Project #2

Design and Implementation of a MIPS CPU with a Multicycle Datapath

By: Jeff Grindel – A20237004

Thomas Demeter – A20236867

Instructor: Dr. Suresh Borkar

ECE 485-01

Due Date: 11/15/12

II. Executive Summary

This report goes over what project the group was assigned with, how the group went about designing and implementing the project by making certain assumptions, backing up those assumptions with correct output data. The report finishes with a longer conclusion and all the related code attached.

III. Introduction

The project assigned to the group is designing a custom RISC processor, a highly basic implementation of a MIPS processor. The group is striving to learn more about computer architecture by having a more in depth and hands on experience than project one dealing with design problems. The group will design a 32-bit MIPS processor but with a reduced instruction set of the actual MIPS instruction set. Multicycle datapath implementation will be achieved using the VHDL hardware descriptive language. The processor should support the three instruction formats of R, I, and J, along with store word and load word. A table is provided of which all the instructions must be designed.

Table I: Core MIPS Instruction Set to be Designed (with example)

OpCode [31 : 26]	Function Field [5 : 0]	Instruction	Operation
100011	--	lw	lw $s3, 100($t2)
101011	--	sw	sw $s4, 200($t5)
000000	100000	add	add $t3, $t2, $s2
000000	100101	or	or $t5, $s6, $t5
000000	101010	slt	slt $t6, $s1, $t2
000100	--	beq	beq $t5, $s2, 600
000010	--	j	j 700
		(Custom set)	

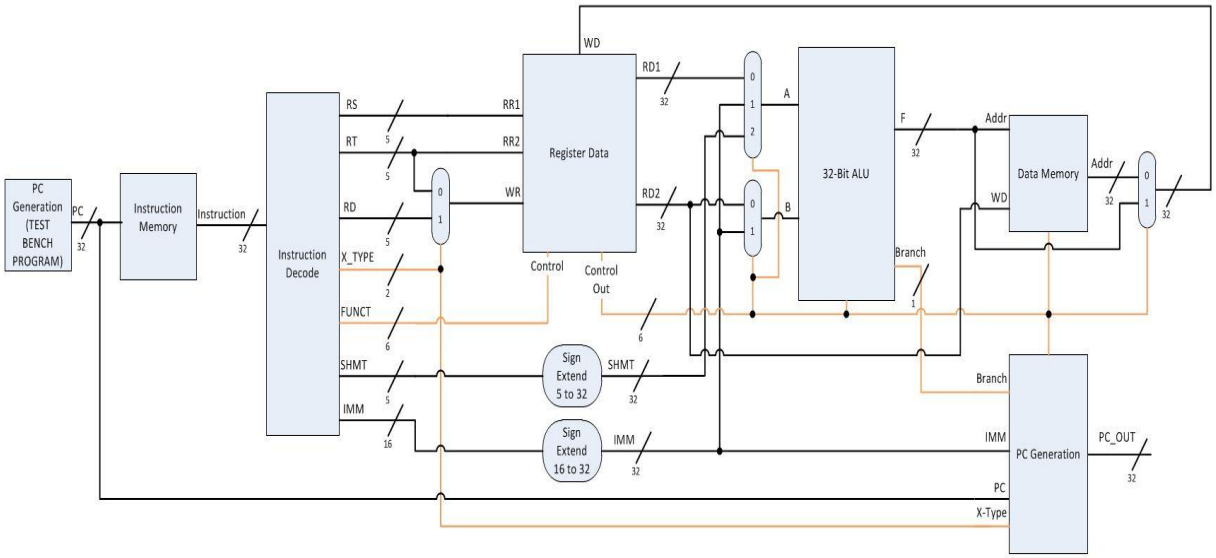
As seen in the table, there is a custom set that also needs to be implemented, which is chosen based on the last digit of the student ID's of the group. The group decided to use set 4.

The total set you need to design is the core set as above + a custom set designated for you as follows.
Student ID ending in:

1. BNE, ANDI, SRL, LUI
2. NAND, BNE, SUBI, SLL
3. SUBI, ANDI, SRL, LUI
4. BNE, ORI, SRL, LUI
5. NAND, ORI, SRL, LUI
6. OR, ANDI, LUI, JR
7. BNE, SUBI, SLL, LUI
8. NAND, ANDI, SRL, LUI

IV. Design

The group decided after an amount of time, discussion, and changes, to use this design for the entire processor and data path:



There are a few things in the design portion that need to be discussed. The orange lines are control signals. If it is not so clear from the picture, the group put together a couple of things in the data path to make some things simpler. Starting from the left side of the data path, the first thing encountered is the program counter (PC) generator. It is a test bench that generates a hard coded PC starting from $500 to $528 that will include all the PC values for the instructions that need to be executed. In the group's design, the OP codes are hard coded to specific PC values, and as such, with every new instruction that gets implemented; a fresh, correct PC value is given, for testing purposes and such. The PC Generator at the end gives a PC value that the CPU should have gone to. This design made it easier to make sure things were working properly and as they should. In a real CPU, the PC values are not new every time, they are tied to the instruction that completed before hand and to whatever PC value was generated.

The second item from the left is the instruction memory. It is 32 bits wide, and it has all the instructions stored here. What happens is the memory located at the addresses given by the PC will be fetched and sent. So, with the implementation discussed previously of the PC, all the instructions are actually located sequentially and will be implemented as such. The picture provided will show what is meant by the instruction memory receiving the subsequent PC value.

Instruction	Instruction w/Reg	PC
lw $s3, 100($t2)	lw $19,100($10)	500
sw $s4, 200($t5)	sw $20,200($13)	504
add $t3, $t2, $s2	add $11,$10,$18	508
or $t5, $s6, $t5	or $13,$22,$13	50C
slt $t6, $s1, $t2	slt $14,$17,$10	510
beq $t5, $s2, 600	beq $13,$18,0[$600-0x00000514]	514
j 700	j 0x00000000[$700]	518
bne $t5, $s2, 600	bne $13,$18,0[$600-0x0000051C]	51C
ori $t5,$t6,10	ori $13,$14,10	520
srl $t6,$t1,10	srl $14,$9,5	524
lui $t3,40	lui $11,40	528

The instructions will be fetched and decoded in the next segment, the instruction decode. After the instruction decode, the instruction will be split up into smaller segments, representing Op Code, Rs, Rt, Rd, Imm, Funct, Shamt, all depending on what type of instruction it was. If it was a R-type instruction, it would output Rs, Rt, Rd, Shamt, and Funct. For a J-type, it would use the Imm bits, and for I-type it would use Rs, Rt, and Imm. All of instructions would have an Op Code for them, signifying what kind of instruction they are, and how the decode should split the bits up. The picture following shows how the instruction decode functions and what it splits things up into:

OpCode[31:26]	Rs[25:21]	Rt[20:16]	Rd[15:10]	shamt[10:6]	function[5:0]	Hex
OpCode[31:26]	Rs[25:21]	Rt[20:16]	imm[15:0]			
OpCode[31:26]	Adress[25:0]					
(35)100011	(10)01010	(19)10011	(100)0000000001100100			0X8D530064
(43)101011	(13)01101	(20)10100	(200)0000000011001000			0XADB400C8
(0)000000	(10)01010	(18)10010	(11)01011	(0)00000	(32)100000	0X01525820
(0)000000	(22)10110	(13)01101	(13)01101	(0)00000	(37)100101	0X02CD6825
(0)000000	(17)10001	(10)01010	(14)01110	(0)00000	(42)101010	0X022A702A
(4)000100	(13)01101	(18)10010	(59)0000000000111011			0X11B2003B
(2)000010	(700 but really 175*4)00 0000 0000 0000 0000 1010 1111					0X080000AF
(5)000101	(13)01101	(18)10010	(57)0000000000111001			0X15B20039
(13)001101	(14)01110	(13)01101	(10)0000000000001010			0X35CD000A
(0)000000	(0)00000	(9)01001	(14)01110	(5)00101	(2)000010	0X00097142
(15)001111	(0)00000	(11)01011	(40)0000000000101000			0X3C0B0028

As with the pc, the instructions are sequentially decoded and split into certain parts.

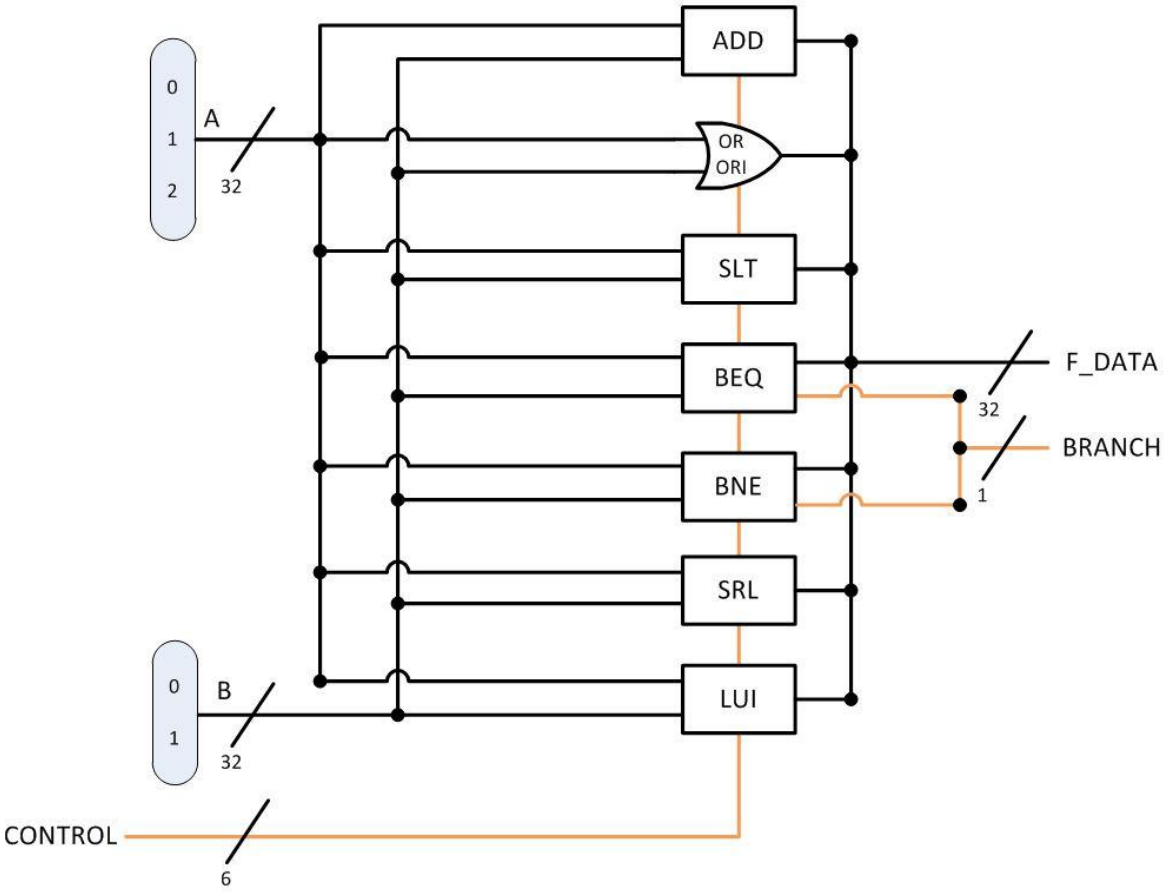
The next box is the register data. Within it, the data contained in registers is kept. In the group's design, there are two sets of register data values: a constant set and a set that changes. The constant set is there so that for every instruction, there is an unaltered set of values that the group knows. This constant set is the one the data is read from. The changeable set is the one that gets written to once the ALU finishes and the data gets written back. This helps the group see and easily know if something worked out properly or not.

Register (MIPS Name)	Register Name	Initial Value (In Hex)
ZERO	REG_0	0
$t0	REG_8	80
$t1	REG_9	90
$t2	REG_10	100
$t3	REG_11	110
$t4	REG_12	120
$t5	REG_13	130
$t6	REG_14	140
$t7	REG_15	150
$s0	REG_16	160
$s1	REG_17	170
$s2	REG_18	180
$s3	REG_19	190
$s4	REG_20	200
$s5	REG_21	210
$s6	REG_22	220
$s7	REG_23	230
$t8	REG_24	240
$t9	REG_25	250

As shown the table, the group just took the register values for the registers, and added a zero at the end. The group took this liberty since the project guidelines left it up for the group to decide on how to progress and

deal with the data contained within the registers. Also, an internal write control signal was added so that instructions that do not write back data do not actually write anything back. The issue that was happening before was that all the instructions were writing back, regardless if it should or should not. Instructions such as the branches or the sets were writing back whatever value came out of the ALU into whatever register was the destination register.

The next large section is the ALU. For the ALU design, the group went with:



In the ALU itself, the top MUX has as the 0 being Rs, 1 as Imm, and 2 as Shamt. The bottom MUX has 0 as Rt and 1 Imm. The MUX's are controlled by Control Out signal. What the ALU does is just simply get the input values and do math (or whatever the instruction requires) on them. It then just outputs the data that has been altered and potentially a branch address if that is what the instruction was.

Afterwards is the data memory, which stores the next address and potential data needed to be written back. Data memory will only be really used for load word and store word instructions. Other things within the data path are sign extenders, which take the current number and sign extend it to a full 32 bits. The first MUX is controlled by what type of instruction it is and whether it needs to use a separate Rd value, or if it can use Rt again.

There is another PC generator that will calculate the program counter value after instructions are decoded. Depending on what kind of instruction it was, whether jump, branch, or a simple R-type, the PC_OUT value will be updated.

ALU OP	PC Value	New PC Value	
		Branch EQ	Branch Not EQ
add	500	504	-
add	504	508	-
add	508	50C	-
OR	50C	510	-
set	510	514	-
b	514	600(PC+Off*4)	518
no op	518	700(2BC)	-
b	51C	520	600(PC+Off*4)
or	520	524	-
shift	524	528	-
load	528	52C	-

As shown in the table, the values for PC_OUT are calculated as they should be. But, they are not used, since the PC generator at the beginning is used.

V. Implementation

To show the implementation of everything, code snippets will be shown in respect to their programs. Most everything related to the instructions was hard coded in; it made things easier to work with and to know exactly what was going on and what should theoretically happen. Since the first PC generator was simply an incremental thing, the code for that will be shown in the overall scheme.

In going with the hard coded ideal, the instruction memory's main section was just a list of all the instructions.

```
architecture behave of Instruction_Memory is
begin
    INSTRUCTION <=  x"8D530064" when PC = x"00000500" else  --lw
                    x"ADB400C8" when PC = x"00000504" else  --sw
                    x"01525820" when PC = x"00000508" else  --add
                    x"02CD6825" when PC = x"0000050C" else  --or
                    x"022A702A" when PC = x"00000510" else  --slt
                    x"11B2003B" when PC = x"00000514" else  --beq
                    x"080000AF" when PC = x"00000518" else  --j
                    x"15B20039" when PC = x"0000051C" else  --bne
                    x"35CD000A" when PC = x"00000520" else  --ori
                    x"00097142" when PC = x"00000524" else  --slt
                    x"3C0B0028" when PC = x"00000528" else  --lui
                    x"FFFFFFFF";
end architecture behave;
```

As seen above, the PC increments by one every time, and a new instruction is sent to the instruction decode.

For the decoder, the major section was a part that was created as a temp that stood in for the type of the instruction. As it will be shown in the code following, '00' was a R-type instruction, '10' an I-type, '01' was J-type, and '11' was an error code, if something went wrong, this got thrown. The TEMP variable was made to be X_TYPE which was sent along the rest of the data path, which allowed the other pieces to know what kind of instruction it was.

```
TEMP <= "10" when INST = x"8D530064" else   --lw
            "10" when INST = x"ADB400C8" else --sw
```

```
                "00" when INST = x"01525820" else --add
                "00" when INST = x"02CD6825" else --or
                "00" when INST = x"022A702A" else --slt
                "10" when INST = x"11B2003B" else --beq
                "01" when INST = x"080000AF" else --j
                "10" when INST = x"15B20039" else --bne
                "10" when INST = x"35CD000A" else --ori
                "00" when INST = x"00097142" else --srl
                "10" when INST = x"3C0B0028" else --lui
                "11";

    X_TYPE <= TEMP;
```

For the register data section, what happened was based on the instruction and what the instruction used, the certain registers were read. It was just based off on the register values. A section will be shown for an example.

```
RD1 <= C_REG_8 when (RR1 =  "01000") else
           C_REG_9 when (RR1 =  "01001") else
           C_REG_10 when (RR1 = "01010") else
           C_REG_11 when (RR1 = "01011") else
           C_REG_12 when (RR1 = "01100") else
```

As an example, when an instruction used register $t0, which correlates to a register value of 8, the value of 01000 was passed.

For the two sign extends, since the group knew what values would be passed, it was easy to fill in the upper bits in relation to what they should be.

```
architecture behavior of Sign_Extend_16 is
begin
    OUTPUT(31 downto 16) <= x"0000";
    OUTPUT(15 downto 0) <= INPUT;
end architecture behavior;
```

The biggest part of the ALU was making it do the correct things when handed the instructions. It was all based on the control signals going in that were based on the function and the type of instruction that was decoded earlier in the instruction decode section.

```
      begin

    F_DATA <= std_logic_vector(unsigned(A_DATA) + unsigned(B_DATA)) when (CONTROL =
"100011" OR CONTROL ="101011" or CONTROL = "100000") else --LW, SW, ADD
                A_DATA OR B_DATA when (CONTROL = "100101" OR CONTROL = "001101") else --OR,
ORI
                x"00000001" when (CONTROL = "101010" AND (A_DATA < B_DATA)) else --SLT A<B
                x"00000000" when (CONTROL = "101010" AND (A_DATA > B_DATA)) else --SLT A>B
                B_DATA when ((CONTROL = "000100" OR CONTROL = "000101") AND (A_DATA =
B_DATA)) else --BEQ
                B_DATA when ((CONTROL = "000100" OR CONTROL = "000101") AND (A_DATA /=
B_DATA)) else --BNE
                std_logic_vector(unsigned(B_DATA) srl to_integer(unsigned(A_DATA))) when
CONTROL = "000010" else --SRL
                std_logic_vector(unsigned(B_DATA) or (unsigned(A_DATA) sll 16)) when
CONTROL = "001111" else      --Imm in A: B: RT
                x"FFFFFFFF";
```

As seen in the code section, the two data inputs were altered and sent to F_DATA which was the output from the ALU.

The data memory section was implemented knowing that everything was hard coded and what the expected outputs should be once the expected inputs came in.

```
architecture behave of Data_Memory is


signal MEM_164: std_logic_vector(31 downto 0) := x"00001234";   --For lw, data will be
put into Reg 19
signal MEM_1F8: std_logic_vector(31 downto 0) := x"00001111";   --For sw, data will
change to $200
signal MEM_ERROR: std_logic_vector(31 downto 0) := x"FFFFFFFF"; --For sw, data will
change to $200
```

What is shown here is how things should happen for the separate lw and sw instructions

The last piece that ties everything together is the second PC, the one that calculates what the PC should be at the very end of every instruction. This is based on what the instruction type was, and if it was one that needed a jump or branch, to calculate it.

```
PC_OUT <= std_logic_vector((unsigned(IMM) sll 2) + unsigned(PC_IN)) when (Branch = '1'
and (Control = "000100" or Control = "000101")) else --For branch
              std_logic_vector(unsigned(IMM) sll 2) when (X_Type = "01") else
              std_logic_vector(unsigned(PC_IN) + 4); --Default case of adding 4
```

There were basically 3 different things it could do, branch, jump, or increment. All three cases are covered with the code.

At the end of each instruction, there would a couple of outputs. One would be the PC value that should be next. The other would be the changed register data values, if they were changed at all.

VI. Results

As mentioned in the paragraph previous, there was not much of an output in terms of the number of things outputted. The picture that will be shown will be a conglomeration of a bunch of the registers, the PC, and the separate decoded sections of the instructions. There will also be a section of the overall test bench shown. What the group did was follow the hints given in the project description that stated to make the individual components, and test all of them, then put them all together.

lw $s3,100($t2) or lw $19,100($10)

The instruction load word is meant to load the word stored at 100($10)(With the initialized values this turns out to be memory location $164 which has the value $1234 stored in it) and store it in Reg 19. In this case we can see in the picture attached that the new value of Register 19 was changed to $1234. It can also be seen that the PC out value shows that it should be a sequential output, incrementing the PC value to PC+4 to a value of $504.

sw $s4,200($t5) or sw $20,200($13)

The instruction Store word is meant to store the word at Reg 20 in the memory location designated by 200($13) (With initialized values, this memory location is $1F8). It can be seen in the picture that the value of Reg 20 is now the same as the value at memory location $1F8.Store word also increments the PC value sequentially to PC+4 to a value of $508.

add $t3,$t2,$s2 or add $11,$10,$18

The instruction Add simply adds the value of Reg 10 and Reg 18, and puts it Reg 11. The initial values of Reg 10 was $100 and Reg 18 was $180, thus when added results in $280. It can be seen in the picture of the results

that the Value of Reg 11 was in fact $280. The PC value is incremented sequentially to PC+4 to a value of $50C.

or $t5,$s6,$t5 or or $13,$22,$13

The instruction or takes the logical OR of Reg 22 and Reg 13. It then places this value into Reg 13. The initial values of Reg 22 was $220 and Reg 13 was $130. The logical OR of these two hex numbers is $330. In the results picture it can be seen that the new value of Reg 13 is $330. The PC is also incremented sequentially to PC + 4 to a value of $510.

slt $t6,$s1,$t2 or slt $14,$17,$10

The instruction Set Less Than will set the value to $1 in Reg 14 if Reg 17 is less than Reg 10. Or it will set the value to $0 if Reg 17 is not less than Reg 10. In this case the value of Reg 17 was not less than Reg 10 the value of Reg 14 was set to $0. The PC is also incremented sequentially to PC+4 to a value of $514.

beq $t5,$s2,600 or beq $13,$18,0[$600-$514]

The instruction branch if equal will branch to the branch address (which is calculated by taking the branch address, subtracting the current PC, and the adding that to the PC value) if the value in Reg 13 and Reg 18 are equal. In this case the value of Reg 13 is $130 and Reg 18 is $180, so they are not equal thus resulting in just a sequential increment of the PC to PC+4 to a value of $518.

j 700

The instruction jump will set the PC value to the immediate value assigned, in this case 700. It can be seen in the picture that the new PC value is $2BC (700 in hex).

bne $t5,$s2,600 or beq $13,$18,0[$600-$51C]

The instruction branch if not equal will branch to the branch address (which is calculated by taking the branch address, subtracting the current PC, and the adding that to the PC value) if the value in Reg 13 and Reg 18 are not equal. In this case the value of Reg 13 is $130 and Reg 18 is $180, so they are not equal thus resulting in the branch to the new PC. It can be seen in the picture of the results that the new value of the PC is $600.

ori $t5,$t6,10 or ori $13,$14,10

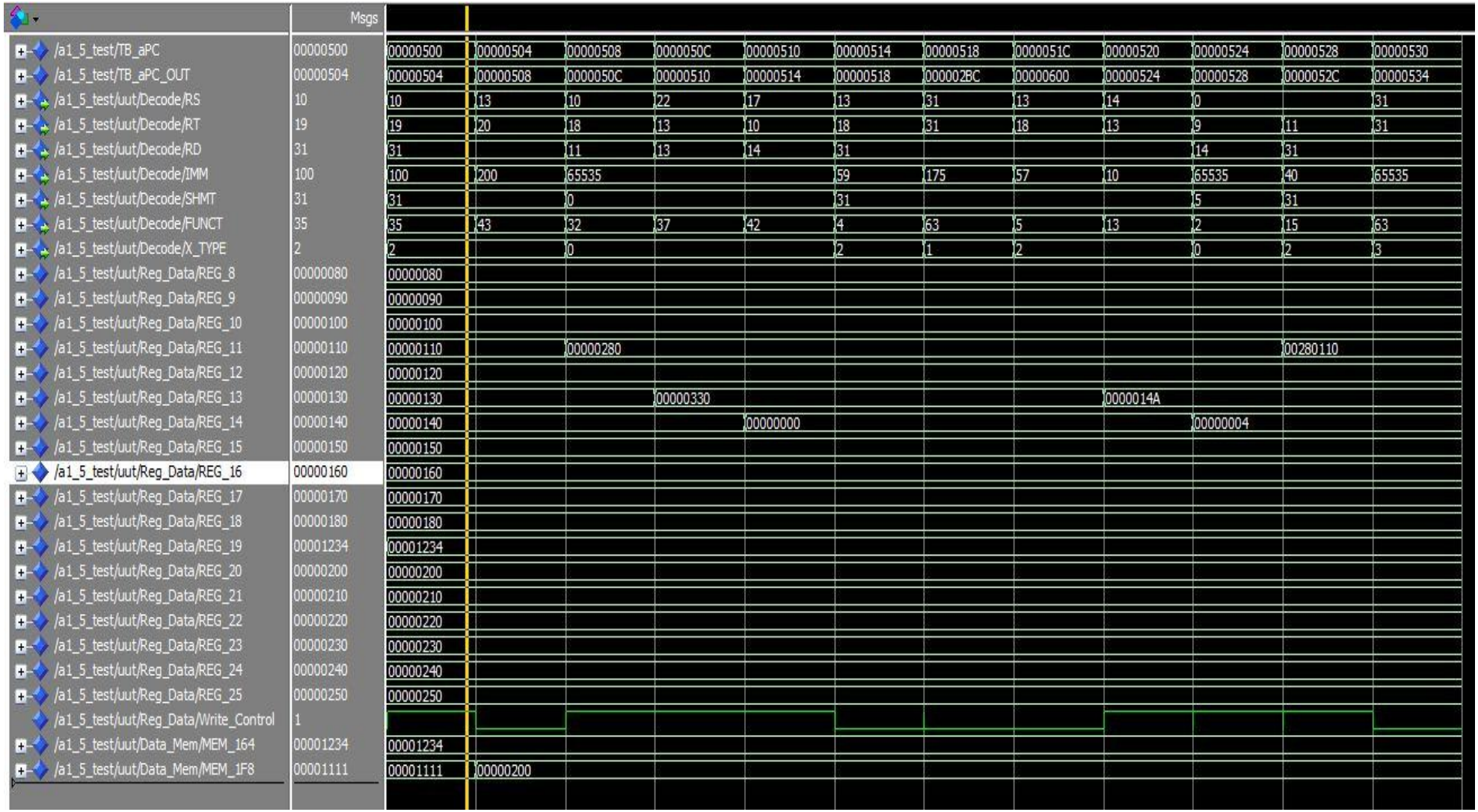
The instruction Immediate Or takes the logical OR between Reg 14 and the immediate value 10 and places the result in Reg 13. In this case the value of Reg 14 is $140, when this is OR'ed with 10 the result is $14A. This can be seen as the result placed in Reg 13 in the results picture. The PC is also sequentially incremented to PC+4 to a value of $524.

srl $t6,$t1,10 or srl $14,$9,5

The instruction Shift Right Logical takes the value in Reg 9 and shifts it to the right 5 places and puts the result in Reg 14. The value if Reg 9 is $90, this shift to the right by 5 places results in $4. This can be seen as the result in Reg 14 in the results picture. The PC is also sequentially incremented to PC+4 to a value of $528.

Lui $t3,40 or lui $11,40

The instruction load upper immediate take the immediate value of 40 and loads it into the upper half of the word in Reg 11. This would result in a value of $00280110 in Reg 11. This can be seen in the results picture. The PC is also sequentially incremented to PC+4 to a value of $52C.


Msgs
/a1_5_test/TB_aPC 00000500 00000500 00000504 00000508 0000050C 00000510 00000514 00000518 0000051C 00000520 00000524 00000528 00000530
/a1_5_test/TB_aPC_OUT 00000504 00000504 00000508 0000050C 00000510 00000514 00000518 000002BC 00000600 00000524 00000528 0000052C 00000534
/a1_5_test/uut/Decode/RS 10 10 13 10 22 17 13 31 13 14 0 31
/a1_5_test/uut/Decode/RT 19 19 20 18 13 10 18 31 18 13 9 11 31
/a1_5_test/uut/Decode/RD 31 31 11 13 14 31 14 31
/a1_5_test/uut/Decode/IMM 100 100 200 65535 59 175 57 10 65535 40 65535
/a1_5_test/uut/Decode/SHMT 31 31 0 31 5 31
/a1_5_test/uut/Decode/FUNCT 35 35 43 32 37 42 4 63 5 13 2 15 63
/a1_5_test/uut/Decode/X_TYPE 2 2 0 2 1 2 0 2 3
/a1_5_test/uut/Reg_Data/REG_8 00000080 00000080
/a1_5_test/uut/Reg_Data/REG_9 00000090 00000090
/a1_5_test/uut/Reg_Data/REG_10 00000100 00000100
/a1_5_test/uut/Reg_Data/REG_11 00000110 00000110 00000280 00280110
/a1_5_test/uut/Reg_Data/REG_12 00000120 00000120
/a1_5_test/uut/Reg_Data/REG_13 00000130 00000130 00000330 0000014A
/a1_5_test/uut/Reg_Data/REG_14 00000140 00000140 00000000 00000004
/a1_5_test/uut/Reg_Data/REG_15 00000150 00000150
/a1_5_test/uut/Reg_Data/REG_16 00000160 00000160
/a1_5_test/uut/Reg_Data/REG_17 00000170 00000170
/a1_5_test/uut/Reg_Data/REG_18 00000180 00000180
/a1_5_test/uut/Reg_Data/REG_19 00001234 00001234
/a1_5_test/uut/Reg_Data/REG_20 00000200 00000200
/a1_5_test/uut/Reg_Data/REG_21 00000210 00000210
/a1_5_test/uut/Reg_Data/REG_22 00000220 00000220
/a1_5_test/uut/Reg_Data/REG_23 00000230 00000230
/a1_5_test/uut/Reg_Data/REG_24 00000240 00000240
/a1_5_test/uut/Reg_Data/REG_25 00000250 00000250
/a1_5_test/uut/Reg_Data/Write_Control 1
/a1_5_test/uut/Data_Mem/MEM_164 00001234 00001234
/a1_5_test/uut/Data_Mem/MEM_1F8 00001111 00001111 00000200

What the group did was get the individual pieces of the data path, make them components in the overall test bench, and map them basically for the inputs and outputs that should be going to each component.

```
entity a1_5 is
    port (aPC: in std_logic_vector(31 downto 0);         --PC Input
          aPC_OUT: out std_logic_vector(31 downto 0);
          aF_DATA: out std_logic_vector(31 downto 0);
          aTEMP_TYPE: out std_logic_vector(1 downto 0));
end entity a1_5;
```

The section of code shows the entity of the test bench and shows what is accomplished overall. A PC goes in, and a PC comes out along with the data, and the instruction type. The PC that comes out was figured out by the second PC component,

```
PC: PC_Calc port map(PC_IN =>aPC,
                          IMM => TEMP_SE16OUT,
                          Branch => TEMP_BRANCH,
                          Control => TEMP_FUNCT,
                          X_TYPE => TEMP_TYPE,
                          PC_OUT => aPC_OUT);
```

As can be seen, the PC_OUT was mapped to the aPC_OUT value. The input PC was accomplished by the first PC that was fed to the instruction memory,

```
Instruction_Mem: Instruction_Memory port map(PC => aPC,
                                     INSTRUCTION => TEMP_INST);
```

The instruction memory clearly gets its PC value via the aPC value. The type of instruction and the data out were figured out in earlier components and just passed along,

```
aF_DATA <= TEMP_F_DATA;
     aTEMP_TYPE <= TEMP_TYPE;
```

In the picture itself, the values such as 31 and 65535 are in there for just when there needed to be a value written in there for the instructions to go through. It is apparent in the picture which registers' data was changed when an instruction came up that did exactly that. It shows each instruction's decoded bits, and proves shows that everything was and is in working order, it even shows when registers were being written to. Each instruction was decoded, register data grabbed, sent to the ALU, worked on, and spit out a value.

VII. Conclusions

In conclusion, everything works. With hardcoding everything, the group knew what values would appear, and what each instruction would do. Knowing it all made everything a little bit easier to code. The report shows how each component was set up and how it all came together at the end in the final test bench program. The picture, in line with the report and also attached to Blackboard in its glory, shows each iteration of the program: what PC values there were, and how things progressed for each instruction type. The project gave the group a better understanding of how VHDL works and how it is used to power processors.

Attachments

Each individual program's code will be attached, the individual test benches for the components won't be necessary, since the overall test bench works without issue.

Instruction Memory Program

```
--Instruction Memory: Where all the opcodes for the specific instructions are stored
--Takes in a PC value and generates what insturction it is.
--Jeff Grindel, Tom Demeter
library ieee;
use ieee.std_logic_1164.all;

entity Instruction_Memory is
   port (PC: in std_logic_vector(31 downto 0);  --32-bit instruction
         INSTRUCTION: out std_logic_vector(31 downto 0)); --32-bit instruction(hardcoded
in)
end entity Instruction_Memory;

architecture behave of Instruction_Memory is
begin
    INSTRUCTION <=  x"8D530064" when PC = x"00000500" else  --lw
                    x"ADB400C8" when PC = x"00000504" else  --sw
                    x"01525820" when PC = x"00000508" else  --add
                    x"02CD6825" when PC = x"0000050C" else  --or
                    x"022A702A" when PC = x"00000510" else  --slt
                    x"11B2003B" when PC = x"00000514" else  --beq
                    x"080000AF" when PC = x"00000518" else  --j
                    x"15B20039" when PC = x"0000051C" else  --bne
                    x"35CD000A" when PC = x"00000520" else  --ori
                    x"00097142" when PC = x"00000524" else  --slt
                    x"3C0B0028" when PC = x"00000528" else  --lui
                    x"FFFFFFFF";
end architecture behave;
```

Instruction Decode Program

```
--Insturction Decoder
--Input 32-bit Instruction In binary/hex format
--Output a 26-bit output that is J-type, R-type, or I-Type
--Jeff Grindel, Tom Demeter
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Instruction_Decode is
   port (INST: in std_logic_vector(31 downto 0);     --32-bit instruction
         RS: out std_logic_vector(4 downto 0);       --5-bit RS output
         RT: out std_logic_vector(4 downto 0);       --5-bit RT output
         RD: out std_logic_vector(4 downto 0);       --5-bit RD output(R-Type only)
         IMM: out std_logic_vector(15 downto 0);     --5-bit IMM output(I-Type only)
         SHMT: out std_logic_vector(4 downto 0);     --5-bit Shift Amount (R-Type only) --
was 10 down to 6
         FUNCT: out std_logic_vector(5 downto 0);    --6-bit Function Code (R-Type only)
         X_TYPE: out std_logic_vector(1 downto 0));  --2-bit output to determine type:
00:R-Type, 01:J-Type, 10:I-Type, 11: error
end entity Instruction_Decode;

architecture behave of Instruction_Decode is

signal TEMP: std_logic_vector(1 downto 0);
```

```
begin
    TEMP <= "10" when INST = x"8D530064" else   --lw
            "10" when INST = x"ADB400C8" else --sw
            "00" when INST = x"01525820" else --add
            "00" when INST = x"02CD6825" else --or
            "00" when INST = x"022A702A" else --slt
            "10" when INST = x"11B2003B" else --beq
            "01" when INST = x"080000AF" else --j
            "10" when INST = x"15B20039" else --bne
            "10" when INST = x"35CD000A" else --ori
            "00" when INST = x"00097142" else --srl
            "10" when INST = x"3C0B0028" else --lui
            "11";

    X_TYPE <= TEMP;

    RS <= INST(25 downto 21) when (TEMP ="00" or TEMP ="10") else
          "11111";
    RT <= INST(20 downto 16) when (TEMP ="00" or TEMP ="10") else
          "11111";
    RD <= INST(15 downto 11) when (TEMP ="00") else
          "11111";
    IMM <= INST(15 downto 0) when (TEMP ="10" or TEMP ="01") else
          x"FFFF";
    SHMT <= INST(10 downto 6) when (TEMP ="00") else
          "11111";
    FUNCT <= INST(5 downto 0) when (TEMP ="00") else        --Function or Opcode, will be
used as Control for ALU and Register_Data
             INST(31 downto 26) when (TEMP ="10") else
          "111111";

end architecture behave;
```

Register Data

```
--Register_Data
--Input: Rs, Rt, Rd, Write Data
--Controls
--Outpus: Read Data 1, Read Data 2
--Jeff Grindel, Tom Demeter
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Register_Data is
   port (RR1: in std_logic_vector(4 downto 0);      --5-bit Read Reg. 1
         RR2: in std_logic_vector(4 downto 0);      --5-bit Read Reg. 2
         WR: in std_logic_vector(4 downto 0);       --5-bit Write Register
         WD: in std_logic_vector(31 downto 0);      --32-bit Write Data
         Control: in std_logic_vector(5 downto 0);  --6-bit Opcode/Function
         RD1: out std_logic_vector(31 downto 0);    --32-bit output1
         RD2: out std_logic_vector(31 downto 0);    --32-bit output2
         CTRL_OUT: out std_logic_vector(5 downto 0));--Control Output to pass through the
control values for alu operation
end entity Register_Data;

architecture behave of Register_Data is

constant C_REG_0: std_logic_vector(31 downto 0) := x"00000000";
constant C_REG_8: std_logic_vector(31 downto 0) := x"00000080";
constant C_REG_9: std_logic_vector(31 downto 0) := x"00000090";
```

```
constant C_REG_10: std_logic_vector(31 downto 0) := x"00000100";
constant C_REG_11: std_logic_vector(31 downto 0) := x"00000110";
constant C_REG_12: std_logic_vector(31 downto 0) := x"00000120";
constant C_REG_13: std_logic_vector(31 downto 0) := x"00000130";     --Change to determine
BEQ or BNE
constant C_REG_14: std_logic_vector(31 downto 0) := x"00000140";
constant C_REG_15: std_logic_vector(31 downto 0) := x"00000150";
constant C_REG_16: std_logic_vector(31 downto 0) := x"00000160";
constant C_REG_17: std_logic_vector(31 downto 0) := x"00000170";     --check if SLT is
working
constant C_REG_18: std_logic_vector(31 downto 0) := x"00000180";
constant C_REG_19: std_logic_vector(31 downto 0) := x"00000190";
constant C_REG_20: std_logic_vector(31 downto 0) := x"00000200";
constant C_REG_21: std_logic_vector(31 downto 0) := x"00000210";
constant C_REG_22: std_logic_vector(31 downto 0) := x"00000220";
constant C_REG_23: std_logic_vector(31 downto 0) := x"00000230";
constant C_REG_24: std_logic_vector(31 downto 0) := x"00000240";
constant C_REG_25: std_logic_vector(31 downto 0) := x"00000250";


--Changing Vectors
signal REG_8: std_logic_vector(31 downto 0) := x"00000080";
signal REG_9: std_logic_vector(31 downto 0) := x"00000090";
signal REG_10: std_logic_vector(31 downto 0) := x"00000100";
signal REG_11: std_logic_vector(31 downto 0) := x"00000110";
signal REG_12: std_logic_vector(31 downto 0) := x"00000120";
signal REG_13: std_logic_vector(31 downto 0) := x"00000130";
signal REG_14: std_logic_vector(31 downto 0) := x"00000140";
signal REG_15: std_logic_vector(31 downto 0) := x"00000150";
signal REG_16: std_logic_vector(31 downto 0) := x"00000160";
signal REG_17: std_logic_vector(31 downto 0) := x"00000170";
signal REG_18: std_logic_vector(31 downto 0) := x"00000180";
signal REG_19: std_logic_vector(31 downto 0) := x"00000190";
signal REG_20: std_logic_vector(31 downto 0) := x"00000200";
signal REG_21: std_logic_vector(31 downto 0) := x"00000210";
signal REG_22: std_logic_vector(31 downto 0) := x"00000220";
signal REG_23: std_logic_vector(31 downto 0) := x"00000230";
signal REG_24: std_logic_vector(31 downto 0) := x"00000240";
signal REG_25: std_logic_vector(31 downto 0) := x"00000250";

signal Write_Control: std_logic := '0';

begin

    --Read Operation
    RD1 <= C_REG_8 when (RR1 =  "01000") else
           C_REG_9 when (RR1 =  "01001") else
           C_REG_10 when (RR1 = "01010") else
           C_REG_11 when (RR1 = "01011") else
           C_REG_12 when (RR1 = "01100") else
           C_REG_13 when (RR1 = "01101") else
           C_REG_14 when (RR1 = "01110") else
           C_REG_15 when (RR1 = "01111") else
           C_REG_16 when (RR1 = "10000") else
           C_REG_17 when (RR1 = "10001") else
           C_REG_18 when (RR1 = "10010") else
           C_REG_19 when (RR1 = "10011") else
           C_REG_20 when (RR1 = "10100") else
           C_REG_21 when (RR1 = "10101") else
           C_REG_22 when (RR1 = "10110") else
           C_REG_23 when (RR1 = "10111") else
           C_REG_24 when (RR1 = "11000") else
           C_REG_25 when (RR1 = "11001") else
```

```
            x"FFFFFFFF";

    RD2 <= C_REG_8 when (RR2 = "01000") else
           C_REG_9 when (RR2 = "01001") else
           C_REG_10 when (RR2 = "01010") else
           C_REG_11 when (RR2 = "01011") else
           C_REG_12 when (RR2 = "01100") else
           C_REG_13 when (RR2 = "01101") else
           C_REG_14 when (RR2 = "01110") else
           C_REG_15 when (RR2 = "01111") else
           C_REG_16 when (RR2 = "10000") else
           C_REG_17 when (RR2 = "10001") else
           C_REG_18 when (RR2 = "10010") else
           C_REG_19 when (RR2 = "10011") else
           C_REG_20 when (RR2 = "10100") else
           C_REG_21 when (RR2 = "10101") else
           C_REG_22 when (RR2 = "10110") else
           C_REG_23 when (RR2 = "10111") else
           C_REG_24 when (RR2 = "11000") else
           C_REG_25 when (RR2 = "11001") else
           x"FFFFFFFF";

    Write_Control <= '1' when (Control = "100011" or Control = "100000" or Control =
"100101" or Control = "101010" or Control = "001101" or Control = "000010" or Control =
"001111") else
                     '0';

    --Write Operations
    REG_8 <= WD when (WR = "01000" and Write_Control = '1');
    REG_9 <= WD when (WR = "01001" and Write_Control = '1');
    REG_10 <= WD when (WR = "01010" and Write_Control = '1');
    REG_11 <= WD when (WR = "01011" and Write_Control = '1');
    REG_12 <= WD when (WR = "01100" and Write_Control = '1');
    REG_13 <= WD when (WR = "01101" and Write_Control = '1');
    REG_14 <= WD when (WR = "01110" and Write_Control = '1');
    REG_15 <= WD when (WR = "01111" and Write_Control = '1');
    REG_16 <= WD when (WR = "10000" and Write_Control = '1');
    REG_17 <= WD when (WR = "10001" and Write_Control = '1');
    REG_18 <= WD when (WR = "10010" and Write_Control = '1');
    REG_19 <= WD when (WR = "10011" and Write_Control = '1');
    REG_20 <= WD when (WR = "10100" and Write_Control = '1');
    REG_21 <= WD when (WR = "10101" and Write_Control = '1');
    REG_22 <= WD when (WR = "10110" and Write_Control = '1');
    REG_23 <= WD when (WR = "10111" and Write_Control = '1');
    REG_24 <= WD when (WR = "11000" and Write_Control = '1');
    REG_25 <= WD when (WR = "11001" and Write_Control = '1');


    CTRL_OUT <= Control;

end architecture behave;
```

ALU

```
--32-Bit ALU Takes 2 32-bit inputs(A and B) and determines what needs to
--be done by the control signals. Outputs the respective output
--and also a branch signal 1: to branch, 0: no branch
--Jeff Grindel, Tom Demeter
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

```
entity ALU_32_Bit is
   port (A_DATA: in std_logic_vector(31 downto 0);
         B_DATA: in std_logic_vector(31 downto 0);
         CONTROL: in std_logic_vector(5 downto 0);
         BRANCH: out std_logic;                    --1 To Branch, 0: No branch
         F_DATA: out std_logic_vector(31 downto 0));

end entity ALU_32_Bit;

architecture behave of ALU_32_Bit is
begin
    F_DATA <= std_logic_vector(unsigned(A_DATA) + unsigned(B_DATA)) when (CONTROL =
"100011" OR CONTROL ="101011" or CONTROL = "100000") else --LW, SW, ADD
              A_DATA OR B_DATA when (CONTROL = "100101" OR CONTROL = "001101") else --OR,
ORI
              x"00000001" when (CONTROL = "101010" AND (A_DATA < B_DATA)) else --SLT A<B
              x"00000000" when (CONTROL = "101010" AND (A_DATA > B_DATA)) else --SLT A>B
              B_DATA when ((CONTROL = "000100" OR CONTROL = "000101") AND (A_DATA =
B_DATA)) else --BEQ
              B_DATA when ((CONTROL = "000100" OR CONTROL = "000101") AND (A_DATA /=
B_DATA)) else --BNE
              std_logic_vector(unsigned(B_DATA) srl to_integer(unsigned(A_DATA))) when
CONTROL = "000010" else --SRL
              std_logic_vector(unsigned(B_DATA) or (unsigned(A_DATA) sll 16)) when
CONTROL = "001111" else      --Imm in A: B: RT
              x"FFFFFFFF";


    BRANCH <= '1' when (CONTROL = "000100" AND (A_DATA = B_DATA)) else      --BEQ
              '1' when (CONTROL = "000101" AND (A_DATA /= B_DATA)) else     --BNE
              '0';
end architecture behave;
```

Data Memory

```
--Data_Memory: Just has 2 values for the memory one for SW and LW
--Jeff Grindel, Tom Demeter
library ieee;
use ieee.std_logic_1164.all;

entity Data_Memory is
   port (ADDR: in std_logic_vector(31 downto 0);    --32-bit Address location
         WD: in std_logic_vector(31 downto 0);      --32-bit data
         Control: in std_logic_vector(5 downto 0);  --LW for Read SW for Write
         RD: out std_logic_vector(31 downto 0));    --32-bit data at specific address,
only going to output when lw
end entity Data_Memory;

architecture behave of Data_Memory is

signal MEM_164: std_logic_vector(31 downto 0) := x"00001234";   --For lw, data will be
put into Reg 19
signal MEM_1F8: std_logic_vector(31 downto 0) := x"00001111";   --For sw, data will
change to $200
signal MEM_ERROR: std_logic_vector(31 downto 0) := x"FFFFFFFF"; --For sw, data will
change to $200

begin
    RD <= MEM_164 when (ADDR = x"00000164" and Control = "100011") else
          MEM_1F8 when (ADDR = x"000001F8" and Control = "100011") else
          Mem_ERROR;
```

```
  --MEM_164 <= WD when (ADDR = x"00000164" and Control = "101011");
    MEM_1F8 <= WD when (ADDR = x"000001F8" and Control = "101011");
end architecture behave;
```

PC Calculator

```
--PC_CALC: takes in the PC value and control siganls
--to determine if the instruction needs to jump, branch, or just
--return PC+4
--Jeff Grindel, Tom Demeter
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity PC_Calc is
   port (PC_IN: in std_logic_vector(31 downto 0);   --32-bit PC
         IMM: in std_logic_vector(31 downto 0);      --32-bit Immidiate value
         Branch: in std_logic;                       --Branch Control Signal from ALU
         Control: in std_logic_vector(5 downto 0);   --Control signal from Instruction Mem
         X_Type: in std_logic_vector(1 downto 0);    --Type of instruction: Going to be
used for Jump Instruction "01"
         PC_OUT: out std_logic_vector(31 downto 0)); --32-bit PC output
end entity PC_Calc;

architecture behave of PC_Calc is

begin

    PC_OUT <= std_logic_vector((unsigned(IMM) sll 2) + unsigned(PC_IN)) when (Branch =
'1' and (Control = "000100" or Control = "000101")) else --For branch
              std_logic_vector(unsigned(IMM) sll 2) when (X_Type = "01") else
              std_logic_vector(unsigned(PC_IN) + 4); --Default case of adding 4

end architecture behave;
```

Sign Extend 5

```
--Takes a 5-bit number and sign extends it to a 32-bit number
--Jeff Grindel, Tom Demeter
library ieee;
use ieee.std_logic_1164.all;

entity Sign_Extend_5 is
    port (INPUT: in std_logic_vector(4 downto 0);
          OUTPUT: out std_logic_vector(31 downto 0));
end entity Sign_Extend_5;

architecture behavior of Sign_Extend_5 is
begin
    OUTPUT(31 downto 8) <= x"000000";
    OUTPUT(7 downto 5) <= "000";
    OUTPUT(4 downto 0) <= INPUT;
end architecture behavior;
```

Sign Extend 16

```
--Takes a 16-bit number and sign extends it to a 32-bit number
--Jeff Grindel, Tom Demeter
library ieee;
```

```
use ieee.std_logic_1164.all;

entity Sign_Extend_16 is
    port (INPUT: in std_logic_vector(15 downto 0);
          OUTPUT: out std_logic_vector(31 downto 0));
end entity Sign_Extend_16;

architecture behavior of Sign_Extend_16 is
begin
    OUTPUT(31 downto 16) <= x"0000";
    OUTPUT(15 downto 0) <= INPUT;
end architecture behavior;
```

2 Input Mux

```
--MUX FOR 2-input 5-bit Information
--Jeff Grindel, Tom Demeter
library ieee;
use ieee.std_logic_1164.all;

entity Mux_2 is
    port (ZERO: in std_logic_vector(4 downto 0);
          ONE: in std_logic_vector(4 downto 0);
          CTRL: in std_logic_vector(1 downto 0);
          OUTPUT: out std_logic_vector(4 downto 0));
end entity Mux_2;

architecture behavior of Mux_2 is
begin
    OUTPUT <= ZERO when CTRL = "10" else     --I-TYPE, Outputs Rt
              ONE when CTRL = "00" else      --R-TYPE, Outputs Rd
              "11111";
end architecture behavior;
```

3 Input Mux

```
--MUX for 3 input 32 bit numbers
--Jeff Grindel, Tom Demeter
library ieee;
use ieee.std_logic_1164.all;

entity Mux_3 is
    port (ZERO: in std_logic_vector(31 downto 0);
          ONE: in std_logic_vector(31 downto 0);
          TWO: in std_logic_vector(31 downto 0);
          CTRL: in std_logic_vector(5 downto 0);
          OUTPUT: out std_logic_vector(31 downto 0));
end entity Mux_3;

architecture behavior of Mux_3 is
begin
    OUTPUT <= TWO when CTRL = "000010" else      --Shift Ammount
              ONE when CTRL = "001111" else      --Load Upper Imm
              ZERO;                              --Else Zero(RD1(Rs Data)
end architecture behavior;
```

First 32 Bit Mux

```
--MUX FOR 2-input 32-bit Information
```

```
--Jeff Grindel, Tom Demeter
library ieee;
use ieee.std_logic_1164.all;

entity Mux2_32 is
    port (ZERO: in std_logic_vector(31 downto 0);
          ONE: in std_logic_vector(31 downto 0);
         CTRL: in std_logic_vector(5 downto 0);
          OUTPUT: out std_logic_vector(31 downto 0));
end entity Mux2_32;

architecture behavior of Mux2_32 is
begin
    OUTPUT <= ONE when (CTRL = "100011" or CTRL = "101011" or CTRL = "001101") else     -
-I-TYPE, Outputs IMM
              ZERO;
end architecture behavior;
```

Second 32 Bit Mux

```
--MUX FOR 2-input 32-bit Information for data mem or alu out
--Jeff Grindel, Tom Demeter
library ieee;
use ieee.std_logic_1164.all;

entity Mux2_32_1 is
    port (ZERO: in std_logic_vector(31 downto 0);
          ONE: in std_logic_vector(31 downto 0);
         CTRL: in std_logic_vector(5 downto 0);
          OUTPUT: out std_logic_vector(31 downto 0));
end entity Mux2_32_1;

architecture behavior of Mux2_32_1 is
begin
    OUTPUT <= ZERO when (CTRL = "100011") else      --Just reads from data for
              ONE;
end architecture behavior;
```

Overall Test Bench Program

```
--Combination of Instruction Memory, Instruction Decode,
--Register, and ALU and misc. MUX's and Sign extenders, Data Memory, and pc generation
--Final version of the Datapath of the 32-bit processor
--Jeff Grindel, Tom Demeter
library ieee;
use ieee.std_logic_1164.all;

entity a1_5 is
    port (aPC: in std_logic_vector(31 downto 0);        --PC Input
          aPC_OUT: out std_logic_vector(31 downto 0);
          aF_DATA: out std_logic_vector(31 downto 0);
          aTEMP_TYPE: out std_logic_vector(1 downto 0));
end entity a1_5;

architecture behave of a1_5 is

--Temp Signals used as connections between components
signal TEMP_INST: std_logic_vector(31 downto 0);
signal TEMP_RS: std_logic_vector(4 downto 0);
signal TEMP_RT: std_logic_vector(4 downto 0);
```

```
signal TEMP_RD: std_logic_vector(4 downto 0);
signal TEMP_IMM: std_logic_vector(15 downto 0);
signal TEMP_SHMT: std_logic_vector(4 downto 0);
signal TEMP_FUNCT: std_logic_vector(5 downto 0);
signal TEMP_TYPE: std_logic_vector(1 downto 0);
signal TEMP_M2OUT: std_logic_vector(4 downto 0);
signal TEMP_SE5OUT: std_logic_vector(31 downto 0);
signal TEMP_SE16OUT: std_logic_vector(31 downto 0);
signal TEMP_RD1: std_logic_vector(31 downto 0);
signal TEMP_RD2: std_logic_vector(31 downto 0);
signal TEMP_CTRL_OUT: std_logic_vector(5 downto 0);
signal TEMP_A_DATA: std_logic_vector(31 downto 0);
signal TEMP_B_DATA: std_logic_vector(31 downto 0);
signal TEMP_BRANCH: std_logic;
signal TEMP_F_DATA: std_logic_vector(31 downto 0);
signal TEMP_MEM_RD: std_logic_vector(31 downto 0);
signal TEMP_Reg_Write: std_logic_vector(0 downto 0);
signal TEMP_Reg_Value: std_logic_vector(31 downto 0);

component Instruction_Memory
    port (PC: in std_logic_vector(31 downto 0);              --32-bit instruction
            INSTRUCTION: out std_logic_vector(31 downto 0)); --32-bit
instruction(hardcoded in)
end component;

component Instruction_Decode
    port (INST: in std_logic_vector(31 downto 0);         --32-bit instruction
            RS: out std_logic_vector(4 downto 0);        --5-bit RS output
            RT: out std_logic_vector(4 downto 0);        --5-bit RT output
            RD: out std_logic_vector(4 downto 0);        --5-bit RD output(R-Type only)
            IMM: out std_logic_vector(15 downto 0);      --5-bit IMM output(I-Type only)
            SHMT: out std_logic_vector(10 downto 6);     --5-bit Shift Amount (R-Type
only)
            FUNCT: out std_logic_vector(5 downto 0);     --6-bit Function Code (R-Type
only)
            X_TYPE: out std_logic_vector(1 downto 0));   --2-bit output to determine type:
00:R-Type, 01:J-Type, 10:I-Type, 11: error
end component;

component Register_Data
     port (RR1: in std_logic_vector(4 downto 0);         --5-bit Read Reg. 1
         RR2: in std_logic_vector(4 downto 0);           --5-bit Read Reg. 2
         WR: in std_logic_vector(4 downto 0);            --5-bit Write Register
         WD: in std_logic_vector(31 downto 0);           --32-bit Write Data
         Control: in std_logic_vector(5 downto 0);       --6-bit Opcode/Function
         RD1: out std_logic_vector(31 downto 0);         --32-bit output1
         RD2: out std_logic_vector(31 downto 0);         --32-bit output2
         CTRL_OUT: out std_logic_vector(5 downto 0));    --Control Output to pass through
the control values for alu operation
end component;

component ALU_32_Bit
   port (A_DATA: in std_logic_vector(31 downto 0);       --32-bit A input
         B_DATA: in std_logic_vector(31 downto 0);       --32-bit B input
         CONTROL: in std_logic_vector(5 downto 0);       --6-bit control
         BRANCH: out std_logic;                          --1 for BRANCH, 0 for not BRANCH
         F_DATA: out std_logic_vector(31 downto 0));     --32-bit ALU output
end component;

component Data_Memory
   port (ADDR: in std_logic_vector(31 downto 0);         --32-bit Address location
         WD: in std_logic_vector(31 downto 0);           --32-bit data
         Control: in std_logic_vector(5 downto 0);       --LW for Read SW for Write
```

```
        RD: out std_logic_vector(31 downto 0));        --32-bit data at specific
address, only going to output when lw
end component;

component Mux_2
    port (ZERO: in std_logic_vector(4 downto 0);       --Zero Input for 5-bit mux
          ONE: in std_logic_vector(4 downto 0);        --One input for 5-bit mux
          CTRL: in std_logic_vector(1 downto 0);       --Control signal
          OUTPUT: out std_logic_vector(4 downto 0));   --5-bit output
end component;

component Mux2_32                                      --Mux for B_Data of ALU
    port (ZERO: in std_logic_vector(31 downto 0);      --Zero Input for 32-bit mux
          ONE: in std_logic_vector(31 downto 0);       --One Input for 32-bit mux
          CTRL: in std_logic_vector(5 downto 0);       --Control signal
          OUTPUT: out std_logic_vector(31 downto 0));  --32-bit output
end component;

component Mux2_32_1                                    --Mux for Write Data
    port (ZERO: in std_logic_vector(31 downto 0);      --Zero Input for 32-bit mux
          ONE: in std_logic_vector(31 downto 0);       --One Input for 32-bit mux
          CTRL: in std_logic_vector(5 downto 0);       --Control signal
          OUTPUT: out std_logic_vector(31 downto 0));  --32-bit output
end component;

component Mux_3                                        --Mux for A_Data of ALU
    port (ZERO: in std_logic_vector(31 downto 0);      --Zero input for 32-bit mux
          ONE: in std_logic_vector(31 downto 0);       --One input for 32-bit mux
          TWO: in std_logic_vector(31 downto 0);       --Two inptu for 32-bit mux
          CTRL: in std_logic_vector(5 downto 0);       --Controla signla
          OUTPUT: out std_logic_vector(31 downto 0));  --32-bit output
end component;

component Sign_Extend_5                                --Takes 5 bit number and
truncattes with 0
    port (INPUT: in std_logic_vector(4 downto 0);      --5-bit Input
          OUTPUT: out std_logic_vector(31 downto 0));  --32-bit ouput
end component;

component Sign_Extend_16                               --Takes 16 bit number and
trancates with 0
    port (INPUT: in std_logic_vector(15 downto 0);     --16-bit Input
          OUTPUT: out std_logic_vector(31 downto 0));  --32-bit output
end component;

component PC_Calc
   port (PC_IN: in std_logic_vector(31 downto 0);      --32-bit PC
         IMM: in std_logic_vector(31 downto 0);        --32-bit Immidiate value
         Branch: in std_logic;                         --Branch Control Signal from ALU
         Control: in std_logic_vector(5 downto 0);     --Control signal from Instruction
Mem
         X_Type: in std_logic_vector(1 downto 0);      --Type of instruction: Going to
be used for Jump Instruction "01"
         PC_OUT: out std_logic_vector(31 downto 0));   --32-bit PC output
end component;
begin

Instruction_Mem: Instruction_Memory port map(PC => aPC,
                                     INSTRUCTION => TEMP_INST);

Decode: Instruction_Decode port map(INST => TEMP_INST,
                                    RS => TEMP_RS,
                                    RT => TEMP_RT,
```

```
                                        RD => TEMP_RD,
                                        IMM => TEMP_IMM,
                                        SHMT => TEMP_SHMT,
                                        FUNCT => TEMP_FUNCT,
                                        X_TYPE => TEMP_TYPE);

Rd_Select: Mux_2 port map(ZERO => TEMP_RT,
                          ONE => TEMP_RD,
                          CTRL => TEMP_TYPE,
                          OUTPUT => TEMP_M2OUT);

Reg_Data: Register_Data port map(RR1 => TEMP_RS,
                                 RR2 => TEMP_RT,
                                 WR => TEMP_M2OUT,
                                 WD => TEMP_Reg_Value,
                                 Control => TEMP_FUNCT,
                                 RD1 => TEMP_RD1,
                                 RD2 => TEMP_RD2,
                                 CTRL_OUT => TEMP_CTRL_OUT);

Shmt_Extend: Sign_Extend_5 port map(INPUT => TEMP_SHMT,
                                    OUTPUT => TEMP_SE5OUT);

Imm_Extend: Sign_Extend_16 port map(INPUT => TEMP_IMM,
                                    OUTPUT => TEMP_SE16OUT);


A_Data_Select: Mux_3 port map(ZERO => TEMP_RD1,
                          ONE => TEMP_SE16OUT,
                          TWO => TEMP_SE5OUT,
                          CTRL => TEMP_FUNCT,
                          OUTPUT => TEMP_A_DATA);

B_Data_Select: Mux2_32 port map(ZERO => TEMP_RD2,
                            ONE =>  TEMP_SE16OUT,
                            CTRL => TEMP_FUNCT,
                            OUTPUT => TEMP_B_DATA);

ALU: ALU_32_Bit port map(A_DATA => TEMP_A_DATA,
                             B_DATA => TEMP_B_DATA,
                             CONTROL => TEMP_FUNCT,
                             BRANCH => TEMP_BRANCH,
                             F_DATA => TEMP_F_DATA);

Data_Mem: Data_Memory port map(ADDR => TEMP_F_DATA,
                               WD => TEMP_RD2,
                               Control => TEMP_FUNCT,
                               RD => TEMP_MEM_RD);


Write_DataMux: Mux2_32_1 port map(ZERO => TEMP_MEM_RD,
                              ONE => TEMP_F_DATA,
                              CTRL => TEMP_FUNCT,
                              OUTPUT => TEMP_Reg_Value);

PC: PC_Calc port map(PC_IN =>aPC,
                             IMM => TEMP_SE16OUT,
                             Branch => TEMP_BRANCH,
                             Control => TEMP_FUNCT,
                             X_TYPE => TEMP_TYPE,
                             PC_OUT => aPC_OUT);

     aF_DATA <= TEMP_F_DATA;
```

```
     aTEMP_TYPE <= TEMP_TYPE;
end architecture behave;
```