sub	R[rd] = R[rs] – R[rt] – R[rm]
2	mac	R[rd] = R[rs] * R[rt] + R[rm]
3	and	R[rd] = R[rs] & R[rt] & R[rm]
4	or	R[rd] = R[rs] | R[rt] | R[rm]
5	xor	R[rd] = R[rs] ^ R[rt] ^ R[rm]
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[rm][low bits 11:0]
10	bne	if (R[rs] != R[rt]) pc = R[rm] [low bits 11:0]
11	blt	if (R[rs] < R[rt]) pc = R[rm] [low bits 11:0]
12	bgt	if (R[rs] > R[rt]) pc = R[rm] [low bits 11:0]
13	ble	if (R[rs] <= R[rt]) pc = R[rm] [low bits 11:0]
14	bge	if (R[rs] >= R[rt]) pc = R[rm] [low bits 11:0]
15	jal	R[rd] = pc + 1 (next instruction address), pc = R[rm][11:0]

16	lw	R[rd] = MEM[R[rs]+R[rt]] + R[rm]
17	sw	MEM[R[rs]+R[rt]] = R[rm] + R[rd]
18	reti	PC = IORegister[7]
19	in	R[rd] = IORegister[R[rs] + R[rt]]
20	out	IORegister [R[rs]+R[rt]] = R[rm]
21	halt	Halt execution, exit simulator

Input/Output

The processor supports Input/Output using the in and out instructions, which access an array of "hardware IO registers", as detailed in the table below. The initial values of the hardware registers on reset are 0.

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 512 byte 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 belongs to the timer, and the assemble code can program the timer to select how often the interrupt will occur.

Interrupt 1 belongs to the simulated hard disk. Using interrupt 1 the disk notifies the processer when it finished performing a read or write command.

Interrupt 2 is connected to an external (to the processor) interrupt line, irq2. An input file to the simulator specifies when the interrupt occurs.

In the clock cycle in which the interrupt is received, the processor turns on one of the registers irq0status, irq1status, irq2status (respectively). In case several interrupt requests arrive in the same clock cycle, multiple bits can be set to one simultaneously.

Every clock cycle, the processor computes and checks the logical signal:

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

In case irq == 1, and provided the CPU is not currently inside the interrupt service routine (ISR), the processor jumps to the ISR, whose address in memory is specified in the hardware register irqhandler. That is to say, in this clock cycle the instruction at address PC = irqhandler is executed instead of the instruction in the original PC. In the same clock cycle, the original PC is saved in the hardware register irqreturn.

[No nested irq support] On the other hand, in case irq == 1 but the CPU is still in the ISR for the previous interrupt request (meaning still has not run the reti instruction), the CPU will ignore the new irq, not jump to the ISR again, and continue to run the code as usual in the old ISR (when the processor returns from the ISR, it'll check irq again and if set, will jump again to the ISR).

The assembly code of the ISR should check the bits of irqstatus, and after servicing the interrupt, clear the bits for the interrupts that were serviced.

Return from the ISR happens using the reti instruction, that will set PC = irqreturn.

Timer

The SIMP processor supports a 32-bit timer, connected to interrupt irq0. It is enabled when timerenable = 1.

The current value of the timer is stored in the hardware register timercurrent. In every clock cycle in which the timer is enable, the timercurrent register is incremented by one.

In the clock cycle in which timercurrent == timermax, the timer hardware sets irqstatus0. In that clock cycle, instead of incrementing timercurrent, it is reset back to zero.

LED lights

The SIMP processor is connected to 32 leds. The assembly code turns on/off leds by writing a 32-bit word to the hardware register leds, where bit 0 turns on or off led number 0 (the rightmost led), and bit 31 controls led 31 (leftmost led).

Monitor

The SIMP processor is connector to a gray monitor (can display only shades of white/black, no color) with a resolution of 256x256 pixels. Every pixel is represented by 8 bits which represent the luminance (gray level) of the pixel, where the value 0 corresponds to a black pixel, and the value 255 corresponds to a white pixel. Any other value in-between represents a gray level between black and white, in a linear fashion.

The monitor has an internal frame buffer with the size 256x256, which represents the pixel values that are currently displayed. At the beginning the frame buffer contains zeroes. The frame buffer contains lines of 256 bytes each that correspond to raster scan of the display from the top to the bottom. Meaning line 0 in the buffer contains the pixels of the top line in the display. In every line, the pixel scanning proceeds from left to right.

Register monitoraddr contains the offset in the buffer of the pixel which the processor wants to write.
Register monitordata contains the pixel value that the processor wishes to write.
Register monitorcmd is used to write a pixel. In the clock cycle in which a pixel is written, an out command is being used to set monitorcmd = 1, and then the pixel value specified by monitordata is written to the display.
A read from monitorcmd using the in instruction will return the value 0.

Disk Drive

The SIMP processor is connected to a disk drive of size 64 kilobytes, which is composed of 128 sectors, each sector with a size of 512 bytes. The disk is connected to interrupt number 1, irq1, and uses DMA to copy the sector contents from memory to the disk or vice versa.

The initial contents of the disk drive is given in the input file diskin.txt, and the contents of the disk at the end of the simulation should be written to the output file diskout.txt.

The assembly code can check that the disk drive is ready to receive a new command by checking the value of the hardware register diskstatus.

Assuming the disk is ready, the assembly code writes to the hardware register disksector the sector number that we wish to read or write, and to the hardware register diskbuffer the memory address of the buffer in main memory. Once those two registers are initialized, a write or read command can be started by writing to the hardware register diskcmd.

The service time of the disk drive for a read or write command is 1024 clock cycles. During this time, the disk copies, using DMA, the contents of the memory buffer to the disk in case of a write command, or vice versa in case of a read command.

As long as 1024 clock cycles have not passed since receiving the command, diskstatus will mark that the disk is busy.

After 1024 clock cycles, in the same cycle diskcmd and diskstatus will be changed to the value 0 to mark that the disk is now free, and the disk will notify an interrupt by turning on irqstatus1.

The Simulator

The simulator simulates the fetch-decode-execute loop. At the beginning of the run PC = 0. Each iteration we bring the next instruction from address PC, decode the instruction according to the encoding, and afterwards execute the instruction. At the end of the instruction we update PC to the value PC + 1 (unless it was updated to another value for example by a jump command or an interrupt). The simulation ends and we exit the simulator once the HALT command is executed.

The simulator will be written in the C programming language and compiled into a command line application that received 14 command line parameters in accordance with the following command line:

sim.exe imemin.txt dmemin.txt diskin.txt irq2in.txt dmemout.txt regout.txt trace.txt hwregtrace.txt cycles.txt leds.txt display7seg.txt diskout.txt monitor.txt monitor.yuv

The file **imemin.txt** is an input file in text format that contains the instruction memory contents at the start of the run. Every line in the file contains the contents of a line in the instruction memory, starting from address 0, with a format of 12 hexadecimal letters. In case the number of lines in the file is smaller than 4096, the assumption is that the rest of the memory above the last address that was initialized in the file, is initialized to 0. It can be assumed that the syntax of the input file is valid.

The file **dmemin.txt** is an input file in text format that contains the data memory contents at the start of the run. Every line in the file contains the contents of a line in the data memory, starting from address 0, with a format of 8 hexadecimal letters. In case the number of lines in the file is smaller than 4096, the assumption is that the rest of the memory above the last address that was initialized in the file, is initialized to 0. It can be assumed that the syntax of the input file is valid.

The file **diskin.txt** is an input file in text format that contains the disk contents at the start of the run. Every line in the file contains the contents of a line in the disk, starting from address 0, with a format of 8 hexadecimal letters (such that each sector is 128 lines). In case the number of lines in the file is smaller than 16384, the assumption is that the rest of the disk above the last address that was initialized in the file, is initialized to 0. It can be assumed that the syntax of the input file is valid.

The file **irq2in.txt** is an input file in text format that contains the clock numbers in which the external interrupt line 2, irq2, was raised to 1, each such cycle in a separate line in ascending order. The interrupt line is set to 1 for a duration of a single clock cycle, and then goes back to 0 (unless in the input file there is an additional line that sets it to 1 in the next clock cycle as well).

The input files should be present even if in your assembly code they are not being used (for example even for an assembly code that does not use the disk drive, an input file diskin.txt should exist. You are allowed to leave its contents empty).

The file **dmemout.txt** is an output text file, in the same format as dmemin.txt, that contains the contents of the data memory at the end of the run.

The file **regout.txt** is an output text file, that contains the values of the registers R3-R15 at the end of the run (note that the constants R0-R2 should not be printed in this file). Every line will be written in 8 hexadecimal letters.

The file **trace.txt** is an output text file, that contains a line of text for every instruction 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

Every field is printed in hexadecimal letters. The PC is the Program Counter of the instruction (printed in 3 hex letters). INST is the instruction encoding as was read form the instruction memory (12 letters), and afterwards the contents of the registers **before** the execution of instruction (meaning the result of execution can be seen only in the registers of the next line). Each register is printed with 8 digits.

In the R0 field 8 zeroes should be written. R1 and R2 should print the contents of the immediates after sign extension was performed to 32 bits. For example if bits 11:0 in the instruction were all '1', the value for R1 will be FFFFFFFF.

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

CYCLE READ/WRITE NAME DATA

Where the field CYCLE is the clock cycle number in decimal.

The next field contains READ or WRITE depending on whether a read or write was performed to the hardware register.

The field NAME contains the hardware register name as detailed in the table of I/O registers.

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

The output file **cycles.txt** contains the number of clock cycles it took the program to run.

The output file **leds.txt** contains the status of the 32 leds. In each clock cycle in which one of the leds changes (turns on or off), a line is written to the file with two numbers

and a space in-between: the left number is the clock cycle in decimal, and the right number is the status of all 32 leds in 8 hexadecimal numbers.

The file **display7seg.txt** contains the 7-segment display. In every clock cycle in which the display changes, a line is written to the file with two numbers and a space in-between: the left number is the clock cycle in decimal, and the right number is the 7-segment status in 8 hexadecimal numbers.

The file **diskout.txt** is an output file, with the same syntax as memin.txt, that contains the contents of the disk drive at the end of the run.

The file **monitor.txt** contains the values of the pixels on the screen at the end of the run. It is a text file, where every line contains the value of a single pixel (8 bits) in two hexadecimal numbers. The scanning of the display is performed from top to bottom and then from left to right. For example the first line in the file contains the pixel value at the top-left corner of the monitor, and the last line contains the pixel value at the bottom-right corner.

The file **monitor.yuv** is a binary file (not a text file) that contains the same data as monitor.txt, and can be displayed graphically on the PC monitor using the program yuvplayer:

https://github.com/Tee0125/yuvplayer/releases/download/2.5.0/yuvplayer-2.5.zip

Where the parameters are chosen such that size = 256x256 and color = Y.

The Assembler

To be able to conveniently program the CPU and generate the initial memory images in the files imemin.txt and dmemin.txt, we'll also write an assembler. The assembler will be written in the C programming language, and will program the assemble program (written in text) into SIMP assembly language. It can be assumed that the input file syntax is valid.

Similar to the simulator, the assembler is a command line application, with the following run command line:

```
asm.exe program.asm imemin.txt dmemin.txt
```

The input file **program.asm** contains the assembly program, the output file **imemin.txt** contains the initial instruction memory image, and the output file **dmemin.txt** contains the initial data memory image. The output files of the assembler are later used as input files for the simulator.

Each line of code in the assemble file contains all the 7 parameters in the encoding of the instruction, where the first parameter is the opcode, and afterwards the parameters are separated by comma. After the last parameters it is allowed to add the symbol # and then add a comment on the right side, for example:

```
# opcode, rd, rs, rt, rm, imm1, imm2
add $t3, $t2, $t1, $t0, 0, 0                # $t3 = $t2 + $t1 + $t0
add $t1, $t1, $imm1, $zero, 2, 0            # $t1 = $t1 + 2
add $t1, $imm1, $imm1, $imm1, 2, 0          # $t1 = 2 + 2 + 2 = 6
sub $t1, $imm1, $imm2, $imm2, 2, 3          # $t1 = 2 – 3 – 3  = -4
```

In each instruction, there are 3 options for the immediate fields:

- A decimal number, positive or negative.
- A hexadecimal number that begins with 0x and then contains hexadecimal digits.
- A label, which is a symbolic name starting with a letter and ending with :

Examples:

```
        bne $zero, $t0, $t1, $imm1, L1, 0          # if ($t0 != $t1) goto L1
                                                   # (reg1 = address of L1)
        add $t2, $t2, $imm1, $zero, 1, 0           # $t2 = $t2 + 1 (reg1 = 1)
        beq $zero, $zero, $zero, $imm1, L2, 0      # jump to L2 (reg1 = address L2)
L1:
        sub $t2, $t2, $imm1, $zero, 1, 0           # $t2 = $t2 – 1 (reg1 = 1)
L2:
        add $t1, $zero, $imm1, $zero, L3, 0        # $t1 = address of L3
        beq $zero, $zero, $zero, $t1, 0, 0     # jump to the address specified in t
L3:
        jal $ra, $zero, $zero, $imm1, L4, 0    # function call L4, save return addr in
$ra
        halt $zero, $zero, $zero, $zero, 0, 0  # halt execution
L4:
        beq $zero, $zero, $zero, $ra, 0, 0     # return from function in address in $ra
```

To support labels, the assembler performs two passes on the code. In the first pass we remember the addresses of all labels, and in the second pass in each place that was a use of a label in the immediate field, we replace it with the actual address of the label as calculated in the first pass.

Take note of the use of the special registers $imm1, $imm2 and $zero in the various instructions. For example the beq instruction in the example jumps in case zero is equal to zero. This condition is always true and therefore it is a method to implement unconditional jump.

In addition to the instructions of the code, the assembler supports a pseudo-instruction that allows to set the contents of a 32-bit data word directly in the initial data memory image:

.word address data

Where address is the address of the word and data its value. Each one of the fields can be in decimal, or hexadecimal when preceded with 0x, for example:

```
.word 256 1                 # set MEM[256] = 1
.word 0x100 0x1234ABCD      # MEM[0x100] = MEM[256] = 0x1234ABCD
```

Additional assumptions

1. It can be assumed that the maximum line length in the input file is 500.
2. It can be assumed that the maximum label length is 50.
3. The label format starts with a letter, and then all the letters and numbers are allowed.
4. Whitespaces such as space or tab or empty lines should be ignored.
5. Hexadecimal numbers in the input file should be supported both in lower case and upper case.
6. You should follow questions, answers and updates in the course forum in moodle.

Submission guidelines

1. You should submit an external documentation file for the project, with the name project1_id1_id2_id3.pdf, where id1, id2, and id3 are your ID numbers.
2. In addition to the external documentation, there should be internal documentation: comments inside the source code that explain its operation.
3. The project will be written in the C programming language in the Microsoft visual studio environment under the Windows operating system.
4. The assembler and the simulator are separate projects in visual studio, each submitted in a separate subdirectory and contains its own solution file (with extension .sln) such that it can be recompiled and run by double clicking the solution file. The two projects should be called sim and asm. Make sure to target 64-bit debug executable.
5. Assembly test programs. Your project will be tested, among other tests, with test programs that you will not receive in advance, as well as with 4 test programs that you'll write in assembly. Make sure to document the test programs both in the source code, as well as in the external pdf documentation.
6. The test programs should be submitted in four subdirectories with the names:

   mulmat, binom, circle, disktest

   Each subdirectory will contain the input files of the test, as well as the output files generated by running the test through the assembler and simulator. For example the mulmat subdirectory will contain the following files:

   mulmat.asm, imemin.txt, dmemin.txt, diskin.txt, irq2in.txt, dmemout.txt, regout.txt, trace.txt, hwregtrace.txt, cycles.txt, leds.txt, display7seg.txt, diskout.txt, monitor.txt, monitor.yuv

7. It is important to make sure that both the assembler and the simulator run from within a windows cmd window by typing the command line and not only from within visual studio. It's also important to make sure that you use configurable command line parameters and not fixed names in the run command since we'll check your code using automated scripts running from batch files in a cmd window with file names that can be different.

Assembly Test Programs

You should submit the following assembly test programs:

1. A program **mulmat.asm** that performs a multiplication of two 4x4 matrices (each entry in the result matrix is a scalar multiplication of the corresponding row and column of the two matrices). The values of the first matrix are given in addresses 0x100 to 0x10F, the second matrix in 0x110 to 0x11F, and the result matrix should be written to 0x120 to 0x12F. You can assume that there is no overflow in the computation.
   Each matrix is ordered in memory in an increasing row order, and each row from left to right. For example in the first matrix, a_{11} will be in address 0x100, a_{12} in address 0x101 and a_{21} in 0x104.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

2. A program **binom.asm** that calculates the Newton binom coefficient recursively according to the following algorithm. At the start of the run n is given at address 0x100, k is given at address 0x101, and the result should be written to address 0x102. It can be assumed that n is small enough such that overflow doesn't happen.

```
int binom(n, k)
{
        if (k == 0 || n == k)
                return 1;
        return binom(n-1, k-1) + binom(n-1, k)
}
```

3. A program **circle.asm** that draws a circle on the screen. The circle should be placed at the center of the screen, in the white color, and should be a full circle (all the pixels on the edge and inside the circle should be white). The radius of the circle is given at address 0x100. It can be assumed that the circle fits in the screen.

4. A program **disktest.asm**, that moves the contents of the first 8 sectors in the disk drive (sectors 0 to 7) one sector forward such that at the end of the run, sector 1 will contain the contents of the original sector 0 and so on till sector 8 that will contain the contents of the original sector 7.