Abstract

This emulator was created to test the properties of a basic computer. The report outlines the design of the computer, how it was emulated in software and how it was tested. Two showcase programs were used to verify the functionality of the computer and test the relative performance. The results of the tests confirmed the functionality of the computer as well as highlighting the success of using detailed data logging as a method of debugging the emulator and the programs being executed.

Design

A 32-bit instruction set was used. Having all instructions the same length was an important feature of the design as it simplified how the instructions were decoded. Two formats were used for encoding information into the instructions, these were referred to as R-type and I-type. A 32-bit instruction was required for data storage and program flow control operations. 16-bits were required to encode all the possible memory locations. When the bits required for the other parts of the instruction were considered as well, the use of the 32-bit instruction seemed logical. The design used a total of 12 instructions meaning 4-bits would have been enough for the opcode. The register array had 16 elements, 4-bits would have covered all the register locations. This showed some of the instructions could have been condensed, but as this meant having a more complex decoding stage, there wasn't enough gain. By having the additional bits available, the design had more flexibility and meant that more functions could be incorporated later.

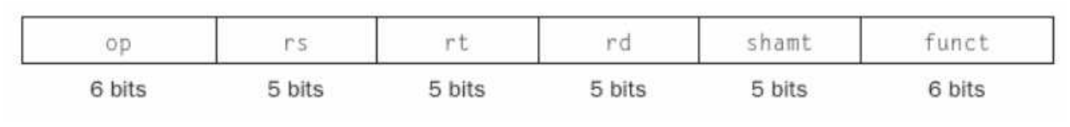


Figure 1: R-type Instruction format [1]

The R-type instruction format was used for data processing operations. The available operations were addition, subtraction, multiplication and division. The opcode represented the chosen arithmetic operation; 'rs' and 'rt' encoded the address of the source operands; 'rd' encoded the address where the result would be stored. 'Shamt' and 'funct' weren't used for these instructions, these values were always set to 0_{10}.

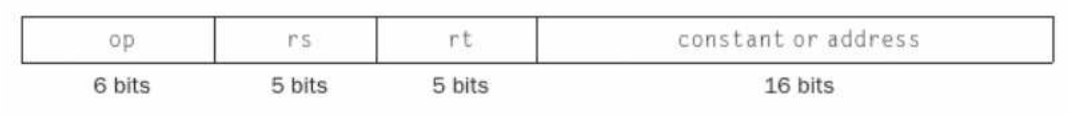


Figure 2: I-type Instruction format [1]

The I-type instruction format was used for multiple purposes. It represented data storage, program flow control and input/output functions. How the instruction was interpreted depended upon the opcode. Akin to the R-type, the opcode represented what function would be carried out. The bits of the opcode instructed the controller to configure the data path so that the encoded function was carried out. Table 1 shows how each I-type instruction was encoded.

Instruction	Op	Rs	Rt	Address
Data storage (LOAD/STORE)	Instruction for controller	Memory location to be read from/written to is Rs + Address	Register address to be read from /written to	Memory location to be read from/written to is Rs + Address
Branch (BNE, BEQ, BLT)	Instruction forcontroller	Register address of operand being compared	Register address of operand being compared	The address of the next instruction
JUMP	Instruction for controller	Not used, set to 0x0	Not used, set to 0x0	The address of the next instruction
I/O (WRTIE TO DISPLAY)	Instruction for controller	Not used, set to 0x0	Register address containing the data to be written to the display	Not used, set to 0x0
No operation (NOP)	Instruction for controller	Not used, set to 0x0	Not used, set to 0x0	Not used, set to 0x0

Table 1: I-type instructions

A simple architecture for the processor was used. A 15-bit program counter was used to point to the next instruction to be fetched from memory. It was 15-bit so that it could only point to the lower half of the memory (where the instructions were stored). This was loaded into the instruction register. From here, the instruction was decoded. The decoded data was sent along the 32-bit bus to execute the instruction. A 16 element, general purpose register file was used loading data from memory into the processor and writing back results from arithmetic operations. The Von Neumann architecture was specified, so a shared 64K memory was used for instructions and data. This was divided up so that the bottom half of the memory stored the program instructions (0-32767) and the top half stored the data (32768-65535). Initial data provided by the program started at address 32768 and increased. When data was written to memory, it started at address 65535 and decreased. Separate buses were used for the address and data to improve speed.

Implementation

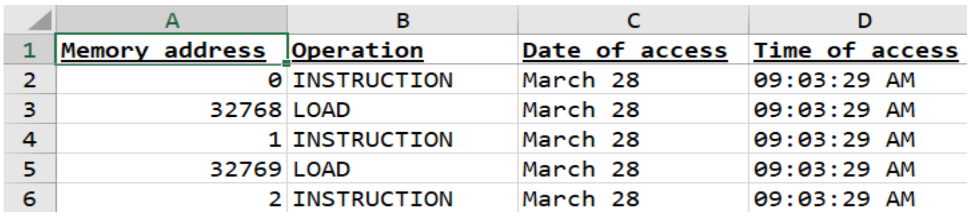
A 65536-element array (64K) was used for the memory subsystem. Each element was represented as a 32-bit signed integer. This meant positive and negative values could be stored in memory and because a 6-bit opcode was used, the most significant bit of all the instructions was always zero so the instructions were never misinterpreted as negative values. Structures were used for modelling the program counter and the buses for the decoded instruction. This meant the bit width for the program counter and the instruction could be specified. The decoded instruction was broken down a step further so that it had a total of 32 bits, but each instruction field had its bit width specified individually. This meant that each field of the instruction was transferred to the correct part of the processor for the execution stage. A single register (IR) was defined for holding the fetched instruction. This had the unsigned 32-bit data type as it wasn't used for holding data. A 16-element array (reg) was used to replicate the register file. This had a signed 32-bit data type as it was only used for storing data.

The fetch stage assigned an element from memory which was indexed by the program counter to the IR register. The program counter was incremented after the instruction was fetched. Three functions were used to implement the decode stage (I_TYPE, R_TYPE and decode). The first two signalled the decoding of their respective instruction formats. These would be called once the opcode had been evaluated. The third function decoded the instruction. It did this by taking the instruction, number of bits to be decoded and the maximum bit position to be decoded as parameters. The number of bits and maximum bit position identified which field of the instruction was being decoded (see figures 1 and 2). To separate the bits being decoded from the full instruction, a 32-bit mask was used. This had all its bits set to logical 0 except the ones in the same position as the bits being decoded from the instruction. These were set to logical 1. A logical '&' was used to set all the bits which were logical 1 in the mask and the instruction to logical 1 in the new variable (masked_instruct). These bits were shifted to the right by x places and assigned to its respective bus (x = maximum bit position – number of bits to be decoded). If-statements were used to determine what was being decoded so that the bits were shifted onto the correct bus.

To implement the execute and write back (where applicable) stages, a switch statement was used to evaluate the opcode. This was equivalent to the opcode controlling the data path. Whichever opcode was evaluated was what the emulator would branch to. Here the function of the corresponding opcode would execute and write back (if applicable). This represented the function of the ALU.

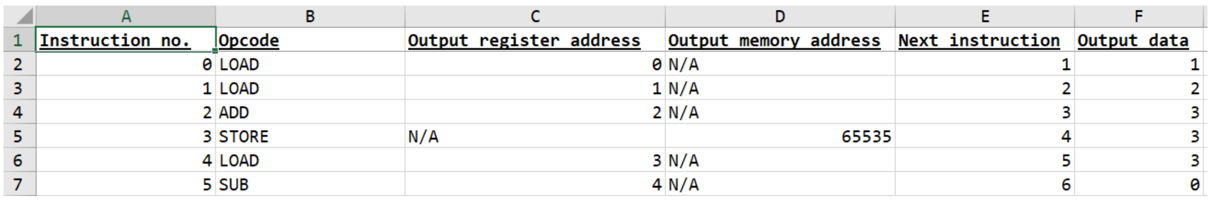
Testing

To ensure the emulator was thoroughly tested, a debug mode was built into the source file of the emulator. This was implemented using pre-processor directives. A constant 'DEBUG' was defined at the top of the file, if this was set to 1, debug mode was activated. When the emulator was in debug mode, it would create two spreadsheets. One spreadsheet would log the memory accesses and the other would log various signals within the emulator. Figures 3 and 4 show examples of what these files looked like. A software library called 'XLSX I/O' *[2]* was implemented to incorporate this method of data logging. As a lot of data was being logged, the execution time increased significantly; but because of the organised structure of the data log, it made debugging much easier.



	A	B	C	D
1	Memory address	Operation	Date of access	Time of access
2	0	INSTRUCTION	March 28	09:03:29 AM
3	32768	LOAD	March 28	09:03:29 AM
4	1	INSTRUCTION	March 28	09:03:29 AM
5	32769	LOAD	March 28	09:03:29 AM
6	2	INSTRUCTION	March 28	09:03:29 AM

Figure 3: Example of the memory access spreadsheet



	A	B	C	D	E	F
1	Instruction no.	Opcode	Output register address	Output memory address	Next instruction	Output data
2	0	LOAD	0	N/A	1	1
3	1	LOAD	1	N/A	2	2
4	2	ADD	2	N/A	3	3
5	3	STORE	N/A	65535	4	3
6	4	LOAD	3	N/A	5	3
7	5	SUB	4	N/A	6	0

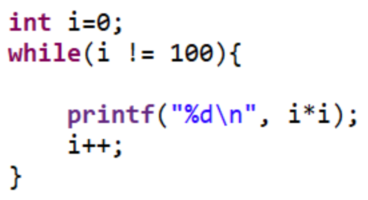
Figure 4: Example of the debug data spreadsheet

A separate source file was used for loading programs into the memory of the emulator. A function named 'test_program' loaded a set of test instructions and some data into the memory of the emulator. To begin with, the emulator needed to demonstrate it fetched the correct instruction from memory and incremented the program counter. The instruction number and next instruction columns of the debug spreadsheet verified this. The memory access instructions needed to demonstrate they interpreted the data correctly and that they loaded/stored to the correct register/memory addresses. The output data, register address and memory address columns provided the evidence for this. The data processing instructions needed to show they used the correct data addresses for the operands and carried out the correct operation. They also needed to show they computed the correct result for positive and negative values. The output data column was used to show the results of these operations and the register address column showed which register the result was written back to. The program flow instructions needed to demonstrate that the program branched to the correct instruction. The next instruction column was used to show this. Additionally, the branch instructions needed to display they only changed the sequence of the program if the condition was met. Two test instructions were written for this, one where the program was expected to branch and another where it wasn't. The output instruction demonstrated it worked as expected by printing the correct result to the console. The spreadsheet showed the data it was expected to output in the output data column. Finally, the no operation instruction was tested, this needed to show that it didn't do anything. This was verified because the next instruction was the following one in the sequence and there was no output data. If any unexpected results were displayed on the spreadsheet, it had to be caused by an issue with the emulator or the instruction being entered incorrectly. The simplest cause was that the instruction had been entered incorrectly, this was always checked first. If there was an error with the emulator, the spreadsheet explicitly showed what the error was which made finding the error in the code faster to locate.

Provided each instruction type had been tested thoroughly, any sequence of instructions would be executed correctly. If the program didn't execute as expected, the issue would lie in the program's algorithm. The debug spreadsheet was used for finding errors in the showcase programs and even making them more efficient. For example, in the second showcase program (calculating the primes), the program initially looked for every factor of the number being evaluated. For a number to be composite, there must be at least two smaller numbers which can be multiplied together to make this number. An extra bit of logic was added to the program which meant that it would stop evaluating a number once two smaller factors had been found.

Results

A simple algorithm was used for finding the squares of all the integers between 0 and 99. Figure 5 shows the algorithm written in C.



```
int i=0;
while(i != 100){

    printf("%d\n", i*i);
    i++;
}
```

Figure 5: Basic algorithm used for finding the squares of integers between 0 and 99

When this program ran, it executed 505 instructions. Figures 8 and 9 in Appendix A show extracts from the debug file for this program. The file shows that 5 instructions were executed between the previous square being displayed and the next square. Figures 6 and 7 show extracts of the memory accesses made by this program. After loading the initial data to the registers, the only reason memory was accessed was to fetch the instructions. This suggested the program had a fast execution since memory accesses are slow in comparison to register accesses. The memory access log showed that the program was executed in 1 second or less. This was when the program was in debug mode where additional CPU time was used for data logging.

The program which found the primes between 1 and 1000 executed 955,906 instructions. Figures 12 and 13 in Appendix B show extracts from the debug data log. It took 12 instructions to confirm 3 was a prime number. As the value being tested increased, so did the number of instructions required to check if a value was prime. Figures 10 and 11 show extracts of the memory accesses made throughout the program. Six values were loaded into the registers from memory at the start of the program, after that the only memory accesses were when instructions were fetched. The program took 24s to run in debug mode, this was due to the large amount of data being logged. When the program was executed without debug mode it took 1s or less.

Critical Evaluation

Only the functionality of the emulator and the efficiency of the algorithm were tested. The testing stage could have included measuring performance metrics such as execution time and MIPS so that relative performance of the emulator could have been calculated. That way, if further improvements are made to the emulator, the improvements can be quantified.

To improve the design of the emulator, the branch less than instruction could be replaced. These instructions force the computer to use a slower clock speed as they take a relatively long time to execute in comparison to other instructions. A set less than instruction could have been used instead. The less than operation could be executed using two instructions- set less than and branch equal or branch not equal. This would increase the number of instructions used, but it would allow the clock speed to be kept higher meaning the other instructions wouldn't be slowed down.

The emulator could also be improved by introducing pipelining. This would increase the throughput of the emulator by using more stages of the system in parallel. In order to implement this into the emulator, threads would have to be used and this would increase the complexity of the design as it would require making sure only one thread is using a stage of the pipeline at one time.

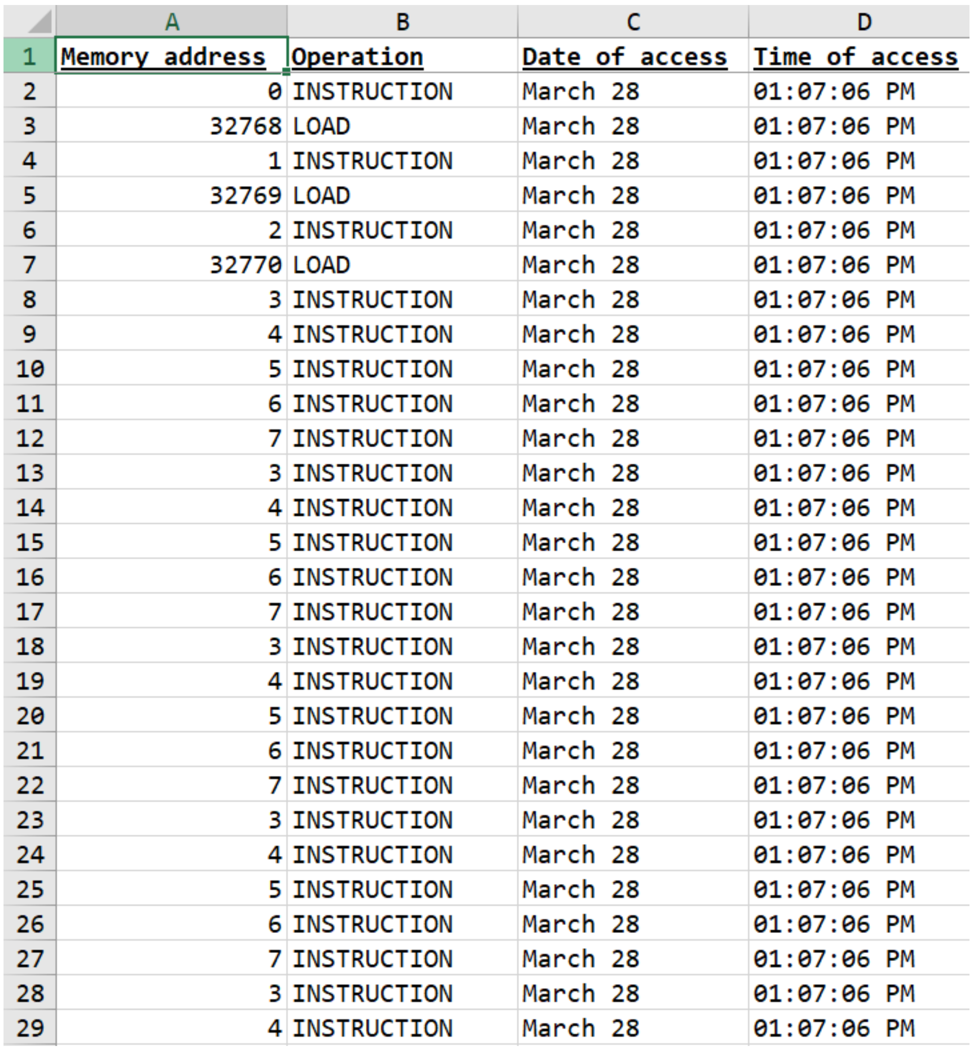
References

[1] Patterson, D. and Hennessy, J. (2008). Computer Organization and Design, Fourth Edition. N/A: Elsevier Science & Technology, pp 96-97

[2] XLSX I/O- C library for reading and writing values to .xlsx files (2016). Belgium: Brecht Sanders.

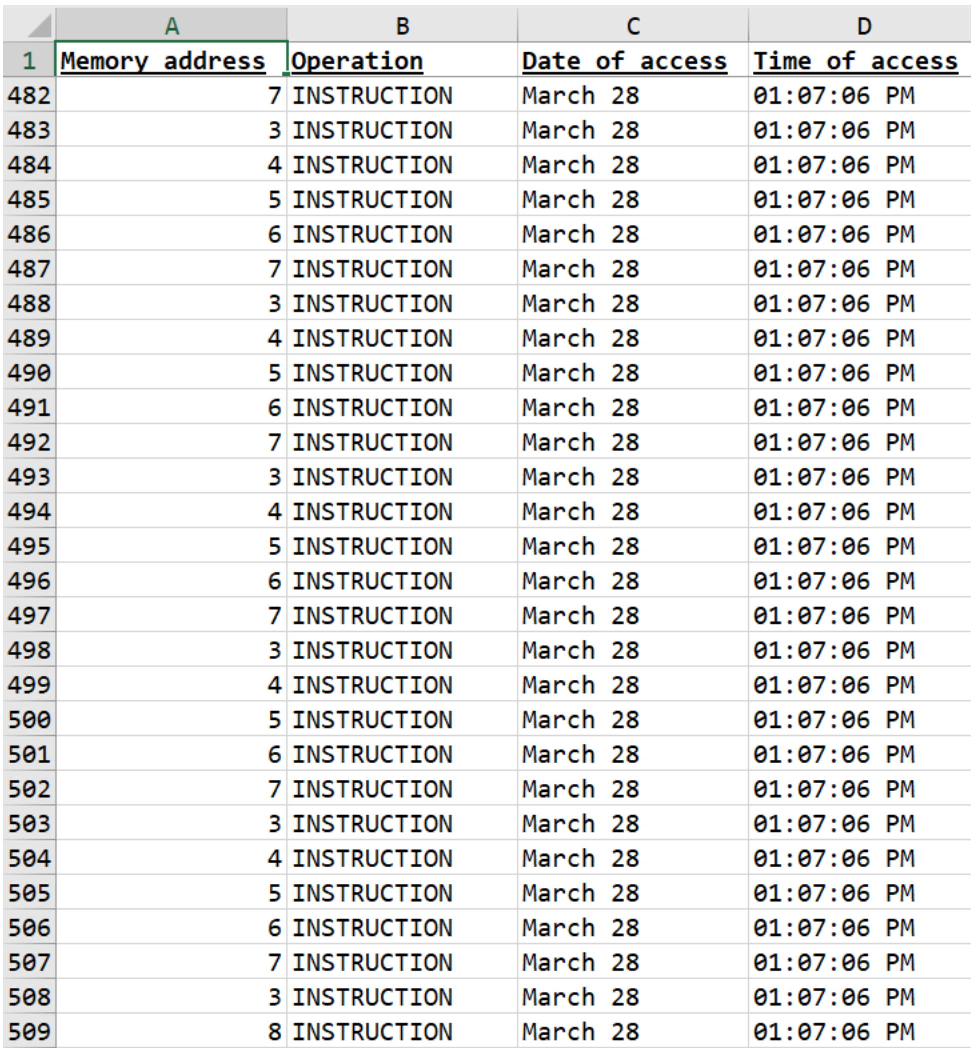
Appendix A- showcase program 1

A snapshot of the time history of memory accesses for the ‘squares of all integers between 0 and 99’ showcase program is shown below. Figure 6 shows the first 28 memory accesses, three of which are for loading data from memory to the registers, all the others are fetching instructions. Figure 7 shows the last 28 memory accesses, all of which are instruction fetches.



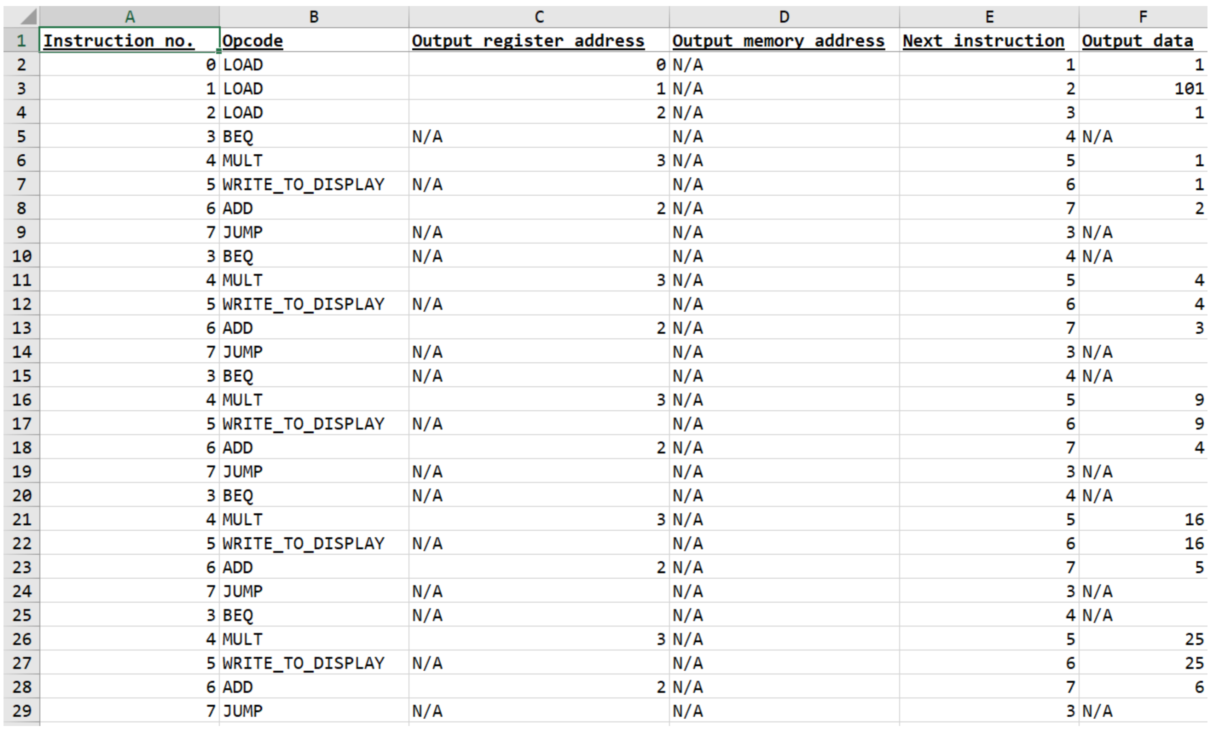
	A	B	C	D
1	Memory address	Operation	Date of access	Time of access
2	0	INSTRUCTION	March 28	01:07:06 PM
3	32768	LOAD	March 28	01:07:06 PM
4	1	INSTRUCTION	March 28	01:07:06 PM
5	32769	LOAD	March 28	01:07:06 PM
6	2	INSTRUCTION	March 28	01:07:06 PM
7	32770	LOAD	March 28	01:07:06 PM
8	3	INSTRUCTION	March 28	01:07:06 PM
9	4	INSTRUCTION	March 28	01:07:06 PM
10	5	INSTRUCTION	March 28	01:07:06 PM
11	6	INSTRUCTION	March 28	01:07:06 PM
12	7	INSTRUCTION	March 28	01:07:06 PM
13	3	INSTRUCTION	March 28	01:07:06 PM
14	4	INSTRUCTION	March 28	01:07:06 PM
15	5	INSTRUCTION	March 28	01:07:06 PM
16	6	INSTRUCTION	March 28	01:07:06 PM
17	7	INSTRUCTION	March 28	01:07:06 PM
18	3	INSTRUCTION	March 28	01:07:06 PM
19	4	INSTRUCTION	March 28	01:07:06 PM
20	5	INSTRUCTION	March 28	01:07:06 PM
21	6	INSTRUCTION	March 28	01:07:06 PM
22	7	INSTRUCTION	March 28	01:07:06 PM
23	3	INSTRUCTION	March 28	01:07:06 PM
24	4	INSTRUCTION	March 28	01:07:06 PM
25	5	INSTRUCTION	March 28	01:07:06 PM
26	6	INSTRUCTION	March 28	01:07:06 PM
27	7	INSTRUCTION	March 28	01:07:06 PM
28	3	INSTRUCTION	March 28	01:07:06 PM
29	4	INSTRUCTION	March 28	01:07:06 PM

Figure 6: The first 28 memory accesses



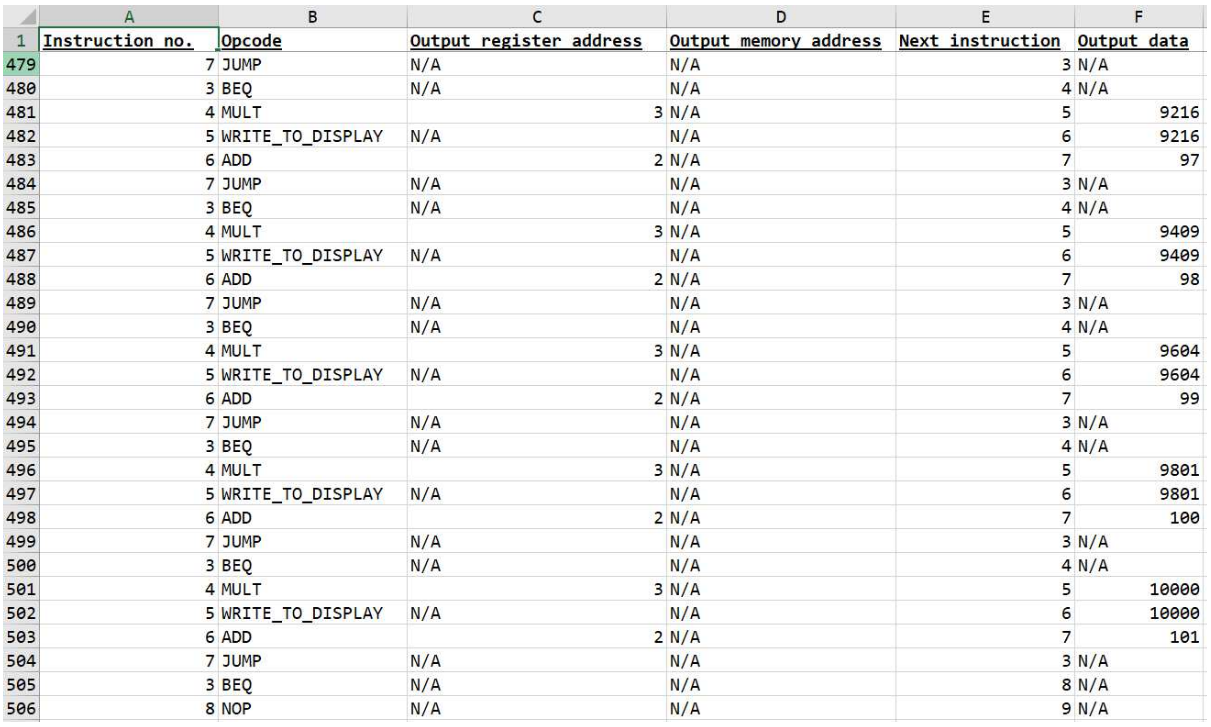
	A	B	C	D
1	Memory address	Operation	Date of access	Time of access
482	7	INSTRUCTION	March 28	01:07:06 PM
483	3	INSTRUCTION	March 28	01:07:06 PM
484	4	INSTRUCTION	March 28	01:07:06 PM
485	5	INSTRUCTION	March 28	01:07:06 PM
486	6	INSTRUCTION	March 28	01:07:06 PM
487	7	INSTRUCTION	March 28	01:07:06 PM
488	3	INSTRUCTION	March 28	01:07:06 PM
489	4	INSTRUCTION	March 28	01:07:06 PM
490	5	INSTRUCTION	March 28	01:07:06 PM
491	6	INSTRUCTION	March 28	01:07:06 PM
492	7	INSTRUCTION	March 28	01:07:06 PM
493	3	INSTRUCTION	March 28	01:07:06 PM
494	4	INSTRUCTION	March 28	01:07:06 PM
495	5	INSTRUCTION	March 28	01:07:06 PM
496	6	INSTRUCTION	March 28	01:07:06 PM
497	7	INSTRUCTION	March 28	01:07:06 PM
498	3	INSTRUCTION	March 28	01:07:06 PM
499	4	INSTRUCTION	March 28	01:07:06 PM
500	5	INSTRUCTION	March 28	01:07:06 PM
501	6	INSTRUCTION	March 28	01:07:06 PM
502	7	INSTRUCTION	March 28	01:07:06 PM
503	3	INSTRUCTION	March 28	01:07:06 PM
504	4	INSTRUCTION	March 28	01:07:06 PM
505	5	INSTRUCTION	March 28	01:07:06 PM
506	6	INSTRUCTION	March 28	01:07:06 PM
507	7	INSTRUCTION	March 28	01:07:06 PM
508	3	INSTRUCTION	March 28	01:07:06 PM
509	8	INSTRUCTION	March 28	01:07:06 PM

Figure 7: The last 28 memory accesses



	A	B	C	D	E	F
1	Instruction no.	Opcode	Output register address	Output memory address	Next instruction	Output data
2	0	LOAD	0	N/A	1	1
3	1	LOAD	1	N/A	2	101
4	2	LOAD	2	N/A	3	1
5	3	BEQ	N/A	N/A	4	N/A
6	4	MULT	3	N/A	5	1
7	5	WRITE_TO_DISPLAY	N/A	N/A	6	1
8	6	ADD	2	N/A	7	2
9	7	JUMP	N/A	N/A	3	N/A
10	3	BEQ	N/A	N/A	4	N/A
11	4	MULT	3	N/A	5	4
12	5	WRITE_TO_DISPLAY	N/A	N/A	6	4
13	6	ADD	2	N/A	7	3
14	7	JUMP	N/A	N/A	3	N/A
15	3	BEQ	N/A	N/A	4	N/A
16	4	MULT	3	N/A	5	9
17	5	WRITE_TO_DISPLAY	N/A	N/A	6	9
18	6	ADD	2	N/A	7	4
19	7	JUMP	N/A	N/A	3	N/A
20	3	BEQ	N/A	N/A	4	N/A
21	4	MULT	3	N/A	5	16
22	5	WRITE_TO_DISPLAY	N/A	N/A	6	16
23	6	ADD	2	N/A	7	5
24	7	JUMP	N/A	N/A	3	N/A
25	3	BEQ	N/A	N/A	4	N/A
26	4	MULT	3	N/A	5	25
27	5	WRITE_TO_DISPLAY	N/A	N/A	6	25
28	6	ADD	2	N/A	7	6
29	7	JUMP	N/A	N/A	3	N/A

Figure 8: Shows the debug file for the first 28 instructions

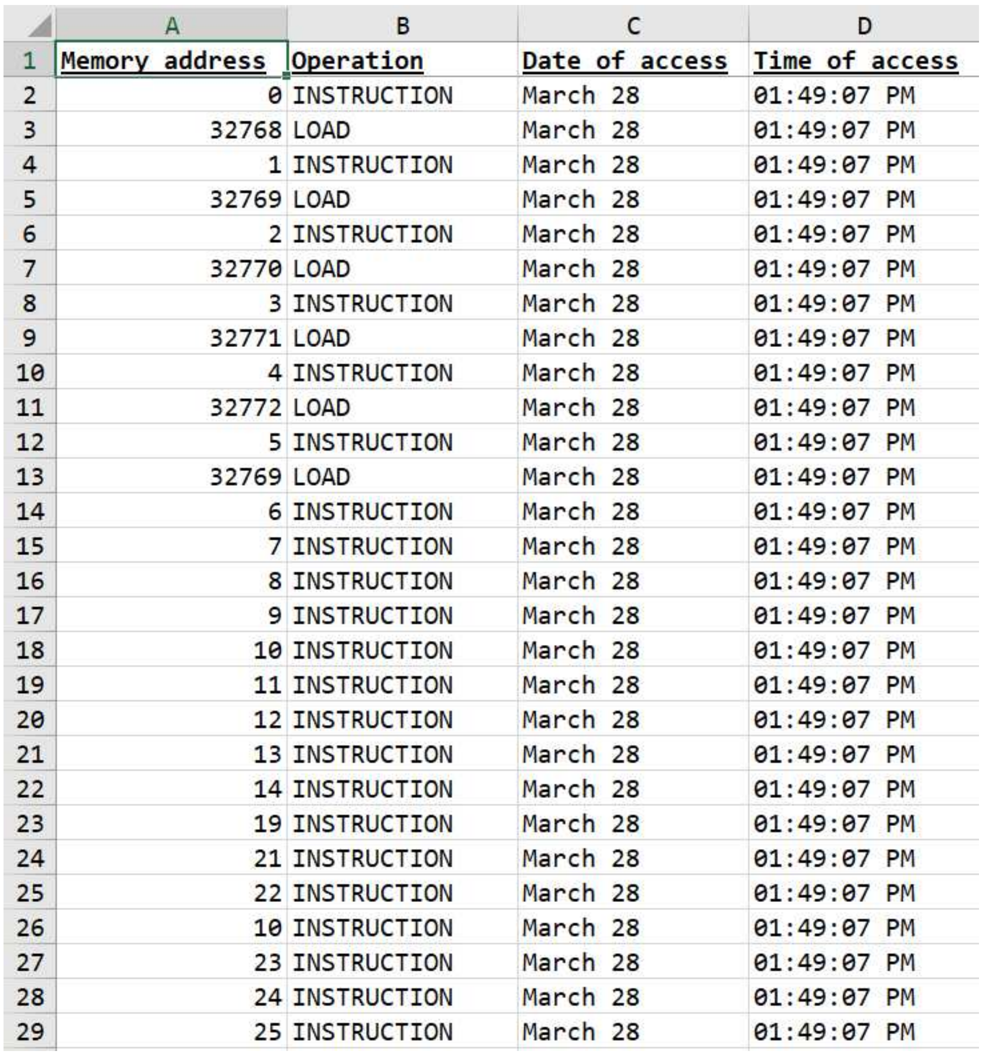


	A	B	C	D	E	F
1	Instruction no.	Opcode	Output register address	Output memory address	Next instruction	Output data
479	7	JUMP	N/A	N/A	3	N/A
480	3	BEQ	N/A	N/A	4	N/A
481	4	MULT	3	N/A	5	9216
482	5	WRITE_TO_DISPLAY	N/A	N/A	6	9216
483	6	ADD	2	N/A	7	97
484	7	JUMP	N/A	N/A	3	N/A
485	3	BEQ	N/A	N/A	4	N/A
486	4	MULT	3	N/A	5	9409
487	5	WRITE_TO_DISPLAY	N/A	N/A	6	9409
488	6	ADD	2	N/A	7	98
489	7	JUMP	N/A	N/A	3	N/A
490	3	BEQ	N/A	N/A	4	N/A
491	4	MULT	3	N/A	5	9604
492	5	WRITE_TO_DISPLAY	N/A	N/A	6	9604
493	6	ADD	2	N/A	7	99
494	7	JUMP	N/A	N/A	3	N/A
495	3	BEQ	N/A	N/A	4	N/A
496	4	MULT	3	N/A	5	9801
497	5	WRITE_TO_DISPLAY	N/A	N/A	6	9801
498	6	ADD	2	N/A	7	100
499	7	JUMP	N/A	N/A	3	N/A
500	3	BEQ	N/A	N/A	4	N/A
501	4	MULT	3	N/A	5	10000
502	5	WRITE_TO_DISPLAY	N/A	N/A	6	10000
503	6	ADD	2	N/A	7	101
504	7	JUMP	N/A	N/A	3	N/A
505	3	BEQ	N/A	N/A	8	N/A
506	8	NOP	N/A	N/A	9	N/A

Figure 9: Shows the debug file for the last 28 instructions

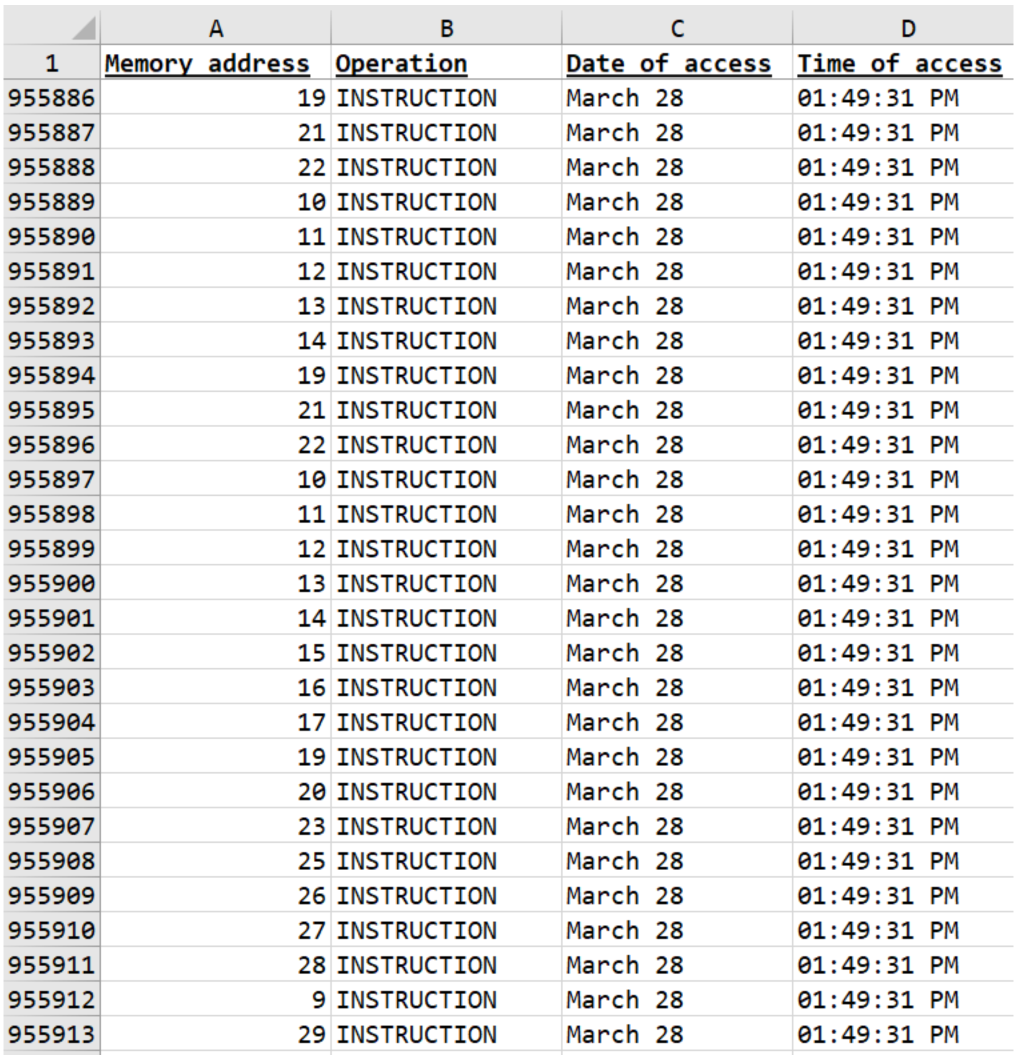
Appendix B- showcase program 2

A snapshot of the time history of memory accesses for the ‘prime numbers between 1 and 1000’ showcase program is shown below. Figure 10 shows the first 28 memory accesses, six of which are for loading data from memory to the registers, all the others are fetching instructions. Figure 11 shows the last 28 memory accesses, all of which are instruction fetches.



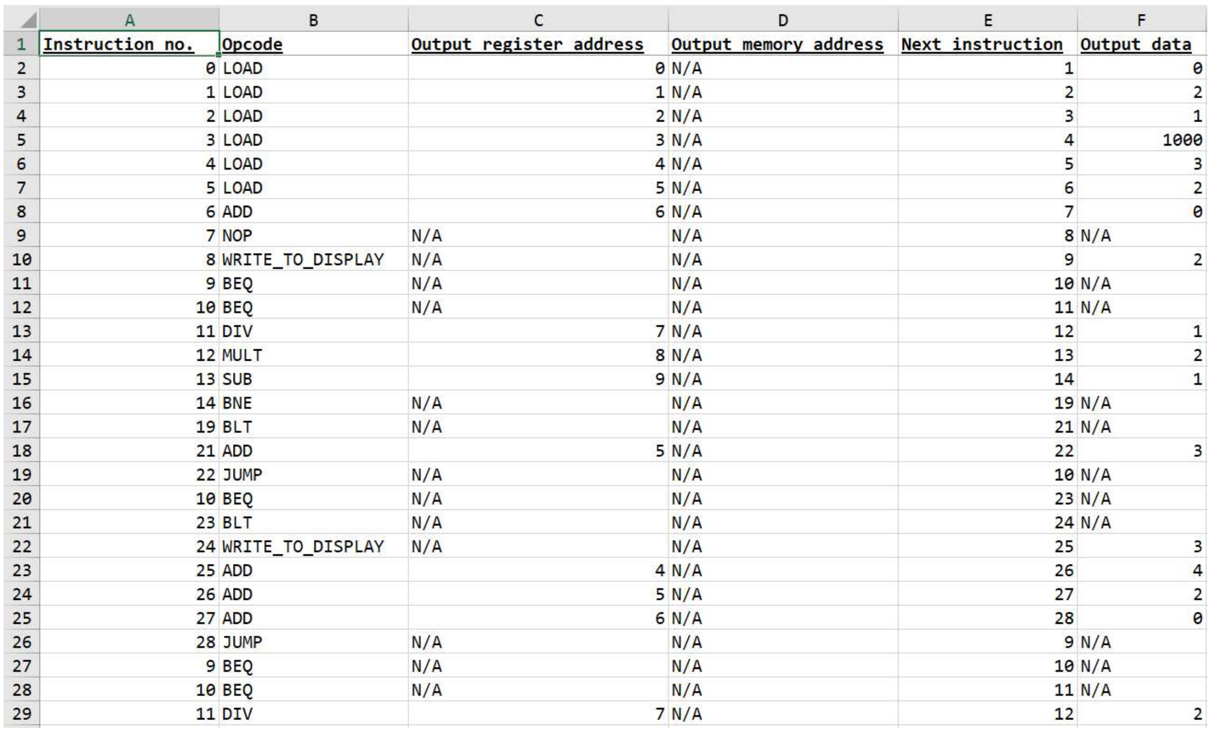
	A	B	C	D
1	Memory address	Operation	Date of access	Time of access
2	0	INSTRUCTION	March 28	01:49:07 PM
3	32768	LOAD	March 28	01:49:07 PM
4	1	INSTRUCTION	March 28	01:49:07 PM
5	32769	LOAD	March 28	01:49:07 PM
6	2	INSTRUCTION	March 28	01:49:07 PM
7	32770	LOAD	March 28	01:49:07 PM
8	3	INSTRUCTION	March 28	01:49:07 PM
9	32771	LOAD	March 28	01:49:07 PM
10	4	INSTRUCTION	March 28	01:49:07 PM
11	32772	LOAD	March 28	01:49:07 PM
12	5	INSTRUCTION	March 28	01:49:07 PM
13	32769	LOAD	March 28	01:49:07 PM
14	6	INSTRUCTION	March 28	01:49:07 PM
15	7	INSTRUCTION	March 28	01:49:07 PM
16	8	INSTRUCTION	March 28	01:49:07 PM
17	9	INSTRUCTION	March 28	01:49:07 PM
18	10	INSTRUCTION	March 28	01:49:07 PM
19	11	INSTRUCTION	March 28	01:49:07 PM
20	12	INSTRUCTION	March 28	01:49:07 PM
21	13	INSTRUCTION	March 28	01:49:07 PM
22	14	INSTRUCTION	March 28	01:49:07 PM
23	19	INSTRUCTION	March 28	01:49:07 PM
24	21	INSTRUCTION	March 28	01:49:07 PM
25	22	INSTRUCTION	March 28	01:49:07 PM
26	10	INSTRUCTION	March 28	01:49:07 PM
27	23	INSTRUCTION	March 28	01:49:07 PM
28	24	INSTRUCTION	March 28	01:49:07 PM
29	25	INSTRUCTION	March 28	01:49:07 PM

Figure 10: The first 28 memory accesses



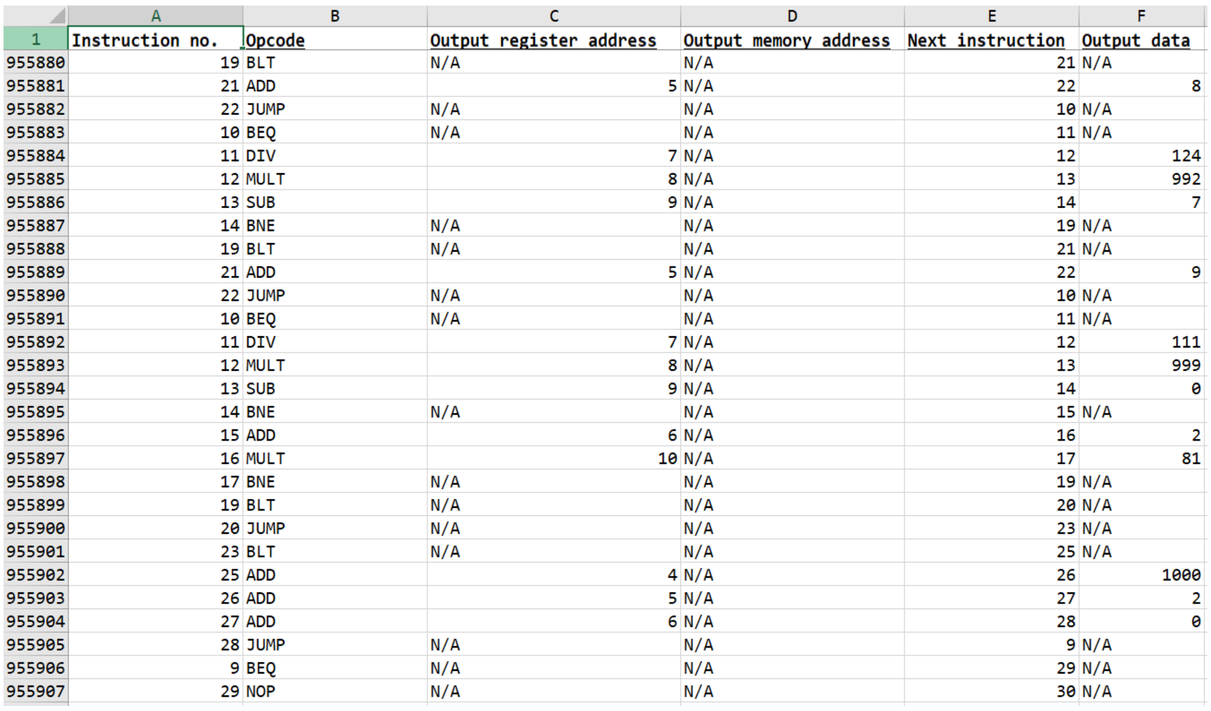
	A	B	C	D
1	Memory address	Operation	Date of access	Time of access
955886	19	INSTRUCTION	March 28	01:49:31 PM
955887	21	INSTRUCTION	March 28	01:49:31 PM
955888	22	INSTRUCTION	March 28	01:49:31 PM
955889	10	INSTRUCTION	March 28	01:49:31 PM
955890	11	INSTRUCTION	March 28	01:49:31 PM
955891	12	INSTRUCTION	March 28	01:49:31 PM
955892	13	INSTRUCTION	March 28	01:49:31 PM
955893	14	INSTRUCTION	March 28	01:49:31 PM
955894	19	INSTRUCTION	March 28	01:49:31 PM
955895	21	INSTRUCTION	March 28	01:49:31 PM
955896	22	INSTRUCTION	March 28	01:49:31 PM
955897	10	INSTRUCTION	March 28	01:49:31 PM
955898	11	INSTRUCTION	March 28	01:49:31 PM
955899	12	INSTRUCTION	March 28	01:49:31 PM
955900	13	INSTRUCTION	March 28	01:49:31 PM
955901	14	INSTRUCTION	March 28	01:49:31 PM
955902	15	INSTRUCTION	March 28	01:49:31 PM
955903	16	INSTRUCTION	March 28	01:49:31 PM
955904	17	INSTRUCTION	March 28	01:49:31 PM
955905	19	INSTRUCTION	March 28	01:49:31 PM
955906	20	INSTRUCTION	March 28	01:49:31 PM
955907	23	INSTRUCTION	March 28	01:49:31 PM
955908	25	INSTRUCTION	March 28	01:49:31 PM
955909	26	INSTRUCTION	March 28	01:49:31 PM
955910	27	INSTRUCTION	March 28	01:49:31 PM
955911	28	INSTRUCTION	March 28	01:49:31 PM
955912	9	INSTRUCTION	March 28	01:49:31 PM
955913	29	INSTRUCTION	March 28	01:49:31 PM

Figure 11: The last 28 memory accesses



	Instruction no.	Opcode	Output register address	Output memory address	Next instruction	Output data
2	0	LOAD	0	N/A	1	0
3	1	LOAD	1	N/A	2	2
4	2	LOAD	2	N/A	3	1
5	3	LOAD	3	N/A	4	1000
6	4	LOAD	4	N/A	5	3
7	5	LOAD	5	N/A	6	2
8	6	ADD	6	N/A	7	0
9	7	NOP	N/A	N/A	8	N/A
10	8	WRITE_TO_DISPLAY	N/A	N/A	9	2
11	9	BEQ	N/A	N/A	10	N/A
12	10	BEQ	N/A	N/A	11	N/A
13	11	DIV	7	N/A	12	1
14	12	MULT	8	N/A	13	2
15	13	SUB	9	N/A	14	1
16	14	BNE	N/A	N/A	19	N/A
17	19	BLT	N/A	N/A	21	N/A
18	21	ADD	5	N/A	22	3
19	22	JUMP	N/A	N/A	10	N/A
20	10	BEQ	N/A	N/A	23	N/A
21	23	BLT	N/A	N/A	24	N/A
22	24	WRITE_TO_DISPLAY	N/A	N/A	25	3
23	25	ADD	4	N/A	26	4
24	26	ADD	5	N/A	27	2
25	27	ADD	6	N/A	28	0
26	28	JUMP	N/A	N/A	9	N/A
27	9	BEQ	N/A	N/A	10	N/A
28	10	BEQ	N/A	N/A	11	N/A
29	11	DIV	7	N/A	12	2

Figure 12: The data log of the first 28 instructions



	Instruction no.	Opcode	Output register address	Output memory address	Next instruction	Output data
955880	19	BLT	N/A	N/A	21	N/A
955881	21	ADD	5	N/A	22	8
955882	22	JUMP	N/A	N/A	10	N/A
955883	10	BEQ	N/A	N/A	11	N/A
955884	11	DIV	7	N/A	12	124
955885	12	MULT	8	N/A	13	992
955886	13	SUB	9	N/A	14	7
955887	14	BNE	N/A	N/A	19	N/A
955888	19	BLT	N/A	N/A	21	N/A
955889	21	ADD	5	N/A	22	9
955890	22	JUMP	N/A	N/A	10	N/A
955891	10	BEQ	N/A	N/A	11	N/A
955892	11	DIV	7	N/A	12	111
955893	12	MULT	8	N/A	13	999
955894	13	SUB	9	N/A	14	0
955895	14	BNE	N/A	N/A	15	N/A
955896	15	ADD	6	N/A	16	2
955897	16	MULT	10	N/A	17	81
955898	17	BNE	N/A	N/A	19	N/A
955899	19	BLT	N/A	N/A	20	N/A
955900	20	JUMP	N/A	N/A	23	N/A
955901	23	BLT	N/A	N/A	25	N/A
955902	25	ADD	4	N/A	26	1000
955903	26	ADD	5	N/A	27	2
955904	27	ADD	6	N/A	28	0
955905	28	JUMP	N/A	N/A	9	N/A
955906	9	BEQ	N/A	N/A	29	N/A
955907	29	NOP	N/A	N/A	30	N/A

Figure 13: The data log of the last 28 instructions