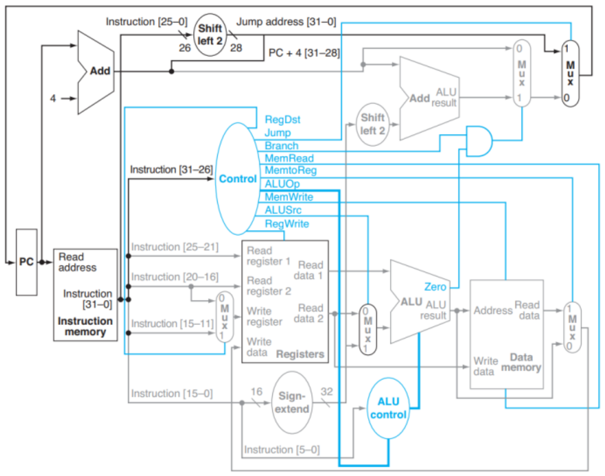
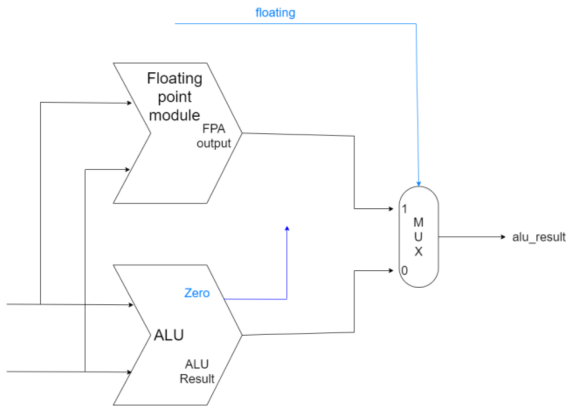
Single Cycle Implementation of MIPS Architecture





A project by

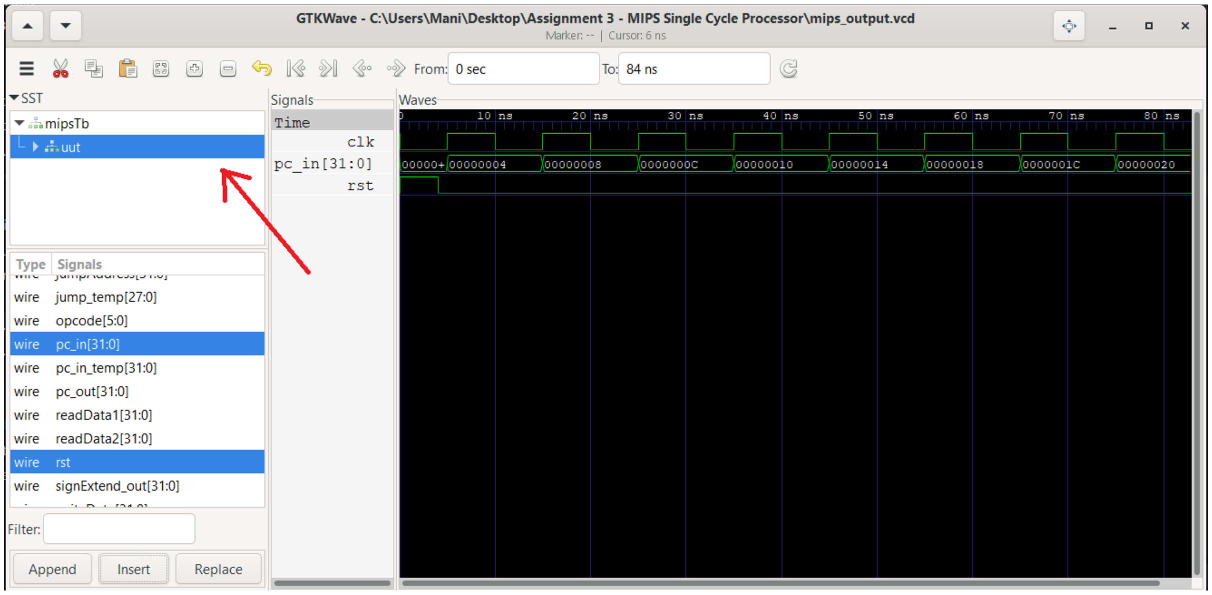
Manikandan Gunaseelan

Table of Contents

Simulation of the Design

To run the processor, one must simulate the “mipsTb” module. Do note that this module only has clk and rst as inputs as all of the other signals are internal. So to view the internal signals, we have to go to the “uut” section in the waveform window and then select the signals we wish to see.

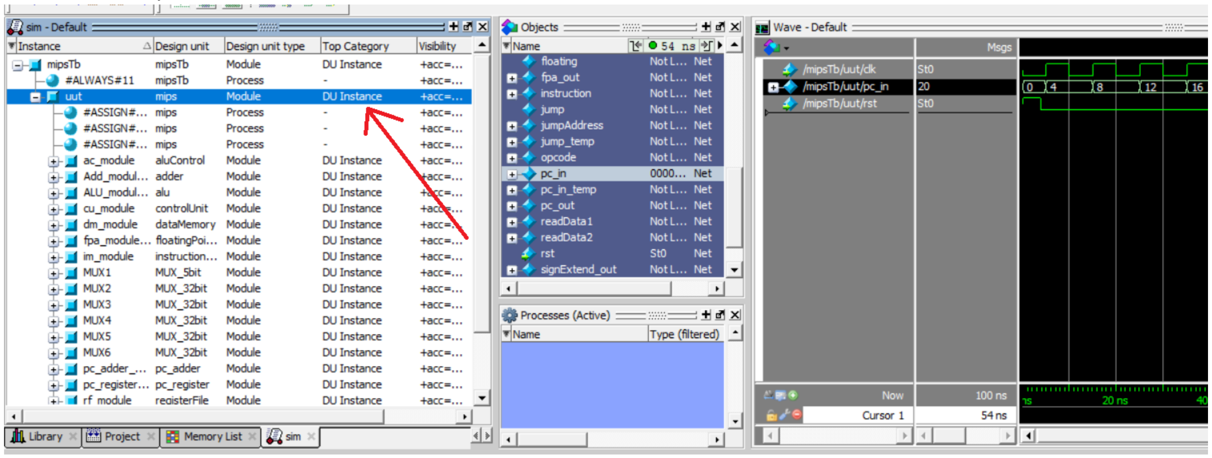
On GTKWave, this can be done here -


GTKWave - C:\Users\Mani\Desktop\Assignment 3 - MIPS Single Cycle Processor\mips_output.vcd
Marker: -- | Cursor: 6 ns
From: 0 sec
To: 84 ns
SST
mipsTb
uut
Signals
Waves
Time
clk
pc_in[31:0]
rst
Filter:
Append
Insert
Replace

Type	Signals
wire	jump_temp[27:0]
wire	opcode[5:0]
wire	pc_in[31:0]
wire	pc_in_temp[31:0]
wire	pc_out[31:0]
wire	readData1[31:0]
wire	readData2[31:0]
wire	rst
wire	signExtend_out[31:0]

```
00000+ 00000004 00000008 0000000C 00000010 00000014 00000018 0000001C 00000020
```

And on ModelSim, it can be done here -



Instance	Design unit	Design unit type	Top Category	Visibility
mipsTb	mipsTb	Module	DU Instance	+acc=...
#ALWAYS#11	mipsTb	Process	-	+acc=...
uut	mips	Module	DU Instance	+acc=...
#ASSIGN#...	mips	Process	-	+acc=...
#ASSIGN#...	mips	Process	-	+acc=...
#ASSIGN#...	mips	Process	-	+acc=...
ac_module	aluControl	Module	DU Instance	+acc=...
Add_modul...	adder	Module	DU Instance	+acc=...
ALU_modul...	alu	Module	DU Instance	+acc=...
cu_module	controlUnit	Module	DU Instance	+acc=...
dm_module	dataMemory	Module	DU Instance	+acc=...
fpa_module...	floatingPoi...	Module	DU Instance	+acc=...
im_module	instruction...	Module	DU Instance	+acc=...
MUX1	MUX_5bit	Module	DU Instance	+acc=...
MUX2	MUX_32bit	Module	DU Instance	+acc=...
MUX3	MUX_32bit	Module	DU Instance	+acc=...
MUX4	MUX_32bit	Module	DU Instance	+acc=...
MUX5	MUX_32bit	Module	DU Instance	+acc=...
MUX6	MUX_32bit	Module	DU Instance	+acc=...
pc_adder_...	pc_adder	Module	DU Instance	+acc=...
pc_register...	pc_register	Module	DU Instance	+acc=...
rf module	registerFile	Module	DU Instance	+acc=...

Name		
floating	Not L...	Net
fpa_out	Not L...	Net
instruction	Not L...	Net
jump	Not L...	Net
jumpAddress	Not L...	Net
jump_temp	Not L...	Net
opcode	Not L...	Net
pc_in	0000...	Net
pc_in_temp	Not L...	Net
pc_out	Not L...	Net
readData1	Not L...	Net
readData2	Not L...	Net
rst	St0	Net
signExtend_out	Not L...	Net

	Msgs
/mipsTb/uut/clk	St0
/mipsTb/uut/pc_in	20
/mipsTb/uut/rst	St0
Now	100 ns
Cursor 1	54 ns

In this report, I will be showing the outputs on GTKWave as it is easier to run and view. However, the same outputs are also obtained on ModelSim.

Instructions Implemented

The following instructions have been implemented in the processor -

1. R - type instructions
    - add
    - sub
    - AND
    - OR
    - slt

2. addi instruction

3. lw instruction

4. sw instruction

5. beq instruction

6. jump instruction

7. floating point arithmetic

Different programs have been written to demonstrate all of these instructions separately and I will now show the outputs of each of them, before moving on to the actual design of the processor.

Demonstration of the Processor

In this section, different outputs for various programs will be shown. For each of these, the program has to be written in the instruction memory (instructionMemory.v file). The programs demonstrated have been given as comments in the file.

As mentioned previously, I have used GTKWave to show the results in the report as it is easier to simulate but the same results can be seen on ModelSim too.

There are five programs that are simulated. Each of them show different instructions implemented by the processor. The programs are

1. Basic register arithmetic demonstration
2. Load word and store word demonstration
3. Beq instruction demonstration
4. Jump instruction demonstration
5. Floating point arithmetic demonstration

The different control signals, inputs and outputs to the internal modules required for each instruction have been highlighted separately.

The signal names follow the notation given in the diagram of the Patterson and Hennessy book, the image of which is also shown in the cover page.

Note - We have to load the instruction memory from instructionMem[1] and not [0] as instructionMem[1] is the first instruction that the PC reads.

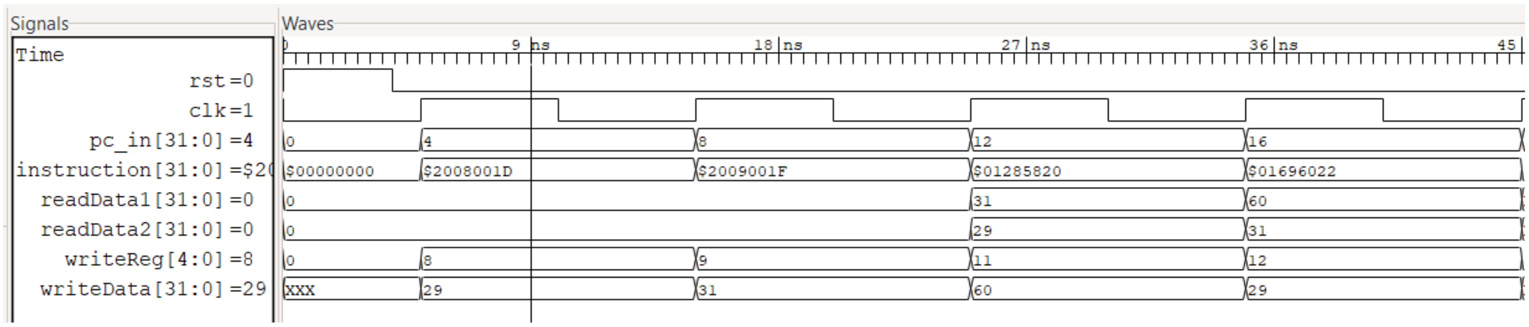
1. Basic Register Arithmetic (R-type instructions)

The program is as follows -

```
addi $t0, $zero, 29
addi $t1, $zero, 31
add $t3, $t1, $t0
sub $t4, $t3, $t1
```

The instruction memory is loaded

```
instructionMem[1] = {6'd8, 5'd0, 5'd8, 16'd29};                  // addi $t0, $zero, 29
instructionMem[2] = {6'd8, 5'd0, 5'd9, 16'd31};                  // addi $t1, $zero, 31
instructionMem[3] = {6'd0, 5'd9, 5'd8, 5'd11, 5'd0, 6'd32};      // add $t3, $t1, $t0
instructionMem[4] = {6'd0, 5'd11, 5'd9, 5'd12, 5'd0, 6'd34};     // sub $t4, $t3, $t1
```



In the register file, $t0 starts from register number 8.

Here, we can see that in the first cycle, 29 is being written into register number 8, ie. $t0.

In the second cycle, 31 is being written into $t1 (register number 9).

In the third cycle, 31 and 29 are read from $t1 and $t0 respectively and their sum, 60, is being written into register number 11, that is $t3.

In the final cycle, 60 and 31 are read from $t3 and $t1, and their subtraction, 29, is being written to register number 12, which is $t4.

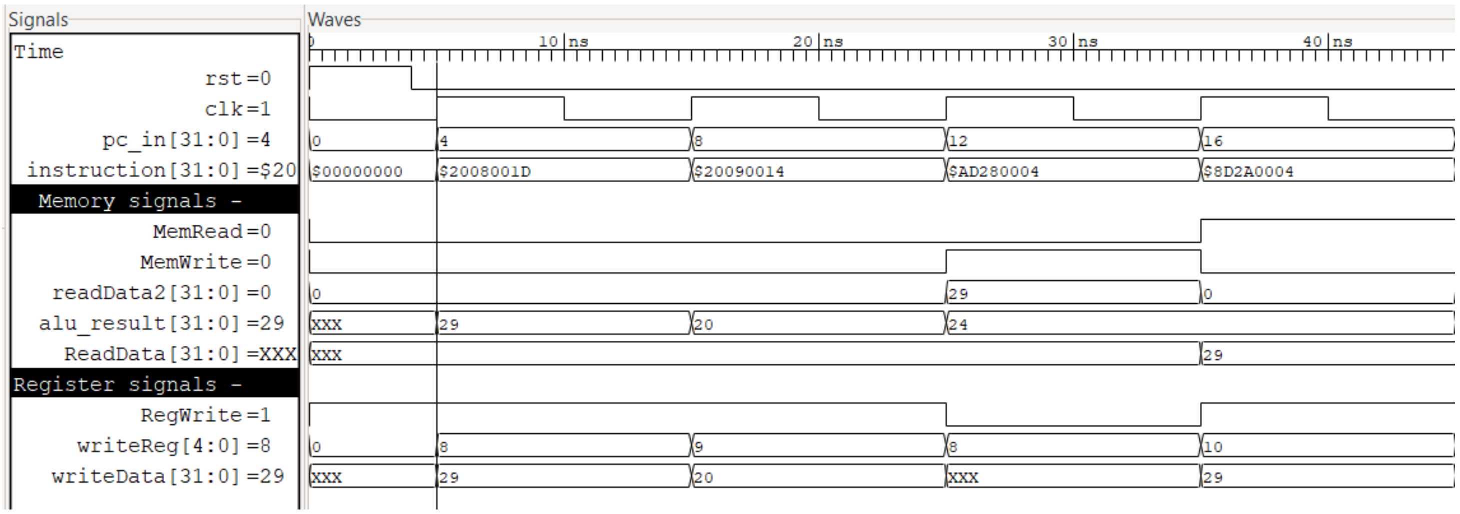
Hence, R-type instructions work as expected. Instructions like AND, OR, slt can also be implemented similarly.

2. lw and sw instructions

```
addi $t0, $zero, 29
addi $t1, $zero, 20          (Address)
sw $t0, 4($t1)               (4 + 20 = 24 will be the memory address)
lw $t2, 4($t1)
```

The instruction memory has

```
instructionMem[1] = {6'd8, 5'd0, 5'd8, 16'd29};      // addi $t0, $zero, 29
instructionMem[2] = {6'd8, 5'd0, 5'd9, 16'd20};      // addi $t1, $zero, 20
instructionMem[3] = {6'd43, 5'd9, 5'd8, 16'd4};      // sw $t0, 4($t1)
instructionMem[4] = {6'd35, 5'd9, 5'd10, 16'd4};     // lw $t2, 4($t1)
```



Here, in the first and second clock cycles, we focus on the register signals. 29 is written to $t0 (register 8) and 20 is written to $t1 (register 9).

Then in the third cycle, MemWrite becomes 1. alu_result points to the address in the memory (24 = 4 offset from $t1) and readData2 will be the data written in the memory (29).

In the fourth cycle, MemRead becomes 1. The data 29 is read from memory location 24 and written to register number 10, which is $t2 (RegWrite becomes 1).

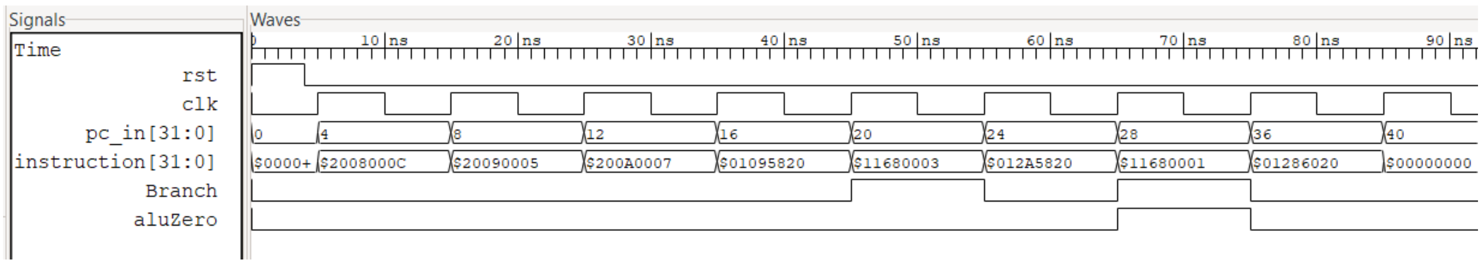
Hence, lw and sw work correctly.

3. beq instruction

```
addi $t0, $zero, 12
addi $t1, $zero, 5
addi $t2, $zero, 7
add $t3, $t0, $t1
beq $t3, $t0, +3          (This branch will not be taken)
add $t3, $t1, $t2
beq $t3, $t0, +1          (This branch will be taken – change can be seen in PC)
sub $t0, $t3, $t0         (This instruction will be skipped)
add $t4, $t1, $t0
```

Instruction memory contents are

```
instructionMem[1] = {6'd8, 5'd0, 5'd8, 16'd12};               // addi $t0, $zero, 12
instructionMem[2] = {6'd8, 5'd0, 5'd9, 16'd5};                 // addi $t1, $zero, 5
instructionMem[3] = {6'd8, 5'd0, 5'd10, 16'd7};               // addi $t2, $zero, 7
instructionMem[4] = {6'd0, 5'd8, 5'd9, 5'd11, 5'd0, 6'd32};   // add $t3, $t0, $t1
instructionMem[5] = {6'd4, 5'd11, 5'd8, 16'd3};               // beq $t3, $t0, +3
instructionMem[6] = {6'd0, 5'd9, 5'd10, 5'd11, 5'd0, 6'd32};  // add $t3, $t1, $t2
instructionMem[7] = {6'd4, 5'd11, 5'd8, 16'd1};               // beq $t3, $t0, +1
instructionMem[8] = {6'd0, 5'd8, 5'd11, 5'd8, 5'd0, 6'd34};   // sub $t0, $t3, $t0
instructionMem[9] = {6'd0, 5'd9, 5'd8, 5'd12, 5'd0, 6'd32};   // add $t4, $t1, $t0
```



We have covered add, sub & addi instructions before so we will focus only on branch.

As we can see, Branch signal goes high on instruction 5 and 7. However, aluZero is only active in instruction 7, so that is when the branch is taken. We can confirm the branch is taken by looking at the PC value, which increases by 8 to go from 28 to 36 instead of 32.

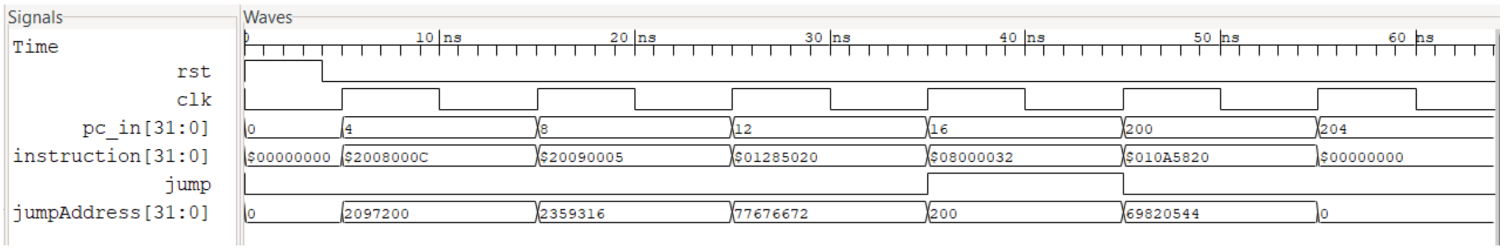
Hence, beq instruction works correctly.

4. jump instruction

```
addi $t0, $zero, 12
addi $t1, $zero, 5
add $t2, $t1, $t0
j 50
sub $t4, $t3, $t2                  (This will be skipped)
add $t4, $t1, $t0                  (This will also be skipped)
…
add $t3, $t0, $t2            (This is written at the 50th location in the instruction memory)
```

Instruction memory contents are as follows

```
instructionMem[1] = {6'd8, 5'd0, 5'd8, 16'd12};                  // addi $t0, $zero, 12
instructionMem[2] = {6'd8, 5'd0, 5'd9, 16'd5};                   // addi $t1, $zero, 5
instructionMem[3] = {6'd0, 5'd9, 5'd8, 5'd10, 5'd0, 6'd32};     //  add $t2, $t1, $t0
instructionMem[4] = {6'd2, 26'd50};                              //  j 50
instructionMem[5] = {6'd0, 5'd11, 5'd10, 5'd12, 5'd0, 6'd34};    // sub $t4, $t3, $t2
instructionMem[6] = {6'd0, 5'd9, 5'd8, 5'd12, 5'd0, 6'd32};     // add $t4, $t1, $t0
instructionMem[50] = {6'd0, 5'd8, 5'd10, 5'd11, 5'd0, 6'd32};   // add $t3, $t0, $t2
```



```
pc_in[31:0]        0          4          8          12         16         200        204
instruction[31:0]  $00000000  $2008000C  $20090005  $01285020  $08000032  $010A5820  $00000000
jumpAddress[31:0]  0          2097200    2359316    77676672   200        69820544   0
```

We can see the first three addi, addi & add instructions. Then in the fourth instruction, jump signal is asserted. The jump address can be seen to be 200 at this point (50 shifted left by 2).

Then we also see PC's next value as 200, confirming that the jump did happen.

Hence, we can see that the jump instruction works.

5. floating point arithmetic instructions

Note – In this implementation, a new opcode has been used for floating point instructions (opcode 110011). This adds two 32 bit binary numbers in IEEE754 standard of representation and also returns the result in IEEE754 format. The code from the previous assignment of floating point adder (floatingPointAdder.v) is used.

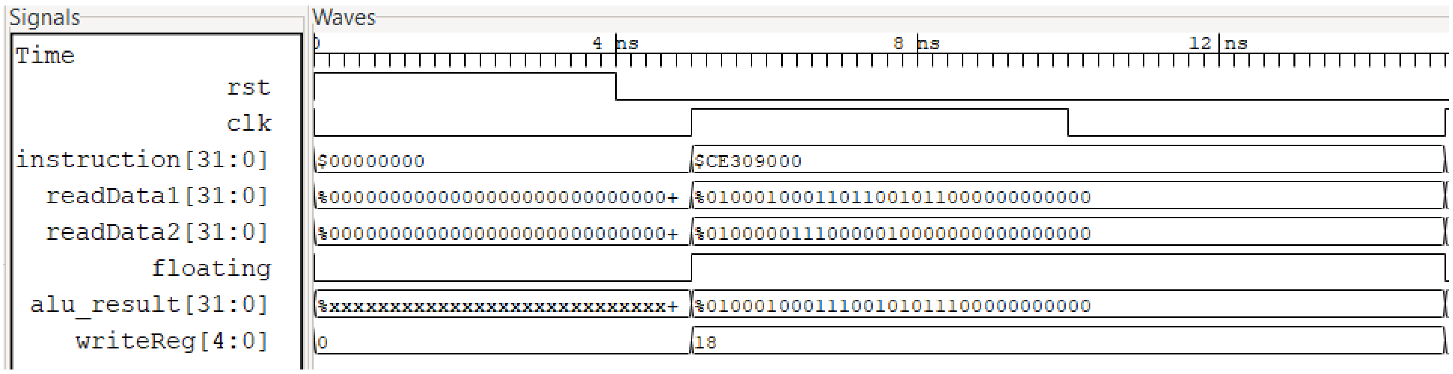
** The registers $s0 and $s1 are pre-loaded with values –

```
registerMem[16] = 32'b01000001110000010000000000000000;        // 24.125 in $s0
registerMem[17] = 32'b01000100011011001011000000000000;        // 946.75 in $s1
```

fpa $s2, $s1, $s0

The instruction memory contents are

```
instructionMem[1] = {6'b110011, 5'd17, 5'd16, 5'd18, 5'd0, 6'd0};       //  fpa $s2, $s1, $s0
```



As we can see, the ‘floating’ control signal is asserted, which selects the output from the floating point module instead of the normal ALU. Floating point arithmetic is performed and the result obtained is 01000100011100101011100000000000, which is the IEEE754 format of representation for 970.875, which is the correct sum.

Hence, we can conclude that floating point instructions also work on the processor.

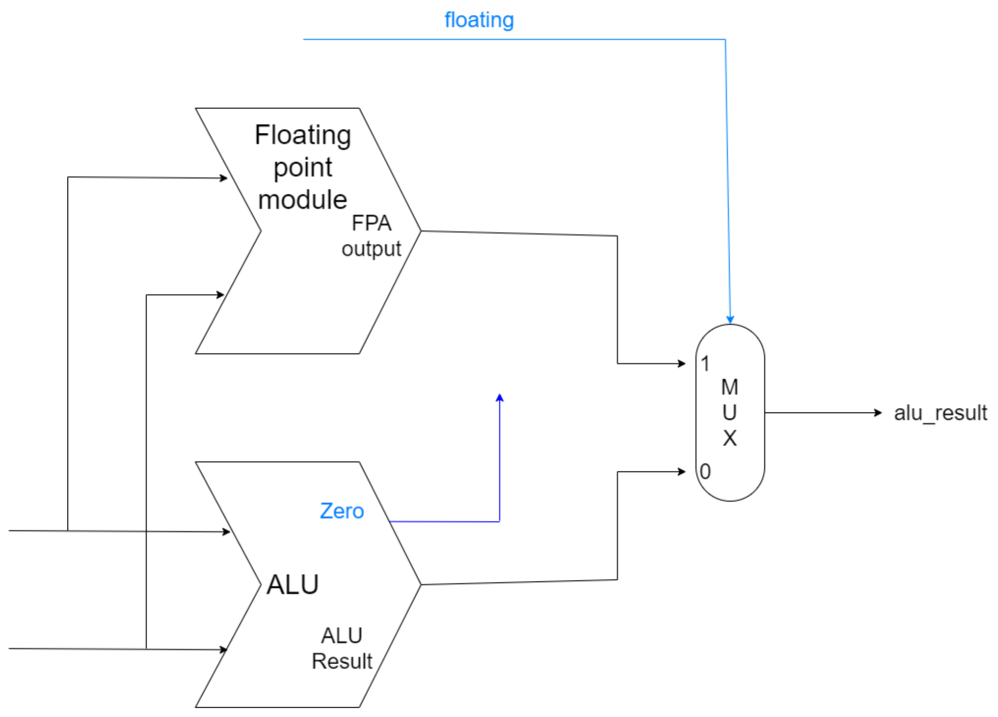
Floating Point Arithmetic Module

The design of the processor is based on the picture shown on the cover page, from Patterson and Hennessy’s book. However, a key addition is the floating point arithmetic module.

This has been added for floating point addition and runs parallel to the normal ALU. Its inputs are the same as the normal ALU and there is a new MUX introduced to select between the ALU output and the output of the floating point module.

This MUX has a select signal which is called “floating” and is asserted when there is a floating point instruction. The output of this MUX is alu_result.

This can be visualized using this small circuit diagram



Design of the Processor

In this section, a brief overview of all the modules implemented will be provided. The entire code has been divided into 14 different files for easy organization and readability of the code. I will now describe the actions of each module one by one.

For a detailed description, do go through the code as it is self-explanatory and has plenty of comments.

1. mips.v

This file has the module **mips**, which contains all the connections of the circuit and instantiates all of the internal modules. The top module only has two inputs – clk and rst, as all of the other signals are internal. This is the module that is simulated in the testbench and so, to view the waveforms we need to delve into uut signals as mentioned before.

2. mipsTb.v

This file has the testbench module **mipsTb**, which is the module needed to be run for simulation. It makes the simulation time 104 ns, which is enough for all the programs demonstrated earlier, however for writing your own programs, do feel free to increase this time as needed. Also, these two lines of code (line 14 and 15)

```
$dumpfile("mips_output.vcd");
$dumpvars(0, mipsTb);
```

Are used for saving the waveform for viewing in GTKWave and can be removed if the code is being simulated using ModelSim.

3. pc.v

The file pc.v has two modules. **pc_register** is the PC register which is set to 0 when rst is enabled and otherwise works like a normal register during a clock edge. This is done to make sure the PC is 0 at the beginning of the program.

pc_adder increments the pc_in by 4 to make pc_out whenever pc_in changes, that is, at the clock edge or rst. This value can be used to make pc_in later depending on branch and jump conditions.

4. **instructionMemory.v**

The module **instructionMemory** stores the instructions of the program. It is made up of 128 memory locations each of 32 bits. pc_in is given as the input to the instructionMemory module, which returns the 32 bit instruction pointed to by the PC.

All of the sample programs discussed before have been written as comments in the instruction memory. Do note that you should leave the first location (which is instructionMem[0]) blank and start giving your instructions from instructionMem[1].

For running your own programs, feel free to add your instructions in this module.

5. **controlUnit.v**

The **controlUnit** module has one input, which is the 6 bit opcode of the instruction. It is given by instruction[31:26] and it generates the different control signals based on the opcode. The instructions implemented in the processor have been summarized before but I will list those again here with their opcode –

R-type instructions (opcode 0)
Floating point addition (opcode 51) - Assigned arbitrarily
addi instruction (opcode 8)
lw instruction (opcode 35)
sw instruction (opcode 43)
beq instruction (opcode 4)
jump instruction (opcode 2)

The different control signals generated by the control unit are –
RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, ALUOp, jump and floating (used for floating point arithmetic, all of the other signals are based on the Patterson and Hennessy book).

These control signals are derived from the opcode using a case statement.

6. **MUX.v**

MUX.v contains two modules – **MUX_5bit**, which is a 5 bit MUX and **MUX_32bit**, which is a 32 bit MUX. There are 6 MUXes in the processor and these modules are used to implement them.

7. **registerFile.v**

This has the module **registerFile**, which stores, updates, and writes to the 32 registers in the processor. If the input control signal regWrite is 1, it will write the writeData on to writeReg. Otherwise, outputs readData1 and readData2 will get the values present in readReg1 and readReg2, which are provided from the appropriate bits of the 32 bit instruction.

In the demonstration programs, we have used $zero, which is the first register of the file at registerMem[0]. Temporary registers start from registerMem[8] and $s0 register is at registerMem[16], as given in the Patterson and Hennessy book.

Do note that $s0 and $s1 are already loaded with IEEE754 values for demonstration of the floating point arithmetic module. All other registers are initially zero.

8. **signExtend.v**

The **singExtend** module returns a 32 bit sign extend version of a 16 bit number. It does this by repeating the sign bit of the 16 bit number 16 more times.

9. **aluControl.v**

The **aluControl** module generates the aluControlSignal, which tells the ALU what instruction has to be performed. The inputs to the aluControl module are the function field of the instruction and ALUOp.

If ALUOp is 0 (LW or SW or addi instruction), aluControlSignal is 2 for addition. If ALUOp is 1 (beq instruction), aluControlSignal is 6 for subtraction. Else, for R-type instructions, aluControlSignal depends on the function field of the operation.

10. **alu.v**

The **alu** module has two 32 bit inputs, a 4 bit control signal and two outputs – aluZero, to show when the result is zero, and aluOut, the 32 bit output of the computation. A case statement is written to calculate the output depending on the control signal.

One of the 32 bit inputs is directly from the register file and the other is taken, using a MUX, either from the register file or from the sign extended version of instruction[15:0] depending on the ALUSrc control signal.

In the Patterson and Hennessy book, this output is used directly as the alu result. However, I have also implemented floating point arithmetic so that must be taken into account while deciding on the alu result.

11. **floatingpointadder.v**

This file is my combined code from the floating-point-adder repository. It has many internal modules but the main one to focus on is the **floatingpointadder** module. It takes in two 32 bit numbers in the IEEE754 format of representation and gives their sum, also in IEEE754 format of floating point representation.

In this processor, the outputs to the **alu** module and to the **floatingpointadder** module are the same. However, the alu_result is selected using a MUX with the select signal 'floating', which is set to 1 whenever we have a floating point instruction.

12. **dataMemory.v**

The **dataMemory** module has 128 memory locations of 32 bits each. Using the control signals MemRead and MemWrite, data can be read from or written to the data memory, as we have seen in the sample program shown for sw and lw demonstration.

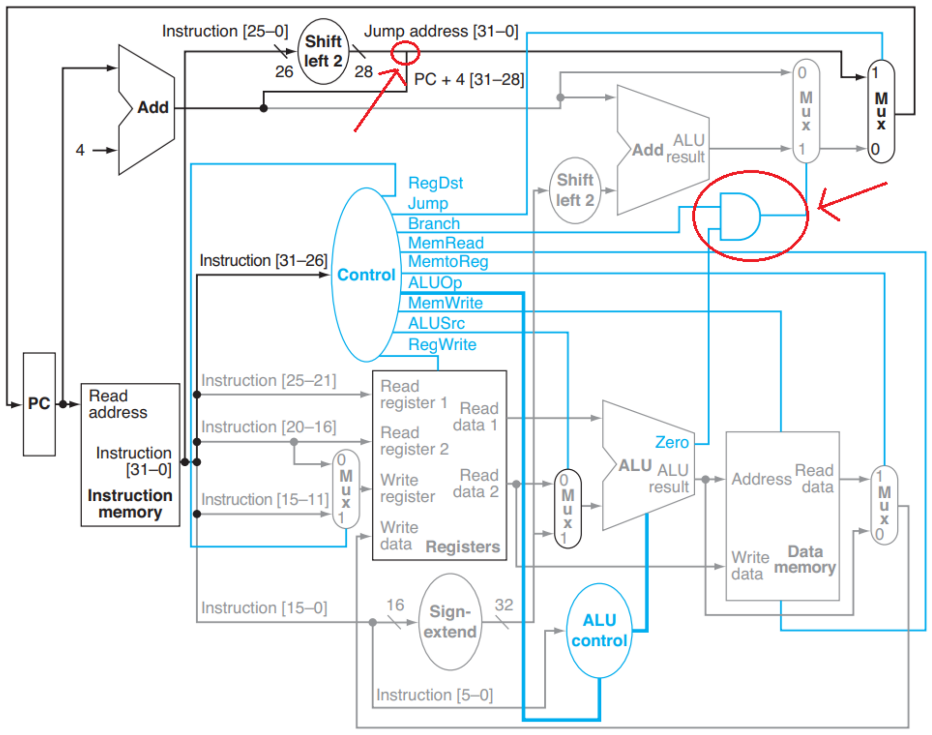
13. **shiftLeft2.v**

This file has two modules, one for each shiftLeft2 block in the processor circuit. The **shiftLeft2_26bit** module takes a 26 bit value and shifts it left 2 bits to make a 28 bit value. This is used for calculating the jump address. The **shiftLeft2_32bit** has a 32 bit input and a 32 bit output which is the input shifted left by 2 bits. It is used in the branch operation.

14. **adder.v**

The **adder** module is used to calculate the new PC value in case of branch instructions. The PC + 4 value is added to the branch offset and fed to a MUX, where a select signal determines whether the branch will be taken. This is demonstrated in the sample program regarding the beq instruction.

Some other points to note –



Note that the node for the jump address and the and gate highlighted in the figure above have been implemented directly in mips.v without any other module instantiations using the following two lines of code –

```
assign adder_mux_select = Branch & aluZero;

assign jumpAddress = {pc_out[31:28], jump_temp};
```