CS21 PROJECT 02

ENGINEERING NOTEBOOK

MIPS Single Cycle Processor Extension

SECTION LAB4

Submitted to

Sir Ivan Carlo Balingit
Sir Wilson M. Tan

Prepared by

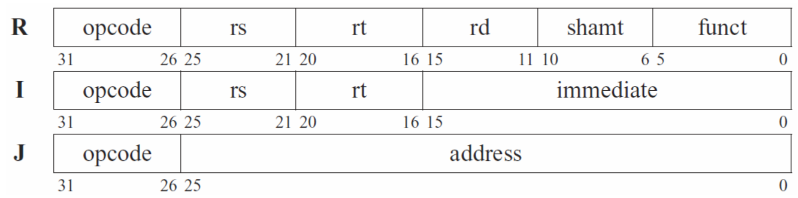
Yenzy Urson S. Hebron
202003090

JUNE 2022
A.Y. 2122.2

Table of Contents

Basic Instruction Formats

Before we begin, we recall the Basic Instruction Formats in MIPS32 for later reference:



R	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0
I	opcode	rs	rt	immediate		
	31 26	25 21	20 16	15 0		
J	opcode	address				
	31 26	25 0				

Important: Note that this documentation was made after the additional instructions have been already implemented and tested. In other words, the HDL and schematic modifications we did for each instruction overlaps. Hence for our explanations, we will be adding multiple **retrospective caveats** in order to emulate our step-by-step assembly of the extended MIPS Single Cycle Processor, and hopefully you'll get a good picture of how we went from adding shift left logical to adding zero from right. To be more precise, we added instructions in the same order that they're specified in the Project 2 Instructions.

Moreover, as the processor got more capable, we also integrated multiple instructions in the testbenches, both original and new. And I'll already mention that I reused the testbenches and memory files from Lab 12 in constantly verifying the integrity of the processor with regards to the original instructions. And for the *ad hoc* testing, we made use of multiple versions of the memory file corresponding to the testbenches for each specific instruction.

Normal Instructions

sll rd, rt, shamt

Instruction Format

The **shift left logical** or **sll** instruction is an R-type instruction with the following properties:

- Coded in MIPS Assembly as sll rd, rt, shamt.
- Opcode = 000000
- Funct = 000000
- The rs field is a Don't Care field. **Note** that for Don't Cares in the simulation machine codes, we will use random bits to emulate them, but initially we'll be using 0s before going full random.
- Operation: [rd] = [rt] << shamt
    - Shifts the value in rt by shamt bits and stores the result in rd.

Overview of Modification

- For the ALU Decoder, we modify it to recognize the funct code of the sll instruction, and if funct == 000000, the aludecoder emits aluctonrol == 011 and a shift signal that goes to a new multiplexer in the datapath named srcamux.
- As the shift signal goes out of the aludecoder into the controller and then to the mips module, we add a node that handles the wiring of the shift signal to the proper mux in the datapath.
- srcamux selects between the 32-bit zero-extended shamt and the regular 32-bit RD1 (contents of rs). This is controlled by the shift signal. (Currently) outputs to srca of the ALU.

- For the ALU module, we add a shift left functionality, given by if (alucontrol == 011) result = b << a to be considered before the other cases in the . Recall that alucontrol == 011 was a previously unrecognized (i.e. vacant) control signal for the ALU. Adding this in completes the operation line-up for this specific ALU.
- For testing purposes, we also allow the shift signal to reach the testbench itself, so we add output logic wires catering to the shift signal from the aludecoder, to the controller, to mips, and to the top.

Control Signals

The following are the final control signals for this instruction. In yellow are new signals.

Remark: For consistency with the HARRIS implementation, we always take the Don't Care signals to be 0. In this case only sb can be a Don't Care (because it's only checked when memwrite = 1, explained later).

From Main Decoder:

regwrite	regdst	alusrc	beq	memwrite	memtoreg	jump	aluop	sb	ble	li
1	1	0	0	0	0	0	10	0	0	0

From ALU Decoder:

alucontrol	shift	zfr
011 (sll)	1	0

Modifications in the HDL Code

We now explain the HDL modules we modified in order to implement the sll instruction. For coherence, we shall explain starting from the modules at the bottom of the hierarchy and ending with those at the top. More specifically, we will usually take the route of discussing the Control Unit, then the Data Path, a possible back and forth, and also possibly some things in the Top.

maindec.sv

The maindec module, instantiated as md, is the module that handles the bulk of the instruction decoding.

In retrospect, we haven't actually modified the main decoder for the sll instruction and we simply worked with the control signals that are generated by default for R-type instructions (which includes sll with an opcode of 000000), so we do not discuss this here.

aludec.sv

This is where we begin with the modifications for the sll instruction. The aludec module, instantiated as aluldec, generates the signals that govern the ALU. Line-by-line explanations are shown below. The numberings correspond to their actual lines in the code.

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively. This means that any given delay (denoted by $\#\mathrm{N}$) will take $1\mathrm{ns} \times \mathrm{N}$. This also means that the design runs at indivisible time steps of 1ps. In other words, all delay values will be rounded to a time precision of 1ps. This timescale is given to be constant across all designs for our safety. In practice, the smallest observable unit of time in all the simulations will be in steps of 1ps.

2-8: Creates a module named aludec with 6-bit logic-type input port (node) named funct with MSB:LSB = 5:0 , 2-bit logic-type input port named aluop with MSB:LSB = 1:0, 3-bit logic-type output port named alucontrol with MSB:LSB = 2:0, and logic-type output port named shift. We currently ignore zfr as it was added only during the implementation of the zfr custom instruction.

The funct port takes in the funct field of the instruction, the aluop port takes in the generated logic simplifier by the main decoder, the bits of the alucontrol controls the operation of the ALU, and the shift is the **current** main modification here that handles the shift signal.

10: Declares a 5-bit logic type variable named controls. Actually for sll this was only declared as a 4-bit logic type variable, only considering the added bit for the shift control signal.

11: Wires the controls variable with a concatenation of the alucontrol and shift (in that order). Again, ignore the zfr signal for now. Now controls will store the actual control signals generated by the aludec module. These control signals are currently alucontrol and shift.

13: The start of an always_comb block. This kinds of block exemplifies the combinational nature of a circuit. This has an implied sensitivity list that watches for changes in all the variables that the block works with. For each change, the contents of the always_comb block will be executed.

14: The start of a case construct (similar to the C switch-case construct). The execution of this case construct will be dictated by aluop value. As we'll see over and over again, the case construct is a very convenient way of coding for *expected* constant values with predefined behaviors or consequences.

15-17: We first look at the aluop to check for the most common operations defined by aluop = 00 → alucontrol = 010 = add, and aluop = 01 → alucontrol = 110 = sub. The shift signal is going to be 0 for both add and sub dictated by aluop (because of course shift will only be asserted for sll instructions, and maybe in the future we can also assert this for more shift-type instructions). If aluop is neither 00 nor 01, then we look at the funct field by default.

18: The case construct that handles the *indefinite case* of the aluop. This handles R-type instructions.

19-26: Each funct case in all of these lines correspond to a specific R-type instruction. For example, funct = 100000 → controls = 010_0 = add. I won't specify other mappings for brevity except for the sll mapping. Note that it is implicit that the bit literals here correspond to a predefined behavior for the ALU. Also, just pretend the zfr bit is not there for the moment. Moreover, since the default control signals for R-type instructions (which includes sll) generates an aluop of neither 00 nor 01, we will have to look at the funct field for sll (the right thing to do for R-type instructions).

The changes here is that during the implementation of the sll instruction, we expand the width of the bit literal assignments to controls in order to accommodate the shift signal bit. As we can see, the shift signal bit is only asserted for the sll instruction, it is 0 for all else. It is important we either assert or do not assert this because this controls a multiplexer in the data path and we cannot give those multiplexers a don't care select signal. More specifically, funct field of shift = 000000 → controls = 011_1. The rightmost bit is the shift bit.

Also, I would like to mention that the always_comb block uses non-blocking assignments (<=), in other words they are *wiring statements* (similar to assign) that do not block the execution of lines below them, as needed for combinational circuits. We don't have to worry about race conditions here because of how the cases are carefully constructed to avoid dependences.

Also, note that the default case corresponds to when the processor cannot recognize the funct field of the supposedly R-type instruction, so we give don't cares for that (and that is undesired in general).

27: End of the case(funct) construct spanning lines 18-27.

28: End of the case(aluop) construct spanning lines 14-28.

29: End of the always_comb block spanning lines 13-29.

30: End of the aludec module.

Remark: For future explanations, the concept of I/O ports also being nodes or rather variables that the design can also work with internally will just be implied. We will also ramp up to just implying the types, width, and [MSB:LSB] assigment of the nodes in each design. In particular, when we say that a node is declared with width N, then we take it to mean that the node has MSB:LSB of N-1:0.

controller.sv

This module pertains to the control unit and also houses the pcsrc logic. This contains the main decoder and the ALU decoder. The modification here is that we added as output the shift signal because we want to feed it to the datapath.

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively.

2-13: Creates a module named controller and declares the input and output ports (also considered as nodes) of the module. Inputs are 6-bit op and funct (fields), and 1-bit zero flag from the ALU in the datapath. Outputs are 1-bit control signals memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump, and shift (the new one), 3-bits for the alucontrol.

Please ignore anything pertaining to the sb, sign, li, zfr, beq, and ble. We will instead explore them later for their respective instructions.

16: Declares 2-bit logic-type aluop. Wires aluop from maindec to aludec.

17: Originally declared as logic-type branch. Will serve as the control signal for beq.

18: Ignore, return to at ble.

20-22: Instantiates the main decoder as md. Has as input op (the opcode), and as outputs memtoreg, memwrite, beq (originally branch), alusrc, regdst, regwrite, jump, aluop. Ignore sb, ble, li.

24: Instantiates the alu decoder as ad. Inputs are funct and aluop. Outputs are alucontrol, shift, and zfr, as desired. Focus on the shift signal, and take note of how this is one of the signals that I believe can only be decoded within the aludec due to this being R-type, same with zfr.

27: Wires pcsrc to the result of branch AND zero. Note that for the addition of ble this will be expanded to assign pcsrc = (beq & zero) | (ble & (sign | zero));. PCSrc controls which next instruction address will be fed to the instruction memory, whether it be for branch (BTA) or in our situation, jump (JTA) is also possible. These effects are dictated by the appropriate branch and jump control signals that controls the multiplexers pcbrmux and pcmux in the datapath respectively.

28: End of the controller module.

mips.sv

Now we go to the mips module which contains the control unit and the datapath. This is the SCP minus the instruction memory (imem) and the data memory (dmem).

Remark: At this point, we will now stop specifying the type of a node because all things considered, for our situation they're always logic-types. Also, as you may have noticed, if width is not specified, then it is 1-bit.

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively.

2-9: Creates a module named mips with inputs clk (the clock driver), reset (for resetting the PC register), instr (the whole 32-bit instruction), and 32-bit readdata (pertains to result of combinational read of dmem, only actually used for lw). Outputs are pc (from the data path, gives the next PC address, default is PC+4, special cases for branches and jumps), memwrite (from controller, controls whether to write or not to dmem, aluout (because we want to watch aluout for testing), writedata (we also want to watch this).

Additions are the shift output signal (I placed this here because I personally want to watch this). We'll discuss sb later.

12-13: Declares variables memtoreg, alusrc, regdst, regwrite, jump, pcsrc, and zero. Used for wiring between the controller and the datapath, usually controls multiplexers in the datapath, except for the zero flag which is wired from the ALU to the controller to control PCSrc.

14: Declares 2-bit alucontrol, also wired from controller's aludec to the controller.

16-18: Ignore for the moment. They are for ble, li, and zfr.

20-27: Instantiates a controller module and names it c. It has as inputs 6-bit instr[31:26] (opcode) and instr[5:0] (possibly funct), and the zero flag. Output ports are memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump, and 2-bit alucontrol. Aka the control unit.

Additions include new output port shift, for the shift signal.

30-37: Instantiates an datapath module and names it dp. Has as inputs clk, reset, memtoreg pcsrc, alusrc, regdst, regwrite, jump, 2-bit alucontrol, the 32-bit instruction, 32-bit read data chiming in from dmem, and the shift signal for the shift multiplexer. Again, ignore those not related to sll.

38: End of the mips module.

datapath.sv

The datapath is where all the main computations that generates register results take place.

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively.

2-15: Creates a module named datapath. Has as inputs clk, reset, memtoreg, pcsrc, alusrc, regdst, regwrite, jump, 2-bit alucontrol, 32-bit instr, 32-bit readdata.

Remark on Variables: If we do not mention a variable, that means they are not a part of the modifications for this instruction, so please ignore them for the moment.

Additions include the shift signal asserted for when sll is the instruction. So the main changes we did here are for some of the variable names and the addition of a new mux to handle the shamt.

18-25: Declare 5-bit writereg; and 32-bit pcnext, pcnextbr, pcplus4, pcbranch, signimm, signimmsh, RD1, RD2, srca, srcb, liout (new!), result, and zfrout (new!). After some erroneous simulations, I learned the value of first declaring nodes and most importantly their sizes before wiring together the modules using said nodes.

These declarations help guide the wires to connect to the write places, those preceeded with pc deals with the instruction addressing, and the other deal with the main data path.

Changes I did here is to add RD1 and RD2 to correspond to the rs and rt value outputs of the regfile respectively. This is done for clarity.

28: always_comb block start. As explained previously, this watches all variables that are manipulated inside, and executes on changes. I placed this here to watch the changes in dataaddr and writedata more internally (because for the testbench we simply just use comparisons of dataaddr and writedata actual values to their expected values to verify the correctness of the simulation).

31: Displays the dataaddr and writedata contents whenever there are changes with them. Note that dataaddr is wired to alutout and writedata is wired to RD2.

32: End of the always_comb block above.

35: Instantiates flopr as pcreg, parameter WIDTH=32 with inputs clk, reset, 32-bit pcnext (seen as PC' in the HARRIS schematic), and output 32-bit pc. This serves as the pc register which updates every clock rising edge. Flopr is used because of the need for a resettable flipflop (reset required usually for initializing the instruction address). Please refer to my Lab Report 9 for line-by-line explanations of the contents of flopr.

Note that pc goes out to imem to specify the next instruction address to be read.

36: Instantiates adder as pcadd1, parameter WIDTH = 32, with inputs pc, 32’b100 (constant 4), and ‘b0 (0 carry in), and outputs pcplus4 (namesake). Constantly generates PC+4 for *consideration*. Please refer to my Lab Report 9 for line-by-line explanations of the contents of adder.

37: Instantiates sl2 as immsh with input signimm (sign-extended instr[15:0]) and output signimmsh. This shifts the input to the left by 2 (multiplies by 4) as the first step to generating BTA. Please refer to my Lab Report 9 for line-by-line explanations of the contents of sl2.

38: Instantiates adder as pcadd2, WIDTH = 32, with addends pcplus4, signimmsh, and 0 for carry in. Output is pcbranch, for consideration. When we say consideration, we imply that the value is being muxed with something else. This completes BTA = PC+4 + (SignImm << 2). Please refer to my Lab Report 9 for line-by-line explanations of the contents of mux2.

39: Instantiates mux2 as pcbrmux, WIDTH = 32, to choose between 0: pcplus4 and 1: pcbranch. Controlled by pcsrc. Result becomes pcnextbr.

40: Instantiates mux2 as pcmux, WIDTH = 32, to choose between pcnextbr and JTA = {(PC + 4)[31:28], addr, 2’b0}. Controlled by jump.

Remark on Multiplexers: When we say “choose between D1 and D2”, we imply that select signal = 0 passes D1 and select signal = 1 passes D2. This is appropriate because we **only use 2-way multiplexers**. Please keep this remark and all other remarks in mind.

All these complete the choice for the next PC value, whether it be PC+4, BTA, or JTA. Note that we did not made any modifications for this portion, so these explanations will just repeat over and over again, as is.

44: Clarity modification, we wire writedata to RD2. Note that RD2 comes from the register file, and it is wired to WriteData WD of data memory with nothing in between (constantly zero multiplexers in the connection). This is a watched variable.

45-46: Instantiates regfile as rf. Inputs are the clk (for possibly writing sequentially), regwrite signal, instr[25:21] = rs, instr[20:16] = rt, writereg or value to be written, and result coming in from resmux. Outputs are RD1 and RD2. The register file will be written too on clock rising edge if WE3 is asserted by RegWRite.

This regfile is very integral to the processor for as we all know this contains the registers (indexed by their register address) and their respective contents that the processor works with.

Remark on Submitted Modules: Note that all the submitted modules contain annotations that might help you in examining this project documentation. I sincerely hope that you check them out, along with the rest of our exhausting projects. Please, we really worked hard on these projects, especially with respect to the need for line-by-line explanations that we consistently obeyed without sleep and without question, so I beg you to please, please have the time to go through each sentence, each word, of the projects we submitted. Thank you so much for this learning experience.

47-48: Instantiates mux2 as wrmux with WIDTH = 5 (begause register address is always just 5 bits). This chooses between rt and rd over which to use as the write address for the register file. This is controlled by regdst (register destination). Outputs the address to writereg and is fed back to the register file.

49: Ignore limux, this is for li instruction.

50: Instantiates mux2 with width 32 as resmux. Chooses between the output of ALU, aluout, and the data that was combinationally read from dmem, readdata. Controlled by memtoreg. Outputs to result, and that result is fed back to the register file.

51: Instantiates signext as se. Takes in the perceived immediate field instr[15:0] of any instruction and sign-extends the contents to 32-bit, and the result is passed to signimm. Please refer to my Lab Report 9 for line-by-line explanations of the contents of signext.

56: This is an **sll modification**. Instantiates mux2 as srcamux, 32-bit. Chooses between RD1 and the zero-extended contents of the shamt field of the instruction (instr[10:6]). This is controlled by the shift signal. The result of this is passed to srca which is wired to the ALU.

57: Instantiates mux2 as srcbmux, 32-bit. Chooses between RD2 and signimm (sign-extended immediate). Controlled by alusrc. Output passed to srcb. Originally this connects directly to srcb port of the ALU, but this will change for zfr which introduces a new mux, zfrmux, between that connection. Note that previously RD2 was just written as writedata, but that is not so descriptive so we wired writedata and RD1 to each other.

The motivation for lines 56-57 is that we turned that unused alucontrol assignment to a shift operation within the ALU, to be precise, it is result = b << a. Sa if the instruction is shift, srca of ALU receives the desired shift amount stored in the instruction's shamt field, and srcb receives the value to shift from rt.

58: Ignore zfr for now. Hence, we imagine srcb to be directly connected to alu until before zfr is implemented.

59: Instantiates alu as alu. Operand A is srca, and Operand B is aluout (zfrout later on). Controlled by alucontrol. Outputs result to aluout, and also generates zero flag (and sign flag for ble later on).

Per HARRIS, an Arithmetic/Logical Unit (ALU) combines a variety of mathematical and logical operations into a single unit. For example, a typical ALU might perform addition, subtraction, magnitude comparison, AND, and OR operations. In our case, we use a 3-bit control signal alucontrol which specifies 7 default functions or operations (8 if we include the recently added sll instruction for alucontrol 011). The ALU forms the heart of most computer systems, where the alucontrol is like the passive brain signal that directs which artery or multiplexer data source (I know, kind of stretching the analogy here) will the *actual* processed result come from, because most ALU implementations actually computes the respective operation results simultaneously, and the alucontrol just selects which of those results will actually be presented.

61: Ends the datapath module.

alu.sv

This ALU is modified to be able to handle sll by assigning result = b << a to when alucontrol = 011 (a previously unused assignment).

1-5: Creates a module named alu. Inputs 32-bit a and b, and 3-bit alucontrol. Outputs 32-bit result, and zero flag (also sign flag from ble onwards).

8: Declares 32-bit condinvb and 32-bit sum.

Note that if alucontrol[2] = 0, we assume addition, and elif alucontrol[2] = 1, we assume subtraction or slt.

14: Wires condinvb with the result of (alucontrol[2] ? ~b : b). The name condinvb stems from its ternary conditional computational expression that chooses between regular operand B or a negated operand B (hence this can be represented by a mux controlled by alucontrol[2] in the circuit diagram).

Some other remarks are that:

- If alucontrol[2] = 0, condinvb + alucontrol[2] = b.
- Elif alucontrol[2] = 1, condinvb + alucontrol[2] = -b (in 2C, recall –b = ~b + 1).
- 2C Addition of a + (-b) is then essentially a subtraction.
- So sum will instead contain a difference (a-b).
- We also note that despite the availability of the A AND ~B, and A OR ~B functions in the HARRIS ALU operation set, it appears that the original alu.sv isn't coded to perform such operations since condinvb is only confined to the generation of either a true sum or a difference. However, we leave this be because we are not interested in isntructions that do this.

19: Wires sum to result of a + condinvb + alucontrol[2]. In other words, addition or subtraction is always performed regardless of the instruction, the question is now just whether to return that result or not. Sum here can either contain a true sum or a difference depending on alucontrol[2].

23: Start of an always_comb block. Properties explained earlier.

24: This is an **sll modification**. If alucontrol is 011 (corresponding to sll), then ALU returns b << a. Note that a is the shift amount and b is from rt.

25: Else if alu is not 011, then we consider other cases.

26-35: Having already considered alucontrol[2] for sum or difference in lines 14-19, we now just look at alucontrol[1:0]. If it is 00, then apply AND (also done for ZFR later on). If it is 01, then apply OR. If it is 10, then return sum (possibly a true sum or a difference depending on alucontrol[2]). If it is 11, then this is actually for slt, implying alucontrol[2] = 1 (usually do subtraction for comparisons), so we return the sign bit (note that if the difference of a – b is negative, then that means a < b, so sign bit of sum in this case would be 1, which would be the output we desire for slt a b).

These are preset behaviors and we need not explain them further.

We note however that the given , however, we leave them be as we are not so interested in them.

38: Assign zero flag with whether result equals zero. For use with branches.

39: For ble onwards, we also set a sign flag that tells us whether A < B. Again, just taking the sign bit of (A – B) is simpler than actually doing A < B comparison.

40: Ends the module.

top.sv

The reason we also explain top is because I modified this to also output the shift signal from the mips module because we want to watch it for debugging purposes.

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively.

2-5: Creates a module named top. This module combines the mips control unit and data path with the instruction memory and data memory modules. Inputs are simply clk and reset. Clk used to drive the instructions and reset used to set PC register to 0. Outputs are 32-bit writedata and dataadr (for testbench checking), memwrite (again for testbench checking), and shift (optional, for watching out for sll instructions which can serve as checkpoints when looking at the waveforms.

8: Declares 32-bit variables pc, instr, and readdata.

9: Declares sb. For sb and onwards, connects decoded sb signal from mips to dmem.

12-14: Instantiates mips as mips. Inputs are clk and reset. Outputs are pc, instrm ,e,write, dataadr, writedata, and readdata.

Additional outputs include shift for sll onwards (for watching), and sb for sb onwards.

sb is the only new control signal that comes out of mips and into dmem.

15: Instantiates imem as imem with input pc[7:2] and outpuit instr. Only pc[7:2] is taken for word-addressing of instructions.

16-17: Instantiates dmem as dmem with inputs clk, memwrite, dataadr, writedata, and sb (from sb onwards). Output is readdata.

Remark: We will now start omitting information that can be immediately be understood.

18: Ends the top module.

imem.sv

Not modified for any added instruction.

dmem.sv

Not modified for the sll instruction.

Schematic Diagram

Here we show our own diagram of what changed in the processor. Heaviest modification was made on the datapath, hence we will only show it (changes to controller contents already explained above). Moreover, only a portion of the data path will be shown. Note that all changes were done on the regfile logic and alu logic regions of the processor, hence we ignore the pc logic region in general.

Note that the bit width of the wires are implied by the context of the MIPS processor (32-bit words, 6-bit opcode and possibly funct fields, 5-bit rs, rt, and possibly rd and shamt fields, and 16-bit immediates).

Furthermore, note that the control signals and flags of the visible components are implied to be connected to the Control Unit, with the exception of the CLK.

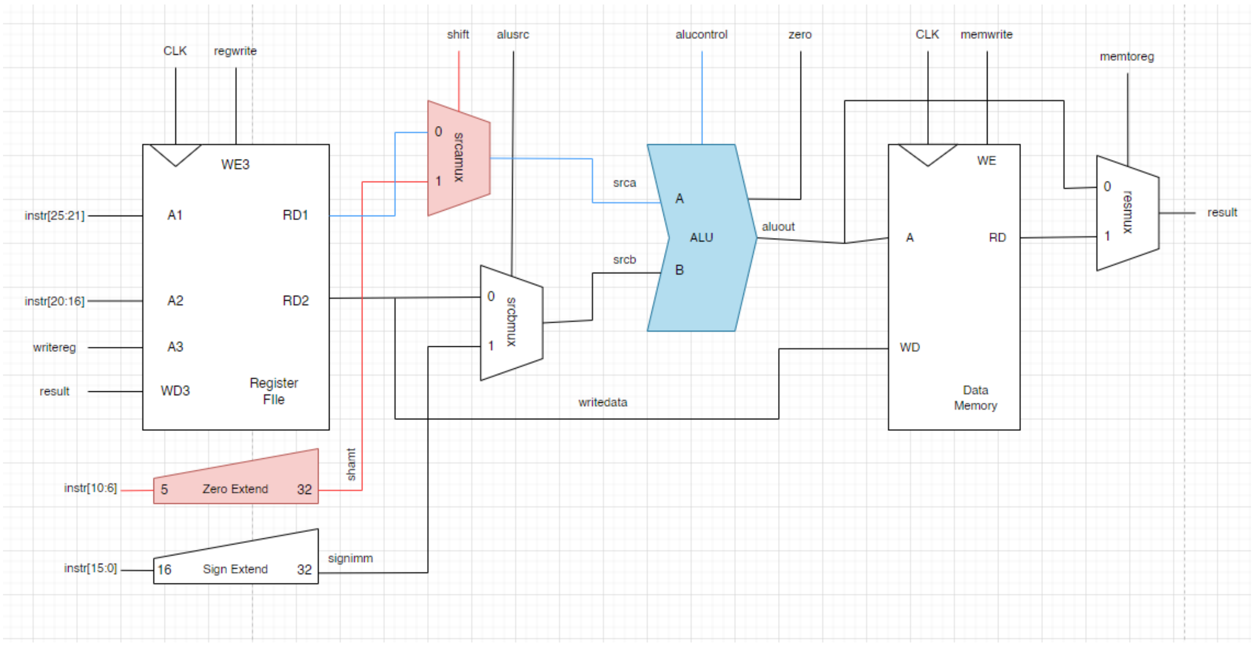


Figure 1. sll schematic diagram (partial view of the datapath only).

Modified components are shown in blue, and new components are shown in red. The new additions above are the zero extender, the srcamux, and the shift signal.

- The zero extender zero-extends the 5-bit contents of the shamt field of the presumably R-type instruction (instr[10:6]) into 32-bits so that it can be muxed with the 32-bit RD1. This is implemented in the code as a direct manipulation of the input to srcamux's d1.
- The srcamux multiplexer chooses between RD1 = [rs] and the now 32-bit shamt for input to srca. Controlled by shift. Recall remark regarding "between" statements for multiplexers.
- The shift signal on the srcamux also makes it apparent how the rs field is ignored by sll.

Implicit additions are for the control unit and the ALU for generation of new alucontrol = 011 and recognition of said alucontrol respectively.

Testbenches and Test Programs

We first use the following two testbenches, testbench.sv and testbench_2.sv, to verify the integrity of the modified processor for the originally supported instructions:

add, sub, and, or, slt, lw, sw, beq, addi, and j.

Many thanks to Lab 12 for generously providing the aforementioned testbenches.

testbench.sv

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

11: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite.

14: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

15: Beginning of the initial block.

16: Sets reset to 1 for the first 22 ns of the simulation, then sets it to 0 afterwards.

17: End of this initial block.

20: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

21: Beginning of this always block.

22: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

23: End of the always block.

26: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

27: Start of this always block.

28: An if conditional that executes its block if memwrite is asserted. Corresponds to initializing the actual test when sw or sb is the instruction.

29-35: CHECKING: If dataadr === 84 & writedata === 7, then simulation is successful (print "Simulation suceeded" then stop the program). Else if dataadr !=== 80, simulation failed (print "Simulation failed" then stop the program). The reason for that failure condition is that in the test program, there's a non-terminating store word with effective address 80, so if it happens that dataaddr === 80 when memwrite == 1, then it means that there's a mix-up in the computations.

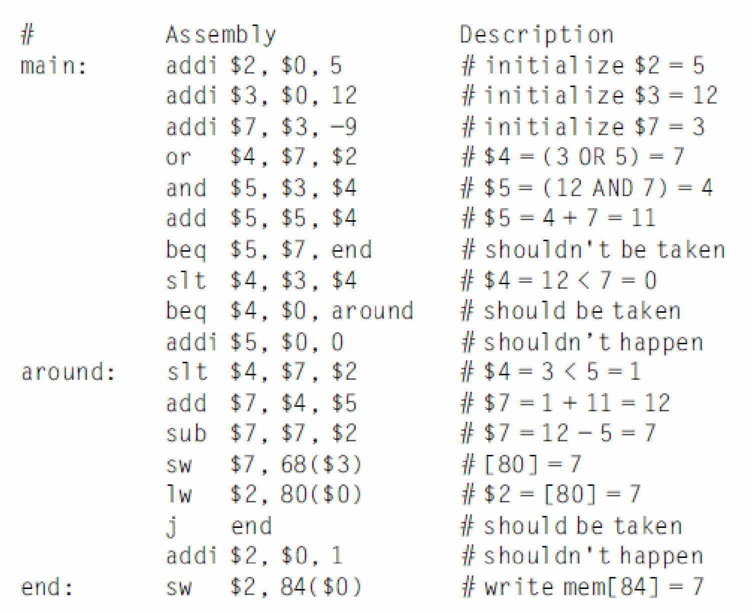
36: End of the memwrite condition block.

37: End of the always block.

38: End of writing the module.

Assembly Program for testbench.sv

Below is the MIPS program that this testbench will run through. Courtesy of HARRIS.



```
#         Assembly               Description
main:     addi $2, $0, 5         # initialize $2 = 5
          addi $3, $0, 12        # initialize $3 = 12
          addi $7, $3, −9        # initialize $7 = 3
          or   $4, $7, $2        # $4 = (3 OR 5) = 7
          and  $5, $3, $4        # $5 = (12 AND 7) = 4
          add  $5, $5, $4        # $5 = 4 + 7 = 11
          beq  $5, $7, end       # shouldn't be taken
          slt  $4, $3, $4        # $4 = 12 < 7 = 0
          beq  $4, $0, around    # should be taken
          addi $5, $0, 0         # shouldn't happen
around:   slt  $4, $7, $2        # $4 = 3 < 5 = 1
          add  $7, $4, $5        # $7 = 1 + 11 = 12
          sub  $7, $7, $2        # $7 = 12 − 5 = 7
          sw   $7, 68($3)        # [80] = 7
          lw   $2, 80($0)        # $2 = [80] = 7
          j    end               # should be taken
          addi $2, $0, 1         # shouldn't happen
end:      sw   $2, 84($0)        # write mem[84] = 7
```

The machine code below is equivalent to the program above. We fill the memory file memfile.mem with this machine code if we want run this testbench.

1. 20020005
2. 2003000c
3. 2067fff7
4. 00e22025
5. 00642824
6. 00a42820
7. 10a7000a
8. 0064202a
9. 10800001

10. 20050000
11. 00e2202a
12. 00853820
13. 00e23822
14. ac670044
15. 8c020050
16. 08000011
17. 20020001
18. ac020054

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):

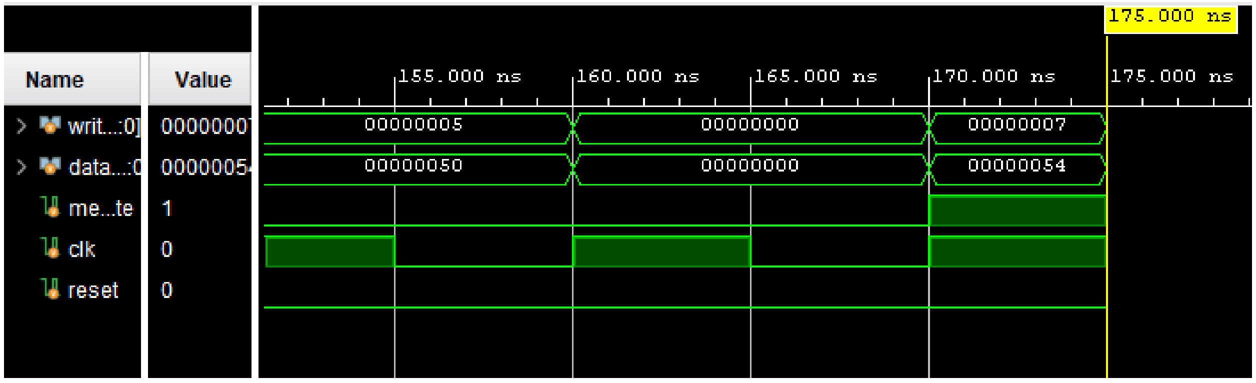


Figure 2. testbench.sv simulation waveform diagram. Do note that we have rearranged the waveform signals above.

As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a "Simulation suceeded" print out in Vivado's TCL console).

testbench_2.sv

This is just a slightly modified version of testbench.sv to accommodate the new test program.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named testbench_2. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

11: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite.

14: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

15: Beginning of the initial block.

16: Sets reset to 1 for the first 5 ns of the simulation, then sets it to 0 afterwards (totally arbitrary). Adds a delay of 255 ns that corresponds to the actual runtime of a successful simulation. This explicitly states that reset = 0 for that time frame.

17: End of this initial block.

20: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

21: Beginning of this always block.

22: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

23: End of the always block.

26: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

27: Start of this always block.

28: An if conditional that executes its block if memwrite is asserted. Corresponds to initializing the actual test when sw or sb is the instruction.

Remark: We are now only concerned in getting the Simulation Success Conditions so we leave the Simulation Failed Condition as simply the result of not attaining the simulation success condition.

29-35: CHECKING: If dataadr === 16 & writedata === 0xBBAAB0B0, then simulation is successful (print “Simulation suceeded” then stop the program). Else, simulation failed (print “Simulation failed” then stop the program).

36: End of the memwrite condition block.

37: End of the always block.

38: End of writing the module.

Assembly Program for testbench_2.sv

```
1.  # Write the hex value 0xBBAA0000 to register 1
2.  addi    $1, $0, 0xBBAA  # 2001BBAA
3.  add     $1, $1, $1      # 00210820
4.  add     $1, $1, $1      # ...
5.  add     $1, $1, $1
```

```
6.  add     $1, $1, $1
7.  add     $1, $1, $1
8.  add     $1, $1, $1
9.  add     $1, $1, $1
10. add     $1, $1, $1
11. add     $1, $1, $1
12. add     $1, $1, $1
13. add     $1, $1, $1
14. add     $1, $1, $1
15. add     $1, $1, $1
16. add     $1, $1, $1
17. add     $1, $1, $1
18. add     $1, $1, $1
19. # Others
20. addi    $2, $0, 176       # 200200B0
21. addi    $3, $2, 45056     # 2043B000
22. addi    $4, $0, 0x7FFF    # 20047fff
23. add     $4, $4, $4        # 00842020
24. addi    $4, $4, 1         # 20840001
25. and     $3, $3, $4        # 00641824
26. add     $4, $1, $3        # 00232020
27. addi    $5, $0, 16        # 20050010
28. sw      $4, 0($5)         # aca40000
```

Explanation:

1: Insert 0x0000 BBAA to register 1.

2-18: Repeatedly add it to itself 16 times in order to emulate a load upper immediate (lui) instruction. After this register 1 contains 0xBBAA 0000.

20: Insert value 176 to register 2.

21: $3 = $2 + 45056 = 176 + 45056.

22: Insert 0x0000 7FFF to $4.

23. Add $4 to itself and store result to $4.

24: Add a 1 to the content of register 4 and store result to register 4.

25: $3 = $3 AND $4.

26: $4 = $1 + $3

27: Store value 16 to $5.

28: Store word content of $4 (0xBBAAB0B0 after all the previous computations) to effective address given by $5 (16).

The above program correspond to the machine code below:

1. 2001bbaa
2. 00210820
3. 00210820
4. 00210820
5. 00210820
6. 00210820
7. 00210820
8. 00210820
9. 00210820
10. 00210820
11. 00210820
12. 00210820
13. 00210820
14. 00210820
15. 00210820
16. 00210820
17. 00210820
18. 200200b0
19. 2043b000
20. 20047fff
21. 00842020
22. 20840001
23. 00641824
24. 00232020
25. 20050010
26. aca40000

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):

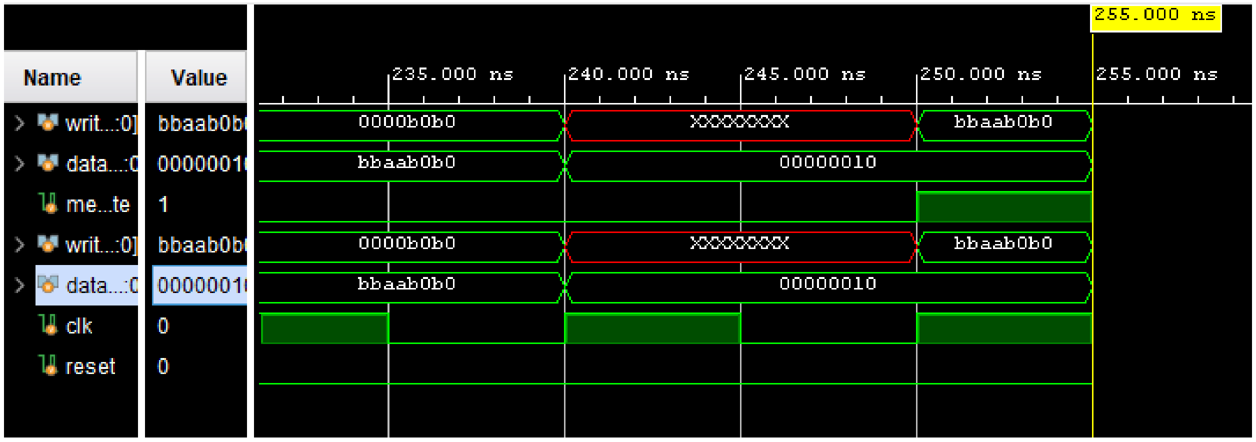


Figure 3. testbench_2.sv simulation waveform diagram.

As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a “Simulation suceeded” print out in Vivado’s TCL console).

sll_testbench.sv (new!)

The SystemVerilog code for the sll_testbench.sv module is provided below.

```
`timescale 1ns / 1ps

module sll_testbench();


 logic      clk;

 logic      reset;


 logic [31:0] writedata, dataadr;

 logic      memwrite;

 // NOTE, watching shift from mips to top

 logic      shift;


 // instantiate device to be tested

 top dut(clk, reset, writedata, dataadr, memwrite, shift);
```

```
// initialize test
initial
  begin
    reset <= 1; #1;
    reset <= 0;
    #99 $finish;
  end

// generate clock to sequence tests
always
  begin
    clk <= 1; # 5; clk <= 0; # 5;
  end

// check results
always @(negedge clk)
  begin
    if(memwrite) begin
      if(dataadr === 84 & writedata === 'h0C00000C) begin
        $display("Simulation succeeded");
        $stop;
      end else begin
        $display("Simulation failed");
        $stop;
      end
    end
  end
```

endmodule

This is the main testbench for sll. This is just an appropriation of the previous testbenches.

The explanations for the sll_testbench.sv code above are provided below.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named sll_testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

10: Declared a watcher for the shift signal.

13: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite and also shift.

16: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

17: Beginning of the initial block.

18-20: Sets reset to 1 for the first 1 ns of the simulation, then sets it to 0 afterwards. This initializes the PC register to instruction address 0. After 99 ns end the program, corresponding to actual runtime of program (we could just remove this).

21: End of this initial block.

24: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

25: Beginning of this always block.

26: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

27: End of the always block.

30: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

31: Start of this always block.

32: An if conditional that executes its block if memwrite is asserted. Corresponds to initializing the actual test when sw or sb is the instruction.

Remark: We are now only concerned in getting the Simulation Success Conditions so we leave the Simulation Failed Condition as simply the result of not attaining the simulation success condition.

33-39: CHECKING: If dataadr === 84 & writedata === 0x0C00000C (expected results, see assembly program explanation or annotations), then simulation is successful (print "Simulation suceeded" then stop the program). Else, simulation failed (print "Simulation failed" then stop the program).

40: End of the memwrite condition block.

41: End of the always block.

42: End of writing the module.

Assembly Program for sll_testbench.sv

For the sll_testbench.sv, we have a relatively simple test program, as follows:

1. addi $s0, $0, 0x000C
2. sll $s0, $s0, 8
3. sll $s0, $s0, 8
4. sll $s0, $s0, 8
5. addi $s0, $s0, 0x000C
6. sw $s0, 84($0)

Explanation:

1: $s0 = 0 + 0x0000 000C = 0x0000 000C

2-3: SLL TESTING: 0x0000 000C should be shifted by 24 bits to the left in total. Result would be 0x0C00 0000.

4: Then add 0x000C to 0x0C00 0000 and store that back to $s0. $s0 should be 0x0C00 000C at this point.

5: Store word contained by $s0 to effective address 84 (may break MARS but for the Vivado simulation it's just fine). $s0 must contain 0x0C00 000C for this simulation to be successful.

The corresponding machine code for this is simply

1. 2010000c
2. 00108200
3. 00108200
4. 00108200
5. 2210000c
6. ac100054

Said machine code is as is and did not need any adjustments to work in the context of the memory file and the processor. Moreover, recall that the **rs field** is a **Don't Care** for sll; in our case, we simply set the rs field to 0s, but for future instructions we also use variations of bits for the don't cares. Also, by design, we can be assured that the rs field of sll is really treated as a don't care because our shift signal selects between the rs contents and the zero-extended shamt, so it's just either one of the two.

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):

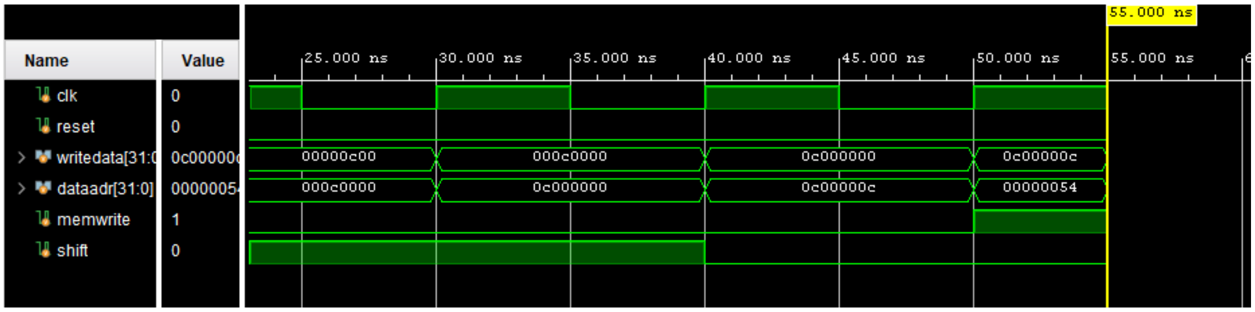


Figure 4. sll_testbench.sv simulation waveform diagram.

As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a "Simulation suceeded" print out in Vivado's TCL console).

All these testbenches were successfully passed by our modified processor. Thus we confirm that this processor is now fully capable of the **sll rd, rt, shamt** instruction as it was specified in HARRIS Appendix B.

Remark on Captions for Waveform Figures: In order to retain the sense of adding figure labels to our attached waveform diagrams, we will only be adding figure labels at waveforms found at their respective instruction documentations so as to avoid repeated labels.

Important Remark on Assembly Program Explanations: From now on, we will be doing the line-by-line explanations of our assembly programs as **concise line-by-line annotations on the programs themselves** that focus on the **execution flow** and resulting **destination register contents** of each instruction, like what HARRIS did. The behavior of the instructions and what registers they are operating on will be implied by the specific instruction being annotated.

Remark on Waveform Diagram Radix: We will be using hexadecimal as the default radix for our waveform diagrams.

Remark on Line Numbers: We will now stop adding line numbers due to formatting difficulties encountered with such a lengthy document, especially with respect to breaking the paper apart into multiple sections (Word > Page Layout > Breaks > Section Breaks) in an attempt to introduce proper, non-overlapping line numbers (as I did with previous Lab Reports). Furthermore, the use of such section breaks messed-up the headings navigation of the document, so I believe it's time to forego them. Rest assured that the line-by-line explanations here correspond to the actual line numbers seen at the IDEs I used.

And from my perspective the code is actually more readable without the line numbers, and I imagine that the non-inclusion of line numbers would make the code blocks here more amenable to the reader's own testing since they can copy-paste the code without having to remove line numbers (as this will be submitted as a pdf file). Thank you for your kind understanding.

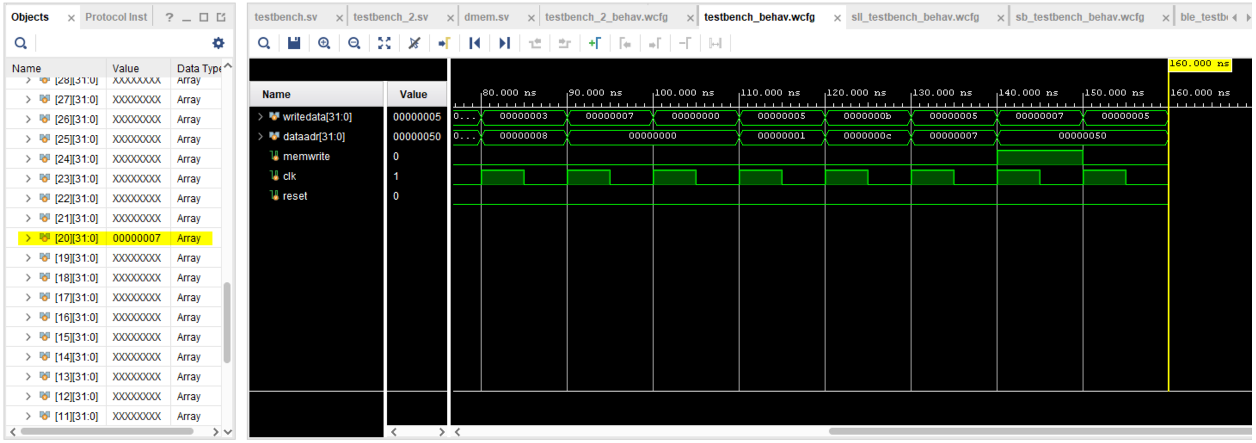
sb rt, imm(rs)

Instruction Format

The **store byte** or **sb** instruction is an I-type instruction with the following properties:

- Coded in MIPS Assembly as sb rt, imm(rs).
- Opcode = 101000
- No funct field, as with all I-types.
- All I-type fields are used.
- Operation: $[\text{Address}]_{7:0} = [\text{imm} + [\text{rs}]]_{7:0} = [\text{rt}]_{7:0}$
    - Stores the lowest order byte of [rt] to the byte address specified by the effective address imm + [rs].
    - Note that in the Data Memory, the variable wd's (write data) lowest order byte (relative to the general functionality of the processor) corresponds to the data to be written. I.e. the byte data to be written is given by wd[7:0].
    - Since we will now be accessing the Data Memory at the byte-level, we will have to store the desired byte with respect to **both** the word address given by a[31:2] and the byte offset given by a[1:0]. This is similar to the use of block addressing in caches (but note that we are not storing entire blocks!), with a side note that we are still essentially doing word-addressing. Moreover, we will respect **Big Endian** addressing **for the Data Memory**. To be more precise:
        - a[1:0] = 00 corresponds to RAM[a[31:2]][31:24].
        - a[1:0] = 01 corresponds to RAM[a[31:2]][23:16].
        - a[1:0] = 10 corresponds to RAM[a[31:2]][15:8].
        - a[1:0] = 11 corresponds to RAM[a[31:2]][7:0].

Note that since the original HARRIS MIPS processor (and its accompanying instructions, especially the load and store word instructions) is designed with Little Endian in mind, supported by HARRIS' remark that "we



Name	Value	Data Type
[28][31:0]	XXXXXXXX	Array
[27][31:0]	XXXXXXXX	Array
[26][31:0]	XXXXXXXX	Array
[25][31:0]	XXXXXXXX	Array
[24][31:0]	XXXXXXXX	Array
[23][31:0]	XXXXXXXX	Array
[22][31:0]	XXXXXXXX	Array
[21][31:0]	XXXXXXXX	Array
[20][31:0]	00000007	Array
[19][31:0]	XXXXXXXX	Array
[18][31:0]	XXXXXXXX	Array
[17][31:0]	XXXXXXXX	Array
[16][31:0]	XXXXXXXX	Array
[15][31:0]	XXXXXXXX	Array
[14][31:0]	XXXXXXXX	Array
[13][31:0]	XXXXXXXX	Array
[12][31:0]	XXXXXXXX	Array
[11][31:0]	XXXXXXXX	Array

Name	Value
writedata[31:0]	00000005
dataadr[31:0]	00000050
memwrite	0
clk	1
reset	0

```
writedata: 0... | 00000003 | 00000007 | 00000000 | 00000005 | 0000000b | 00000005 | 00000007 | 00000005
dataadr:   0... | 00000008 | 00000000 | 00000001 | 0000000c | 00000007 | 00000050
```

```
sw   $7, 68($3)        # [80] = 7
lw   $2, 80($0)        # $2 = [80] = 7
```

Figure 5. Original HARRIS MIPS program isntructions corresponding to sw and lw at cycles 15-16 at the accompanying testbench. Note how the store is done in Little Endian (so we have 0x00000007 instead of 0x07000000 at RAM[20]).

will use little-endian format whenever byte ordering matters" (p. 304), as can be seen here in the execution of the sw and lw instruction at cycles 15-16 of testbench.sv's waveform diagram along with the data memmory contents (RAM):

In order to maintain the integrity of our original instructions with respect to the original testbench and test programs (and HARRIS), then our use of Big Endian for the Data Memory sb instruction is implemented in such a way to not affect our original lw and sw instruction behaviors to avoid any potential conflicts in our processor's byte addressing.

We also keep in mind that endianness only affects byte-addressing, not the word-addressing.

Overview of Modification

- No modification was made to the datapath, only modified were the path from maindec to the data memory, and the data memory itself. Also, aludec is not modified, the addition aluop from maindec is enough.
- Originally, the data memory can only handle storing word instructions. So we modified it by implementing a new nested case construct controlled by a new **sb signal** originating from the decoder.
- This case construct separates the sw and the new sb functionality using the following conditions:
  - If sb == 0: Do store word. Consider only word address and the entire wd.
  - If sb == 1: Do store byte. Consider both word address and byte offset given by a.
- Now, if sb is asserted, then again we look at the word address a[31:2] of the effective address. We use this to access the desired RAM or data memory element (recall that the RAM is declared as an array of 64 32-bit elements). Once we locate the word address, we then access the byte offset given by a[1:0] and store wd[7:0] at that location.
- Motivation for the a[31:2] and a[1:0] nuance is that for word addressing, the first 2 bits are essentially 00 all the time, and after accessing the 32-bit elements of the RAM, a[1:0] can now specify which byte we will access.

Control Signals

The following are the final control signals for this instruction. In yellow are new signals.

Remark: For consistency with the HARRIS implementation, we always take the Don't Care signals to be 0 in the code. In this case, regdst and memtoreg are in fact Xs but taken as 0s. sb is also X for all instructions where memwrite = 0 by nature of the dmem write conditional.

From Main Decoder:

regwrite	regdst	alusrc	beq	memwrite	memtoreg	jump	aluop	sb	ble	li
0	0	1	0	1	0	0	00	1	0	0

From ALU Decoder:

alucontrol	shift	zfr
010 (add)	0	0

Modifications in the HDL Code

We now explain the HDL modules we modified in order to implement the sb instruction. This time we did not modify the datapath so we will not discuss it. We begin with the main decoder module and end at the data memory module.

maindec.sv

The maindec module, instantiated as md, is the module that handles the bulk of the instruction decoding. Line-by-line explanations are shown below. The numberings correspond to their actual lines in the code.

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively. This means that any given delay (denoted by $\#\mathrm{N}$) will take $1\mathrm{ns} \times \mathrm{N}$. This also means that the design runs at indivisible time steps of 1ps. In other words, all delay values will be rounded to a time precision of 1ps. This timescale is given to be constant across all designs for our safety. In practice, the smallest observable unit of time in all the simulations will be in steps of 1ps.

2-11: Creates a main decoder module named maindec with the following ports:

- 6-bit logic-type input port (node) named op with [MSB:LSB] = [5:0]. Used for accepting the instruction's opcode.
- 1-bit logic-type output ports named memtoreg, memwrite, beq, alusrc, regdst, regwrite, and jump. These are control signals that will then go on to dictate the flow of the datapath,
- 2-bit logic-type output port aluop with [MSB:LSB] = [1:0]. Is fed to the ALU decoder module that helps simplify the decoding of the ALU operation.
- The output ports **sb** (for sb onwards), ble (for ble onwards), and li (for li onwards) to handle the signals corresponding to them.

This module will be a subset of the control unit.

14: Declares a 12-bit logic-type variable named controls with [MSB:LSB] = [11:0]. This will be used to handle all the control signals generated by the main decoder as a concatenated, convenient 12-bit wire value.

- Originally, this was just a 9-bit value without the sb, ble, and li signals.
- After adding **sb**, this became a 10-bit value.
- After adding ble, this became an 11-bit value.
- After adding li, this became a 12-bit value.

16-17: Wirings of the proper control signals using a concatenation operator for ease of assignment. This kind of formatting is useful for simultaneously assigning multiple values that goes to different components, such as what happens with the control signals.

22: Start of an always_comb block for the combinational assignment of control signals.

23-34: A case block controlled by the opcode op. Control signals are predefined for each recognized instructions. As can be seen, R-type corresponds to opcode == 000000, and the rest are R-types.

Additions here are:

- 6'b101000: controls <= 12'b001010000_100; // **SB** (sb = 1) for SB onwards.
- 6'b011111: controls <= 12'b000000001_010; // BLE (beq=0, ble=1) for BLE onwards.
- 6'b010001: controls <= 12'b100000000_001; // LI (regwrite=1, li=1) for LI onwards.

Our main focus here is the SB addition. SB corresponds to opcode 101000, hence the way it's decided in the case block. The SB control signals are very similar to the SW control signals owing to the fact that they are both "store-type" instructions. The difference between the two is that for sw, the sb signal is not aserted, and for sb, the sb signal is asserted. As can be seen, only the SB mapping has sb asserted. Other control signals for sb were already mentioned earlier.

Also worth noting is that unlike the other control signals that goes into the datapath, the sb control signal is unique in that it instead goes to the Data Memory to dictate which store behavior to take.

35: Ends the writing of the maindec module.

controller.sv

This module pertains to the control unit and also houses the pcsrc logic. This contains the main decoder and the ALU decoder.

The current main modification here is the addition of the output sb wiring from the Main Decoder to outside Controller because we want to feed it to the Data Memory module.

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively.

2-13: Creates a module named controller and declares the input and output ports (also considered as nodes) of the module. Inputs are 6-bit op and funct (fields), and 1-bit zero flag from the ALU in the datapath. Outputs are 1-bit control signals memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, and jump; and 3-bits for the alucontrol.

Port modifications are as follows:

- Output shift signal from sll onwards. From alu decoder, goes to srcamux.
- Output **sb** signal from sb onwards. From main decoder, goes to dmem module.
- Input sign flag from ble onwards. From datapath's ALU, goes to this controller.
- Output li signal from li onwards. From main decoder, goes
- Output zfr signal from zfr onwards. From alu decoder, goes to zfrmux.

16: Declares 2-bit logic-type aluop. Wires aluop from maindec to aludec.

17: Originally declared as logic-type branch but is now beq from beq implementation onwards. Will serve as the control signal for beq.

18: Declares ble from ble implementation onwards. Will serve as control signal for ble.

Note that beq and ble are not declared as ports because they are only internally used by the controller, together with the zero flag and the sign flag (from ble onwards) to determine the pcsrc control signal (i.e. whether to branch or not).

Remark on Supplementing Retrospective Explanations: We will now start talking about *currently unimplemented instructions* in a precognizant manner for consistency, this remark shall override any previous conflicting remarks.

20-22: Instantiates the main decoder as md. Has as input op (the opcode), and as outputs memtoreg, memwrite, beq (originally branch), alusrc, regdst, regwrite, jump, aluop.

Additions are the outputs sb (from sb onwards), ble (from ble onwards), and li signals (from li onwards).

Focus on the sb signal which again works with the data memory module.

24: Instantiates the alu decoder as ad. Inputs are funct and aluop. Outputs are alucontrol, shift, and zfr, as desired.

Note that shift and zfr are generated in this module because they are R-type instructions, and they can only be distinguished from each other by examining the funct field since they all have the same 000000 opcode (while also conveniently maintaining the proper control signals for R-type instructions as given by the main decoder).

Generation of sb signal is in maindec, not in the aludec.

27: Wires pcsrc to the result of branch AND zero, i.e. we branch when the instruction is in fact branch and the contents of the two registers are equal (given by their difference being zero).

Note that for the addition of ble this will expanded to be assign pcsrc = (beq & zero) | (ble & (sign | zero)). In other words, there are now two possibilities for branching: when beq or when ble. This Boolean expression is generated by the simple expression that the processor should:

Branch if the instruction is beq and the difference of [rs] and [rt] is zero (meaning [rs] == [rt]), or branch if the instruction is ble and the difference of [rs] and [rt] is negative (meaning [rs] < [rt]) or is zero (meaning [rs] == [rt]). Essentially read as branch if equal or branch if less than or equal. Note that

$$\text{pcsrc} = \text{beq} \,\&\, \text{zero} \mid (\text{beq} \,\&\, \text{sign} \mid \text{beq} \,\&\, \text{zero}) = \text{beq} \,\&\, \text{zero} \mid (\text{beq} \,\&(\text{sign} \mid \text{zero}))$$

The signal PCSrc controls which next instruction address will be fed to the instruction memory, whether it be for branch (BTA) or in our situation, jump (JTA) is also possible.

28: End of writing the controller module.

mips.sv

Now we go to the mips module which contains the control unit and the datapath. This is the SCP minus the instruction memory (imem) and the data memory (dmem). In essence, the mips module serves as relay between the controller and the datapath and this allows the two components to communicate.

We added an sb wiring here.

1: Sets the module’s delay unit and delay resolution to 1ns and 1ps respectively.

2-9: Creates a module named mips with inputs clk (the clock driver), reset (for resetting the PC register), instr (the whole 32-bit instruction), and 32-bit readdata (pertains to result of combinational read of dmem, only actually used for lw). Outputs are pc (from the data path, gives the next PC address, default is PC+4, special cases for branches and jumps), memwrite (from controller, controls whether to write or not to dmem, aluout (because we want to watch aluout for testing), writedata (we also want to watch this).

Additions are the shift output signal (I placed this here because I personally want to watch this).

sb is added as output port from sb onwards. We still want it to go out of this module because it should head to data memory.

12-13: Declares variables memtoreg, alusrc, regdst, regwrite, jump, pcsrc, and zero. Used for wiring between the controller and the datapath, usually controls multiplexers in the datapath, except for the zero flag which is wired from the ALU to the controller to control PCSrc.

14: Declares 2-bit alucontrol, also wired from controller's aludec to the controller.

16-18: Declares the wirings for

- the sign flag (from ble onwards), goes from datapath to controller.
- the li signal (from li onwards), goes from controller to datapath.
- and the zfr signal (from zfr onwards). goes from controller to datapath.

20-27: Instantiates a controller module and names it c. It has as inputs 6-bit instr[31:26] (opcode) and instr[5:0] (possibly funct), and the zero flag. Output ports are memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump, and 2-bit alucontrol. Aka the control unit.

Port additions are:

- output shift signal, for controlling srcamux (sll onwards).
- output sb signal, for controlling dmem store (sb onwards).
- input sign flag, for deciding pcsrc (ble onwards).
- output li signal, for controlling limux (li onwardsd).
- output zfr signal, for controlling zfrmux (zfr onwards).

Focus on sb and how it goes from controller to outside.

30-37: Instantiates an datapath module and names it dp. Has as inputs clk, reset, memtoreg pcsrc, alusrc, regdst, regwrite, jump, 2-bit alucontrol, the 32-bit instruction, 32-bit read data chiming in from dmem, and the shift signal for the shift multiplexer.

Port additions are (some may be reiterated):

- input shift signal, for controlling srcamux (sll onwards).
- output sign flag, for deciding pcsrc (ble onwards).
- input li signal, for controlling limux (li onwardsd).
- input zfr signal, for controlling zfrmux (zfr onwards).

38: End of the mips module.

top.sv

This module serves as the interface between the controller plus the datapath, and instruction memory plus data memory.

The reason we discuss this for sb is because this module wires the sb signal from mips to dmem.

1: Sets the module’s delay unit and delay resolution to 1ns and 1ps respectively.

2-5: Creates a module named top. This module combines the mips control unit and data path with the instruction memory and data memory modules. Inputs are simply clk and reset. Clk used to drive the instructions and reset used to set PC register to 0. Outputs are 32-bit writedata and dataadr (for testbench checking), memwrite (again for testbench checking), and shift (optional, for watching out for sll instructions which can serve as checkpoints when looking at the waveforms.

8: Declares 32-bit variables pc, instr, and readdata.

9: Declares sb. For sb and onwards, connects decoded sb signal from mips to dmem.

12-14: Instantiates mips as mips. Inputs are clk and reset. Outputs are pc, instrm ,e,write, dataadr, writedata, and readdata.

> Additional outputs include shift for sll onwards (for watching), and sb for sb onwards (for deciding the store case, whether it be store word (sb = 0), or store byte (sb = 1)).
>
> sb is the only new control signal that comes out of mips and into dmem.

15: Instantiates imem as imem with input pc[7:2] and outpuit instr. Only pc[7:2] is taken for word-addressing of instructions.

16-17: Instantiates dmem as dmem with inputs clk, memwrite, dataadr, writedata, and sb (from sb onwards). Output is readdata.

18: Ends the writing of the top module.

dmem.module

This is where the major modifications took place for store byte.

As mentioned, this was previously only able to handle store word, but we modify it to handle store byte and be able to recognize the byte offset to store to.

This will be the only time we discuss the dmem module since this remains the same for all other purposes.

1: Sets the module’s delay unit and delay resolution to 1ns and 1ps respectively.

2-6: Creates a module named dmem with inputs clk and we (coming from top which it gets from the testbench); 32-bit address a and write data wd. Outputs 32-bit read data rd which are only considered for load word instructions.

Also now has the sb signal as input.

10: Declares an array of 64 32-bit elements and calls it RAM. This will be the actual memory component of this module.

13: This line corresponds to the reading of the contents of the word address given by a[31:2]. As can be seen, this statement reads the memory combinationally, but it's the resmux in datapath controlled by the memtoreg signal that gets to decide whether or not to use the value this outputs.

First, note again that a[31:2] gives the word address and a[1:0] gives the byte offset for that word address. This will be an important concept for store byte.

20: Start of an always_ff block that is sensitive posedge clk. This corresponds to a positive edge triggered flipflop, thus the processor only considers writing to the memory sequentially on every clock rising edge.

21: An if conditional controlled by the write enable we signal. If we == 0, we don't write. If we == 1, we write, and proceed to the if block.

22: Now if the execution manages to get here, it means that we will be writing, we now next look at sb.

23: If sb == 0, then store word. This is facilitated by chucking in the whole wd contents to the RAM word address given by a[31:2]

24: If sb == 1, then store byte. In which case we also look at the byte offset a[1:0].

- If a[1:0] == 00, then store wd[7:0] (lowest order byte of wd) at byte 3 of the accessed word.
- If a[1:0] == 01, then store wd[7:0] (lowest order byte of wd) at byte 2 of the accessed word.
- If a[1:0] == 10, then store wd[7:0] (lowest order byte of wd) at byte 1 of the accessed word.
- If a[1:0] == 11, then store wd[7:0] (lowest order byte of wd) at byte 0 of the accessed word.

Note that the above behavior is motivated by the Big Endian byte addressing that we assumed for the Data Memory. A recall of Big Endian applied to Data Memory contents is shown below:

Bit Numbers	31 24	23 16	15 8	7 0
Word Contents				
Byte Offset	0	1	2	3

Schematic Diagram

We present here the changes in the datapath to implement the sb instruction. Modifications from the previous version are shown in blue, while additions are shown in red.

Once again, only a portion of the datapath will be shown for brevity.

Implications as with the sll schematic carries over.

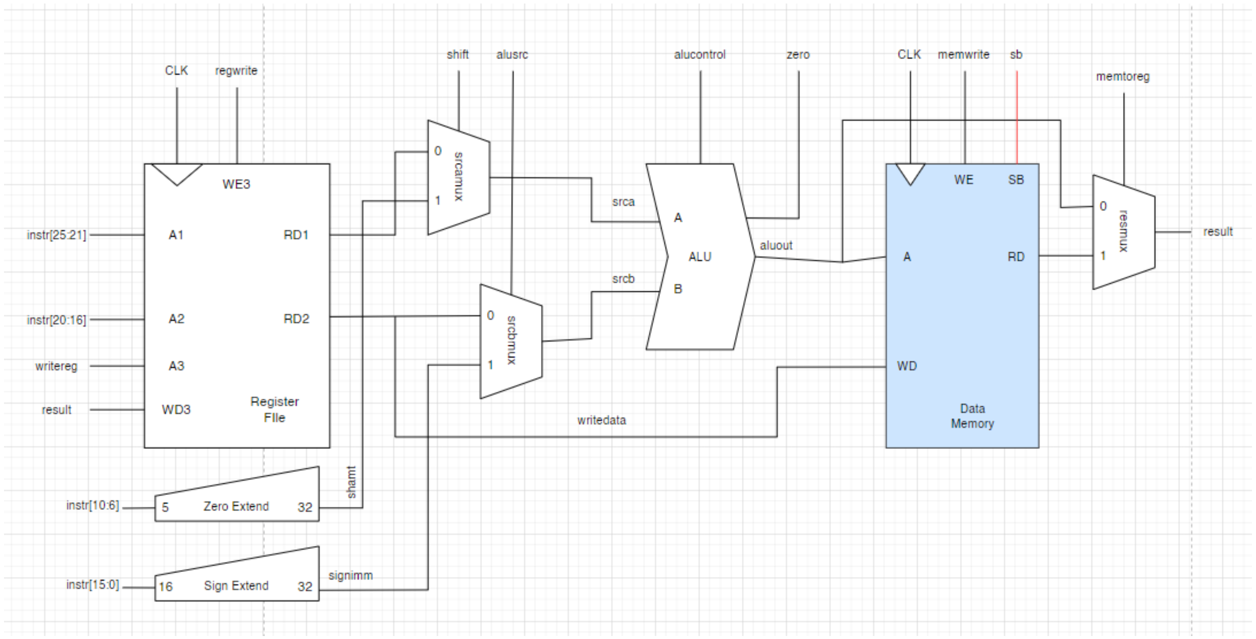


Figure 6. sb schematic diagram (partial view of the datapath only).

The new addition here is the sb signal for the data memory (implied to becoming from the Control Unit's main decoder), and as discussed previously, we modified data memory to accommodate store byte requests.

Testbenches and Test Programs

Once again, like with sll, and like with all else, we reuse testbench.sv and testbench_2.sv for verifying the integrity of the original set of recognized instructions. We also reuse sll_testbench.sv here to verify if we didn't break sll while we were trying to implement sb.

testbench.sv

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

11: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite.

14: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

15: Beginning of the initial block.

16: Sets reset to 1 for the first 22 ns of the simulation, then sets it to 0 afterwards.

17: End of this initial block.

20: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

21: Beginning of this always block.

22: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

23: End of the always block.

26: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

27: Start of this always block.

28: An if conditional that executes its block if memwrite is asserted. Corresponds to initializing the actual test when sw or sb is the instruction.

29-35: CHECKING: If dataadr === 84 & writedata === 7, then simulation is successful (print "Simulation suceeded" then stop the program). Else if dataadr !=== 80, simulation failed (print "Simulation failed" then stop the program). The reason for that failure condition is that in the test program, there's a non-terminating store word with effective address 80, so if it happens that dataaddr === 80 when memwrite == 1, then it means that there's a mix-up in the computations.

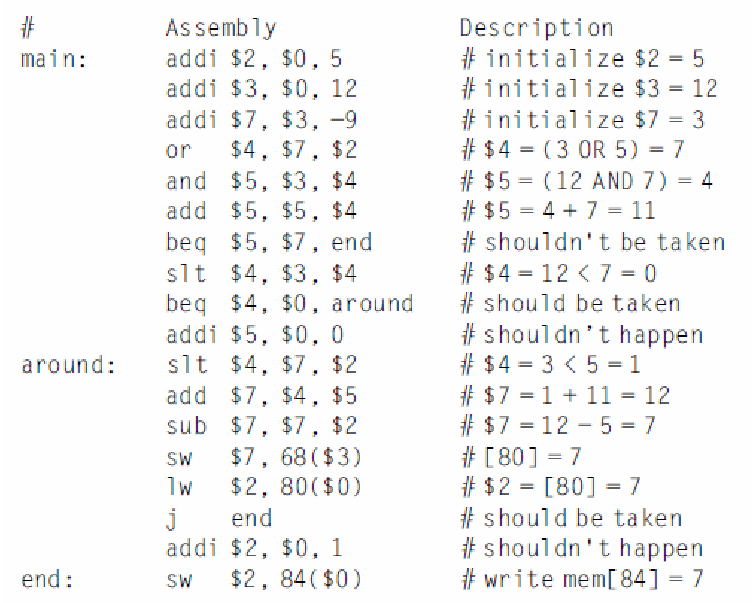
36: End of the memwrite condition block.

37: End of the always block.

38: End of the testbench module.

Assembly Program for testbench.sv

Below is the MIPS program that this testbench will run through. Courtesy of HARRIS.



```
#          Assembly              Description
main:      addi $2, $0, 5        # initialize $2 = 5
           addi $3, $0, 12       # initialize $3 = 12
           addi $7, $3, -9       # initialize $7 = 3
           or   $4, $7, $2       # $4 = (3 OR 5) = 7
           and  $5, $3, $4       # $5 = (12 AND 7) = 4
           add  $5, $5, $4       # $5 = 4 + 7 = 11
           beq  $5, $7, end      # shouldn't be taken
           slt  $4, $3, $4       # $4 = 12 < 7 = 0
           beq  $4, $0, around   # should be taken
           addi $5, $0, 0        # shouldn't happen
around:    slt  $4, $7, $2       # $4 = 3 < 5 = 1
           add  $7, $4, $5       # $7 = 1 + 11 = 12
           sub  $7, $7, $2       # $7 = 12 - 5 = 7
           sw   $7, 68($3)       # [80] = 7
           lw   $2, 80($0)       # $2 = [80] = 7
           j    end              # should be taken
           addi $2, $0, 1        # shouldn't happen
end:       sw   $2, 84($0)       # write mem[84] = 7
```

The machine code below is equivalent to the program above. We fill the memory file memfile.mem with this machine code if we want run this testbench.

20020005

2003000c

2067fff7

00e22025

00642824

00a42820

10a7000a

0064202a

10800001

20050000

00e2202a

00853820

00e23822

ac670044

8c020050

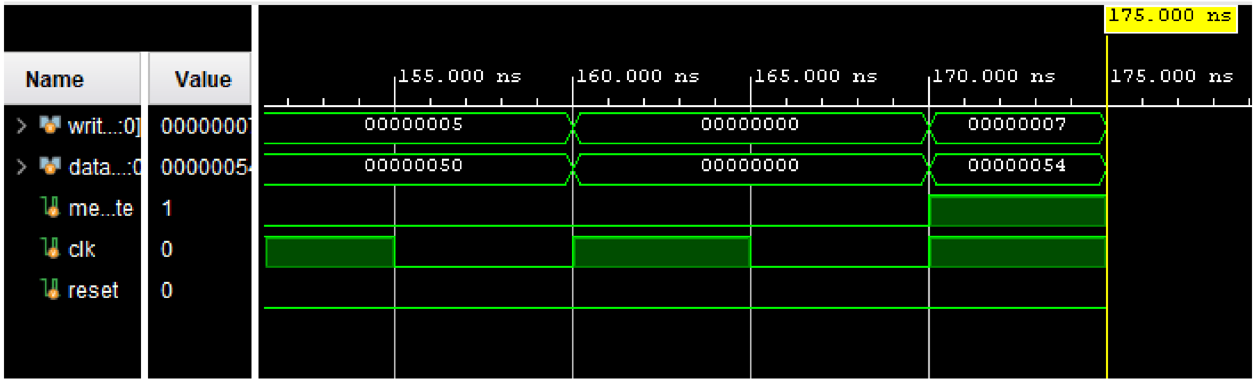
08000011

20020001

ac020054

Line-by-line description of the program is already given by HARRIS.

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a “Simulation suceeded” print out in Vivado’s TCL console).

testbench_2.sv

This is just a slightly modified version of testbench.sv to accommodate the new test program.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named testbench_2. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

11: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite.

14: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

15: Beginning of the initial block.

16: Sets reset to 1 for the first 5 ns of the simulation, then sets it to 0 afterwards (totally arbitrary). Adds a delay of 255 ns that corresponds to the actual runtime of a successful simulation. This explicitly states that reset = 0 for that time frame.

17: End of this initial block.

20: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

21: Beginning of this always block.

22: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

23: End of the always block.

26: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

27: Start of this always block.

28: An if conditional that executes its block if memwrite is asserted. Corresponds to initializing the actual test when sw or sb is the instruction.

Remark: We are now only concerned in getting the Simulation Success Conditions so we leave the Simulation Failed Condition as simply the result of not attaining the simulation success condition.

29-35: CHECKING: If dataadr === 16 & writedata === 0xBBAAB0B0, then simulation is successful (print “Simulation suceeded” then stop the program). Else, simulation failed (print “Simulation failed” then stop the program).

36: End of the memwrite condition block.

37: End of the always block.

38: End of writing the testbench module.

Assembly Program for testbench_2.sv

```
# Write the hex value 0xBBAA0000 to register 1

addi    $1, $0, 0xBBAA  # 2001BBAA

add     $1, $1, $1      # 00210820

add     $1, $1, $1      # ...

add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
# Others
addi    $2, $0, 176       # 200200B0
addi    $3, $2, 45056     # 2043B000
addi    $4, $0, 0x7FFF    # 20047fff
add     $4, $4, $4        # 00842020
addi    $4, $4, 1         # 20840001
and     $3, $3, $4        # 00641824
add     $4, $1, $3        # 00232020
addi    $5, $0, 16        # 20050010
sw      $4, 0($5)         # aca40000
```

Explanation:

1: Insert 0x0000 BBAA to register 1.

2-18: Repeatedly add it to itself 16 times in order to emulate a load upper immediate (lui) instruction. After this register 1 contains 0xBBAA 0000.

20: Insert value 176 to register 2.

21: $3 = $2 + 45056 = 176 + 45056.

22: Insert 0x0000 7FFF to $4.

23. Add $4 to itself and store result to $4.

24: Add a 1 to the content of register 4 and store result to register 4.

25: $3 = $3 AND $4.

26: $4 = $1 + $3

27: Store value 16 to $5.

28: Store word content of $4 (0xBBAAB0B0 after all the previous computations) to effective address given by $5 (16).

The above program correspond to the machine code below:

2001bbaa

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

200200b0

2043b000

20047fff

00842020

20840001

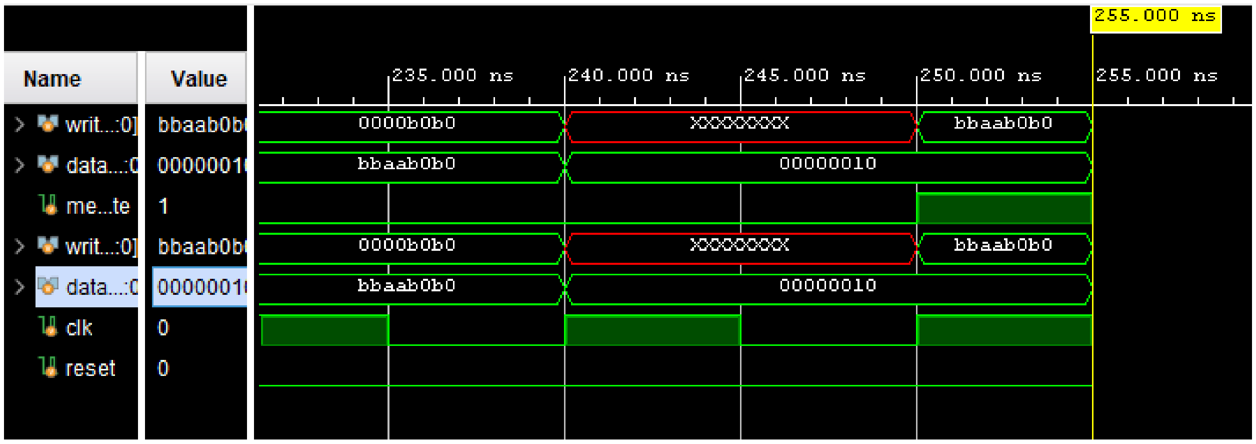
00641824

00232020

20050010

aca40000

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a "Simulation suceeded" print out in Vivado's TCL console).

sll_testbench.sv

The SystemVerilog code for the sll_testbench.sv module is provided below.

```
`timescale 1ns / 1ps

module sll_testbench();


  logic       clk;

  logic       reset;


  logic [31:0] writedata, dataadr;
  logic      memwrite;
  // NOTE, watching shift from mips to top
  logic      shift;

  // instantiate device to be tested
  top dut(clk, reset, writedata, dataadr, memwrite, shift);

  // initialize test
  initial
    begin
      reset <= 1; #1;
      reset <= 0;
      #99 $finish;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  // check results
  always @(negedge clk)
    begin
      if(memwrite) begin
        if(dataadr === 84 & writedata === 'h0C00000C) begin
          $display("Simulation succeeded");
          $stop;
        end else begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
  endmodule
```

This is the main testbench for sll. This is just an appropriation of the previous testbenches.

The explanations for the sll_testbench.sv code above are provided below.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named sll_testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

10: Declared a watcher for the shift signal.

13: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite and also shift.

16: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

17: Beginning of the initial block.

18-20: Sets reset to 1 for the first 1 ns of the simulation, then sets it to 0 afterwards. This initializes the PC register to instruction address 0. After 99 ns end the program, corresponding to actual runtime of program (we could just remove this).

21: End of this initial block.

24: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

25: Beginning of this always block.

26: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

27: End of the always block.

30: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

31: Start of this always block.

32: An if conditional that executes its block if memwrite is asserted. Corresponds to initializing the actual test when sw or sb is the instruction.

Remark: We are now only concerned in getting the Simulation Success Conditions so we leave the Simulation Failed Condition as simply the result of not attaining the simulation success condition.

33-39: CHECKING: If dataadr === 84 & writedata === 0x0C00000C (expected results, see assembly program explanation or annotations), then simulation is successful (print "Simulation suceeded" then stop the program). Else, simulation failed (print "Simulation failed" then stop the program).

40: End of the memwrite condition block.

41: End of the always block.

42: End of writing the testbench module.

Assembly Program for sll_testbench.sv

```
addi $s0, $0, 0x000C

sll  $s0, $s0, 8

sll  $s0, $s0, 8

sll  $s0, $s0, 8

addi $s0, $s0, 0x000C

sw   $s0, 84($0)
```

Explanation:

1: $s0 = 0 + 0x0000 000C = 0x0000 000C

2-3: SLL TESTING: 0x0000 000C should be shifted by 24 bits to the left in total. Result would be 0x0C00 0000.

4: Then add 0x000C to 0x0C00 0000 and store that back to $s0. $s0 should be 0x0C00 000C at this point.

5: Store word contained by $s0 to effective address 84 (may break MARS but for the Vivado simulation it's just fine). $s0 must contain 0x0C00 000C for this simulation to be successful.

The corresponding machine code for this is simply

2010000c

00108200

00108200

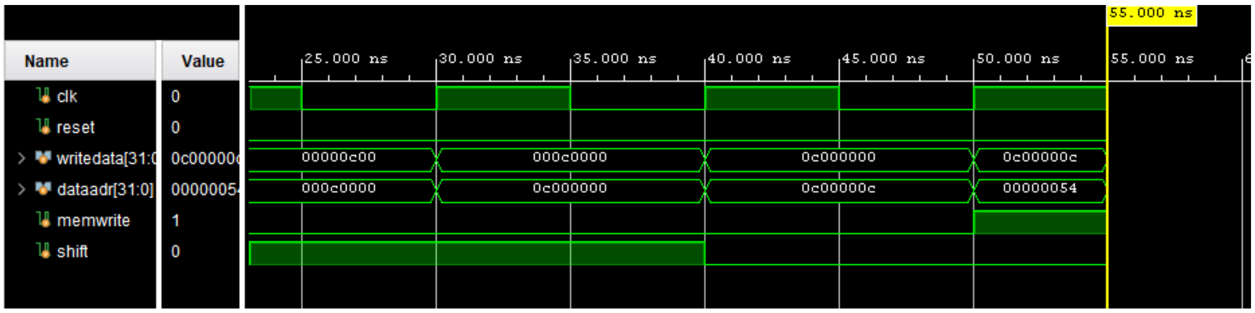
00108200

2210000c

ac100054

Said machine code is as is and did not need any adjustments to work in the context of the memory file and the processor. Moreover, recall that the **rs field** is a don't care for sll; in our case, we simply set the rs field to 0s, but for future instructions we also use variations of bits for the don't cares. Also, by design, we can be assured that the rs field of sll is really treated as a don't care because our shift signal selects between the rs contents and the zero-extended shamt, so it's just either one of the two.

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a "Simulation suceeded" print out in Vivado's TCL console).

sb_testbench.sv (new!)

The SystemVerilog code for the sb_testbench.sv module is provided below.

```
`timescale 1ns / 1ps

module sb_testbench();


  logic     clk;

  logic     reset;
  logic [31:0] writedata, dataadr;
  logic       memwrite;
  // NOTE, watching shift from mips to top
  logic       shift;

  // instantiate device to be tested
  top dut(clk, reset, writedata, dataadr, memwrite, shift);

  // initialize test
  initial
    begin
      reset <= 1; #1;
      reset <= 0;
      //#99 $finish;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5; $display("%d", $realtime);
    end

  // check results
  always @(negedge clk)
    begin
      if($realtime == 125) begin
```

```
        if(dataadr === 84 & writedata === 'hFFFF0000) begin
          $display("Simulation succeeded");
          $stop;
        end else begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule
```

The explanations for the sb_testbench.sv code above are provided below.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named sb_testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

10: Declared a watcher for the shift signal.

13: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite and also shift.

16: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

17: Beginning of the initial block.

18-19: Sets reset to 1 for the first 1 ns of the simulation, then sets it to 0 afterwards. This initializes the PC register to instruction address 0.

21: End of this initial block.

24: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

25: Beginning of this always block.

26: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns. And from now on, we also print the current simulation time for verification purposes.

27: End of the always block.

30: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

31: Start of this always block.

32: An if conditional that executes at realtime == 125 ns, and this corresponds with the expected time that the processor reads the final instruction in the memory file.

Remark: We are now only concerned in getting the Simulation Success Conditions so we leave the Simulation Failed Condition as simply the result of not attaining the simulation success condition.

33-39: CHECKING: If dataadr === 84 & writedata === 0xFFFF0000 (expected results, see assembly program explanation or annotations), then simulation is successful (print "Simulation suceeded" then stop the program). Else, simulation failed (print "Simulation failed" then stop the program).

40: End of the memwrite condition block.

41: End of the always block.

42: End of testbench module.

Assembly Program for sb_testbench.sv

```
addi   $s0, $0, 0x65FF # s0 = 0x0000 65FF

sw     $s0, 8($0)      # [8] = 0x0000 65FF

sb     $s0, 0($0)      # [0] = 0xFF : [0] = 0xFFXX XXXX

sb     $s0, 1($0)      # [1] = 0xFF : [0] = 0xFFFF XXXX

sb     $s0, 2($0)      # [2] = 0xFF : [0] = 0xFFFF FFXX

sb     $s0, 3($0)      # [3] = 0xFF : [0] = 0xFFFF FFFF

sb     $s0, 4($0)      # [4] = 0xFF : [4] = 0xFFXX XXXX

sb     $s0, 5($0)      # [5] = 0xFF : [4] = 0xFFFF XXXX

sb     $0, 6($0)       # [6] = 0x00 : [4] = 0xFFFF 00XX

sb     $0, 7($0)       # [7] = 0x00 : [4] = 0xFFFF 0000
```

```
lw      $s1, 0($0)       # s1 = [0] = 0xFFFF FFFF
lw      $s2, 4($0)       # s2 = [4] = 0xFFFF 0000
sw      $s2, 84($0)      # [84] = 0xFFFF 0000
# expected results: dataadr === 84 = 0x00000054, writedata === 0xFFFF0000
```

Using this program, we verify if sb is indeed just storing the lowest order byte of wd and if it is being stored in the correct byte offset according to Big Endian as desired. This is a 13 instruction program so for our simulation this ends at 125 ns, at which point we check the sw results (TODO, pag nag store word ng 0xBABA 0000 to address 0, anong result 0x0000 BABA or 0xBABA 0000).

The above program is equivalent to the machine code below:

```
201065ff
ac100008
a0100000
a0100001
a0100002
a0100003
a0100004
a0100005
a0000006
a0000007
8c110000
8c120004
ac120054
```

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):

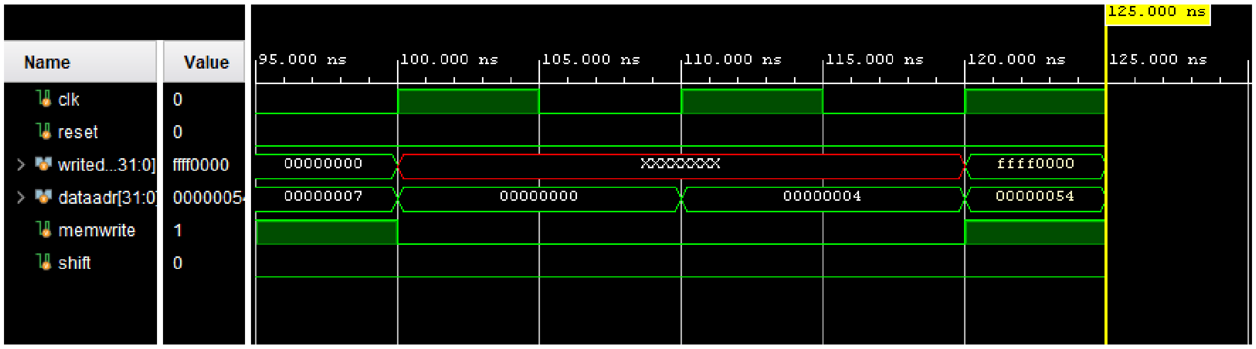


Figure 7. sb_testbench.sv simulation waveform diagram.

As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a "Simulation suceeded" print out in Vivado's TCL console).

Having passed all these test benches, we certify this MIPS processor to be capable of executing the instructions **shift left logical** (**sll**) and **store byte (sb)**.

Pseudo-Instructions

ble rs, rt, label

Instruction Format

The **branch on less than or equal** or **ble** pseudoinstruction is an (arguably) I-type instruction with the following properties:

- Coded in MIPS Assembly as ble rs, rt, label.
- Opcode = 0x1F = 011111
- No funct field, as with all I-types.
- All I-type fields are used.
- Operation: if ([rs] <= [rt]) PC = BTA.
    - Branches if rs is less than or equal rt.

Overview of Modification

- We modify the main decoder to recognize ble's opcode which is 011111. Note that the aludec module is not modified, the subtraction aluop from main decoder is enough. For further information, please see the comments on the modules themselves.
- We modify the ALU so it will be able to produce a sign flag (similar to the zero flag) which tells whether the sign of ALUOut is positive (0: implying A >= B) or negative (1: implying A < B).
- We modify $\text{pcsrc} = \text{beq AND zero}$ to $\text{pcsrc} = (\text{beq AND zero}) \text{ OR } \big(\text{ble AND } (\text{sign OR zero})\big)$.
- To that end, we have to dedicate a new ble signal separate from the beq signal.

This is because if we just rely on a single signal, say branch, to tell whether the instruction is ble or beq, then for instance, we can have something like $\text{pcsrc} = \text{branch AND } (\text{sign OR zero})$ which can produce wrong decisions. For instance if the instruction is beq, branch is asserted, and then if A < B, then it will mistakenly branch even if A is not equal to B. For this reason we make use of separate signals in order to be more explicit.

- The datapath, controller, and their interface the mips module are all modified to handle the new wirings given by beq (renamed branch), ble, and sign.

Control Signals

The following are the final control signals for this instruction. In yellow are new signals.

Remark: For consistency with the HARRIS implementation, we always take the Don't Care signals to be 0 in the code. In this case, regdst and memtoreg are in fact Xs but taken as 0s. sb can also be X.

From Main Decoder:

regwrite	regdst	alusrc	beq	memwrite	memtoreg	jump	aluop	sb	ble	li
0	0	0	0	0	0	0	01	0	1	0

From ALU Decoder:

alucontrol	shift	zfr
110 (sub)	0	0

Modifications in the HDL Code

We now explain the HDL modules we modified in order to implement the ble instruction.

maindec.sv

The maindec module, instantiated as md, is the module that handles the bulk of the instruction decoding.

For the ble instruction, this is modified to recognize ble's opcode of 011111 and generate the appropriate control signals. To be concise, the signals are aluop = 01 (for subtraction) and ble = 1, and the rest are 0s. Moreover, an output port ble is declared for this signal.

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively.

2-11: Creates a main decoder module named maindec with the following ports:

- 6-bit logic-type input port (node) named op with [MSB:LSB] = [5:0]. Used for accepting the instruction's opcode.
- 1-bit logic-type output ports named memtoreg, memwrite, beq, alusrc, regdst, regwrite, and jump. These are control signals that will then go on to dictate the flow of the datapath,
- 2-bit logic-type output port aluop with [MSB:LSB] = [1:0]. Is fed to the ALU decoder module that helps simplify the decoding of the ALU operation.
- The output ports sb (for sb onwards), ble (for ble onwards), and li (for li onwards) to handle the signals corresponding to them.

This module will be a subset of the control unit.

14: Declares a 12-bit logic-type variable named controls with [MSB:LSB] = [11:0]. This will be used to handle all the control signals generated by the main decoder as a concatenated, convenient 12-bit wire value.

- Originally, this was just a 9-bit value without the sb, ble, and li signals.
- After adding sb, this became a 10-bit value.
- After adding ble, this became an 11-bit value.
- After adding li, this became a 12-bit value.

16-17: Wirings of the proper control signals using a concatenation operator for ease of assignment. This kind of formatting is useful for simultaneously assigning multiple values that goes to different components, such as what happens with the control signals.

22: Start of an always_comb block for the combinational assignment of control signals.

23-34: A case block controlled by the opcode op. Control signals are predefined for each recognized instructions. As can be seen, R-type corresponds to opcode == 000000, and the rest are R-types.

Additions here are:

- 6'b101000: controls <= 12'b001010000_100; // SB (sb = 1) for SB onwards.
- 6'b011111: controls <= 12'b000000001_010; // BLE (beq=0, ble=1) for BLE onwards.
- 6'b010001: controls <= 12'b100000000_001; // LI (regwrite=1, li=1) for LI onwards.

Our main focus here is the BLE addition. By making 6'b011111 a case controlled by op, then we make BLE recognizable to the processor, and then we set the control signals for it, where beq = 0, aluop = 01, ble = 1, and the rest are 0s (check Don't Care caveat). Aluop is 01 because like with beq, we use subtraction to ascertain the relation of operand [rs] or A with operand [rt] or B. Simply put (let A = [rs] and B = [rt]):

- If A-B == 0, then A == B. (ALU asserts zero flag)
- If A-B < 0, then A < B. (ALU asserts sign flag)
- If A-B > 0, then A > B.

35: Ends the writing of the maindec module.

controller.sv

This module pertains to the control unit and also houses the pcsrc logic. This contains the main decoder and the ALU decoder.

The current main modifications here are the wiring of the output ble signal from maindec to pcsrc, and of course the modification itself to pcsrc to decide branching in the context of ble and beq.

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively.

2-13: Creates a module named controller and declares the input and output ports (also considered as nodes) of the module. Inputs are 6-bit op and funct (fields), and 1-bit zero flag from the ALU in the datapath. Outputs are 1-bit control signals memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, and jump; and 3-bits for the alucontrol.

Port modifications are as follows:

- Output shift signal from sll onwards. From alu decoder, goes to srcamux.
- Output sb signal from sb onwards. From main decoder, goes to dmem module.
- Input sign flag from **ble** onwards. From datapath's ALU, goes to this controller.
- Output li signal from li onwards. From main decoder, goes
- Output zfr signal from zfr onwards. From alu decoder, goes to zfrmux.

16: Declares 2-bit logic-type aluop. Wires aluop from maindec to aludec.

17: Originally declared as logic-type branch but is now beq from beq implementation onwards. Will serve as the control signal for beq.

18: Declares ble from ble implementation onwards. Will serve as control signal for ble.

Note that beq and ble are not declared as ports because they are only internally used by the controller, together with the zero flag and the sign flag (from ble onwards) to determine the pcsrc control signal (i.e. whether to branch or not).

20-22: Instantiates the main decoder as md. Has as input op (the opcode), and as outputs memtoreg, memwrite, beq (originally branch), alusrc, regdst, regwrite, jump, aluop.

Additions are the outputs sb (from sb onwards), ble (from ble onwards), and li signals (from li onwards).

Focus on the ble signal which works with beq, zero, and sign to produce the correct pcsrc.

24: Instantiates the alu decoder as ad. Inputs are funct and aluop. Outputs are alucontrol, shift, and zfr, as desired.

Note that shift and zfr are generated in this module because they are R-type instructions, and they can only be distinguished from each other by examining the funct field since they all have the same 000000 opcode (while also conveniently maintaining the proper control signals for R-type instructions as given by the main decoder).

Generation of sb signal is in maindec, not in the aludec.

27: Wires pcsrc to the result of branch AND zero, i.e. we branch when the instruction is in fact branch and the contents of the two registers are equal (given by their difference being zero).

Note that for the addition of ble this is expanded to become assign pcsrc = (beq & zero) | (ble & (sign | zero)). In other words, there are now two possibilities for branching: when beq or when ble. This Boolean expression is generated by the simple expression that the processor should:

Branch if the instruction is beq and the difference of [rs] and [rt] is zero (meaning [rs] == [rt]), or branch if the instruction is ble and the difference of [rs] and [rt] is negative (meaning [rs] < [rt]) or is zero (meaning [rs] == [rt]). Essentially read as branch if equal or branch if less than or equal. Note that

$$\text{pcsrc} = \text{beq} \,\&\, \text{zero} \mid (\text{beq} \,\&\, \text{sign} \mid \text{beq} \,\&\, \text{zero}) = \text{beq} \,\&\, \text{zero} \mid (\text{beq} \,\&(\text{sign} \mid \text{zero}))$$

The signal PCSrc controls which next instruction address will be fed to the instruction memory, whether it be for branch (BTA) or in our situation, jump (JTA) is also possible.

28: End of writing the controller module.

mips.sv

Now we go to the mips module which contains the control unit and the datapath. This is the SCP minus the instruction memory (imem) and the data memory (dmem). In essence, the mips module serves as relay between the controller and the datapath and this allows the two components to communicate.

We modify this module for ble to wire the sign flag from the datapth to the controller where it will be used with the beq, ble, and zero to generate pcsrc. We add a caveat about jump with respect to the branches later on in the datapath.

1: Sets the module’s delay unit and delay resolution to 1ns and 1ps respectively.

2-9: Creates a module named mips with inputs clk (the clock driver), reset (for resetting the PC register), instr (the whole 32-bit instruction), and 32-bit readdata (pertains to result of combinational read of dmem, only actually used for lw). Outputs are pc (from the data path, gives the next PC address, default is PC+4, special cases for branches and jumps), memwrite (from controller, controls whether to write or not to dmem, aluout (because we want to watch aluout for testing), writedata (we also want to watch this).

Additions are the shift output signal (I placed this here because I personally want to watch this).

sb is added as output port from sb onwards. We still want it to go out of this module because it should head to data memory.

12-13: Declares variables memtoreg, alusrc, regdst, regwrite, jump, pcsrc, and zero. Used for wiring between the controller and the datapath, usually controls multiplexers in the datapath, except for the zero flag which is wired from the ALU to the controller to control PCSrc.

14: Declares 2-bit alucontrol, also wired from controller's aludec to the controller.

16-18: Declares the wirings for

- the sign flag (from ble onwards), goes from datapath to controller.
- the li signal (from li onwards), goes from controller to datapath.
- and the zfr signal (from zfr onwards). goes from controller to datapath.

20-27: Instantiates a controller module and names it c. It has as inputs 5-bit op and funct, and the zero flag. Output ports are memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump, and 2-bit alucontrol. Aka the control unit.

Port additions are:

- output shift signal, for controlling srcamux (sll onwards).
- output sb signal, for controlling dmem store (sb onwards).
- input sign flag, for deciding pcsrc (ble onwards).
- output li signal, for controlling limux (li onwardsd).
- output zfr signal, for controlling zfrmux (zfr onwards).

Focus on ble and how it goes from datapath ALU to controller maindec.

30-37: Instantiates a datapath module and names it dp. Has as inputs clk, reset, memtoreg pcsrc, alusrc, regdst, regwrite, jump, 2-bit alucontrol, the 32-bit instruction, 32-bit read data chiming in from dmem, and the shift signal for the shift multiplexer.

Port additions are (some may be reiterated):

- input shift signal, for controlling srcamux (sll onwards).
- output sign flag, for deciding pcsrc (ble onwards).
- input li signal, for controlling limux (li onwardsd).
- input zfr signal, for controlling zfrmux (zfr onwards).

38: End of the mips module.

datapath.sv

The datapath is where all the main computations that generates register results take place. Here in the datapath the modification for ble can be found inside the ALU and that can be seen with the sign signal now going out of the ALU.

Remark: We explain in further detail the modifications for li (includes a new zero extender and a new multiplexer) and zfr (includes a new constant, left shifter, bit selector, and multiplexer) in their respective documentations.

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively.

2-15: Creates a module named datapath.

- Has as inputs clk for the clock driver, reset for the reset initializer or just plain reset; control signals memtoreg, pcsrc, alusrc, regdst, regwrite, jump, and 2-bit alucontrol; 32-bit instr which is read from imem via interpreting PC; 32-bit readdata that comes in from dmem for load word; and new signals shift (for sll onwards), li (for li onwards), and zfr (for zfr onwards).
- Has as outputs zero flag, 32-bit pc for the next instruction address (goes to imem), 32-bit aluout tied to dataadr on mips module instantiation of datapath and beyond for testbench watching, 32-bit writedata for the data to be written to the dmem via store instructions, and a new sign flag for branch instructions.

New muxes srcamux, limux, and zfrmux are added/to be added on their respective instructions.

18-25: Declare 5-bit writereg; and 32-bit pcnext, pcnextbr, pcplus4, pcbranch, signimm, signimmsh, RD1, RD2, srca, srcb, liout (new!), result, and zfrout (new!). After some erroneous simulations, I learned the value of first declaring nodes and most importantly their sizes before wiring together the modules using said nodes.

These declarations help guide the wires to connect to the write places, those preceeded with pc deals with the instruction addressing, and the other deal with the main data path.

Changes I did here is to add RD1 and RD2 to correspond to the rs and rt value outputs of the regfile respectively. This is done for clarity.

28: always_comb block start. As explained previously, this watches all variables that are manipulated inside, and executes on changes. I placed this here to watch the changes in dataaddr and writedata more internally (because for the testbench we simply just use comparisons of dataaddr and writedata actual values to their expected values to verify the correctness of the simulation).

31: Displays the dataaddr and writedata contents whenever there are changes with them. Note that dataaddr is wired to alutout and writedata is wired to RD2.

32: End of the always_comb block above.

35: Instantiates flopr as pcreg, parameter WIDTH=32 with inputs clk, reset, 32-bit pcnext (seen as PC' in the HARRIS schematic), and output 32-bit pc. This serves as the pc register which updates every clock rising edge. Flopr is used because of the need for a resettable flipflop (reset required usually for initializing the instruction address). Please refer to my Lab Report 9 for line-by-line explanations of the contents of flopr.

Note that pc goes out to imem to specify the next instruction address to be read.

36: Instantiates adder as pcadd1, parameter WIDTH = 32, with inputs pc, 32’b100 (constant 4), and ‘b0 (0 carry in), and outputs pcplus4 (namesake). Constantly generates PC+4 for *consideration*. Please refer to my Lab Report 9 for line-by-line explanations of the contents of adder.

37: Instantiates sl2 as immsh with input signimm (sign-extended instr[15:0]) and output signimmsh. This shifts the input to the left by 2 (multiplies by 4) as the first step to generating BTA. Please refer to my Lab Report 9 for line-by-line explanations of the contents of sl2.

38: Instantiates adder as pcadd2, WIDTH = 32, with addends pcplus4, signimmsh, and 0 for carry in. Output is pcbranch, for consideration. When we say consideration, we imply that the value is being muxed with something else. This completes BTA = PC+4 + (SignImm << 2). Please refer to my Lab Report 9 for line-by-line explanations of the contents of mux2.

39: Instantiates mux2 as pcbrmux, WIDTH = 32, to choose between 0: pcplus4 and 1: pcbranch. Controlled by pcsrc. Result becomes pcnextbr.

40: Instantiates mux2 as pcmux, WIDTH = 32, to choose between pcnextbr and JTA = {(PC + 4)[31:28], addr, 2’b0}. Controlled by jump.

Caveat on this as promised earlier is that we have to realize that there are two multiplexers multiplexing which instruction address to feed to the PC register. The first is the pcbrmux, controlled by pcsrc that is of clear importance to ble. And the next is pcmux, in that order, which with a slight modification now decides whether to branch, or jump, or neither. This order is the reason why if we want to branch, we also need jump to be 0, but if we just want to jump, branch (ble or beq) is a Don’t Care, along with the majority of other control signals.

Remark on Multiplexers: When we say “choose between D1 and D2”, we imply that select signal = 0 passes D1 and select signal = 1 passes D2. This is appropriate because we **only use 2-way multiplexers**. Please keep this remark and all other remarks in mind.

All these complete the choice for the next PC value, whether it be PC+4, BTA, or JTA. Note that we did not made any modifications for this portion, so these explanations will just repeat over and over again, as is.

44: Clarity modification, we wire writedata to RD2. Note that RD2 comes from the register file, and it is wired to WriteData WD of data memory with nothing in between (constantly zero multiplexers in the connection). This is a watched variable.

45-46: Instantiates regfile as rf. Inputs are the clk (for possibly writing sequentially), regwrite signal, instr[25:21] = rs, instr[20:16] = rt, writereg or value to be written, and result coming in from resmux. Outputs are RD1 and RD2. The register file will be written too on clock rising edge if WE3 is asserted by RegWRite.

This regfile is very integral to the processor and as we all know this is like the heart that beats with every instruction we pump in it in line with the synapses of the control unit which is kind of like the pace maker of this system.

47-48: Instantiates mux2 as wrmux with WIDTH = 5 (begause register address is always just 5 bits). This chooses between rt and rd over which to use as the write address for the register file. This is controlled by regdst (register destination). Outputs the address to writereg and is fed back to the register file.

49: For li onwards, a new multiplexer named limux is introduced. It is located somewhere between dmem and resmux in our schematic diagram (for li onwards!). This chooses between aluout and ZeroImm, and it decides using the li signal.

50: Instantiates mux2 with width 32 as resmux. Chooses between the output of ALU, aluout (rather liout from li onwards!), and the data that was combinationally read from dmem, readdata. Controlled by memtoreg. Outputs to result, and that result is fed back to the register file.

51: Instantiates signext as se. Takes in the perceived immediate field instr[15:0] of any instruction and sign-extends the contents to 32-bit, and the result is passed to signimm. Please refer to my Lab Report 9 for line-by-line explanations of the contents of signext.

56: This is an sll modification. Instantiates mux2 as srcamux, 32-bit. Chooses between RD1 and the zero-extended contents of the shamt field of the instruction (instr[10:6]). This is controlled by the shift signal. The result of this is passed to srca which is wired to the ALU.

57: Instantiates mux2 as srcbmux, 32-bit. Chooses between RD2 and signimm (sign-extended immediate). Controlled by alusrc. Output passed to srcb. Originally this connects directly to srcb port of the ALU, but this will change for zfr which introduces a new mux, zfrmux, between that connection. Note that previously RD2 was just written as writedata, but that is not so descriptive so we wired writedata and RD1 to each other.

The motivation for lines 56-57 is that we turned that unused alucontrol assignment to a shift operation within the ALU, to be precise, it is result = b << a. Sa if the instruction is shift, srca of ALU receives the desired shift amount stored in the instruction’s shamt field, and srcb receives the value to shift from rt.

58: Ignore zfr for now. Hence, we imagine srcb to be directly connected to alu until before zfr is implemented.

59: Instantiates alu as alu. Operand A is srca, and Operand B is aluout (zfrout later on). Controlled by alucontrol. Outputs result to aluout, and also generates zero flag (and sign flag for ble later on).

Per HARRIS, an Arithmetic/Logical Unit (ALU) combines a variety of mathematical and logical operations into a single unit. For example, a typical ALU might perform addition, subtraction, magnitude comparison, AND, and OR operations. In our case, we use a 3-bit control signal alucontrol which specifies 7 default functions or operations (8 if we include the recently added sll instruction for alucontrol 011). The ALU forms the heart of most computer systems, where the alucontrol is like the passive brain signal that directs which artery or multiplexer data source (I know, kind of stretching the analogy here) will the *actual* processed result come from, because most ALU implementations actually computes the respective operation results simultaneously, and the alucontrol just selects which of those results will actually be presented.

The **sign flag** is also now added as an output of the ALU. This signal helps control our branches.

61: Ends the datapath module.

alu.sv

This ALU is modified to handle taking the sign bit **especially** of the **difference** A-B for **ble** (had to emphasize that because the ALU doesn't always store A-B in the result). To this end, we simply take the MSB of the result, like what the case for slt does. The reason we cannot reuse slt is because we want a separate, dedicated sign flag for this so it is more visible. We also could've just taken the sign bit from aluout in the datapath but we won't do that for the extra visibility given by a dedicated sign flag.

1-5: Creates a module named alu. Inputs 32-bit a and b, and 3-bit alucontrol. Outputs 32-bit result, and zero flag (also sign flag from ble onwards).

8: Declares 32-bit condinvb and 32-bit sum.

Note that if alucontrol[2] = 0, we assume addition, and elif alucontrol[2] = 1, we assume subtraction or slt.

14: Wires condinvb with the result of (alucontrol[2] ? ~b : b). The name condinvb stems from its ternary conditional computational expression that chooses between regular operand B or a negated operand B (hence this can be represented by a mux controlled by alucontrol[2] in the circuit diagram).

Some other remarks are that:

- If alucontrol[2] = 0, condinvb + alucontrol[2] = b.
- Elif alucontrol[2] = 1, condinvb + alucontrol[2] = -b (in 2C, recall –b = ~b + 1).
- 2C Addition of a + (-b) is then essentially a subtraction.
- So sum will instead contain a difference (a-b).
- We also note that despite the availability of the A AND ~B, and A OR ~B functions in the HARRIS ALU operation set, it appears that the original alu.sv isn't coded to perform such operations since condinvb is only confined to the generation of either a true sum or a difference. However, we leave this be because we are not interested in isntructions that do this.

19: Wires sum to result of a + condinvb + alucontrol[2]. In other words, addition or subtraction is always performed regardless of the instruction, the question is now just whether to return that result or not. Sum here can either contain a true sum or a difference depending on alucontrol[2].

23: Start of an always_comb block. Properties explained earlier.

24: This is an **sll modification**. If alucontrol is 011 (corresponding to sll), then ALU returns b << a. Note that a is the shift amount and b is from rt.

25: Else if alu is not 011, then we consider other cases.

26-35: Having already considered alucontrol[2] for sum or difference in lines 14-19, we now just look at alucontrol[1:0]. If it is 00, then apply AND (also done for ZFR later on). If it is 01, then apply OR. If it is 10, then return sum (possibly a true sum or a difference depending on alucontrol[2]). If it is 11, then this is actually for slt, implying alucontrol[2] = 1 (usually do subtraction for comparisons), so we return the sign bit (note that if the difference of a – b is negative, then that means a < b, so sign bit of sum in this case would be 1, which would be the output we desire for slt a b).

These are preset behaviors and we need not explain them further.

38: Assign zero flag with whether result equals zero. For use with branches.

39: For **ble** onwards, we also set a sign flag that tells us whether A < B. Again, just taking the sign bit of (A – B) is simpler than actually doing A < B comparison in terms of low-level circuits.

40: End of the alu module.

Schematic Diagram

We now present the schematic for a portion of the datapath and the controller

Note that we consider the newly added gates for the PCSrc to be INSIDE the controller module (as it is in the code) despite being outside of the control unit (an explication adopted from the original HARRIS schematic).

Once again, the modified components are in blue and the new components are in red. Modified here is the ALU for the new sign flag, the controller's main decoder for recognizing ble and generating proper signals for it, and for the wirings of the new components. Added here are the logic gates that augments the pcsrc control signal to recognize both beq and ble with the help of the sign and zero flags.

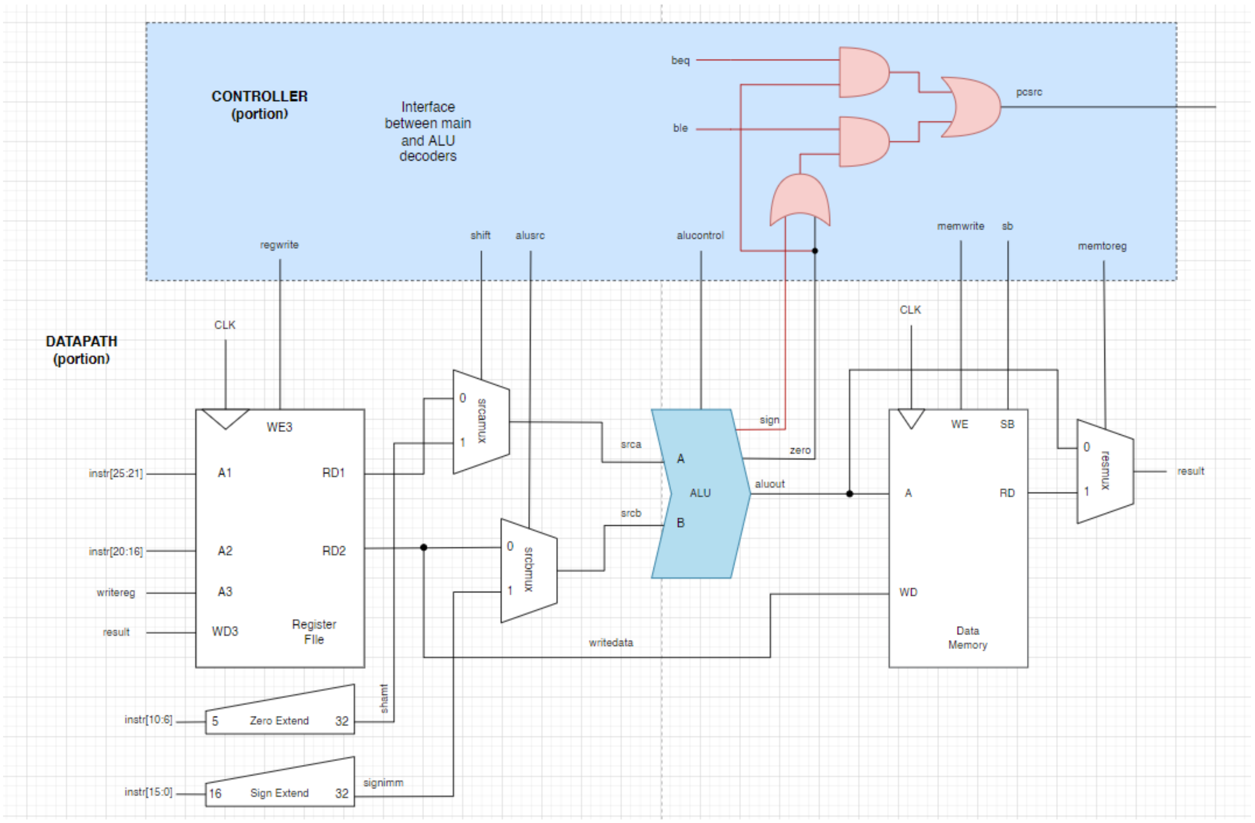


Figure 8. ble schematic diagram (partial view of the datapath and controller only).

Also I've been tidying up the diagram, so it might look fresher compared to the previous ones.

Testbenches and Test Programs

For the ble instruction verification, we first check if the modified MIPS processor can still perform the original and the previously added instructions by running it through a gauntlet of the previous testbenches.

Moreover, the new testbench, ble_testbench.sv, tests all the recently added instructions for stronger verification.

The testbenches and their corresponding test programs now follow.

testbench.sv

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

11: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite.

14: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

15: Beginning of the initial block.

16: Sets reset to 1 for the first 22 ns of the simulation, then sets it to 0 afterwards.

17: End of this initial block.

20: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

21: Beginning of this always block.

22: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

23: End of the always block.

26: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

27: Start of this always block.

28: An if conditional that executes its block if memwrite is asserted. Corresponds to initializing the actual test when sw or sb is the instruction.

29-35: CHECKING: If dataadr === 84 & writedata === 7, then simulation is successful (print "Simulation suceeded" then stop the program). Else if dataadr !=== 80, simulation failed (print "Simulation failed" then stop the program). The reason for that failure condition is that in the test program, there's a non-terminating store word with effective address 80, so if it happens that dataaddr === 80 when memwrite == 1, then it means that there's a mix-up in the computations.

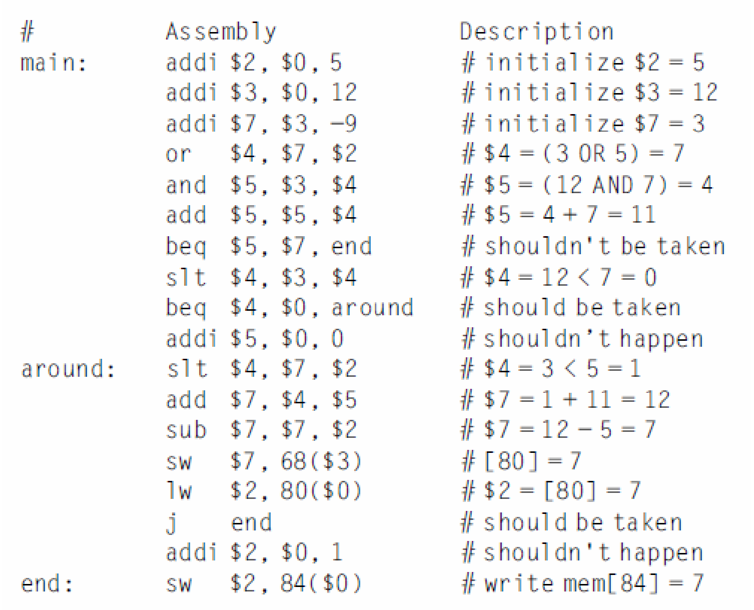
36: End of the memwrite condition block.

37: End of the always block.

38: End of the testbench module.

Assembly Program for testbench.sv

Below is the MIPS program that this testbench will run through. Courtesy of HARRIS.



```
#         Assembly              Description
main:     addi $2, $0, 5        # initialize $2 = 5
          addi $3, $0, 12       # initialize $3 = 12
          addi $7, $3, −9       # initialize $7 = 3
          or   $4, $7, $2       # $4 = (3 OR 5) = 7
          and  $5, $3, $4       # $5 = (12 AND 7) = 4
          add  $5, $5, $4       # $5 = 4 + 7 = 11
          beq  $5, $7, end      # shouldn't be taken
          slt  $4, $3, $4       # $4 = 12 < 7 = 0
          beq  $4, $0, around   # should be taken
          addi $5, $0, 0        # shouldn't happen
around:   slt  $4, $7, $2       # $4 = 3 < 5 = 1
          add  $7, $4, $5       # $7 = 1 + 11 = 12
          sub  $7, $7, $2       # $7 = 12 − 5 = 7
          sw   $7, 68($3)       # [80] = 7
          lw   $2, 80($0)       # $2 = [80] = 7
          j    end              # should be taken
          addi $2, $0, 1        # shouldn't happen
end:      sw   $2, 84($0)       # write mem[84] = 7
```

The machine code below is equivalent to the program above. We fill the memory file memfile.mem with this machine code if we want run this testbench.

20020005

2003000c

2067fff7

00e22025

00642824

00a42820

10a7000a

0064202a

10800001

20050000

00e2202a

00853820

00e23822

ac670044

8c020050

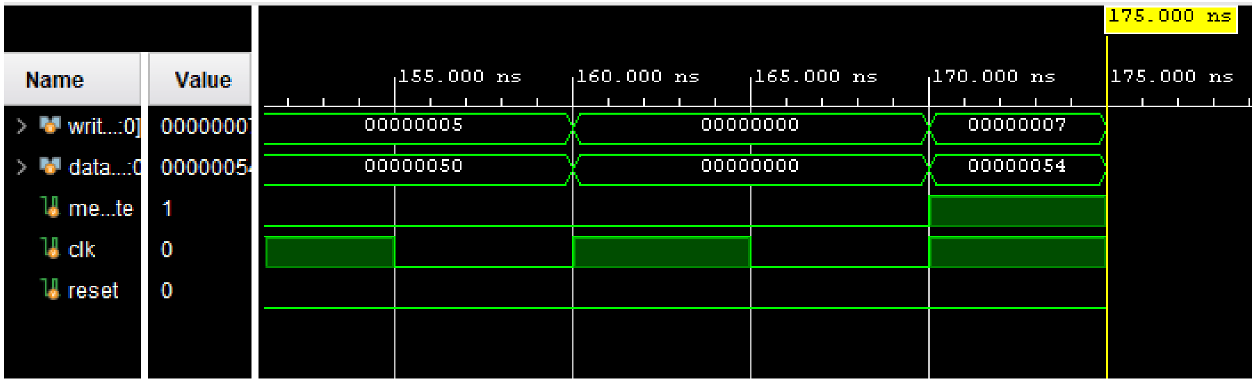
08000011

20020001

ac020054

Line-by-line description of the program is already given by HARRIS.

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a "Simulation suceeded" print out in Vivado's TCL console).

testbench_2.sv

This is just a slightly modified version of testbench.sv to accommodate the new test program.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named testbench_2. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

11: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite.

14: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

15: Beginning of the initial block.

16: Sets reset to 1 for the first 5 ns of the simulation, then sets it to 0 afterwards (totally arbitrary). Adds a delay of 255 ns that corresponds to the actual runtime of a successful simulation. This explicitly states that reset = 0 for that time frame.

17: End of this initial block.

20: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

21: Beginning of this always block.

22: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

23: End of the always block.

26: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

27: Start of this always block.

28: An if conditional that executes its block if memwrite is asserted. Corresponds to initializing the actual test when sw or sb is the instruction.

Remark: We are now only concerned in getting the Simulation Success Conditions so we leave the Simulation Failed Condition as simply the result of not attaining the simulation success condition.

29-35: CHECKING: If dataadr === 16 & writedata === 0xBBAAB0B0, then simulation is successful (print “Simulation suceeded” then stop the program). Else, simulation failed (print “Simulation failed” then stop the program).

36: End of the memwrite condition block.

37: End of the always block.

38: End of writing the testbench module.

Assembly Program for testbench_2.sv

```
# Write the hex value 0xBBAA0000 to register 1
addi    $1, $0, 0xBBAA  # 2001BBAA
add     $1, $1, $1      # 00210820
add     $1, $1, $1      # ...
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
# Others
addi    $2, $0, 176     # 200200B0
addi    $3, $2, 45056   # 2043B000
addi    $4, $0, 0x7FFF  # 20047fff
```

```
add     $4, $4, $4      # 00842020
addi    $4, $4, 1       # 20840001
and     $3, $3, $4      # 00641824
add     $4, $1, $3      # 00232020
addi    $5, $0, 16      # 20050010
sw      $4, 0($5)       # aca40000
```

Explanation:

1: Insert 0x0000 BBAA to register 1.

2-18: Repeatedly add it to itself 16 times in order to emulate a load upper immediate (lui) instruction. After this register 1 contains 0xBBAA 0000.

20: Insert value 176 to register 2.

21: $3 = $2 + 45056 = 176 + 45056.

22: Insert 0x0000 7FFF to $4.

23. Add $4 to itself and store result to $4.

24: Add a 1 to the content of register 4 and store result to register 4.

25: $3 = $3 AND $4.

26: $4 = $1 + $3

27: Store value 16 to $5.

28: Store word content of $4 (0xBBAAB0B0 after all the previous computations) to effective address given by $5 (16).

The above program correspond to the machine code below:

```
2001bbaa
00210820
00210820
00210820
00210820
00210820
00210820
```

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

200200b0

2043b000

20047fff

00842020

20840001

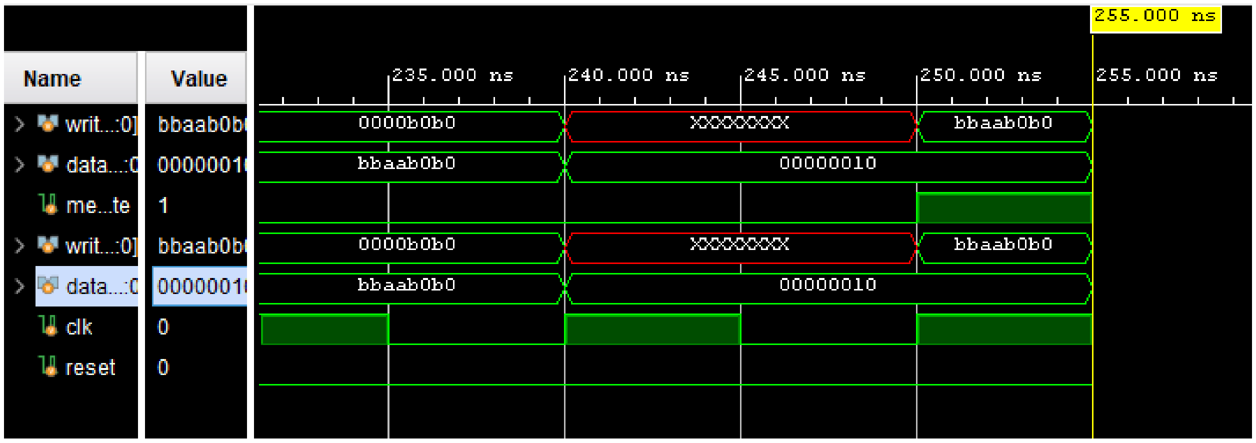
00641824

00232020

20050010

aca40000

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a “Simulation suceeded” print out in Vivado’s TCL console).

sll_testbench.sv

The SystemVerilog code for the sll_testbench.sv module is provided below.

```
`timescale 1ns / 1ps

module sll_testbench();


 logic       clk;

 logic       reset;


 logic [31:0] writedata, dataadr;

 logic       memwrite;

 // NOTE, watching shift from mips to top

 logic       shift;


 // instantiate device to be tested

 top dut(clk, reset, writedata, dataadr, memwrite, shift);


 // initialize test

 initial

   begin

     reset <= 1; #1;

     reset <= 0;

     #99 $finish;

   end


 // generate clock to sequence tests

 always
```

```
  begin
    clk <= 1; # 5; clk <= 0; # 5;
  end


 // check results
 always @(negedge clk)
  begin
    if(memwrite) begin
      if(dataadr === 84 & writedata === 'h0C00000C) begin
        $display("Simulation succeeded");
        $stop;
      end else begin
        $display("Simulation failed");
        $stop;
      end
    end
  end
endmodule
```

This is the main testbench for sll. This is just an appropriation of the previous testbenches.

The explanations for the sll_testbench.sv code above are provided below.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named sll_testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

10: Declared a watcher for the shift signal.

13: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite and also shift.

16: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

17: Beginning of the initial block.

18-20: Sets reset to 1 for the first 1 ns of the simulation, then sets it to 0 afterwards. This initializes the PC register to instruction address 0. After 99 ns end the program, corresponding to actual runtime of program (we could just remove this).

21: End of this initial block.

24: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

25: Beginning of this always block.

26: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

27: End of the always block.

30: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

31: Start of this always block.

32: An if conditional that executes its block if memwrite is asserted. Corresponds to initializing the actual test when sw or sb is the instruction.

Remark: We are now only concerned in getting the Simulation Success Conditions so we leave the Simulation Failed Condition as simply the result of not attaining the simulation success condition.

33-39: CHECKING: If dataadr === 84 & writedata === 0x0C00000C (expected results, see assembly program explanation or annotations), then simulation is successful (print "Simulation suceeded" then stop the program). Else, simulation failed (print "Simulation failed" then stop the program).

40: End of the memwrite condition block.

41: End of the always block.

42: End of writing the testbench module.

Assembly Program for sll_testbench.sv

addi $s0, $0, 0x000C

```
sll  $s0, $s0, 8

sll  $s0, $s0, 8

sll  $s0, $s0, 8

addi $s0, $s0, 0x000C

sw   $s0, 84($0)
```

Explanation:

1: $s0 = 0 + 0x0000 000C = 0x0000 000C

2-3: SLL TESTING: 0x0000 000C should be shifted by 24 bits to the left in total. Result would be 0x0C00 0000.

4: Then add 0x000C to 0x0C00 0000 and store that back to $s0. $s0 should be 0x0C00 000C at this point.

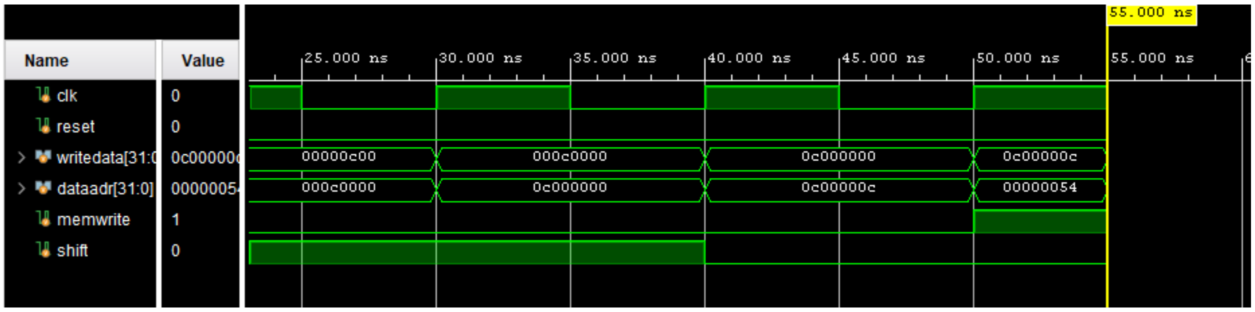
5: Store word contained by $s0 to effective address 84 (may break MARS but for the Vivado simulation it's just fine). $s0 must contain 0x0C00 000C for this simulation to be successful.

The corresponding machine code for this is simply

```
2010000c

00108200

00108200

00108200

2210000c

ac100054
```

Said machine code is as is and did not need any adjustments to work in the context of the memory file and the processor. Moreover, recall that the **rs field** is a don't care for sll; in our case, we simply set the rs field to 0s, but for future instructions we also use variations of bits for the don't cares. Also, by design, we can be assured that the rs field of sll is really treated as a don't care because our shift signal selects between the rs contents and the zero-extended shamt, so it's just either one of the two.

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



Name	Value				
clk	0				
reset	0				
writedata[31:0]	0c00000c	00000c00	000c0000	0c000000	0c00000c
dataadr[31:0]	00000054	000c0000	0c000000	0c00000c	00000054
memwrite	1				
shift	0				

As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a “Simulation suceeded” print out in Vivado’s TCL console).

sb_testbench.sv

The SystemVerilog code for the sb_testbench.sv module is provided below.

```
`timescale 1ns / 1ps

module sb_testbench();


 logic       clk;

 logic       reset;


 logic [31:0] writedata, dataadr;

 logic       memwrite;

 // NOTE, watching shift from mips to top

 logic       shift;


 // instantiate device to be tested

 top dut(clk, reset, writedata, dataadr, memwrite, shift);


 // initialize test

 initial

  begin

    reset <= 1; #1;
    reset <= 0;
    //#99 $finish;
  end

 // generate clock to sequence tests
 always
  begin
    clk <= 1; # 5; clk <= 0; # 5; $display("%d", $realtime);
  end

 // check results
 always @(negedge clk)
  begin
    if($realtime == 125) begin
      if(dataadr === 84 & writedata === 'hFFFF0000) begin
        $display("Simulation succeeded");
        $stop;
      end else begin
        $display("Simulation failed");
        $stop;
      end
    end
  end
endmodule
```

The explanations for the sb_testbench.sv code above are provided below.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named sb_testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

10: Declared a watcher for the shift signal.

13: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite and also shift.

16: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

17: Beginning of the initial block.

18-19: Sets reset to 1 for the first 1 ns of the simulation, then sets it to 0 afterwards. This initializes the PC register to instruction address 0.

21: End of this initial block.

24: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

25: Beginning of this always block.

26: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns. And from now on, we also print the current simulation time for verification purposes.

27: End of the always block.

30: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

31: Start of this always block.

32: An if conditional that executes at realtime == 125 ns, and this corresponds with the expected time that the processor reads the final instruction in the memory file.

Remark: We are now only concerned in getting the Simulation Success Conditions so we leave the Simulation Failed Condition as simply the result of not attaining the simulation success condition.

33-39: CHECKING: If dataadr === 84 & writedata === 0xFFFF0000 (expected results, see assembly program explanation or annotations), then simulation is successful (print "Simulation suceeded" then stop the program). Else, simulation failed (print "Simulation failed" then stop the program).

40: End of the memwrite condition block.

41: End of the always block.

42: End of testbench module.

Assembly Program for sb_testbench.sv

```
addi    $s0, $0, 0x65FF # s0 = 0x0000 65FF

sw      $s0, 8($0)      # [8] = 0x0000 65FF

sb      $s0, 0($0)      # [0] = 0xFF : [0] = 0xFFXX XXXX

sb      $s0, 1($0)      # [1] = 0xFF : [0] = 0xFFFF XXXX

sb      $s0, 2($0)      # [2] = 0xFF : [0] = 0xFFFF FFXX

sb      $s0, 3($0)      # [3] = 0xFF : [0] = 0xFFFF FFFF

sb      $s0, 4($0)      # [4] = 0xFF : [4] = 0xFFXX XXXX

sb      $s0, 5($0)      # [5] = 0xFF : [4] = 0xFFFF XXXX

sb      $0, 6($0)       # [6] = 0x00 : [4] = 0xFFFF 00XX

sb      $0, 7($0)       # [7] = 0x00 : [4] = 0xFFFF 0000

lw      $s1, 0($0)      # s1 = [0] = 0xFFFF FFFF

lw      $s2, 4($0)      # s2 = [4] = 0xFFFF 0000

sw      $s2, 84($0)     # [84] = 0xFFFF 0000

# expected results: dataadr === 84 = 0x00000054, writedata === 0xFFFF0000
```

Using this program, we verify if sb is indeed just storing the lowest order byte of wd and if it is being stored in the correct byte offset according to Big Endian as desired. This is a 13 instruction program so for our simulation this ends at 125 ns, at which point we check the sw results.

The above program is equivalent to the machine code below:

```
201065ff

ac100008

a0100000

a0100001
```

a0100002

a0100003

a0100004

a0100005

a0000006

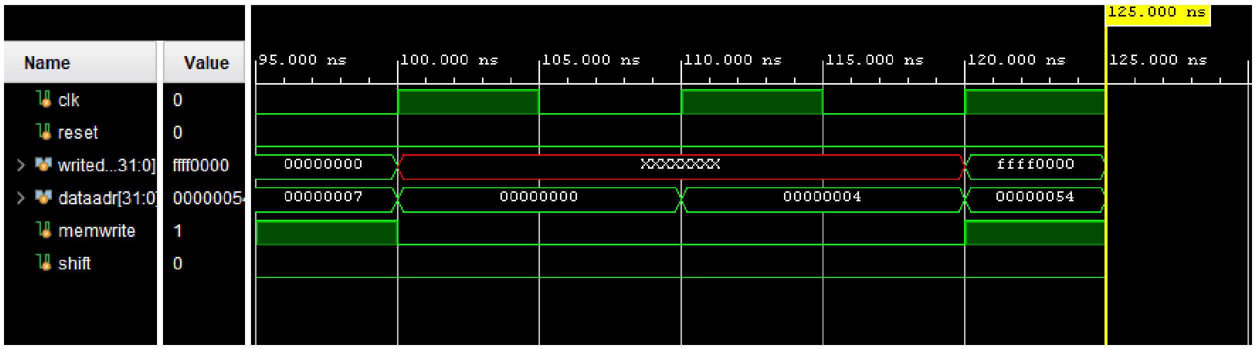
a0000007

8c110000

8c120004

ac120054

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a “Simulation suceeded” print out in Vivado’s TCL console).

ble_testbench.sv (new!)

The SystemVerilog code for ble_testbench.sv is shown below.

```
`timescale 1ns / 1ps

module ble_testbench();


  logic       clk;

  logic       reset;


  logic [31:0] writedata, dataadr;

  logic       memwrite;
```

```
// NOTE, watching shift from mips to top
logic       shift;

// instantiate device to be tested
top dut(clk, reset, writedata, dataadr, memwrite, shift);

// initialize test
initial
  begin
    reset <= 1; #1;
    reset <= 0;
    //#99 $finish;
  end

// generate clock to sequence tests
always
  begin
    clk <= 1; # 5; clk <= 0; # 5;
    $display("%d", $realtime);
  end

// check results
always @(negedge clk)
  begin
    if($realtime == 85) begin
      if(dataadr === 255 & writedata === 251) begin
        $display("Simulation succeeded");
        $stop;
      end else begin
        $display("Simulation failed");
        $stop;
      end
    end
  end
endmodule
```

The explanations for the above ble_testbench.sv code are then shown below.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named sb_testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

10: Declared a watcher for the shift signal.

13: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite and also shift.

16: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

17: Beginning of the initial block.

18-19: Sets reset to 1 for the first 1 ns of the simulation, then sets it to 0 afterwards. This initializes the PC register to instruction address 0.

21: End of this initial block.

24: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

25: Beginning of this always block.

26: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

27: And from now on, we also print the current simulation time for verification purposes. This is also inside the always block.

28: End of the always block.

31: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

32: Start of this always block.

33: An if conditional that executes at realtime == 85 ns, and this corresponds with the expected time that the processor reads the final instruction in the memory file (note that we skip one instruction by branching).

34-40: CHECKING: If dataadr === 255 & writedata === 251 (expected results), then simulation is successful (print "Simulation suceeded" then stop the program). Else, simulation failed (print "Simulation failed" then stop the program).

41: End of the memwrite condition block.

42: End of the always block.

43: End of testbench module.

Assembly Program for ble_testbench.sv

```
addi    $s0, $0, 62       # s0 = 62

sll     $s0, $s0, 1       # s0 = 62*2 = 124

sll     $s0, $s0, 1       # s0 = 124*2 = 248

addi    $s1, $0, -5       # s1 = -5 = ...1 1111 1011 (2C) = 0xFFFF FFFB

sw      $0,  0($0)        # [0] = 0x0000 0000

sb      $s1, 3($0)        # [0] = 0x0000 FB00

lw      $s2, 0($t0)       # s2 = 251 = ...0 1111 1011 = 0x0000 00FB

ble     $s0, $s2, target # should be taken since s0 <= s2 : 248 <= 251

addi    $s2, $s2, 5       # should not be executed, would cause s2 = 256

target:

addi    $s2, $s2, 4       # s2 = 255 (expected in aluout = dataadr)

# terminate after 85 sec since only 9 instructions will be executed.
```

# expected results: dataadr === 255 = 0x000000FF, writedata === 251 = 0x000000FB

Note that for ble $s0, $s2, target, we modified it so that the BTA *offset* stored in the imm field corresponds to the actual distance of the labelled address from PC+4. For the next testbenches we will keep doing this whenever we do ble.

The above test program corresponds to the following machine code.

2010003e

00108040

00108040

2011fffb

ac000000

a0110003

8c120000

7e320001

22520005

22520004

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):

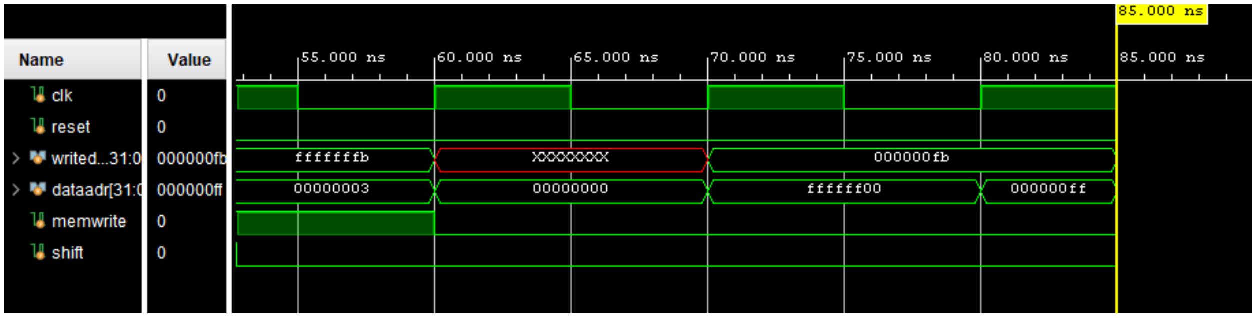


Figure 9. ble_testbench.sv simulation waveform diagram.

As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a "Simulation suceeded" print out in Vivado's TCL console).

Having passed these series of simulations, we now certify that the modified MIPS processor can now run **shift left logical (sll), store byte (sb)** and **branch on less than or equal (ble).**

Remark: As an early notice, I want to say that the next test programs will be bigger and more complicated as we culminate with this project, i.e. they will be a MIPS Assembly collage of both the original instructions and added instructions with emphasis on the most recently added.

li rt, imm

Instruction Format

The **load immediate** or **li** pseudoinstruction is an (arguably) I-type instruction with the following properties:

- Coded in MIPS Assembly as li, rt, imm.
- Opcode = 0x11 = 010001.
- No funct field, as with all I-types.
- The rs field is a Don't Care field.
- Operation: [rt] = ZeroImm = {16'b0, imm}.
    - We zero-extend any 16-bit value to 32 before storing it to the register rt.
- A simplifying assumption is that the values to be "stored are always no longer than 16-bits".

Overview of Modification

- First we modify the maindec module to recognize the li instruction opcode = 010001 and produce the appropriate signals for it. For the control signals emanating from the main decoder, only regwrite and li (a new signal) is asserted.
- For such I-type pseudoinstruction, we do not modify the ALU decoder. In fact, we don't care about aluout when li is asserted because our design decision is to simply mux the zero-extended immediate ZeroImm with the aluout using a multixplexer called limux (controlled by li) and directly connect the output to the resmux, for which resmux (since for li we have memtoreg = 0) chooses the ZeroImm over ReadData for loading into register rt.
- As mentioned, we shall introduce a multiplexer called limux

Control Signals

The following are the final control signals for this instruction. In yellow are new signals.

Remark: For consistency with the HARRIS implementation, we always take the Don't Care signals to be 0 in the code. alusrc, aluop, and sb can be X since we don't consider aluout and we don't write to memory for the li instruction.

From Main Decoder:

regwrite	regdst	alusrc	beq	memwrite	memtoreg	jump	aluop	sb	ble	li
1	0	0	0	0	0	0	0	0	0	1

From ALU Decoder:

alucontrol	shift	zfr
010 (add)	0	0

Modifications in the HDL Code

We now explain the HDL modules we modified in order to implement the li instruction.

maindec.sv

The maindec module, instantiated as md, is the module that handles the bulk of the instruction decoding.

For the li instruction, this is modified to recognize li's opcode of 010001 and generate the appropriate control signals. To be concise, the signals are regwrite = 1 and li = 1, and the rest are 0s (check Don't Care caveat). Moreover, an output port li is declared for this signal to channel it onto the mips module and into the datapath where it shall control a multiplexer named limux.

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively.

2-11: Creates a main decoder module named maindec with the following ports:

- 6-bit logic-type input port (node) named op with [MSB:LSB] = [5:0]. Used for accepting the instruction's opcode.
- 1-bit logic-type output ports named memtoreg, memwrite, beq, alusrc, regdst, regwrite, and jump. These are control signals that will then go on to dictate the flow of the datapath,
- 2-bit logic-type output port aluop with [MSB:LSB] = [1:0]. Is fed to the ALU decoder module that helps simplify the decoding of the ALU operation.
- The output ports sb (for sb onwards), ble (for ble onwards), and li (for li onwards) to handle the signals corresponding to them.

This module will be a subset of the control unit.

14: Declares a 12-bit logic-type variable named controls with [MSB:LSB] = [11:0]. This will be used to handle all the control signals generated by the main decoder as a concatenated, convenient 12-bit wire value.

- Originally, this was just a 9-bit value without the sb, ble, and li signals.
- After adding sb, this became a 10-bit value.
- After adding ble, this became an 11-bit value.
- After adding li, this became a 12-bit value.

16-17: Wirings of the proper control signals using a concatenation operator for ease of assignment. This kind of formatting is useful for simultaneously assigning multiple values that goes to different components, such as what happens with the control signals.

22: Start of an always_comb block for the combinational assignment of control signals.

23-34: A case block controlled by the opcode op. Control signals are predefined for each recognized instructions. As can be seen, R-type corresponds to opcode == 000000, and the rest are R-types.

Additions here are:

- 6'b101000: controls <= 12'b001010000_100; // SB (sb = 1) for SB onwards.
- 6'b011111: controls <= 12'b000000001_010; // BLE (beq=0, ble=1) for BLE onwards.
- 6'b010001: controls <= 12'b100000000_001; // LI (regwrite=1, li=1) for LI onwards.

Our main focus here is the LI addition. By making 6'b010001 a case controlled by op, then we make LI recognizable to the processor, and then we set the control signals for it, where regwrite = 1 and li = 1 and the rest are either 0s or Xs. Regwrite is 1 because we write to rt, and li is 1 because we want the result to be the zeroimm.

35: End of the maindec module.

ALU Decoder is not modified for the li instruction.

controller.sv

This module pertains to the control unit and also houses the pcsrc logic. This contains the main decoder and the ALU decoder.

The current main modification here is the wiring of the output li signal from maindec to outside the controller where it will be channeled to the datapath's limux.

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively.

2-13: Creates a module named controller and declares the input and output ports (also considered as nodes) of the module. Inputs are 6-bit op and funct (fields), and 1-bit zero flag from the ALU in the datapath. Outputs are 1-bit control signals memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, and jump; and 3-bits for the alucontrol.

Port modifications are as follows:

- Output shift signal from sll onwards. From alu decoder, goes to srcamux.
- Output sb signal from sb onwards. From main decoder, goes to dmem module.
- Input sign flag from ble onwards. From datapath's ALU, goes to this controller.
- Output li signal from **li** onwards. From main decoder, goes
- Output zfr signal from zfr onwards. From alu decoder, goes to zfrmux.

16: Declares 2-bit logic-type aluop. Wires aluop from maindec to aludec.

17: Originally declared as logic-type branch but is now beq from beq implementation onwards. Will serve as the control signal for beq.

18: Declares ble from ble implementation onwards. Will serve as control signal for ble.

Note that beq and ble are not declared as ports because they are only internally used by the controller, together with the zero flag and the sign flag (from ble onwards) to determine the pcsrc control signal (i.e. whether to branch or not).

20-22: Instantiates the main decoder as md. Has as input op (the opcode), and as outputs memtoreg, memwrite, beq (originally branch), alusrc, regdst, regwrite, jump, aluop.

Additions are the outputs sb (from sb onwards), ble (from ble onwards), and **li** signals (from li onwards).

Focus on the ble signal which works with beq, zero, and sign to produce the correct pcsrc.

24: Instantiates the alu decoder as ad. Inputs are funct and aluop. Outputs are alucontrol, shift, and zfr, as desired.

Note that shift and zfr are generated in this module because they are R-type instructions, and they can only be distinguished from each other by examining the funct field since they all have the same 000000 opcode (while also conveniently maintaining the proper control signals for R-type instructions as given by the main decoder). Please hang tight because we are about to discuss zfr.

Generation of sb signal is in maindec, not in the aludec.

27: Wires pcsrc to the result of branch AND zero, i.e. we branch when the instruction is in fact branch and the contents of the two registers are equal (given by their difference being zero).

Note that for the addition of ble this is expanded to become assign pcsrc = (beq & zero) | (ble & (sign | zero)). In other words, there are now two possibilities for branching: when beq or when ble. This Boolean expression is generated by the simple expression that the processor should:

Branch if the instruction is beq and the difference of [rs] and [rt] is zero (meaning [rs] == [rt]), or branch if the instruction is ble and the difference of [rs] and [rt] is negative (meaning [rs] < [rt]) or is zero (meaning [rs] == [rt]). Essentially read as branch if equal or branch if less than or equal. Note that

$$\text{pcsrc} = \text{beq} \,\&\, \text{zero} \mid (\text{beq} \,\&\, \text{sign} \mid \text{beq} \,\&\, \text{zero}) = \text{beq} \,\&\, \text{zero} \mid (\text{beq} \,\&(\text{sign} \mid \text{zero}))$$

The signal PCSrc controls which next instruction address will be fed to the instruction memory, whether it be for branch (BTA) or in our situation, jump (JTA) is also possible.

28: End of the controller module.

mips.sv

Now we go to the mips module which contains the control unit and the datapath. This is the SCP minus the instruction memory (imem) and the data memory (dmem). In essence, the mips module serves as relay between the controller and the datapath and this allows the two components to communicate.

We modify this module for li to wire the li signal from the controller to the datapath where it will control a multiplexer named limux which would decide whether or not to take zeroimm as result.

1: Sets the module’s delay unit and delay resolution to 1ns and 1ps respectively.

2-9: Creates a module named mips with inputs clk (the clock driver), reset (for resetting the PC register), instr (the whole 32-bit instruction), and 32-bit readdata (pertains to result of combinational read of dmem, only actually used for lw). Outputs are pc (from the data path, gives the next PC address, default is PC+4, special cases for branches and jumps), memwrite (from controller, controls whether to write or not to dmem, aluout (because we want to watch aluout for testing), writedata (we also want to watch this).

Additions are the shift output signal (I placed this here because I personally want to watch this).

sb is added as output port from sb onwards. We still want it to go out of this module because it should head to data memory.

12-13: Declares variables memtoreg, alusrc, regdst, regwrite, jump, pcsrc, and zero. Used for wiring between the controller and the datapath, usually controls multiplexers in the datapath, except for the zero flag which is wired from the ALU to the controller to control PCSrc.

14: Declares 2-bit alucontrol, also wired from controller's aludec to the controller.

16-18: Declares the wirings for

- the sign flag (from ble onwards), goes from datapath to controller.
- the **li** signal (from li onwards), goes from controller to datapath.
- and the zfr signal (from zfr onwards), goes from controller to datapath.

20-27: Instantiates a controller module and names it c. It has as inputs 6-bit instr[31:26] (opcode) and instr[5:0] (possibly funct), and the zero flag. Output ports are memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump, and 2-bit alucontrol. Aka the control unit.

Port additions are:

- output shift signal, for controlling srcamux (sll onwards).
- output sb signal, for controlling dmem store (sb onwards).
- input sign flag, for deciding pcsrc (ble onwards).
- output **li** signal, for controlling limux (li onwardsd).
- output zfr signal, for controlling zfrmux (zfr onwards).

Focus on ble and how it goes from datapath ALU to controller maindec.

30-37: Instantiates a datapath module and names it dp. Has as inputs clk, reset, memtoreg pcsrc, alusrc, regdst, regwrite, jump, 2-bit alucontrol, the 32-bit instruction, 32-bit read data chiming in from dmem, and the shift signal for the shift multiplexer.

Port additions are (some may be reiterated):

- input shift signal, for controlling srcamux (sll onwards).
- output sign flag, for deciding pcsrc (ble onwards).
- input **li** signal, for controlling limux (li onwardsd).
- input zfr signal, for controlling zfrmux (zfr onwards).

38: End of the mips module.

datapath.sv

The datapath is where all the main computations that generates register results take place.

This is modified with an insertion of two new components: an implicit zero extender from 16 bits to 32 bits that would take in instr[15:0] to produce ZeroImm, and then we have a new multiplexer named limux that will choose between aluout and the ZeroImm. The limux is controlled by the signal li (only asserted for li instructions) and its output goes to D0 of resmux for further choosening.

Remark: We explain in further detail the modifications for zfr (includes a new constant, left shifter, bit selector, and multiplexer) in its documentation.

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively.

2-15: Creates a module named datapath.

- Has as inputs clk for the clock driver, reset for the reset initializer or just plain reset; control signals memtoreg, pcsrc, alusrc, regdst, regwrite, jump, and 2-bit alucontrol; 32-bit instr which is read from imem via interpreting PC; 32-bit readdata that comes in from dmem for load word; and new signals shift (for sll onwards), li (for li onwards), and zfr (for zfr onwards).
- Has as outputs zero flag, 32-bit pc for the next instruction address (goes to imem), 32-bit aluout tied to dataadr on mips module instantiation of datapath and beyond for testbench watching, 32-bit writedata for the data to be written to the dmem via store instructions, and a new sign flag for branch instructions.

New muxes srcamux, limux, and zfrmux are added/to be added on their respective instructions.

18-25: Declare 5-bit writereg; and 32-bit pcnext, pcnextbr, pcplus4, pcbranch, signimm, signimmsh, RD1, RD2, srca, srcb, liout (new!), result, and zfrout (new!). After some erroneous simulations, I learned the value of first declaring nodes and most importantly their sizes before wiring together the modules using said nodes.

These declarations help guide the wires to connect to the write places, those preceeded with pc deals with the instruction addressing, and the other deal with the main data path.

Changes I did here is to add RD1 and RD2 to correspond to the rs and rt value outputs of the regfile respectively. This is done for clarity.

28: always_comb block start. As explained previously, this watches all variables that are manipulated inside, and executes on changes. I placed this here to watch the changes in dataaddr and writedata more internally (because for the testbench we simply just use comparisons of dataaddr and writedata actual values to their expected values to verify the correctness of the simulation).

31: Displays the dataaddr and writedata contents whenever there are changes with them. Note that dataaddr is wired to alutout and writedata is wired to RD2.

32: End of the always_comb block above.

35: Instantiates flopr as pcreg, parameter WIDTH=32 with inputs clk, reset, 32-bit pcnext (seen as PC' in the HARRIS schematic), and output 32-bit pc. This serves as the pc register which updates every clock rising edge. Flopr is used because of the need for a resettable flipflop (reset required usually for initializing the instruction address). Please refer to my Lab Report 9 for line-by-line explanations of the contents of flopr.

Note that pc goes out to imem to specify the next instruction address to be read.

36: Instantiates adder as pcadd1, parameter WIDTH = 32, with inputs pc, 32’b100 (constant 4), and ‘b0 (0 carry in), and outputs pcplus4 (namesake). Constantly generates PC+4 for *consideration*. Please refer to my Lab Report 9 for line-by-line explanations of the contents of adder.

37: Instantiates sl2 as immsh with input signimm (sign-extended instr[15:0]) and output signimmsh. This shifts the input to the left by 2 (multiplies by 4) as the first step to generating BTA. Please refer to my Lab Report 9 for line-by-line explanations of the contents of sl2.

38: Instantiates adder as pcadd2, WIDTH = 32, with addends pcplus4, signimmsh, and 0 for carry in. Output is pcbranch, for consideration. When we say consideration, we imply that the value is being muxed with something else. This completes BTA = PC+4 + (SignImm << 2). Please refer to my Lab Report 9 for line-by-line explanations of the contents of mux2.

39: Instantiates mux2 as pcbrmux, WIDTH = 32, to choose between 0: pcplus4 and 1: pcbranch. Controlled by pcsrc. Result becomes pcnextbr.

40: Instantiates mux2 as pcmux, WIDTH = 32, to choose between pcnextbr and JTA = {(PC + 4)[31:28], addr, 2’b0}. Controlled by jump.

Caveat on this as promised earlier is that we have to realize that there are two multiplexers multiplexing which instruction address to feed to the PC register. The first is the pcbrmux, controlled by pcsrc that is of clear importance to ble. And the next is pcmux, in that order, which with a slight modification now decides whether to branch, or jump, or neither. This order is the reason why if we want to branch, we also need jump to be 0, but if we just want to jump, branch (ble or beq) is a Don’t Care, along with the majority of other control signals.

Remark on Multiplexers: When we say “choose between D1 and D2”, we imply that select signal = 0 passes D1 and select signal = 1 passes D2. This is appropriate because we **only use 2-way multiplexers**. Please keep this remark and all other remarks in mind.

All these complete the choice for the next PC value, whether it be PC+4, BTA, or JTA. Note that we did not made any modifications for this portion, so these explanations will just repeat over and over again, as is.

44: Clarity modification, we assign writedata to RD2. Note that RD2 comes from the register file, and it is wired to WriteData WD of data memory with nothing in between (constantly zero multiplexers in the connection). This is a watched variable.

45-46: Instantiates regfile as rf. Inputs are the clk (for possibly writing sequentially), regwrite signal, instr[25:21] = rs, instr[20:16] = rt, writereg or value to be written, and result coming in from resmux. Outputs are RD1 and RD2. The register file will be written too on clock rising edge if WE3 is asserted by RegWRite.

This regfile is very integral to the processor and as we all know this is like the heart that beats with every instruction we pump in it in line with the synapses of the control unit which is kind of like the pace maker of this system.

47-48: Instantiates mux2 as wrmux with WIDTH = 5 (begause register address is always just 5 bits). This chooses between rt and rd over which to use as the write address for the register file. This is controlled by regdst (register destination). Outputs the address to writereg and is fed back to the register file.

49: This is an **li modification**. For li onwards, a new multiplexer named limux is introduced. It is located somewhere between dmem and resmux in our schematic diagram (for li onwards!). This chooses between aluout and ZeroImm, and it decides using the li signal.

50: Instantiates mux2 with width 32 as resmux. Chooses between the output of ALU, aluout (rather liout from li onwards!), and the data that was combinationally read from dmem, readdata. Controlled by memtoreg. Outputs to result, and that result is fed back to the register file.

50: Instantiates mux2 with width 32 as resmux. Chooses between the output of ALU, aluout, and the data that was combinationally read from dmem, readdata. Controlled by memtoreg. Outputs to result, and that result is fed back to the register file.

51: Instantiates signext as se. Takes in the perceived immediate field instr[15:0] of any instruction and sign-extends the contents to 32-bit, and the result is passed to signimm. Please refer to my Lab Report 9 for line-by-line explanations of the contents of signext.

56: This is an sll modification. Instantiates mux2 as srcamux, 32-bit. Chooses between RD1 and the zero-extended contents of the shamt field of the instruction (instr[10:6]). This is controlled by the shift signal. The result of this is passed to srca which is wired to the ALU.

57: Instantiates mux2 as srcbmux, 32-bit. Chooses between RD2 and signimm (sign-extended immediate). Controlled by alusrc. Output passed to srcb. Originally this connects directly to srcb port of the ALU, but this will change for zfr which introduces a new mux, zfrmux, between that connection. Note that previously RD2 was just written as writedata, but that is not so descriptive so we wired writedata and RD1 to each other.

The motivation for lines 56-57 is that we turned that unused alucontrol assignment to a shift operation within the ALU, to be precise, it is result = b << a. Sa if the instruction is shift, srca of ALU receives the desired shift amount stored in the instruction's shamt field, and srcb receives the value to shift from rt.

58: This is a zfr modification. From zfr onwards, we insert a multiplexer aptly named zfrmux which chooses between srcb (from srbmux) and the bitmask for the upcoming AND operation (given by 32'hFFFF_FFFE << RD2[4:0]). We explain this in further detail during the zfr documentation.

59: Instantiates alu as alu. Operand A is srca, and Operand B is aluout (zfrout later on). Controlled by alucontrol. Outputs result to aluout, and also generates zero flag (and sign flag for ble later on).

Per HARRIS, an Arithmetic/Logical Unit (ALU) combines a variety of mathematical and logical operations into a single unit. For example, a typical ALU might perform addition, subtraction, magnitude comparison, AND, and OR operations. In our case, we use a 3-bit control signal alucontrol which specifies 7 default functions or operations (8 if we include the recently added sll instruction for alucontrol 011). The ALU forms the heart of most computer systems, where the alucontrol is like the passive brain signal that directs which artery or multiplexer data source (I know, kind of stretching the analogy here) will the *actual* processed result come from, because most ALU implementations actually computes the respective operation results simultaneously, and the alucontrol just selects which of those results will actually be presented.

The **sign flag** is also now added as an output of the ALU. This signal helps control our branches.

61: Ends the datapath module.

alu.sv

The ALU was not modified for the li instruction. In fact we don't care about aluout during the li instruction, since we are assured that if the control signal li = 1, then the ALU result will be ignored by the limux which would instead pass ZeroImm for loading into rt.

Schematic Diagram

We now present the schematic diagram for the li instruction modifications.

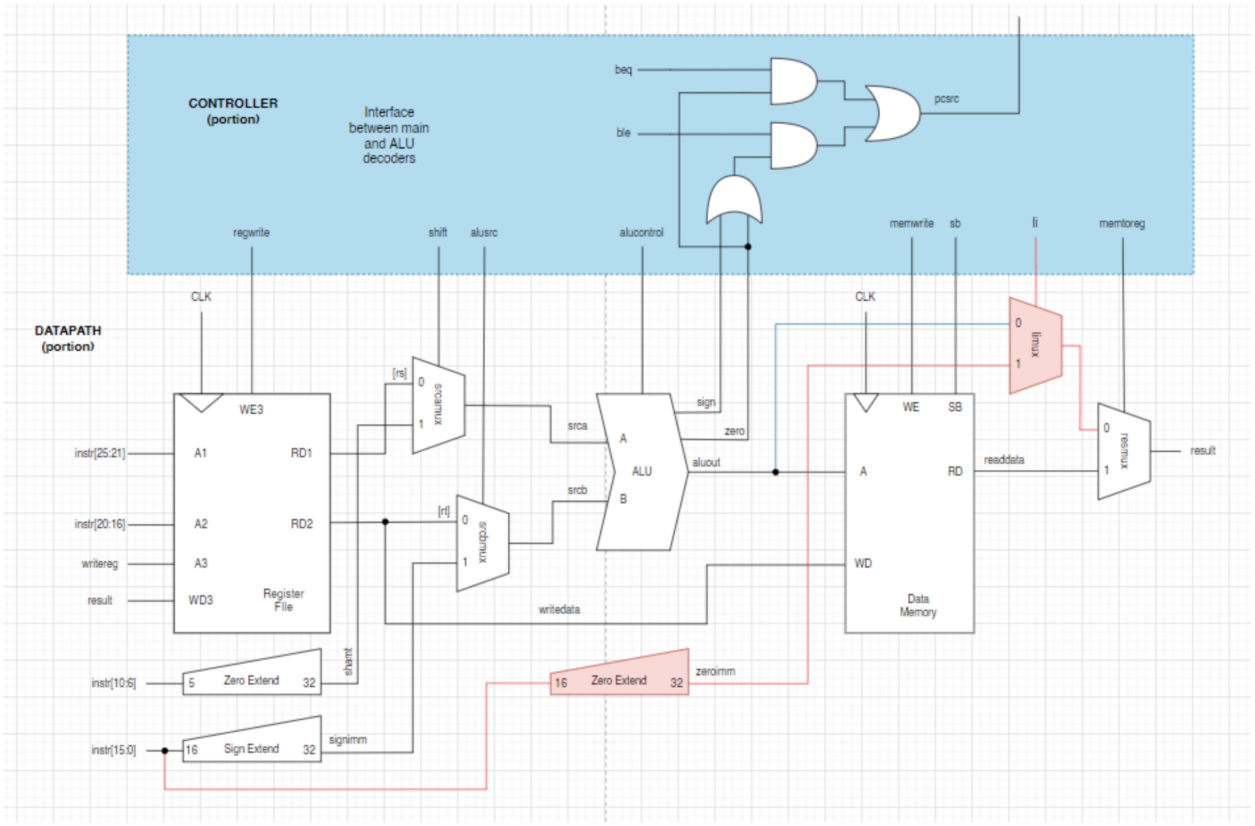


Figure 10. li schematic diagram (partial view of the datapath and controller only).

Again, the modifications are in blue, and the additions are in red.

Modified is the Controller in order to make it recognize the li instruction so that it generates the proper control signals for the instruction. Also modified is the datapath wiring from aluout to resmux, wherein limux is inserted in the path of the wire.

Additions are the zero extender that takes in instr[15:0] and Zero Extends it to 32-bits ZeroImm, and the limux which chooses between aluout and ZeroImm which is decided by the li signal.

Testbenches and Test Programs

Once again, we run the modified MIPS processor through a gauntlet of integrity tests consisting of the previous testbenches, and finally a new testbench dedicated to this instruction.

The new testbench, named li_testbench, also takes the opportunity to further test sll and ble as ble_testbench also did for sll and sb.

The testbenches and their corresponding test programs now follow.

testbench.sv

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

11: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite.

14: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

15: Beginning of the initial block.

16: Sets reset to 1 for the first 22 ns of the simulation, then sets it to 0 afterwards.

17: End of this initial block.

20: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

21: Beginning of this always block.

22: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

23: End of the always block.

26: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

27: Start of this always block.

28: An if conditional that executes its block if memwrite is asserted. Corresponds to initializing the actual test when sw or sb is the instruction.

29-35: CHECKING: If dataadr === 84 & writedata === 7, then simulation is successful (print “Simulation suceeded” then stop the program). Else if dataadr !=== 80, simulation failed (print “Simulation failed” then stop the program). The reason for that failure condition is that in the test program, there’s a non-terminating store word with effective address 80, so if it happens that dataadr === 80 when memwrite == 1, then it means that there’s a mix-up in the computations.

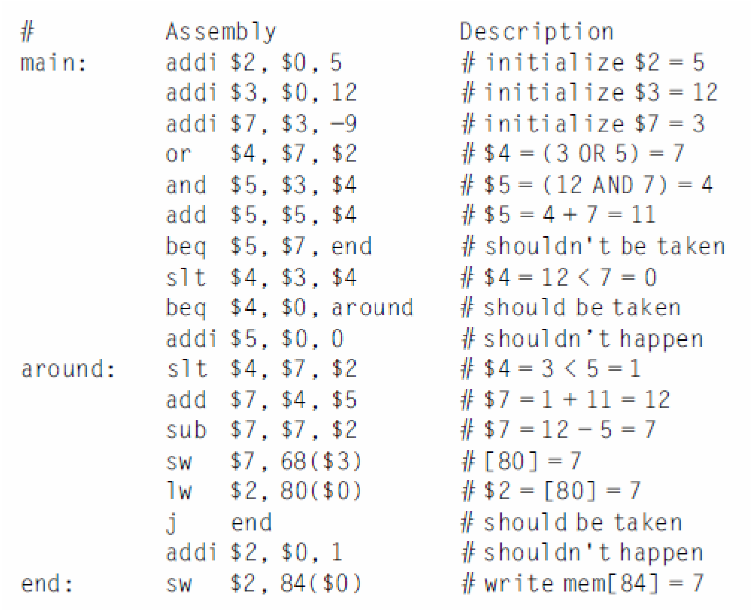
36: End of the memwrite condition block.

37: End of the always block.

38: End of the testbench module.

Assembly Program for testbench.sv

Below is the MIPS program that this testbench will run through. Courtesy of HARRIS.



```
#         Assembly              Description
main:     addi $2, $0, 5        # initialize $2 = 5
          addi $3, $0, 12       # initialize $3 = 12
          addi $7, $3, −9       # initialize $7 = 3
          or   $4, $7, $2       # $4 = (3 OR 5) = 7
          and  $5, $3, $4       # $5 = (12 AND 7) = 4
          add  $5, $5, $4       # $5 = 4 + 7 = 11
          beq  $5, $7, end      # shouldn't be taken
          slt  $4, $3, $4       # $4 = 12 < 7 = 0
          beq  $4, $0, around   # should be taken
          addi $5, $0, 0        # shouldn't happen
around:   slt  $4, $7, $2       # $4 = 3 < 5 = 1
          add  $7, $4, $5       # $7 = 1 + 11 = 12
          sub  $7, $7, $2       # $7 = 12 − 5 = 7
          sw   $7, 68($3)       # [80] = 7
          lw   $2, 80($0)       # $2 = [80] = 7
          j    end              # should be taken
          addi $2, $0, 1        # shouldn't happen
end:      sw   $2, 84($0)       # write mem[84] = 7
```

The machine code below is equivalent to the program above. We fill the memory file memfile.mem with this machine code if we want run this testbench.

20020005

2003000c

2067fff7

00e22025

00642824

00a42820

10a7000a

0064202a

10800001

20050000

00e2202a

00853820

00e23822

ac670044

8c020050

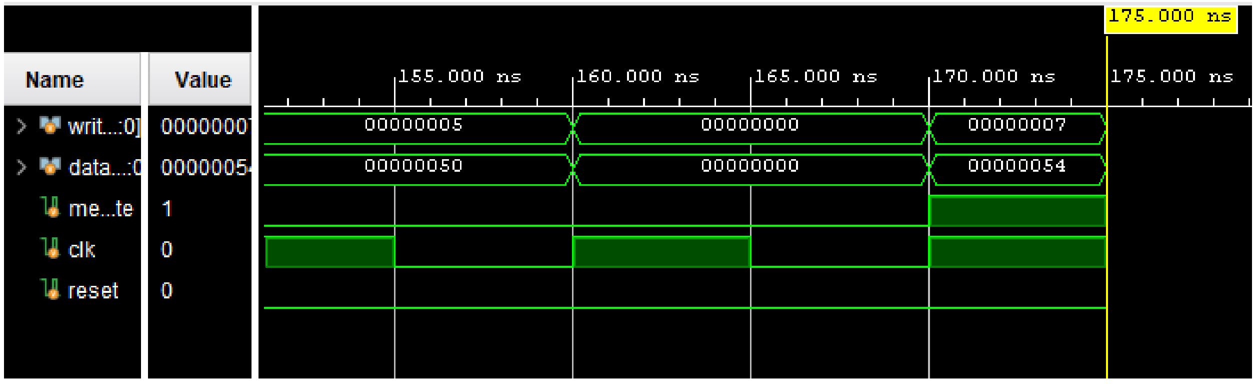
08000011

20020001

ac020054

Line-by-line description of the program is already given by HARRIS.

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a "Simulation suceeded" print out in Vivado's TCL console).

testbench_2.sv

This is just a slightly modified version of testbench.sv to accommodate the new test program.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named testbench_2. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

11: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite.

14: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

15: Beginning of the initial block.

16: Sets reset to 1 for the first 5 ns of the simulation, then sets it to 0 afterwards (totally arbitrary). Adds a delay of 255 ns that corresponds to the actual runtime of a successful simulation. This explicitly states that reset = 0 for that time frame.

17: End of this initial block.

20: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

21: Beginning of this always block.

22: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

23: End of the always block.

26: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

27: Start of this always block.

28: An if conditional that executes its block if memwrite is asserted. Corresponds to initializing the actual test when sw or sb is the instruction.

Remark: We are now only concerned in getting the Simulation Success Conditions so we leave the Simulation Failed Condition as simply the result of not attaining the simulation success condition.

29-35: CHECKING: If dataadr === 16 & writedata === 0xBBAAB0B0, then simulation is successful (print "Simulation suceeded" then stop the program). Else, simulation failed (print "Simulation failed" then stop the program).

36: End of the memwrite condition block.

37: End of the always block.

38: End of writing the testbench module.

Assembly Program for testbench_2.sv

```
# Write the hex value 0xBBAA0000 to register 1

addi    $1, $0, 0xBBAA  # 2001BBAA

add     $1, $1, $1      # 00210820

add     $1, $1, $1      # ...

add     $1, $1, $1

add     $1, $1, $1

add     $1, $1, $1

add     $1, $1, $1

add     $1, $1, $1

add     $1, $1, $1

add     $1, $1, $1

add     $1, $1, $1

add     $1, $1, $1

add     $1, $1, $1

add     $1, $1, $1

add     $1, $1, $1

add     $1, $1, $1

add     $1, $1, $1

# Others

addi    $2, $0, 176     # 200200B0

addi    $3, $2, 45056   # 2043B000

addi    $4, $0, 0x7FFF  # 20047fff
```

```
add     $4, $4, $4      # 00842020
addi    $4, $4, 1       # 20840001
and     $3, $3, $4      # 00641824
add     $4, $1, $3      # 00232020
addi    $5, $0, 16      # 20050010
sw      $4, 0($5)       # aca40000
```

Explanation:

1: Insert 0x0000 BBAA to register 1.

2-18: Repeatedly add it to itself 16 times in order to emulate a load upper immediate (lui) instruction. After this register 1 contains 0xBBAA 0000.

20: Insert value 176 to register 2.

21: $3 = $2 + 45056 = 176 + 45056.

22: Insert 0x0000 7FFF to $4.

23. Add $4 to itself and store result to $4.

24: Add a 1 to the content of register 4 and store result to register 4.

25: $3 = $3 AND $4.

26: $4 = $1 + $3

27: Store value 16 to $5.

28: Store word content of $4 (0xBBAAB0B0 after all the previous computations) to effective address given by $5 (16).

The above program correspond to the machine code below:

```
2001bbaa
00210820
00210820
00210820
00210820
00210820
00210820
```

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

200200b0

2043b000

20047fff

00842020

20840001

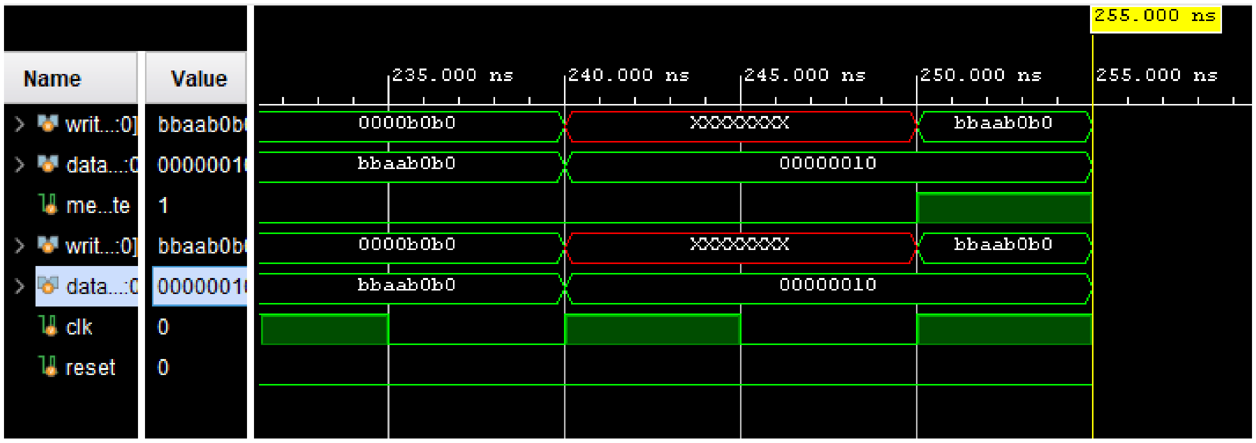
00641824

00232020

20050010

aca40000

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a “Simulation suceeded” print out in Vivado’s TCL console).

sll_testbench.sv

The SystemVerilog code for the sll_testbench.sv module is provided below.

```
`timescale 1ns / 1ps

module sll_testbench();


 logic      clk;

 logic      reset;


 logic [31:0] writedata, dataadr;

 logic      memwrite;

 // NOTE, watching shift from mips to top

 logic      shift;


 // instantiate device to be tested

 top dut(clk, reset, writedata, dataadr, memwrite, shift);


 // initialize test

 initial

  begin

   reset <= 1; #1;

   reset <= 0;

   #99 $finish;

  end


 // generate clock to sequence tests

 always
```

```
  begin
    clk <= 1; # 5; clk <= 0; # 5;
  end


 // check results
 always @(negedge clk)
  begin
    if(memwrite) begin
      if(dataadr === 84 & writedata === 'h0C00000C) begin
        $display("Simulation succeeded");
        $stop;
      end else begin
        $display("Simulation failed");
        $stop;
      end
    end
  end
endmodule
```

This is the main testbench for sll. This is just an appropriation of the previous testbenches.

The explanations for the sll_testbench.sv code above are provided below.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named sll_testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

10: Declared a watcher for the shift signal.

13: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite and also shift.

16: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

17: Beginning of the initial block.

18-20: Sets reset to 1 for the first 1 ns of the simulation, then sets it to 0 afterwards. This initializes the PC register to instruction address 0. After 99 ns end the program, corresponding to actual runtime of program (we could just remove this).

21: End of this initial block.

24: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

25: Beginning of this always block.

26: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

27: End of the always block.

30: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

31: Start of this always block.

32: An if conditional that executes its block if memwrite is asserted. Corresponds to initializing the actual test when sw or sb is the instruction.

Remark: We are now only concerned in getting the Simulation Success Conditions so we leave the Simulation Failed Condition as simply the result of not attaining the simulation success condition.

33-39: CHECKING: If dataadr === 84 & writedata === 0x0C00000C (expected results, see assembly program explanation or annotations), then simulation is successful (print “Simulation suceeded” then stop the program). Else, simulation failed (print “Simulation failed” then stop the program).

40: End of the memwrite condition block.

41: End of the always block.

42: End of writing the testbench module.

Assembly Program for sll_testbench.sv

addi $s0, $0, 0x000C

```
sll  $s0, $s0, 8

sll  $s0, $s0, 8

sll  $s0, $s0, 8

addi $s0, $s0, 0x000C

sw   $s0, 84($0)
```

Explanation:

1: $s0 = 0 + 0x0000 000C = 0x0000 000C

2-3: SLL TESTING: 0x0000 000C should be shifted by 24 bits to the left in total. Result would be 0x0C00 0000.

4: Then add 0x000C to 0x0C00 0000 and store that back to $s0. $s0 should be 0x0C00 000C at this point.

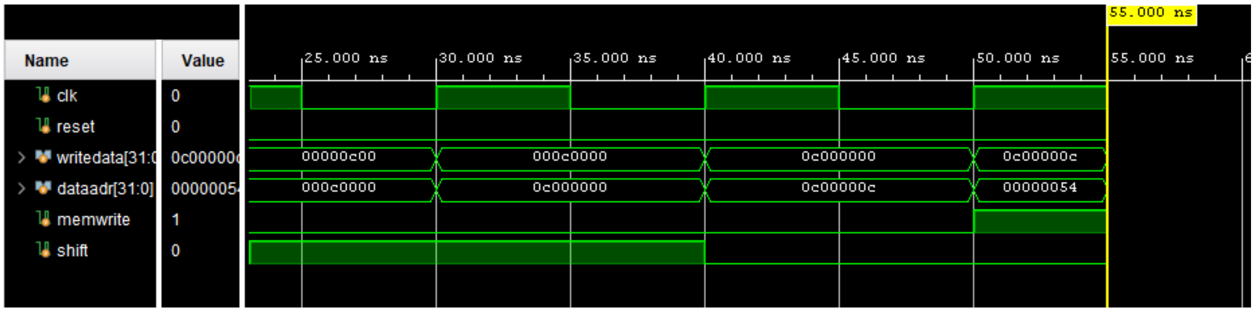
5: Store word contained by $s0 to effective address 84 (may break MARS but for the Vivado simulation it's just fine). $s0 must contain 0x0C00 000C for this simulation to be successful.

The corresponding machine code for this is simply

```
2010000c

00108200

00108200

00108200

2210000c

ac100054
```

Said machine code is as is and did not need any adjustments to work in the context of the memory file and the processor. Moreover, recall that the **rs field** is a don't care for sll; in our case, we simply set the rs field to 0s, but for future instructions we also use variations of bits for the don't cares. Also, by design, we can be assured that the rs field of sll is really treated as a don't care because our shift signal selects between the rs contents and the zero-extended shamt, so it's just either one of the two.

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a "Simulation suceeded" print out in Vivado's TCL console).

sb_testbench.sv

The SystemVerilog code for the sb_testbench.sv module is provided below.

```
`timescale 1ns / 1ps

module sb_testbench();


  logic       clk;

  logic       reset;


  logic [31:0] writedata, dataadr;

  logic       memwrite;

  // NOTE, watching shift from mips to top

  logic       shift;


  // instantiate device to be tested

  top dut(clk, reset, writedata, dataadr, memwrite, shift);


  // initialize test

  initial

    begin

      reset <= 1; #1;
      reset <= 0;
      //#99 $finish;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5; $display("%d", $realtime);
    end

  // check results
  always @(negedge clk)
    begin
      if($realtime == 125) begin
        if(dataadr === 84 & writedata === 'hFFFF0000) begin
          $display("Simulation succeeded");
          $stop;
        end else begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule
```

The explanations for the sb_testbench.sv code above are provided below.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named sb_testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

10: Declared a watcher for the shift signal.

13: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite and also shift.

16: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

17: Beginning of the initial block.

18-19: Sets reset to 1 for the first 1 ns of the simulation, then sets it to 0 afterwards. This initializes the PC register to instruction address 0.

21: End of this initial block.

24: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

25: Beginning of this always block.

26: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns. And from now on, we also print the current simulation time for verification purposes.

27: End of the always block.

30: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

31: Start of this always block.

32: An if conditional that executes at realtime == 125 ns, and this corresponds with the expected time that the processor reads the final instruction in the memory file.

Remark: We are now only concerned in getting the Simulation Success Conditions so we leave the Simulation Failed Condition as simply the result of not attaining the simulation success condition.

33-39: CHECKING: If dataadr === 84 & writedata === 0xFFFF0000 (expected results), then simulation is successful (print “Simulation suceeded” then stop the program). Else, simulation failed (print “Simulation failed” then stop the program).

40: End of the memwrite condition block.

41: End of the always block.

42: End of testbench module.

Assembly Program for sb_testbench.sv

```
addi    $s0, $0, 0x65FF # s0 = 0x0000 65FF
sw      $s0, 8($0)      # [8] = 0x0000 65FF
sb      $s0, 0($0)      # [0] = 0xFF : [0] = 0xFFXX XXXX
sb      $s0, 1($0)      # [1] = 0xFF : [0] = 0xFFFF XXXX
sb      $s0, 2($0)      # [2] = 0xFF : [0] = 0xFFFF FFXX
sb      $s0, 3($0)      # [3] = 0xFF : [0] = 0xFFFF FFFF
sb      $s0, 4($0)      # [4] = 0xFF : [4] = 0xFFXX XXXX
sb      $s0, 5($0)      # [5] = 0xFF : [4] = 0xFFFF XXXX
sb      $0, 6($0)       # [6] = 0x00 : [4] = 0xFFFF 00XX
sb      $0, 7($0)       # [7] = 0x00 : [4] = 0xFFFF 0000
lw      $s1, 0($0)      # s1 = [0] = 0xFFFF FFFF
lw      $s2, 4($0)      # s2 = [4] = 0xFFFF 0000
sw      $s2, 84($0)     # [84] = 0xFFFF 0000
# expected results: dataadr === 84 = 0x00000054, writedata === 0xFFFF0000
```

Using this program, we verify if sb is indeed just storing the lowest order byte of wd and if it is being stored in the correct byte offset according to Big Endian as desired. This is a 13 instruction program so for our simulation this ends at 125 ns, at which point we check the sw results.

The above program is equivalent to the machine code below:

```
201065ff

ac100008

a0100000

a0100001

a0100002

a0100003

a0100004
```

a0100005

a0000006

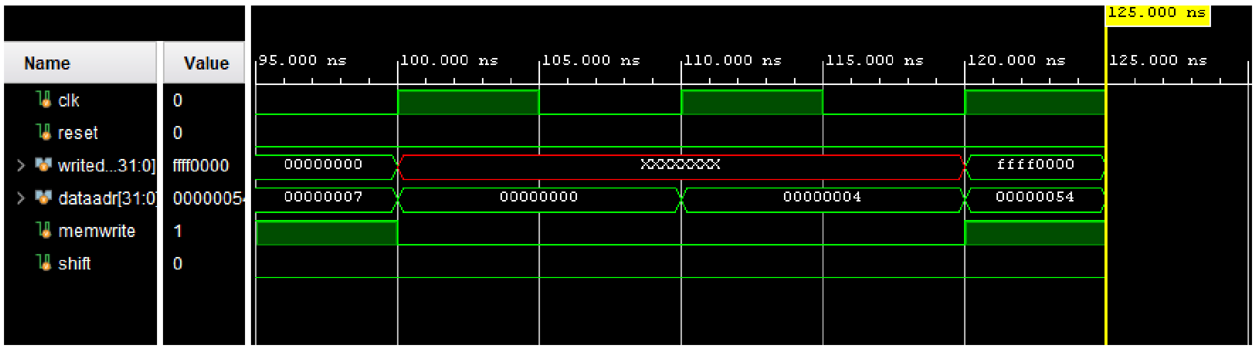
a0000007

8c110000

8c120004

ac120054

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a "Simulation suceeded" print out in Vivado's TCL console).

ble_testbench.sv

The SystemVerilog code for ble_testbench.sv is shown below.

```
`timescale 1ns / 1ps

module ble_testbench();


 logic      clk;

 logic      reset;


 logic [31:0] writedata, dataadr;

 logic      memwrite;

 // NOTE, watching shift from mips to top

 logic      shift;
```

```
// instantiate device to be tested
top dut(clk, reset, writedata, dataadr, memwrite, shift);


// initialize test
initial
  begin
    reset <= 1; #1;
    reset <= 0;
    //#99 $finish;
  end


// generate clock to sequence tests
always
  begin
    clk <= 1; # 5; clk <= 0; # 5;
    $display("%d", $realtime);
  end


// check results
always @(negedge clk)
  begin
    if($realtime == 85) begin
      if(dataadr === 255 & writedata === 251) begin
        $display("Simulation succeeded");
        $stop;
      end else begin
        $display("Simulation failed");
        $stop;
      end
    end
  end
  endmodule
```

The explanations for the above ble_testbench.sv code are then shown below.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named sb_testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

10: Declared a watcher for the shift signal.

13: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite and also shift.

16: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

17: Beginning of the initial block.

18-19: Sets reset to 1 for the first 1 ns of the simulation, then sets it to 0 afterwards. This initializes the PC register to instruction address 0.

21: End of this initial block.

24: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

25: Beginning of this always block.

26: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

27: And from now on, we also print the current simulation time for verification purposes. This is also inside the always block.

28: End of the always block.

31: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

32: Start of this always block.

33: An if conditional that executes at realtime == 85 ns, and this corresponds with the expected time that the processor reads the final instruction in the memory file (note that we skip one instruction by branching).

34-40: CHECKING: If dataadr === 255 & writedata === 251 (expected results, see assembly program explanation or annotations), then simulation is successful (print "Simulation suceeded" then stop the program). Else, simulation failed (print "Simulation failed" then stop the program).

41: End of the memwrite condition block.

42: End of the always block.

43: End of testbench module.

Assembly Program for ble_testbench.sv

```
addi    $s0, $0, 62       # s0 = 62

sll     $s0, $s0, 1       # s0 = 62*2 = 124

sll     $s0, $s0, 1       # s0 = 124*2 = 248

addi    $s1, $0, -5       # s1 = -5 = ...1 1111 1011 (2C) = 0xFFFF FFFB

sw      $0,  0($0)        # [0] = 0x0000 0000

sb      $s1, 3($0)        # [0] = 0x0000 FB00

lw      $s2, 0($t0)       # s2 = 251 = ...0 1111 1011 = 0x0000 00FB

ble     $s0, $s2, target # should be taken since s0 <= s2 : 248 <= 251

addi    $s2, $s2, 5       # should not be executed, would cause s2 = 256

target:

addi    $s2, $s2, 4       # s2 = 255 (expected in aluout = dataadr)

# terminate after 85 sec since only 9 instructions will be executed.

# expected results: dataadr === 255 = 0x000000FF, writedata === 251 = 0x000000FB
```

Note that for ble $s0, $s2, target, we modified it so that the BTA *offset* stored in the imm field corresponds to the actual distance of the labelled address from PC+4. For the next testbenches we will keep doing this whenever we do ble.

The above test program corresponds to the following machine code.

2010003e

00108040

00108040

2011fffb

ac000000

a0110003

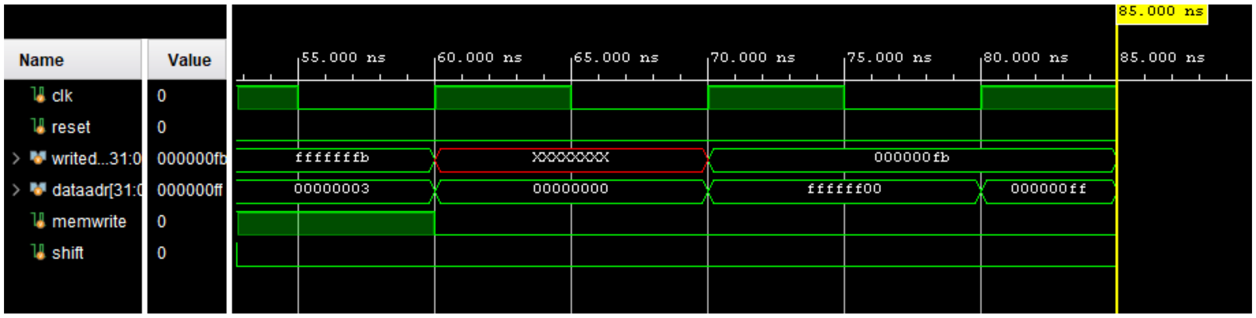
8c120000

7e320001

22520005

22520004

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a "Simulation suceeded" print out in Vivado's TCL console).

li_testbench.sv (new!)

The SystemVerilog code for li_testbench.sv is shown below.

```
`timescale 1ns / 1ps

module li_testbench();


  logic      clk;

  logic      reset;


  logic [31:0] writedata, dataadr;
  logic      memwrite;
  // NOTE, watching shift from mips to top
  logic      shift;

  // instantiate device to be tested
  top dut(clk, reset, writedata, dataadr, memwrite, shift);

  // initialize test
  initial
    begin
      reset <= 1; #1;
      reset <= 0;
      //#99 $finish;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
      $display("%d", $realtime);
    end

  // check results
  always @(negedge clk)
    begin
      if(memwrite) begin
        if(dataadr === 32'h1001_0004 & writedata === 32'h7FFF_8000) begin
          $display("Simulation succeeded");
          $stop;
        end else begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
    endmodule
```

The explanations for the above li_testbench.sv code are then shown below.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named sb_testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

10: Declared a watcher for the shift signal.

13: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite and also shift.

16: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

17: Beginning of the initial block.

18-19: Sets reset to 1 for the first 1 ns of the simulation, then sets it to 0 afterwards. This initializes the PC register to instruction address 0.

21: End of this initial block.

24: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

25: Beginning of this always block.

26: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

27: And from now on, we also print the current simulation time for verification purposes. This is also inside the always block.

28: End of the always block.

31: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

32: Start of this always block.

33: An if conditional that executes when memwrite is asserted. We have written the test program to execute sw at the end, so the sole execution of this conditional will correspond to the actual checking of the results.

34-40: CHECKING: If dataadr === 32'h1001_0004 & writedata === 32'h7FFF_8000 (expected results, see annotated assembly program), then simulation is successful (print “Simulation suceeded” then stop the program). Else, simulation failed (print “Simulation failed” then stop the program).

41: End of the memwrite condition block.

42: End of the always block.

43: End of testbench module.

Assembly Program for li_testbench.sv

```
        addi    $s0, $0, 0x0010         #s0 = 0x0010
        add     $s1, $s0, $s0           #s1 = 0x0020
        slt     $s2, $s0, $s1           #s2 = 0x0001
        ble     $s2, $s0, L1            #branch to L1
        addi    $s2, $s2, 0x7FFF        #don't run, s2=8000
        addi    $s2, $s2, 0x0001        #don't run, s2=8001
L1:     addi    $s2, $s2, 0x7FFE        #s2 = 0x7FFF
        li      $s3, 0x7FFF             #s3 = 0x7FFF
        and     $s2, $s2, $s3           #s2 = 0x7FFF
        ble     $s2, $s3, L2            #branch to L2
        sub     $s2, $s2, $s3           #don't run, s2 = 0x0000
```

```
L2:     sll     $s2, $s2, 16            #s2 = 0x7FFF 0000
        or      $s3, $s2, $s3           #s3 = 0x7FFF 7FFF
        li      $s0, 0x0001             #s0 = 0x0000 0001
        add     $s3, $s3, $s0           #s3 = 0x7FFF 8000
        li      $s1, 0x1001             #s1 = 0x0000 1001
        sll     $s1, $s1, 16            #s1 = 0x1001 0000
        j       L3                      #skip addi below
        addi    $s3, $s3, 1             #skipped
L3:     sw      $s3, 4($s1)             #store word for checking
        # expected results: dataaddr = 0x1001 0004, writedata = 0x7FFF 8000
```

The above assembly program corresponds to the following machine code:

```
20100010
02108820
0211902a
7e500002
22527fff
22520001
22527ffe
46b37fff
02539024
7e530001
02539022
00129400
02539825
46b00001
02709820
46b11001
```

00118c00

08000013

22730001

ae330004

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):

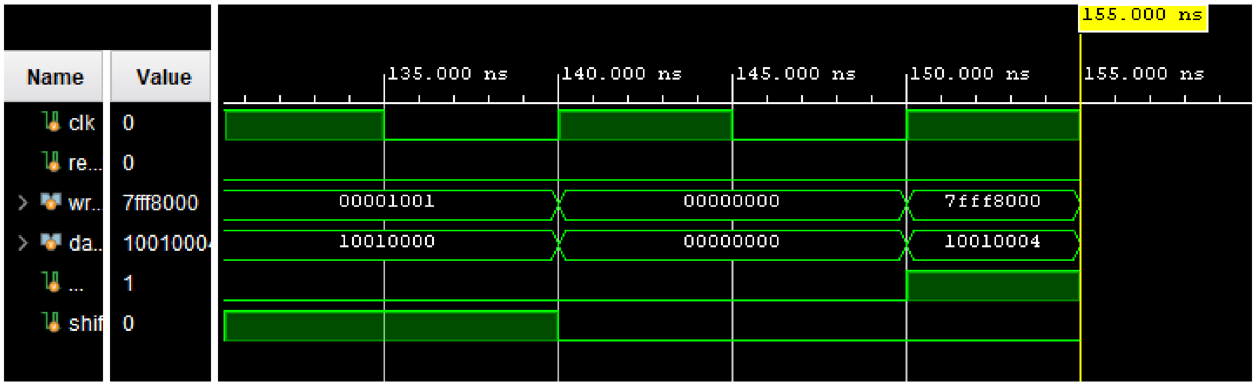


Figure 11. li_testbench.sv simulation waveform diagram.

As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a “Simulation suceeded” print out in Vivado’s TCL console).

Having passed these simulations, we can now certify that the modified MIPS processor can now run **shift left logical (sll), store byte (sb)**, **branch on less than or equal (ble),** and **load immediate (li).**

Custom Instruction: zfr rd, rs, rt

Instruction Format

The **zero from right** or **zfr** custom instruction is an R-type instruction with the following properties:

- Coded in MIPS Assembly as zfr rd, rs, rt.
- Opcode = 000000.
- Funct = 0x33 = 110011.
- The shamt field is a Don't Care field. For our implementation we usually fill them with 10101.
- Operation: [rd] = [rs] AND (0xFFFF_FFFE << [rt][4:0]).
    - This formula was derived after extensive testing.
    - Essentially, we AND [rs] with the bitmask produced by 0xFFFF_FFFE << [rt][4:0] in order to 0 bits [rs][[rt][4:0] : 0] of [rs].

Overview of Modification

- As an R-type instruction, and by the nature of zfr, we found no need to modify the main decoder, since our main decoder already recognizes R-type instructions by the 000000 opcode.
- We instead modify the ALU decoder to distinguish the zfr instruction's funct field 110011 from the R-type instructions and to give it the appropriate control signals. It is assumed that zfr already uses the same main decoder control signals, so the changes in the alu decoder will be in applying a new zfr signal (that controls a mux named zfrmux in the datapath) and giving it an alucontrol that corresponds to the AND operation of the ALU, this is in preparation for receiving both rs and the bitmask in the ALU.
- The bitmask is generated by 0xFFFF_FFFE << [rt][4:0]

Control Signals

The following are the final control signals for this instruction. In yellow are new signals.

Remark: For consistency with the HARRIS implementation, we always take the Don't Care signals to be 0 in the code.

Since this is an R-type instruction, all main decoder control signals are used in the same way as other R-types (aside from the sb caveat, and we also consider ble = 0 and li = 0 here). As for the ALU Decoder control signals, we use alucontrol = 000 for the AND operation, shift = 0 since this is not a shift instruction, and zfr = 1 since this is the zfr instruction.

From Main Decoder:

regwrite	regdst	alusrc	beq	memwrite	memtoreg	jump	aluop	sb	ble	li
1	1	0	0	0	0	1	0	0	0	0

From ALU Decoder:

alucontrol	shift	zfr
000 (and)	0	1

Modifications in the HDL Code

We now explain the HDL modules we modified in order to implement the zfr instruction.

maindec.sv

This is not modified for the zfr instruction.

aludec.sv

The main modification here for the zfr is the addition of a new zfr signal decoded from the funct field of the zfr instruction. This zfr signal will control a new multiplexer in the datapath, zfrmux, that chooses between the regular RD2 source and the zfr-specific bitmask. Note that zfrmux will be inserted in the path from srcbmux to the ALU's operand B input port.

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively.

2-8: Creates a module named aludec with 6-bit logic-type input port (node) named funct with MSB:LSB = 5:0, 2-bit logic-type input port named aluop with MSB:LSB = 1:0, 3-bit logic-type output port named alucontrol with MSB:LSB = 2:0.

Additions here are the logic-type output port named shift (added during the sll implementation for the shift signal) and the logic-type output port named zfr (added during the zfr implementation for the zfr signal). logic-type output port named shift.

The funct port takes in the funct field of the instruction, the aluop port takes in the generated logic simplifier by the main decoder, the bits of the alucontrol controls the operation of the ALU, the shift handles the shift signal, and the zfr handles the zfr signal.

10: Declares a 5-bit logic type variable named controls. Originally a 3-bit variable, but with the addition of the shift and zfr control signals, this is now 5-bit.

11: Wires the controls variable with a concatenation of the alucontrol, shift, and zfr (in that order). Now controls will store the actual control signals generated by the aludec module.

13: The start of an always_comb block. This kinds of block exemplifies the combinational nature of a circuit. This has an implied sensitivity list that watches for changes in all the variables that the block works with. For each change, the contents of the always_comb block will be executed.

14: The start of a case construct (similar to the C switch-case construct). The execution of this case construct will be dictated by the aluop value. As we'll see over and over again, the case construct is a very convenient way of coding for *expected* constant values with predefined behaviors or effects.

15-17: We first look at the aluop to check for the most common operations defined by aluop = 00 → alucontrol = 010 = add, and aluop = 01 → alucontrol = 110 = sub.

The shift signal is going to be 0 for both add and sub dictated by aluop (because of course shift will only be asserted for sll instructions, and maybe in the future we can also assert this for more shift-type instructions).

Similarly, the zfr signal is going to be 0 for both add and sub because it actually desires the AND operation.

If aluop is neither 00 nor 01, then we look at the funct field by default.

18: The case construct that handles the *indefinite case* of the aluop. This handles R-type instructions.

19-26: Each funct case in all of these lines correspond to a specific R-type instruction. For example, funct = 100000 → controls = 010_0 = add. We reach this portion of the code since the default control signals for R-type instructions (which includes sll and zfr) generates an aluop of neither 00 nor 01, so we'll have to look at the funct field.

Within this case construct, we find the added case for the zfr, which corresponds to its funct field of 110011. We decided to give zfr an alucontrol corresponding to the AND operation because we want to use the ALU as a bit masker by taking rs as source A, taking the zfr bit mask as source B, and ANDing them with the ALU.

Once again, as with the sll instruction's shift signal, we have to expand the number of controls bits to fit in the zfr signal.

Also, note that the default case corresponds to when the processor cannot recognize the funct field of the supposedly R-type instruction, so we give don't cares for that (and that is undesired in general).

27: End of the case(funct) construct spanning lines 18-27.

28: End of the case(aluop) construct spanning lines 14-28.

29: End of the always_comb block spanning lines 13-29.

30: End of the aludec module.

controller.sv

This module pertains to the control unit and also houses the pcsrc logic. This contains the main decoder and the ALU decoder.

Here in the controller we declare a new signal zfr as an output logic port in order to wire the zfr signal from aludec towards outside the controller (with the intention of then wiring it to the datapath).

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively.

2-13: Creates a module named controller and declares the input and output ports (also considered as nodes) of the module. Inputs are 6-bit op and funct (fields), and 1-bit zero flag from the ALU in the datapath. Outputs are 1-bit control signals memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, and jump; and 3-bits for the alucontrol.

Port modifications are as follows:

- Output shift signal from sll onwards. From alu decoder, goes to srcamux.
- Output sb signal from sb onwards. From main decoder, goes to dmem module.

- Input sign flag from ble onwards. From datapath's ALU, goes to this controller.
- Output li signal from **li** onwards. From main decoder, goes
- Output zfr signal from zfr onwards. From alu decoder, goes to zfrmux.

16: Declares 2-bit logic-type aluop. Wires aluop from maindec to aludec.

17: Originally declared as logic-type branch but is now beq from beq implementation onwards. Will serve as the control signal for beq.

18: Declares ble from ble implementation onwards. Will serve as control signal for ble.

Note that beq and ble are not declared as ports because they are only internally used by the controller, together with the zero flag and the sign flag (from ble onwards) to determine the pcsrc control signal (i.e. whether to branch or not).

20-22: Instantiates the main decoder as md. Has as input op (the opcode), and as outputs memtoreg, memwrite, beq (originally branch), alusrc, regdst, regwrite, jump, aluop.

Additions are the outputs sb (from sb onwards), ble (from ble onwards), and **li** signals (from li onwards).

Focus on the ble signal which works with beq, zero, and sign to produce the correct pcsrc.

24: Instantiates the alu decoder as ad. Inputs are funct and aluop. Outputs are alucontrol, shift, and zfr, as desired.

Note that shift and zfr are generated in this module because they are R-type instructions, and they can only be distinguished from each other by examining the funct field since they all have the same 000000 opcode (while also conveniently maintaining the proper control signals for R-type instructions as given by the main decoder).

27: Wires pcsrc to the result of branch AND zero, i.e. we branch when the instruction is in fact branch and the contents of the two registers are equal (given by their difference being zero).

Note that for the addition of ble this is expanded to become assign pcsrc = (beq & zero) | (ble & (sign | zero)). In other words, there are now two possibilities for branching: when beq or when ble. This Boolean expression is generated by the simple expression that the processor should:

Branch if the instruction is beq and the difference of [rs] and [rt] is zero (meaning [rs] == [rt]), or branch if the instruction is ble and the difference of [rs] and [rt] is negative (meaning [rs] < [rt]) or is zero (meaning [rs] == [rt]). Essentially read as branch if equal or branch if less than or equal. Note that

$$\text{pcsrc} = \text{beq} \,\&\, \text{zero} \mid (\text{beq} \,\&\, \text{sign} \mid \text{beq} \,\&\, \text{zero}) = \text{beq} \,\&\, \text{zero} \mid (\text{beq} \,\&(\text{sign} \mid \text{zero}))$$

The signal PCSrc controls which next instruction address will be fed to the instruction memory, whether it be for branch (BTA) or in our situation, jump (JTA) is also possible.

28: End of the controller module.

mips.sv

Now we go to the mips module which contains the control unit and the datapath. This is the SCP minus the instruction memory (imem) and the data memory (dmem). In essence, the mips module serves as relay between the controller and the datapath and this allows the two components to communicate.

We modify this module for zfr to wire the zfr signal from the controller to the datapath where it will control a multiplexer named zfrmux which would choose the ultimate source for the operand B of the ALU, either the result of the srcbmux, or the zfr bitmask (we choose the zfr bitmask when the zfr signal is 1).

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively.

2-9: Creates a module named mips with inputs clk (the clock driver), reset (for resetting the PC register), instr (the whole 32-bit instruction), and 32-bit readdata (pertains to result of combinational read of dmem, only actually used for lw). Outputs are pc (from the data path, gives the next PC address, default is PC+4, special cases for branches and jumps), memwrite (from controller, controls whether to write or not to dmem, aluout (because we want to watch aluout for testing), writedata (we also want to watch this).

Additions are the shift output signal (I placed this here because I personally want to watch this).

sb is added as output port from sb onwards. We still want it to go out of this module because it should head to data memory.

12-13: Declares variables memtoreg, alusrc, regdst, regwrite, jump, pcsrc, and zero. Used for wiring between the controller and the datapath, usually controls multiplexers in the datapath, except for the zero flag which is wired from the ALU to the controller to control PCSrc.

14: Declares 2-bit alucontrol, also wired from controller's aludec to the controller.

16-18: Declares the wirings for

- the sign flag (from ble onwards), goes from datapath to controller.
- the **li** signal (from li onwards), goes from controller to datapath.
- and the zfr signal (from zfr onwards), goes from controller to datapath.

20-27: Instantiates a controller module and names it c. It has as inputs 6-bit instr[31:26] (opcode) and instr[5:0] (possibly funct), and the zero flag. Output ports are memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump, and 2-bit alucontrol. Aka the control unit.

Port additions are:

- output shift signal, for controlling srcamux (sll onwards).
- output sb signal, for controlling dmem store (sb onwards).
- input sign flag, for deciding pcsrc (ble onwards).
- output **li** signal, for controlling limux (li onwardsd).
- output zfr signal, for controlling zfrmux (zfr onwards).

Focus on ble and how it goes from datapath ALU to controller maindec.

30-37: Instantiates a datapath module and names it dp. Has as inputs clk, reset, memtoreg pcsrc, alusrc, regdst, regwrite, jump, 2-bit alucontrol, the 32-bit instruction, 32-bit read data chiming in from dmem, and the shift signal for the shift multiplexer.

Port additions are (some may be reiterated):

- input shift signal, for controlling srcamux (sll onwards).
- output sign flag, for deciding pcsrc (ble onwards).
- input **li** signal, for controlling limux (li onwardsd).
- input zfr signal, for controlling zfrmux (zfr onwards).

38: End of the mips module.

datapath.sv

The datapath is where all the main computations that generates register results take place.

For zfr, this module is modified with the insertion of four additional components, all of which works to create the zfr bitmask and wire it towards the zfrmux for its possible selection:

- The zfrmux which chooses between the output of srcbmux and the zfr bitmask. This is controlled by the zfr signal. Note that the zfr bitmask is coded directly into the zfrmux's d1 input.
- A constant 0xFFFF FFFE that is wired to a new left shifter. Motivation:
    - We choose 0xFFFF FFFE as our base bitmask (i.e. the bitmask for when rt[4:0] == 0) in order to fulfill the edge constraints or behaviors of the zfr. Said edge behaviors are given by examples 4 and 5 of the zfr prompt, as follows (brackets for implying register contents omitted):
        - rs = 0xFFFF FFFF, rt = 0x0000 0000, rt[4:0] = 00000 = 0 → rd = 0xFFFF FFFE : the rightmost bit was zeroed as needed (i.e. zeroing bits rt[4:0]=0 to 0 of rs).
        - rs = 0xFFFF FFFF, rt = 0xFFFF FFFF, rt[4:0] = 11111 = 31 → rd = 0x0000 0000 : all bits were zeroed as needed (i.e. zeroing bits rt[4:0] = 31 to 0 of rs).
    - Because with the zfr instruction definition, even with rt[4:0] == 0, we're going to have to *zero* bits rt[4:0] = 0 to 0 as previously shown, and with rt[4:0] == 0, it is easy to see why all bits are to be zeroed.
- A new left shifter that takes in two inputs: the base bitmask 0xFFFF FFFE and the shift amount given by rt[4:0]. The component constantly left shifts 0xFFFF FFFE logically by rt[4:0]. The result is then wired into the d1 input of the zfrmux so that it may be selected as ALU source B whenever the instruction is zfr (zfr signal is asserted). Since in the code the left shifted result is wired directly to the zfrmux, we don't necessarily have a label for it, but for the circuit diagram we refer to it as the bitmask.
    - The left shift component is shown as a circle with << in the datapath.
- A 5-bit selector was also added in order to select the 5 rightmost bits of the contents of rt, giving rt[4:0]. This value shall serve as the shift amount for the left shifter described above.
    - The 5-bit selector component is shown as a square with [4:0] in the datapath.

With these modifications in place and assuming that the zfr signal is asserted (i.e. zfrmux is currently selecting the zfr bitmask), we now turn our attention to the ALU which for the zfr instruction is supposed to produce ALUout = [rs] AND zfr bitmask as the result to be stored in register rd. As such, we expect the ALU to take in srcamux's d0 or RD1 (contents of rs, so shift signal must be 0) as Source A, and for the ALU to be fed the alucontrol = 000 (recall the ALU decoder modification) for the AND operation during zfr instructions.

These datapath modifications then accomplishes the goal of zeroing bits rt[4:0] to 0 of rs and storing the result to rd. While the correctness of our method can be proved via loop invariant if we want to get more technical, due to the small problem space (as given by rt[4:0] = 0 to 31, and sticking to specific rs values) we can also prove this using a much simpler exhaustive approach. I have then written a simple C program that will help us see this through, as follows:

```
#include <stdio.h>
#include <stdint.h>
#define newline printf("\n")

void zfr_prove(int32_t rs){
    int32_t rt, rd;
    int32_t bitmask;
    rt = 0b00000;
    printf("For rs: %x\n", rs);
    while (rt <= 0b11111) {
        bitmask = 0xFFFFFFFE << rt;
        rd = rs & bitmask;
        printf("rt: %d\t rd: %x\n", rt, rd);
        rt++;
    }
    newline;
}

int main() {
    zfr_prove(0xFFFFFFFF);
    zfr_prove(0x00000000);
    zfr_prove(0x0000FFFF);
    zfr_prove(0xC0DEFFFF);
    return 0;
}
```

This C program prints out all the results of rd after a zfr instruction given a specified rs value for our manual visual verification. Unfortunately this program cannot accept a user-defined rt value for taking rt[4:0], instead incrementing rt from 0 to 31, but for our SystemVerilog program we can be assured that we'll always get rt[4:0] with the help of built-in SystemVerilog packed arrays, wherein the rightmost 5-bits of the 32-bit rt contents can be simply accessed using the syntax rt[4:0]. This is reflected as the bit selector in the datapath schematic.

Now for the datapath.sv explanations:

1: Sets the module's delay unit and delay resolution to 1ns and 1ps respectively.

2-15: Creates a module named datapath.

- Has as inputs clk for the clock driver, reset for the reset initializer or just plain reset; control signals memtoreg, pcsrc, alusrc, regdst, regwrite, jump, and 2-bit alucontrol; 32-bit instr which is read from imem via interpreting PC; 32-bit readdata that comes in from dmem for load word; and new signals shift (for sll onwards), li (for li onwards), and zfr (for zfr onwards).
- Has as outputs zero flag, 32-bit pc for the next instruction address (goes to imem), 32-bit aluout tied to dataadr on mips module instantiation of datapath and beyond for testbench watching, 32-bit writedata for the data to be written to the dmem via store instructions, and a new sign flag for branch instructions.

New muxes srcamux, limux, and zfrmux are added/to be added on their respective instructions.

18-25: Declare 5-bit writereg; and 32-bit pcnext, pcnextbr, pcplus4, pcbranch, signimm, signimmsh, RD1, RD2, srca, srcb, liout (new!), result, and zfrout (new!). After some erroneous simulations, I learned the value of first declaring nodes and most importantly their sizes before wiring together the modules using said nodes.

These declarations help guide the wires to connect to the write places, those preceeded with pc deals with the instruction addressing, and the other deal with the main data path.

Changes I did here is to add RD1 and RD2 to correspond to the rs and rt value outputs of the regfile respectively. This is done for clarity and consistency with HARRIS.

28: always_comb block start. As explained previously, this watches all variables that are manipulated inside, and executes on changes. I placed this here to watch the changes in dataaddr and writedata more internally (because for the testbench we simply just use comparisons of dataaddr and writedata actual values to their expected values to verify the correctness of the simulation).

31: Displays the dataaddr and writedata contents whenever there are changes with them. Note that dataaddr is wired to alutout and writedata is wired to RD2.

32: End of the always_comb block above.

35: Instantiates flopr as pcreg, parameter WIDTH=32 with inputs clk, reset, 32-bit pcnext (seen as PC’ in the HARRIS schematic), and output 32-bit pc. This serves as the pc register which updates every clock rising edge. Flopr is used because of the need for a resettable flipflop (reset required usually for initializing the instruction address). Please refer to my Lab Report 9 for line-by-line explanations of the contents of flopr.

Note that pc goes out to imem to specify the next instruction address to be read.

36: Instantiates adder as pcadd1, parameter WIDTH = 32, with inputs pc, 32’b100 (constant 4), and ‘b0 (0 carry in), and outputs pcplus4 (namesake). Constantly generates PC+4 for *consideration*. Please refer to my Lab Report 9 for line-by-line explanations of the contents of adder.

37: Instantiates sl2 as immsh with input signimm (sign-extended instr[15:0]) and output signimmsh. This shifts the input to the left by 2 (multiplies by 4) as the first step to generating BTA. Please refer to my Lab Report 9 for line-by-line explanations of the contents of sl2.

38: Instantiates adder as pcadd2, WIDTH = 32, with addends pcplus4, signimmsh, and 0 for carry in. Output is pcbranch, for consideration. When we say consideration, we imply that the value is being muxed with something else. This completes BTA = PC+4 + (SignImm << 2). Please refer to my Lab Report 9 for line-by-line explanations of the contents of mux2.

39: Instantiates mux2 as pcbrmux, WIDTH = 32, to choose between 0: pcplus4 and 1: pcbranch. Controlled by pcsrc. Result becomes pcnextbr.

40: Instantiates mux2 as pcmux, WIDTH = 32, to choose between pcnextbr and JTA = {(PC + 4)[31:28], addr, 2'b0}. Controlled by jump.

Caveat on this as promised earlier is that we have to realize that there are two multiplexers multiplexing which instruction address to feed to the PC register. The first is the pcbrmux, controlled by pcsrc that is of clear importance to ble. And the next is pcmux, in that order, which with a slight modification now decides whether to branch, or jump, or neither. This order is the reason why if we want to branch, we also need jump to be 0, but if we just want to jump, branch (ble or beq) is a Don't Care, along with the majority of other control signals.

All these complete the choice for the next PC value, whether it be PC+4, BTA, or JTA. Note that we did not made any modifications for this portion, so these explanations will just repeat over and over again, as is.

44: Clarity modification, we assign writedata to RD2. Note that RD2 comes from the register file, and it is wired to WriteData WD of data memory with nothing in between (constantly zero multiplexers in the connection). This is a watched variable.

45-46: Instantiates regfile as rf. Inputs are the clk (for possibly writing sequentially), regwrite signal, instr[25:21] = rs, instr[20:16] = rt, writereg or value to be written, and result coming in from resmux. Outputs are RD1 and RD2. The register file will be written too on clock rising edge if WE3 is asserted by RegWRite.

This regfile is very integral to the processor and as we all know this is like the heart that beats with every instruction we pump in it in line with the synapses of the control unit which is kind of like the pace maker of this system.

47-48: Instantiates mux2 as wrmux with WIDTH = 5 (begause register address is always just 5 bits). This chooses between rt and rd over which to use as the write address for the register file. This is controlled by regdst (register destination). Outputs the address to writereg and is fed back to the register file.

49: This is an **li modification**. For li onwards, a new multiplexer named limux is introduced. It is located somewhere between dmem and resmux in our schematic diagram (for li onwards!). This chooses between aluout and ZeroImm, and it decides using the li signal.

50: Instantiates mux2 with width 32 as resmux. Chooses between the output of ALU, aluout (rather liout from li onwards!), and the data that was combinationally read from dmem, readdata. Controlled by memtoreg. Outputs to result, and that result is fed back to the register file.

50: Instantiates mux2 with width 32 as resmux. Chooses between the output of ALU, aluout, and the data that was combinationally read from dmem, readdata. Controlled by memtoreg. Outputs to result, and that result is fed back to the register file.

51: Instantiates signext as se. Takes in the perceived immediate field instr[15:0] of any instruction and sign-extends the contents to 32-bit, and the result is passed to signimm. Please refer to my Lab Report 9 for line-by-line explanations of the contents of signext.

56: This is an sll modification. Instantiates mux2 as srcamux, 32-bit. Chooses between RD1 and the zero-extended contents of the shamt field of the instruction (instr[10:6]). This is controlled by the shift signal. The result of this is passed to srca which is wired to the ALU.

57: Instantiates mux2 as srcbmux, 32-bit. Chooses between RD2 and signimm (sign-extended immediate). Controlled by alusrc. Output passed to srcb. Originally this connects directly to srcb port of the ALU, but this will change for zfr which introduces a new mux, zfrmux, between that connection. Note that previously RD2 was just written as writedata, but that is not so descriptive so we wired writedata and RD1 to each other.

The motivation for lines 56-57 is that we turned that unused alucontrol assignment to a shift operation within the ALU, to be precise, it is result = b << a. Sa if the instruction is shift, srca of ALU receives the desired shift amount stored in the instruction's shamt field, and srcb receives the value to shift from rt.

58: **This is a zfr modification**. From zfr onwards, we insert a multiplexer aptly named zfrmux which chooses between srcb (from srbmux) and the bitmask for the expected AND operation (given by 32'hFFFF_FFFE << RD2[4:0]).

The result of this is then channeled towarads the SrcB input of the ALU, where if the instruction is zfr (zfr signal is asserted), then it is expected that

59: Instantiates alu as alu. Operand A is srca, and Operand B is aluout (zfrout later on). Controlled by alucontrol. Outputs result to aluout, and also generates zero flag (and sign flag for ble later on).

Per HARRIS, an Arithmetic/Logical Unit (ALU) combines a variety of mathematical and logical operations into a single unit. For example, a typical ALU might perform addition, subtraction, magnitude comparison, AND, and OR operations. In our case, we use a 3-bit control signal alucontrol which specifies 7 default functions or operations (8 if we include the recently added sll instruction for alucontrol 011). The ALU forms the heart of most computer systems, where the alucontrol is like the passive brain signal that directs which artery or multiplexer data source (I know, kind of stretching the analogy here) will the *actual* processed result come from, because most ALU implementations actually computes the respective operation results simultaneously, and the alucontrol just selects which of those results will actually be presented.

The **sign flag** is also now added as an output of the ALU. This signal helps control our branches.

61: Ends the datapath module.

alu.sv

While the ALU plays a significant role in the zfr instruction, serving as the site of the ANDing between the rs contents and the bitmask (assuming zfr instruction is being executed), we did not have to modify it because the ALU can already process AND operations, it just need the right control signals (and of course the desired operands).

Schematic Diagram

We now present the schematic diagram for the zfr instruction modifications.

Again, the modified components are in blue, and the added components are in red.

Here, we modified the controller's ALU decoder, hence the controller is in blue. Said modification is the creation of a new zfr output signal (in red as it is new) for the ALU decoder, which the ALU decoder now recognizes from the added funct case of funct = 110011 which corresponds to the zfr custom R-type instruction. Said case gives out a signal of {alucontrol, shift, zfr} = 000_01, i.e. we will be doing the rs AND bitmask operation as our approach requires, with shift = 0 to select rs as the ALU's operand A, zfr = 1 to select the zfr bitmask as the ALU's operand B, and alucontrol = 000 (as per the HARRIS table of ALU controls, and of course the actual HDL implementation) to make the ALU perform result = A AND B.

For the added components, we have the three new ones that can be grouped as the zfr bitmask generator, and a new one that's simply a multiplexer that we introduce between the srcbmux and the ALU's Source B operand in order to potentially choose the zfr bitmask for when the zfr instruction is being executed (thus it is controlled by a zfr control signal) and pass the result to the line zfrout.

The zfr bitmask generator is composed of a left shifter that takes in a constant 0xFFFF FFFE (our base bitmask), left shifts it by the amount given by the bits rt[4:0], and passes the output to the line called bitmask.

In reality, these are all actually implied by a one-liner in the HDL code given by mux2 #(32) zfrmux(srcb, 32'hFFFF_FFFE << RD2[4:0], zfr, zfrout), but nonetheless these still translate into the components that we've described just now.

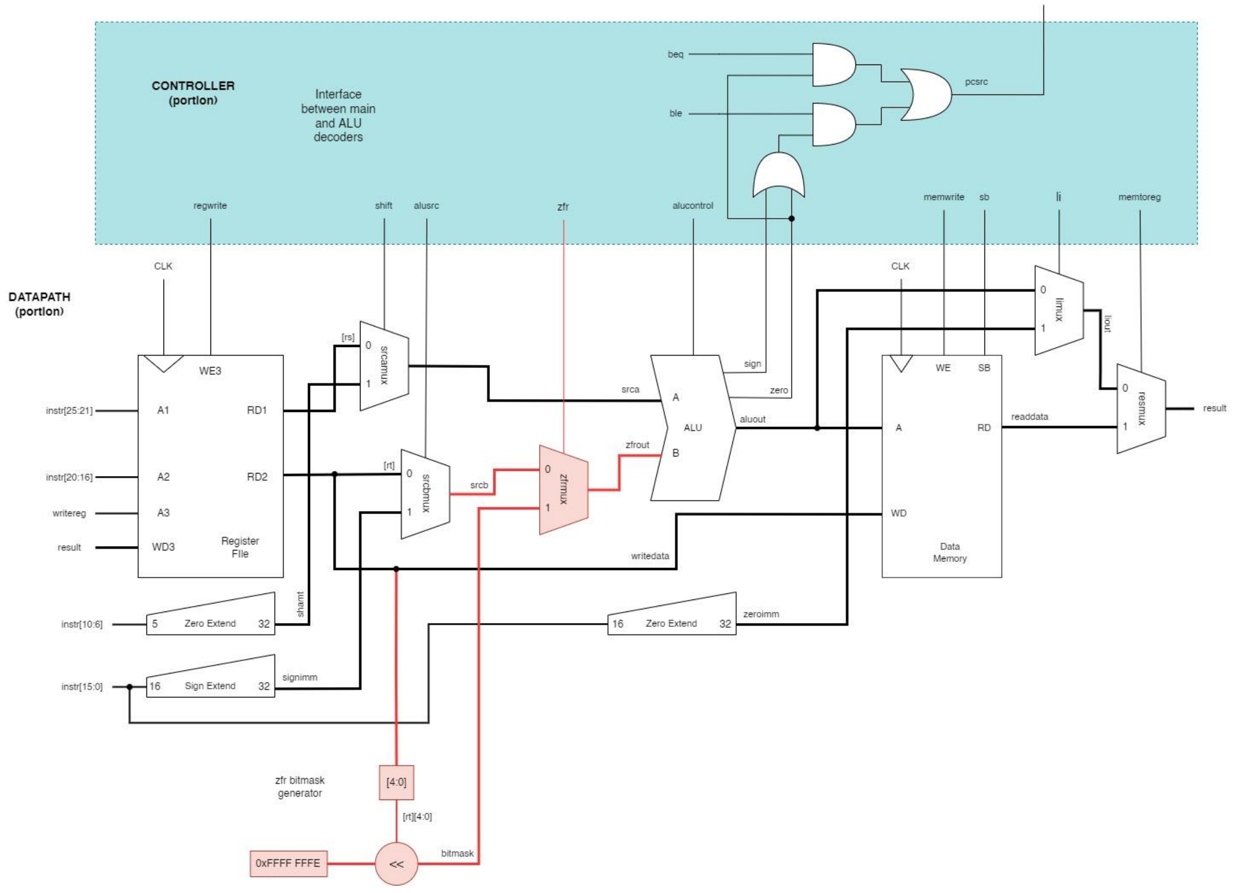


Figure 12. zfr schematic diagram (partial view of the datapath and controller only).

Within the ALU, we also want emphasize the use of the AND operation, so we glimpse into the inside of the ALU defined by the alu.sv module. In orange are the lines and components we want to emphasize, in particular is the AND component that constantly performs A AND B and is chosen as the result when alucontrol[1:0] = 00, hence choosing the data line d00 corresponding to A & B in the operation mux, and when alucontrol[2:0] != 011 (which now corresponds to the sll instruction), hence choosing the data line d0 corresponding to A & B (as passed by the operation mux) in the result mux, which eventually gets passed as the ALU result (to a line called aluout).

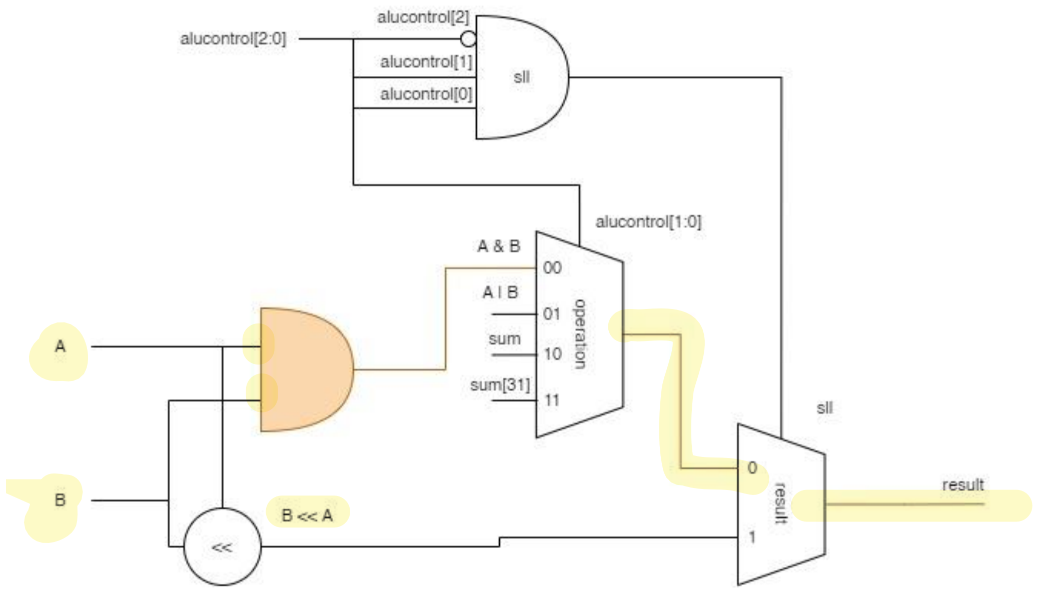


Figure 13. zfr ALU schematics (ommitted sum, condinvb, and flags circuitry).

Testbenches and Test Programs

As the final instruction to be implemented, aside from the repeat integrity tests provided by the previous testbenches and their respective test programs, we create a larger and more complicated test program (and a relatively simple testbench) for this one that would test in an *ad hoc* manner the correctness of our implementation of the zfrout instruction.

The testbenches and their respective test programs used for the zfr instruction are as follows.

testbench.sv

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

11: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite.

14: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

15: Beginning of the initial block.

16: Sets reset to 1 for the first 22 ns of the simulation, then sets it to 0 afterwards.

17: End of this initial block.

20: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

21: Beginning of this always block.

22: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

23: End of the always block.

26: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

27: Start of this always block.

28: An if conditional that executes its block if memwrite is asserted. Corresponds to initializing the actual test when sw or sb is the instruction.

29-35: CHECKING: If dataadr === 84 & writedata === 7, then simulation is successful (print "Simulation suceeded" then stop the program). Else if dataadr !=== 80, simulation failed (print "Simulation failed" then stop the program). The reason for that failure condition is that in the test program, there's a non-terminating store word with effective address 80, so if it happens that dataaddr === 80 when memwrite == 1, then it means that there's a mix-up in the computations.

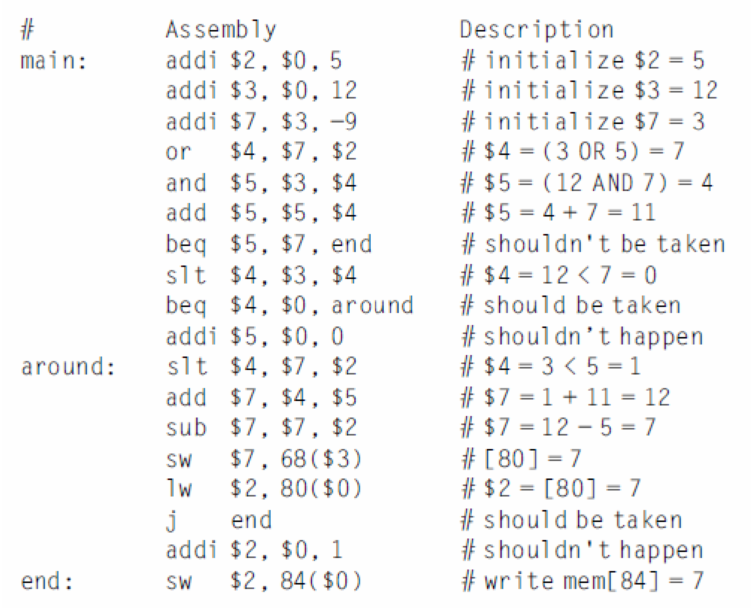
36: End of the memwrite condition block.

37: End of the always block.

38: End of the testbench module.

Assembly Program for testbench.sv

Below is the MIPS program that this testbench will run through. Courtesy of HARRIS.



```
#         Assembly              Description
main:     addi $2, $0, 5        # initialize $2 = 5
          addi $3, $0, 12       # initialize $3 = 12
          addi $7, $3, −9       # initialize $7 = 3
          or   $4, $7, $2       # $4 = (3 OR 5) = 7
          and  $5, $3, $4       # $5 = (12 AND 7) = 4
          add  $5, $5, $4       # $5 = 4 + 7 = 11
          beq  $5, $7, end      # shouldn't be taken
          slt  $4, $3, $4       # $4 = 12 < 7 = 0
          beq  $4, $0, around   # should be taken
          addi $5, $0, 0        # shouldn't happen
around:   slt  $4, $7, $2       # $4 = 3 < 5 = 1
          add  $7, $4, $5       # $7 = 1 + 11 = 12
          sub  $7, $7, $2       # $7 = 12 − 5 = 7
          sw   $7, 68($3)       # [80] = 7
          lw   $2, 80($0)       # $2 = [80] = 7
          j    end              # should be taken
          addi $2, $0, 1        # shouldn't happen
end:      sw   $2, 84($0)       # write mem[84] = 7
```

The machine code below is equivalent to the program above. We fill the memory file memfile.mem with this machine code if we want run this testbench.

20020005

2003000c

2067fff7

00e22025

00642824

00a42820

10a7000a

0064202a

10800001

20050000

00e2202a

00853820

00e23822

ac670044

8c020050

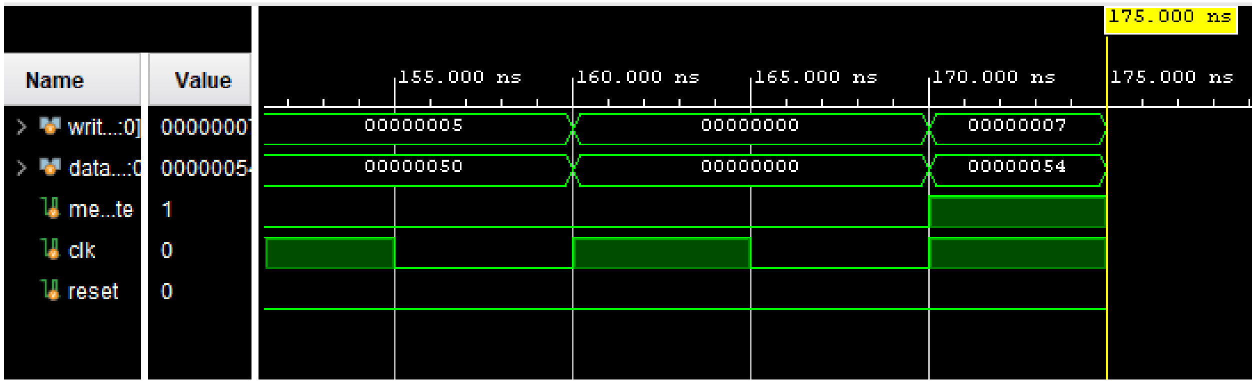
08000011

20020001

ac020054

Line-by-line description of the program is already given by HARRIS.

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a “Simulation suceeded” print out in Vivado’s TCL console).

testbench_2.sv

This is just a slightly modified version of testbench.sv to accommodate the new test program.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named testbench_2. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

11: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite.

14: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

15: Beginning of the initial block.

16: Sets reset to 1 for the first 5 ns of the simulation, then sets it to 0 afterwards (totally arbitrary). Adds a delay of 255 ns that corresponds to the actual runtime of a successful simulation. This explicitly states that reset = 0 for that time frame.

17: End of this initial block.

20: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

21: Beginning of this always block.

22: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

23: End of the always block.

26: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

27: Start of this always block.

28: An if conditional that executes its block if memwrite is asserted. Corresponds to initializing the actual test when sw or sb is the instruction.

Remark: We are now only concerned in getting the Simulation Success Conditions so we leave the Simulation Failed Condition as simply the result of not attaining the simulation success condition.

29-35: CHECKING: If dataadr === 16 & writedata === 0xBBAAB0B0, then simulation is successful (print "Simulation suceeded" then stop the program). Else, simulation failed (print "Simulation failed" then stop the program).

36: End of the memwrite condition block.

37: End of the always block.

38: End of writing the testbench module.

Assembly Program for testbench_2.sv

```
# Write the hex value 0xBBAA0000 to register 1

addi    $1, $0, 0xBBAA  # 2001BBAA

add     $1, $1, $1      # 00210820

add     $1, $1, $1      # ...

add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
add     $1, $1, $1
# Others
addi    $2, $0, 176       # 200200B0
addi    $3, $2, 45056     # 2043B000
addi    $4, $0, 0x7FFF    # 20047fff
add     $4, $4, $4        # 00842020
addi    $4, $4, 1         # 20840001
and     $3, $3, $4        # 00641824
add     $4, $1, $3        # 00232020
addi    $5, $0, 16        # 20050010
sw      $4, 0($5)         # aca40000
```

Explanation:

1: Insert 0x0000 BBAA to register 1.

2-18: Repeatedly add it to itself 16 times in order to emulate a load upper immediate (lui) instruction. After this register 1 contains 0xBBAA 0000.

20: Insert value 176 to register 2.

21: $3 = $2 + 45056 = 176 + 45056.

22: Insert 0x0000 7FFF to $4.

23. Add $4 to itself and store result to $4.

24: Add a 1 to the content of register 4 and store result to register 4.

25: $3 = $3 AND $4.

26: $4 = $1 + $3

27: Store value 16 to $5.

28: Store word content of $4 (0xBBAAB0B0 after all the previous computations) to effective address given by $5 (16).

The above program correspond to the machine code below:

```
2001bbaa

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

00210820

200200b0
```

2043b000

20047fff

00842020

20840001

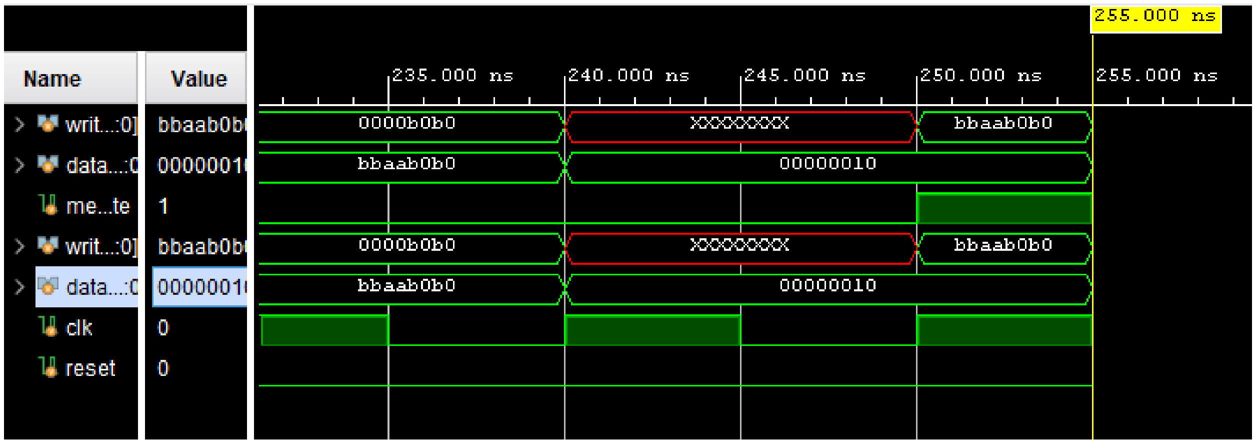
00641824

00232020

20050010

aca40000

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a "Simulation suceeded" print out in Vivado's TCL console).

sll_testbench.sv

The SystemVerilog code for the sll_testbench.sv module is provided below.

```
`timescale 1ns / 1ps

module sll_testbench();


  logic       clk;

  logic       reset;


  logic [31:0] writedata, dataadr;
  logic     memwrite;
  // NOTE, watching shift from mips to top
  logic     shift;

  // instantiate device to be tested
  top dut(clk, reset, writedata, dataadr, memwrite, shift);

  // initialize test
  initial
    begin
      reset <= 1; #1;
      reset <= 0;
      #99 $finish;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  // check results
  always @(negedge clk)
    begin
      if(memwrite) begin
        if(dataadr === 84 & writedata === 'h0C00000C) begin
          $display("Simulation succeeded");
          $stop;
        end else begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
  endmodule
```

This is the main testbench for sll. This is just an appropriation of the previous testbenches.

The explanations for the sll_testbench.sv code above are provided below.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named sll_testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

10: Declared a watcher for the shift signal.

13: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite and also shift.

16: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

17: Beginning of the initial block.

18-20: Sets reset to 1 for the first 1 ns of the simulation, then sets it to 0 afterwards. This initializes the PC register to instruction address 0. After 99 ns end the program, corresponding to actual runtime of program (we could just remove this).

21: End of this initial block.

24: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

25: Beginning of this always block.

26: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

27: End of the always block.

30: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

31: Start of this always block.

32: An if conditional that executes its block if memwrite is asserted. Corresponds to initializing the actual test when sw or sb is the instruction.

Remark: We are now only concerned in getting the Simulation Success Conditions so we leave the Simulation Failed Condition as simply the result of not attaining the simulation success condition.

33-39: CHECKING: If dataadr === 84 & writedata === 0x0C00000C (expected results, see assembly program explanation or annotations), then simulation is successful (print "Simulation suceeded" then stop the program). Else, simulation failed (print "Simulation failed" then stop the program).

40: End of the memwrite condition block.

41: End of the always block.

42: End of writing the testbench module.

Assembly Program for sll_testbench.sv

```
addi $s0, $0, 0x000C

sll  $s0, $s0, 8

sll  $s0, $s0, 8

sll  $s0, $s0, 8

addi $s0, $s0, 0x000C

sw   $s0, 84($0)
```

Explanation:

1: $s0 = 0 + 0x0000 000C = 0x0000 000C

2-3: SLL TESTING: 0x0000 000C should be shifted by 24 bits to the left in total. Result would be 0x0C00 0000.

4: Then add 0x000C to 0x0C00 0000 and store that back to $s0. $s0 should be 0x0C00 000C at this point.

5: Store word contained by $s0 to effective address 84 (may break MARS but for the Vivado simulation it's just fine). $s0 must contain 0x0C00 000C for this simulation to be successful.

The corresponding machine code for this is simply

2010000c

00108200

00108200

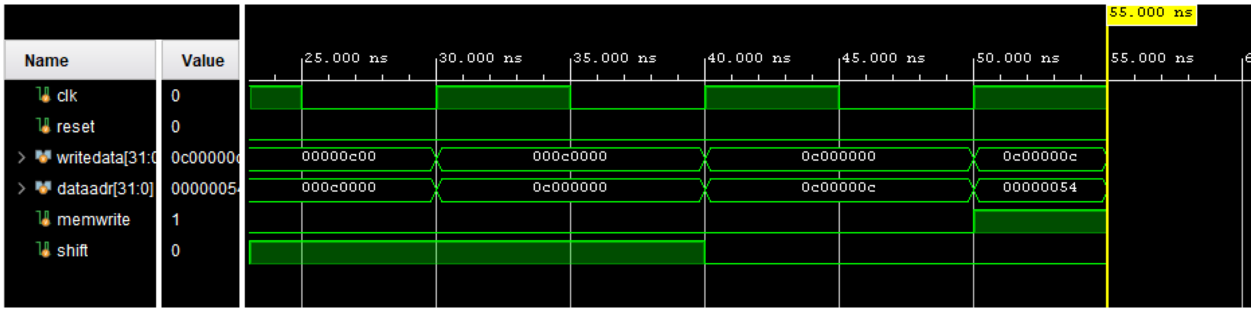
00108200

2210000c

ac100054

Said machine code is as is and did not need any adjustments to work in the context of the memory file and the processor. Moreover, recall that the **rs field** is a don't care for sll; in our case, we simply set the rs field to 0s, but for future instructions we also use variations of bits for the don't cares. Also, by design, we can be assured that the rs field of sll is really treated as a don't care because our shift signal selects between the rs contents and the zero-extended shamt, so it's just either one of the two.

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a "Simulation suceeded" print out in Vivado's TCL console).

sb_testbench.sv

The SystemVerilog code for the sb_testbench.sv module is provided below.

```
`timescale 1ns / 1ps

module sb_testbench();


  logic      clk;

  logic      reset;
  logic [31:0] writedata, dataadr;
  logic        memwrite;
  // NOTE, watching shift from mips to top
  logic        shift;

  // instantiate device to be tested
  top dut(clk, reset, writedata, dataadr, memwrite, shift);

  // initialize test
  initial
    begin
      reset <= 1; #1;
      reset <= 0;
      //#99 $finish;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5; $display("%d", $realtime);
    end

  // check results
  always @(negedge clk)
    begin
      if($realtime == 125) begin
```

```
        if(dataadr === 84 & writedata === 'hFFFF0000) begin
          $display("Simulation succeeded");
          $stop;
        end else begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule
```

The explanations for the sb_testbench.sv code above are provided below.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named sb_testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

10: Declared a watcher for the shift signal.

13: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite and also shift.

16: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

17: Beginning of the initial block.

18-19: Sets reset to 1 for the first 1 ns of the simulation, then sets it to 0 afterwards. This initializes the PC register to instruction address 0.

21: End of this initial block.

24: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

25: Beginning of this always block.

26: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns. And from now on, we also print the current simulation time for verification purposes.

27: End of the always block.

30: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

31: Start of this always block.

32: An if conditional that executes at realtime == 125 ns, and this corresponds with the expected time that the processor reads the final instruction in the memory file.

Remark: We are now only concerned in getting the Simulation Success Conditions so we leave the Simulation Failed Condition as simply the result of not attaining the simulation success condition.

33-39: CHECKING: If dataadr === 84 & writedata === 0xFFFF0000 (expected results), then simulation is successful (print “Simulation suceeded” then stop the program). Else, simulation failed (print “Simulation failed” then stop the program).

40: End of the memwrite condition block.

41: End of the always block.

42: End of testbench module.

Assembly Program for sb_testbench.sv

```
addi    $s0, $0, 0x65FF # s0 = 0x0000 65FF

sw      $s0, 8($0)      # [8] = 0x0000 65FF

sb      $s0, 0($0)      # [0] = 0xFF : [0] = 0xFFXX XXXX

sb      $s0, 1($0)      # [1] = 0xFF : [0] = 0xFFFF XXXX

sb      $s0, 2($0)      # [2] = 0xFF : [0] = 0xFFFF FFXX

sb      $s0, 3($0)      # [3] = 0xFF : [0] = 0xFFFF FFFF

sb      $s0, 4($0)      # [4] = 0xFF : [4] = 0xFFXX XXXX

sb      $s0, 5($0)      # [5] = 0xFF : [4] = 0xFFFF XXXX

sb      $0, 6($0)       # [6] = 0x00 : [4] = 0xFFFF 00XX

sb      $0, 7($0)       # [7] = 0x00 : [4] = 0xFFFF 0000
```

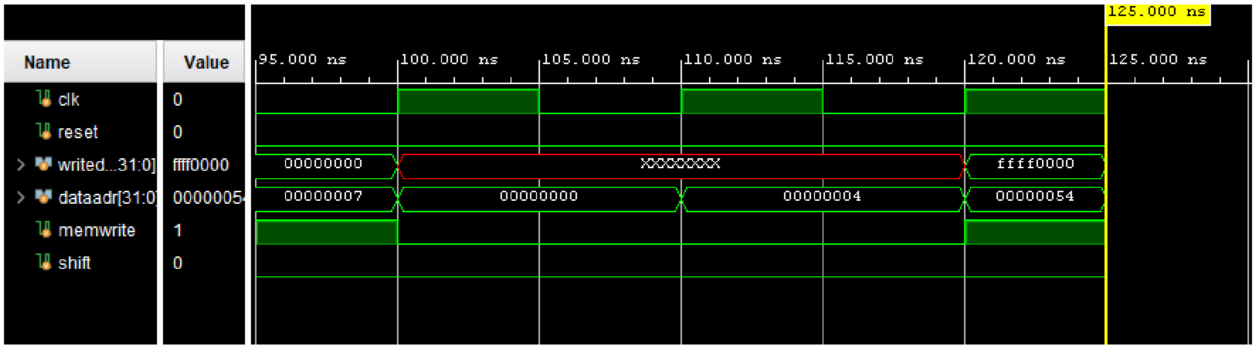
```
lw      $s1, 0($0)      # s1 = [0] = 0xFFFF FFFF
lw      $s2, 4($0)      # s2 = [4] = 0xFFFF 0000
sw      $s2, 84($0)     # [84] = 0xFFFF 0000
# expected results: dataadr === 84 = 0x00000054, writedata === 0xFFFF0000
```

Using this program, we verify if sb is indeed just storing the lowest order byte of wd and if it is being stored in the correct byte offset according to Big Endian as desired. This is a 13 instruction program so for our simulation this ends at 125 ns, at which point we check the sw results.

The above program is equivalent to the machine code below:

```
201065ff
ac100008
a0100000
a0100001
a0100002
a0100003
a0100004
a0100005
a0000006
a0000007
8c110000
8c120004
ac120054
```

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a “Simulation suceeded” print out in Vivado’s TCL console).

ble_testbench.sv

The SystemVerilog code for ble_testbench.sv is shown below.

```
`timescale 1ns / 1ps

module ble_testbench();


 logic       clk;

 logic       reset;


 logic [31:0] writedata, dataadr;

 logic       memwrite;

 // NOTE, watching shift from mips to top

 logic       shift;


 // instantiate device to be tested

 top dut(clk, reset, writedata, dataadr, memwrite, shift);


 // initialize test

 initial

  begin

   reset <= 1; #1;
```

```
    reset <= 0;
    //#99 $finish;
  end

 // generate clock to sequence tests
 always
  begin
    clk <= 1; # 5; clk <= 0; # 5;
    $display("%d", $realtime);
  end

 // check results
 always @(negedge clk)
  begin
    if($realtime == 85) begin
      if(dataadr === 255 & writedata === 251) begin
        $display("Simulation succeeded");
        $stop;
      end else begin
        $display("Simulation failed");
        $stop;
      end
    end
  end
endmodule
```

The explanations for the above ble_testbench.sv code are then shown below.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named sb_testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

10: Declared a watcher for the shift signal.

13: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite and also shift.

16: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

17: Beginning of the initial block.

18-19: Sets reset to 1 for the first 1 ns of the simulation, then sets it to 0 afterwards. This initializes the PC register to instruction address 0.

21: End of this initial block.

24: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

25: Beginning of this always block.

26: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

27: And from now on, we also print the current simulation time for verification purposes. This is also inside the always block.

28: End of the always block.

31: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

32: Start of this always block.

33: An if conditional that executes at realtime == 85 ns, and this corresponds with the expected time that the processor reads the final instruction in the memory file (note that we skip one instruction by branching).

34-40: CHECKING: If dataadr === 255 & writedata === 251 (expected results, see assembly program explanation or annotations), then simulation is successful (print “Simulation suceeded” then stop the program). Else, simulation failed (print “Simulation failed” then stop the program).

41: End of the memwrite condition block.

42: End of the always block.

43: End of testbench module.

Assembly Program for ble_testbench.sv

```
addi    $s0, $0, 62       # s0 = 62

sll     $s0, $s0, 1       # s0 = 62*2 = 124

sll     $s0, $s0, 1       # s0 = 124*2 = 248

addi    $s1, $0, -5       # s1 = -5 = ...1 1111 1011 (2C) = 0xFFFF FFFB

sw      $0,  0($0)        # [0] = 0x0000 0000

sb      $s1, 3($0)        # [0] = 0x0000 FB00

lw      $s2, 0($t0)       # s2 = 251 = ...0 1111 1011 = 0x0000 00FB

ble     $s0, $s2, target  # should be taken since s0 <= s2 : 248 <= 251

addi    $s2, $s2, 5       # should not be executed, would cause s2 = 256

target:

addi    $s2, $s2, 4       # s2 = 255 (expected in aluout = dataadr)

# terminate after 85 sec since only 9 instructions will be executed.

# expected results: dataadr === 255 = 0x000000FF, writedata === 251 = 0x000000FB
```

Note that for ble $s0, $s2, target, we modified it so that the BTA *offset* stored in the imm field corresponds to the actual distance of the labelled address from PC+4. For the next testbenches we will keep doing this whenever we do ble.

The above test program corresponds to the following machine code.

```
2010003e

00108040

00108040

2011fffb

ac000000
```

a0110003

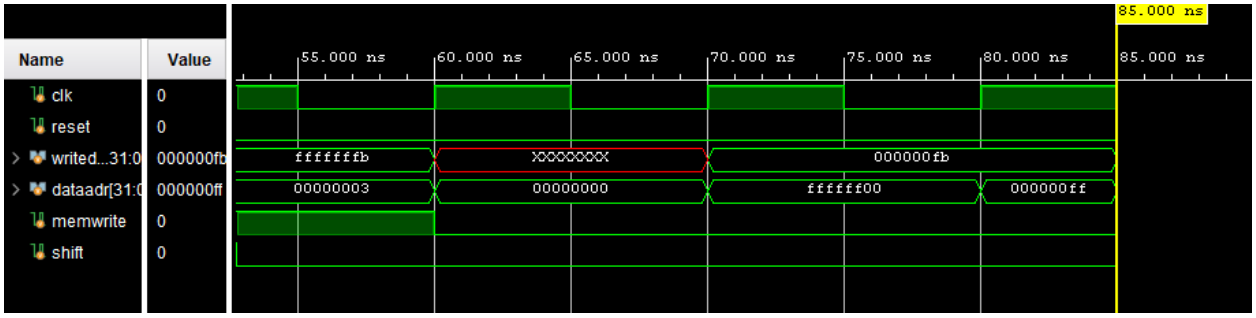
8c120000

7e320001

22520005

22520004

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a “Simulation suceeded” print out in Vivado’s TCL console).

li_testbench.sv

The SystemVerilog code for li_testbench.sv is shown below.

```
`timescale 1ns / 1ps

module li_testbench();


  logic      clk;

  logic      reset;


  logic [31:0] writedata, dataadr;

  logic      memwrite;

  // NOTE, watching shift from mips to top

  logic      shift;


  // instantiate device to be tested

  top dut(clk, reset, writedata, dataadr, memwrite, shift);
```

```
// initialize test
initial
  begin
    reset <= 1; #1;
    reset <= 0;
    //#99 $finish;
  end

// generate clock to sequence tests
always
  begin
    clk <= 1; # 5; clk <= 0; # 5;
    $display("%d", $realtime);
  end

// check results
always @(negedge clk)
  begin
    if(memwrite) begin
      if(dataadr === 32'h1001_0004 & writedata === 32'h7FFF_8000) begin
        $display("Simulation succeeded");
        $stop;
      end else begin
        $display("Simulation failed");
        $stop;
      end
    end
  end
endmodule
```

The explanations for the above li_testbench.sv code are then shown below.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named sb_testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

10: Declared a watcher for the shift signal.

13: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite and also shift.

16: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

17: Beginning of the initial block.

18-19: Sets reset to 1 for the first 1 ns of the simulation, then sets it to 0 afterwards. This initializes the PC register to instruction address 0.

21: End of this initial block.

24: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

25: Beginning of this always block.

26: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

27: And from now on, we also print the current simulation time for verification purposes. This is also inside the always block.

28: End of the always block.

31: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

32: Start of this always block.

33: An if conditional that executes when memwrite is asserted. We have written the test program to execute sw at the end, so the sole execution of this conditional will correspond to the actual checking of the results.

34-40: CHECKING: If dataadr === 32'h1001_0004 & writedata === 32'h7FFF_8000 (expected results, see annotated assembly program), then simulation is successful (print “Simulation suceeded” then stop the program). Else, simulation failed (print “Simulation failed” then stop the program).

41: End of the memwrite condition block.

42: End of the always block.

43: End of testbench module.

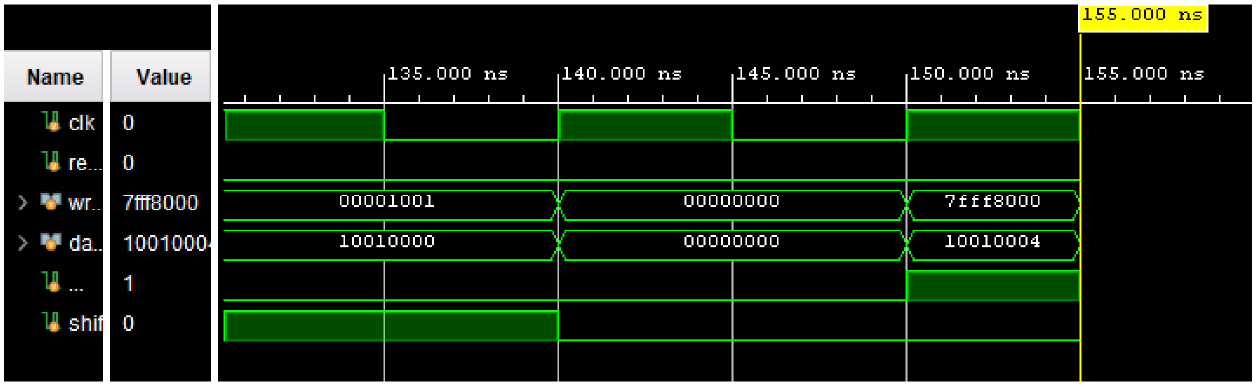
Assembly Program for li_testbench.sv

```
        addi    $s0, $0, 0x0010         #s0 = 0x0010
        add     $s1, $s0, $s0           #s1 = 0x0020
        slt     $s2, $s0, $s1           #s2 = 0x0001
        ble     $s2, $s0, L1            #branch to L1
        addi    $s2, $s2, 0x7FFF        #don't run, s2=8000
        addi    $s2, $s2, 0x0001        #don't run, s2=8001
L1:     addi    $s2, $s2, 0x7FFE        #s2 = 0x7FFF
        li      $s3, 0x7FFF             #s3 = 0x7FFF
        and     $s2, $s2, $s3           #s2 = 0x7FFF
        ble     $s2, $s3, L2            #branch to L2
        sub     $s2, $s2, $s3           #don't run, s2 = 0x0000
L2:     sll     $s2, $s2, 16            #s2 = 0x7FFF 0000
        or      $s3, $s2, $s3           #s3 = 0x7FFF 7FFF
        li      $s0, 0x0001             #s0 = 0x0000 0001
        add     $s3, $s3, $s0           #s3 = 0x7FFF 8000
        li      $s1, 0x1001             #s1 = 0x0000 1001
        sll     $s1, $s1, 16            #s1 = 0x1001 0000
        j       L3                      #skip addi below
        addi    $s3, $s3, 1             #skipped
L3:     sw      $s3, 4($s1)             #store word for checking
        # expected results: dataaddr = 0x1001 0004, writedata = 0x7FFF 8000
```

The above assembly program corresponds to the following machine code:

```
20100010
02108820
0211902a
7e500002
22527fff
22520001
22527ffe
46b37fff
02539024
7e530001
02539022
00129400
02539825
46b00001
02709820
46b11001
00118c00
08000013
22730001
ae330004
```

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):



As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a “Simulation suceeded” print out in Vivado’s TCL console).

zfr_testbench.sv (new!)

The SystemVerilog code for the zfr_testbench.sv module is shown below.

```
`timescale 1ns / 1ps

module zfr_testbench();


 logic      clk;

 logic      reset;


 logic [31:0] writedata, dataadr;

 logic      memwrite;

 // NOTE, watching shift from mips to top

 logic      shift;


 // instantiate device to be tested

 top dut(clk, reset, writedata, dataadr, memwrite, shift);


 // initialize test
```

```
initial
 begin
  reset <= 1; #1;
  reset <= 0;
  //#99 $finish;
 end

// generate clock to sequence tests
always
 begin
  clk <= 1; # 5; clk <= 0; # 5;
  $display("%d", $realtime);
 end

// check results
always @(negedge clk)
 begin
  if($realtime == 365) begin
   if(dataadr === 32'h4618_FE7E & writedata === 32'h0000_0040) begin
    $display("Simulation succeeded");
    $stop;
   end else begin
    $display("Simulation failed");
    $stop;
   end
  end
 end
```

endmodule

The explanations for the above code now follow.

1: Sets the modules delay unit and delay resolution to 1ns and 1ps respectively.

2: Creates a module named sb_testbench. Note that this is declared without ports, as is typical for testbenches.

4: Declares clk to handle the clock signal. Recall remark on declaration of variables (assumed to be logic type and 1-bit, unless otherwise specified).

5: Declares reset to handle the reset signal.

7: Declares 32-bit writedata and dataadr nodes. These watches values coming from inside the DUT: writedata is now tied to RD2 and dataadr is tied to aluout within the datapath (assumed to be computed effective address), respectively.

8: Declares memwrite. Watches memwrite signal that is only asserted by either sw and sb.

10: Declared a watcher for the shift signal.

13: Instantiates the top module (literally the top-level of the processor) and considers it the DUT. Input ports are clk and reset. Output ports are writedata, dataadr and memwrite and also shift.

16: Signals the start of an initial block. This is used to initialize the test by momentarily setting reset to 0 at the beginning. This also serves the additional purpose of verifying whether the processor and the reading of instructions respond correctly to the reset signal.

17: Beginning of the initial block.

18-19: Sets reset to 1 for the first 1 ns of the simulation, then sets it to 0 afterwards. This initializes the PC register to instruction address 0.

21: End of this initial block.

24: Signals the start of an always block with empty sensitivity list. This block is the clock driver.

25: Beginning of this always block.

26: Gives the clock a period of 10 ns, with high duration of 5 ns and low duration of 5 ns.

27: And from now on, we also print the current simulation time for verification purposes. This is also inside the always block.

28: End of the always block.

31: Signals the start of an always block which is executed at every negative clock edge (sensitive to negedge clk). This always block serves to check the results.

32: Start of this always block.

33: In this line we find an if conditional that executes at $realtime == 365. This corresponds to the 37 instructions that the MIPS program will actually execute (some instructions are skipped by a ble instruction), and we check for the expected results at $realtime == 365, i.e. in the middle of clock cycle 37, because if we check at the end of the clock cycle we might see undefined values due to the program execution already reaching an undefined portion of the memory file.

34-40: CHECKING: If dataadr === 32'h4618_FE7E & writedata === 32'h0000_0040 (expected results, see annotated assembly program), then simulation is successful (print “Simulation suceeded” then stop the program). Else, simulation failed (print “Simulation failed” then stop the program).

41: End of the memwrite condition block.

42: End of the always block.

43: End of testbench module.

Assembly Program for zfr_testbench.sv

The MIPS program below is the test program for zfr_testbench.sv. Note that the program is inspired by the zfr examples in the zfr prompt, with a checking subroutine appended at lines 33-43. Also note that the program also tests all the other recently added instructions except for sb.

```
li      $s0, 0xFFFF     # s0 = 0000 FFFF      46b0ffff
sll     $s1, $s0, 16    # s1 = FFFF 0000
or      $s2, $s0, $s1   # s2 = FFFF FFFF
li      $s3, 5          # s3 = 0000 0005      46b30005
#zfr    $s4, $s2, $s3   # s4 = FFFF FFC0      0253a573
sw      $s4, 0($0)      # [0] = FFFF FFC0

li      $s0, 0xC0DE     # s0 = 0000 C0DE
sll     $s1, $s0, 16    # s1 = C0DE 0000
li      $s0, 0xFFFF     # s0 = 0000 FFFF
add     $s2, $s0, $s1   # s2 = C0DE FFFF
li      $s3, 5          # s3 = 0000 0005
#zfr    $s4, $s2, $s3   # s4 = C0DE FFC0
sw      $s4, 4($0)      # [4] = C0DE FFC0

li      $t0, 0xBABE     # t0 = 0000 BABE
```

```
        sll     $t1, $t0, 16        # t1 = BABE 0000
        addi    $t1, $t1, 5         # t1 = BABE 0005
        #zfr    $t2, $s2, $t1       # t2 = C0DE FFC0
        ble     $t1, $t2, target
        addi    $t2, $t2, 256
target: sw      $t2, 8($0)          # [8] = C0DE FFC0

        li      $s0, 0xFFFF         # s0 = 0000 FFFF
        sll     $s1, $s0, 16        # s1 = FFFF 0000
        or      $s2, $s0, $s1       # s2 = FFFF FFFF
        #zfr    $s3, $s2, $0        # s3 = FFFF FFFE
        sw      $s3, 12($0)         # [12] = FFFF FFFE
        #zfr    $s3, $s2, $s2       # s3 = 0000 0000
        sw      $s3, 16($0)         # [16] = 0000 0000

        #checking
        lw      $s0, 0($0)          # s0 = FFFF FFC0
        lw      $s1, 4($0)          # s1 = C0DE FFC0
        and     $t0, $s0, $s1       # t0 = C0DE FFC0
        lw      $s2, 8($0)          # s2 = C0DE FFC0
        lw      $s3, 12($0)         # s3 = FFFF FFFE
        sll     $s2, $s2, 3         # s2 = 06F7 FE00
        sub     $t1, $s3, $t0       # t1 = 3F21 003E
        add     $t2, $s2, $t1       # t2 = 4618 FE3E
        lw      $s4, 16($0)         # s4 = 0000 0000
        addi    $s4, $s4, 64        # s4 = 0000 0040
        add     $t3, $t2, $s4       # t3 = 4618 FE7E
```

# at realtime 365 ns, terminate

# expected results: dataadr = 32'h4618_FE7E, writedata = 32'h0000_0040

Observe keenly how at lines 33-43 of the assembly program we load all the memory locations where we saved the results corresponding to the zfr examples into registers that we all operated on with respect to each other, and all these instructions build up to a final result that can only be correct if ALL the previous instructions executed correctly.

With these we can be certain that our expected result should be the result of the simulation.

The above MIPS test program for zfr_testbench.sv is equivalent to the machine code below:

```
46b0ffff

00108c00

02119025

46b30005

0253a573

ac140000

46b0c0de

00108c00

46b0ffff

02119020

46b30005

0253a573

ac140004

4508babe

00084c00

21290005

02495573

7d2a0001

214a0100

ac0a0008

47f0ffff
```

00108c00

02119025

02409d73

ac13000c

02529d73

ac130010

8c100000

8c110004

02114024

8c120008

8c13000c

001290c0

02684822

02495020

8c140010

22940040

01545820

Remark: As said in an earlier remark, we use random bits for the Don't Care fields of sll, li, and zfr for stronger verification. This decision is very apparent on the machine code instructions above.

Running the testbench with a memory file that contains the above sequence of machine code instructions produced the following waveform diagram (only the last few cycles are shown):

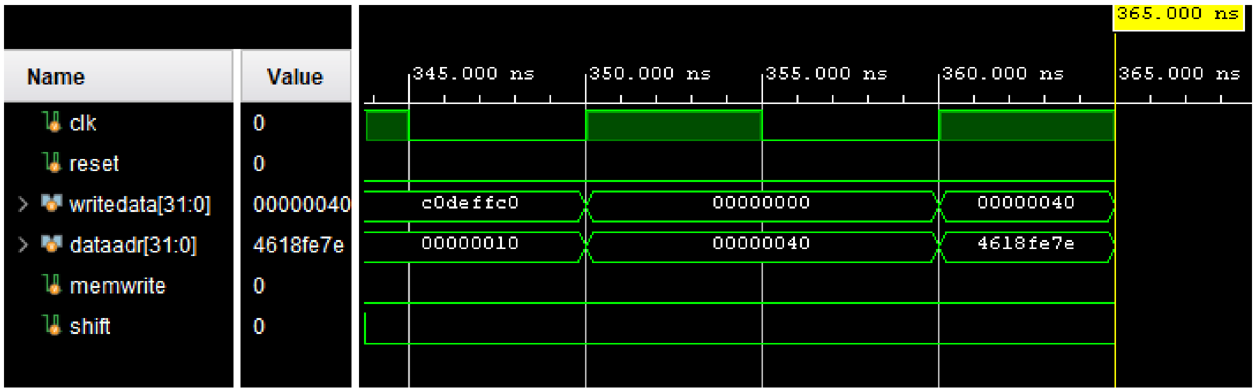


Figure 14. zfr_testbench.sv simulation waveform diagram.

As can be seen at the final half-cycle, we have gotten our expected results, hence the simulation has succeeded (accompanied by a “Simulation suceeded” print out in Vivado’s TCL console).

Having passed these series of simulations, we can now certify our extended MIPS processor to be capable of handling the **sll, sb, ble, li, and zfr instructions**.

// END OF PROJECT 02 DOCUMENTATION //

Link to Video Documentation

Attributions