



EEE 446

MODULE #5

Programming & Verification of a Pipelined CPU with Basic Cache
Hierarchy

AMO

2021, Spring Semester

OBJECTIVE

The main objective of this final lab is to assimilate all the pieces built over the semester into a single computer, verify its functionality, as well as measure its performance and energy consumption. This report will firstly go over the ideology behind the main design approach. This will be followed by a discussion regarding the challenges faced during this journey, and how AMO was debugged. Afterwards, ISA design choices will be explained briefly regarding the main instruction formats, full ISA is illustrated in Appendix A. Similarly, a summary regarding the AMO's Datapath and control design will follow, whereas in Appendix B the Datapath and Control will be shown in detail, along with the system Verilog files. Furthermore, a brief description, with the help of some basic statistics, regarding the algorithms associated with each benchmark will be given, where the Assembly and Machine code for the respective benchmarks are provided in Appendix C. Finally, the Verilator tests will be described concisely regarding the critical parameter calculations and verification of correct functionality, the tests will be listed in Appendix D, whereas Quartus II's critical parameters will be illustrated in Appendix E such as f_{\max} , Power Dissipation etc., generated with the help of Quartus II tools.

MAIN DESIGN APPROACH

The 4 key principles while optimizing the performance were the main pillars on which this CPU was built.

- i. Simplicity favors regularity
- ii. Make the Common Case fast
- iii. Good design demands good compromises
- iv. Smaller is faster

Simplistic small design while making the common case fast were the keys designing the ISA. This will be further explained in the ISA section, whereas performance was preferred in some key areas such as multiplication at the cost of more power i.e., extra hardware. Another key parameter that was kept in mind while designing the CPU was to use the hardware to its fullest extent. Prime example of this is the utilization of the two ALUs in the design to perform different useful combinations of arithmetic instructions on the same data, saving time, hence increasing performance. The ISA section will also cover these various combinations of instructions that can come in handy in common yet tedious cases, allowing to use a single instruction instead of two in a normal 5-stage pipelined CPU. Another key feature that was targeted in this design was to optimize the control over the byte-addressable memory, to use it an efficient manner. This was done so by using a memory controller but more on that in the Datapath section.

CHALLENGES

Over the course of this semester, many challenges were encountered while designing the CPU and the two-level memory hierarchy. To mention a few, the first challenge that was encountered was to fully optimize the usage of the second ALU in the Datapath. To do so, as mentioned before, and more on that in the ISA, special instruction formats were created. Another hindrance was encountered while testing the CPU design, where we found the f_{\max} to take a significant hit, due to combinational multiplication and division. To counter this, extra-special hardware was incorporated into the Datapath to handle multiplication & division, resulting in a significant boost in performance. Implementation of forwarding

hardware was also done successfully to avoid any dependency hazards. Finally, the integration of the two-level memory hierarchy with the CPU design was found to be challenging due to an occurrence of a bug during the integration. This was successfully fixed prior to the final report, so that a fully functional CPU integrated with the basic Cache hierarchy is ready for some rigorous testing with the help of benchmarks, where key parameters can be found out for AMO.

ISA

Following a Big-Endian notation, our ISA supports 8 instruction formats, with all the detail of them explained thoroughly in Appendix A. Supporting the basic Register-Type, Immediate-Type, Branch-Type & a Jump-Type format were kept simple and similar to RISC-V, to provide coherence and regularity. These essential formats were kept similar, with func2 field i.e., ALU1 control, being the same in all cases, providing regularity, apart from the Jump-Type instruction. All the instruction formats perform an implicit PC increment (PC+4). Two new types of Instruction formats were introduced, to optimize 2nd ALU functionality, which are:

i. Fused-Arithmetic Instruction Type

ii. Store-Jump Type

Fused-Arithmetic Instruction Type allowed us to manipulate data more efficiently, as we are using a single instruction for an operation which would normally require two instructions. This decision was made with the key benchmarks in mind. For instance, the IAXPY benchmark, requires multiplication & addition to be done on the data. With this Fused-Arithmetic Instruction Type, it enabled us to execute multiplication & addition on the data, with the help of this format. Furthermore, other key combinations can also be used for executing a process on the given data, for instance, the 1st ALU can be used for XOR operation, while the 2nd ALU, can be used for Shift operation. A basic combination, which can provide efficient multiplication or division operations for given data. Looking ahead, if required anywhere, the versatility of this instruction format allows us to use all kinds of combinations supported by the two ALUs.

The Store-Jump Type format was introduced bearing in mind the Sorter + CRC benchmark. For these kind of memory extensive operations, it provided us with a unique opportunity to enhance the performance further. This also helped us to hide the flush cycle incurred by the jump in some cases, while accessing the memory.

The main Instruction formats are as listed below in Table 1, all support implicit PC incrementation:

R-Type	Op [31:26]	Rd [25:21]	Rs1 [20:16]	Rs2[15:11]	Func[10:4]		Func2[3:0]
I-Type	Op [31:26]	Rd [25:21]	Rs1 [20:16]	IMM[15:4]			Func2[3:0]
B-Type	Op [31:26]	SIMM[25:21]	Rs1 [20:16]	Rs2[15:11]	SIMM[10:4]		Func2[3:0]
F-Type	Op [31:26]	Rd [25:21]	Rs1 [20:16]	Rs2[15:11]	Rs3[10:6]	Func3[5:4]	Func2[3:0]
SJ-Type	Op [31:26]	FJIMM[25:21]	Rs1 [20:16]	Rs2[15:11]	FJIMM[10:8]	FSIMM[7:0]	
J-Type	Op [31:26]	JIMM[25:0]					
Jal-Type	Op [31:26]	Rd [25:21]	JIMM[20:0]				

Table 1

The key features in terms of regularity, as shown above in Table 1 are as follows:

- i. Opcode at the same location (6 MSBs)
- ii. Destination & Source registers at the same location (Bit 20 to 11)
- iii. Func2 field controlling ALU at the same location (4 LSBs)
- iv. I-Type format is used for Load operation as well
- v. B-Type format is used for Store operation as well

For further detail, see Appendix A.

DATAPATH

The main features of this 5-Stage Pipelined CPU, from where it differs from a standard 5-stage RISC-V pipeline, are as following:

- i. 2-ALUs
- ii. Data Memory Control
- iii. Multiplication & Division Hardware
- iv. Branch & Jump evaluation in the Memory Stage

The main components of this Datapath are as follows:

i. Instruction Memory (2 kB)	FETCH STAGE
ii. 3-read ports Register File (RF)	DECODE STAGE
iii. 1 st ALU	EXECUTE STAGE
iv. Multiplication & Division Hardware	EXECUTE STAGE
v. Data Memory (2 kB) & 2 nd ALU in parallel	MEMORY STAGE
vi. Forwarding & Hazard Detection Hardware	EXECUTE & DECODE STAGE
vii. 5 Pipeline registers	[PC, IF/ID, ID/EX, EX/MEM, MEM/WB]

To see the System Verilog files for the Datapath, refer to Appendix B.

Below in Table 2, Control Design is illustrated, where Figure 1 shows the CPU Datapath:

Instructions	Execution Stage/Address Calculation	Memory Access Stage/Second ALU/Jump/Branch- Address Calculation						Write-Back Stage	
	ALUSrc	Mem Read	Mem Write	Store- Jump	Second- ALU_en	Branch	Jump	Mem toReg	Reg Write
R-Type	0	0	0	0	0	0	0	0	1
I-Type	1	0	0	0	0	0	0	0	1
I-Type-Load	1	1	0	0	0	0	0	1	1
B-Type	0	0	0	0	0	1	0	0	0
B-Type-Store	1	0	1	0	0	0	0	0	0

F-Type	0	0	0	0	1	0	0	0	1
SJ-Type	1	0	1	1	0	0	1	0	0
J-Type	0	0	0	0	0	0	1	0	0

Table 2

Figure 1 below shows the Datapath design of AMO:

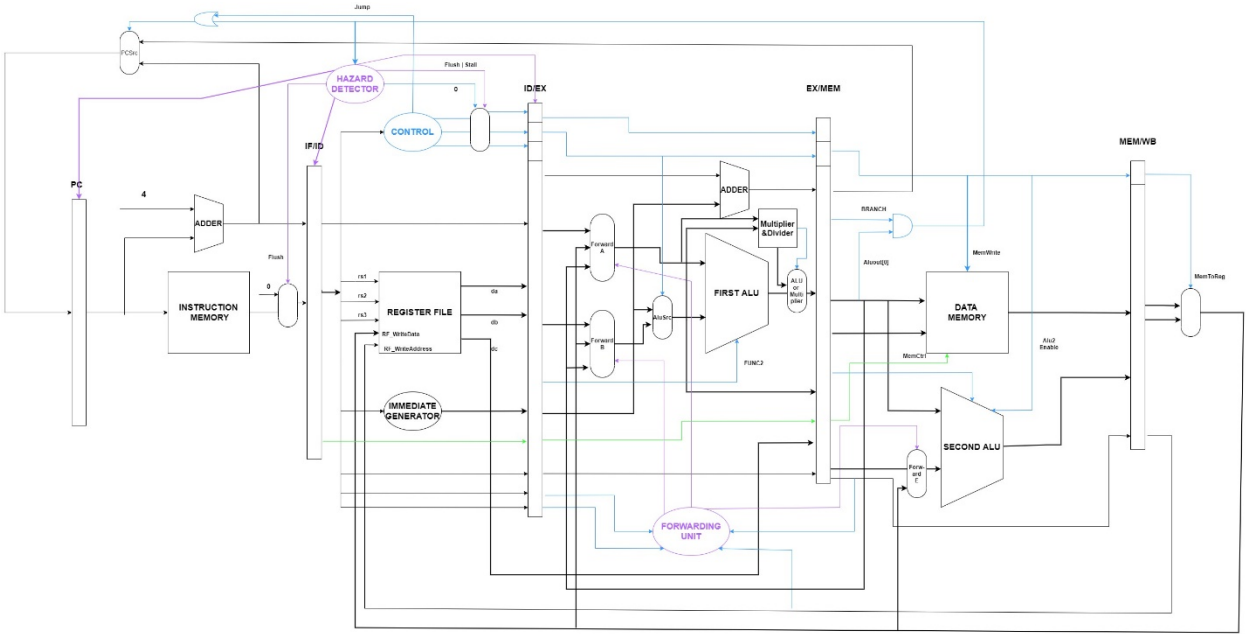


Figure 1

Legend:

Data Bus	
Control	
Forwarding & Hazard	
Func2-Mem-ctrl	

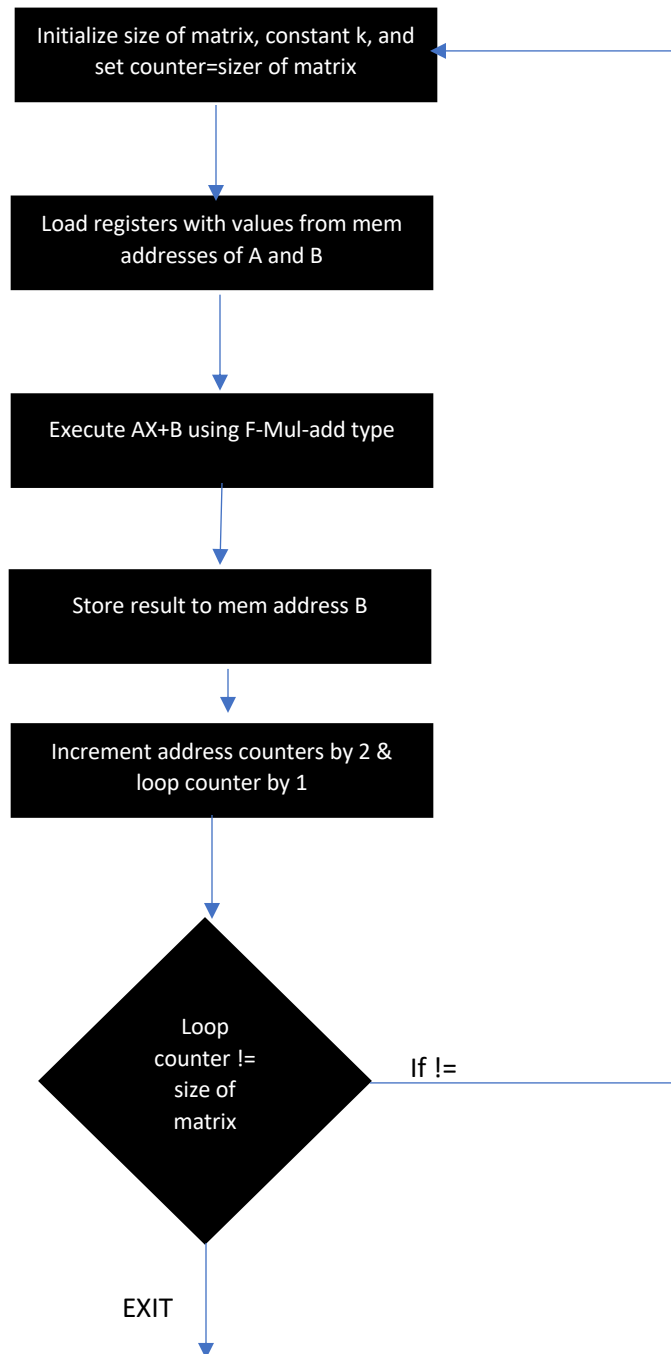
ALGORITHMS & BENCHMARKS

This section illustrates the algorithms associated with each of the three benchmarks, IAXPY, Sorter + CRC & Text Parser. In Table 3 below number of instructions took to write the code for each benchmark will also be mentioned. Refer to Appendix C for the full versions of the assembly and machine codes for each benchmark.

Benchmark	No. of Instructions
IAXPY	13
Sorter + CRC	131
Text Parser	27
CRC only	25

Table 3

IAXPY Below is an algorithmic flowchart, to describe the steps taken to write this benchmark:

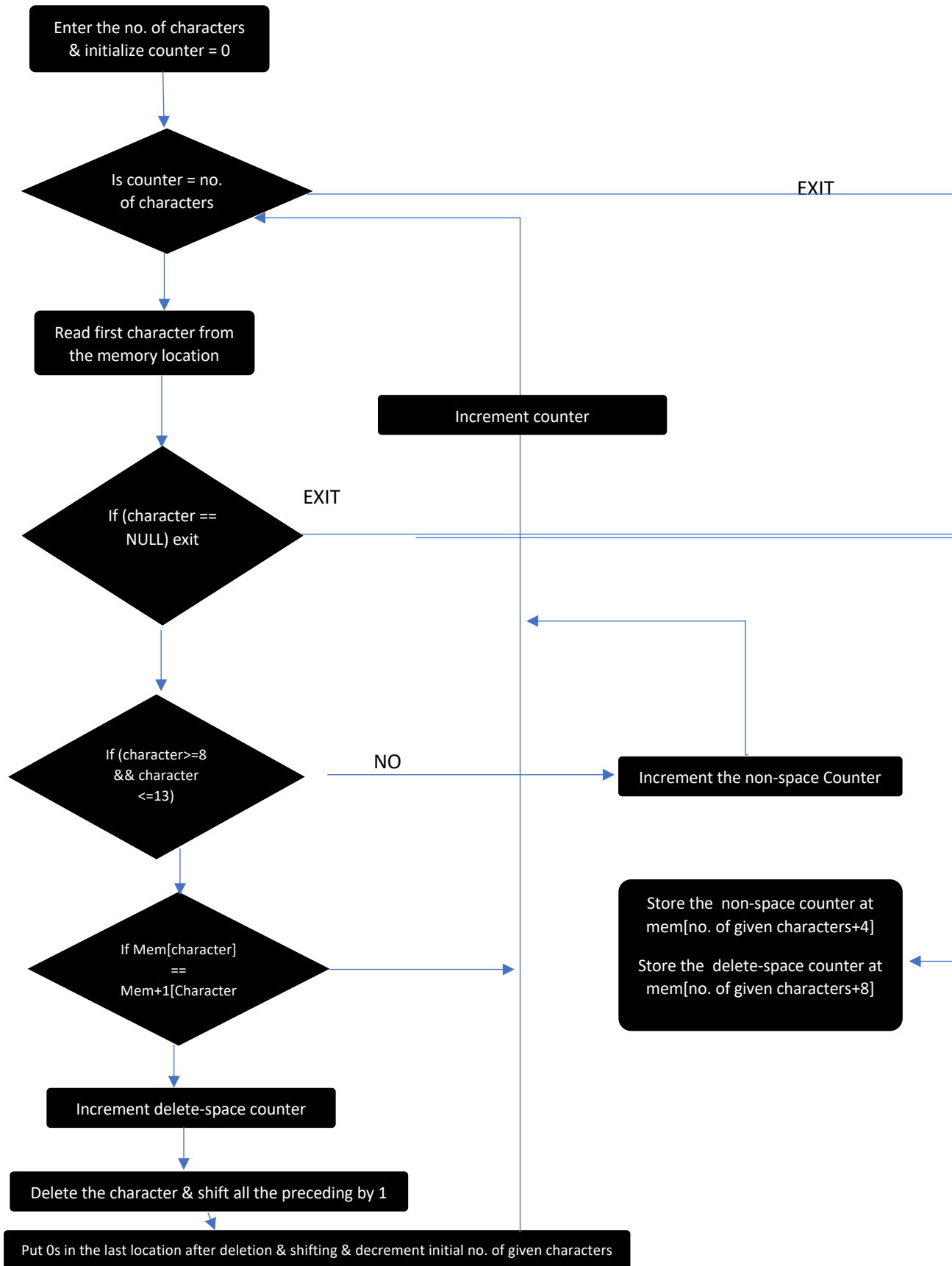


Sorter + CRC

Steps:

- i. Send size of array & the array to Merge-sort function
- ii. 1st recursively, break down the left part of the array using the midpoint index
$$\text{middle } m = l + (r-l)/2$$
- iii. 2nd recursion should sort the right half
- iv. Call Merge function, to merge them in ascending order
See Appendix C for its detailed explanation
- v. Start to append the CRC to the integers
- vi. Load a CRC register and another register with value 0x1021
- vii. Compare the LSB of the integer and LSB of CRC one by one
- viii. If they are the same, shift CRC register to right
- ix. If not, XOR CRC register with 0x1021 and then shift right
- x. When finished with all the bits of the 16-bit integer, append it at the upper half of integer
- xi. Store it as word in the original location.

Text Parser: Below is an algorithmic flowchart, to describe the steps taken to write this benchmark:



VERILATOR & QUARTUS VERIFICATION

To see the Verilator code for each benchmark, please refer to Appendix D. Verilator has been used to calculate the critical parameters for each benchmark such as CPI, total number of instructions, total number of clock cycles, number of stall cycles due to memory, execution time & energy consumption. The gist of all the Verilator codes associated with each benchmark was to check, when the benchmark has finished executing and report parameters such as total number of instructions and the other mentioned parameters above. For verification purposes, GTKWave was used to see the results, this was accomplished by the help of generating a vcd file using the Verilator. The results are shown in detail in Appendix D. With the help of Time-Quest Timing Analyzer and Powerplay Power Analyzer Tool in Quartus, see Appendix E for the results, 2 key parameters were extracted and used in the Verilator codes for each benchmark. These two parameters were the f_{\max} and the Average power dissipation. Table 4 below shows the critical parameters found for each benchmark using Verilator, while Table 5 illustrates the cost & critical parameters of the CPU availed from Quartus. F_{\max} was found to 83.8 MHz, for further detail, please refer to Appendix E. Avg Power from previous analysis with Cache integration was found to be approximately 250 mW.

Benchmark	Instruction Count	CC	CPI	ET	Energy	CC Memory Stalls	Estimated % functionality verified
IAXPY	813	2240	2.756	24.95 μ s	6.238 μ J	665	100
Sorter+CRC	-	-	-	-	-	-	-
Text Parser	1665	3953	2.374	449.58 μ s	112.4 μ J	2848	90%

Table 4

Note: Please refer to Appendix C, for the Sorter code. Only CRC was implemented successfully.

Cost Analysis	Number of Logic Elements	Total Registers	Total Memory Bits	Embedded Multiplier Elements	Number of M4Ks	F_{\max}	Total Thermal Power Diss.	Thermal Power by block type
	185820	37735	7680	0	-	83.8 MHz	250 mW	-

Table 5

Appendix A

As shown, the structure of the Instruction formats in the report ISA section, details of the fields are as below.

R-Type	Op [31:26]	Rd [25:21]	Rs1 [20:16]	Rs2[15:11]	Func[10:4]		Func2[3:0]
I-Type	Op [31:26]	Rd [25:21]	Rs1 [20:16]	IMM[15:4]			Func2[3:0]
B-Type	Op [31:26]	SIMM[25:21]	Rs1 [20:16]	Rs2[15:11]	SIMM[10:4]		Func2[3:0]
F-Type	Op [31:26]	Rd [25:21]	Rs1 [20:16]	Rs2[15:11]	Rs3[10:6]	Func3[5:4]	Func2[3:0]
SJ-Type	Op [31:26]	FJIMM[25:21]	Rs1 [20:16]	Rs2[15:11]	FJIMM[10:8]		FSIMM[7:0]
J-Type	Op [31:26]	JIMM[25:0]					

Jal-Type	Op [31:26]	Rd [25:21]	JIMM[20:0]
-----------------	------------	------------	------------

Table 1

Table 2 below shows the range, type and number of bits allocated for different Immediate fields:

IMM	Number of bits	Range	Type
I-type IMM[15:4]	12	[-2048,2047]	signed
SIMM[25:21,10:4]	12	[-2048,2047]	Signed
LIMM[15:4]	12	[-2048,2047]	signed
BIMM	12	[-2048,2047]	signed
FJIMM[25:21,10:8] FSIMM[7:0]	8 each	[-256,255]	signed
JIMM [25:0]	26	[-67108864, 67108863]	signed

Table 3, 4, & 5 below shows the functionality & versatility of the func2 field. It is used not only as control for the 1st ALU but also is the memory control, when using Load or Store operations. It can also be used to support different types of Branches:

Func2	0000	0001	0010	0011	0100	1001
Operation	Add	sub	Sll	Srl	sra	eq
Func2	0101	1111	0111	1000	1100	1110
Operation	or	and	xor	slt	mul	div

Table 3 shows the 1st ALU and Multiplication/Division Hardware control capabilities of func2 field

The last 3-bits of the func2 field are used as the control for the byte-addressable Data memory.

Func2 [2:0]	000	001	010	011	100
Load/store	Sw/lw	Sh/lh	Sb/lb	LHU	LBU

Table 4

Table 4 shows, the func2 field used as memory control, where it can load or store a word, half-word, byte, & unsigned versions of half-word and byte.

Func2	1001	1101	1000	0110
Branch	BEQ	BNE	BLT	BGE

Table 5 shows the different type of branches, ALU can accommodate, to make life simpler for the user

Table 6 below shows the Opcodes of each instruction format. The diversity of this ISA is highlighted here, as the same opcode is used while changing the func2 field to provide all sorts of different functions

supported by the ALU, for instance R-type can be used for add, sub, and, xor etc, where I-type can be used for andi, addi, subi, xori, lw, lb, lh etc. Similarly, func3 field supporting 4 instructions can be utilized to perform combination of operations on the same data, using the 2nd ALU, where it only supports 4 operations, but can be expanded if needed. Supported operations are add, sub, sll & srl.

Instruction Name	Instruction Format	Opcode (6-bits)
Add, Sub, Slt, Sgt etc.	R-type	110011
Addi, Subi, Slti, Sgti etc.	I-type	010011
Lw/lh/lb etc.	I-type	000011
Beq, bne, bge, blt	B-type	000100
Sw/sh/sb etc.	B-type	100011
Fma (mul&add) Fxs (xor&shift) etc.	Fused type	000111
Swj	SJ-Type	111001
Jump	J-type	000010

Table 6 shows the versatility of the respective formats with the help of func2 & func3 fields

Table 7 shows some key examples, of different instructions, commonly used in the benchmarks attached in Appendix C.

Instruction	Function	
Lw & Lb	$RF[Rd] = Mem[L-IMM + RF[rs1]]$	$RF[Rd] [7:0] = Mem[LIMM + RF[rs1]] [7:0]$
Sw & Sb	$Mem[L-IMM + RF[rs2]] = RF[Rs1]$	$Mem[L-IMM + RF[rs2]] [7:0] = RF[Rs1] [7:0]$
Fused Multiply & Add	$RF[Rd] = RF[rs1] * RF[rs2] + RF[rs3]$	
Fused Store Jump	$Mem[L-IMM + RF[rs2]] = RF[Rs1]$	$pc = pc + 4 + JIMM$
Bge	$RF[rs2] \geq RF[Rs1]$	$pc = pc + 4 + SIMM$
Immediate andi	$RF[Rd] = RF[rs1] + IMM$	
Jump	$Pc = pc + 4 + JIMM$	
Jump and link	$RF[Rd] = Pc + 4, Pc = pc + 4 + JIMM$	

Table 7 shows some example implementations of the instructions, with all having implicit PC increment

Appendix B

Config.sv

// This is a common configuration file for SystemVerilog macros to support the development

// in Computer Architecture / Organization courses at METU Northern Cyprus Campus.

// Ali Muhtaroglu

//

////////////////////////////////////

```

//      DRAM config      //
////////////////////////////////////

// DRAM address and data bus/word size in bits
`define DRAM_ADDRESS_SIZE  12
`define DRAM_WORD_SIZE    32

// Definition of DRAM Block Size in terms of
// # of Data Words. Block size is assumed same
// for the full memory hierarchy.
`define DRAM_BLOCK_SIZE    8

// Definition of DRAM latencies in clock cycles
`define DRAM_READ_ACCESS_TIME  8
`define DRAM_WRITE_ACCESS_TIME 12
`define DRAM_CYCLE_TIME      4

// Definition of DRAM file
`define DRAM_HEX "simple_test.hex.dat"

////////////////////////////////////
//      I-CACHE config    //
////////////////////////////////////

// CACHE index and data bus sizes for direct-mapped cache
`define ICACHE_INDEX    3

// number of blocks in cache
`define ICACHE_SIZE    2**`ICACHE_INDEX

```

```
`define ITAGMSB      `DRAM_ADDRESS_SIZE-1 //tag msb
```

```
`define ITAGLSB      8 //tag lsb
```

```
////////////////////////////////////
```

```
//      D-CACHE config      //
```

```
////////////////////////////////////
```

```
// CACHE index and data bus sizes for direct-mapped cache
```

```
`define DCACHE_INDEX  4
```

```
// number of blocks in cache
```

```
`define DCACHE_SIZE    2**`DCACHE_INDEX
```

```
`define DTAGMSB      `DRAM_ADDRESS_SIZE-1 //tag msb
```

```
`define DTAGLSB      9 //tag lsb
```

Constants.sv

```
// This is a common constants file for SystemVerilog macros to support the development
```

```
// in Computer Architecture / Organization courses at METU Northern Cyprus Campus.
```

```
// Ali Muhtaroglu
```

```
//
```

```
////////////////////////////////////
```

```
//      Constants      //
```

```
////////////////////////////////////
```

```
`define ON          1'b1
```

```
`define OFF         1'b0
```

```
`define ZERO        32'b0
```

Control.sv

```
module control (input logic [5:0] Op,
                output logic MemRead, MemWrite, MemToReg, ALUSrc, swj,
                output logic RegWrite, Branch, Jump, Second_alu_en
                );

    always_comb begin
        //Init Signals

        MemRead = 1'b0;
        MemWrite = 1'b0;
        MemToReg = 1'b0;
        ALUSrc = 1'b0;
        RegWrite = 1'b0;
        Branch = 1'b0;
        Jump = 1'b0;
        swj = 1'b0;
        Second_alu_en = 1'b0;

        case(Op)
        6'b110011: begin // R-type
            RegWrite = 1'b1;

            end

        6'b010011: begin // I-type
            ALUSrc = 1'b1;

            RegWrite = 1'b1;
            end

        6'b000011: begin // lw
            //RegDst = 1'b0;

            ALUSrc = 1'b1;
        end
    end
end
```

```

MemToReg = 1'b1;
RegWrite = 1'b1;
MemRead = 1'b1;
end

6'b100011: begin // sw

    ALUSrc = 1'b1;

    MemWrite = 1'b1;
end

6'b000100: begin // B-type

    Branch = 1'b1;

end

6'b000111: begin //fused mul and add

    RegWrite = 1'b1;
    Second_alu_en = 1'b1;
end

    6'b111001: begin //fused store and jump
        swj = 1;

        ALUSrc = 1'b1;

        MemWrite = 1'b1;

        Jump = 1'b1;
end

6'b000010: begin // jump

    Jump = 1'b1;

end

default: begin

    MemRead = 1'b0;
    MemWrite = 1'b0;
    MemToReg = 1'b0;

```

```
        ALUSrc = 1'b0;

        RegWrite = 1'b0;

        Branch = 1'b0;

        Jump = 1'b0;

        Second_alu_en = 1'b0;
```

```
    end
```

```
endcase
```

```
end
```

```
endmodule
```

```
cpu.sv
```

```
`include "./config.sv"
```

```
`include "./constants.sv"
```

```
module cpu(input logic clk,
            input logic    reset,
            input logic    cache_miss_stall,
            output logic [`DRAM_ADDRESS_SIZE-1:0] icache_address,
            output logic    icache_valid,
            input logic [31:0] icache_data_out,
            input logic [31:0] dcache_data_out,
            output logic [`DRAM_ADDRESS_SIZE-1:0] dcache_address,
            output logic [31:0] dcache_data_in,
            output logic [3:0] dcache_byte_en,
            output logic    dcache_rw,
            output logic    dcache_valid,
            output logic    regwritememwb);
```



```
logic [5:0] Op;
logic MemRead;
logic MemWrite;
logic MemToReg;
logic ALUSrc;
logic RegWrite;
logic Branch;
logic Jump;
logic ExMemRegWrite;
logic MemWbRegWrite;
logic Second_alu_en;
logic Alu2or1;
logic [4:0] IdExrs2;
logic [4:0] IdExrs1;
logic [4:0] ExMemrs3;
logic [4:0] ExMemrd;
logic [4:0] MemWbrd;
logic ExMemSecond_alu_en;
logic [1:0] ForwardA;
logic [1:0] ForwardB;
logic ForwardE;
logic [31:0] Instruction;
logic [31:0] IMM;
logic swj;
logic [4:0] rs11;
logic [4:0] rs22;
logic ForwardC;
logic ForwardD;
assign regwritememwb = MemWbRegWrite;
```

```
control u0(.Op(Op), .MemRead(MemRead), .MemWrite(MemWrite), .MemToReg(MemToReg),  
.ALUSrc(ALUSrc),
```

```
.RegWrite(RegWrite), .Branch(Branch), .Jump(Jump),
```

```
.Second_alu_en(Second_alu_en), .swj(swj));
```

```
datapath u1(.clk(clk), .reset(reset), .Op(Op), .MemRead(MemRead), .MemWrite(MemWrite),  
.MemToReg(MemToReg),
```

```
.ALUSrc(ALUSrc), .RegWrite(RegWrite), .Branch(Branch), .Jump(Jump),
```

```
.ExMemRegWrite(ExMemRegWrite),
```

```
.MemWbRegWrite(MemWbRegWrite), .IdExrs1(IdExrs1),
```

```
.IdExrs2(IdExrs2), .ExMemrs3(ExMemrs3), .ExMemrd(ExMemrd),
```

```
.MemWbrd(MemWbrd), .ExMemSecond_alu_en(ExMemSecond_alu_en), .ForwardA(ForwardA),
```

```
.ForwardB(ForwardB),
```

```
.Second_alu_en(Second_alu_en), .Instruction(Instruction), .IMM(IMM),
```

```
.ForwardE(ForwardE), .swj(swj), .ForwardC(ForwardC), .ForwardD(ForwardD),
```

```
.rs11(rs11), .rs22(rs22), .icache_data_out(icache_data_out),
```

```
.cache_miss_stall(cache_miss_stall), .icache_address(icache_address), .icache_valid(icache_valid),
```

```
.dcache_data_out(dcache_data_out),
```

```
.dcache_address(dcache_address), .dcache_data_in(dcache_data_in),
```

```
.dcache_byte_en(dcache_byte_en), .dcache_rw(dcache_rw), .dcache_valid(dcache_valid));
```

```
forwarding u3(.ExMemRegWrite(ExMemRegWrite), .MemWbRegWrite(MemWbRegWrite),  
.IdExrs1(IdExrs1), .IdExrs2(IdExrs2), .ExMemrs3(ExMemrs3),
```

```
.ExMemrd(ExMemrd), .MemWbrd(MemWbrd),
```

```
.ExMemSecond_alu_en(ExMemSecond_alu_en), .ForwardA(ForwardA), .ForwardB(ForwardB),
```

```
.ForwardE(ForwardE), .rs11(rs11), .rs22(rs22),
```

```
.ForwardC(ForwardC), .ForwardD(ForwardD));
```

```
imm_Gen u4(.Instruction(Instruction), .IMM(IMM));
```

```

endmodule

dcache_sram.sv

// This is the data sram unit for direct-mapped write-allocate, write-back data cache,
// developed by Ali Muhtaroglu in support of education for Computer Architecture / Organization
// courses
// at METU Northern Cyprus Campus.

`include "./config.sv"
`include "./constants.sv"

module data_cache_sram(input clk,
                      input we, // data request/command, e.g. RW
                      input [`DCACHE_INDEX-1:0] index, // cache index
                      input [`DRAM_WORD_SIZE-1:0] data_write[`DRAM_BLOCK_SIZE-1:0], //write port
                      output [`DRAM_WORD_SIZE-1:0] data_read[`DRAM_BLOCK_SIZE-1:0]); //read port

bit [`DRAM_WORD_SIZE-1:0] dcache_sram[`DCACHE_SIZE-1:0][`DRAM_BLOCK_SIZE-1:0];

initial begin
    for (int i=0; i<`DCACHE_SIZE; i++)
        dcache_sram[i] = '{default:0}; // initialize all bits in cache to 0
end

assign data_read = dcache_sram[index];

always_ff @(posedge(clk)) begin
    if (we)
        dcache_sram[index] <= data_write;
end

```

```
end
```

```
endmodule
```

```
dcache_tag
```

```
// This is the tag-ram unit for direct-mapped write-allocate, write-back data cache,
```

```
// developed by Ali Muhtaroglu in support of education for Computer Architecture / Organization  
courses
```

```
// at METU Northern Cyprus Campus.
```

```
`include "./config.sv"
```

```
`include "./constants.sv"
```

```
module data_cache_tag(input clk, //write clk
```

```
                    input we, // tag request/command, e.g. RW
```

```
                    input [`DCACHE_INDEX-1:0] index, // cache index
```

```
                    input valid_in,
```

```
                    input dirty_in,
```

```
                    input [`DTAGMSB:`DTAGLSB] tag_in, // input tag bits
```

```
                    output valid_out,
```

```
                    output dirty_out,
```

```
                    output [`DTAGMSB:`DTAGLSB] tag_out); // output tag bits
```

```
bit dcache_valid[`DCACHE_SIZE-1:0];
```

```
bit dcache_dirty[`DCACHE_SIZE-1:0];
```

```
bit [`DTAGMSB:`DTAGLSB] dcache_tag[`DCACHE_SIZE-1:0];
```

```
initial begin
```

```
    dcache_valid = '{default:0}; // initialize all valid bits in cache to 0
```

```
    dcache_dirty = '{default:0}; // initialize all dirty bits in cache to 0
```

```
    dcache_tag = '{default:0}; // initialize all tag bits in cache to 0
end
```

```
assign tag_out = dcache_tag[index];
assign valid_out = dcache_valid[index];
assign dirty_out = dcache_dirty[index];
```

```
always_ff @(posedge(clk)) begin
    if (we) begin
        dcache_tag[index] <= tag_in;
        dcache_valid[index] <= valid_in;
        dcache_dirty[index] <= dirty_in;
    end
end
```

```
endmodule
```

```
datapath.sv
```

```
`include "./config.sv"
```

```
`include "./constants.sv"
```

```
/* verilator lint_off UNPACKED */
```

```
/* verilator lint_off CASEX */
```

```
/* verilator lint_off CASEINCOMPLETE */
```

```
module datapath(input logic clk, reset, MemRead, MemWrite, MemToReg, Second_alu_en,
                input logic ALUSrc, RegWrite, Branch, Jump, swj,
                input logic [1:0] ForwardA,
                input logic [1:0] ForwardB,
                input logic [31:0] IMM,
                input logic ForwardE,
                input logic ForwardC,
```

```

input logic ForwardD,
output logic ExMemRegWrite,
output logic MemWbRegWrite,
output logic [4:0]      IdExrs1,
output logic [4:0]      IdExrs2,
output logic [4:0]      ExMemrd,
output logic [4:0]      ExMemrs3,
output logic [4:0]      MemWbrd,
output logic    ExMemSecond_alu_en,
output logic [31:0] Instruction,
output logic [4:0] rs11,
output logic [4:0] rs22,
output logic [5:0] Op,
// stall signal
input logic cache_miss_stall,
//icache signals
output logic [`DRAM_ADDRESS_SIZE-1:0] icache_address,
output logic icache_valid,
input logic [31:0] icache_data_out,
//dcache signals
input logic [31:0] dcache_data_out,
output logic [`DRAM_ADDRESS_SIZE-1:0] dcache_address,
output logic [31:0] dcache_data_in,
output logic [3:0] dcache_byte_en,
output logic dcache_rw,
output logic dcache_valid
);

```

parameter PCSTART = 2048; //starting address of instruction memory

```

logic PCsrc;

//mem control and aluctrl

logic [3:0] aluctr;

logic [2:0] memctrl;


always_comb begin

if ((Op == 6'b000011) | (Op == 6'b100011)) begin

aluctr = 4'b0;

memctrl = Ifld.instruction[2:0];

end

else if (Op == 6'b111001) begin

memctrl = 3'b0;

aluctr = 4'b0;

end

else begin

aluctr = Ifld.instruction[3:0];

memctrl = 3'b0;

end

end

// Instruction memory internal storage, input address and output data bus signals

logic [31:0] instmem_data;


// Data memory internal storage, input address and output data bus signals

logic [31:0] datamem_data;


//multiplier signals

logic [31:0] multiplier_out;

logic multiplier_done;

logic run_multiplier;

```

```

logic pause_pipeline;

//data and instruction memory

//internal signals
    logic [31:0] PC;                // Program Counter
    logic PCWrite;                  // PC Write Enable
    logic [31:0] OpA1;              // Alu input 1
    logic [31:0] OpB1;              // Alu input 2
    logic [3:0] ALUCtrl;
    logic [31:0] da;                //read data 1
    logic [31:0] db;                //read data 2
    logic [31:0] dc;                //read data 3 for mul and add
    logic [31:0] result;            //output of ALU
    logic [31:0] ForwardA_data;     //input to ALU 1
    logic [31:0] ForwardB_data;     //input to ALUSrc Mux

    logic [1:0] aluctrl2;           // func 3 from IMM gen for mul and add
    logic [31:0] OpA2;              //read data 1 for second ALU
    logic [31:0] OpB2;              //read data 2 for second ALU
    logic [31:0] ALUResult2;        //output of 2nd ALU

//Hazard Detection signals
    logic stall;
    logic flush;

    assign flush = IdEx.Jump | ExMem.Jump | (IdEx.Branch && result[0]) | (ExMem.Branch &&
ExMem.AluOut[0]);

// IF/ID register

```



```
logic Ifld_Write; //write enable signal for IFID
```

```
struct {  
    logic [31:0] instruction;  
    logic [31:0] PCincremented; // PC 7 bit?  
} Ifld;
```

```
// ID/EX Register
```

```
ID/EX struct {  
    logic [31:0] PCincremented;  
    logic MemRead; // control signals coming in  
    logic MemWrite;  
    logic MemToReg;  
    logic ALUSrc;  
    logic RegWrite;  
    logic Second_alu_en; //Enable for second ALU  
    logic Branch;  
    logic [31:0] da; // Read data 1 of register file  
    logic [31:0] db; // Read data 2 of register file  
    logic [31:0] SignExtend; // Sign Extended offset  
    logic [4:0] rs2; //rs2 address added for forwarding  
    logic [4:0] rs1; // rs1 address replacing rt  
    logic [4:0] rd; // rd address  
    logic [3:0] func2; // for control of ALU  
    logic [2:0] memctr; // to control memory  
    logic [4:0] rs3; // rs1 address replacing rt  
    logic Jump;  
    logic swj;
```

```

        logic [4:0] rdwb;
    } IdEx;

    // EX/MEM Register
    struct {
        logic MemRead;                                // control signals coming in
EX/MEM
        logic MemWrite;
        logic MemToReg;
        logic RegWrite;
        logic Branch;
        logic Second_alu_en;                            //Enable for second ALU
        logic [1:0] func3;                            //2 Lsbs of IMM generated for Mul add
        logic [3:0] func2;                            // might be needed to control
DataMem
        logic [2:0] memctr;    // to control memory
        logic [4:0] rd;    // Register Write Address (Rd or Rt)
        logic [31:0] AluOut;                            // output of ALU
        logic [31:0] ForwardB_data;    // WriteData to DataMemory
        logic [31:0] dc;                            // for input to the second ALU
        logic [31:0] jumpAddrss;
        logic [4:0] rs3;                            // rs1 address replacing rt
        logic Jump;
    } ExMem;

    // MEM/Wb Register
    struct {
        logic MemToReg;                                // control signals coming in MEM/WB
        logic RegWrite;

```

```

        logic [31:0] AluOut;                // output of ALU
        logic [31:0] DataMemRead;          // Data Memory Read Output
        logic [4:0] rd;    // Register Write Address (Rd or Rt)
    } MemWb;

//FETCH Stage ++++++
    // Instruction Memory Address
    assign icache_address = PC[`DRAM_ADDRESS_SIZE-1:0];
    //icash valid req
    always@ (posedge clk)begin
if (reset) begin
        icache_valid <= 0;
end else begin
        icache_valid <= 1;
end
end

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

    always@ (posedge clk)begin
        if (reset) begin
            Ifld.PCincremented <= 0;
            Ifld.instruction <= 0;
        end
        else if (pause_pipeline) begin

```

```

    end

    else if (Ifld_Write) begin
        Ifld.PCIncremented <= PC+4;
        Ifld.instruction <= (flush) ? 0 : instmem_data;
    end

end

assign Op = Ifld.instruction[31:26]; // Send OpCode to Control Unit

//FETCH END =====

//DECODE Stage ++++++

// Register File description
logic [31:0] RF[31:0]; // Register File array
logic [31:0] RF_WriteData; // write data

// Register File Read Logic
assign da = (Ifld.instruction[20:16] != 0) ? RF[Ifld.instruction[20:16]] : 0; //rs1
assign db = (Ifld.instruction[15:11] != 0) ? RF[Ifld.instruction[15:11]] : 0; //rs2
assign dc = (IdEx.rs3 != 0) ? RF[IdEx.rs3] : 0; //rs3
assign RF_WriteData = (MemWb.MemToReg) ? MemWb.DataMemRead : MemWb.AluOut;

assign stall = ((IdEx.MemRead && ((IdEx.rd == Ifld.instruction[20:16]) ||
(IdEx.rd == Ifld.instruction[15:11]))) || (IdEx.Second_alu_en && ((IdEx.rd == Ifld.instruction[20:16]) ||
(IdEx.rd == Ifld.instruction[15:11]) || (IdEx.rd == Ifld.instruction[10:6]))));

assign PCWrite = ~stall;
assign Ifld_Write = ~stall;

```

```
//func 3 field assignment from the input IMM
```

```
assign aluctrl2[1:0] = IMM[5:4];
```

```
always@ (posedge clk)begin
```

```
if (reset)begin
```

```
    IdEx.MemRead          <= 1'b0;
```

```
        IdEx.MemWrite      <= 1'b0;
```

```
        IdEx.MemToReg      <= 1'b0;
```

```
        IdEx.ALUSrc        <= 1'b0;
```

```
        IdEx.RegWrite      <= 1'b0;
```

```
        IdEx.Branch        <= 1'b0;
```

```
        IdEx.PCincremented <= 0;
```

```
        IdEx.Second_alu_en <= 1'b0;
```

```
        IdEx.da <= 0;
```

```
        IdEx.db <= 0;
```

```
        // IMM gen output
```

```
        IdEx.SignExtend <= 32'b0;
```

```
        IdEx.rd <= 5'b0;
```

```
        IdEx.rs1 <= 5'b0;
```

```
        IdEx.rs2 <= 5'b0;
```

```
        IdEx.func2 <= 4'b0;
```

```
        IdEx.rs3 <= 5'b0;
```

```
        IdEx.memctr <= 3'b0;
```

```
        IdEx.Jump <= 0;
```

```
        // fused swj
```

```
        IdEx.swj <= 0;
```

```
        //forwarding rd
```

```
        IdEx.rdwb <= 0;
```

end

else if (stall | flush) begin

// Control signals coming to ID/EX

```
IdEx.MemRead      <= 1'b0 ;
IdEx.MemWrite      <= 1'b0 ;
IdEx.MemToReg      <= 1'b0 ;
IdEx.ALUSrc        <= 1'b0 ;
IdEx.RegWrite      <= 1'b0 ;
IdEx.Branch        <= 1'b0 ;
IdEx.PCIncremented <= 0 ;
IdEx.Second_alu_en <= 1'b0 ;
IdEx.da <= 0 ;
IdEx.db <= 0 ;
```

// IMM gen output

```
IdEx.SignExtend [31:0] <= 0;
IdEx.rd <= 5'b0;
IdEx.rs1 <= 5'b0;
IdEx.rs2 <= 5'b0;
IdEx.func2 <= 4'b0;
IdEx.rs3 <= 5'b0;
IdEx.memctr <= 3'b0;
IdEx.Jump <= Jump;
// fused swj
IdEx.swj <= swj;
//forwarding
IdEx.rdw b <= ExMem.rd;
```

end

else if (pause_pipeline) begin

```

end

else begin

    // Control signals coming to ID/EX

    IdEx.MemRead      <= MemRead;

    IdEx.MemWrite     <= MemWrite;

    IdEx.MemToReg     <= MemToReg;

    IdEx.ALUSrc       <= ALUSrc;

    IdEx.RegWrite     <= RegWrite;

    IdEx.Branch       <= Branch;

    IdEx.PCincremented <= Ifld.PCincremented;

    IdEx.Second_alu_en <= Second_alu_en;

    IdEx.da <= (ForwardC) ? MemWb.AluOut : da;

    IdEx.db <= (ForwardD) ? MemWb.AluOut : db;


    // IMM gen output

    IdEx.SignExtend [31:0] <= IMM[31:0];

    IdEx.rd <= Ifld.instruction[25:21];

    IdEx.rs1 <= Ifld.instruction[20:16];

    IdEx.rs2 <= Ifld.instruction[15:11];

    IdEx.func2 <= aluctr;

    IdEx.rs3 <= Ifld.instruction[10:6];

    IdEx.memctr <= memctr;

    IdEx.Jump <= Jump;

    // fused swj

    IdEx.swj <= swj;

    //forwarding

    IdEx.rdw b <= ExMem.rd;

end

```

end

// ++++++

logic [31:0] jumpAddrsss;

assign jumpAddrsss = (IdEx.swj) ?

IdEx.PCincremented+{{{(16){IdEx.SignExtend[31]}},IdEx.SignExtend[31:16]}:IdEx.PCincremented +
IdEx.SignExtend; //Jump address

// DECODE END =====

// EXECUTE Stage ++++++

//Forwarding MUXes

always_comb

begin

casex(ForwardA)

2'b01: ForwardA_data = RF_WriteData;

2'b10: ForwardA_data = ExMem.AluOut;

2'b11: ForwardA_data = da;

default: ForwardA_data = IdEx.da;

endcase

end

always_comb

begin

casex(ForwardB)

2'b01: ForwardB_data = RF_WriteData;

2'b10: ForwardB_data = ExMem.AluOut;


```

        2'b11: ForwardB_data = db;

        default: ForwardB_data = IdEx.db;

    endcase

end

// Data 1 = AluIn 1
assign OpA1 = ForwardA_data;


// Input of ALU Control
//assign Function = IdEx.SignExtend[5:0];


// Changing the names to get rid of the dot, outputs from the datapath to other modules
assign Instruction = Ifld.instruction;


assign IdExrs1 = IdEx.rs1;
assign IdExrs2 = IdEx.rs2;
assign ExMemSecond_alu_en = ExMem.Second_alu_en;
assign ExMemrs3 = ExMem.rs3;
assign ExMemrd = ExMem.rd;
assign MemWbrd = MemWb.rd;
assign ExMemRegWrite = ExMem.RegWrite;    //
assign MemWbRegWrite = MemWb.RegWrite; //control inputs for forwarding
assign rs11 = Ifld.instruction[20:16];
assign rs22 = Ifld.instruction[15:11];
////////////////////////////////////

// AluSrc Mux
always_comb begin
    if (IdEx.swj) begin
        OpB1 = {{{16}{IdEx.SignExtend[15]}},IdEx.SignExtend[15:0]};
    end
end

```

```

end

else if(IdEx.ALUSrc) // If it is not R type use Sign Extended
    OpB1 = IdEx.SignExtend;
else
    OpB1 = ForwardB_data; // Else Data 2 = AluIn 1
end

assign ALUCtrl = IdEx.func2[3:0];

//
assign pause_pipeline = ((run_multiplier & ~multiplier_done) | cache_miss_stall) ? 1'b1:1'b0;
//mult
multiplier multiplier (
.clk(clk), .reset(reset),
.a(OpA1[15:0]),
.b(OpB1[15:0]),
.start(run_multiplier),
.product(multiplier_out),
.done(multiplier_done)
);

assign run_multiplier = (IdEx.func2[3:0] == 4'b1100);
// ALU ONE-----
always_comb
begin
    case(ALUCtrl)
        4'b0000: result = OpA1 + OpB1; //add
    endcase
end

```

```

4'b0001: result = OpA1 - OpB1; //sub for bne
and beq as well, with two zero flags

```

```

4'b0010: result = OpA1 << OpB1[4:0]; //sll

```

```

4'b0011: result = OpA1 >> OpB1[4:0]; //srl

```

```

4'b0100: result = $signed(OpA1) >>> OpB1[4:0]; //sra

```

```

4'b0101: result = OpA1 | OpB1 ; //or

```

```

4'b0111: result = OpA1 ^ OpB1; //xor

```

```

4'b1000: result = OpA1 < OpB1 ? 1:0; //slt so also for blt

```

```

//4'b1100: result = OpA1[15:0] * OpB1[15:0]; //Mul

```

```

//4'b1110: result = OpA1 / OpB1; //Div

```

```

4'b1001: result =(OpA1 == OpB1) ? 1 : 0; //Equal

```

```

4'b1101: result =(OpA1 != OpB1) ? 1 : 0; //not Equal

```

```

4'b0110: result =(OpA1 >= OpB1) ? 1 : 0; //Greater Than or equal

```

```

4'b1111: result = OpA1 & OpB1; //and

```

```

4'b1100: result = multiplier_out;

```

```

default: result = OpA1 + OpB1 ;

```

```

endcase

```

```

end

```

```

// ALU ONE END-----

```

```

always@ (posedge clk)begin

```

```

if (reset) begin

```

```

    // Control signals from ID/EX Stage

```

```

    ExMem.RegWrite <= 0;

```

```

    ExMem.MemRead <= 0;

```

```

    ExMem.MemWrite <= 0;

```

```

    ExMem.MemToReg <= 0;

```

```

    ExMem.Branch <= 0;

```

```

    ExMem.ForwardB_data <= 0; //outout of AluSrc

```

```

    ExMem.dc <= 0;

    ExMem.func2 <= 0;

    ExMem.func3 <= 0;

    ExMem.Second_alu_en <= 0;


    // Branch Address
    ExMem.jumpAddrss <= 0;


    // ALUOut to ExMem
    ExMem.AluOut <= 0;

    ExMem.rd <= 0;

    ExMem.rs3 <= 0;

    ExMem.Jump <= 0;


    //mem control
    ExMem.memctr <= 0;
end
else if (pause_pipeline) begin
end
else begin
    // Control signals from ID/EX Stage
    ExMem.RegWrite <= IdEx.RegWrite;

    ExMem.MemRead <= IdEx.MemRead;

    ExMem.MemWrite <= IdEx.MemWrite;

    ExMem.MemToReg <= IdEx.MemToReg;

    ExMem.Branch <= IdEx.Branch;

    ExMem.ForwardB_data <= ForwardB_data;    //outout of AluSrc

```

```

    ExMem.dc <= dc;

    ExMem.func2 <= IdEx.func2;

    ExMem.func3 <= aluctrl2;

    ExMem.Second_alu_en <= IdEx.Second_alu_en;


    // Branch Address

    ExMem.jumpAddrss <= jumpAddrsss;


    // ALUOut to ExMem

    ExMem.AluOut <= result;

    ExMem.rd <= IdEx.rd;

    ExMem.rs3 <= IdEx.rs3;

    ExMem.Jump <= IdEx.Jump;


    //mem control

    ExMem.memctr <= IdEx.memctr;

    end

end

// EXECUTE END =====

// MEM STAGE ++++++

    // Data Memory Read Logic

    assign dcache_valid = ExMem.MemRead || ExMem.MemWrite;

    assign dcache_rw = (ExMem.MemWrite) ? 1'b1:1'b0;                                // read 0

write 1

```

```

        assign dcache_address = ExMem.AluOut[`DRAM_ADDRESS_SIZE-1:0];

/* verilator lint_off LATCH */

always_comb begin
//if (ExMem.MemRead) begin
caseex(ExMem.memctr)
    3'b000: begin //LW
        datamem_data = dcache_data_out;
        end
    3'b001: begin //LH
        datamem_data[31:16] = 16'b0;
        datamem_data[15:0] = dcache_data_out[15:0];
        end
    3'b010: begin //LB
        datamem_data[31:8] = 24'b0;
        datamem_data[7:0] = dcache_data_out[7:0];
        end
    3'b011: begin //LHU
        datamem_data[31:16] = dcache_data_out[31:16];
        datamem_data[15:0] = 16'b0;
        end
    3'b100: begin //LBU
        datamem_data[31:24] = 8'b0;
        datamem_data[23:16] = dcache_data_out[23:16];
        datamem_data[15:0] = 16'b0;
        end
    3'b101: begin //LBU
        datamem_data = 32'bx;
        end
    3'b110: begin //LBU

```

```

        datamem_data = 32'bx;
        end
        3'b111: begin //LBU
            datamem_data = 32'bx;
            end
        endcase
    end
//      end
//      end
/* verilator lint_off LATCH */

// ALU TWO-----

assign OpA2 = ExMem.AluOut;
assign OpB2 = (ForwardE) ? RF_WriteData : ExMem.dc;

always_comb begin
    if (ExMem.Second_alu_en) begin
        case(ExMem.func3)
            2'b00: ALUResult2 = OpA2 + OpB2;           //add
            2'b01: ALUResult2 = OpA2 - OpB2;           //sub
            2'b10: ALUResult2 = OpA2 >> OpB2[4:0];      //srl
            2'b11: ALUResult2 = OpA2 << OpB2[4:0];      //sll
            default: ALUResult2 = OpA2 + 0;
        endcase
    end
    else ALUResult2 = OpA2;
end

// ALU TWO END=====

```

```

always@ (posedge clk)begin
if (reset) begin
// Control Signals
    MemWb.RegWrite <= 0;
    MemWb.MemToReg <= 0;
    MemWb.AluOut <= 0;          //output from the mux of Alu2or1 to memwb
    MemWb.rd <= 0;
    MemWb.DataMemRead <= 0;
end
else if (pause_pipeline) begin
end
else begin
    // Control Signals
    MemWb.RegWrite <= ExMem.RegWrite;
    MemWb.MemToReg <= ExMem.MemToReg;
    MemWb.AluOut <= ALUResult2;      //output from the mux of Alu2or1 to memwb
    MemWb.rd <= ExMem.rd;
    MemWb.DataMemRead <= datamem_data;
end
end

//*****//

// Data Memory Write Data
always @(posedge clk) begin
if (ExMem.MemWrite && !pause_pipeline) begin
    dcache_data_in <= ExMem.ForwardB_data;
end
end

end

```



```

//byte enable cache
always_comb begin
    if(ExMem.memctr == 3'b000) dcache_byte_en = 4'b1111;
    else if(ExMem.memctr == 3'b001) dcache_byte_en = 4'b0011;
    else if (ExMem.memctr == 3'b010) dcache_byte_en = 4'b0001;
    else dcache_byte_en = 4'b0000;
end

```

```

// Data Memory Write Address

```

```

// MEM STAGE END =====

```

```

// WRITE BACK STAGE +++++

```

```

// Register File Write Logic
always @(posedge clk) begin
    int i;
    if(reset)begin
        for (i = 0; i < 32; i = i + 1)
            RF[i] <= 0;
    end
    else begin
        if (MemWb.RegWrite)
            RF[MemWb.rd] <= RF_WriteData; //MemtoReg Mux
    end
end

```

```
end
```

```
// WRITE BACK END =====
```

```
assign PCsrc = ExMem.Jump | (ExMem.Branch && ExMem.AluOut[0]);
```

```
// PC LOGIC ++++++
```

```
always@ (posedge clk)begin
```

```
    if(reset)
```

```
        PC <= PCSTART;
```

```
    else if (pause_pipeline) begin
```

```
    end
```

```
    else
```

```
        if (PCWrite) begin
```

```
            case(PCsrc)
```

```
                1'b0: PC <= PC+4;
```

```
                1'b1: PC <= ExMem.jumpAddrss;
```

```
            endcase
```

```
        end
```

```
    end
```

```
// PC LOGIC END =====
```

```
endmodule // datapath
```

```
/* verilator lint_on UNPACKED */
```

```
/* verilator lint_on CASEX */
```

```
/* verilator lint_on CASEINCOMPLETE */
```

Dcache_controller.sv

// This is a direct-mapped write-allocate, write-back data cache controller fsm,

// developed by Ali Muhtaroglu in support of education for Computer Architecture / Organization courses

// at METU Northern Cyprus Campus.

`include "./config.sv"

`include "./constants.sv"

module dcache_controller (input clk, reset,

(CPU->cache) input [`DRAM_ADDRESS_SIZE-1:0] dcache_address, // cpu request address

(CPU->cache) input [`DRAM_WORD_SIZE-1:0] dcache_data_in, // cpu request data

enable (CPU->cache) input [`DRAM_WORD_SIZE/8-1:0] dcache_byte_en, // cpu request byte

input dcache_rw, // cpu R/W request (CPU->cache)

input dcache_valid, // cpu request valid (CPU->cache)

input [`DRAM_WORD_SIZE-1:0] mem_data_in[`DRAM_BLOCK_SIZE-1:0], // memory read data (memory->cache)

input mem_ready, // memory read data ready (memory->cache)

output [`DRAM_ADDRESS_SIZE-1:0] mem_address, // cache request address (cache->memory)

output [`DRAM_WORD_SIZE-1:0] mem_data_out[`DRAM_BLOCK_SIZE-1:0], // memory write data (cache->memory)

output mem_rw, // R/W request to memory (cache->memory)

output mem_valid, // request to memory valid (cache->memory)

output [`DRAM_WORD_SIZE-1:0] dcache_data_out, // data to CPU (cache->CPU)

```

                                output                                dcache_data_ready // data to CPU ready
(cache->CPU)
                                );

```

```

// log2 function quite useful for parametric design

```

```

function automatic int log2 (input int n);

```

```

    if (n <=1) return 1; // abort function

```

```

    log2 = 0;

```

```

    while (n > 1) begin

```

```

        n = n/2;

```

```

        log2++;

```

```

    end

```

```

// changed the following line for quartus

```

```

// endfunction; // log2

```

```

    // into:

```

```

    endfunction

```

```

// one-hot state assignment, and definition of next state and current state variables

```

```

enum logic [2:0] {compare_tag=3'b001, allocate=3'b010, write_back= 3'b100} nstate, cstate;

```

```

// interface signals to tag memory

```

```

logic          dcache_valid_read;

```

```

logic          dcache_dirty_read;

```

```

logic [ `DTAGMSB:`DTAGLSB] dcache_tag_read;

```

```

logic          dcache_valid_write;

```

```

logic          dcache_dirty_write;

```

```

logic [ `DTAGMSB:`DTAGLSB] dcache_tag_write;

```

```

logic [ `DCACHE_INDEX-1:0] dcache_tag_index;

```

```

logic          dcache_tag_we;

```

```

// interface signals to cache sram memory
logic [`DRAM_WORD_SIZE-1:0] dcache_sram_read[`DRAM_BLOCK_SIZE-1:0];
logic [`DRAM_WORD_SIZE-1:0] dcache_sram_write[`DRAM_BLOCK_SIZE-1:0];
logic [`DCACHE_INDEX-1:0] dcache_sram_index;
logic          dcache_sram_we;

// temporary variables for cache controller output to CPU
logic [`DRAM_WORD_SIZE-1:0] dcache_data_to_cpu;
logic          dcache_data_to_cpu_ready;

// temporary variables for memory requests
logic [`DRAM_ADDRESS_SIZE-1:0] mem_request_address;
logic [`DRAM_WORD_SIZE-1:0] mem_request_data[`DRAM_BLOCK_SIZE-1:0];
logic          mem_request_rw;
logic          mem_request_valid;

// temporary variables for writing bytes to cache
logic [`DRAM_WORD_SIZE-1:0] dcache_sram_write_temp;

assign mem_data_out = mem_request_data;
assign mem_address = mem_request_address;
assign mem_rw = mem_request_rw;
assign mem_valid = mem_request_valid;
assign dcache_data_out = dcache_data_to_cpu;
assign dcache_data_ready = dcache_data_to_cpu_ready;

// replace the following for lines for quartus

```

```

// genvar i;

// generate

// for (i=0; i < `DRAM_WORD_SIZE/8; i++) begin

    // assign dcache_sram_write_temp[(i+1)*8-1:i*8] = (dcache_byte_en[i]==1) ?
dcache_data_in[(i+1)*8-1:i*8] :
dcache_sram_read[dcache_address[log2(`DRAM_BLOCK_SIZE)+1:2]][(i+1)*8-1:i*8];

    // with the following lines:

    assign dcache_sram_write_temp[7:0] = (dcache_byte_en[0]==1) ? dcache_data_in[7:0] :
dcache_sram_read[dcache_address[log2(`DRAM_BLOCK_SIZE)+1:2]][7:0];

    assign dcache_sram_write_temp[15:8] = (dcache_byte_en[1]==1) ? dcache_data_in[15:8] :
dcache_sram_read[dcache_address[log2(`DRAM_BLOCK_SIZE)+1:2]][15:8];

    assign dcache_sram_write_temp[23:16] = (dcache_byte_en[2]==1) ? dcache_data_in[23:16] :
dcache_sram_read[dcache_address[log2(`DRAM_BLOCK_SIZE)+1:2]][23:16];

    assign dcache_sram_write_temp[31:24] = (dcache_byte_en[3]==1) ? dcache_data_in[31:24] :
dcache_sram_read[dcache_address[log2(`DRAM_BLOCK_SIZE)+1:2]][31:24];

    // remove the following 2 lines for quartus

// end

//endgenerate

always_comb begin

// default settings for all signals

nstate = cstate;

dcache_data_to_cpu = '{default:0};

dcache_data_to_cpu_ready = '0;

dcache_valid_write = '0;

dcache_dirty_write = '0;

dcache_tag_write = '{default:0};

mem_request_valid = '0;

dcache_tag_we = '0;

```

```

// direct mapped cache index for cache tag memory and cache sram portion

dcache_tag_index =
dcache_address[log2(`DRAM_BLOCK_SIZE)+1+`DCACHE_INDEX:log2(`DRAM_BLOCK_SIZE)+2];

dcache_sram_index =
dcache_address[log2(`DRAM_BLOCK_SIZE)+1+`DCACHE_INDEX:log2(`DRAM_BLOCK_SIZE)+2];


// default is reading cache line

dcache_sram_we = '0;


// access only the correct word and byte of the cache sram block when writing

dcache_sram_write = dcache_sram_read;

dcache_sram_write[dcache_address[log2(`DRAM_BLOCK_SIZE)+1:2]] = dcache_sram_write_temp;


// access only the correct word of the cache sram block when reading

dcache_data_to_cpu = dcache_sram_read[dcache_address[log2(`DRAM_BLOCK_SIZE)+1:2]];


// main memory request address is a copy of cpu request address to cache

mem_request_address = dcache_address;


// when requesting write to main memory use cache sram content

mem_request_data = dcache_sram_read;

mem_request_rw = '0;


// Cache FSM with four states:


case(cstate)

    // compare tag state

    compare_tag : begin

        /*If there is a CPU request, then compare cache tag*/

```

```

/*cache hit (tag match and cache entry is valid)*/
    if (dcache_valid && (dcache_address[`DTAGMSB:`DTAGLSB] == dcache_tag_read) &&
dcache_valid_read) begin
        dcache_data_to_cpu_ready = '1;

/*write hit*/
if (dcache_rw) begin
    /*read/modify cache line*/
    dcache_tag_we = '1; dcache_sram_we = '1;

    /*no change in tag*/
    dcache_tag_write = dcache_tag_read;
    dcache_valid_write = '1;

    /*cache line is dirty*/
    dcache_dirty_write = '1;
end

/*finished*/
nstate = compare_tag;
    end

    /*cache miss*/
    else if (dcache_valid) begin
/*generate new tag*/
dcache_tag_we = '1;
dcache_valid_write = '1;

/*new tag*/
dcache_tag_write = dcache_address[`DTAGMSB:`DTAGLSB];

/*cache line is dirty if write*/
dcache_dirty_write = dcache_rw;

/*generate memory request on miss*/

```



```

mem_request_valid = '1;

/*compulsory miss or miss with clean block*/
if (dcache_valid_read == 1'b0 || dcache_dirty_read == 1'b0)

    /*wait till a new block is allocated*/

    nstate = allocate;

else begin

    /*miss with dirty line*/

    /*write back address*/

    mem_request_address = {dcache_tag_read, dcache_address[`DTAGLSB-1:0]};

    mem_request_rw = '1;

    /*wait till write is completed*/

    nstate = write_back;

end

end

end

// state for allocating a new cache line before proceeding
allocate: begin

    /*memory controller has responded*/

    if (mem_ready) begin

        /*re-compare tag for write miss (need modify correct word)*/

        nstate = compare_tag;

        dcache_sram_write = mem_data_in;

        /*update cache line data*/

        dcache_sram_we = '1;

    end

end

// state writing back a dirty cache line before proceeding
write_back : begin

    /*write back is completed*/

```

```

        if (mem_ready) begin
/*issue new memory request (allocating a new line)*/
            mem_request_valid = '1;
            mem_request_rw = '0;
            nstate = allocate;
        end
    end

    default: nstate = cstate;
endcase
end

always_ff @(posedge(clk)) begin
    if (reset)
        cstate <= compare_tag; //reset to compare tag
    else
        cstate <= nstate;
end

```

// data cache tag ram and sram instantiations:

```

data_cache_tag data_cache_tag (
    .clk      (clk),
    .we       (dcache_tag_we),
    .index    (dcache_tag_index),
    .valid_in  (dcache_valid_write),
    .dirty_in  (dcache_dirty_write),
    .tag_in    (dcache_tag_write),
    .valid_out (dcache_valid_read),
    .dirty_out (dcache_dirty_read),
    .tag_out   (dcache_tag_read)
)

```

```
);
```

```
data_cache_sram data_cache_sram (  
    .clk      (clk),  
    .we       (dcache_sram_we),  
    .index    (dcache_sram_index),  
    .data_write (dcache_sram_write),  
    .data_read  (dcache_sram_read)  
);
```

```
Endmodule
```

```
Dram.sv
```

```
// It is a simple byte-writeable main memory bus design with programmable  
// read and write (clk) latency, which has been developed by Ali Muhtaroglu  
// in support of education for Computer Architecture / Organization courses  
// at METU Northern Cyprus Campus.
```

```
`include "./config.sv"
```

```
`include "./constants.sv"
```

```
module dram (  
  
    input
```

```
        clk, reset,
```

```
    input [`DRAM_ADDRESS_SIZE-1:0] address,
```

```
    inout wire [`DRAM_WORD_SIZE-1:0] data,
```

```
    input      read_enable,
```

```
    input      write_enable,
```

```
    output     write_data_enable,
```

```
    output     read_data_enable,
```

```

        output          acknowledge

    );

    /* verilator lint_off LITENDIAN */

    logic [(`DRAM_READ_ACCESS_TIME+((`DRAM_BLOCK_SIZE-1)*`DRAM_CYCLE_TIME))-2:0]
    read_acknowledge_delay_line;

    logic [(`DRAM_WRITE_ACCESS_TIME+((`DRAM_BLOCK_SIZE-1)*`DRAM_CYCLE_TIME))-2:0]
    write_acknowledge_delay_line;

    /* verilator lint_on LITENDIAN */

    logic int_read_acknowledge;

    logic int_write_acknowledge;

`ifdef DRAM_WRITE_ACCESS_TIME

    assign write_data_enable = write_acknowledge_delay_line[`DRAM_WRITE_ACCESS_TIME-
`DRAM_CYCLE_TIME-2];

`else

    assign write_data_enable = write_enable;

`endif

`ifdef DRAM_READ_ACCESS_TIME

    assign read_data_enable = read_acknowledge_delay_line[`DRAM_READ_ACCESS_TIME-
`DRAM_CYCLE_TIME+1];

`else

    assign read_data_enable = read_enable;

`endif

dram_array dram_array (
    .clk      (clk),
    .address  (address),
    .data     (data),

```

```

        .wren      (write_data_enable)
    );

    // acknowledge logic for generating memory latency

    // removed next 2 lines for quartus

    // `ifdef DRAM_READ_ACCESS_TIME
    // if ((`DRAM_READ_ACCESS_TIME+`DRAM_BLOCK_SIZE) > 0) begin
        always @(posedge clk)
            begin
                if (reset || !read_enable)
                    read_acknowledge_delay_line <= 'b0;

                else
                    begin
                        if((`DRAM_READ_ACCESS_TIME+`DRAM_BLOCK_SIZE) > 1) begin
                            /* verilator lint_off WIDTH */

                            /* verilator lint_off SELRANGE */

                                read_acknowledge_delay_line <=
{read_acknowledge_delay_line[(`DRAM_READ_ACCESS_TIME+((`DRAM_BLOCK_SIZE-
1)*`DRAM_CYCLE_TIME))-2:0], read_enable};

                            /* verilator lint_on SELRANGE */
                            /* verilator lint_on WIDTH */

                        end else begin

                            /* verilator lint_off WIDTH */

                                read_acknowledge_delay_line <= read_enable;

                            /* verilator lint_on WIDTH */

                        end
                    end
            end
    end // always @ (posedge clk or negedge read_enable)

```

```

    assign int_read_acknowledge =
read_acknowledge_delay_line[(`DRAM_READ_ACCESS_TIME+((`DRAM_BLOCK_SIZE-
1)*`DRAM_CYCLE_TIME))-2];

    // removed the following 3 lines for quartus

        // end else begin

        // assign int_read_acknowledge = read_enable;

// end // else: !if(`DRAM_READ_ACCESS_TIME > 0)

    // removed following 3 lines for quartus

    //`else

// assign int_read_acknowledge = read_enable;

//`endif


    // removed next 2 lines for quartus

// `ifdef DRAM_WRITE_ACCESS_TIME

// if((`DRAM_WRITE_ACCESS_TIME+`DRAM_BLOCK_SIZE) > 0) begin

    always @(posedge clk)

        begin

            if (reset || !write_enable)

                write_acknowledge_delay_line <= 'b0;

            else

                begin

                    if((`DRAM_WRITE_ACCESS_TIME+`DRAM_BLOCK_SIZE) > 1) begin

                        /* verilator lint_off WIDTH */

                        /* verilator lint_off SELRANGE */

                            write_acknowledge_delay_line <=
{write_acknowledge_delay_line[(`DRAM_WRITE_ACCESS_TIME+((`DRAM_BLOCK_SIZE-
1)*`DRAM_CYCLE_TIME))-2:0], write_enable};

                        /* verilator lint_on WIDTH */

                        /* verilator lint_on SELRANGE */

                    end else begin

```

```

        /* verilator lint_off WIDTH */

        write_acknowledge_delay_line <= write_enable;

        /* verilator lint_on WIDTH */

        end

    end

end // always @ (posedge clk or negedge read_enable)

assign int_write_acknowledge =
write_acknowledge_delay_line[(`DRAM_WRITE_ACCESS_TIME+((`DRAM_BLOCK_SIZE-
1)*`DRAM_CYCLE_TIME))-2];

// remove the following 6 lines for quartus

    // end else begin // if (`DRAM_WRITE_ACCESS_TIME > 0)

    // assign int_write_acknowledge = write_enable;

// end

// `else // !`ifdef DRAM_WRITE_ACCESS_TIME

// assign int_write_acknowledge = write_enable;

// `endif

assign acknowledge = int_read_acknowledge || int_write_acknowledge;

endmodule

dram_array.sv

// This memory core has been developed by Ali Muhtaroglu for education
// in support of Computer Architecture / Organization courses at METU
// Northern Cyprus Campus.

`include "./config.sv"

`include "./constants.sv"

module dram_array (

```

```

        input [`DRAM_ADDRESS_SIZE-1:0] address,

        input          clk,

        inout wire [`DRAM_WORD_SIZE-1:0] data,

        input          wren

    );

    logic [7:0]          mem[0:2*(`DRAM_ADDRESS_SIZE)-1];
    logic [`DRAM_WORD_SIZE-1:0]    din;

    assign din = data;

    /* verilator lint_off WIDTH */
    // removed the following three lines for Quartus and replaced with following 4 lines:
    //  genvar i;
    //  generate
    //      for (i=0; i < `DRAM_WORD_SIZE/8; i++) begin
        assign data[7:0] = (wren==0) ? mem[address] : (wren==1) ? 8'bz : 8'bx;
        assign data[15:8] = (wren==0) ? mem[address+1] : (wren==1) ? 8'bz : 8'bx;
        assign data[23:16] = (wren==0) ? mem[address+2] : (wren==1) ? 8'bz : 8'bx;
        assign data[31:24] = (wren==0) ? mem[address+3] : (wren==1) ? 8'bz : 8'bx;

        // removed the following two lines for quartus:
    //          end
    //      endgenerate
    /* verilator lint_on WIDTH */

    //  assign data = (wren==0) ? mem[address] : (wren==1) ? `DRAM_WORD_SIZE'bz :
    //  `DRAM_WORD_SIZE'bx;

```



```

// removed following 3 lines for Quartus
// genvar j;
// generate
//   for (j=0; j < `DRAM_WORD_SIZE/8; j++) begin
//       always_ff @(posedge clk)
//           if (wren) begin
//               mem[address+j] <= din[(j+1)*8-1:j*8];
//           // changed for Quartus as the following 4 lines:
//               mem[address] <= din[7:0];
//               mem[address+1] <= din[15:8];
//               mem[address+2] <= din[23:16];
//               mem[address+3] <= din[31:24];
//           end
//       // removed following 2 lines for Quartus
//       end
//   endgenerate
/* verilator lint_on WIDTH */

`ifdef DRAM_HEX
    initial
        $readmemh(`DRAM_HEX, mem);
`endif

endmodule

dram_interface.sv

// It is a simple main memory bus interface, which has been developed by Ali Muhtaroglu
// in support of education for Computer Architecture / Organization courses
// at METU Northern Cyprus Campus.

```

```

`include "./config.sv"

`include "./constants.sv"

module dram_interface (
    input                clk, reset,
    input [`DRAM_ADDRESS_SIZE-1:0] bus_address_to_mem,
    input [`DRAM_WORD_SIZE-1:0]  bus_data_to_mem[`DRAM_BLOCK_SIZE-1:0],
    input                bus_read_enable,
    input                bus_write_enable,
    output [`DRAM_WORD_SIZE-1:0] bus_data_from_mem[`DRAM_BLOCK_SIZE-1:0],
    output               acknowledge_from_mem
);

function automatic int log2 (input int n);
    if (n <=1) return 1; // abort function
        log2 = 0;
        while (n > 1) begin
            n = n/2;
            log2++;
        end
// endfunction;
// fixed for Quartus as:
endfunction

wire [`DRAM_WORD_SIZE-1:0]          data_pin;
logic [`DRAM_WORD_SIZE-1:0]         data_from_mem[`DRAM_BLOCK_SIZE-1:0];
logic [`DRAM_WORD_SIZE-1:0]         data_to_mem;
logic [`DRAM_ADDRESS_SIZE-1:0]      address_to_mem;

```

```

logic [`DRAM_ADDRESS_SIZE-1:0] address_to_mem_with_word_offset;

logic                we;

logic                we_from_mem;

logic                re;

logic                re_from_mem;

logic                acknowledge;

logic [log2(`DRAM_BLOCK_SIZE)-1:0] word_count;

logic [log2(`DRAM_CYCLE_TIME)-1:0] cycle_count;


    // remove next 2 lines for Quartus

//`ifndef DRAM_CYCLE_TIME
//  if(log2(`DRAM_CYCLE_TIME) > 1) begin
    always @(posedge clk)
        begin
            if (reset || (!re_from_mem && !we_from_mem))
                cycle_count <= 'b0;
            else
                cycle_count <= cycle_count + 1;
        end

    // remove next 7 lines for quartus

//          end
//  else begin
//      assign cycle_count = 0;
//  end
//  `else
//      assign cycle_count = 0;
//  `endif


/* verilator lint_off WIDTH */

```

```

        // remove next 2 lines for Quartus
// `ifdef DRAM_BLOCK_SIZE
//   if(log2(`DRAM_BLOCK_SIZE) > 0) begin
//       always @(posedge clk)
//           begin
//               if (reset || (!re_from_mem && !we_from_mem))
//                   word_count <= 'b0;
//               else if (cycle_count == 0)
//                   word_count <= word_count + 1;
//           end
//       // remove next 7 lines for quartus
//   end // if (log2(`DRAM_BLOCK_SIZE) > 0)
//   else begin
//       assign bus_data_from_mem[0] = data_from_mem[0];
//   end
// `else
//   assign bus_data_from_mem[0] = data_from_mem[0];
// `endif

assign acknowledge_from_mem = acknowledge;
assign data_pin = (we_from_mem==1) ? data_to_mem :
    (we_from_mem==0) ? `DRAM_WORD_SIZE'bz : `DRAM_WORD_SIZE'bx;
assign address_to_mem_with_word_offset = address_to_mem+{word_count,2'b00};

/* verilator lint_off SYNCASYNCNET */
always_ff @(posedge clk)
    if (re_from_mem && (cycle_count==`DRAM_CYCLE_TIME-2 || word_count==0))
        data_from_mem[word_count] <= data_pin;

```

```

genvar i;

generate

// replaced the following for quartus
//   for (i=0; i < `DRAM_BLOCK_SIZE; i++) begin
// with the following line:
        for (i=0; i < `DRAM_BLOCK_SIZE; i++) begin : bus_data_from_mem_generation
                assign bus_data_from_mem[i] = data_from_mem[i];

        end
endgenerate


always_ff @(posedge clk)
    if (reset)
        address_to_mem <= `DRAM_ADDRESS_SIZE'b0;
    else if ((bus_read_enable && !we) || bus_write_enable)
//   address_to_mem <= bus_address_to_mem + {word_count,2'b00};
        address_to_mem <= bus_address_to_mem;


always_ff @(posedge clk)
    if (we_from_mem)
        data_to_mem <= bus_data_to_mem[word_count];
/* verilator lint_on WIDTH */


always_ff @(posedge clk or posedge acknowledge or posedge reset)
    if (reset || acknowledge)
        re <= 1'b0;
    else if ((we==0) && bus_read_enable && (word_count==0))
        re <= 1'b1;


always_ff @(posedge clk or posedge acknowledge or posedge reset)

```

```

if (reset || acknowledge)

    we <= 1'b0;

else if (bus_write_enable && (word_count==0))

    we <= 1'b1;

/* verilator lint_on SYNCASYNCNET */

dram dram (
    .reset      (reset),
    .clk        (clk),
    .address     (address_to_mem_with_word_offset),
    .data        (data_pin),
    .write_enable (we),
    .read_enable  (re),
    .write_data_enable (we_from_mem),
    .read_data_enable (re_from_mem),
    .acknowledge  (acknowledge)
);

Endmodule

Forwarding.sv

module forwarding(

    input logic ExMemRegWrite, MemWbRegWrite,
    input logic [4:0] IdExrs1,
    input logic [4:0] IdExrs2,
    input logic [4:0] ExMemrs3,
    input logic [4:0] ExMemrd,
    input logic [4:0] MemWbrd,
    input logic ExMemSecond_alu_en,

```

```

        input logic [4:0] rs11,
        input logic [4:0] rs22,
        output logic [1:0] ForwardA,
        output logic ForwardE,
        output logic ForwardC,
        output logic ForwardD,
        output logic [1:0] ForwardB
    );

```

```

        assign ForwardA = ((ExMemRegWrite) && (ExMemrd != 0) && (ExMemrd ==
        IdExrs1)) ? 2'b10 :

```

```

                                                                    ((MemWbRegWrite) &&
        (MemWbrd != 0) && (MemWbrd == IdExrs1)) ? 2'b01 : 2'b00;

```

```

        assign ForwardB = ((ExMemRegWrite) && (ExMemrd != 0) && (ExMemrd ==
        IdExrs2)) ? 2'b10 :

```

```

                                                                    ((MemWbRegWrite) &&
        (MemWbrd != 0) && (MemWbrd == IdExrs2)) ? 2'b01 : 2'b00;

```

```

        assign ForwardE = ((MemWbRegWrite) && (ExMemSecond_alu_en) &&
        (MemWbrd != 0) && (MemWbrd == ExMemrs3)) ? 1'b1 : 1'b0;

```

```

        assign ForwardC = ((MemWbRegWrite) && (MemWbrd != 0) && (MemWbrd ==
        rs11)) ? 1'b1 : 1'b0;

```

```

        assign ForwardD = ((MemWbRegWrite) && (MemWbrd != 0) && (MemWbrd ==
        rs22)) ? 1'b1 : 1'b0;

```

```

endmodule

```

```

Imm_Gen.sv

```

```

module imm_Gen(

```

```

    input logic [31:0] Instruction,

```

```

    output logic [31:0] IMM

```

```
);
```

```
//made a new module for sign extension, since might need additional functionality
```

```
// later on when implementing new instructions
```

```
always_comb
```

```
    case (Instruction[31:26])
```

```
        6'b010011:    //Immediate instruction format
```

```
        IMM = {Instruction[15]? 20'hFFFFFF:20'b0 , Instruction[15:4]};
```

```
        6'b000011:    //Load Word
```

```
        IMM = {Instruction[15]? 20'hFFFFFF:20'b0 , Instruction[15:4]};
```

```
        6'b100011:    //Store Word
```

```
        IMM = {Instruction[25]? 20'hFFFFFF:20'b0 , Instruction[25:21], Instruction[10:4]};
```

```
        6'b000100:    //Branch Opcode might be changed later
```

```
        IMM = {Instruction[25]? 20'hFFFFFF:20'b0 , Instruction[25:21], Instruction[10:4]};
```

```
        6'b000111: //Multiply and addition
```

```
        IMM = {{(26){1'b0}},Instruction[5:0]};
```

```
        6'b000010: //Jump
```

```
        IMM = {{(6){Instruction[25]}},Instruction[25:0]};
```

```
        6'b111001: //FSJ
```

```
        IMM =
```

```
{{(8){Instruction[25]}},Instruction[25:21],Instruction[10:8],{(8){Instruction[7]}},Instruction[7:0]};
```

```
        ///Might need to add, half word, or byte load store, for now controlling
```

```
        // with funct2
```

```
        default: IMM = {32'b0};
```

```
    endcase
```

```
endmodule
```


Appendix C

IAXPY (assuming size of matrix $n^2 = 100$)

Assembly Code:

Addi x6, x0, 200	#starting location for B in memory
Addi x1, x0, 2	#scalar k
Addi x9, x0, 100	#counter value to reach for termination
L1: Lh x3, 0(x4)	#starting location for A in memory, loading 1 st 16-bit integer
Lh x5, 0(x6)	#loading 1 st 16-bit signed integer from B
Fma x7, x1, x3, x5	#fused-mul-add, $k \cdot A[0,0] + B[0,0]$ (1 st iteration)
Sh x7, 0(x6)	#storing result back to B[0,0]
Addi x4, x4, 2	#incrementing address for A, to load the next half-word
Addi x6, x6, 2	#incrementing address for B by 2
Addi x2, x2, 1	#incrementing counter
Bne x2, x9, L1	#till the whole matrix is finished

Machine Code:

4C C0 0C C0
4C 20 00 20
4D 20 06 40
0C 64 00 01
0C A6 00 01
1C E1 19 4C
00 00 00 00
8C 06 38 01
4C 84 00 20
4C C6 00 20
4C 42 00 10
13 E2 4D CD

SORTER + CRC

Addi x2, x0, 0x7FC

#Stack pointer initial place

	Addi x18, x0, 0x18F	#right-most index r = 399
	Subi x2, x2, 4	#adjust for stack
	Sw x1, 0(x2)	
	Jal MergeSort	#function call for Merge-Sort
	Lw x1, 0(x2)	
	Addi x2, x2, 4	
CRC:	addi x4, x0, 0x0010	
	Slli x4, x4, 0x00C2	
	Addi x4, x4, 0x0210	#x4 has the value of 0x1021
	Lh x9, 0(x23)	#first integer loaded
L2:	Addi x24, x9, 0x0000	#copied to x24
	Beq x9, x0, exit	#if null character, exit
	Andi x21, x9, 0x001	#take LSB of integer
	Andi x22, x3, 0x001	#take LSB of CRC
	Beq x21, x22, SRL	#if LSB's are same shift right
	Xor x3, x3, x4	#if not, xor CRC & 0x1021
SRL:	Srli x3, x3, 0x001	#right shift CRC reg
	Srli x9, x9, 0x001	#to check the next bit of integer
	Addi x8, x8, 0x001	#increment counter
	Beq x5, x8, next	# (counter ==16) go to next
	J L1	#16 iterations for each int
Next:	addi x23, x23, 0x001	#increment x23
	Addi x8, x0, 0x000	#clear counter
	Slli x23, x23, 0x001	#adjust for half-word address
	Lh x9, 0(x23)	#load the next half-word
	Slli x3, x3, 16	#shift the CRC to [31:16]
	Add x24, x24, x3	#appending CRC to int
	Slli x25, x25, 0x002	#adjusting for word offset
	Sw x24, 0(x25)	#storing the data to original loc.

Addi x25, x25, 0x001	#incrementing for next word
J L2	#process the next int

Exit:

x19 has left-most index, x18 has right-most index, x20 has the middle point

Merge-Sort:	slt x5, x19, x18	#if (l<r)
	Bne x5, x0 Proceed	
	Jr x1	# x1 has return address
Proceed:	subi x6, x18, 1	
	Srl x6, x6, 1	
	Add x20, x6, x19	# $m = l + (r-1)/2$
	Subi x2, x2, 16	# 4 items on stack to be placed
	Sw x1, 0(x2)	# return address
	Sw x19, 4(x2)	# l
	Sw x18, 8(x2)	#r
	Sw x20, 12(x2)	#m
	Addi x18, x20, 0x000	# move m to r
	Jal Merge-Sort	#1 st recursive call, 1 st half of array
	Subi x2, x2, 16	#4 more items on stack adjustment
	Sw x1, 0(x2)	# return address
	Sw x19, 4(x2)	# l
	Sw x18, 8(x2)	#r
	Sw x20, 12(x2)	#m
	Addi x19, x20, 1	# move m+1 to l
	Jal Merge-Sort	#2 nd recursive call, 2 nd half of array
	lw x1, 0(x2)	# return address
	lw x19, 4(x2)	# l
	lw x18, 8(x2)	#r
	lw x20, 12(x2)	#m
	Addi x2, x2, 16	#pop the values

	lw x1, 0(x2)	# return address
	lw x19, 4(x2)	# l
	lw x18, 8(x2)	#r
	lw x20, 12(x2)	#m
	Addi x2, x2, 16	#popped the values
	jal Merge	#calling Merge function
	jr x1	#return recursively
Merge:	Sw x1, 0(x2)	# return address
	Sw x19, 4(x2)	# l
	Sw x18, 8(x2)	#r
	Sw x20, 12(x2)	#m
	Addi x29, x20, 1	#n1 = (m+1)
	Sub x29, x29, x19	#n1 = (m+1-l)
	Sub x30, x18, x20	#n2 = r - m
FOR1:	bge x6, x29, FOR2	#x6=i , x7=j, x28 = k
	Add x10, x6, x19	
	Slli x10, x10, 1	
	Lh x11, 0(x10)	
	Slli x12, x6, 1	
	Sh x11, 800(x12)	#L[i] = arr[l + i]
	Addi x6, x6, 1	#inc i
	J FOR1	
FOR2:	bge x7, x30, ENDFOR	
	Addi x12, x7, 1	
	Add x13, x13, x20	
	Slli x13, x13, 1	
	Lh x14, 0(x13)	
	Slli x15, x7, 1	
	Sh x14, 1200(x15)	#R[j] = arr[m+1+j]

	Addi x7, x7, 1	#inc j
	J FOR2	
ENDFOR:	addi x6, x0, 0	#clearing regs
	addi x7, x0, 0	
	addi x10, x0, 0	
	addi x11, x0, 0	
	addi x12, x0, 0	
	addi x13, x0, 0	
	addi x14, x0, 0	
	addi x15, x0, 0	
	addi x28, x19, 0	# k = l
While1:	slt x10, x6, x29	
	Slt x11 x7, x30	
	And x12, x10, x11	# (l < n1 && j < n2)
	Beq x12, x0, While2	
	Slli x13, x6, 1	
	Lh x14, 800(x13)	# L[i]
	Slli x15, x7, 1	
	Lh x16, 1200(x15)	# R[j]
If:	Bltn x16, x14, else	# jump if R[j] < L[i]
	Slli x17, x28, 1	# arr[k]
	Sh x14, 0(x17)	# arr[k] = L[i]
	Addi x6, x6, 1	
	J Increment-K	
Else:	slli x7, x28, 1	
	Sh x16, 0(x17)	# arr[k] = R[j]
	Addi x7, x7, 1	
Increment-K:	addi x28, x28, 1	# k + 1
	J While1	

While2:	bge x6, x29, While3	#remaining elements
	Sh x14, 0(x17)	# arr[k] = L[i]
	Addi x6, x6, 1	
	Addi x28, x28, 1	
	J While2	
While3:	bge x7, x30, end-merge	
	Sh x16, 0(x17)	
	Addi x7, x7, 1	
	Addi x28, 28, 1	
	J While3	
End-merge:	lw x1, 0(x2)	# return address
	lw x19, 4(x2)	# l
	lw x18, 8(x2)	#r
	lw x20, 12(x2)	#m
	Addi x2, x2, 16	#popped the values
	Jr x1	#return

Machine Code in HEX:

Only for CRC:

4C 80 00 10

4C 84 00 C2

4C 84 02 10

0D 37 00 01

4F 09 00 00

10 09 04 C9

4E A9 00 1F

4E C3 00 1F

10 15 B0 49

4C 63 00 13

4D 29 00 13

4D 08 00 10

10 05 40 49

0B FF FF DC

4E F7 00 10

4D 00 00 00

4E F7 00 12

0D 37 00 01

4C 63 01 02

CF 18 18 00

4F 39 00 22

8C 19 C0 00

4F 39 00 10

0B FF FF AC

For Merge-Sort:

TEXT PARSER

Assembly Code:

Addi x19, x19, 0x30	# putting ASCII value '0' in reg 19
Addi x15, x0, 0x3E7	# assuming 1000 characters in total
L1: Beq x8, x15, EXIT	# counter to go from 0->999
Lb x9, 0(x8)	#loading first character from mem[0], counter value in x8
Beq x9, x0, EXIT	# if NULL character encountered, EXIT
Sgei x10, x9, 8	# is character ASCII value greater than or equal to 8
Slti x11, x9, 14	# is character ASCII value less than 14
Andi x12, x10, x11	# checking if it is a space character
Beq x12, x0, non-space	#if not space character jump
Lb x13, 1(x8)	#loading next character
Bne x13, x9, count	#checking for consecutive identical space characters
Subi x15, x15, 1	#decrement no. of characters

Addi x14, x14, 1	#increment no. of deleted spaces
Addi x17, x8, 1	#initialize a new counter with value of counter + 1
L2: Beq x17, x15, put'0'	#start second loop to shift back preceding characters
Lb x18, 1(x17)	#load next character
Sb x18, 0(x17)	#store to previous byte-address
Addi x17, x17, 1	#increment new counter
J L2	
Put'0': sb x19, 0(x15)	#avoiding any duplication of characters
count: addi x8, x8, 1	#increment counter
j L1	
non-space: addi x20, x20, 1	#increment non-space counter
addi x8, x8, 1	#increment counter
j L1	
EXIT: sw x20, 4(x15)	#printing the results, after parsing
Sw x14, 8(15)	

Machine Code in HEX:

```

4D 00 00 00
4E 60 03 00
4D E0 3E 70
10 08 7D 89
0D 28 00 02
10 09 05 09
4D 49 00 86
4D 69 00 E8
CD 8A 58 0F
10 0C 03 49
0D A8 00 12
10 0D 4A 4D
4D EF 00 11

```


4D CE 00 10
4E 28 00 10
10 11 79 09
0E 51 00 12
8C 11 90 02
4E 31 00 10
8C 0F 98 02
4D 08 00 10
0B FF FF B0
4D 08 00 10
4E 94 00 10
0B FF FF A4
8C 0F A0 40
8C 0F 70 80

Appendix D

Verilator test for IAXPY results

```
Ex time: 2.71604
Functional verification Status: PASS
root@DESKTOP-GNORGNU:/home/othman/lab5# verilator --trace --trace-max-array 4096 --cc top.sv --top-module top --exe -bui
ld sim_main.cpp
make: Entering directory '/home/othman/lab5/obj_dir'
g++ -I. -MMD -I/usr/local/share/verilator/include -I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=
0 -DVM_TRACE=1 -DVM_TRACE_FST=0 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-sign-compare -Wno-uninitiali
zed -Wno-unused-but-set-variable -Wno-unused-parameter -Wno-unused-variable -Wno-shadow -std=gnu++14 -Os -c -o sim_
main.o ../sim_main.cpp
g++ sim_main.o verilated.o verilated_vcd.o Vtop_ALL.a -o Vtop
make: Leaving directory '/home/othman/lab5/obj_dir'
root@DESKTOP-GNORGNU:/home/othman/lab5# obj_dir/Vtop
Giving the system 1 cycle to initialize with reset...
Running verification ...
test = 4test = 69test = 32775test = 292test = 6test = 3test = 98307test = 780test = 98310test = 288test = 3test = 102tes
t = 9test = 48test = 6test = 3test = 98307test = 780test = 98310test = 288test = 3test = 102test = 9test = 48test = 6tes
t = 3test = 98307test = 780test = 98310test = 288test = 3test = 102test = 9test = 48test = 6test = 3test = 98307test = 7
80test = 98310test = 288test = 3test = 102test = 9test = 48test = 6test = 3test = 98307test = 780test = 98310test = 288t
est = 3test = 102test = 9test = 48test = 6test = 3test = 98307test = 780test = 98310test = 288test = 3test = 102test = 9
test = 48test = 6test = 3test = 98307test = 780test = 98310test = 288test = 3test = 102test = 9test = 48test = 6test = 3
test = 98307test = 780test = 98310test = 288test = 3test = 102test = 9test = 48test = 6test = 3test = 98307test = 780tes
t = 98310test = 288test = 3test = 102test = 9test = 48test = 6test = 3test = 32771test = 328test = 32776test = 128Execut
ion completed successfully (simulation waveforms in .vcd file) ... !
Total number of instructions: 813
Elapsed Clock Cycles: 140736731369184
number of clock cycles spent on memory stalls: 9536
CPI: 3.166052
Ex time: 2.866370
Functional verification Status: PASS
root@DESKTOP-GNORGNU:/home/othman/lab5#
```

IAXPY Verilator Code

```
// CPU top level verilator simulation file:

// EEE 446 Spring 2021

// Ali Muhtaroglu, 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:
```

```
// Internal signals defined here:
```

```
// Note systemverilog design hierarchy can be traced by appending __DOT__ at every level:
```

```
#define regwritememwb      top__DOT__regwritememwb
#define dcache_data_ready top__DOT__dcache_data_ready
#define dcache_data_out   top__DOT__dcache_data_out
#define dcache_valid      top__DOT__dcache_valid
#define dcache_byte_en    top__DOT__dcache_byte_en
#define dcache_data_in    top__DOT__dcache_data_in
#define dcache_address    top__DOT__dcache_address
#define dcachce_mem_valid top__DOT__dcachce_mem_valid
#define dcache_rw         top__DOT__dcache_rw
#define icache_valid      top__DOT__icache_valid
#define icache_data_out   top__DOT__icache_data_out
#define icache_data_ready top__DOT__icache_data_ready
#define icache_mem_valid  top__DOT__icache_mem_valid
#define cache_miss_stall  top__DOT__cache_miss_stall
#define PCinc             top__DOT__cpu__DOT__u1__DOT__PCincremented
```

```

#define RF                top__DOT__cpu__DOT__u1__DOT__RF
#define Branch            top__DOT__cpu__DOT__Branch

//cache-mem

#define dram_busy          top__DOT__dram_busy
#define dram_port2_acknowledge    top__DOT__dram_port2_acknowledge
#define dram_port2_write_data    top__DOT__dram_port2_write_data
#define dram_port2_read_data     top__DOT__dram_port2_read_data
#define dram_port2_we           top__DOT__dram_port2_we
#define dram_port2_address      top__DOT__dram_port2_address
#define dram_port2_request      top__DOT__dram_port2_request
#define dram_port1_acknowledge   top__DOT__dram_port1_acknowledge
#define dram_port1_read_data     top__DOT__dram_port1_read_data
#define dram_port1_we           top__DOT__dram_port1_we
#define dram_port1_address      top__DOT__dram_port1_address
#define dram_port1_request      top__DOT__dram_port1_request
#define mem_data_out           top__DOT__mem_data_out
#define icache_mem_address      top__DOT__icache_mem_address
#define dcache_mem_address      top__DOT__dcache_mem_address
#define dcache_mem_rw           top__DOT__dcache_mem_rw
//#define dcache_mem_data_out    top__DOT__dcache_mem_data_out
#define mem_rw                 top__DOT__mem_rw


// In case you would like he simulator to do operations conditional to DEBUG mode:
#define DEBUG    1


//parameters needed for calc

```

```

#define fmax    80

#define energy   10

#define IC      813

#define pow     0.0002


// 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
    // simulation tests and/or signal outputs for debug can be put into action:
    void tick(void) {

        TESTBENCH<Vtop>::tick();

        // we are often interested in keeping track of number of clock ticks:
        m_tickcount++;

        //if (DEBUG) {
        // printf("%04lx-pipe: ", m_tickcount);
        // printf("%s:%5x",
        //      "instructio op code :",m_topsim->opcode);
        // printf("\n");

```

```
    // }  
}  
};
```

```
int main(int argc, char** argv, char** env){
```

```
    // input and output numbers to help with testing:
```

```
    // some parameters we may want to keep track of:
```

```
    long clock_count = 0;
```

```
    long counter = 0;
```

```
    long counter2 = 0;
```

```
    long error_count = 0;
```

```
    bool test_pass = false;
```

```
    long n = 100; //matrix size
```

```
    //long f = 512; //
```

```
    unsigned int test;
```

```
    signed short test2;
```

```
    long nwr =0; //number of writes and reads
```

```
    float EXtime =0; //execution time
```

```
    float CPI =0;
```

```
    float clk=0;
```

```
    float ener=0;
```

```
    // Initialize Verilators variables
```

```
    Verilated::commandArgs(argc, argv);
```

```

// Create an instance of our module under test
TOPLEVEL_TB *tb = new TOPLEVEL_TB;

// Message to standard output that test is starting:
//printf("Executing the test ...\n");

// Data will be dumped to trace file in gtkwave format to look at waveforms later:
tb->opentrace("top.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++;
//tb->m_topsim->RF = RF_WriteData;

// checking the memory at each write
if (DEBUG) printf("Running verification ...\n");
for (int k = 0; k < 20000 ; k++){
    tb->tick();
    clock_count++;
    //find the wasted cycle due to memory stalls
    if (tb->m_topsim->cache_miss_stall){
        counter++;
    }
}

```

```

//check functionality
if (tb->m_topsim->Branch && (tb->m_topsim->RF[2] < n)){
    test = ((tb->m_topsim->RF[1])*(tb->m_topsim->RF[3])+(tb->m_topsim->RF[5]));
    printf("test = %ld",test);
    clk = clock_count;
    if (test != tb->m_topsim->RF[7]){
        error_count++;
        counter2++;
    }
    if (counter2 == 100){

        printf("counter2");
        counter2++;
    }
    //for (int k = 0; k < 256 ; k++){
    // printf("\n");
    //printf("the memory value");
    //printf("datamem: %8x :: n: %8x \n",tb->m_topsim->datamem[k]);
    //printf("\n");
    //}
}
}

nwr = 768;
CPI = clk/IC;
EXtime = (IC*CPI/(90*10^6));
ener = EXtime * pow;
test_pass = (error_count > 0) ? 0 : 1;
printf("Execution completed successfully (simulation waveforms in .vcd file) ... !\n");
printf("Total number of instructions: %ld\n",IC);

```

```

printf("Elapsed Clock Cycles: %ld\n",clk);

printf("number of clock cycles spent on memory stalls: %ld\n",counter);

printf("CPI: %f\n",CPI);

printf("Ex time: %f\n",EXtime);

printf("Energy: %f\n",ener);


printf("Functional verification Status: %s\n",test_pass?"PASS":"FAIL");

if (error_count > 0){

    printf("Error count is: %ld\n",error_count);

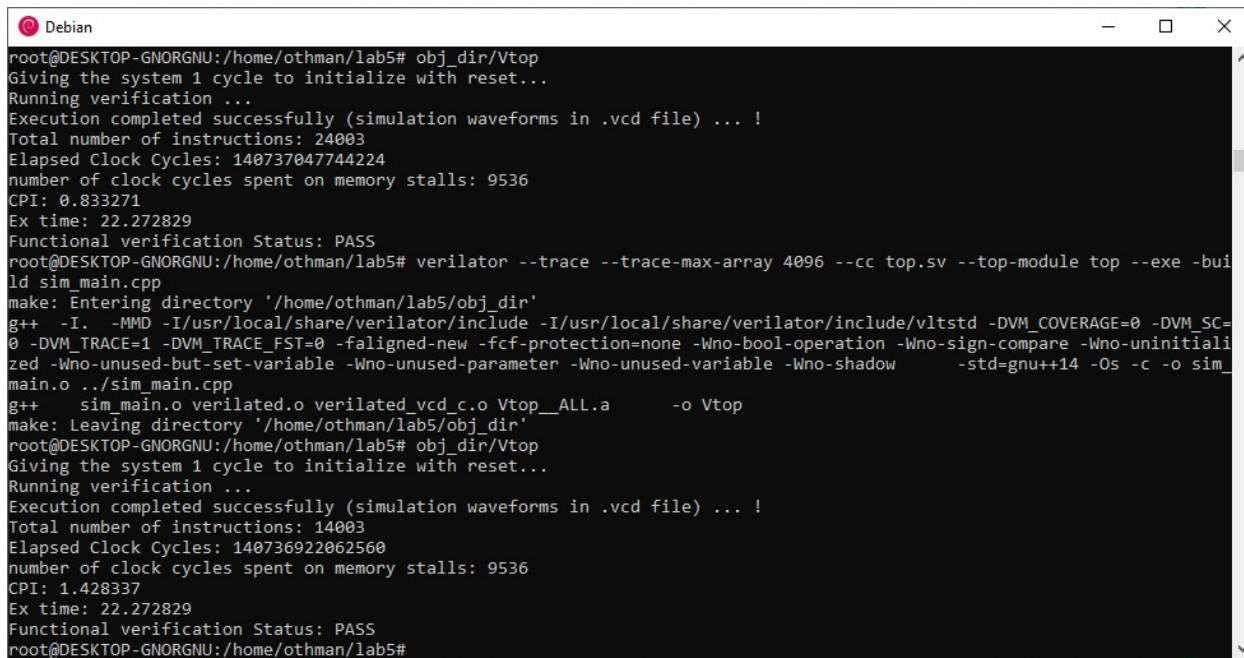
}

exit(EXIT_SUCCESS);

}

```

Text Parser Verilator Code & results



```

Debian
root@DESKTOP-GNORGNU:/home/othman/lab5# obj_dir/Vtop
Giving the system 1 cycle to initialize with reset...
Running verification ...
Execution completed successfully (simulation waveforms in .vcd file) ... !
Total number of instructions: 24003
Elapsed Clock Cycles: 140737047744224
number of clock cycles spent on memory stalls: 9536
CPI: 0.833271
Ex time: 22.272829
Functional verification Status: PASS
root@DESKTOP-GNORGNU:/home/othman/lab5# verilator --trace --trace-max-array 4096 --cc top.sv --top-module top --exe -build
sim_main.cpp
make: Entering directory '/home/othman/lab5/obj_dir'
g++ -I. -MMD -I/usr/local/share/verilator/include -I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=
0 -DVM_TRACE=1 -DVM_TRACE_FST=0 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-sign-compare -Wno-uninitiali
zed -Wno-unused-but-set-variable -Wno-unused-parameter -Wno-unused-variable -Wno-shadow -std=gnu++14 -O5 -c -o sim_
main.o ../sim_main.cpp
g++ sim_main.o verilated.o verilated_vcd_c.o Vtop__ALL.a -o Vtop
make: Leaving directory '/home/othman/lab5/obj_dir'
root@DESKTOP-GNORGNU:/home/othman/lab5# obj_dir/Vtop
Giving the system 1 cycle to initialize with reset...
Running verification ...
Execution completed successfully (simulation waveforms in .vcd file) ... !
Total number of instructions: 14003
Elapsed Clock Cycles: 140736922062560
number of clock cycles spent on memory stalls: 9536
CPI: 1.428337
Ex time: 22.272829
Functional verification Status: PASS
root@DESKTOP-GNORGNU:/home/othman/lab5#

```



```
#include <stdio.h>

#include <verilated.h>

#include <verilated_vcd_c.h>

#include "testbench.h"

#include "Vtop.h"
```

```
// Top level interface signals defined here:
```

```
// Internal signals defined here:
```

```
// Note systemverilog design hierarchy can be traced by appending __DOT__ at every level:
```

```
#define regwritememwb      top__DOT__regwritememwb
#define dcache_data_ready  top__DOT__dcache_data_ready
#define dcache_data_out    top__DOT__dcache_data_out
#define dcache_valid       top__DOT__dcache_valid
#define dcache_byte_en     top__DOT__dcache_byte_en
#define dcache_data_in     top__DOT__dcache_data_in
#define dcache_address     top__DOT__dcache_address
#define dcachce_mem_valid  top__DOT__dcachce_mem_valid
#define dcache_rw          top__DOT__dcache_rw
#define icache_valid       top__DOT__icache_valid
#define icache_data_out    top__DOT__icache_data_out
#define icache_data_ready  top__DOT__icache_data_ready
#define icache_mem_valid   top__DOT__icache_mem_valid
#define cache_miss_stall   top__DOT__cache_miss_stall
#define PCinc              top__DOT__cpu__DOT__u1__DOT__PCincremented
#define RF                 top__DOT__cpu__DOT__u1__DOT__RF
#define Branch             top__DOT__cpu__DOT__Branch
```

```
//cache-mem
```

```

#define dram_busy          top__DOT__dram_busy
#define dram_port2_acknowledge    top__DOT__dram_port2_acknowledge
#define dram_port2_write_data    top__DOT__dram_port2_write_data
#define dram_port2_read_data     top__DOT__dram_port2_read_data
#define dram_port2_we           top__DOT__dram_port2_we
#define dram_port2_address      top__DOT__dram_port2_address
#define dram_port2_request      top__DOT__dram_port2_request
#define dram_port1_acknowledge    top__DOT__dram_port1_acknowledge
#define dram_port1_read_data     top__DOT__dram_port1_read_data
#define dram_port1_we           top__DOT__dram_port1_we
#define dram_port1_address      top__DOT__dram_port1_address
#define dram_port1_request      top__DOT__dram_port1_request
#define mem_data_out           top__DOT__mem_data_out
#define icache_mem_address      top__DOT__icache_mem_address
#define dcache_mem_address      top__DOT__dcache_mem_address
#define dcache_mem_rw           top__DOT__dcache_mem_rw
//#define dcache_mem_data_out    top__DOT__dcache_mem_data_out
#define mem_rw                  top__DOT__mem_rw

```

// In case you would like the simulator to do operations conditional to DEBUG mode:

```

#define DEBUG    1

```

//parameters needed for calc

```

#define fmax    80

```

```

//#define energy    10

```

```

#define IC    813

```

```

#define power    0.000002

```

```

// 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
    // simulation tests and/or signal outputs for debug can be put into action:
    void tick(void) {

        TESTBENCH<Vtop>::tick();

        // we are often interested in keeping track of number of clock ticks:
        m_tickcount++;

        //if (DEBUG) {
        // printf("%04lx-pipe: ", m_tickcount);
        // printf("%s:%5x",
        //      "instructio op code :",m_topsim->opcode);
        // printf("\n");
        // }
    }
};

```

```
int main(int argc, char** argv, char** env){  
    // input and output numbers to help with testing:  
  
  
  
  
  
  
    // some parameters we may want to keep track of:  
    long clock_count = 0;  
    long counter = 0;  
    long counter2 = 0;  
    long error_count = 0;  
    bool test_pass = false;  
    long n = 100; //matrix size  
    //long f = 512; //  
    unsigned int test;  
    signed short test2;  
    long nwr =0; //number of writes and reads  
    float EXtime =0; //execution time  
    float CPI =0;  
    float clk=0;  
    float energy=0;  
    char string [1000];  
    int c = 0;  
  
  
  
  
  
  
  
  
  
    // Initialize Verilators variables  
    Verilated::commandArgs(argc, argv);  
    // Create an instance of our module under test  
    TOPLEVEL_TB *tb = new TOPLEVEL_TB;
```

```

// Message to standard output that test is starting:
//printf("Executing the test ...\n");

// Data will be dumped to trace file in gtkwave format to look at waveforms later:
tb->opentrace("top.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++;
//tb->m_topsim->RF = RF_WriteData;

// checking the memory at each write
if (DEBUG) printf("Running verification ...\n");
for (int k = 0; k < 20000 ; k++){
    tb->tick();
    clock_count++;
    //find the wasted cycle due to memory stalls
    if (tb->m_topsim->cache_miss_stall){
        counter++;
    }
    //check functionality
    if ((tb->m_topsim->Branch && (tb->m_topsim->RF[8] < n)) && (c != 1)){

```

```

    clk = clock_count;

    if((tb->m_topsim->RF[9] == 8) || (tb->m_topsim->RF[9] == 11) || (tb->m_topsim->RF[9] == 13) ||
        (tb->m_topsim->RF[9] == 12) || (tb->m_topsim->RF[9] == 14) || (tb->m_topsim->RF[9] == 15)){
        counter2++;
    }

    if (tb->m_topsim->RF[20] != counter2){
        error_count++;
    }
}

}

}

if (tb->m_topsim->RF[9] != 0){
    clk = clock_count;
    c = 1;
}

=

}

}

CPI = clk/IC;

EXtime = (IC*CPI/(90*10^6));

energy = EXtime * power;

test_pass = (error_count > 0) ? 0 : 1;

printf("Execution completed successfully (simulation waveforms in .vcd file) ... !\n");

printf("Total number of instructions: %ld\n",IC);

printf("Elapsed Clock Cycles: %ld\n",clk);

printf("number of clock cycles spent on memory stalls: %ld\n",counter);

printf("CPI: %f\n",CPI);

printf("Ex time: %f\n",EXtime);


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);

}

```

Appendix E

Fitter could not complete the simulation firstly due to size. When the FPGA with larger size was chosen, the fitter a lot of time to respond and got stuck 81%. However, in module 4, we had already calculated the f_{\max} parameters, as shown below, and since there weren't any noticeable changes in the Datapath, we decided to use that for the calculation of parameters.

Slow 1200mV 85C Model Fmax Summary				
	Fmax	Restricted Fmax	Clock Name	Note
1	83.8 MHz	83.8 MHz	clk	

The screenshot shows the Quartus Prime Lite Edition interface. The 'Flow Summary' window is open, displaying the following information:

- Flow Status:** Flow Failed - Sun Jun 27 19:02:40 2021
- Quartus Prime Version:** 20.1.1 Build 720 11/11/2020 SJ Lite Edition
- Revision Name:** top
- Top-level Entity Name:** top
- Family:** Cyclone V
- Device:** 5C9KFC8E7F35C8
- Timing Models:** Final
- Logic utilization (in ALMs):** 68,285 / 113,560 (78 %)
- Total registers:** 41324
- Total pins:** 3 / 616 (<1 %)
- Total virtual pins:** 0
- Total block memory bits:** 0 / 12,492,800 (0 %)
- Total DSP Blocks:** 0 / 342 (0 %)
- Total HSSI RX PCSs:** 0 / 12 (0 %)
- Total HSSI PMA RX Deserializers:** 0 / 12 (0 %)
- Total HSSI TX PCSs:** 0 / 12 (0 %)
- Total HSSI PMA TX Serializers:** 0 / 12 (0 %)
- Total PLLs:** 0 / 20 (0 %)
- Total DLLs:** 0 / 4 (0 %)

The 'Messages' window at the bottom shows the following error and warning messages:

- 11888** Total time spent on timing analysis during the fitter is 941.78 seconds.
- 11802** Can't fit design in device. Modify your design to reduce resources, or choose a larger device. The Intel FPGA Knowledge Database contains many articles with specific details on how to resolve this error. visit the knowle...
- 144001** Generated suppressed messages file c:/users/pcront/Downloads/r/output_Files/top_fit.smsg
- Quartus Prime Fitter was unsuccessful. 2 errors, 9 warnings**
- 293001** Quartus Prime Full compilation was unsuccessful. 4 errors, 82 warnings

The 'Tasks' window on the left shows the compilation process with the following tasks:

- Compile Design** (Failed)
- Analysis & Synthesis** (Completed)
- Fitter (Place & Route)** (Failed)
- Assembler (Generate programming file)** (Completed)
- Timing Analysis** (Completed)
- EDA Netlist Writer** (Completed)
- Edit Settings** (Completed)
- Program Device (Open Programmer)** (Completed)

