MIDDLE EAST TECHNICAL UNIVERSITY

Spring -2021

EEE 446

Final Delivery of 32 Bits Integer CPU :

Programming & Verification of a Pipelined CPU with Basic Cache Hierarchy

CPU name: NASH CPU

Submitted to: Prof. Ali Muhtaroglu

Team Members: Abdelaziz Al-Najjar ID:2269512

Abduallah Damash ID: 2281772

Contents

Table of figures

"The content of the report represents the work completed by the submitting team only, and no material has been borrowed in any form."

I.Introduction and Objectives:

In previous modules, the team successfully designed and built a 32-bit in-order integer pipelined CPU with data forwarding, an instruction set architecture named NASH, and actively integrated a two-level memory hierarchy model using physical addressing provided by Professor Ali Muhtaroglu. This last module came intending to complete the verification of our NASH CPU and fix any missing pieces.

The CPU functionality will be verified by applying three benchmarks aimed to test different aspects. The first benchmark is the integer DAXPY with 2-d victors. This benchmark is mainly aimed to apply a multiply-accumulate operation on two matrices. The second benchmark is the Merge Sorter plus CRC. This benchmark has two main parts that could be divided into smaller functions. It is aiming to re-order a given list and appending a CRC code to that list. Our last benchmark is the Text Parser. This benchmark will investigate a given text and report all types of spaces in that list, as well as removing any duplicated space. All these benchmarks are going to be written and assembled using NASH ISA.

The work was divided as follows:

- Abdelaziz was responsible for updating the NASH ISA with any crucial changes. He was also responsible for implementing and testing the integer DAXPY benchmark. In addition, he was responsible for testing the Merge Sorter part of the second benchmark. He made sure to include all Verilator tests results and relative codes in the appendix.
- Abdullah was responsible for reporting all Quartus II performance test results. He was also responsible for implementing and testing the Text Parser. In addition, he was responsible for the CRC appending section of the second benchmark. He made sure to include all Verilator tests results and relative codes in the appendix.

In the following Implementation Results section, some information on the NASH ISA and the CPU details will be given. Afterward, details on the benchmarks will be displayed. Moreover, testing code details for each benchmark will be shown, along with the results. Finally, Quartus II verification results will be given. ISA specification and all used codes and test outputs will be illustrated in the appendix.

II. Implementation Results:

i. Information on NASH ISA:

The main aim of the NASH ISA is to cover the most common programmer needs and be capable of completing the required benchmarks. NASH ISA has all essential instructions, starting from simple ALU instructions, such as ADD, SUB, MUL, Shift left/ right, and Shift Arithmetic/ Logic, to conditional and unconditional jumps, such as Branch if Equal or CALL. NASH ISA tries to best take advantage of a second ALU in the MEM stage by having instructions like Multiply-Accumulate, where it multiplies in the first ALU then adds in the second ALU.

Our ISA has four main instruction formats to cover all necessary instructions. Some of our formats are based on MIPS ISA, such as R-type and I-type. R-type is borrowed to cover most of our simple arithmetic instructions, where we operate on two registers. At the same time, I-type is mainly for instructions where you need an immediate value, such as arithmetic operation with immediate, or memory instructions where the immediate is used as an offset. Next, we have our new type R-CAL. this type is mainly for complex operations where you need to operate on three registers, especially if you would like to use the second ALU for additional calculations. Our last type is the Jump-type. this type is used to perform different types of unconditional jumps. Such as jumping to a subroutine using a CALL instruction.

More details on our ISA specifications are available in Appendix A.

ii.NASH CPU details:

a. NASH Datapath:

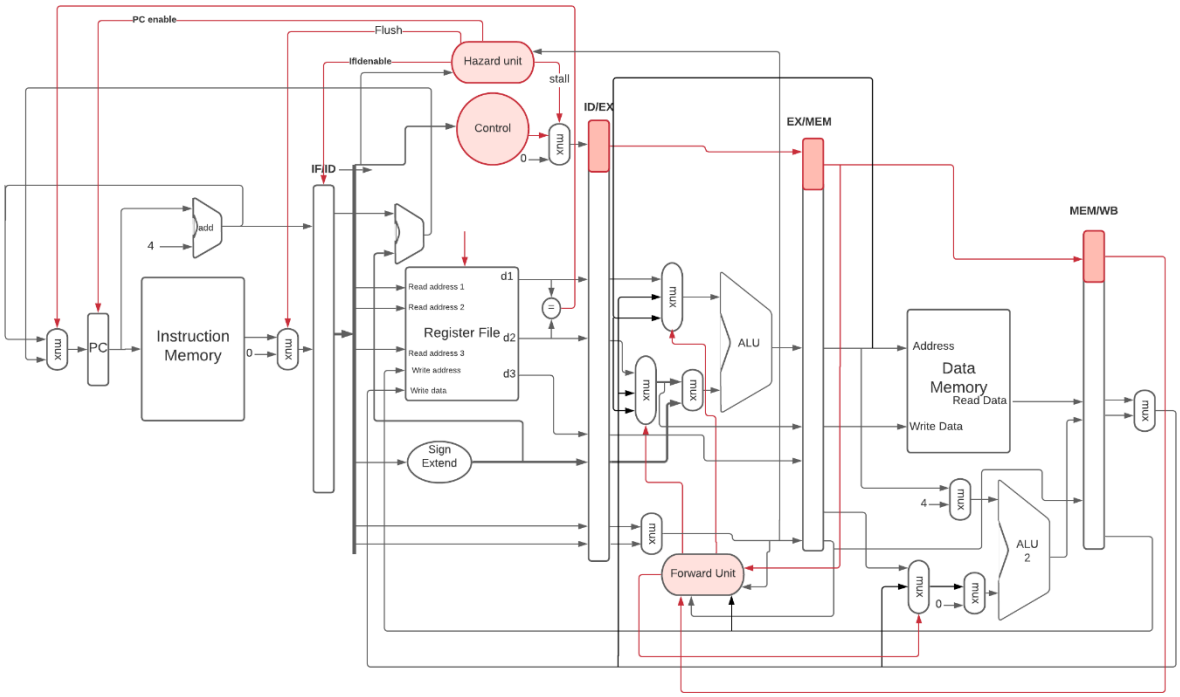


Figure 1 NASH CPU Datapath

Note: some details are not shown in the Datapath above, such as ALU control 1 and ALU control 2

The main features of the NASH pipeline can be described as follows:

- The NASH pipeline Datapath consists of five stage registers: Program Counter (PC), IF/ID, ID/EX, EX/MEM, and MEM/WB.
- The 32-bit PC register represents the address for the I-Cache and can be controlled by the hazard unit through an enable signal.
- A hazard unit that keeps track of the load hazard and cache misses and stalls the pipeline when needed. It is also able to flush one instruction in case of a taken branch or Jump instructions.
- The stall is implemented by inserting zeros as control unit outputs, while flush inserts zeros to the IF/ID stage. Cache miss stall is implemented by not updating all stage registers.
- IF/ID register contains the fetched instruction from I-cache and the incremented PC.
- The register file in the decode stage has three reading ports, one writing port. The third port is mainly used by the R-Call type instructions.

- A comparator is added in the decode stage to benefit from the early branch decision-taking and flush only one instruction for a taken condition.
- The EX-stage consists of an ALU capable of doing all types of arithmetic and logic operations and a forwarding unit.
- The forwarding unit ensures that all types of data dependencies are resolved through forwarding. It can forward data to both the first and second ALU.
- The Mem stage has the addition of a second ALU; this ALU is capable of doing all fundamental operations. In case this ALU is not being used, it works as a buffer for the first ALU.
- Lastly, the writeback stage consists of writing the result of the second ALU (buffered when not used) or the memory output, depending on the signals coming from the ALU.

More details on the Datapath and the implemented code are available in appendix B.

b. NASH Control Unit:

The control unit is built as combinational logic that produces the required signals for each instruction, as shown in Table.1.

Note that some spaces in the Operand fields are left blank in purpose in case the team decided to invent a new instruction similar to that category.

Table 1. Control Unit Design.

Category	Instruction	**OpCode** (6)	**Funct (6)**	**ALUOp (4)**	**ALUcontrol (4)**
Logical Operations (ALU operations)	Add (Add)	0x00 000000	0x20 100000	0x2 0010	0x0 0000
	Subtract (Sub)	0x00 000000	0x21 100001	0x2 0010	0x1 0001
	Multiply (Mul)	0x00 000000	0x22 100010	0x2 0010	0x2 0010
	Divide (Div)	0x00 000000	0x23 100011	0x2 0010	0x3 0011
	And (And)	0x00 000000	0x24 100100	0x2 0010	0x4 0100
	Or (Or)	0x00 000000	0x25 100101	0x2 0010	0x5 0101
	Xor (Xor)	0x00	0x26	0x2	0x6

		000000	100110	0010	0110
	Shift Left Logical (Sll)	0x00 000000	0x27 100111	0x2 0010	0x7 0111
	Shift Right Logical (Srl)	0x00 000000	0x28 101000	0x2 0010	0x8 1000
	Shift Right Arithmetic (Sra)	0x00 000000	0x29 101001	0x2 0010	0x9 1001
	Set on Less Than (Slt)	0x00 000000	0x2a 101010	0x2 0010	0xa 1010
Memory Ins.	Load Word (Lw)	0x01 000001	xxxx	0x0 0000	0x0 0000
	Store Word (Sw)	0x02 000010	xxxx	0x0 0000	0x0 0000
	Load Half (Lh)	0x03 000011	xxxx	0x0 0000	0x0 0000
	Store Half (Sh)	0x04 000100	xxxx	0x0 0000	0x0 0000
Empty Spaces (0x05 -0x08)					
ALU operation immediate	Set on Less Than Immediate (Slti)	0x09 001001	xxxx	0x1 0001	0x6 0110
	Add Immediate (Addi)	0x0a 001010	xxxx	0x0 0000	0x0 0000
	And Immediate (Andi)	0x0b 001011	xxxx	0x4 0100	0x4 0100
	Or Immediate (Ori)	0x0c 001100	xxxx	0x5 0101	0x5 0101
Empty Spaces (0x0d -0x0f)					
	Load Upper Immediate (Lui)	0x10 010000	xxxx	0x0 0000	0x0 0000
Empty Spaces (0x11)					
Data Flow	Br On Equal (Beq)	0x12 010010	xxxx	xxxx	xxxx
	Br On Not Equal (Bne)	0x13 010011	xxxx	xxxx	xxxx

	Jump (J)	0x14 010100	xxxx	xxxx				
	Jump register. (Jr)	0x15 010101	xxxx	xxxx	xxxx			
	Call (Call)	0x16 010110	xxxx	xxxx	xxxx			
		Opcode	Funct1	ALU Op1	ALU control1	Funct2	ALU Op2	ALU control2
News Ins.	Jump and Store (Jsw)	0x17 010111	xxxx	0x0 0000	0x0 0000	xxxx	xxxx	xxxx
	Multiply then Add (Mula)	0x18 011000	0x2 0010	0x2 0010	0x2 0010	0x0 0000	0x2 0010	0x0 0000
	Xor then Shift lift (XoSL)	0x19 011001	0x6 0110	0x2 0010	0x6 0110	0x7 0111	0x2 0010	0x7 0111
	Store and Increment (Swin)	0x1a 011010	0x0 0000	0x2 0010	0x0 0000	xxxx	xxxx	xxxx

Now, the next table will show the ALU control that used for both ALU.

Table 2. ALU Control Design.

ALUOP (4)	**ACTION**
0X0 0000	Add
0X1 0001	Sub
0X2 0010	Decide on function
0X3 0011	Mul
0X4 0100	And
0X5 0101	OR
0X6 0110	Slt
0X7 0111	Xor
0X8 1000	Shift lift
0X9 1001	Shift Right
0XA 1010	No Action

Notice that in our ISA, we add four new instructions which are Jump and Store (Jsw), Multiply then Add (Mula), Xor then Shift lift (XoSL), and Store and Increment (Swin) that used in implementing our benchmark.

iii.Benchmark details:

To test the performance of our CPU, we were required to apply three different benchmarks. In this section of the report, we will describe each benchmark and introduce the developed algorithms and how our ISA is helping to improve the performance.

Note that All NASH ISA codes for all functions are available in Appendix C.

a. Integer Daxpy with 2-d Vectors:

The first benchmark is the Integer Daxpy with 2-d Vectors. The idea behind this benchmark is to test the capability of the CPU of handling complex mathematical operations. This program will apply a multiply-accumulate operation on two matrices, A and B, of size n x n, where n<20, and a constant *k.* Both *k* constant and matrices A and B are 16-bit signed integers. The result of the multiply-accumulate operation will be stored back to matrix B, forming the following formula:

$$\mathbf{B = k * A + B}$$

A and B matrices are stored in sequential order in the memory. Thus, the first n x n locations are for matrix A, following n x n locations are for B.

The program will be called with the three arguments, the constant *k*, the size n, and the first location of matrix A. The proposed algorithm will start with initializing a counter. This counter will be used directly to load values from matrix A; then, an $\mathbf{n^2}$ will be added to the counter to load the B value. After loading both values, we will use our special multiply-accumulate instruction, *MULA*. Afterward, we will use our store and jump instruction, *JSW*, to store the result back to B location and jumping to continue for the next value in our matrices. The complete simplified algorithm flowchart is shown below, and the full assembled code is given in Appendix C.

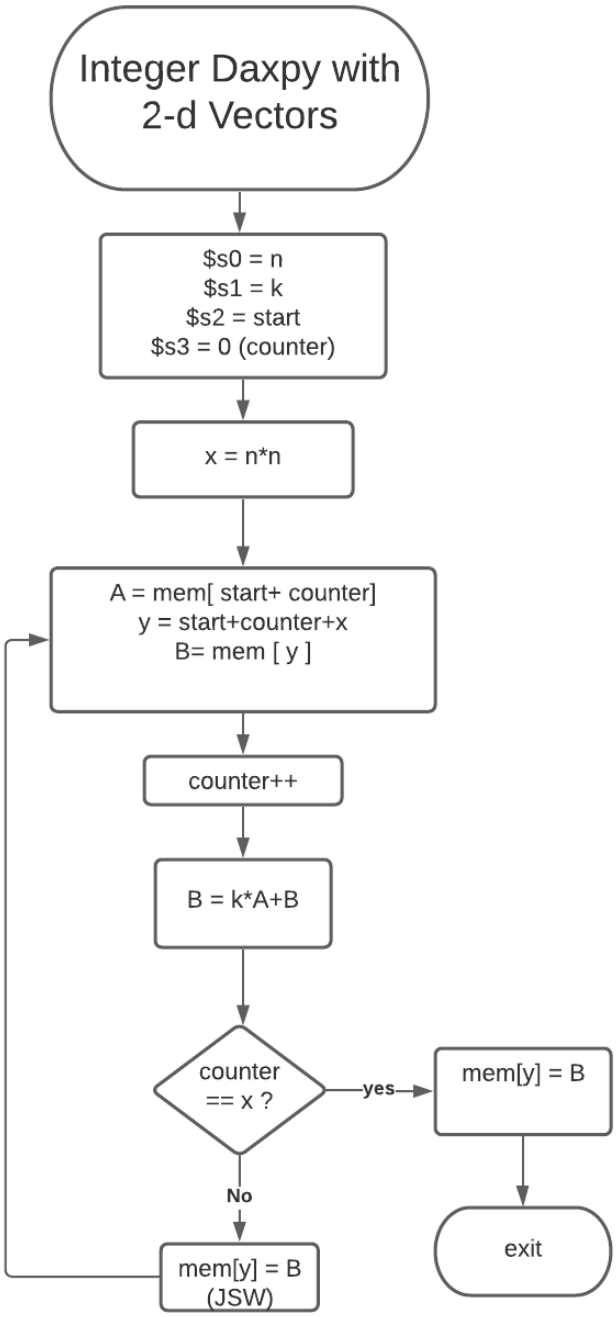


Figure 2: Integer Daxpy Flowchart

b. Sorter + CRC generator:

This benchmark is considered to be the most advanced test for our CPU. It can be divided into two primary operations, a. sorting, and b. CRC generating. Therefore, the team has divided the code accordingly.

II.iii.b.1 1. Sorting

For sorting, a MergeSorter algorithm has been used. It consists of three main functions, Length function, MergeSort function, and Merge function.

1. Length function: this function's job is to determine the length of any given list. It will count the filled locations starting from the given list until a null is detected and report the length.

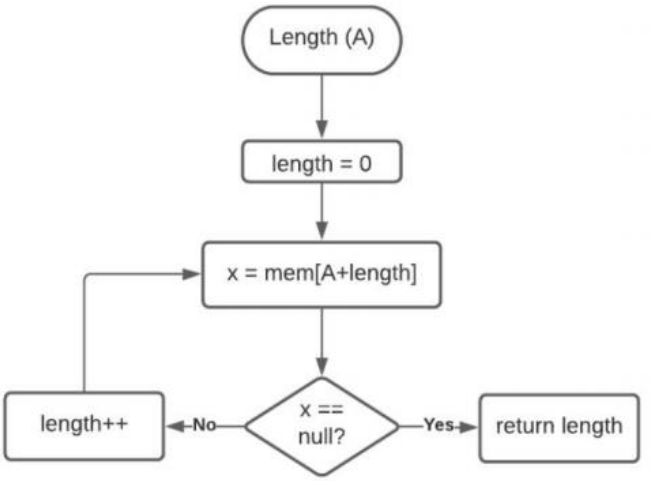


Figure 3: Length function algorithm

2. MergeSort function: this function task recursively divides a given list until reaching an atomic list. It will divide the given list into two halves and divide these halves into smaller lists. When the length of the list is less than 2, it will repeatedly call the Merge function to merge these small lists back.

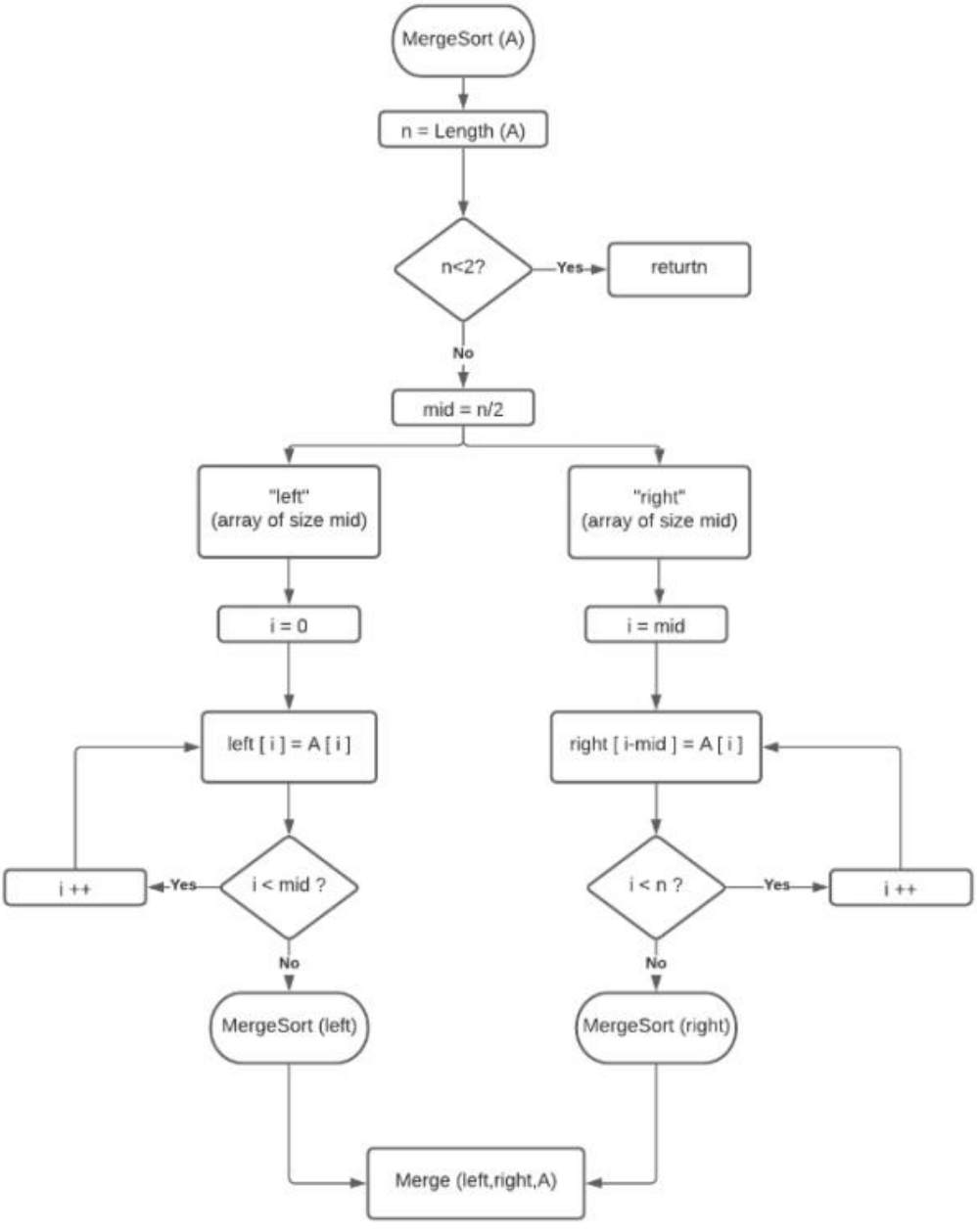


Figure 4: MergeSort function algorithm

3. Merge function: this function is responsible for reconstructing the divided lists by the MergeSort function. It will get three arguments: the left list address, the correct list address, and an address to store the combined lists. Then, it will simply compare each element of the left list with the right list and store the larger value first. It will repeat this operation until null is reached.

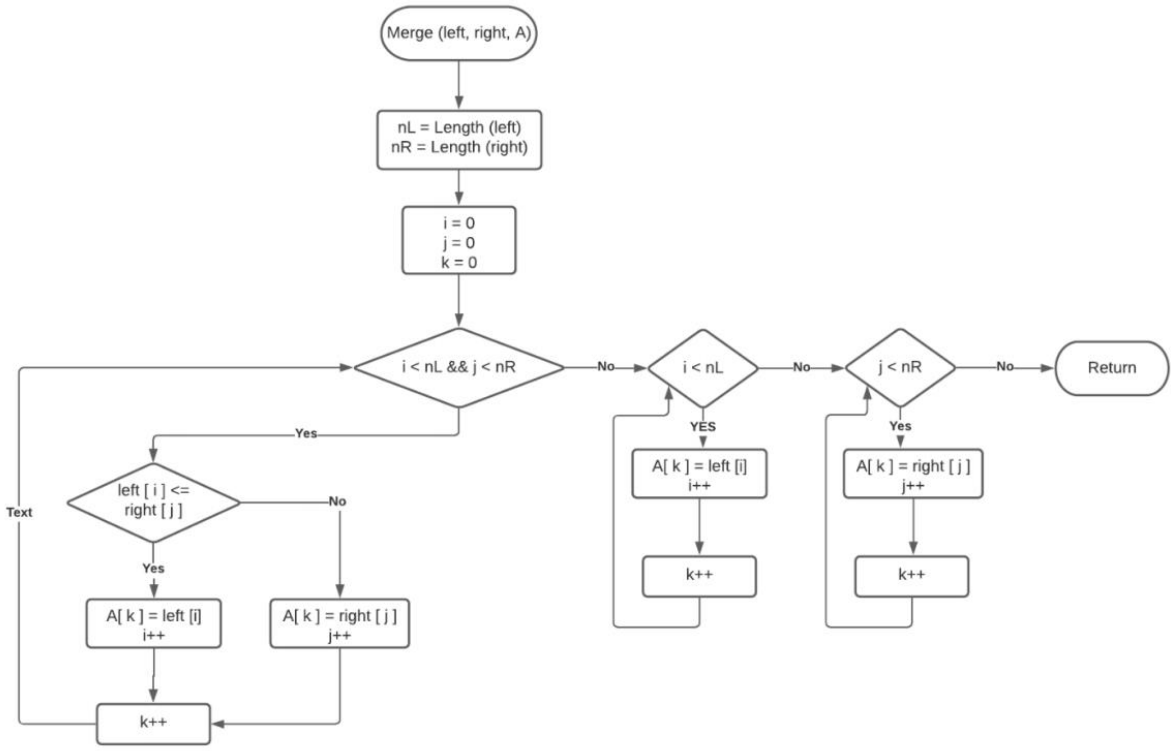


Figure 5: Merge function algorithm

II.iii.b.2 2. CRC Generator:

The Cyclic Redundancy Check (CRC) function is one of the essential benchmark applications in the IoT field and wireless sensor networks since it detects accidental changes/errors in the communication channel. In our benchmark, the function will calculate 16-bit CRC-CCITT that has a truncated polynomial of 0x1021 on a given 16 bits inputs. Then, it will append the CRC code in the following address of the addresses of the inputs. The flowchart of the algorithm is shown in Fig. 6.

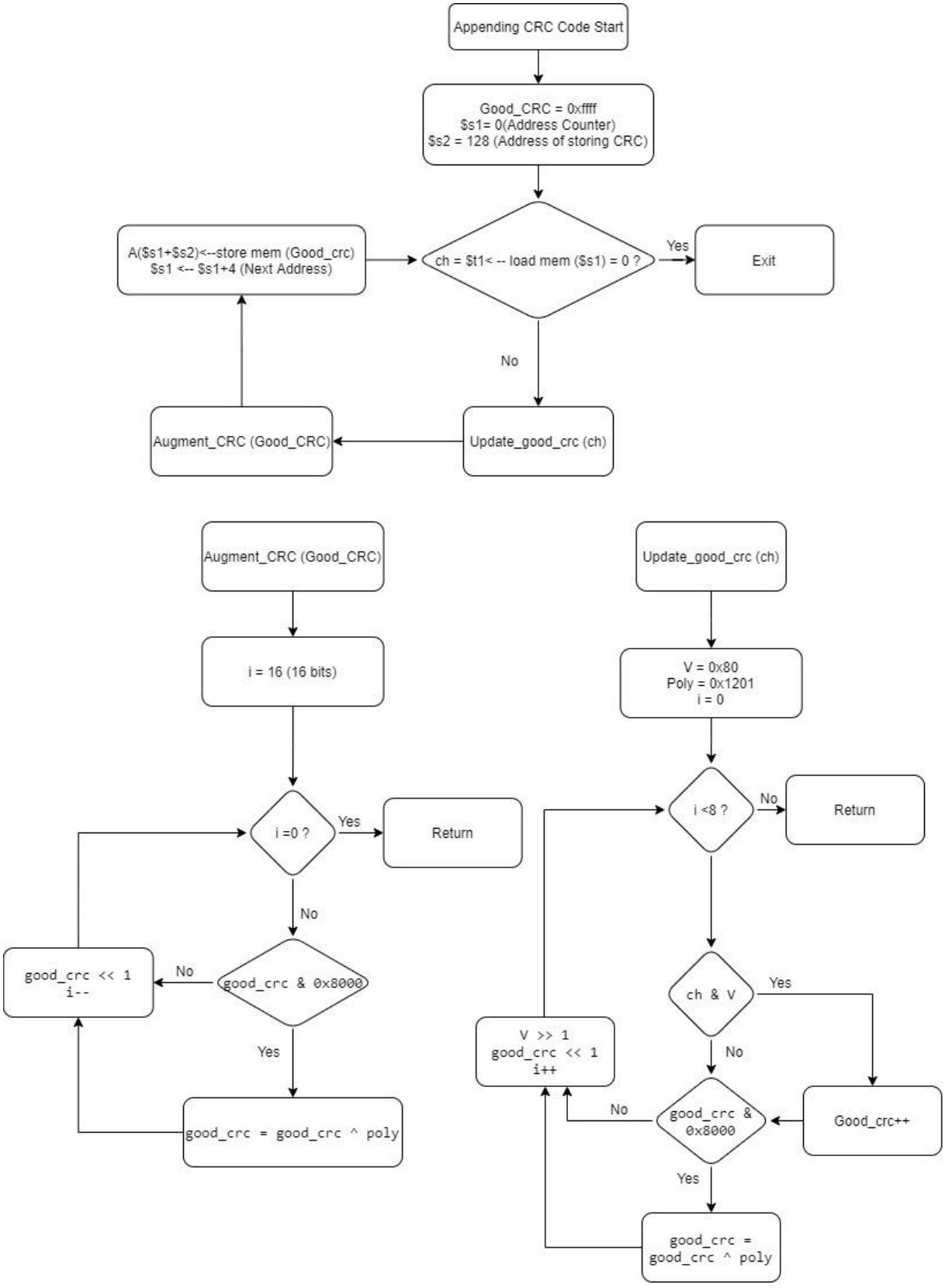


Figure 6. CRC Function Flowchart.

c. Text Parse:

This benchmark is one of the Natural Language Processing (NLP) applications as "it is focused on enabling computers to understand and process human languages" based on specific rules. In our benchmark, the text parse counts all space formats such as space, form feed, tab, etc., and counts the non-space characters given in ASCII format, and it will stop when it reads a null input. Also, if there are identical spaces, it will replace the second one with null to organize the text. Then, it will store the number of deleted spaces and characters in the following two addresses after the text is finished. The flow chart of the algorithm is shown in Fig.7.

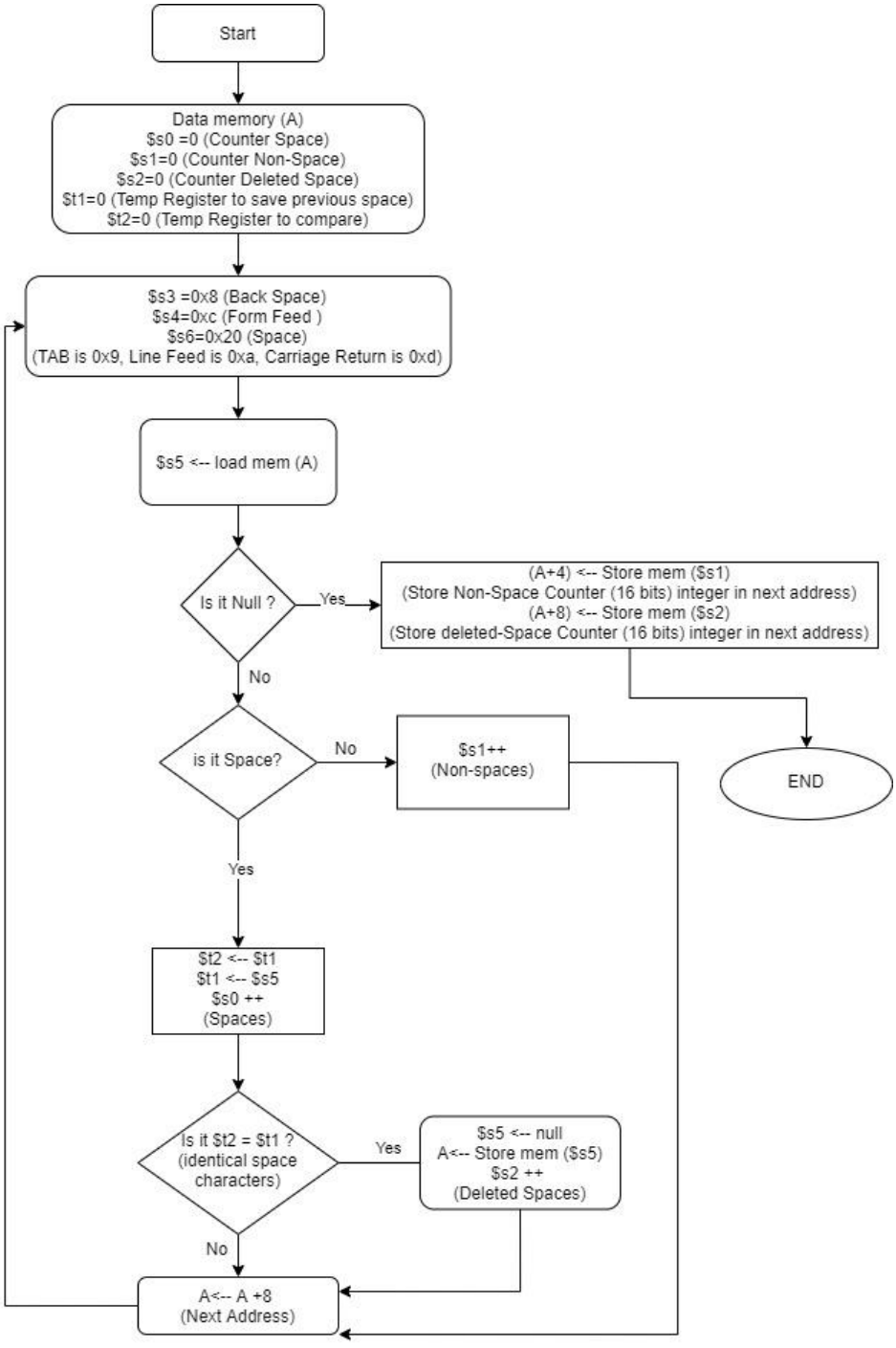


Figure 7. Text Parser Flowchart.

III. Testing results:

After introducing our benchmarks in section 3, this section will display the Verilator test descriptions along with the test results for each benchmark. Afterward, Quartus II verification results of the CPU will be explicitly represented.

Note that All NASH Verilator codes for all testing and output screenshot are available in Appendix D, and F, respectively.

i. Verilator tests

a. Integer Daxpy with 2-d Vectors:

III.i.a.1 1. Testing functionality:

For checking the functionality of the proposed Daxpy algorithm, we assigned the size of the matrices to be 3 x 3 (9 values). We made sure that the values are as large as 16 bit, also contain signed values. For example, A[1] = 0x10DE, B[1] = 0xFC18, and *k* = 0x7FFC. This will ensure that we covered all possible value cases. At a certain cycle, *i.e., 180,* we will check the stored value in B[1] location, if the result is equal to A[1]*k+B[1], i.e., = 0x086EB8A0, then the algorithm passed the first functionality test. Another test is to make sure that the load and store operation is correct. If load operations equal to the size of both matrices combined and store operations are equal to matrix B size, then the algorithm works as desired.

III.i.a.2 2. Checking performance:

The test will report the number of instructions needed for this benchmark, how many of them are load/store instructions, how many stall cycles, how many of these stalls are due to load hazard, and how many are due to memory hierarchy. Also, we made sure that the test reports total clock cycles and execution time concerning fmax found from Quartus II tests. Finally, report effective CPI with and without memory stalls.

The results were as follows:

Table 3. Daxpy Output Results.

Metric	Load instructions	Store instructions	Clock cycles	Memory stalls	Hazard stalls	IC	Execution time (us)	Effective CPI	CPI with Memory	Functionality result
daxpy	18	9	274	185	9	70	0.35	3.47	1.04	PASS

b. Sorter + CRC:

III.i.b.1 1. Sorting:

For sorting, the implemented code by the team has three main parts, as described earlier.

- Length:

To check the Length function's functionality, we set the DRAM with a list of a certain length. Then, the Verilator test will compare the returned value ($v0) from the length function with the actual size, for example, 48 elements, and report the PASS/FAIL results.

The test will report the number of instructions needed for this function, how many of them are load/store instructions, how many stall cycles, how many of these stalls are due to load hazard, and how many are due to memory hierarchy. Also, we made sure that the test reports total clock cycles and execution time concerning fmax found from Quartus II tests. Finally, report effective CPI with and without memory stalls.
The results were as follows:

Table 4. Length Function Output Results.

Metric	Load instructions	Store instructions	Clock cycles	Memory stalls	Hazard stalls	IC	Execution time (us)	Effective CPI	CPI with Memory	Functionality result
Length	49	0	678	332	49	335	0.87	1.77	1.1	PASS

- MergeSort:

Due to the relatively more extended MergeSorter codes, the provided Memory Hierarchy could not handle their operations. The team believes that the problem was with the way fetching instructions to I-cache was implemented. When an instruction is called by PC, the following eight instructions will be appended to call instructions in the SRAM, and so on. However, when an instruction far from the already fetched instructions is called, i.e., with a call instruction, they will show below the old instructions and has wrong PC values.

Therefore, the team has decided to provide the Verilator tests results of MergeSort functions without the memory hierarchy implementations.

To check the functionality of the merge function, we set the DRAM with two lists of a certain length, i.e., four elements each, and separate these lengths with a null. Also, provide the function with an address to store the merged list. The verilator test will monitor the output list address and compare each address of the list with the following element to ensure the ascending order. The test will report the input lists, merged list, and Pass/Fail result.

The test will report the number of instructions needed for this benchmark, how many of them are load/store instructions, how many stall cycles, how many of these stalls are due to load hazards. Also, we made sure that the test reports total clock cycles and execution time using fmax found from Quartus II tests. Finally, report effective CPI.

Table 5. MergeSort Function Output Results.

Metric	Load instructions	Store instructions	Clock cycles	Load Hazard stalls	IC	Execution time (us)	Effective CPI	Functionality result
MergeSort	32	12	241	32	70	0.31	1.22	PASS

Note: the results shown in the table above do not include the memory hierarchy effect; it is expected to increase significantly when combined with the hierarchy.

III.i.b.2 2. CRC:

In order to test the functionality of our benchmark, we test with input 'A' (0x41) which will produce the CRC-TCITT of (0x9479), and to verify this result is correct, we wrote a program that implement the same program in C language, and in the appendix, there will be an output screenshot of the that program. Also, this Verilator tests the functionality of the output by compering the output with the provided CRC form the C program. Thus, the test will report the number of instructions needed for this benchmark, how many of them are load/store instructions, how many stall cycles, how many of these stalls are due to load hazard, and how many are due to memory hierarchy. Also, we made sure that the test reports total clock cycles and execution time concerning fmax found from Quartus II tests. Finally, report effective CPI with and without memory stalls, and Energy consumption.

The results were as follows:

Table 6. CRC Function Output Results.

Metric	Load ICs	Store ICs	Clock cycles	Memory stalls	Hazard stalls	IC	Execution time (us)	Effective CPI	CPI with Memory	Energy (uJ)	Functionality result
CRC	6	1	694	369	59	199	11.07	2.69	1.11	13.39	PASS

c. Text parser:

III.i.c.1 Testing functionality:

For checking the functionality of the proposed Text parser algorithm, we assigned following text:

Text (Space) Parser (Space) Test (Space) Result (Space) (Space),
(Line Feed)
(Line Feed)
End (Space) of (Space) the (Space) Test (Tab) (Form Feed) (Carri Return) (Null).

Then, the total space counter will be 13, the deleted spaces will be 2, and the characters will be 32.

III.i.c.2 2. Checking performance:

The test will report the number of instructions needed for this benchmark, how many of them are load/store instructions, how many stall cycles, how many of these stalls are due to load hazard, and how many are due to memory hierarchy. Also, we made sure that the test reports total clock cycles and execution time concerning fmax found from Quartus II tests. Finally, report effective CPI with and without memory stalls and energy consumption knowing that the total average power is 1.21 W. Also, the deleted spaces counter, the space counters, and characters counter. In the appendix.

The results were as follows:

Table 7. Text Parser Output Results.

Metric	Load ICs	Store ICs	Clock cycles	Memory stalls	Hazard stalls	IC	Execution time (us)	Effective CPI	CPI with Memory	Energy (uJ)	Functionality result
Text Parser	47	4	1529	443	154	811	24.39	1.09	1.58	24.39	PASS

ii.Quartus II Verification:

Table 8. Cost, Timing, Power Dissipation of Whole CPU.

Parameter	Datapath + Control	D-Cache	I-Cache	Full CPU
Logic elements	4008	5382	2536	11926
Total registers	1455	4179	2090	7724
Total memory bits	39	0	0	39
Embedded multiplier elements	12	0	0	12
M4K elements	1	0	0	1
Fmax (MHz)	62.68	100.72	145.56	62.68
Total thermal power dissipation (mW)	328.74	570.78	311.40	1210.92
Total thermal power by I/O (mW)	51.30	255.34	80.65	387.29
Total thermal power by M4K (mW)	1.26	0	0	1.26
Total thermal power by Embedded multiplier block (mW)	5.22	0	0	5.22
Total thermal power by Combinational cell (mW)	33.78	65.66	9.39	108.828
Total thermal power by Register cell (mW)	27.19	33.26	10.77	71.22
Total thermal power by Clock control block (mW)	16.17	22.37	17.35	55.89

IV.Conclusion:

Throughout the semester, we started the design by building small pieces of the whole 32 bits CPU by going through the pipeline design until we design our ISA instruction, and then we took advantage of the second ALU in the fourth stage to introduce a new instructions such as Jump and store (Jsw) , multiply and accumulate (Mula) that can reduce the latency of a huge benchmarks such as Daxpy, Text parser..etc.

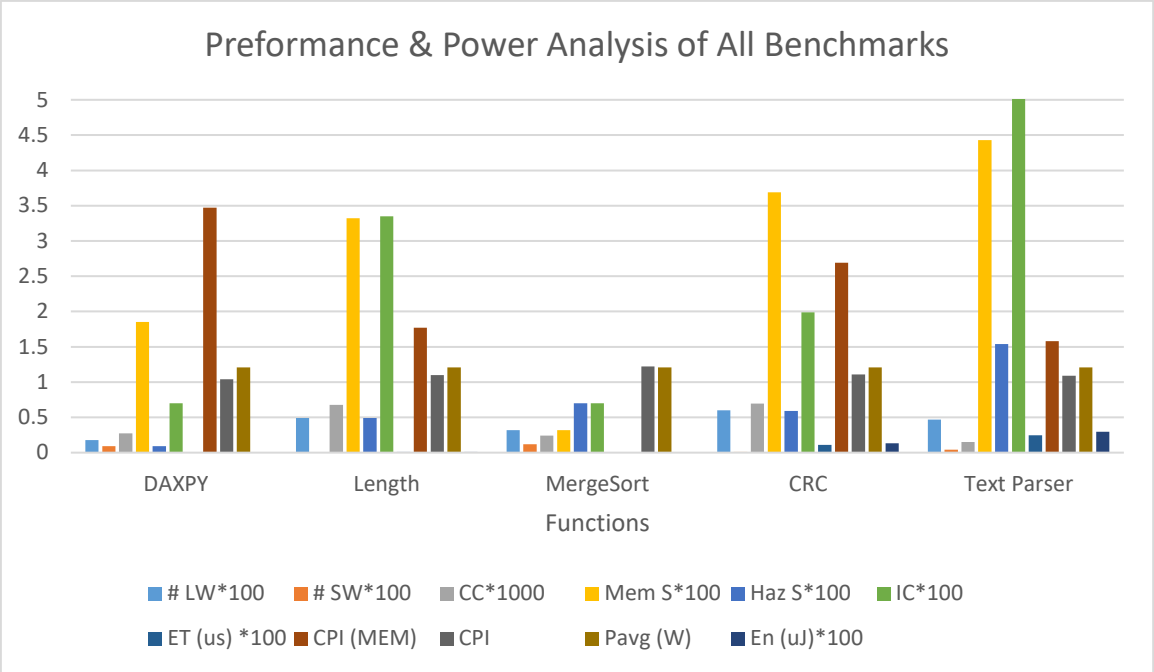
In order to build a realistic CPU that match with the real-world products, we ingrate our CPU with a basic cache memory level so that we verify the work of benchmark in a realistic approach.

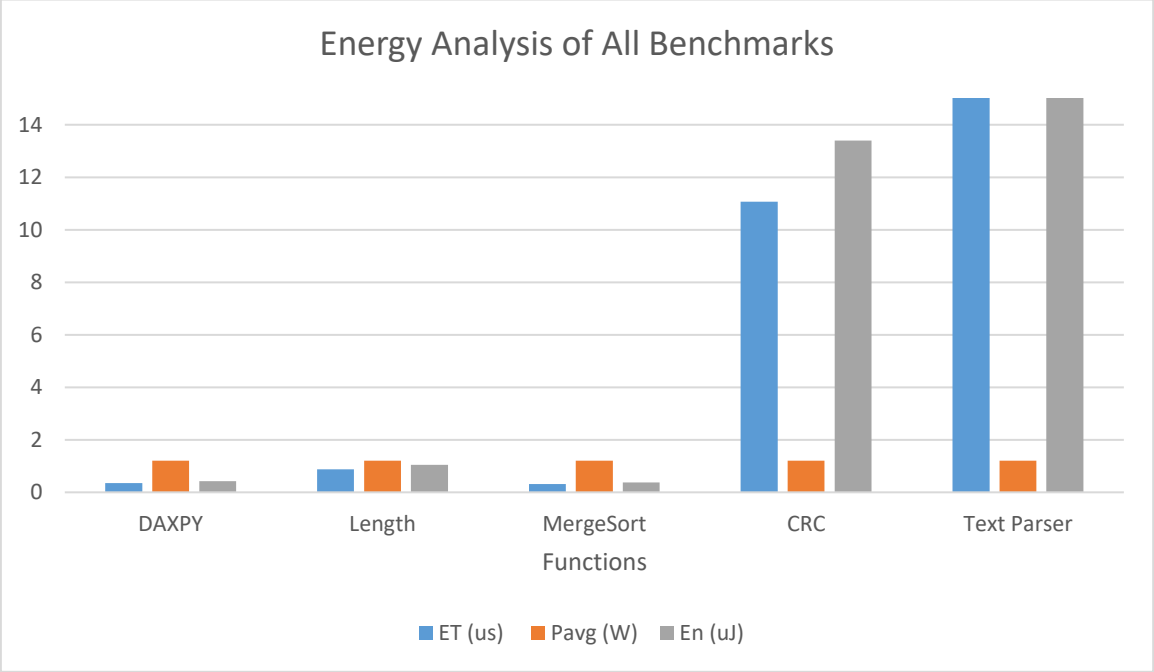
After writing the benchmark in our ISA language and verify the functionality of them, we obtain the following results:

Table 9. Summary of All Benchmarks

Parameter	DAXPY	Length	MergSort	CRC	Text Parser
Clock Cycles	274	678	241	694	1529
Memory Stalls	185	332	-	369	443
Hazard Stalls	9	49	32	59	154
Instruction	70	335	70	199	811
CPI without Mem	1.04	1.1	1.22	1.11	1.09
CPI with Mem	3.47	1.77	-	2.69	1.58
Fmax (MHz)	68.63	68.63	68.63	68.63	68.63
Execution Time (us)	0.35	0.87	0.31	11.07	24.39
Average Power (W)	1.21	1.21	1.21	1.21	1.21
Energy (uJ)	0.42	1.05	0.37	13.39	29.51

It can be seen from the graph shown below that the more input you put for these functions, the more energy consumption; for example, the text parse function is consuming the most energy among the other functions, and that because the inserted text has is big. However, looking at the CRC results, we can say that if we consider more inputs for these functions, it will consume a lot of energy and it will take a longer time and that because these result only for one input, so if there are others, it will scale up the parameters.



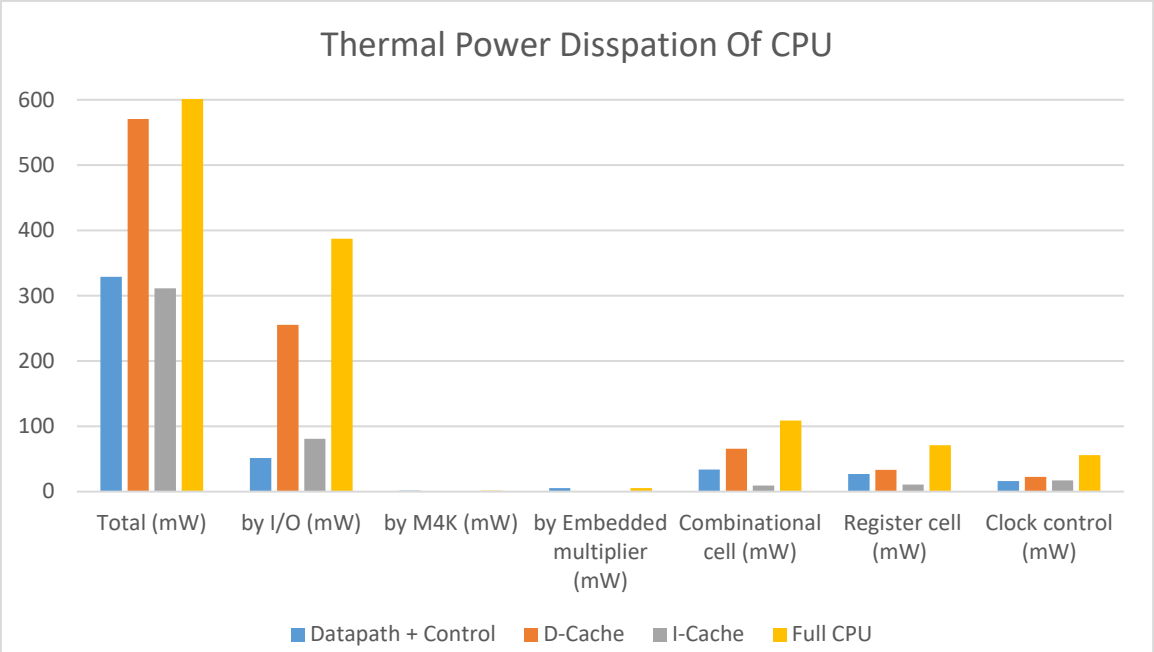


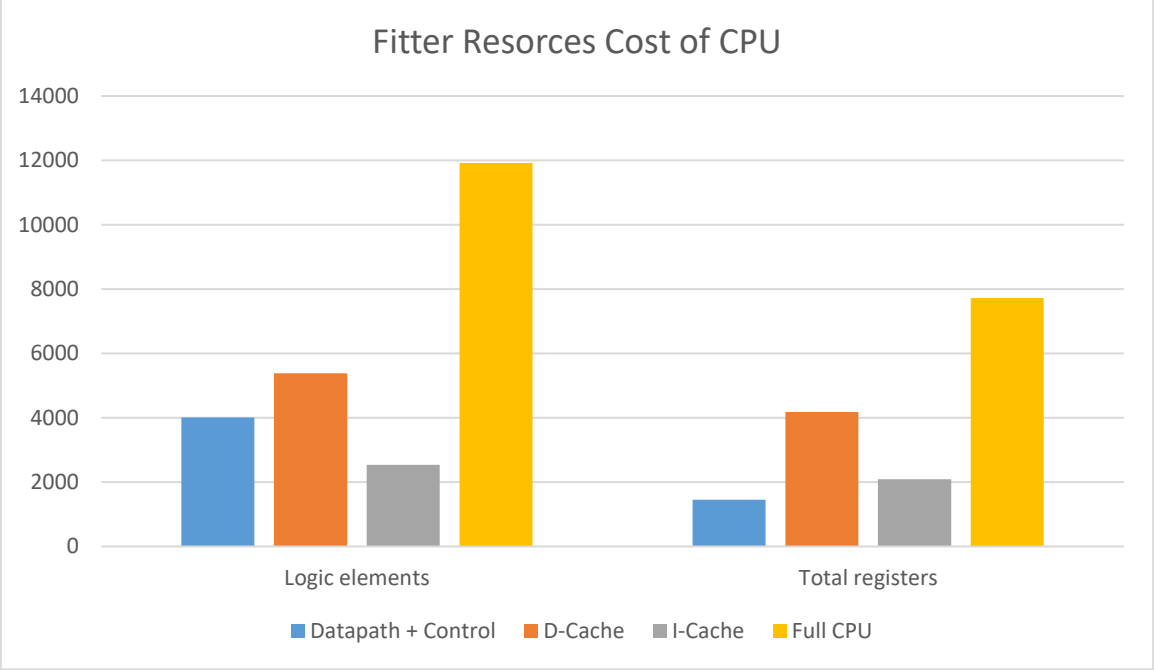
Now, let summarize the cost and power of the whole CPU:

Looking at the graph, we can conclude that most of the power dissipates through the I/O block and that because our CPU has different functions that communicate together, it is expected to be like that.

Also, if we look into the second graph, we will notice that most of the cost spent over the D-cache is expected since it will store the data and need more registers than the other functions.

Also, notice that our Fmax reduced from around 95.5 MHz to 68.63 MHz because of the usage of a combinational multiplier.





In the end, we can conclude that our CPU is working as it is required, and we test it and verify its functionality by implementing different benchmarks that all pass and give the correct results. Also, we can optimize this CPU by implementing booth's multiplier instead of depending on the combinational multiplier that affects our maximum frequency to reduce by 30%, so this is not the end of our Computer Architecture journey, and we will continue improving our CPU and our skills.

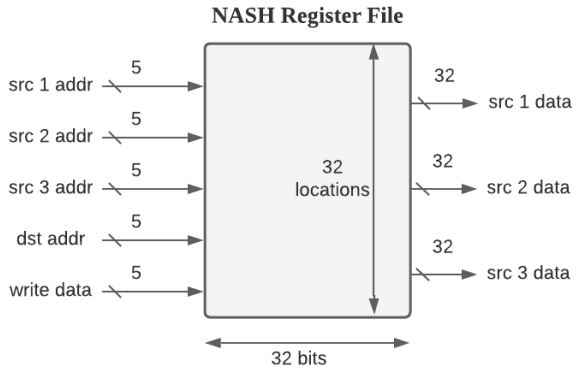
V.Appendix A NASH ISA specification:

Now, after demonstrating the tentative algorithms, we will introduce the NASH ISA Specifications, including Register File details, instruction types, formats, associated operations in register transfer notation, operand sizes, and PC details.

i.NASH Register File:

Starting with the Register File, instead of just assigning all registers to be used the same way, we decided to distribute the register duties similar to MIPS ISA but with slight changes. Since our benchmarks have a lot of access to the memory, we decided that 8 location is enough for temporary use rather than 10. Also, we increased the number of saved values to be 9. We also increased the number of registers used for function returned values since we will use many subroutines in the second benchmark.

- Thirty-two 32-bit registers number 0 to 31
- Three read ports, one write port in RF

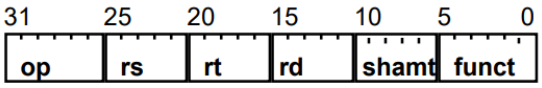


Register File description:

0	$zero	Constant zeros = 0x0000
1	$at	Reserved for Assembler
2 **3**	$v0 $v1	For function results and evaluations
4 **…** **7**	$a0 … $a3	Arguments to be used with functions
8 **…** **15**	$t0 … $t7	For temporary values
16 **…** **23**	$r0 … $r7	For saved values
24 **25**	$t8 $t9	temporary (cont'd)
26 **27**	$v2 $v3	Functions results (cont'd)
28	$le	Used for length function
29	$sp	stack pointer
30	$fp	frame pointer
31	$ra	return address

ii. NASH instruction list:

a. Instruction Type: (R-type):



This format is vital for instructions that requires arithmetic operation such as add, sub, shift left arithmetic etc.

- OP: 6-bit opcode to determine the operation to be performed
- rs: source register (5-bit)
- rt: second source register (5-bit)
- rd: destination register (5-bit)
- shamt: shift amount, 5 bit is enough to shift a 32 bit data
- funct: function to be perfomed on the source registers (6-bit)

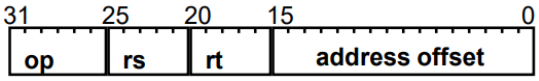
Category	Instruction	opcode	funct	Instruction with operands notation	**Discription**
Arithmetic (R format)	Addition (add)	000000	100000	add rd rs, rt	RF[rd] = RF[rs]+RF[rt]; PC<= PC+4
	Subtract (sub)	000000	100001	sub rd rs, rt	RF[rd] = RF[rs]-RF[rt] ; PC<= PC+4
	Shift Left Logical (sll)	000000	100111	sll rd,rt , shamt	RF[rd] = RF[rt] << shamt PC<= PC+4
	Shift Right Logical	000000	101000	srl rd,rt , shamt	RF[rd] = RF[rt] >> shamt PC<= PC+4
	Shift Right Arithmetic	000000	101001	sra rd,rt , shamt	RF[rd] = RF[rt] >> >shamt PC<= PC+4
	And	000000	100100	and rd rs, rt	RF[rd] = RF[rs] & RF[rt]; PC<= PC+4
	Or	000000	100101	or rd rs, rt	RF[rd] = RF[rs] or RF[rt]; PC<= PC+4
	Xor	000000	100110	xor rd rs, rt	RF[rd] = RF[rs] xor RF[rt]; PC<= PC+4
	Multiplication	000000	100010	Mul rd,rs,rt	RF[rd] = RF[rs] * RF[rt]; PC<= PC+4
	Division	000000	100011	Div rd,rs,rt	RF[rd] = RF[rs] / RF[rt]; PC<= PC+4
	Set On Less Than	000000	101010	Slt rd,rs,rt	if (R[rt]<RF[rs]) RF[rd]=1; else RF[rd]=0

Note that the size of RF[rd] is 32 bit, RF is Register File

V.ii.a.1 Why this R-type is important for the benchmark?

All three problems require core base operations, such as add, sub, mul… etc. Hence, we decided to include this type in the NASH ISA. There still a space to add more instructions in this type, but the current instruction is sufficient to perform the benchmark.

b. Immediate type (I-type):



This format is essential for multiple instructions. It can be used for arithmetic use, such as ADDI, conditional jumping, such as beq, or memory operations such as lh and sh.

- OP: 6-bit opcode to determine the operation to be performed
- rs: source register
- rt: second source register or destination register.
- Address offset (or I): a signed 16 bit integer

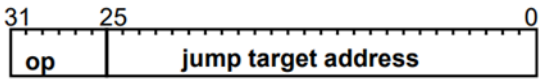
Category	Instruction	**Op** Code	Instruction with operands notation	**Discription**
Arithmetic (I format)	Add Immediate	001010	addi rt, rs, I	RF[rt] = RF[rs]+sign_extended(I) PC<= PC+4
	And Immediate	001011	andi rt, rs, I	RF[rt] = RF[rs]&sign_extended(I) PC<= PC+4
	Or Immediate	001100	ori rt, rs, I	RF[rt] = RF[rs] or sign_extended(I) PC<= PC+4
Cond. Jump (I format)	Set on Less Than Immediate	001001	Slti rt, rs, I	if (RF[rs]<I) RF[rt]=1; else RF[rt]=0 PC<= PC+4
	Jump if Equal	010010	Beq rt, rs, I	if (RF[rs]== RF[rt]) PC = address[I]; else PC<= PC+4
	Jump if Not Equal	010011	Bne rt, rs, I	if (RF[rs]!= RF[rt]) PC = address[I]; else PC<= PC+4
Data transfer (I format)	Load Word	000001	Lw rt, I(rs)	RF[rt] = memory (RF[rs]+I); PC<= PC+4
	Store Word	000010	Sw rt, I(rs)	memory (RF[rs]+I)= RF[rt] ; PC<= PC+4
	Load Half-Word	000011	Lh rt, I(rs)	RF[rt] = memory (RF[rs]+I)[15:0]; PC <= PC+4
	Store Half-Word	000100	Sh rt, I(rs)	memory (RF[rs]+I)= RF[rt] [15:0]; PC<= PC+4
Jump (I format)	Jump to a register value	010101	Jr rt	PC = RF[rt]

Note; conditional jumps here are using direct addressing

V.ii.b.1 Why this I-type is important for the benchmark?

Instructions in this type is being used with every loop in every benchmark. To perform a specific number of loops, we need instructions from this type. Also, loading into and storing from data memory can be done using this type. In addition, including Lh and Sh instruction is specifically aimed at the benchmark since most data loaded and stored is 16-bit.

c. Jump type (J type):



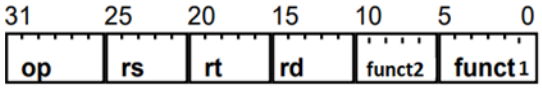
This instruction type is as simple as it seems. We will need two instruction in this type, unconditional jump, and calling a subroutine.

Category	Instruction	**Op Code**	Instruction with operands notation	**Discription**
Unconditional jump	Jump	010100	J target	PC<= target
Unconditional jump to subroutine	CALL	010110	Call target	RF[$ra] = PC+4 PC <= target

V.ii.c.1 Why this J-type is important for the benchmark?

Unconditional jumps are almost required with every code, having such an option improves the flexibility of the bench mark. Moreover, jumping into subroutine is also critical in our benchmark, especially in the MergeSort-CRC benchmark. Hence, adding a CALL instruction for the purpose of jumping to these subroutines and store the PC value of the next instruction to return to it when the subroutine is over.

d. R-CAL type:



This type is basically the same as R-type, however, we substituted the shamt field with a funct2 field for the purpose of controlling the second ALU

Category	Instruction	opcode	Instruction with operands notation	**Discription**	**Note**
Arithmetic (R-CAL format)	Acumlate multiplication	011000	MULA rd,rs,rt	RF[rd] = (RF[rs]*RF[rt])+RF[rd]; PC<= PC+4	
	Xor then Shift lift	011001	XORS rd,rs,rt	RF[rd] = RF[rd] xor RF[rs] ; RF[rt] = RF[rt] << 1 PC<= PC+4	
	Store and jump	010111	JSW rt,rd(rs), target	memory (RF[rs]+RF[Rd]) =RF[rt] ; PC<= target	(target address) is stored in funct1 and funct2 (11-bits)
	Store and Increment counter	011010	SWIN rt, I(rs), rd	memory (RF[rs]+I)= RF[rt] ;PC<=PC+4 ; RF[rd] = RF[rd]+4	(I) is stored in funct1 and funct2 (11-bits)

VI.Appendix B Datapath & Control Codes:

i.The final Datapath code:

```
/* NASH CPU -Datapath- :
This Code is Written With The Support of Prof. Ali Muhtaroglu
And Team Work of Abduallah Damash, and Abdelaziz Al-Najjar
For Implementing The Datapath of NASH CPU
EEE446 Spring 2021 Middle East Technical University - Northern Cyprus Campus */

`include "config.sv"
`include "constants.sv"
module datapath446(
                    input logic clock, reset, MemRead, MemWrite, MemToReg,
                  input logic ALUSrc, RegWrite, Branch, Jump, Alu2Mux1,Alu2Mux2,
                    input logic  RegDst,
                    input logic [3:0] ALUOp,
                    input logic [3:0] ALUOp2,
                    input logic [`DRAM_WORD_SIZE-1:0]  icache_data_out,  // data to CPU (cache->CPU)
                    input logic icache_data_ready,                                  // data to CPU ready (cache->CPU)
                    input logic[`DRAM_WORD_SIZE-1:0]dcache_data_out,       // data to CPU (cache->CPU)
                    input logic dcache_data_ready,                                  // data to CPU ready (cache->CPU)
                    input logic cache_miss_stall,
                    output logic [5:0] Op,
                    output logic [`DRAM_ADDRESS_SIZE-1:0] icache_address,
                    output logic icache_valid,
                    output logic [`DRAM_ADDRESS_SIZE-1:0]  dcache_address,  // cpu request address (CPU->cache)
                    output logic [`DRAM_WORD_SIZE-1:0]       dcache_data_in, // cpu request data (CPU->cache)
                  output logic [`DRAM_WORD_SIZE/8-1:0]   dcache_byte_en,  // cpu request byte enable (CPU->cache)
                  output logic dcache_rw,                                           // cpu R/W request (CPU->cache)
                  output logic dcache_valid                                         // cpu request valid (CPU->cache)
                    );

    //****** Initializing Internal Signals ******//
    //Program Counter Signals ***//
    parameter PCSTART = 0;
    logic [`DRAM_ADDRESS_SIZE-1:0] PC;
```

```
    //Instruction memory internal storage, input address and output data bus signals ***//
    //logic [7:0] instmem [127:0];
    logic [`DRAM_ADDRESS_SIZE-1:0] instmem_address;
    logic [31:0] instmem_data;
    logic [31:0] instmem_datav1;
    logic icache_v;

    //Data memory internal storage, input address and output data bus signals ***//
    //logic [7:0] datamem [127:0];
    logic [6:0] datamem_address;
    logic [31:0] datamem_data;
    logic dcache_write;
    logic dcache_v;
    logic [`DRAM_WORD_SIZE/8-1:0]   dcache_byte_enable;
    logic [`DRAM_WORD_SIZE-1:0]  dcache_datain;
    logic halfbit;

    //Hazard Unit Signals ***//
    logic StallSignal=1'b0;
    logic PCenable = 1'b1;
    logic IdIfenable = 1'b1;
    logic FlushSignal=1'b0;
    logic PCSrc;
    logic [1:0] Pcsel=2'b00;
    logic [1:0] ForwardA=2'b00;
    logic [1:0] ForwardB=2'b00;
    logic ForwardC=1'b0;
    logic ForwardD=1'b0;
    logic ForwardM= 1'b0;
    logic [31:0] FB;
    logic [31:0] FC;
    logic [31:0] FD;
    logic [31:0] FM;
```

```
    //Register File Signals ***//
    logic [31:0] RF[31:0];
    logic [31:0] da; //Read data 1 (Rt)
    logic [31:0] db; //Read data 2 (Rd)
    logic [31:0] dc; //Read dara 3 (Rs)
    logic [31:0] RF_WriteData; // write data

    //Arithmetic Logic Unit One(ALU1) Signals ***//
    logic [3:0]  ALUCtrl;
    logic [31:0] ALUResult;
    logic [31:0] OpA;
    logic [31:0] OpB;
    logic Zero;
    logic [5:0] Function;

    //Arithmetic Logic Unit Two(ALU2) Signals ***//
    logic [3:0]  ALUCtrl2;
    logic [31:0] ALUResult2;
    logic [31:0] OpA2;
    logic [31:0] OpB2;
    //logic Zero2;                  //In case, it is needed
    logic [4:0] Function2;
    logic [4:0]tempf;
    logic [3:0] tempf2;

    //Branch Instruction Signals ***//
    logic [31:0] address_offsett;
    logic [`DRAM_ADDRESS_SIZE-1:0] Branch_address; // After addition with PCincremented
    logic [31:0] ShiftedAdderess;
    logic comparator;

    //****** Initializing Internal Signals End ******//

    //************Start  of initialize instruction and data memory********//
    // Keeping Old Verstion (Without Memory)
    // initialize instruction and data memory arrays
    // this will read the .dat file in the same directory
    // and initialize the memory accordingly.
    //initial   $readmemh("instruction_memory.dat", instmem);
    //initial   $readmemh("data_memory.dat", datamem);

    //**************staging registers initializing***************//

    /*Explanation:*/
    // IF/ID Pipeline staging register fields can be represented using structure format of System Verilog
    // You may refer to the first field in the structure as IfId.instruction for example

    struct packed{
        logic [31:0] instruction;
        logic [`DRAM_ADDRESS_SIZE-1:0] PCincremented;
    } IfId;

    struct packed{
        logic [5:0]OPcode;
        logic [`DRAM_ADDRESS_SIZE-1:0] PCincremented; //coming from IfId
        logic [31:0] da; //read data 1
        logic [31:0] db; //read data 2
        logic [31:0] dc; //read data 3
        logic MemRead, MemWrite, MemToReg;
        logic ALUSrc, RegWrite, Branch, Jump,Alu2Mux1,Alu2Mux2;
        logic RegDst;
        logic [3:0] ALUOp;
        logic [3:0]ALUOp2;
        logic [31:0] address_offset; // for Branch_address
        logic [4:0] Rs;
        logic [4:0] Rt;
        logic [4:0] Rd;
    } IdEx;


    struct packed{
        logic [5:0]OPcode;
        logic [`DRAM_ADDRESS_SIZE-1:0] PCincremented;
        logic [31:0] AluOut;
        logic [31:0] FB; // for store instruction
        logic [31:0] dc; // for the second ALU
        logic [4:0] Rdst;// after RegDst
        logic Zero; // from ALU1 for branching
        logic MemRead, MemWrite, RegWrite, MemToReg, Branch, Jump, Alu2Mux1,Alu2Mux2;
        logic [3:0]ALUOp2;
        logic [4:0] Function2;
        logic [4:0] Rs;
        logic [4:0] Rt;
        logic [4:0] Rd;

    } ExMem;

    struct packed{
        logic [5:0]OPcode;
        logic [`DRAM_ADDRESS_SIZE-1:0] PCincremented;
        logic MemToReg,RegWrite;
        logic [31:0] AluOut2;
        logic [31:0] datamem_data;
        logic [4:0] Rdst;

    }MemWb;

    //*************end of registers initializing**************//

    //*********Fetch start********//

    //Instruction Memory Address
    //assign instmem_address = PC; //For Keeping Old Version (Without Memory)
    assign icache_address = PC;
    assign icache_valid = 1;
    assign instmem_datav1 = (icache_data_ready)? icache_data_out: instmem_datav1;
```

```
    // Instruction Memory Read Logic
    assign instmem_data[31:24] = instmem_datavl[7:0];
    assign instmem_data[23:16] = instmem_datavl[15:8];
    assign instmem_data[15:8] = instmem_datavl[23:16];
    assign instmem_data[7:0] = instmem_datavl[31:24];

    //**storing in IF/ID***//
        always @(posedge clock) begin
        if(!cache_miss_stall)begin
            if (IdIfenable)begin
            IfId.instruction      <=  (FlushSignal)?  32'd0:instmem_data;
            IfId.PCincremented[`DRAM_ADDRESS_SIZE-1:0] <= PC[`DRAM_ADDRESS_SIZE-1:0]+4;
            end
            end
        end

    //**** Fetch Stage end ****//

    //**Decode Stage start***//
        assign Op = IfId.instruction[31:26];
        assign da = (IfId.instruction[25:21]!=0) ? RF[IfId.instruction[25:21]] : 0; //Rs
        assign db = (IfId.instruction[20:16]!=0) ? RF[IfId.instruction[20:16]] : 0; //Rt
        assign dc = (IfId.instruction[15:11]!=0) ? RF[IfId.instruction[15:11]] : 0; //Rd for Alu2Mux1 instructions

        //Computing Branch Target Address
        always_comb begin
            if (Op == 6'b010011) begin // BNE Instruction
                    PCSrc = Branch & !comparator;
                end
            else begin // BEQ Instruction
                    PCSrc = Branch & comparator;
                    end
        end

        assign Branch_address = (cache_miss_stall==0)? IfId.instruction[14:0] : Branch_address;
        assign FC = (cache_miss_stall==0)? (ForwardC)? ExMem.AluOut : da : FC;
        assign FD = (cache_miss_stall==0)? (ForwardD)? ExMem.AluOut : db : FD;
        assign comparator = (cache_miss_stall==0)? (FC == FD): comparator;
```

```

        always @(posedge clock)begin
        if(!cache_miss_stall)begin
            IdEx.OPcode <= IfId.instruction[31:26];
            IdEx.PCincremented <= IfId.PCincremented;
            IdEx.da <= da;
            IdEx.db <= db;
            IdEx.dc <= dc;
            IdEx.MemRead <= (StallSignal)? 1'b0:MemRead;
            IdEx.MemWrite <=    (StallSignal)? 1'b0:MemWrite;
            IdEx.MemToReg <=    (StallSignal)? 1'b0:MemToReg;
            IdEx.ALUSrc <=  (StallSignal)? 1'b0:ALUSrc;
            IdEx.RegWrite <=    (StallSignal)? 1'b0:RegWrite;
            IdEx.RegDst <=  (StallSignal)? 1'b0:RegDst;
            IdEx.Branch <=  (StallSignal)? 1'b0:Branch;
            IdEx.Jump <=        (StallSignal)? 1'b0:Jump;
            IdEx.ALUOp <=       (StallSignal)? 4'b0000:ALUOp;
            IdEx.ALUOp2 <=  (StallSignal)? 4'b0000:ALUOp2;
            IdEx.Alu2Mux1 <=        (StallSignal)? 1'b0:Alu2Mux1;
            IdEx.Alu2Mux2 <=        (StallSignal)? 1'b0:Alu2Mux2;
            IdEx.Rs <=            IfId.instruction[25:21];
            IdEx.Rt <=            IfId.instruction[20:16];
            IdEx.Rd <=            IfId.instruction[15:11];
            IdEx.address_offset <= {{16{IfId.instruction[15]}},IfId.instruction[15:0]}; // sign extended
            end
        end

    //**************Hazard Unit******************//
```

```
    //***Data Hazard Unit (Forwarding Unit)***//
    always_comb begin
      // Ex hazard:
        ForwardA = 2'b00; //default
        ForwardB = 2'b00; //default
        if(cache_miss_stall==0)begin
        if (ExMem.RegWrite)begin
            if(ExMem.Rdst != 0)begin
                if(ExMem.Rdst == IdEx.Rs)begin
                    ForwardA = 2'b10;
                end
            end
        end

        if (ExMem.RegWrite)begin
            if(ExMem.Rdst != 0)begin
                if(ExMem.Rdst == IdEx.Rt)begin
                    ForwardB = 2'b10;
                end
            end
        end

        // Mem hazard with corrected priority:
        if (MemWb.RegWrite)begin
            if(MemWb.Rdst != 0)begin
                if (ForwardA != 2'b10)begin
                    if(MemWb.Rdst == IdEx.Rs)begin
                        ForwardA = 2'b01;
                    end
                end
            end
        end
```

```
275         if (MemWb.RegWrite)begin
276             if(MemWb.Rdst != 0)begin
277                 if (ForwardB != 2'b10)begin
278                     if(MemWb.Rdst == IdEx.Rt)begin
279                         ForwardB = 2'b01;
280                     end
281                 end
282             end
283         end
284         end
285     end
286
287     // Forwarding data3 to ALU2
288     always_comb begin
289        ForwardM = 1'b0;
290     if(cache_miss_stall==0)begin
291         if(MemWb.Rdst != 0)begin
292                 if(MemWb.Rdst == IdEx.Rd)begin
293                     ForwardM = 1'b1;
294                 end
295             end
296     end
297     end
298
299     // Forwarding to branch
300     always_comb begin
301                ForwardC = 1'b0;
302                 ForwardD = 1'b0;
303     if(cache_miss_stall==0)begin
304     if (Branch)begin //checking
305             if(ExMem.Rdst != 0)begin
306                 if(ExMem.Rdst == IfId.instruction[25:21])begin
307                     ForwardC = 1'b1;
308                 end
309             end
310         end
```

```
311         if (Branch)begin //checking
312             if(ExMem.Rdst != 0)begin
313                 if(ExMem.Rdst == IfId.instruction[20:16])begin
314                     ForwardD = 1'b1;
315                 end
316             end
317         end
318         end
319     end
320
321     //*******Forwarding Unit (Data Hazard Unit) End//
322
323     //***Control Hazard Unit***//
324     always_comb begin
325              FlushSignal  = 1'b0;
326                 Pcsel        = 2'b00;
327         if(cache_miss_stall==0)begin
328         if (Jump) begin
329                 FlushSignal  =  1'b1;
330             if (Op == 6'b010101)begin
331                 Pcsel        = 2'b11;
332             end
333             else begin
334                 Pcsel        = 2'b10;
335                 end
336             end
337
338         if (PCSrc) begin
339                 FlushSignal  =  1'b1;
340                 Pcsel        = 2'b01;
341         end
342         end
343     end
344     //*******Control Hazard Unit End//
```

```
345     //***Load Use Hazard Unit***//
346     always_comb begin
347                 StallSignal =  1'b0;
348                 PCenable     =  1'b1;
349                 IdIfenable   =  1'b1;
350     if(cache_miss_stall==0)begin
351          if(IdEx.MemRead)begin
352             if(IdEx.Rt==IfId.instruction[25:21])begin
353                 StallSignal =  1'b1;
354                 PCenable     =  1'b0;
355                 IdIfenable   =  1'b0;
356             end
357             end
358           if(IdEx.MemRead)begin
359             if(IdEx.Rt==IfId.instruction[20:16])begin
360                 StallSignal =  1'b1;
361                 PCenable     =  1'b0;
362                 IdIfenable   =  1'b0;
363             end
364             end
365         end
366     end
367     //*******Load Use Hazard Unit End//
368     //**************End of Hazard Unit******************//
369     //**Decode End***//
370
```

```
    //******Ex begin**********//
    always_comb begin
    case (ForwardA)
        2'b00 : OpA = IdEx.da;
        2'b01 : OpA = RF_WriteData;
        2'b10 : OpA = ExMem.AluOut;
    endcase
    case (ForwardB)
        2'b00 : FB = IdEx.db;
        2'b01 : FB = RF_WriteData;
        2'b10 : FB = ExMem.AluOut;
    endcase
    if (IdEx.OPcode == 6'b010111) begin
            OpB = {{24{IdEx.address_offset[15]}},IdEx.address_offset[15:8]}; // for jump and store
        end
        else begin
        case (IdEx.ALUSrc)
        1'b1 : OpB = IdEx.address_offset;
        1'b0 : OpB = FB;
        endcase
        end
    end
    //****ALU1 Logic****//
    always_comb begin
     Function = IdEx.address_offset[5:0];
    end

    always@(Function or ALUOp) begin
        // AlU Operations besad on Oprend and ALUop from contol unit
        casez({IdEx.ALUOp,Function})
        10'b0000_??????: ALUCtrl= 4'b0000; //ADD operation Load (lw) or Store (sw)
        10'b0001_??????: ALUCtrl= 4'b0001; //SUB operation for Branch (beq)
        10'b0010_??0000: ALUCtrl= 4'b0000; //Add (Add)
        10'b0010_??0001: ALUCtrl= 4'b0001; //Subtract (Sub)
        10'b0010_??0010: ALUCtrl= 4'b0010; //Multiply (Mul)
        10'b0010_??0011: ALUCtrl= 4'b0011; //Divide (Div)
        10'b0010_??0100: ALUCtrl= 4'b0100; //And (And)
        10'b0010_??0101: ALUCtrl= 4'b0101; //Or (Or)
        10'b0010_??0110: ALUCtrl= 4'b0110; //Xor (Xor)
        10'b0010_??0111: ALUCtrl= 4'b0111; //Shift Left Logical (Sll)
        10'b0010_??1000: ALUCtrl= 4'b1000; //Shift Right Logical (Srl)
        10'b0010_??1001: ALUCtrl= 4'b1001; //Shift Right Arithmetic (Sra)
        10'b0010_??1010: ALUCtrl= 4'b1010; //Set on Less Than (Slt)
        10'b0011_??????: ALUCtrl= 4'b0010; //MUL operation
        10'b0100_??????: ALUCtrl= 4'b0100; //And operation
        10'b0101_??????: ALUCtrl= 4'b0101; //OR operation
        10'b0110_??????: ALUCtrl= 4'b1010; //Slt operation
        10'b0111_??????: ALUCtrl= 4'b0110; //Xor operation
        10'b1000_??????: ALUCtrl= 4'b0111; //Shift lift operation
        10'b1001_??????: ALUCtrl= 4'b1000; //Shift Right operation
        default:      ALUCtrl= 4'b0000;
        endcase
    end

    always_comb
    begin
        case(ALUCtrl)
        4'b0000: ALUResult = OpA + OpB;         //Add (Add)
        4'b0001: ALUResult = OpA - OpB;         //Subtract (Sub)
        4'b0010: ALUResult = OpA * OpB;         //Multiply (Mul)
        //4'b0011: ALUResult = OpA / OpB;           //Divide (Div)
        4'b0100: ALUResult = OpA & OpB;         //And (And)
        4'b0101: ALUResult = OpA | OpB;         //Or (Or)
        4'b0110: ALUResult = OpA ^ OpB;         //Xor (Xor)
        4'b0111: ALUResult = OpA << 1;          //Shift Left Logical (Sll)
        4'b1000: ALUResult = OpA >> 1;          //Shift Right Logical (Srl)
        4'b1001: ALUResult = OpA >>> 1;         //Shift Right Arithmetic (Sra)
        4'b1010: ALUResult =  OpA < OpB?1:0;   //Set on Less Than (Slt)
        default: ALUResult = OpA + OpB;
        endcase
        Zero = (ALUResult==0); //Zero == 1 when ALUResult is 0 (for branch)

    end

    //****ALU1 END *****//

    always @(posedge clock) begin
        if(!cache_miss_stall)begin
            ExMem.OPcode <= IdEx.OPcode;
            ExMem.PCincremented <= IdEx.PCincremented;
            ExMem.Rs <= IdEx.Rs;
            ExMem.Rt <= IdEx.Rt;
            ExMem.Rd <= IdEx.Rd;
            ExMem.FB <= FB ;
            ExMem.dc <= IdEx.dc ;
            ExMem.Function2 <= IdEx.address_offset[10:6];
            ExMem.AluOut <= ALUResult;
            case(IdEx.RegDst)
            1'b0: ExMem.Rdst <= IdEx.Rt;
            1'b1: ExMem.Rdst <= IdEx.Rd;
            endcase
```

```
            ExMem.Zero <= Zero;
            ExMem.Alu2Mux1 <= IdEx.Alu2Mux1;
            ExMem.Alu2Mux2 <= IdEx.Alu2Mux2;
            ExMem.ALUOp2 <= IdEx.ALUOp2;
            ExMem.MemRead <= IdEx.MemRead;
            ExMem.MemWrite <= IdEx.MemWrite;
            ExMem.RegWrite <= IdEx.RegWrite;
            ExMem.MemToReg <= IdEx.MemToReg;
            ExMem.Jump <= IdEx.Jump;
            end
        end
    //******EX End***//

    //*********** MEM start********//
        always@(posedge clock)begin
        FM = (ForwardM)? datamem_data : ExMem.dc ;
        end
    always_comb begin
        //OpA2 = ExMem.AluOut;
        case (ExMem.Alu2Mux1)
        1'b1 : OpB2 = FM;
        1'b0 : OpB2 = 32'b0;
        endcase
        case (ExMem.Alu2Mux2)
        1'b1 : OpA2 = 32'bxx0100;
        1'b0 : OpA2 = ExMem.AluOut;
        endcase
    end

    //****ALU2 Logic****//
    assign tempf = ExMem.Function2;
    assign tempf2 = ExMem.ALUOp2;
    always@(tempf or tempf2) begin
        // AlU Operations besad on Oprend and ALUop2 from contol unit

        casez({ExMem.ALUOp2,ExMem.Function2})
        9'b0000_?????: ALUCtrl2= 4'b0000; //ADD operation Load (lw) or Store (sw)
        9'b0001_?????: ALUCtrl2= 4'b0001; //SUB operation for Branch (beq)
        9'b0010_?0000: ALUCtrl2= 4'b0000; //Add (Add)
        9'b0010_?0001: ALUCtrl2= 4'b0001; //Subtract (Sub)
        9'b0010_?0010: ALUCtrl2= 4'b0010; //Multiply (Mul)
        9'b0010_?0011: ALUCtrl2= 4'b0011; //Divide (Div)
        9'b0010_?0100: ALUCtrl2= 4'b0100; //And (And)
        9'b0010_?0101: ALUCtrl2= 4'b0101; //Or (Or)
        9'b0010_?0110: ALUCtrl2= 4'b0110; //Xor (Xor)
        9'b0010_?0111: ALUCtrl2= 4'b0111; //Shift Left Logical (Sll)
        9'b0010_?1000: ALUCtrl2= 4'b1000; //Shift Right Logical (Srl)
        9'b0010_?1001: ALUCtrl2= 4'b1001; //Shift Right Arithmetic (Sra)
        9'b0010_?1010: ALUCtrl2= 4'b1010; //Set on Less Than (Slt)
        9'b0011_?????: ALUCtrl2= 4'b0010; //MUL operation
        9'b0100_?????: ALUCtrl2= 4'b0100; //And operation
        9'b0101_?????: ALUCtrl2= 4'b0101; //OR operation
        9'b0110_?????: ALUCtrl2= 4'b1010; //Slt operation
        9'b0111_?????: ALUCtrl2= 4'b0110; //Xor operation
        9'b1000_?????: ALUCtrl2= 4'b0111; //Shift lift operation
        9'b1001_?????: ALUCtrl2= 4'b1000; //Shift Right operation
        default:      ALUCtrl2= 4'b0000;
        endcase
    end

    always_comb
    begin
        case(ALUCtrl2)
        4'b0000: ALUResult2 = OpA2 + OpB2;          //Add (Add)
        4'b0001: ALUResult2 = OpA2 - OpB2;          //Subtract (Sub)
        4'b0010: ALUResult2 = OpA2 * OpB2;          //Multiply (Mul)
        //4'b0011: ALUResult2 = OpA2 / OpB2;            //Divide (Div)
        4'b0100: ALUResult2 = OpA2 & OpB2;          //And (And)
        4'b0101: ALUResult2 = OpA2 | OpB2;          //Or (Or)
        4'b0110: ALUResult2 = OpA2 ^ OpB2;          //Xor (Xor)
        4'b0111: ALUResult2 = OpA2 << 1;            //Shift Left Logical (Sll)
        4'b1000: ALUResult2 = OpA2 >> 1;            //Shift Right Logical (Srl)
        4'b1001: ALUResult2 = OpA2 >>> 1;           //Shift Right Arithmetic (Sra)
```

```
        4'b1010: ALUResult2 =  OpA2 < OpB2?1:0;      //Set on Less Than (Slt)
        default: ALUResult2 = OpA2 + OpB2;
        endcase
        //Zero2 = (ALUResult2==0); //add if needed

    end
    //****ALU2 END *****//

    //assign datamem_address = ExMem.AluOut[6:0];
    assign halfbit = (ExMem.OPcode == 6'b000100)? 1'b1: 1'b0;
    assign dcache_address = (ExMem.MemRead)? {ExMem.AluOut+512} : ExMem.AluOut+512 ;

    // Data Memory Write Logic
    assign dcache_rw = (!ExMem.MemRead)? ExMem.MemWrite: 32'b0;
    assign dcache_valid = ExMem.MemWrite || ExMem.MemRead;
    assign dcache_byte_en = dcache_byte_enable;
    assign dcache_data_in = dcache_datain;
    assign dcache_datain = (ExMem.MemWrite)? ExMem.FB : dcache_datain ;
    assign dcache_byte_enable = (halfbit)? 4'b0011 : 4'b1111;

    // Data Memory Read Logic
    assign datamem_data = (ExMem.MemRead)? (dcache_data_ready)? dcache_data_out : datamem_data : datamem_data;

    //Load into MemWb
    always@(posedge clock) begin
    if(!cache_miss_stall)begin
        MemWb.OPcode <= ExMem.OPcode;
        MemWb.PCincremented <= ExMem.PCincremented;
        MemWb.Rdst <= ExMem.Rdst;
        MemWb.datamem_data <= datamem_data;
        MemWb.RegWrite <= ExMem.RegWrite;
        MemWb.MemToReg <= ExMem.MemToReg;
        MemWb.AluOut2 <= ALUResult2;
        end
    end
    //*****MEM End***//
```

```
    //********WB Start******//
    assign RF_WriteData = (MemWb.MemToReg == 0)?  MemWb.AluOut2 : (MemWb.OPcode == 6'b000011)?
    {{16{MemWb.datamem_data[15]}},MemWb.datamem_data[15:0]} : MemWb.datamem_data ; // Write Back for R-Type or Load
    always@(negedge clock) begin
    if (MemWb.RegWrite)begin
        if(MemWb.OPcode == 6'b010110)begin // for Call instruction
            RF[31]= MemWb.PCincremented; // store next instruction position in ra
        end
        else begin
         RF [MemWb.Rdst] =RF_WriteData;
        end
    end
    end

    //***WB End***//

    //******* PC Logic******//
    always@ (posedge clock) begin
        if(reset)
            PC <= PCSTART;
        else
        if(!cache_miss_stall)begin
        if (PCenable) begin
            case (Pcsel)
            2'b00: PC <= PC+4;                  //Increamnt PC by four
            2'b10: PC[`DRAM_ADDRESS_SIZE-1:0] <= (Op==6'b010111)? IfId.instruction[6:0]: IfId.instruction[`DRAM_ADDRESS_SIZE-1:0]
            // for jump instructions
            2'b01: PC[`DRAM_ADDRESS_SIZE-1:0] <= Branch_address;        // for branch
            2'b11: PC[`DRAM_ADDRESS_SIZE-1:0] <= RF[IfId.instruction[20:16]];               // for jr
            default: PC[`DRAM_ADDRESS_SIZE-1:0] <= PC[`DRAM_ADDRESS_SIZE-1:0]+4;
        endcase
        end
        end
    end
    //*******PC End ****//
endmodule
```

Figure 8. Final Datapath Code.

ii. Control Unit Code:

```
/* NASH CPU -Control Unit- :
This Code is Written With The Support of Prof. Ali Muhtaroglu
And Team Work of Abduallah Damash, and Abdelaziz Al-Najjar
For Implementing The Control Unit of NASH CPU
EEE446 Spring 2021 Middle East Technical University - Northern Cyprus Campus */

module control446 (
                   input logic [5:0] Op,
                   output logic MemRead, MemWrite, MemToReg, ALUSrc,
                   output logic RegWrite, Branch, Jump,Alu2Mux1,Alu2Mux2,
                   output logic RegDst,
                   output logic [3:0] ALUOp,
                   output logic [3:0] ALUOp2);

    always_comb
    begin
                      MemRead = 1'b0;
                      MemWrite = 1'b0;
                  MemToReg = 1'b0;
                  ALUSrc = 1'b0;
                  RegWrite = 1'b0;
                  RegDst = 1'b0;
                  Branch = 1'b0;
                  Jump = 1'b0;
                      Alu2Mux1 = 1'b0; //{0,1} --> {0, Rd}
                      Alu2Mux2 = 1'b0; //{0,1} --> {Alul result,(inc Pc)}
                  ALUOp = 4'b0000;
                      ALUOp2 = 4'b0000;

        if (Op==6'b000000) begin        // R-type
                  RegWrite = 1'b1;
                  RegDst = 1'b1;
                  ALUOp = 4'b0010;
                     end
        else if (Op==6'b000001) begin // Load Word (Lw)
                      MemRead = 1'b1;
                  MemToReg = 1'b1;
                  ALUSrc = 1'b1;
```

```
                  RegWrite = 1'b1;
                     end
        else if (Op==6'b000010) begin // Store Word (Sw)
                      MemWrite = 1'b1;
                  ALUSrc = 1'b1;
                     end
        else if (Op==6'b000011) begin // Load Half (Lh)
                      MemRead = 1'b1;
                  MemToReg = 1'b1;
                  ALUSrc = 1'b1;
                  RegWrite = 1'b1;
                     end
        else if (Op==6'b000100) begin // Store Half (Sh)
                      MemWrite = 1'b1;
                  ALUSrc = 1'b1;
                     end
        else if (Op==6'b001001) begin // Set on Less Than Immediate (Slti)
                  RegWrite = 1'b1;
                  RegDst = 1'b1;
                  ALUOp = 4'b0001;
                     end
        else if (Op==6'b001010) begin // Add Immediate (Addi)
                     RegWrite = 1'b1;
                     ALUSrc = 1'b1;
                     end
        else if (Op==6'b001011) begin // And Immediate (Andi)
                 RegWrite = 1'b1;
                     ALUSrc = 1'b1;
                 ALUOp = 4'b0100;
                     end
        else if (Op==6'b001100) begin // Or Immediate (Ori)
                 RegWrite = 1'b1;
                     ALUSrc = 1'b1;
                 ALUOp = 4'b0101;
                     end
```

```
74          else if (Op==6'b010000) begin // Load Upper Immediate (Lui)
75                RegWrite = 1'b1;
76                    ALUSrc = 1'b1;
77                    end
78          else if (Op==6'b010010) begin // Br On Equal (Beq)
79                    Branch = 1'b1;
80                    end
81          else if (Op==6'b010011) begin // Br On Not Equal (Bne)
82                Branch = 1'b1;
83                    ALUOp = 4'b0001;
84                    end
85          else if (Op==6'b010100) begin // Jump (J)
86                    Jump = 1'b1;
87                    end
88          else if (Op==6'b010101) begin // Jump register. (Jr)
89                Jump = 1'b1;
90                    RegWrite = 1'b1;
91                    end
92          else if (Op==6'b010110) begin // Call (Call)
93                RegWrite = 1'b1;
94                    Jump = 1'b1;
95                    end
96          else if (Op==6'b010111) begin // Jump and Store (Jsw)
97                Jump = 1'b1;
98                    MemWrite = 1'b1;
99                ALUSrc = 1'b1;
100                   end
101         else if (Op==6'b011000) begin // Multiply then Add (Mula)
102               RegWrite = 1'b1;
103               RegDst = 1'b1;
104                  Alu2Mux1 = 1'b1;
105               ALUOp = 4'b0010;
106                   ALUOp2 = 4'b0010;
107                   end
108         else if (Op==6'b011001) begin // Xor then Shift lift (XoSL)
109               RegWrite = 1'b1;
110               RegDst = 1'b1;
111                  Alu2Mux1 = 1'b1;
112               ALUOp = 4'b0010;
113                   ALUOp2 = 4'b0010;
114                   end
115         else if (Op==6'b011010) begin // Store and Increment (Swin)
116               RegWrite = 1'b1;
117               RegDst = 1'b1;
118                  Alu2Mux1 = 1'b1;
119                   Alu2Mux2 = 1'b1;
120                   ALUOp2 = 4'b0010;
121                   MemWrite = 1'b1;
122               ALUSrc = 1'b1;
123                   end
124
125     end
126 endmodule
127
```

Figure 9. Control Unit Code.

VII. Appendix C Assembly of Full Benchmarks:

i. Daxpy Assembled code using NASH ISA:

Table 10. Daxpy Assembly.

PC	Machine language	CODE	comment
0	28 10 00 03	addi $s0, $zero, 3	S0 = n = 3
4	28 11 04 00	addi $s1, $zero, 1024	S1 = k = 1024// for checking
8	28 13 00 00	addi $s3, $zero ,0	$s3 = 0 = counter = start
C	02 10 80 22	mul $s0, $s0, $s0	s0 = x = n*n
10	02 10 80 27	Sll $s0, $s0, 1	
14	02 10 80 27	Sll $s0, $s0, 1	s0 = x*4
18	02 60 a8 20	loading: add $s5, $zero, $s3	s5 = s3 = counter //
1C	06 b6 00 00	lw $s6, 0($s5)	S6 = A
20	02 15 a8 20	add $s5, $s5, $s0	s5 = s3 + s0 = start + counter+ x*4
24	06 b7 FF FC	lw $s7, -4($s5)	S7 = B
28	2a 73 00 04	addi $s3, $ s3 , 4	Counter++ //+4
2C	62 d1 b8 02	mula $s7, $s6, $s1	$s7= B = $s6*$s1+$s7
30	4a 70 00 38	beq $s3 , $s0, ext	counter =? x pc = ext
34	5e b7 FC 18	jsw $s7, -4($s5), loading	Mem[s5] = B ;; pc = loading
38	0a b7 FF FC	ext: sw $s7, -4($s5)	
3C	50 00 00 3C	exit: j exit	

ii. MergeSort + Merge + Length Codes Using NASH:

Table 11. Merge Sort Assembly.

PC	Machine language	Opcode (hex)	CODE	comment
0	00 00 00 00		nop	
4	2b bd 01 40		Addi $sp, $sp, 320	Stack pointer starts from 320
8	28 07 00 00		addi $a3, $zero, 0	Array starts from 0
C	58 00 00 14		Call MergeSort	
10	32 f7 00 64		Terminator	
14	2b bd ff f0	0A	MergeSort: Addi $sp, $sp, -16	Adgust stack for 5 items
18	0b a7 00 00	02	Sw $a3, 0($sp)	Save argument
1C	0b bf 00 04	02	Sw $ra, 4($sp)	Save return address
20	0b b7 00 08	02	Sw $s7, 8($sp)	Save s7 (right pointer)
24	0b b6 00 0c	02	Sw $s6, 12($sp)	Save s6 (left pointer)
28	28 fc 00 00	0A	Addi $le, $a3, 0	length(input)
2C	58 00 00 e4	16	Call length	
30	07 bf 00 04	01	Lw $ra, 4($sp)	Restore return address
34	28 52 00 00	0A	Addi $s2, $v0, 0	s2 = length (input) = n
38	26 49 00 02	09	Slti $t1, $s2, 2	test for n < 2
3C	00 00 00 00		nop	
40	00 00 00 00		nop	
44	49 20 00 50	12	Beq $t1, $zero, more	if no, go divide
48	2b bd 00 10	0A	Addi $sp, $sp, 16	pop 4 items from stack
4C	54 1f 00 00	15	Jr $ra	Return
50	02 41 98 28	00	More: Srl $s3, $s2, 1	$s3 = Mid = n/2
54	28 0c 00 00	0A	Addi t4, $zero, 0	t4 = counter = 0
58	28 eb 00 00	0A	Addi $t3, $a3, 0	t3 = first location of A = a3 = i
5c	4e 80 00 6c		Bne $s4, $zero, L1	
60	02 47 a0 20	00	Add $s4, $a3, $s2	Left is Array of size Mid, starts from $s4 = a3+n = j
64	02 89 a0 27	00	Sll $s4 , $s4, 1	
68	02 89 a0 27	00	Sll $s4 , $s4, 1	j*4
6c	2a 94 00 04	0A	L1: Addi $s4, $s4, 4	j++
70	2a 96 00 00	0A	Addi $s6, $s4, 0	$s6: left starts from here
74	49 93 00 94	01	Left: Beq $t4, $s3, Right	counter =? mid
78	49 92 00 a0		R1: Beq $t4, $s2, Leave	counter =? n
7C	05 6d 00 00	12	Lw $t5, 0($t3)	t5 = A[i]
80	0a 8d 00 00	02	Sw $t5, 0($s4)	left[j] = A[i] =t5
84	29 8c 00 01	0A	Addi $t4, $t4, 1	counter++
88	29 6b 00 04	0A	Addi $t3, $t3, 4	i++
8C	2a 94 00 04	0A	Addi $s4, $s4, 4	j++
90	50 00 00 74	14	J Left	
94	2a 94 00 04		Right: addi $s4, $s4, 4	
98	2a 97 00 00		Addi $s7, $s4, 0	$s7: right start from here
9C	50 00 00 78		J R1	
A0	2a c7 00 00		Leave: Addi $a3, $s6, 0	Left starts from s6
A4	58 00 00 14	16	Call MergeSort	MergeSort(left)

A8	07 a7 00 00	01	Lw $a3, 0($sp)	Restore original array address
AC	07 bf 00 04	01	Lw $ra, 4($sp)	Restore return address
B0	07 b7 00 08	01	Lw $s7, 8($sp)	Restore s7
B4	07 b6 00 0c	01	Lw $s6, 12($sp)	Restore s6
B8	2b bd ff f0	0A	Addi $sp, $sp, -16	Adgust stack for 5 items
BC	2a e7 00 00	0A	Addi $a3, $s7, 0	Right starts from s7
C0	58 00 00 14	16	Call MergeSort	MergeSort(Right)
C4	07 a7 00 00	01	Here: Lw $a3, 0($sp)	Restore original array address
C8	07 bf 00 04	01	Lw $ra, 4($sp)	Restore return address
CC	07 b7 00 08	01	Lw $s7, 8($sp)	Restore s7
D0	07 b6 00 0c	01	Lw $s6, 12($sp)	Restore s6
D4	02 47 40 20	00	Addi $a0, $s6, 0	Merge (a0 = left,
D8	00 17 28 20	00	Addi $a1, $s7, 0	a1 = right,
DC	07 a6 00 10	0A	lw $a2, 16($sp)	a2 = prev a3)
E0	50 00 01 04		J Merge	

E4	28 02 00 00	0A	Length: addi $v0, $zero, 0	
E8	07 88 00 00	01	start: lw $t0, 0($le)	X = t1= mem[t0]
EC	2b 9c 00 04	0A	addi $le, $le,4	Increase the address
F0	00 00 00 00	00	nop	
F4	49 00 0*1 00*	12	beq $t0, $zero, return	Is it null? Yes? return
F8	28 42 00 01	0A	addi $v0, $v0, 1	Length++
FC	50 00 00 *e8*	14	J start	
100	54 1f 00 00	15	return: jr $ra	

104	28 9c 00 00		Merge: addi $le, $a0, 0	length(left)
108	58 00 00 e4	16	Call length	
10C	07 bf 00 04	01	Lw $ra, 4($sp)	
110	00 49 80 27		sll $s0, $v0, 1	$s0 = length(left) =nL
114	02 10 80 27		sll $s0, $s0, 1	nL*4
118	00 90 80 20		add $s0, $s0, $a0	nL*4 +left
11C	28 bc 00 00		addi $le, $a1, 0	length(right)
120	58 00 00 e4		Call length	
124	07 bf 00 04	01	Lw $ra, 4($sp)	Restore return address
128	00 49 88 27		sll $s1, $v0, 1	$s1 = length (right) =nR
12C	02 30 88 27		sll $s1, $s1, 1	nR *4
130	00 b1 88 20		add $s1, $s1, $a1	S1 =nR*4+right
134	28 88 00 00		addi $t0, $a0,0	$t0 = i = 0
138	28 a9 00 00		addi $t1, $a1,0	$t1 = j = 0
13C	28 ca 00 00		addi $t2, $a2,0	$t2 = k = 0
140	01 10 58 2a		cmpboth: Slt $t3, $t0, $s0	i<nL ? $t3 = 1 : $t3 = 0
144	01 31 60 2a		Slt $t4, $t1, $s1	j<nR ? $t4 = 1 : $t4 = 0
148	01 8b 78 24		And $t7, $t3, $t4	i<nL && j < nR? $t7 = 1
14C	05 0d 00 00		Lw $t5, 0($t0)	$t5 = left(i)
150	05 2e 00 00		Lw $t6, 0($t1)	$t6 = right(j)
154	49 e0 01 88		Beq $t7, $zero, chkL	
158	01 ae 78 2a		Slt $t7, $t5, $t6	left(i) < ?right(j)

15c	00 00 00 00		nop	
160	00 00 00 00		nop	
164	49 e0 01 78		Beq $t7, $zero, right	no? jump to right
168	09 4d 00 00		left: Sw $t5, 0($t2)	A[k] = left [i]
16c	29 08 00 04		Addi $t0, $t0, 4	i++
170	29 4a 00 04		Addi $t2, $t2, 4	k++
174	50 00 01 40		J cmpboth	
178	09 4e 00 00		right: Sw $t6, 0($t2)	A[k] = right(j)
17C	29 29 00 04		Addi $t1, $t1, 4	j++
180	29 4a 00 04		Addi $t2, $t2, 4	k++
184	50 00 01 40		J cmpboth	
188	49 60 01 9c		chkL: beq $t3, $zero, chkR	i<nL? branch if no
18C	09 4d 00 00		Sw $t5, 0($t2)	A[k] = left [i]
190	29 08 00 04		Addi $t0, $t0, 4	i++
194	29 4a 00 04		Addi $t2, $t2, 4	k++
198	50 00 01 40		J cmpboth	
19C	49 80 01 a0		chkR: beq $t4, $zero, return	
1A0	09 4e 00 00		Sw $t6, 0($t2)	A[k] = right(j)
1A4	29 29 00 04		Addi $t1, $t1, 4	j++
1A8	29 4a 00 04		Addi $t2, $t2, 4	k++
1AC	50 00 01 40		J cmpboth	
1B0	2b bd 00 10		return: addi $sp, $sp, 16	Pop 4 items from stack
1B4	50 00 00 c4		J here	

iii. CRC Assembly Code:

Table 12. CRC Assembly.

PC	ML	OpCo	CODE	comment
00	07 31 00 50	01	lw $s1, 80 ($a4)	Load good CRC = 0xffff
04	29 0a 00 00	0A	Append: addi $t2, $t0, 0	Copy the previous input
08	07 29 00 00	01	lw $t1, 0 ($a4)	Load the input
0C	07 28 00 28	01	lw $t0, 40 ($a4)	Load 0x00ff
10	28 12 00 80	0A	addi $s2, $zero, 0x0080	(V) for Checking Most Significant Bit
14	28 13 00 10	0A	addi $s3, $zero, 16	(j) Counter checking the bits
18	49 20 00 a0	12	beq $t1, $zero, Exit	Exit
1C	28 0c 00 08	0A	UpCRC: addi $t4, $zero, 8	Next Address
20	28 0b 00 80	0A	addi $t3, $zero, 0x80	Checking most bit of CRC
24	28 0d 10 21	0A	addi $t5, $zero, 0x1021	(Poly) CRC16-TCITT
28	07 30 00 0f	01	lw $s0, 60 ($a4)	For Mask CRC (0x00ff)
2C	4d 80 00 40	13	Repeat: Bnq $t4, $zero, ChMos	If Null, jump to Augment
30	50 00 00 78	14	j Augment	If not, go find CRC
34	09 19 00 80	02	Done: Sw $t0, 128($a4)	Store the CRC
38	2b 39 00 04	0A	addi $a4, $a4, 4	Go to Next Address
3C	50 00 00 04	14	J Append	Start Over
40	01 2b 78 24	00	ChMos: and $t7, $t3, $t1	Ch & V
44	02 28 70 24	00	And $t6, $t0,$s1	Good CRC & 0x80
48	01 01 40 27	00	sll $t0, $t0,1	Good CRC <<1
4C	02 41 90 28	00	srl $s2, $s2, 1	V>>1
50	49 f2 00 5c	12	beq $t7, $s2, INC	Check Ch & V ?
54	49 d1 00 64	12	ChcXOR: beq $t6, $s1, XOR1	Check Good CRC & 0x80?
58	50 00 00 68	14	j shRt	If not, shift 0x80 right
5C	29 08 00 01	0A	INC: addi $t0, $t0, 1	Good CRC++
60	50 00 00 54	14	j ChcXOR	
64	01 a8 40 26	00	XOR1: xor $t0, $t0,$t5	Good CRC ^ Poly
68	29 8c ff ff	0A	shRt: addi $t4, $t4, -1	i--
6C	01 61 58 28	00	srl $t3, $t3, 1	0x80 >>1
70	02 08 40 24	00	And $t0, $t0,$s0	Mask Good CRC &0x00ff
74	50 00 00 2c	14	j Repeat	
78	02 28 70 24	00	Augment: And $t6, $t0,$s1	Good CRC & 0x80
7C	4a 60 00 34	12	Beq $s3, $zero, Done	Check i=0? (i==16)
80	2a 73 ff ff	0A	addi $s3, $zero, -1	i--
84	01 01 40 27	00	sll $t0, $t0,1	Good CRC <<1
88	02 08 40 24	00	And $t0, $t0,$s0	Mask Good CRC &0x00ff
8C	49 d1 00 98	12	beq $t6, $s1, XOR2	Check Good CRC & 0x80?
90	50 00 00 78	14	j Augment	
94	01 a8 40 26	00	xor $t0, $t0,$t5	Good CRC ^ Poly
98	50 00 00 78	14	j Augment	
9C	50 00 00 a4	14	Exit: j Exit	
0A				

iv.Text Parser Assembly Code:

Table 13. Text Parser Assembly.

PC	Machine language	Op code (hex)	CODE			comment
00	28 10 00 00	0A		Addi	$s0, $zero, $zero	Space Counter =0
04	28 11 00 00	0A		Addi	$s1, $zero, $zero	Non-Space Counter =0
08	28 12 00 00	0A		Addi	$s2, $zero, $zero	Deleted Space Counter =0
0C	28 0a 00 00	0A		Addi	$t2, $zero, $zero	Previous space =0
10	28 09 00 00	0A		Addi	$t1, $zero, $zero	Compare current space =0
14	28 0b 00 00	0A		Addi	$t3, $zero, Adres	Text Address
18	05 75 00 00	01	CNull:	Lw	$s5, 0($t3)	Load Text Address
1C	28 13 00 08	0A		Addi	$s3, $zero, 0x08	Back Space (ASCII)
20	28 14 00 0c	0A		Addi	$s4, $zero, 0x0c	Form Feed (ASCII)
24	28 16 00 20	0A		Addi	$s6, $zero, 0x20	Space (ASCII)
28	4e a0 00 38	13		Bnq	$s5, $zero, CSpa	If not Null, keep checking
2C	09 71 00 04	02		sw	$s1, 4($t3)	Store Non-Space Counter
30	09 72 00 08	02		sw	$s2, 8($t3)	Store Deleted Space Counter
34	50 00 00 8C	14		j	exit	Exit…
38	01 20 50 20	00	CSpa:	Add	$t2, $zero, $t1	Save Previous Address
3C	02 a0 48 20	00		Add	$t1, $zero, $s5	Save current address
40	4a b3 00 6C	12		Beq	$s5, $s3, Space	Check Back Space (0x8)
44	2a 73 00 01	0A		Addi	$s3, $s3, 1	Next ASCII
48	4a b4 00 6C	12		Beq	$s5, $s4, Space	Check Form Feed (0xC)
4C	2a 94 00 01	0A		Addi	$s4, $s4, 1	Next ASCII
50	4a b3 00 6C	12		Beq	$s5, $s3, Space	Check Tab (0x9)
54	2a 73 00 01	0A		Addi	$s3, $s3, 1	Next ASCII
58	4a b4 00 6C	12		Beq	$s5, $s4, Space	Check Carri Return (0xD)
5C	4a b6 00 6C	12		Beq	$s5, $s6, Space	Check Space (0x20)
60	4a b3 00 6C	12		Beq	$s5, $s3, Space	Check Line Feed (0xa)
64	2a 41 00 01	0A		Addi	$s1, $s1, 1	Non-Space ++
68	50 00 00 88	14		j	AddInc	Jump to Increment Address
6C	2a 10 00 01	0A	Space:	Addi	$s0, $s0, 1	Space++
70	49 49 00 78	12		Beq	$t2, $t1, IdenSp	Jump to Identical space
74	50 00 00 88	14		j	AddInc	Jump to Increment Address
78	09 60 00 00	02	IdenSp:	Sw	$zero, 0($t3)	Store it in Memory
7C	2a 52 00 01	0A		Addi	$s2, $s2, 1	Delated Space++
80	50 00 00 88	14		j	AddInc	Jump to Increment Address
84	29 6b 00 04	0A	AddInc:	Addi	$t3, $t3, 4	Next Address
88	50 00 00 18	14		j	CNull	Jump to Check Null
8C	50 00 00 90	14	Exit:	j	exit	Stay Here
90						

VIII.Appendix D Verilator Test Codes:

i. Daxpy Verilator code:

```
/* NASH CPU Top Level Verilator Simulation Test:
This Code is Written By The Support of Prof. Ali Muhtaroglu
And Team Work of Abduallah Damash, and Abdelaziz Al-Najjar
For Verifying the Functionality of NASH CPU
EEE446 Spring 2021 Middle East Technical University - Northern Cyprus Campus */

#include <stdio.h>
#include <verilated.h>
#include <verilated_vcd_c.h>
#include "testbench.h"
#include "Vtop.h"

// Top level interface signals defined here:
#define Opcode_Out      Opcode_Out

// Internal Signals:
// Note SystemVerilog design hierarchy can be traced by appending (__DOT__) at every level:
    #define PCSrc           top__DOT__u1__DOT__PCSrc
    #define RegFile         top__DOT__u1__DOT__RF
    #define t0              top__DOT__u1__DOT__RF[8]    //t0
    #define s1              top__DOT__u1__DOT__RF[17]   //s1
    #define s2              top__DOT__u1__DOT__RF[18]   //s2
    #define PC              top__DOT__u1__DOT__PC
    #define MEMread         top__DOT__MemRead
    #define MEMWrite        top__DOT__MemWrite
    #define Datamem         top__DOT__u3__DOT__data_cache_sram__DOT__dcache_sram
    #define stall           top__DOT__u1__DOT__StallSignal
    #define flush           top__DOT__u1__DOT__FlushSignal
    #define cachetall       top__DOT__cache_miss_stall
//Active DEBUG mode to do operations conditional:
    #define DEBUG       0

// Note the use of top level design name here after 'V' as class type:
class   TOPLEVEL_TB : public TESTBENCH<Vtop> {

long m_tickcount;

public:
TOPLEVEL_TB(void) {}

//Every time this procedure is called, clock is ticked once and associated
void tick(void) {

TESTBENCH<Vtop>::tick();
```

Figure 10: Daxpy Verilator code

```
//keeping track of number of clock ticks:
m_tickcount++;

//For DEBUG MODE ACTVATION
if (DEBUG)
    {
      printf("%08lx: ", m_tickcount);
      printf("%s: %lx", "Opcode_Out" ,m_topsim->Opcode_Out);
      printf("\n");
    }};

int main(int argc, char** argv, char** env){
// Create an instance of our module under test
  TOPLEVEL_TB *tb = new TOPLEVEL_TB;
// Track Parameters:
  long clock_count = 0;
  long error_count = 0;
  long instrs =0;
  long memread  =0;
  long memwrite = 0;
  float CPI = 0;
  long stallhazerd =0;
  long stallmem = 0;
  long totalclock =0;
  bool test_pass = false;
  long totalinst=0;
  float fmax= 78*10^6;

// Save The Previos Progrma Counter to Count the excuited instruction
  long oldPC =tb->m_topsim->Opcode_Out;
// Initialize Verilators variables
  Verilated::commandArgs(argc, argv);
// Starting the Test:
  printf("NASH user, we are cooking the test for you ;) ...\n");


  // Data will be dumped to trace file in gtkwave format to look at waveforms later:
  tb->opentrace("pipeline_waveforms.vcd");

  // Note this message will only be output if we are in DEBUG mode:
  if (DEBUG) printf("Giving the system 1 cycle to initialize with reset...\n");

  // Hit that reset button for one clock cycle:
  tb->reset();
  clock_count++;
```

```
 // Automatic Verification starts here:
   for (int k = 0; k < 50000 ; k++){

   if (tb->m_topsim->Opcode_Out != 20)
   {    tb->tick();
       clock_count++;
   // Count the Load (LW) Opeartions:
   if ( tb->m_topsim->Opcode_Out ==01 && tb->m_topsim->MEMread==1 && tb->m_topsim->cachetall !=1)
   {memread++;}
   // Count the Store (SW or JSW) Opeartions:
   if ((tb->m_topsim->Opcode_Out ==02 | tb->m_topsim->Opcode_Out == 23 /*JSW*/ ) && tb->m_topsim->MEMWrite==1 && tb->m_topsim->cachetall !=1)
   {memwrite++;}
   // Count the Executed Instructions:
   if (oldPC != tb->m_topsim->Opcode_Out  && tb->m_topsim->cachetall !=1)
   {instrs++;}
   // Count the Memory Stalls:
   if (tb->m_topsim->cachetall ==1 /* && tb->m_topsim->Opcode_Out!= 14 && oldPC != tb->m_topsim->Opcode_Out */)
   {stallmem++;}
   // Count the Hazerd Stalls:
   if (tb->m_topsim->stall ==1 || tb->m_topsim->flush == 1  && tb->m_topsim->cachetall !=1)  //&& tb->m_topsim->Opcode_Out!= 20
   {stallhazerd++;}

   /* (Functionality Test) */
   //Check the result
   if (clock_count==180)
   {
       if (tb->m_topsim->Datamem[1][1]!= 32764*4318-1000) error_count++;
   }

   oldPC =tb->m_topsim->Opcode_Out;
   }
   }

   totalclock = clock_count+stallmem;
   totalinst = instrs +stallhazerd +stallmem;
   CPI = (float)clock_count/totalinst;
   test_pass = (error_count > 0) ? 0 : 1;
   printf("Execution completed successfully (simulation waveforms in .vcd file) ... !\n");
   printf("# Read data memory (LW): %ld\n",memread);
   printf("# Write data memory (SW): %ld\n",memwrite);
   printf("Elapsed Clock Cycles: %ld\n",clock_count);
   printf("Stall Cycles due to MEMory: %ld\n",stallmem);
   printf("Stall Cycles due to Loadhazerd: %ld\n",stallhazerd);
   printf("Executed Instructions: %ld\n",instrs);
   printf("Total Executed Instructions with stall: %ld\n",totalinst);
```

```
136     printf("Total Executed Instructions with stall: %ld\n",totalinst);
137     printf("Execution Time: %0.4f us\n",(float)(clock_count)/(fmax));
138     printf("Effective CPI : %0.2f\n",(float)(clock_count)/(instrs+stallhazerd));
139     printf("Effective CPI with Memory Stall : %0.2f\n",CPI);
140     printf("Functional verification Status: %s\n",test_pass?"PASS":"FAIL");
141     if (error_count > 0)
142     {
143     printf("Error count is: %ld\n",error_count);
144     }
145
146   exit(EXIT_SUCCESS);
147
148 }
149
150
```

ii.Added part for Length Verilator code:

```
113
114     /* (Functionality Test) */
115     //Check the result
116     if (tb->m_topsim->Opcode_Out == 12) // this opcode is a terminator
117     {if (tb->m_topsim->length!= 48) error_count++;}
118
119     oldPC =tb->m_topsim->Opcode_Out;
120     }
121     }
122
123     totalclock = clock_count+stallmem;
124     totalinst = instrs +stallhazerd +stallmem;
125     CPI = (float)clock_count/totalinst;
126     test_pass = (error_count > 0) ? 0 : 1;
```

Figure 11: Added part for length checl

iii.MergeSort Verilator code:

```
int main(int argc, char** argv, char** env){
// Create an instance of our module under test
  TOPLEVEL_TB *tb = new TOPLEVEL_TB;
// Track Parameters:
  long clock_count = 0;
  long error_count = 0;
  long instrs =0;
  long memread  =0;
  long memwrite = 0;
  float CPI = 0;
  long stallhazerd =0;
  long totalclock =0;
  bool test_pass = false;
  long totalinst=0;
  float fmax= 78*10^6;
  int j=0;
  int32_t x[32];
  int32_t y[32];
  unsigned char w = 8;
// Save The Previos Progrma Counter to Count the excuited instruction
  long oldPC =tb->m_topsim->Opcode_Out;
// Initialize Verilators variables
  Verilated::commandArgs(argc, argv);
// Starting the Test:
  printf("NASH user, we are cooking the test for you ;) ...\n");

  // Data will be dumped to trace file in gtkwave format to look at waveforms later:
  tb->opentrace("pipeline_waveforms.vcd");

  // Note this message will only be output if we are in DEBUG mode:
  if (DEBUG) printf("Giving the system 1 cycle to initialize with reset...\n");

  // Hit that reset button for one clock cycle:
  tb->reset();
  clock_count++;

  // Automatic Verification starts here:
    for (int k = 0; k < 100000 ; k++){

    if (tb->m_topsim->Opcode_Out != 12)
    {   tb->tick();
        clock_count++;
```

Figure 12: MergeSort Verilator Code

```
    // Count the Load (LW) Opeartions:
    if ( tb->m_topsim->Opcode_Out ==01 && tb->m_topsim->MEMread==1 )
    {memread++;}
    // Count the Store (SW or JSW) Opeartions:
    if ((tb->m_topsim->Opcode_Out ==02 | tb->m_topsim->Opcode_Out == 23 /*JSW*/ ) && tb->m_topsim->MEMWrite==1 )
    {memwrite++;}
    // Count the Executed Instructions:
    if (oldPC != tb->m_topsim->Opcode_Out  )
    {instrs++;}

    // Count the Hazerd Stalls:
    if (tb->m_topsim->stall ==1 || tb->m_topsim->flush == 1  )  //&& tb->m_topsim->Opcode_Out!= 20
    {stallhazerd++;}

    /* (Functionality Test) */
    //Check the result
    if (tb->m_topsim->Opcode_Out == 12) // this opcode is a terminator
    {
        for(int i=0; i<((tb->m_topsim->length)*4)*2+4 ;i=i+4 ){
            y[j] = tb->m_topsim->Datamem[i] & 0xFF;
            y[j] = (y[j] << 8) + (tb->m_topsim->Datamem[i+1] & 0xFF);
            y[j] = (y[j] << 8) + (tb->m_topsim->Datamem[i+2] & 0xFF);
            y[j] = (y[j] << 8) + (tb->m_topsim->Datamem[i+3] & 0xFF);
            j++;
        }

        j=0;
        for(int i=40; i<72 ;i=i+4 ){
            x[j] = tb->m_topsim->Datamem[i] & 0xFF;
            x[j] = (x[j] << 8) + (tb->m_topsim->Datamem[i+1] & 0xFF);
            x[j] = (x[j] << 8) + (tb->m_topsim->Datamem[i+2] & 0xFF);
            x[j] = (x[j] << 8) + (tb->m_topsim->Datamem[i+3] & 0xFF);
            j++;
        }

    }
    oldPC =tb->m_topsim->Opcode_Out;
    }
    }

    totalclock = clock_count;
    totalinst = instrs +stallhazerd ;
```

```
      totalinst = instrs +stallhazerd ;
      CPI = (float)clock_count/totalinst;
      printf("Execution completed successfully (simulation waveforms in .vcd file) ... !\n");
      printf("# Read data memory (LW): %ld\n",memread);
      printf("# Write data memory (SW): %ld\n",memwrite);
      printf("Elapsed Clock Cycles: %ld\n",clock_count);
      printf("Stall Cycles due to Loadhazerd: %ld\n",stallhazerd);
      printf("Executed Instructions: %ld\n",instrs);
      printf("Total Executed Instructions with stall: %ld\n",totalinst);
      printf("Execution Time: %0.4f us\n",(float)(clock_count)/(fmax));
      printf("Effective CPI : %0.2f\n",(float)(clock_count)/(instrs+stallhazerd));
      printf("input lists in hex : {");
          for (int i=0; i<(tb->m_topsim->length)*2+1 ;i++ ){
              printf(" %X",y[i]);
          }
      printf(" }\n");
      printf("output list in ascending order: {");
          for (int i=0; i<(tb->m_topsim->length)*2 ;i++ ){
              if ((i<((tb->m_topsim->length)*2-1))&&(x[i] > x[i+1]))error_count++;
              printf(" %X",x[i]);
          }
      printf(" }\n");
      test_pass = (error_count > 0) ? 0 : 1;
      printf("Functional verification Status: %s\n",test_pass?"PASS":"FAIL");
      if (error_count > 0)
      {
      printf("Error count is: %ld\n",error_count);
      }

    exit(EXIT_SUCCESS);

}
```

iv.CRC Verilator Code:

```
    /* Verification Teseing for the Output is Satart Here (Functionality Test Approch) */
    // When Program is done, Check the Corroct Output
    if (tb->m_topsim->Opcode_Out == 12)
    {if (tb->m_topsim->GoodCRC!= 0x9479) error_count++;}

    //For DEBUG MODE ACTVATION
    if (tb->m_topsim->Instruction == 688521216){
      printf("%s: %lx ", "Opcode_Out" ,tb->m_topsim->Opcode_Out);
      printf("%s: %lx ", "PC" ,tb->m_topsim->PC);
      printf("%s: %lx ", "Input" ,tb->m_topsim->Input);
      printf("%s: %lx ", "Current CRC" ,tb->m_topsim->CurrentCRC);
      printf("\n");
    }
    // Save Previous PC
    oldPC =tb->m_topsim->Opcode_Out;}}
    // End of Testing, Printing the Output:
    totalinst = instrs +stallhazerd +stallmem;
    CPI = (float)clock_count/totalinst;
    test_pass = (error_count > 0) ? 0 : 1;
    printf("Execution completed successfully (simulation waveforms in .vcd file) ... !\n");
    printf("Elapsed Clock Cycles: %ld\n",clock_count);
    printf("Fmax: %.2f MHz\n",fmax);
    printf("Total Averge Power: %.2f W\n",Pavg);
    printf("Execution Time: %0.4f us\n",(float)(clock_count)/(fmax));
    printf("Energy: %0.4f uJ\n",(float)((clock_count)/(fmax))*(Pavg));
    printf("# Read data memory (LW): %ld\n",memread);
    printf("# Write data memory (SW): %ld\n",memwrite);
    printf("Stall Cycles due to MEMory: %ld\n",stallmem);
    printf("Stall Cycles due to Loadhazerd: %ld\n",stallhazerd);
    printf("Executed Instructions: %ld\n",instrs);
    printf("Total Executed Instructions with stall: %ld\n",totalinst);
    printf("Effective CPI without Memory stall: %0.2f\n",(float)(clock_count)/(instrs+stallhazerd));
    printf("Effective CPI with Memory Stall : %0.2f\n",CPI);
    printf("Functional verification Status: %s\n",test_pass?"PASS":"FAIL");
    printf("Input Intger: 0x0041\n");
    printf("TCITT Poly: 0x%x\n",tb->m_topsim->Poly);
    printf("Output CRC: 0x%x\n",tb->m_topsim->GoodCRC);
    if (error_count > 0){printf("Error count is: %ld\n",error_count);}
```

Figure 13. CRC Verilator Code.

v.Text Parser Code:

```
    /* Verification Teseing for the Output is Satart Here (Functionality Test Approch) */
    // When Program is done, Check the Corroct Output
    if (tb->m_topsim->Opcode_Out == 12)
    {if (tb->m_topsim->SpaceCont!= 13) error_count++;}

    // Save Previous PC
    oldPC =tb->m_topsim->Opcode_Out;}}

    // End of Testing, Printing the Output:
    totalinst = instrs +stallhazerd +stallmem;
    CPI = (float)clock_count/totalinst;
    test_pass = (error_count > 0) ? 0 : 1;
    printf("Execution completed successfully (simulation waveforms in .vcd file) ... !\n");
    printf("Elapsed Clock Cycles: %ld\n",clock_count);
    printf("Fmax: %.2f MHz\n",fmax);
    printf("Total Averge Power: %.2f W\n",Pavg);
    printf("Execution Time: %0.4f us\n",(float)(clock_count)/(fmax));
    printf("Energy: %0.4f uJ\n",(float)((clock_count)/(fmax))*(Pavg));
    printf("# Read data memory (LW): %ld\n",memread);
    printf("# Write data memory (SW): %ld\n",memwrite);
    printf("Stall Cycles due to MEMory: %ld\n",stallmem);
    printf("Stall Cycles due to Loadhazerd: %ld\n",stallhazerd);
    printf("Executed Instructions: %ld\n",instrs);
    printf("Total Executed Instructions with stall: %ld\n",totalinst);
    printf("Effective CPI without Memory stall: %0.2f\n",(float)(clock_count)/(instrs+stallhazerd));
    printf("Effective CPI with Memory Stall : %0.2f\n",CPI);
    printf("Functional verification Status: %s\n",test_pass?"PASS":"FAIL");

    printf("Spaces Counter: %d\n",tb->m_topsim->SpaceCont);
    printf("Non-Spaces Counter: %d\n",tb->m_topsim->NonSpaceCont);
    printf("Deleted Space Counter: %d\n",tb->m_topsim->DeltSpacCont);
    if (error_count > 0)
    {
    printf("Error count is: %ld\n",error_count);
    }
```

Figure 14. Text Parser Verilator Code.

IX. Appendix E Quartus II Verification Reports:

i. Datapath & Control Reports:

a. Datapath Cost Report:

Fitter Resource Usage Summary

	Resource	Usage
1	Total logic elements	4,030 / 68,416 (6 %)
1	-- Combinational with no register	2575
2	-- Register only	409
3	-- Combinational with a register	1046
2		
3	Logic element usage by number of LUT inputs	
1	-- 4 input functions	2792
2	-- 3 input functions	714
3	-- <=2 input functions	115
4	-- Register only	409
4		
5	Logic elements by mode	
1	-- normal mode	3372
2	-- arithmetic mode	249
6		
7	Total registers*	1,455 / 70,234 (2 %)
1	-- Dedicated logic registers	1,455 / 68,416 (2 %)
2	-- I/O registers	0 / 1,818 (0 %)
8		
9	Total LABs: partially or completely used	317 / 4,276 (7 %)
10	Virtual pins	0
11	I/O pins	156 / 622 (25 %)
1	-- Clock pins	3 / 8 (38 %)
12		
13	Global signals	5
14	M4Ks	1 / 250 (< 1 %)
15	Total block memory bits	39 / 1,152,000 (< 1 %)
16	Total block memory implementation bits	4,608 / 1,152,000 (< 1 %)
17	Embedded Multiplier 9-bit elements	12 / 300 (4 %)
18	PLLs	0 / 4 (0 %)
19	Global clocks	5 / 16 (31 %)
20	JTAGs	0 / 1 (0 %)

* Register count does not include registers inside RAM blocks or DSP blocks.

Figure 15. Datapath & Control Unit Fitter Usage Report.

b. Datapath Timing Report:

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	62.1 MHz	62.1 MHz	clock	

Figure 16. Datapath & Control Unit Slow Model Report.

c. Datapath Power Report:

PowerPlay Power Analyzer Summary

PowerPlay Power Analyzer Status	Successful - Mon Jul 12 12:51:08 2021
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	top
Top-level Entity Name	datapath446
Family	Cyclone II
Device	EP2C70F896C6
Power Models	Final
Total Thermal Power Dissipation	333.32 mW
Core Dynamic Thermal Power Dissipation	91.74 mW
Core Static Thermal Power Dissipation	155.44 mW
I/O Thermal Power Dissipation	86.15 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Figure 17. Datapath & Control Unit Powerplay Analyzer Report

Thermal Power Dissipation by Block Type

	Block Type	Total Thermal Power by Block Type	Block Thermal Dynamic Power	Block Thermal Static Power (1)
1	I/O	51.50 mW	29.58 mW	18.70 mW
2	M4K	1.27 mW	1.07 mW	--
3	Embedded multiplier block	5.12 mW	5.12 mW	--
4	Embedded multiplier output	0.55 mW	0.00 mW	--
5	Combinational cell	33.95 mW	14.28 mW	--
6	Register cell	27.14 mW	10.57 mW	--
7	Clock control block	20.49 mW	0.00 mW	--

Figure 18. Datapath & Control Unit Powerplay Analyzer Report

ii. D-Cache Reports:

a. D-Cache Cost Report:

Fitter Resource Usage Summary

	Resource	Usage
1	Total logic elements	5,382 / 68,416 (8 %)
1	-- Combinational with no register	1203
2	-- Register only	1950
3	-- Combinational with a register	2229
2		
3	Logic element usage by number of LUT inputs	
4		
5	Logic elements by mode	
6		
7	Total registers*	4,179 / 70,234 (6 %)
1	-- Dedicated logic registers	4,179 / 68,416 (6 %)
2	-- I/O registers	0 / 1,818 (0 %)
8		
9	Total LABs: partially or completely used	422 / 4,276 (10 %)
10	Virtual pins	0
11	I/O pins	612 / 622 (98 %)
12		
13	Global signals	1
14	M4Ks	0 / 250 (0 %)
15	Total block memory bits	0 / 1,152,000 (0 %)
16	Total block memory implementation bits	0 / 1,152,000 (0 %)
17	Embedded Multiplier 9-bit elements	0 / 300 (0 %)

Figure 19. D-Cache Fitter Report

b. D-Cache Timing Report:

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	100.72 MHz	100.72 MHz	clock	

Figure 20. D-Cache Slow Model Report.

c. D-Cache Power Report:

PowerPlay Power Analyzer Summary	
PowerPlay Power Analyzer Status	Successful - Mon Jul 12 12:36:54 2021
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	top
Top-level Entity Name	dcache_controller
Family	Cyclone II
Device	EP2C70F896C6
Power Models	Final
Total Thermal Power Dissipation	570.78 mW
Core Dynamic Thermal Power Dissipation	139.97 mW
Core Static Thermal Power Dissipation	156.29 mW
I/O Thermal Power Dissipation	274.52 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Figure 21. D-Cache Powerplay Analyzer Report.

Thermal Power Dissipation by Block Type

	Block Type	Total Thermal Power by Block Type	Block Thermal Dynamic Power	Block Thermal Static Power (1)
1	I/O	255.34 mW	164.85 mW	71.81 mW
2	Combinational cell	65.66 mW	15.15 mW	--
3	Register cell	33.26 mW	24.27 mW	--
4	Clock control block	22.37 mW	0.00 mW	--

Figure 22. D-Cache Thermal Power by Block Type Report.

iii. I-Cache Reports:

a. I-Cache Cost Report:

Fitter Resource Usage Summary

	Resource	Usage
1	Total logic elements	2,536 / 68,416 (4 %)
1	-- Combinational with no register	446
2	-- Register only	787
3	-- Combinational with a register	1303
2		
3	Logic element usage by number of LUT inputs	
1	-- 4 input functions	1198
2	-- 3 input functions	545
3	-- <=2 input functions	6
4	-- Register only	787
4		
5	Logic elements by mode	
1	-- normal mode	1749
2	-- arithmetic mode	0
6		
7	Total registers*	2,090 / 70,234 (3 %)
1	-- Dedicated logic registers	2,090 / 68,416 (3 %)
2	-- I/O registers	0 / 1,818 (0 %)
8		
9	Total LABs: partially or completely used	160 / 4,276 (4 %)
10	Virtual pins	0
11	I/O pins	318 / 622 (51 %)
1	-- Clock pins	4 / 8 (50 %)
12		
13	Global signals	2
14	M4Ks	0 / 250 (0 %)
15	Total block memory bits	0 / 1,152,000 (0 %)
16	Total block memory implementation bits	0 / 1,152,000 (0 %)
17	Embedded Multiplier 9-bit elements	0 / 300 (0 %)
18	PLLs	0 / 4 (0 %)
19	Global clocks	2 / 16 (13 %)
20	JTAGs	0 / 1 (0 %)

Figure 23. I-Cache Fitter Usage Report.

b. I-Cache Timing Report:

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	146.56 MHz	146.56 MHz	clock	

Figure 24.I-Cache Slow Model Report.

c. I-Cache Power Report:

PowerPlay Power Analyzer Summary	
PowerPlay Power Analyzer Status	Successful - Mon Jul 12 12:42:15 2021
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	top
Top-level Entity Name	icache_controller
Family	Cyclone II
Device	EP2C70F896C6
Power Models	Final
Total Thermal Power Dissipation	311.40 mW
Core Dynamic Thermal Power Dissipation	47.96 mW
Core Static Thermal Power Dissipation	155.36 mW
I/O Thermal Power Dissipation	108.07 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Figure 25. I-Cache Powerplay Analyzer Report.

Thermal Power Dissipation by Block Type

	Block Type	Total Thermal Power by Block Type	Block Thermal Dynamic Power	Block Thermal Static Power (1)
1	I/O	80.65 mW	32.59 mW	37.61 mW
2	Combinational cell	9.39 mW	5.55 mW	--
3	Register cell	10.77 mW	8.55 mW	--
4	Clock control block	17.35 mW	0.00 mW	--

Figure 26. I-Cache Thermal Power by Block Type Report.

X.Appendix F Verilator Test Results:

i.Integer Daxpy Verilator output:

```
aziz@aziz-Aspire-F5-573G:~/uni/EEE446/module5/module5verilator$ ./obj_dir/Vtop
NASH user, we are cooking the test for you ;) ...
Execution completed successfully (simulation waveforms in .vcd file) ... !
# Read data memory (LW): 18
# Write data memory (SW): 9
Elapsed Clock Cycles: 274
Stall Cycles due to MEMory: 185
Stall Cycles due to Loadhazerd: 9
Executed Instructions: 70
Total Executed Instructions with stall: 264
Execution Time: 0.3522 us
Effective CPI : 3.47
Effective CPI with Memory Stall : 1.04
Functional verification Status: PASS
```

Figure 27: Integer Daxpy Verilator Result

ii. Length function Verilator output:

```
aziz@aziz-Aspire-F5-573G:~/uni/EEE446/module5/module5verilator$ ./obj_dir/Vtop
NASH user, we are cooking the test for you ;) ...
Execution completed successfully (simulation waveforms in .vcd file) ... !
# Read data memory (LW): 49
# Write data memory (SW): 0
Elapsed Clock Cycles: 678
Stall Cycles due to MEMory: 332
Stall Cycles due to Loadhazerd: 49
Executed Instructions: 335
Total Executed Instructions with stall: 716
Execution Time: 0.8715 us
Effective CPI : 1.77
Effective CPI with Memory Stall : 0.95
Returned length : 48
Functional verification Status: PASS
```

Figure 28: Length function Verilator result

iii. MergeSorter function Verilator test output:

```
aziz@aziz-Aspire-F5-573G:~/uni/EEE446/module5/module5noMemory$ ./obj_dir/Vtop
NASH user, we are cooking the test for you ;) ...
Execution completed successfully (simulation waveforms in .vcd file) ... !
# Read data memory (LW): 32
# Write data memory (SW): 12
Elapsed Clock Cycles: 241
Stall Cycles due to Loadhazerd: 32
Executed Instructions: 166
Total Executed Instructions with stall: 198
Execution Time: 0.3098 us
Effective CPI : 1.22
input lists in hex : { FF 101 10A 201 0 1 102 1FF 203 }
output list in ascending order: { 1 FF 101 102 10A 1FF 201 203 }
Functional verification Status: PASS
```

Figure 29: MergeSorter Verilator output

iv.CRC function Verilator test output:

```
Hello Dear, Test Executing is Starting Now ...
Execution completed successfully (simulation waveforms in .vcd file) ... !
Elapsed Clock Cycles: 694
Fmax: 62.68 MHz
Total Averge Power: 1.21 W
Execution Time: 11.0721 us
Energy: 13.3973 uJ
# Read data memory (LW): 6
# Write data memory (SW): 1
Stall Cycles due to MEMory: 369
Stall Cycles due to Loadhazerd: 59
Executed Instructions: 199
Total Executed Instructions with stall: 627
Effective CPI without Memory stall: 2.69
Effective CPI with Memory Stall : 1.11
Functional verification Status: PASS
Input Intger: 0x0041
TCITT Poly: 0x1021
Output CRC: 0x9479
abood@DESKTOP-PTQHPLT:~/EEE446Lab5/CRC_SORT$ chmod 755 -R CRC_test.hex
```

Figure 30. CRC Verilator Output.

v.Text Parser:

```
abood@DESKTOP-PTQHPLT:~/EEE446Lab5/Text$ ./obj_dir/Vtop
Hello Dear, Test Executing is Starting Now ...
Execution completed successfully (simulation waveforms in .vcd file) ... !
Elapsed Clock Cycles: 1529
Fmax: 62.68 MHz
Total Averge Power: 1.21 W
Execution Time: 24.3937 us
Energy: 29.5164 uJ
# Read data memory (LW): 47
# Write data memory (SW): 4
Stall Cycles due to MEMory: 443
Stall Cycles due to Loadhazerd: 154
Executed Instructions: 811
Total Executed Instructions with stall: 1408
Effective CPI without Memory stall: 1.58
Effective CPI with Memory Stall : 1.09
Functional verification Status: PASS
Spaces Counter: 13
Non-Spaces Counter: 33
Deleted Space Counter: 2
```

Figure 31. Text Parser Verilator Output.