Advanced Processors Final Project Summary

Temple University

David Bongiorno

December 13, 2017

Objectives

The goal of this assignment is to complete the design of a single-cycled processor that implements the MIPS instruction set architecture as described in the lectures and in the fourth chapter of the book. Students should utilize the Xilinx ISE Design Suite so design and test their modules. The processor should be able execute load, store, R-type (addition, subtraction, multiplication and three selected logical operations), branch, and jump instructions. Additionally students are constrained to implement a 32-bit adder via structural modeling through gradual instantiation of smaller modules. Additionally Multiplication and division should be handled via the one of the iterative algorithms discussed during the lecture. Finally students were sent an set of pseudocode that must be interpreted into MIPS instructions for execution and evaluation.

Tools

- Xilinx ISE Design Suite (Verilog)
- MIPS Instruction Converter https://www.eg.bucknell.edu/~csci320/mips_web/
- Patterson, D. A., & Hennessy, J. L. (2018). Computer Organization and Design: The Hardware/Software Interface. Cambridge, MA: Morgan Kaufmann.

Procedure

The basic procedure implemented was close to what was described in the objectives section: each module was examined designed as a separate entity and tested. Then each module was added to the over-arching top module to create the datapath in the processor needed to satisfy all possible instructions as depicted in figure 1:

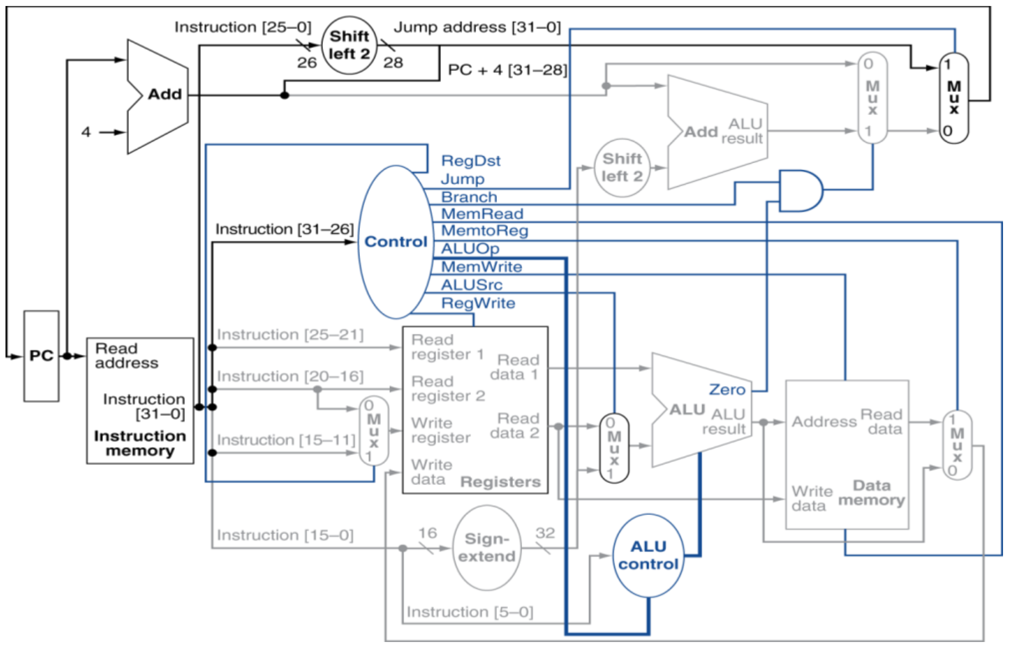


Figure 1: Block Diagram for simplified MIPS single-cycle processor.

I began with the smaller modules (sign extension, the two left shifters, 32-bit adder, and the multiplexor, controllers) that were rather easy to design and test on their own. These modules all

pertain to combinational logic, and thus do not necessitate the need for memory elements or clocking or reset signals.
I then moved onto the sequential modules whose outputs depend upon the state of the processor at a given time (in addition to any sort of wired stimulus). These modules included the program counter (PC), instruction memory (IMEM), register files, arithmetic logic unit (ALU), and data memory.

Finally, I began to incorporate all of the modules and internal wires needed to glue the processor together. From this point, it became necessary to design and test the datapath (and instantiated modules) in a combined, iterative process that essentially comprised of examining the simulated internal behavior in the timing window of the processor for discrepancies against expected behavior.

Module Description

Sign Extension: This section of the overarching processor is responsible, according to MIPS architecture, for specifying the immediate value (for I-type instructions), the address in memory for read/write operations, and it serves to create the target address offset to be added to the current address (PC) for jumps or branching instructions, based on the zero output of the ALU. The module functions by taking in a 16-bit input (the first 16 bits of the instruction signal) and extends it to a 32- bit signal, while maintaining the sign, as indicated by the most significant (16th) bit. For example 16'b1001_0010_0001_1100 becomes 32'b1111_1111_1111_1111_1001_0010_0001_1100.

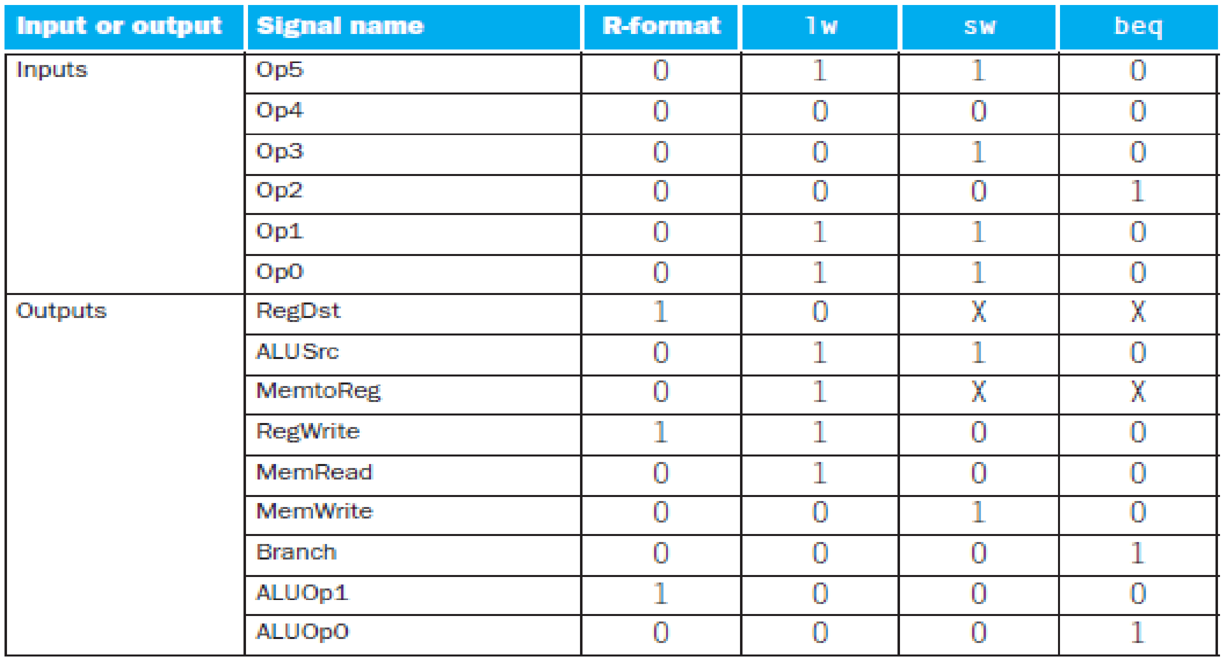
32-Bit Adder: This module implements the set of instantiated submodules (two half adder – four full adders, four 4-bit adders, and two 16-bit adders) to create a top level module that calculates the sum of two 32-bit inputs. The lowest submodule (half adder) was designed using gate level descriptions through a process called structural modeling, as per one of the objective constraints. This module is only used in the ALU. For the sake of ease all summations in the top level module are completed with the addition operator "+".

Multiplexors: There are many multiplexors used throughout the datapath, but they all function the same way – a selecting signal determines which of two inputs will be used as an output. The ternary operator (conditional operator) was used for continuous assignments. The multiplexors are used to determine the register to be written to for R-type instructions (as governed by RegDst); the second operand supplied to the ALU(as governed by ALUSrc); the source of Write Data which should come from data memory during an lw command or directly from the output of the ALU (as governed by MemtoReg); and two registers are used in tandem to determine the ultimate value the next subsequent program count (which could be the normal PC + 4, a jump address, or a branch address).

Left Shifters: There are two shifting utilized in this particular architecture. The first is used for jump commands taking the lowest 26 bits of the current instruction (the destination address in binary shifted right by 2 bits) and padding two zeros to the right of the signal. This 28-bit signal is concatenated with incremented PC (PC +4) to create the complete jump address.

The second shifter retains the original length of the incoming signal (31-bits) and applies an arithmetic shift of two to the left. This shifter is used in conjunction with the sign-extender and adder (whose other operand is the incremented PC) to calculate the branch address should the current instruction be a branch and all valid branching conditions are satisfied (zero flag & Branch control signal == 1).

Control Units:
Main Control: this module is responsible for decoding the top 6-bits of the current instruction pointed to by the PC and sending out the appropriate control signals needed to discern what type of command is to be executed. These outputted signals are supplied to the multiplexers and the ALU Control. These signals in total describe the difference between R-Type, I-Type, lw, sw, and branch commands:



Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Figure 2: Enumerated mapping of possible opcodes. NB a final row for the Jump signal should be added for the inclusion of jump instructions.

ALU Control: This module decodes in the lowest 6-bits of the incoming current instruction (the function field) and supply the control signals to the ALU to regulate the actual arithmetic/logical operation that is to be done on the incoming operands:

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

Figure 3: Enumerated mapping of possible opcodes. NB multiplication and division were treated as R-type instructions so that use of LO & HI registers was avoided.

Arithmetic Logic Unit (ALU): This module is integral to the actual execution of a desired function. Regardless of the type of current instruction, some arithmetic or logical operation must be evaluated; even a NOP (add 0 0 0) must be expressed with arithmetic. The ALU takes in two 32-bit sized operands, as well as the control bus from the ALU controller, which determines the type of operation to be executed (addition, subtraction, set on less than, etc.). The outputs include the actual result of the operation, whatever it may be, and the zero flag, which is raised in the event that the result is 32'd0. This flag is used to determine the satisfaction of a branching condition and then along with the Branching control signal possibly execute the jump.

Program Count (PC): The program count is responsible for supplying the instruction memory location to be referenced in the next instruction fetch. It is a sequential module that updates the next_PC on every positive clock edge. Should the system be reset there is a subroutine that will begin the program, setting next_PC = 0, effectively fetching the zeroth instruction.

Instruction Memory: This module consists of a 2-D register array containing the entire set of 32-bit instructions to be read and implemented through the execution of a given program. The personal decision to manually input all instructions was chosen for ease of access/update. However the module could be modified to support the input of a text file containing an arbitrary set of instruction (allowing for parametrized use, beyond the execution of the pseudo instructions). At each PC an instruction is selected and sent out to the controller, register file, register file mux, sign-extender, and the ALU controller for immediate use/decoding.

Register Files: This is another 2-D array used to house the 32, 32-bit registers that could be used or referenced in any sort of R-Type of I-Type instruction. At the beginning of the program (during the reset state) all of the registers are flashed to contain 32'd0, so that they are ready for immediate use. The register file takes in the control signal RegWrite (do determine if a register is going to store some new value), two register fields for reading, and one for writing. The outputs include two data slots to be read - one for the upper operand of the ALU, and the other for the ALU's multiplexor as well as Data Memory, should a sw instruction need to be executed.

Data Memory: This module is similar to the register Files in that it is a 2-D array that can hold many 32-bit registers, and is flashed upon reset. Along with the desired memory location to be accessed and possible data coming in (Write data), this module takes in the two control signals MemRead and MemWrite to execute a lw instruction or an sw instruction respectively. Data Memory outputs data to be read (Read Data) in the case of an lw command.

Test Plan

Sign Extension: This design was tested by instantiating the Verilog module via unit under test (uut) within a testbench. This testbench references a small set of test vectors that enumerate four possible 16-bit valid inputs and the corresponding expected 32-bit outputs that should be returned from the uut. This particular testbench was not completely exhaustive, since I did not write all 2^{16} possible cases that would need to be tested to ensure correct sign-extension. However, four cases were sufficient to test the necessary condition. These sample cases were successfully simulated and can be seen in the timing windows of the following figures:

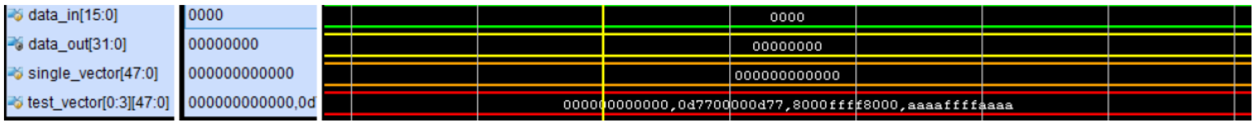


Figure 4: The first test vector (single_vector, orange waveform) from the set of tested vectors (red waveform) contains the input to the uut, which itself consists of the tested instruction (data_in, green waveform) and the target address offset generated by the uut (data_out, yellow waveform). The first input (0x0000) yields (0x00000000) and extends the most significant bit by 16 bits, maintaining the original sign.

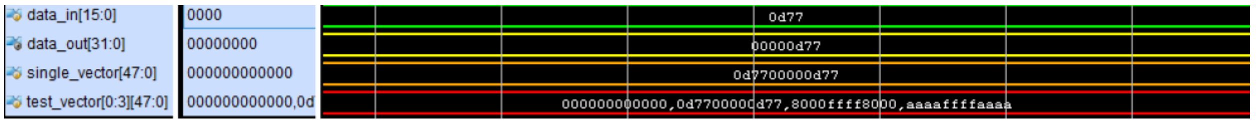


Figure 5: The second input (0x0D77) yields (0x00000D77) and extends the most significant bit by 16 bits, maintaining the original sign.

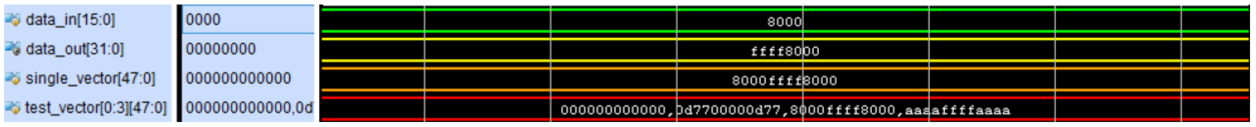


Figure 6: The third input (0x8000) yields (0xFFFF8000) and extends the most significant bit by 16 bits, maintaining the original sign.

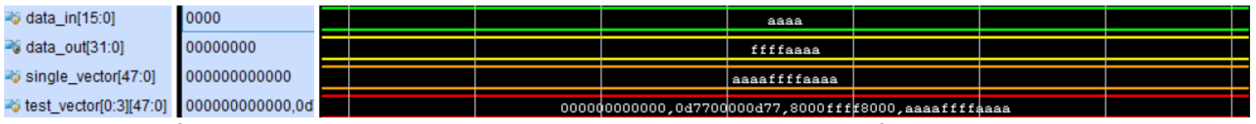


Figure 7: The fourth input (0xAAAA) yields (0xFFFFAAAA) and extends the most significant bit by 16 bits, maintaining the original sign.

32-Bit Adder: A uut of this module was tested with a small set of input stimulus within a testbench. The adder is limited in that overflow cases are unexpected and must be avoided by the user. Regardless the processor could be updated with the inclusion of an overflow flag. Two sets of captured waveforms are depicted below:

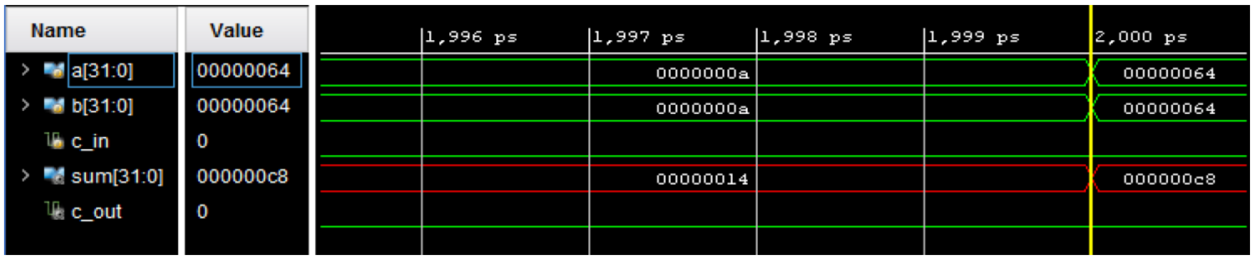


Figure 8: Valid summation: Sum = a + b. 0x14 = 0x0A + 0x0A (20 = 10 + 10).

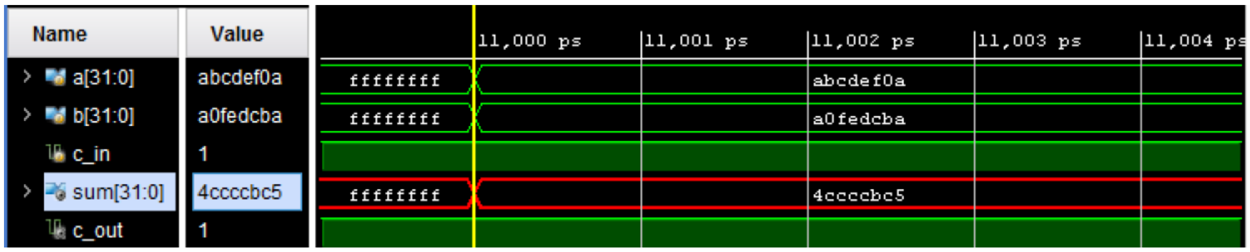


Figure 9: Valid summation however results larger than cannot be accurately represented in cases of arithmetic overflow

Multiplexor: The 32 Bit multiplexor was tested with sample waveforms within a testbench and yielded the correct outputs:

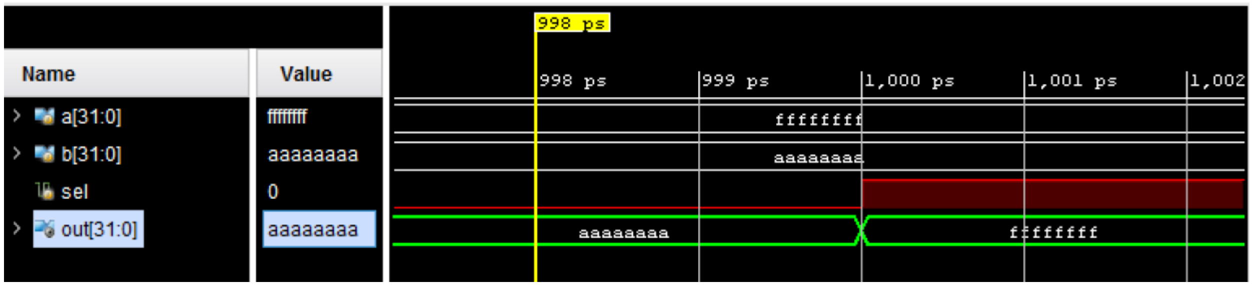


Figure 10: Valid Multiplexing for two possible 32'bit inputs. Selecting signal is switched to show proper operation.

Shifters: The 26-28 Bit and 32 Bit shifters were tested with sample waveforms within a testbench and yielded the correct outputs:

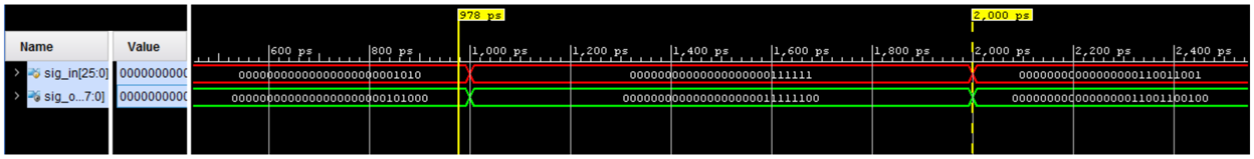
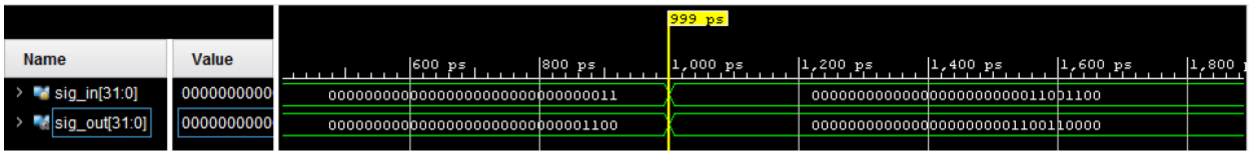


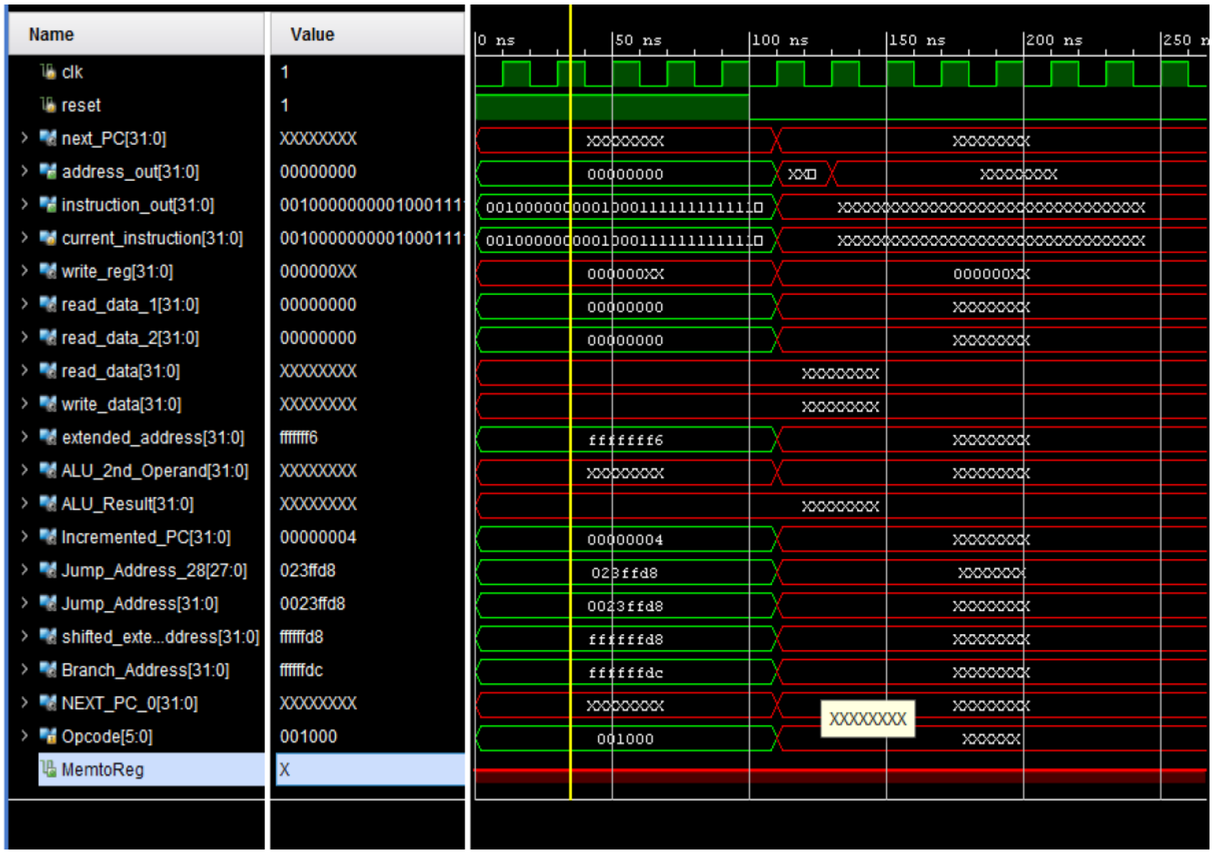
Figure 11: Valid Shift & 00 Padding.



Name	Value
sig_in[31:0]	0000000000
sig_out[31:0]	0000000000

Figure 12: Valid Arithmetic Shift.

The rest of the modules were tested through real-time simulation and had to be updated as needed. I cannot list every instance of a bug solved as it was a continuous process, but I do remember one at the beginning that was dealt with:

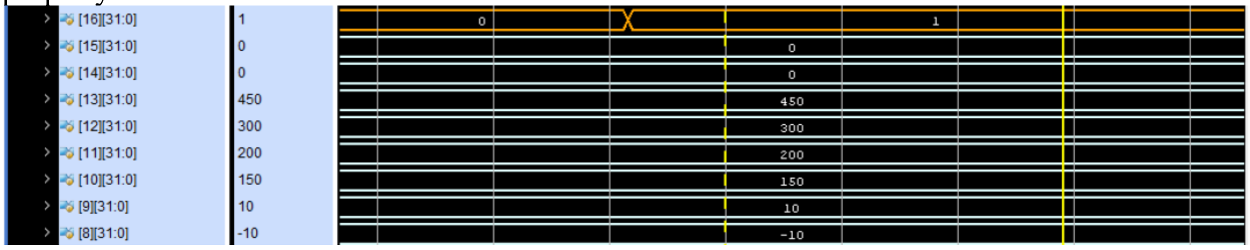


Name	Value
clk	1
reset	1
next_PC[31:0]	XXXXXXXX
address_out[31:0]	00000000
instruction_out[31:0]	0010000000001000111
current_instruction[31:0]	0010000000001000111
write_reg[31:0]	000000XX
read_data_1[31:0]	00000000
read_data_2[31:0]	00000000
read_data[31:0]	XXXXXXXX
write_data[31:0]	XXXXXXXX
extended_address[31:0]	fffffff6
ALU_2nd_Operand[31:0]	XXXXXXXX
ALU_Result[31:0]	XXXXXXXX
Incremented_PC[31:0]	00000004
Jump_Address_28[27:0]	023ffd8
Jump_Address[31:0]	0023ffd8
shifted_exte...ddress[31:0]	ffffffd8
Branch_Address[31:0]	ffffffdc
NEXT_PC_0[31:0]	XXXXXXXX
Opcode[5:0]	001000
MemtoReg	X

Figure 13:This figure depicts the error that was once present during reset: nothing was being updated, and the next_PC could not determine a proper value. The error was solved by noting that the control signal MemtoReg was also erroneous and that my Controller needed to be tweaked.

Results & Observations

At the moment the processor is partially operational: addi, lw, and sw instructions operate properly:



Name	Value
[16][31:0]	1
[15][31:0]	0
[14][31:0]	0
[13][31:0]	450
[12][31:0]	300
[11][31:0]	200
[10][31:0]	150
[9][31:0]	10
[8][31:0]	-10

Figure 14: Variables (i, j, a-d) are all set inside their proper register slots.

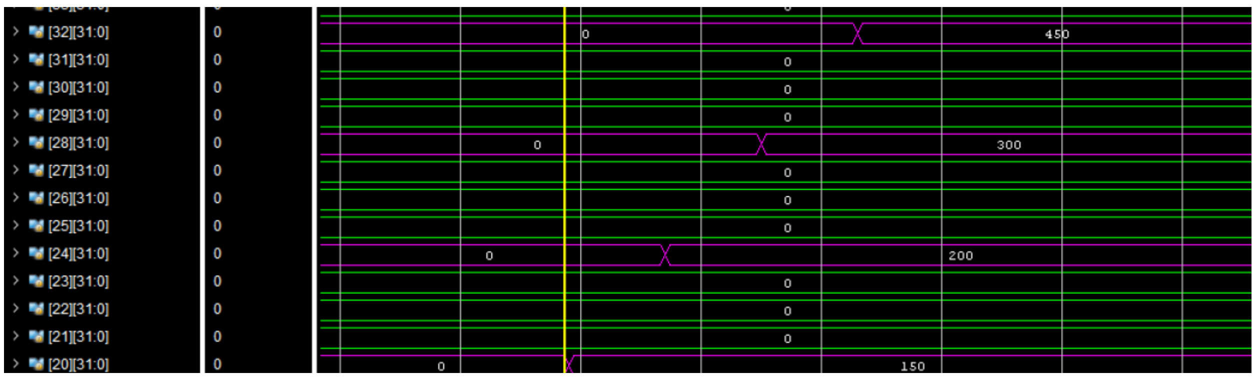


Figure 15: Variables (i, j, a-d) are all properly stored in Data Memory.

However, the multiplication and division instructions would behave as "add" commands. I understand that tried to I forced a multiplication command to behave like an add by altering the function field e.g.

0000 0001 1010 1100 0110 1000 0010 0000 ADD t5 t5 t4
0000 0001 1010 1100 0110 1000 0001 1000 "ADD" (MULTIPLY) t5 t5 t4

I thought that I could tweak this instruction and the controllers to adhere to the syntax and glue logic of the existing Controller, ALU Controller, and ALU, but there is still a bug present as evident with either case with the final result of f (register t7 and data mem location 36 or 40, depending on path taken):

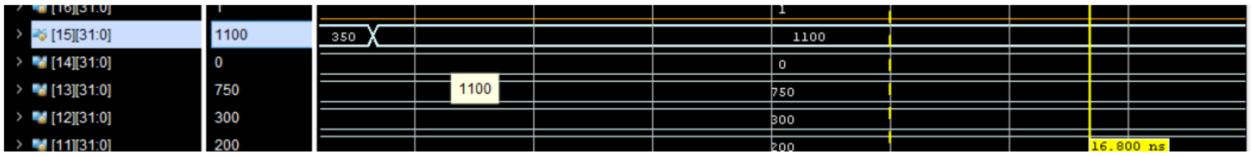


*Figure 16: Case I result is erroneous 150 - (300 * 450) != 1100.*

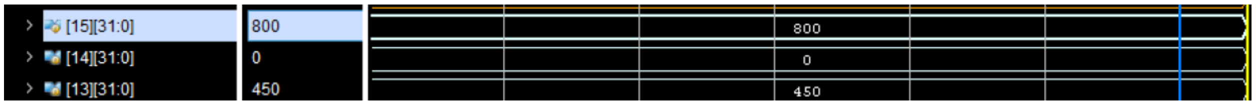
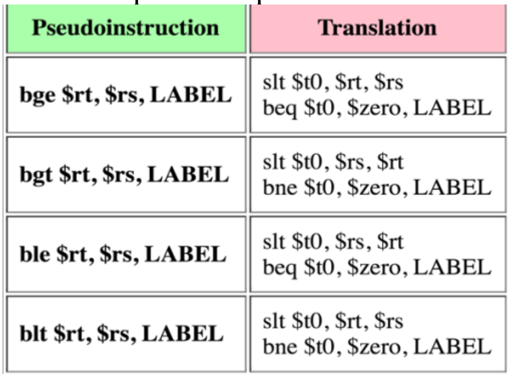


Figure 17: Case II result is erroneous (200+450)/150 = 4 != 1100.

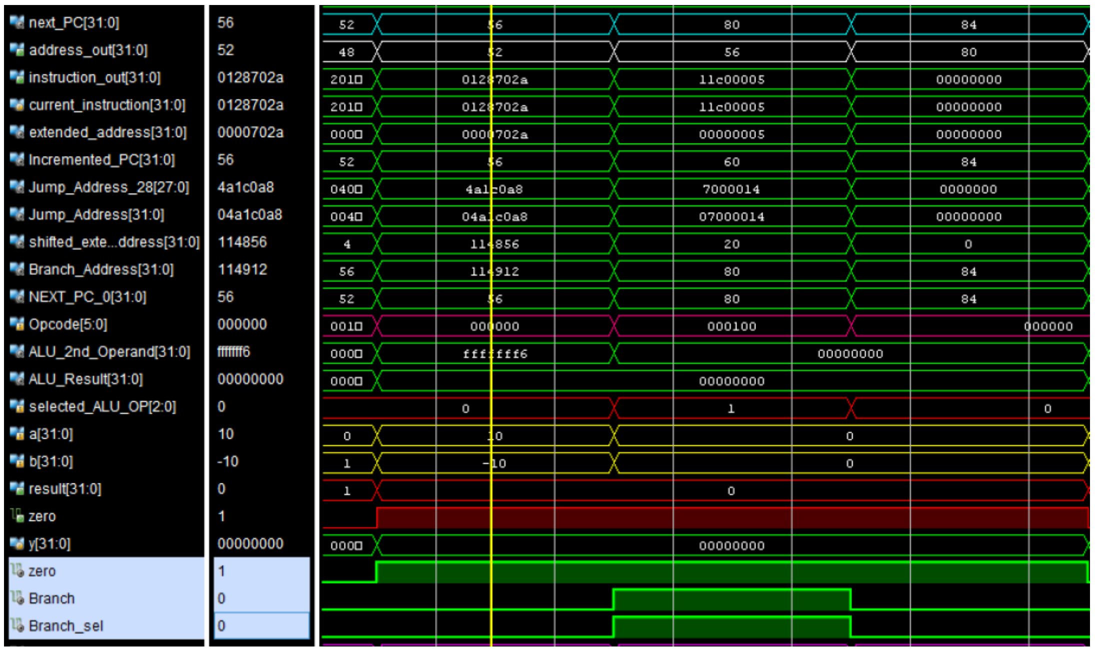
Secondly the branching path (i>j) is erroneously taken at all times. I understand that blt or bgt are pseudo instructions and must be reinterpreted as per this chart:



Pseudoinstruction	Translation
bge $rt, $rs, LABEL	slt $t0, $rt, $rs beq $t0, $zero, LABEL
bgt $rt, $rs, LABEL	slt $t0, $rs, $rt bne $t0, $zero, LABEL
ble $rt, $rs, LABEL	slt $t0, $rs, $rt beq $t0, $zero, LABEL
blt $rt, $rs, LABEL	slt $t0, $rt, $rs bne $t0, $zero, LABEL

Figure 18:Branching instruction pseudo conversion

I cannot find an explicit error in my code, so there must exist a set of logical fallacies embedded in my contollers/alu/ the instructions themselves. I tried to make the second row (bgt) equivalent, but realized I had no bne command. No matter what I tried (tweaking the availability/definition of the zero flag, creating a register set to 1'b1 to test the converse statement, beq testreg reg==1 label, etc.) would not work: as demonstrated the zero flag is always set high during the branch command, even if the condition shouldn't be satisfied in theory:



```
next_PC[31:0]                  56
address_out[31:0]              52
instruction_out[31:0]          0128702a
current_instruction[31:0]      0128702a
extended_address[31:0]         0000702a
Incremented_PC[31:0]           56
Jump_Address_28[27:0]          4a1c0a8
Jump_Address[31:0]             04a1c0a8
shifted_exte...ddress[31:0]    114856
Branch_Address[31:0]           114912
NEXT_PC_0[31:0]                56
Opcode[5:0]                    000000
ALU_2nd_Operand[31:0]          fffffff6
ALU_Result[31:0]               00000000
selected_ALU_OP[2:0]           0
a[31:0]                        10
b[31:0]                        -10
result[31:0]                   0
zero                           1
y[31:0]                        00000000
zero                           1
Branch                         0
Branch_sel                     0
```

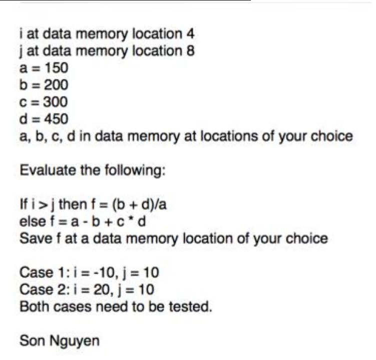
Figure 19: Case I (i.e. i = -10, j = 10). Equivalent branch condition is Bgt, but branch still erroneously taken

Conclusion

In summary the processor had limited functionality and was only able to perform some of the tasks as specified. However I learned a great deal about the implications of how an instruction set is designed within the scope of realizable hardware. In the process I gained practice in implementing structural and behavioral modeling within Verilog and made extensive use of the simulated timing window for output and error analysis.

Appendix

Pseudo-instructions



i at data memory location 4
j at data memory location 8
a = 150
b = 200
c = 300
d = 450
a, b, c, d in data memory at locations of your choice

Evaluate the following:

If i > j then f = (b + d)/a
else f = a - b + c * d
Save f at a data memory location of your choice

Case 1: i = -10, j = 10
Case 2: i = 20, j = 10
Both cases need to be tested.

Son Nguyen

Instruction Set:

```
 IMEM[0] = 32'h2008FFF6;  // addi t0, zero,-10  (0xFFFFFFF6)   // i = -10 CASE I
//IMEM[0] = 32'h0x20080014 // addi t0 , zero, 20 (0x0000024)   // i = 20   CASE II
  IMEM[4] = 32'hAC080004;  // sw t0 0x0004 zero  (location 4)   // store i in data memory location 4
  IMEM[8] = 32'h2009000A;  // addi t1 zero 10    (0x0000000A)   // j = 10
  IMEM[12] = 32'hAC090008; // sw t1 0x0008 zero  (location 8)   // store k in data memory location 8
  IMEM[16] = 32'h200A0096; // addi t2 zero 150   (0x00000096)   // a = 150
  IMEM[20] = 32'hAC0A0014; // sw t2 0x0014 zero  (location 20)  // store a in data memory location 20
  IMEM[24] = 32'h200B00C8; // addi t3 zero 200   (0x000000C8)   // b = 200
  IMEM[28] = 32'hAC0B0018; // sw t3 0x0018 zero  (location 24)  // store b in data memory location 24
  IMEM[32] = 32'h200C012C; // addi t4 zero 300   (0x0000012C)   // c = 300
  IMEM[36] = 32'hAC0C001C; // sw t4 0x001c zero  (location 28)  // store c in data memory location 28
  IMEM[40] = 32'h200D01C2; // addi t5 zero 450   (0x000001C2)   // d = 450
  IMEM[44] = 32'hAC0D0020; // sw t5 0x0020 zero  (location 32)  // store d in data memory location 32
  IMEM[48] = 32'h20100001; // ADDI S0, zero, 1   CREATE VALUE FOR BEQ = 1 (BNE = 0) FOR BRANCH
  IMEM[52] = 32'h0128702A; // slt t6 t1 t0       (PART I of BRANCH CONDITION (BGT) )***
  IMEM[56] = 32'h11C00005; // beq t6 s0 0x0005   (PART II of BRANCH CONDITION (BGT)) -> if satisfied jump to PC = 80
  IMEM[60] = 32'h01AC6818; // "add" t5 t5 t4     (MULTIPLY COMMAND WRITTEN AS AN R-TYPE) // (c * d)
  IMEM[64] = 32'h014B7822; // sub t7 t2 t3       (a - b)
  IMEM[68] = 32'h01ED7820; // add t7 t7 t5       result = (a-b) + (c*d)
  IMEM[72] = 32'hAC0F0024; // sw t7 0x0024 zero  store result in memory (location 36)!!!!
  IMEM[76] = 32'h08000018; // j 0x0018           jump to finish! (PC = 92)
  IMEM[80] = 32'h00000000;
  IMEM[84] = 32'h016D5820; // If BRANCH add t3 t3 t5        (b + d)
  IMEM[88] = 32'h016A781A; //  DIVIDE"add" t7 t3 t2      (DIVIDE COMMAND WRITTEN AS AN R-TYPE) // (b + d ) / a
  IMEM[92] = 32'hAC0F0028; // sw t7 0x0028 zero  store result in memory (location 40)!!!!
  IMEM[94] = 32'h00000000;
```