ELEC2204 Computer Engineering

Coursework: Computer Simulation

Romain Trost

rlt1n18

Design

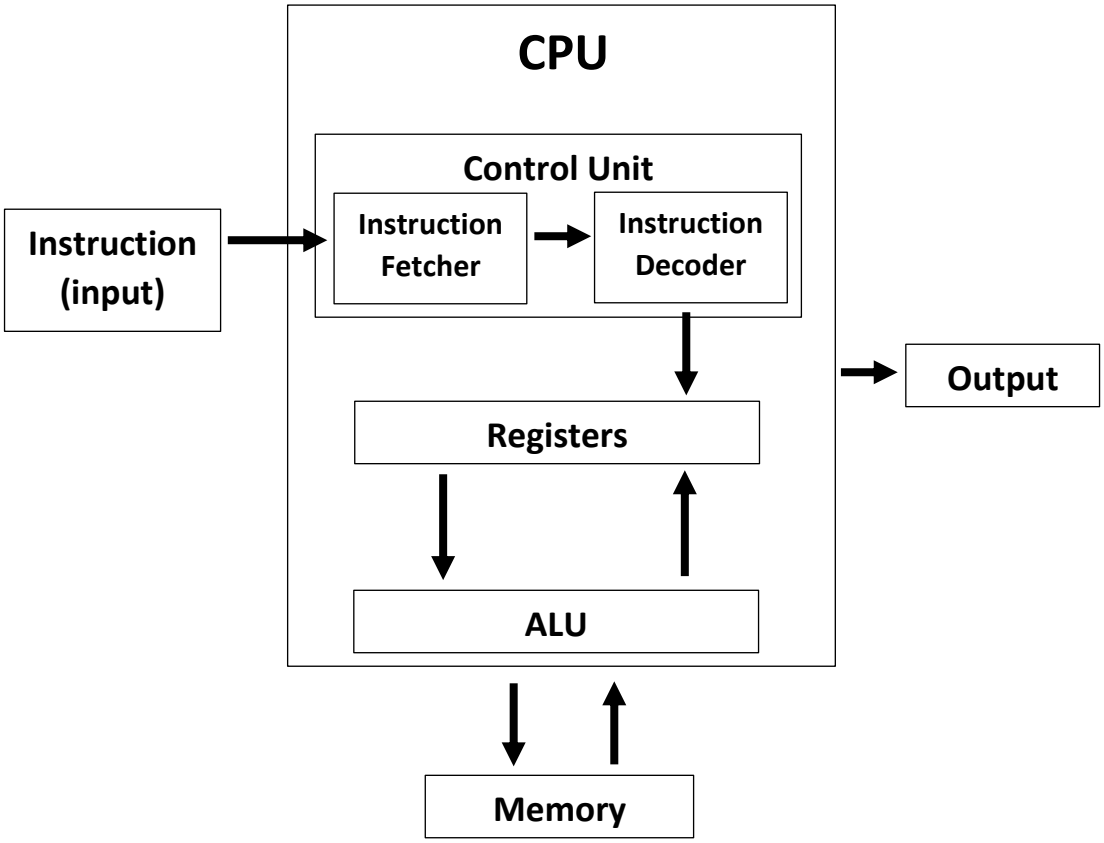

Figure 1: Diagram of the processor

Diagram Overview

The diagram above represents how the processor functions. A 32 bit instruction is sent to the CPU, which holds the control unit, registers and an arithmetic logic unit. The first role of the control unit is to fetch the instruction. It does by using the Program Counter which will be explained in more detail later on. Its second role is to decode the instruction which will allow the CPU to know how to process said instruction.

Also part of the CPU are the reigsters. Registers store 32 bit long data that can be accessed at very fast speeds. The manner in which these are accessed depends on the the instrution and how it's decoded.

The data accessed in these registers will then be sent to the ALU which will perform specific arithmetic operations on them. These operations also depend on the instruction and how it's decoded.

Data that's stored in the registers has the advantage of being easily accessed and at high speeds. However, the CPU also has access to memory in which it can store data. This data can either be written to memory from the CPU, or vice-versa.

After the CPU finishes processing the instruction, it sends out the instruction's intended output.

Romain Trost
rlt1n18@soton.ac.uk
ID: 29547458

Registers

The processor is composed of 36 registers each containing 32 bits of data. They are laid out as such:

- Register 0 is the "Zero Register": It always holds the value of 0.
- Registers 1 through 31 are the "Data Registers": Any 32 bit value can be written to them or read from them.
- Register 32 is the "Program Counter Register": It stores the value of the PC.
- Registers 33 through 35 are not utilised by this processor, but have been implemented out of common practice, allowing for potential future improvement of the processor.

Memory layout

A very simple memory layout has been implemented for this processor: it contains one address to store data.

Instructions

As for the instructions, the processor has the ability to interpret R-Type, I-Type and J-Type instructions. They each possess the format below.

Bits:	[31 - 26] *(6 bits)*	[25 - 21] *(5 bits)*	[20 - 16] *(5 bits)*	[15 - 11] *(5 bits)*	[10 - 6] *(5 bits)*	[5 - 0] *(6 bits)*
R-Type:	op	reg1	reg2	reg3	shamt	funct
I-Type:				offset		
J-Type:		jsec				

Figure 2: Instruction types

The first 6 bits are called the opcode. The opcode *op is* essential for all instructions as it is the opcode that will determine how the instruction should be divided.

For R-Type instructions, the instruction is divided into 5 more parts. The 3 sets of 5 bits: *reg1*, *reg2* and *reg3*, are all mainly used to point at the addresses of specific registers. They can also be used as a means of storing values in specific registers, but this is explained later on. As this processor has a relatively simple design and low instruction set, the use of *op* is sufficient to perform all instructions, allowing to ignore the use of *shamt* and *funct*.

For I-Types instructions, the instruction contains 3 extra parts: *reg1*, *reg2* and *offset*. Both sets of 5 bits *reg1* and *reg2* share the same purpose as in R-Type instructions. The 16 bit long component *offset* is used to store the value of the Program Counter. In other words, the value *offset* is set to, represents the address value of an instruction.

The J-Type format is used for jump instructions, however, this processor makes use of I-Type instructions (branching) to achieve instruction jumping. Therefore, this processor does not make use of any J-Type instructions, however it is able to recognise and divide them correctly.

Functionnality

The processor uses 2 arrays, *Mem[]* which holds all the instructions, and *Reg[]* which contains all the data stored in the registers. Furthermore, it uses 8 control signals that are explained in detail below. As for the data, *data1* and *data2* represent the data being manipulated by the processor. Additionally, *ALUresult* and *Zero* hold the data that the ALU outputs and *memData* holds the data that's stored in memory. Furthermore, all access to registers and memory is logged by using *regAccessed* and *memAccessed* as a means of tracking writes and reads.

The processor uses a straightforward fetch/decode/execute/memory access/write-back cycle. Each part of the cycle has been implemented through different functions.

instructionFetch() fetches the instruction. It uses the value of the Program Counter (*PC*) to fetch a specific instruction inside the memory. The value of *PC* is always 4 times that of the memory *Index*.

instructionDivide() divides the instruction. It reads the first 6 bits (the opcode) of the instruction and stores its value in *op*. Depending on its decimal representation, the instruction will be divided into an R-Type, I-Type or J-Type format.

instructionDecode() decodes the instruction. For every opcode, there are specific tasks that the processor is meant to perform, each having their own configuration of control signals. These signals will determine which datapaths the CPU should make use of, therefore controlling what task the CPU should execute. These are shown in the table below.

Processor's tasks	Control signals
STORE: Stores data into a register.	*RegDst* : Decides whether 2 or 3 registers are needed.
ADD: Adds 2 values together and stores the output.	*Jump* : Controls instruction jumping.
SUB: Subtracts a value from another value and stores the output.	*Branch* : Controls branching.
SLT: Checks if a value is smaller than another value, stores HIGH if true and LOW if not.	*MemRead* : Controls how data is set and how it's read from the registers.
MOD: Performs the modulo operation between 2 values and stores the result.	*MemtoReg* : Decides if data must be written to a register from memory.
JUMP: Jumps to a new instruction.	*ALUOp* : Decides what operation the ALU should perform.
BEQ: If 2 values are equal, the processor jumps to a specified instruction.	*MemWrite* : Decides if data must be written to memory.
BNE: If 2 values aren't equal, the processor jumps to a specified instruction.	*RegWrite* : Decides if data must be written to a register.
REGOUT: Prints out the data stored in the registers, the memory and the output.	
SAVE: Writes data to memory.	
LOAD: Reads data from memory.	

Figure 3: List of the processor's tasks and control signals

read() reads and sets data. If *MemRead = 1*, *data1* is set to the value stored in the register at address *reg1*, and *data2* is equal to the value stored in the register at address *reg2*. If *MemRead = 2*, *data1* is set equal to *reg1* and *data2* is set equal to *reg2*.

ALU() performs specific arithmetic operations depending on the control signal *ALUOp*. It takes in *data1* and *data2* as inputs, and outputs *ALUresult* and *Zero*. It makes use of operations such as adding, subtracting, comparing if *data1* and *data2* are equal or not, comparing if *data1* is smaller than *data2* and also performs the modulo operation. It has an extra operation called when the processor's current

task is *REGOUT,* whose purpose is simply to print out the output to the console along with the data stored in all registers and memory.

memoryAccess() simply writes the value stored in the register at address *reg1* and writes it to memory.

writeBack() writes data into the register at address *reg3*. This data comes either directly from the output of the ALU, or from memory.

PCUpdate() updates the Program Counter. It is always incremented by 4 in order to access the next instruction, unless the current instruction is a branch instruction. In this case, the PC will be set to the 16 bit binary representation of *offset*.

These functions laid out in this order represent each chronological step of the full instruction processing cycle. The processor is set up in a way that forces each instruction to go through each step of the cycle.

To better illustrate the functionality of the processor (and better understand the C code), some examples of instructions being processed are displayed below.

1) STORE 42 in Register 3 : 000001 00001 01010 00011 00000 000000
   (op = 1 | reg1 = 42 | reg2 | reg3 = 3 | shamt | funct)

Firstly, *op = 1* meaning that the instruction is divided into the R-Type format. With this opcode value, *MemRead = 2* which allows *read()* to set *data1 = 00001* (*reg1*) and *data2 = 01010* (*reg2*). *ALUOp = 1* so *ALUresult = (data1 << 5) + data2* in *ALU()*. This shifts *data1* by 5 bits to the left, then adds *data2* to it, essentially combining both sets of 5 bits together to form a set of 10 bits. This 10 bit value has a decimal value of 42 and is stored in *ALUresult*. *RegWrite = 1*, *MemtoReg = 0* and *RegDst = 1* which writes the value of *ALUresult* into the register whose address is equal to *reg3*. So in this case, 42 is stored in Register 3 (Reg[3]).

2) ADD Register 1 + Register 2 = Register 4: 000010 00001 00010 00100 00000 000000
   (op = 2 | reg1 = 1 | reg2 = 2 | reg3 = 4 | shamt | funct)

Here, *op = 2* which also divides the instruction into an R-Type format. Since *MemRead = 1*, *data1* will be set to the value stored in the register at address *reg1*. In this case, that value is 1 (Register 1). Similarly, *data2* will be set to the value stored in the register at address *reg* (Register 2). *ALUOp = 2* causing *ALUresult = data1 + data2*. Again, *RegWrite = 1*, *MemtoReg = 0* and *RegDst = 1* meaning that *ALUresult* will be stored in the register at address *reg3*, which is 4 here (Register 4).

3) BEQ Register 5 = Register 6 -> PC = 8: 000111 00101 00110 0000000000001000
   (op = 7 | reg1 = 5 | reg2 = 6 | offset = 8)

For this instruction, *op = 7* which makes it an I-Type instruction. *MemRead = 1* so *data1* is equal to the value in Register 5 and *data2* is equal to the value in Register 6. *ALUop = 6* meaning that if *data1* and *data2* are equal, *Zero* is set to 1 and if not, *Zero* is set to 0. *Branch = 1*, so if *Zero = 1*, PC becomes equal to the *offset*, 8. If Zero = 0, PC is incremented by 4 as usual. To summarise, this instruction forces the processor to execute the instruction stored at PC = 8 next, if the data in Register 5 is equal to the data in Register 6. If they're different, the processor continues as normal.

Basic testing

Please refer to Appendix A for all information regarding Basic Testing. In order to test the full functionality of the processor, a test program was input to the processor (see Figure A1). This set of instructions makes use of all of the processor's tasks, allowing to verify and ensure a full and optimal functionality. As the figure shows, coming up with this set of instructions first required writing assembly language that would execute each of the processor's tasks in a logical and efficient way. Every task had to be tested in every capacity, for example, **BEQ** can store 2 potential values, so it had to be implemented twice in order to trigger both outputs. This assembly language was then translated into machine code, creating the set of instructions.

Some of the instructions are dedicated to simply printing out the data stored in the processor as stated previously (***REGOUT***). These were implemented regularly in order to continually be aware of how the processor was affected by the instructions, which allowed for better and more efficient interpretation of the processor's performance. The explanation and interpretation of all the test instructions are explained below.

Explanation

R-Type instructions:

STORE: Stores the 10 bit value that *reg1* combined with *reg2* contains and stores it at register address *reg3*.

This means that **3 is stored in Register 1** at PC = 0, **9 is stored in Register 2** at PC = 4 and **10 is stored in Register 4** at PC = 16.

ADD: Adds the value stored in register address *reg1* with the value stored in register address *reg2*, storing the output in register address *reg3*.

At PC = 8, Register 1 (= 3) is being added to Register 2 (= 9) and stored in Register 3, meaning that **12 is stored in R3**.

***REGOUT*:** Prints the data contained in all registers and memory. It prints with emphasis the value in register address *reg1* as "OUTPUT". This task was often used during the testing phase, so that all registers could be viewed after every single instruction, allowing for better understanding of how the CPU was processing tasks.

For example, when PC = 12, the data in all registers and memory is printed to the console and the value stored in Register 3 is printed out as the emphasised "OUTPUT".

SUB: Subtracts the value stored in register address *reg2* from the value stored in register address *reg1*, storing the output in register address *reg3*.

So at PC = 20, Register 4 (= 10) is being subtracted from Register 3 (= 12), meaning that **2 will be stored in Register 5**.

SLT: Compares the value of *data1* stored in register address *reg1* with the value of *data2* stored in register address *reg2*. If *data1* is smaller than *data2*, a 1 is stored in register address *reg3*. If *data1* is equal or bigger than *data2*, a 0 is stored in register address *reg3*.

At PC = 28, Register 5 (= 2) is being compared to Register 4 (= 10). 2 is indeed smaller than 10, allowing a value of **1 to be stored in Register 6**.

At PC = 36, the opposite scenario is triggered. Since 10 is larger than 2, a value of **0 will be stored in Register 7**.

MOD: Performs the modulo operation between the value stored in register address *reg1* and the value stored in register address *reg2*, storing the output in register address *reg3*.

At PC = 44, the modulo operation is applied to Register 4 (= 10) with Register 1 (= 3). The remainder of dividing 10 by 3 is equal to **1, which is stored in Register 8**.

SAVE: Writes the value stored in register address *reg1* to memory.

At PC = 52, the value **10 is written to memory** from Register 4.

LOAD: Writes the value stored in memory to register address *reg3*.

At PC = 56, the value **10 is written to Register 9** from memory.

I-Type instructions:

BEQ: Compares the value stored in register address *reg1* (*data1*) to the value stored in register address *reg2* (*data2*). If *data1* is equal to *data2*, then the PC is set to the 16 bit long value *offset*. If they are different, the PC is incremented by 4 as usual.

At PC = 64, Register 6 (= 1) is compared to Register 7 (= 0). These are different, so the next instruction to execute will be the one at PC = 68.

At PC = 72, Register 6 (= 1) is compared to Register 8 (= 1). These are equal, meaning that the next instruction to be executed by the processor will be the one at PC = 88.

BNE: Acts similarly to BEQ, however, this checks if *data1* is different to *data2*. If they're different, the PC is set to *offset*. If they're equal, the PC is incremented by 4.

At PC = 92, Register 6 (= 1) is being compared to Register 8 (= 1). These are equal, so the next instruction to be executed will be the one stored at PC = 96.

At PC = 100, Register 6 (= 1) is being compared to Register 7 (= 0). These are different, forcing the processor to execute the instruction stored at PC = 116 next.

Interpretation

The data stored in memory and every register when PC = 60 (when all R-Type instructions have been processed) is shown in figure A2. The data displayed lines up perfectly with what the processor was expected to output. This confirms that the R-Type instructions are processed successfully and as intended by the processor.

Figure A3 illustrates the use of the I-Type instruction at PC = 72. This is the BEQ instruction that tells the PC to jump to 88 if Register 6 is equal to Register 8. As shown, Register 6 and Register 8 both hold the value of 1, which in turn forces the processor to execute the instruction at PC = 88 next. As stated previously, PC is stored in Register 32. Figure A3 clearly illustrates a jump from PC = 68 (last REGOUT instruction) to PC = 88, confirming the successful processing of this I-Type instruction.

Register and memory access logging

To help with the testing process, register and memory accessing was logged. This means that whenever data was written or read from either a register or memory, it was recorded by the processor. The final values containing the total amount of register and memory accesses during the entire program are then printed out, as shown in figure A4.

Showcase testing

Showcase programs

Please refer to Appendix B for all information regarding Showcase Testing.

For the first showcase program, the processor calculates and prints out the squares of all the integers between 0 and 99. The set of instructions displayed in Figure B1 was used to execute this program. These instructions make use of 6 registers: Registers 1, 2, 3, 4, 5 and 6.

Registers 4, 5 and 6 store the values 1, 2 and 100 respectively. These values do not change during the whole program, they are merely used to manipulate the values in the other 3 registers. Registers 1, 2 and 3 start off by holding the values 1, 1 and 0 respectively. There are 3 additions that occur. Firstly, Register 3 is incremented by the value in Register 2. Secondly, Register 1 is incremented by the constant value of 1 that's in Register 4. Finally, Register 2 is incremented by the constant value 2 that's in Register 5. The value in Register 3, which holds the squared value of the integer stored in Register 1, is then printed out as the output. The branch instruction at the end ensures that this process keeps repeating itself until Register 1 is equal to the constant value 100 that's in Register 6. See Figure B3 for the output.

For the second showcase program, the processor calculates and prints out all the prime numbers between 1 and 1000. The instruction set, displayed in Figure B2, was used to achieve this. These instructions were based off of the C code algorithm shown in Figure B6, which was inspired by the flow chart in Figure B5.

It makes use 7 processor tasks: STORE, ADD, BEQ, SLT, MOD, BNE, and REGOUT. The STORE instructions are used to initialise the variables. ADD is implemented to increment values by 1 (num at PC = 24 and i at PC = 32), and to set last equal to num (PC = 72). SLT is used to check when the value of i gets larger than the value of num. MOD performs the modulo operation between num and i. BEQ and BNE are used to deal with the if statements (if a condition is met, sends the program to new line of code or PC). REGOUT simply prints out the prime numbers as outputs. The output of the program is shown in Figure B4.

Additionnal program

A third program was also implemented which makes use of the SUB task. The purpose of this program is to print all numbers that are multiples of 3 between 0 and 300 in a descending order. Its set of instructions and its output are displayed in Figure C1 and Figure C2 respectively (Appendix C). It works by continuously subtracting 3 from 303 until 0 is reached.

Conclusions

To conclude, a fully functioning simulator that represents the operation of a very simple computer (processor) has been designed, tested and showcased. It has a Von Neumann architecture and follows the commonly used fetch/decode/execute/memory access/write-back cycle.

Any program, or set of 32 bit instructions, can be input to the simulator, where it will be processed just like in an actual computer. The only difference, is that this simulator isn't a general purpose CPU. It has been designed to execute a specific and limited set of instructions.

It has been tested thoroughly in order to optimise its functionality. To prove the successful design of the processor, 3 showcase programs were input to the processor and were executed successfully.

To further improve this simulator, its efficiency could be improved by evaluating the memory and register accessing. Furthermore, it could be enhanced to be able to process more instructions, making it more of a general purpose system. For example, it is already able to interpret J-Type instructions, but doesn't yet have a way to decode and therefore process such instructions.

Overall, the processor functions properly and as intended, with a respectable efficiency.

Romain Trost
rlt1n18@soton.ac.uk
ID: 29547458

Appendices

Appendix A: Basic Testing

TESTING	PC	INSTRUCTIONS	Bits:	[31 - 26] (6 bits)	[25 - 21] (5 bits)	[20 - 16] (5 bits)	[15 - 11] (5 bits)	[10 - 6] (5 bits)	[5 - 0] (6 bits)	Current register values:
			R-Type:	op	reg1	reg2	reg3	shamt	funct	/
			I-Type:				offset			/
			J-Type:		jsec					/
Mem[0]	0	STORE 3 in R1	R	000001	00000	00011	00001	00000	000000	R1 = 3
Mem[1]	4	STORE 9 in R2	R	000001	00000	01001	00010	00000	000000	R2 = 9
Mem[2]	8	ADD R1 + R2 = R3 // 3 + 9 = 12	R	000010	00001	00010	00011	00000	000000	R3 = 12
Mem[3]	12	REGOUT R3	R	001001	00011	00000	00000	00000	000000	
Mem[4]	16	STORE 10 in R4	R	000001	00000	01010	00100	00000	000000	R4 = 10
Mem[5]	20	SUB R3 - R4 = R5 // 12 - 10 = 2	R	000011	00011	00100	00101	00000	000000	R5 = 2
Mem[6]	24	REGOUT R5	R	001001	00101	00000	00000	00000	000000	
Mem[7]	28	SLT R5 < R4 = R6 // 2 < 10 = 1	R	000100	00101	00100	00110	00000	000000	R6 = 1
Mem[8]	32	REGOUT R6	R	001001	00110	00000	00000	00000	000000	
Mem[9]	36	SLT R4 < R5 = R7 // 10 < 2 = 0	R	000100	00100	00101	00111	00000	000000	R7 = 0
Mem[10]	40	REGOUT R7	R	001001	00111	00000	00000	00000	000000	
Mem[11]	44	MOD R4 % R1 = R8 // 10 % 3 = 1	R	000101	00100	00001	01000	00000	000000	R8 = 1
Mem[12]	48	REGOUT R8	R	001001	01000	00000	00000	00000	000000	
Mem[13]	52	SAVE R4 to MEMORY // 10 -> 10	R	001010	00100	00000	00000	00000	000000	Memory = 10
Mem[14]	56	LOAD MEMORY to R9 // 10 -> 10	R	001011	00000	00000	01001	00000	000000	R9 = 10
Mem[15]	60	REGOUT R9	R	001001	01001	00000	00000	00000	000000	
Mem[16]	64	BEQ R6 == R7 -> PC = 88	I	000111	00110	00111	00000	00001	011000	
Mem[17]	68	REGOUT R0	R	001001	00000	00000	00000	00000	000000	Used to see PC
Mem[18]	72	BEQ R6 == R8 -> PC = 88	I	000111	00110	01000	00000	00001	011000	
Mem[19]	76									
Mem[20]	80									
Mem[21]	84									
Mem[22]	88	REGOUT R0	R	001001	00000	00000	00000	00000	000000	Used to see PC
Mem[23]	92	BNE R6 != R8 -> PC = 116	I	001000	00110	01000	00000	00001	110100	
Mem[24]	96	REGOUT R0	R	001001	00000	00000	00000	00000	000000	Used to see PC
Mem[25]	100	BNE R6 != R7 -> PC = 116	I	001000	00110	00111	00000	00001	110100	
Mem[26]	104									
Mem[27]	108									
Mem[28]	112									
Mem[29]	116	REGOUT R0	R	001001	00000	00000	00000	00000	000000	Used to see PC
Mem[30]	120	END	/	000345	00000	00000	00000	00000	000000	

Figure A1: Set of instructions used to fully test the processor

```
Memory: 10
Register 0: 0
Register 1: 3
Register 2: 9
Register 3: 12
Register 4: 10
Register 5: 2
Register 6: 1
Register 7: 0
Register 8: 1
Register 9: 10
Register 10: 0
Register 11: 0
Register 12: 0
Register 13: 0
Register 14: 0
Register 15: 0
Register 16: 0
Register 17: 0
Register 18: 0
Register 19: 0
Register 20: 0
Register 21: 0
Register 22: 0
Register 23: 0
Register 24: 0
Register 25: 0
Register 26: 0
Register 27: 0
Register 28: 0
Register 29: 0
Register 30: 0
Register 31: 0
Register 32: 60
Register 33: 0
Register 34: 0
Register 35: 0
====================> OUTPUT: 10
```

Figure A2: All the data stored in memory and in the registers after all of the R-Type instructions from A1 have been executed (PC = 60)

```
Register 31: 0
Register 32: 68
Register 33: 0
Register 34: 0
Register 35: 0
====================> OUTPUT: 0
-----------------------------------------
Memory: 10
Register 0: 0
Register 1: 3
Register 2: 9
Register 3: 12
Register 4: 10
Register 5: 2
Register 6: 1
Register 7: 0
Register 8: 1
Register 9: 10
Register 10: 0
Register 11: 0
Register 12: 0
Register 13: 0
Register 14: 0
Register 15: 0
Register 16: 0
Register 17: 0
Register 18: 0
Register 19: 0
Register 20: 0
Register 21: 0
Register 22: 0
Register 23: 0
Register 24: 0
Register 25: 0
Register 26: 0
Register 27: 0
Register 28: 0
Register 29: 0
Register 30: 0
Register 31: 0
Register 32: 88
Register 33: 0
```

Figure A3: Output of the I-Type branch instruction at PC = 72 in A1, working correctly by jumping to instruction at PC = 88

```
Register 32: 120
Register 33: 0
Register 34: 0
Register 35: 0
====================> OUTPUT: 0
Registers accessed 19 times
Memory accessed 2 times
-----------------------------------------
```

Figure A4: Memory and register logging of the program in A1

Romain Trost
rlt1n18@soton.ac.uk
ID: 29547458

Appendix B: Showcase Testing

SQUARES	PC	INSTRUCTIONS	Bits:	[31 - 26] (6 bits)	[25 - 21] (5 bits)	[20 - 16] (5 bits)	[15 - 11] (5 bits)	[10 - 6] (5 bits)	[5 - 0] (6 bits)
			R-Type:	op	reg1	reg2	reg3	shamt	funct
			I-Type:				offset		
			J-Type:		jsec				
Mem[0]	0	STORE 1 in R4	R	000001	00000	00001	00100	00000	000000
Mem[1]	4	STORE 2 in R5	R	000001	00000	00010	00101	00000	000000
Mem[2]	8	STORE 1 in R2	R	000001	00000	00001	00010	00000	000000
Mem[3]	12	STORE 100 in R6	R	000001	00011	00100	00110	00000	000000
Mem[4]	16	STORE 1 in R1	R	000001	00000	00001	00001	00000	000000
Mem[5]	20	ADD R2 + R3 = R3	R	000010	00010	00011	00011	00000	000000
Mem[6]	24	ADD R1 + R4 = R1	R	000010	00001	00100	00001	00000	000000
Mem[7]	28	ADD R2 + R5 = R2	R	000010	00010	00101	00010	00000	000000
Mem[8]	32	REGOUT R3	R	001001	00011	00000	00000	00000	000000
Mem[9]	36	BNE R1 != R6 -> PC = 20	I	001000	00001	00110	00000	00000	010100
Mem[10]	40	END	/	000000	00000	00000	00000	00000	000000

Figure B1: Set of instructions to calculate and print out the squares of all integers between 0 and 99

PRIMES	PC	INSTRUCTIONS	Bits:	[31 - 26] (6 bits)	[25 - 21] (5 bits)	[20 - 16] (5 bits)	[15 - 11] (5 bits)	[10 - 6] (5 bits)	[5 - 0] (6 bits)
			R-Type:	op	reg1	reg2	reg3	shamt	funct
			I-Type:				offset		
			J-Type:		jsec				
Mem[0]	0	STORE 1 in R1	R	000001	00000	00001	00001	00000	000000
Mem[1]	4	STORE 0 in R2 // n = 0	R	000001	00000	00000	00010	00000	000000
Mem[2]	8	STORE 0 in R3 // last = 0	R	000001	00000	00000	00011	00000	000000
Mem[3]	12	STORE 0 in R4 // num = 0	R	000001	00000	00000	00100	00000	000000
Mem[4]	16	STORE 1000 in R5 // m = 1000	R	000001	11111	01000	00101	00000	000000
Mem[5]	20	STORE 2 in R6 // i = 2	R	000001	00000	00010	00110	00000	000000
Mem[6]	24	ADD R4 + R1 = R4 // num++	R	000010	00100	00001	00100	00000	000000
Mem[7]	28	BEQ R4 == R5 -> PC = 84 // i = 1000	I	000111	00100	00101	00000	00001	010100
Mem[8]	32	ADD R6 + R1 = R6 // i++	R	000010	00110	00001	00110	00000	000000
Mem[9]	36	SLT R6 < R4 = R7 // i < num	R	000100	00110	00100	00111	00000	000000
Mem[10]	40	BEQ R7 == R8 -> PC = 60 // SLT = 0	I	000111	00111	01000	00000	00000	111100
Mem[11]	44	MOD R4 % R6 = R9 // num % i	R	000101	00100	00110	01001	00000	000000
Mem[12]	48	BNE R9 != R8 -> PC = 32 // (num % i) != 0	I	001000	01001	01000	00000	00000	100000
Mem[13]	52	STORE 1 in R2 // n = 1	R	000001	00000	00001	00010	00000	000000
Mem[14]	56	BEQ R1 == R1 -> PC = 32 // restarts if loop	I	000111	00001	00001	00000	00000	100000
Mem[15]	60	BNE R2 != R8 -> PC = 76 //	I	001000	00010	01000	00000	00001	001100
Mem[16]	64	BEQ R3 == R4 -> PC = 72 // skips if loop	I	000111	00011	00100	00000	00001	001000
Mem[17]	68	REGOUT R4 // prints number	R	001001	00100	00000	00000	00000	000000
Mem[18]	72	ADD R4 + R8 = R3 // last = num	R	000010	00100	01000	00011	00000	000000
Mem[19]	76	STORE 0 in R2 // n = 0	R	000001	00000	00000	00010	00000	000000
Mem[20]	80	BEQ R1 == R1 -> PC = 16 // restarts for loop	I	000111	00001	00001	00000	00000	010000
Mem[21]	84	END	/	000000	00000	00000	00000	00000	000000

Figure B2: Set of instructions to calculate and print out all the prime numbers between 0 and 1000

```
====================> OUTPUT: 9025
------------------------------------------------
Register 1: 97
Register 2: 193
Register 3: 9216
Register 4: 1
Register 5: 2
Register 6: 100
====================> OUTPUT: 9216
------------------------------------------------
Register 1: 98
Register 2: 195
Register 3: 9409
Register 4: 1
Register 5: 2
Register 6: 100
====================> OUTPUT: 9409
------------------------------------------------
Register 1: 99
Register 2: 197
Register 3: 9604
Register 4: 1
Register 5: 2
Register 6: 100
====================> OUTPUT: 9604
------------------------------------------------
Register 1: 100
Register 2: 199
Register 3: 9801
Register 4: 1
Register 5: 2
Register 6: 100
====================> OUTPUT: 9801
------------------------------------------------
```

Figure B3: Output of the program in B1

```
====================> OUTPUT: 881
------------------------------------------------
Register 1: 1
Register 2: 0
Register 3: 881
Register 4: 883
Register 5: 1000
Register 6: 883
Register 7: 0
Register 8: 0
Register 9: 1
====================> OUTPUT: 883
------------------------------------------------
Register 1: 1
Register 2: 0
Register 3: 883
Register 4: 887
Register 5: 1000
Register 6: 887
Register 7: 0
Register 8: 0
Register 9: 1
====================> OUTPUT: 887
------------------------------------------------
Register 1: 1
Register 2: 0
Register 3: 887
Register 4: 907
Register 5: 1000
Register 6: 907
Register 7: 0
Register 8: 0
Register 9: 1
====================> OUTPUT: 907
------------------------------------------------
```

Figure B4: Output of the program in B2

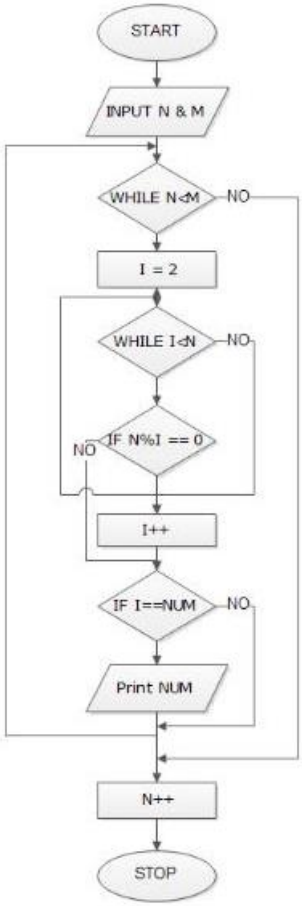

Figure B5: Flow chart to find all prime number between 2 given numbers

```c
int n = 0;
int last = 0;
int m = 1000;

for(int num = 0; num < m; num++){
    for (int i = 2; i<num; i++) {
        if ((num % i) == 0) {
            n = 1;
        }
    }

    if(n==0) {
        if (last!=num) {
            printf( _Format: "%d is a prime number\n", num);
        }
        last = num;
    }
    n = 0;
}
```

Figure B6: C code algorithm that finds all prime numbers between 0 and 1000

Appendix C: Additionnal program

DESCENDING MULTIPLES OF 3	PC	INSTRUCTIONS	Bits:	[31 - 26] *(6 bits)*	[25 - 21] *(5 bits)*	[20 - 16] *(5 bits)*	[15 - 11] *(5 bits)*	[10 - 6] *(5 bits)*	[5 - 0] *(6 bits)*
			R-Type:	op	reg1	reg2	reg3	shamt	funct
			I-Type:				offset		
			J-Type:		jsec				
Mem[0]	0	*STORE 303 in R1*	R	000001	01001	01111	00001	00000	000000
Mem[1]	4	*STORE 3 in R2*	R	000001	00000	00011	00010	00000	000000
Mem[2]	8	*SUB R1 - R2 = R1*	R	000011	00001	00010	00001	00000	000000
Mem[3]	12	*REGOUT R1*	R	001001	00001	00000	00000	00000	000000
Mem[4]	16	*BNE R1 != R4 -> PC = 8*	I	001000	00001	00100	00000	00000	001000
Mem[5]	20	*END*	/	000000	00000	00000	00000	00000	000000

Figure C1: Set of instructions to calculate and print out all numbers that are multiples of 3 between 0 and 300 in a descending order

```
====================> OUTPUT: 54
--------------------------------------------
Register 1: 51
Register 2: 3
Register 4: 0
====================> OUTPUT: 51
--------------------------------------------
Register 1: 48
Register 2: 3
Register 4: 0
====================> OUTPUT: 48
--------------------------------------------
Register 1: 45
Register 2: 3
Register 4: 0
====================> OUTPUT: 45
--------------------------------------------
Register 1: 42
Register 2: 3
Register 4: 0
====================> OUTPUT: 42
--------------------------------------------
Register 1: 39
Register 2: 3
Register 4: 0
====================> OUTPUT: 39
--------------------------------------------
Register 1: 36
Register 2: 3
Register 4: 0
====================> OUTPUT: 36
--------------------------------------------
Register 1: 33
Register 2: 3
Register 4: 0
====================> OUTPUT: 33
--------------------------------------------
```

Figure C2: Output of the program in C1

References

1) GitHub. 2020. *Trevinofernando/SIMULATOR-OF-A-MIPS-PROCESSOR-IN-C*. [online] Available at: https://github.com/trevinofernando/SIMULATOR-OF-A-MIPS-PROCESSOR-IN-C.

2) 2020. [online] Available at: https://www.youtube.com/watch?v=oETOwVBzu1s.

3) Koppella, G., 2020. *Algorithm And Flowchart To Find All Prime Numbers (Range) | Computer Science Simplified - A Website For IGNOU MCA & BCA Students For Solved Assignments, Notes, C Programming, Algorithms - Cssimplified.Com*. [online] Cssimplified.com. Available at: http://cssimplified.com/c-cpp-programming-data-structure/design-an-algorithm-and-draw-corresponding-flowchart-to-find-all-the-prime-numbers-between-two-given-numbers-m-and-n-where-m-n-0-10m-dec2005.