CS 6133

Computer Architecture- I

Midterm Report

-Nimisha Limaye

N12375934

List of attachments:

1. For Fibonacci question-
   mips_ss_v2_fibo.qar
   use mips_ss_v2_fibo_iram.mif
   use mips_ss_v2_fibo_dram.mif

2. For Indexed mode question-
   mips_ss_v2_index.qar
   use mips_ss_v2_sample_iram.mif
   use mips_ss_v2_sample_dram.mif

3. For Autoincrement and Autodecrement mode question-
   mips_ss_v2_inc_dec.qar
   use mips_ss_v2_inc_iram.mif
   use mips_ss_v2_inc_dram.mif

PART I:
MIPS Registers

1. **The data memory in MIPS CPU is running at the same speed as the CPU so it could fetch data in one clock cycle, just like the register file (RF).**

   a. **What is purpose of having a 32 registers? Could RF be reduced to 16 or 8? (2.5)**

MIPS contains 32 registers. The register set is as shown below.

Table 6.1 MIPS register set

Name	Number	Use
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2–3	procedure return values
$a0-$a3	4–7	procedure arguments
$t0-$t7	8–15	temporary variables
$s0-$s7	16–23	saved variables
$t8-$t9	24–25	temporary variables
$k0-$k1	26–27	operating system (OS) temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	procedure return address

The image is taken from Digital Design and Computer Architecture book.

The purpose of having 32 registers is that it allows for faster program execution. The use of each register or set of registers is shown in the fig above. Therefore, more the registers, faster is the program execution. Accessing operands from memory is slow. Hence, having more registers allows the compiler to allocate more variables to registers than to memory. Register files can be reduced to 16 or 8 based on the use of them for the application. But this reduction is only applicable to upcoming architecture. We cannot reduce the RF when number of registers are already defined.

   b. **What is impact and benefits of reducing the size of the RF? (2.5)**

The ultimate goal is to run the system at higher speed. One contributing factor is the time taken by instructions to access the operands. This time can be reduced if instructions are able to access operands quickly and thereby increasing the speed of the system. The challenge to this is that operands take a lot of time to retrieve from memory. The larger the memory, the more time operands will take to retrieve. Hence, most architectures set a small number of registers for holding operands. In conclusion, the lesser the number of registers, the faster the operands can be accessed.

2. **Why is instruction memory read only? Are there any advantages in enabling write access to instruction memory? (5)**

There is only one advantage of enabling write access to instruction memory and that is the ability of self modifying or self writing the code.

However, there are many disadvantages associated with this access, and the only advantage can be achieved using other methods or approaches.

Disadvantages of enabling write access to instruction memory are as follows:

1. Security Risk: Hackers can easily take advantage of this ability and tamper the sequence or data.
2. Bugs: If a bug enters through this write access enable, the program can have unexpected outcomes and debugging can get troublesome.
3. CPU complex: Including this functionality makes CPU design complex and hence CPU programmers would rarely want to use this functionality.
4. Solution: Suppose the design wants to be modular, we can always implement the on the fly code during run time.

In conclusion, although there might be instances where we require on the fly instructions, but providing write access to instruction memory is not the solution because of the above mentioned reasons.

PART II: Understanding Simple CPU

Understanding Simple CPU (mips ss v2) is important to complete the rest of the midterm project. In this part you will write an assembly programs in binary and execute it in mips ss v2 CPU running in DE0-Nano.

Fibonacci Number Generator: (20)

1. **Write a program that generates Fibonacci numbers up to Fib-40 and could backwards to 0 decrementing by 1.**

IRAM.mif

```
WIDTH=32;
DEPTH=1024;

ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
        000 :  10001100000000011000000000000000;          LW R3,[R0]
        001 :  10001100000001000000000000000100;          LW R4,[R1]
        002 :  10000000000001010000000000101001;          ADDI R5,#41
        003 :  10000000000001100000000000000001;          ADDI R6,#1
        004 :  10000000000001110000000000000000;          ADDI R7,#0
        005 :  00000000011001000001100000100000;LOOP1:   ADD R3,R3,R4
        006 :  00000000011001000010000000100000;          ADD R4,R3,R4
        007 :  10101100000000011000000000000000;          SW R3,[R0]
        008 :  00000000101001100010100000100010;          SUB R5,R5,R6
        009 :  00010000101001110000000000000010;          BEQ R5,R7,LOOP2
        00A :  10101100000001000000000000000100;          SW R4,[R1]
        00B :  00000000101001100010100000100010;          SUB R5,R5,R6
        00C :  00000000101001100100000000101010;          SLT R8,R5,R6
        00D :  00010001000001111111111111110111;          BEQ R8,R7,LOOP1
        00E :  00000000100000110010000000100010;LOOP2:   SUB R4,R4,R3
        00F :  00000000011001000001100000100010;          SUB R3,R3,R4
STARTS DECREMENTING:
        010 :  00000000011001110001000000100000;LOOP:    ADD R2,R3,R7
        011 :  00000000100000110010000000100010;          SUB R4,R4,R3
        012 :  00000000011001000001100000100010;          SUB R3,R3,R4
        013 :  00000000010001100001000000100010;LOOP3    SUB R2,R2,R6
        014 :  00000000100000100101000000101010;          SLT R10,R4,R2
        015 :  00010001010001101111111111111101;          BEQ R10,R6,LOOP3
        016 :  10101100000001000000000000000000;          SW R2,[R0]
```

```
        017 :   00000000010001100001000000100010;LOOP4:  SUB R2,R2,R6
        018 :   00000000011000100101000000101010;         SLT R10,R3,R2
        019 :   00010001010001101111111111111101;         BEQ R10,R6,LOOP4
        01A :   10101100000001000000000000000100;         SW R2,[R1]
        01B :   00000000111010010101100000101010;         SLT R11,R7,R9
        01C :   00010001011001101111111111110110;         BEQ R11,R7,LOOP
        [01D..3FF] :  00000000000000000000000000000000;
END;
```

DRAM.mif:

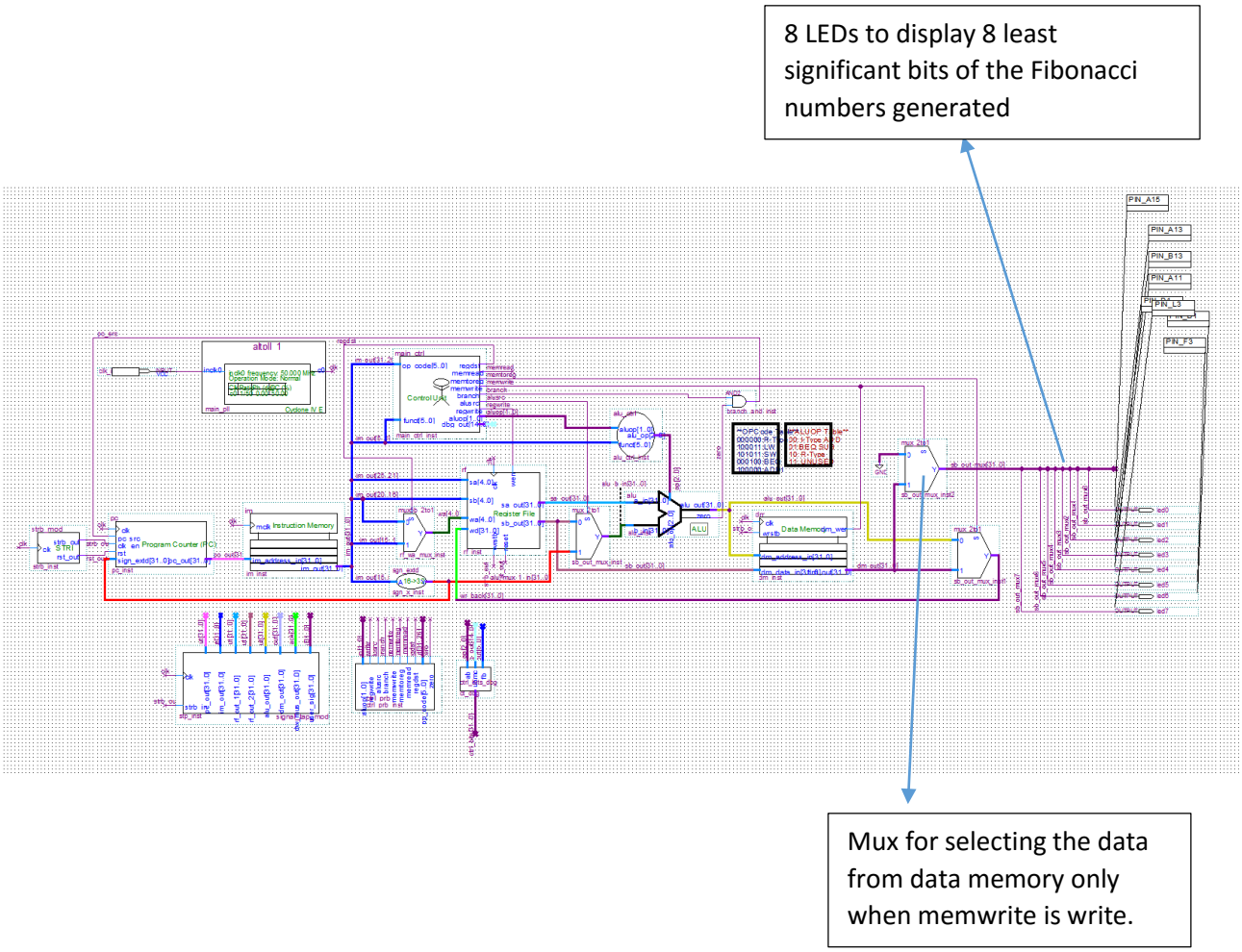
```
WIDTH=32;
DEPTH=1024;

ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN
        000 :   00000000000000000000000000000000; //First input=0
        001 :   00000000000000000000000000000001; //Second input=1
        [002..3FF] :  00000000000000000000000000000000;
END;
```

2. **Your program should display 8 least significant bits of each number you generate in the above part through the LEDs.**

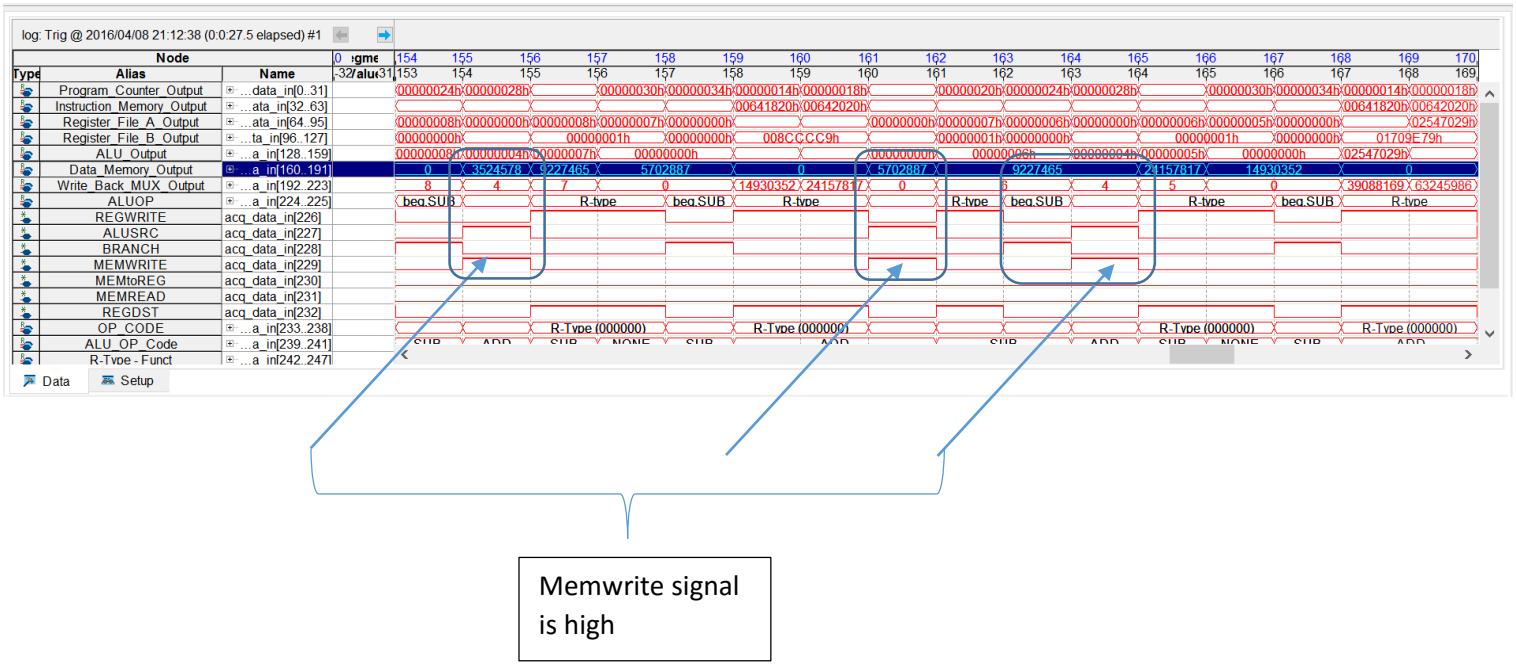
Additions made to the schematic of mips_ss_v2 are shown in the fig. below.



As can be seen from the schematic above, the LEDS are connected to a multiplexor which gets its inputs from dm_out and ground. The select line is memwrite signal from main_ctrl. Operation of this is as follows: When the memwrite signal is high, the data memory output is given to the LEDs and when memwrite is low the LEDs are off.

This behavior can be confirmed with the help of the signal capture screenshot below or look for attached document- mips_ss_v2_signaltap_megafunction_3_fibo.txt. The text file contains all the signals from the signal tap and their values. Column 5 is the data memory and column is the memwrite signal. You will expect to see correct output from data memory column when memwite column contains a 1.

.



Memwrite signal is high

3. While counting back you should blink the LEDs whenever the count value is equal to a Fibonacci number.

The logic of counting back by decrementing by 1 and comparing the result with the already computed Fibonacci numbers, stores the value in Data Memory.

```
STARTS DECREMENTING:
        010 :  00000000011001110001000000100000;LOOP:   ADD R2,R3,R7
        011 :  00000000100000110010000000100010;        SUB R4,R4,R3
        012 :  00000000011001000001100000100010;        SUB R3,R3,R4
        013 :  00000000010001100001000000100010;LOOP3   SUB R2,R2,R6
        014 :  00000000100000100101000000101010;        SLT R10,R4,R2
        015 :  00010001010001101111111111111101;        BEQ R10,R6,LOOP3
        016 :  10101100000001000000000000000000;        SW R2,[R0]
        017 :  00000000010001100001000000100010;LOOP4:  SUB R2,R2,R6
        018 :  00000000011000100101000000101010;        SLT R10,R3,R2
        019 :  00010001010001101111111111111101;        BEQ R10,R6,LOOP4
        01A :  10101100000001000000000000000100;        SW R2,[R1]
        01B :  00000000111010010101100000101010;        SLT R11,R7,R9
        01C :  00010001011001101111111111110110;        BEQ R11,R7,LOOP
```

The logic of using multiplexor as explained above, confirms blinking of LEDs for this part.

PART III: Advanced Addressing modes

As implemented, mips ss v2 only supports register and immediate addressing modes. For this part add the following addressing modes to mips 22 v2:

1 Indexed (30)

Instruction Format:

31:26	25:21	20:16	15:11	10:6	5:0
Opcode	R1	R2	R3	00000	funct
110000	00001	00010	00111	00000	000000

Implementation details:

The logic behind implementing indexed addressing mode is to know which signals from control unit affect the flow.

Regdst should be high,

Memtoreg must be 1,

Memread must be 1,

Alusrc must be 0.

IRAM:

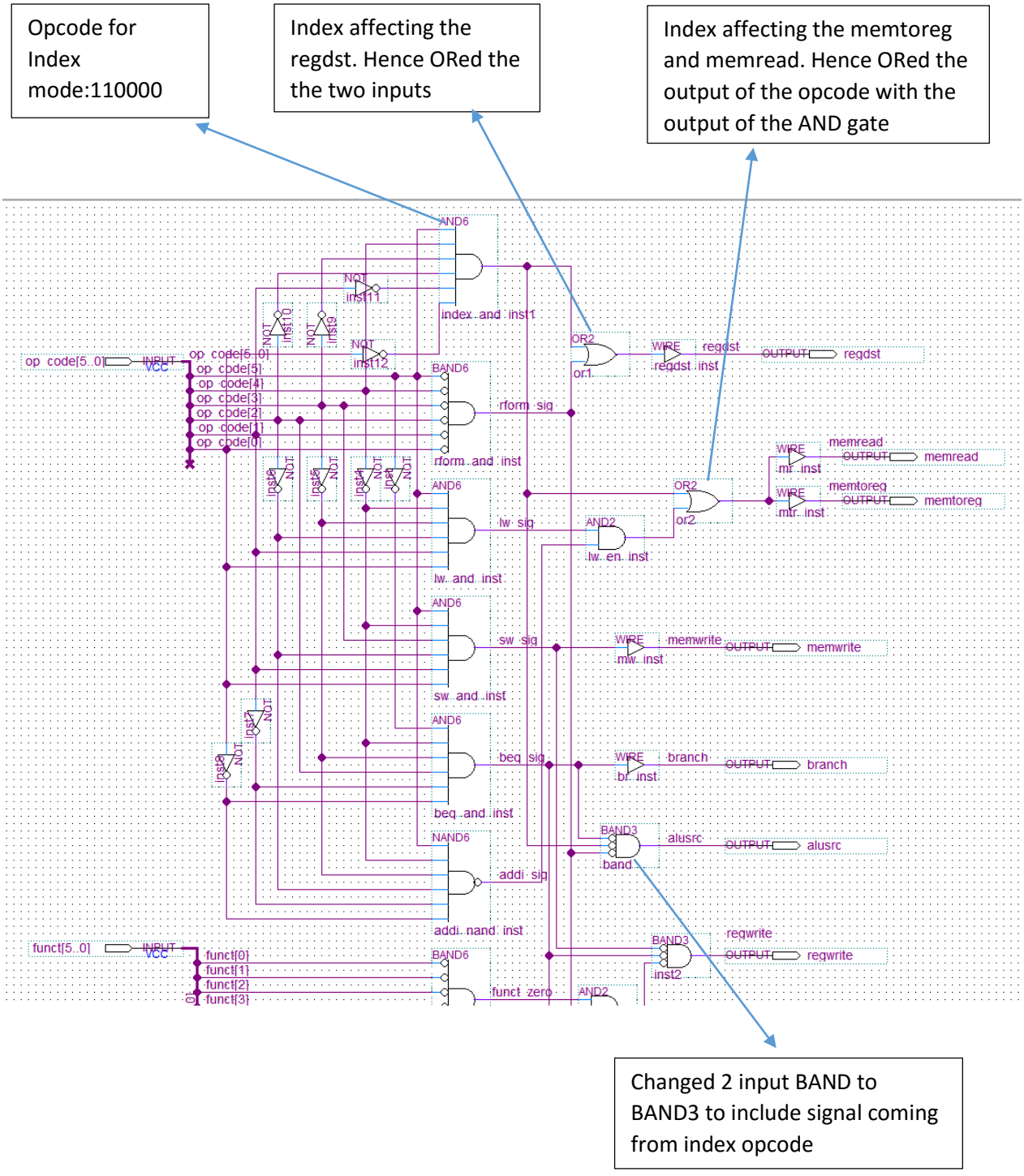
```
WIDTH=32;
DEPTH=1024;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;
CONTENT BEGIN
        000  :  10000000000000010000000000001000; //ADDI R1,#8
        001  :  10000000000000010000000000001000; //ADDI R2,#8
        002  :  11000000001000100011100000000000; //INDEX R7,R1,R2
        003  :  10000000000001000000000000000001; // ADDI R4,#1
        004  :  10001100100001010000000000000000; //LW R5,[R4]
        [005..3FF]  :  00000000000000000000000000000000;
END;
```

DRAM:

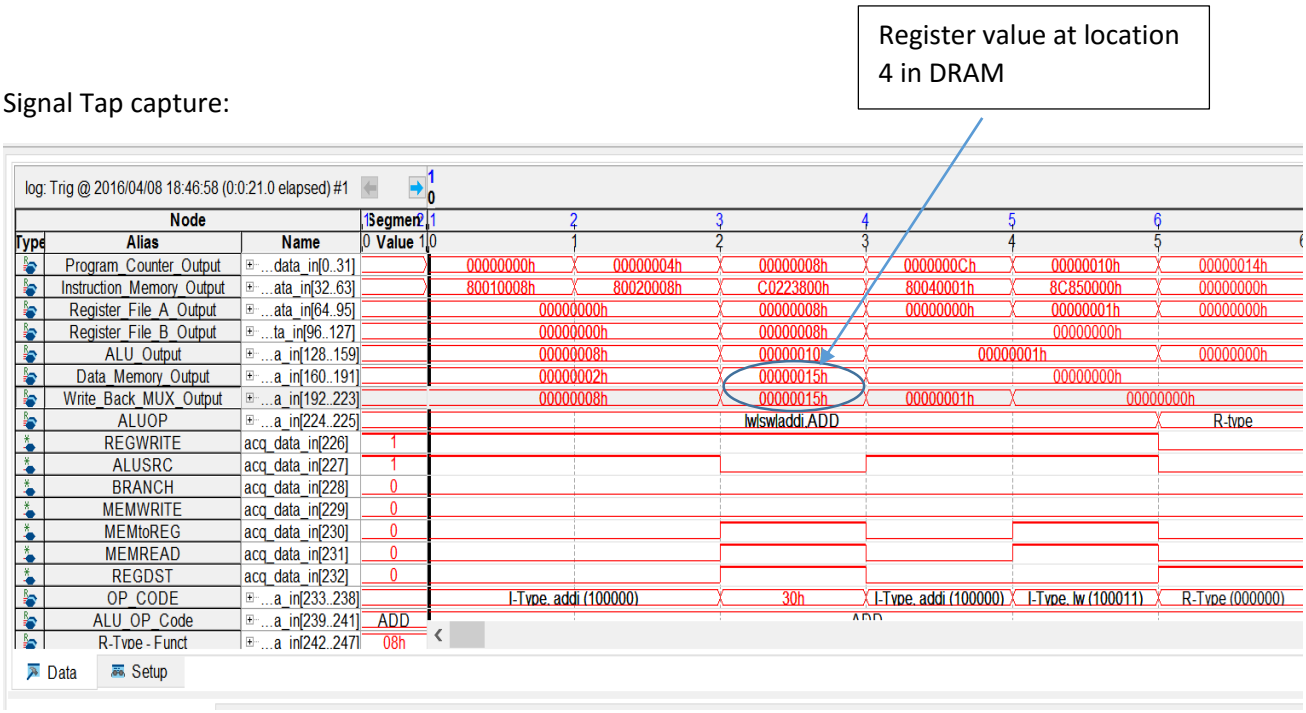
```
WIDTH=32;
DEPTH=1024;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;
CONTENT BEGIN
        000 :  00000000000000000000000000000000;
        001 :  00000000000000000000000000000001;
        002 :  00000000000000000000000000000010;
        003 :  00000000000000000000000000011100;
        004 :  00000000000000000000000000010101; //value 15H or 21 in decimal
        [005..007] :  00000000000000000000000000000000;
        008 :  00000000000000000000000000000110;
        009 :  00000000000000000000000000000001;
        00A :  00000000000000000000000000000010;
        00B :  00000000000000000000000000100011;
        00C :  00000000000000000000000000000111;
        [00D..3FF] :  00000000000000000000000000000000;
END;
```

Modification in the schematic:

Hence to include these additions to the signals, I made changes in the main_ctrl to include indexed addressing mode.



Signal Tap capture:



2 Autoincrement and Autodecrement (40)

Your report should at least contain, but not limited to, the following: Instruction formats, implementation details, modification to control unit and signal, detailed description of any new modules you implemented.

Instruction Format for Autoincrement:

31:26	25:21	20:16	15:11	10:6	5:0
Opcode	R1	R2	R3	00000	funct
000111	00010	00001	00000	00000	000000

example

Instruction Format for Autodecrement:

31:26	25:21	20:16	15:11	10:6	5:0
Opcode	R1	R2	R3	00000	funct
100111	00010	00001	00000	00000	000000

example

Implementation Details:

Increment and Decrement opcodes affect the following signals:

Alusrc, regdst, regwrite, memwrite and memtoreg. I have also added 2 multiplexors and one demultiplexor. The fig below explains how the increment and decrement opcode outputs affect these three blocks.

The only difference between operation of increment and decrement modes is the lpm add/sub block.

IRAM:

WIDTH=32;

DEPTH=1024;

ADDRESS_RADIX=HEX;

DATA_RADIX=BIN;

CONTENT BEGIN

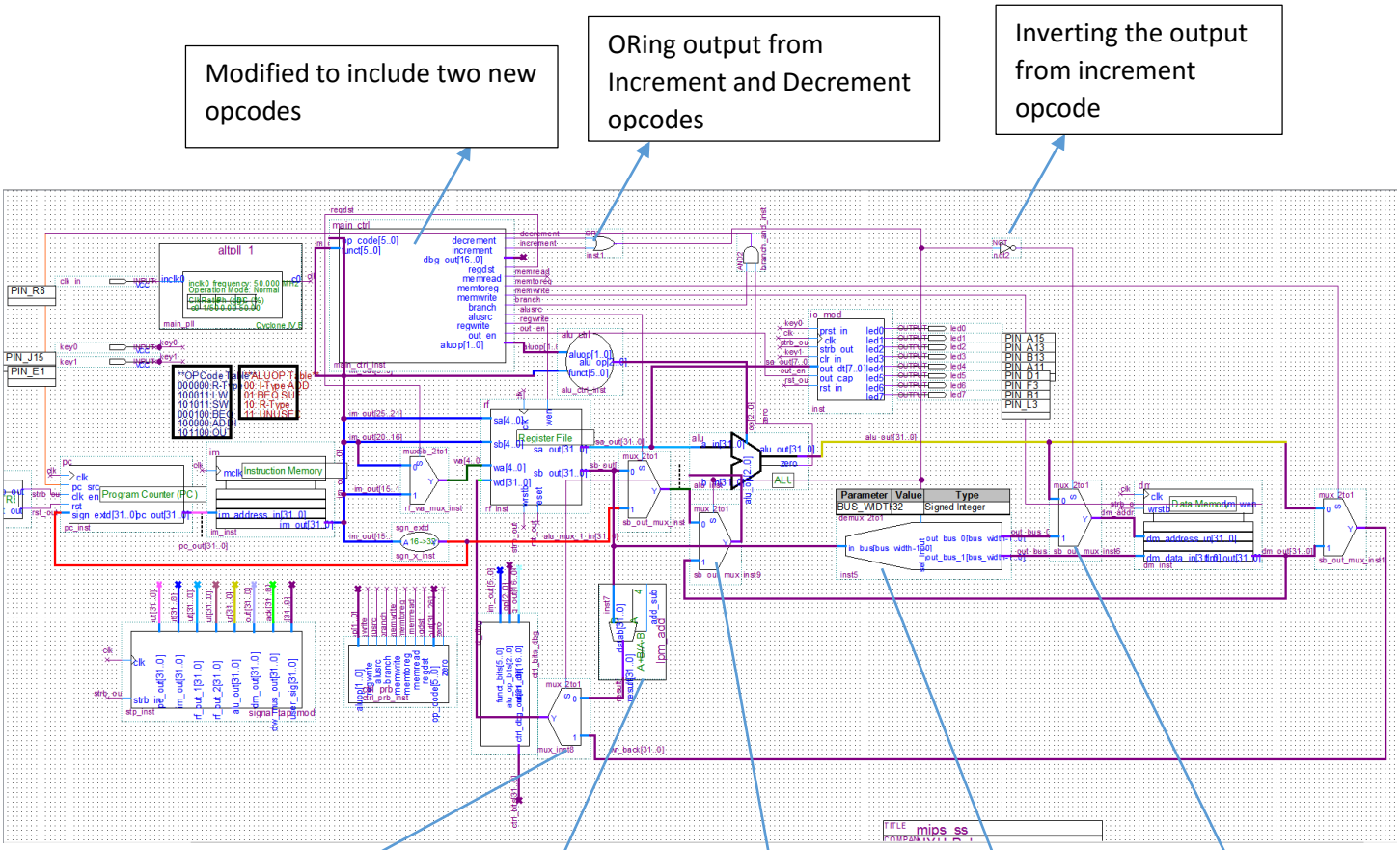
```
        000 :  10000000000000010000000000000100; // ADDI R1,#4
        001 :  10000000000000010000000000001000; //ADDI R2,$8
        002 :  00011100010000010000000000000000; //INC R1,R2
        003 :  10011100010000010000000000000000; //DEC R1,R2
        [004..3FF] :  00000000000000000000000000000000;
END;
```

DRAM:

```
WIDTH=32;
DEPTH=1024;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;
CONTENT BEGIN
        000 :  00000000000000000000000000000001;
        001 :  00000000000000000000000000000101;
        002 :  00000000000000000000000000001101;
        003 :  00000000000000000000000000000011;
        004 :  00000000000000000000000000010101;
        005 :  00000000000000000000000000000110;
        006 :  00000000000000000000000000001001;
        007 :  00000000000000000000000000100000;
        008 :  00000000000000000000000001010101;
        009 :  00000000000000000000000000000001;
        [00A..3FF] :  00000000000000000000000000000000;
END;
```

Modifications in the circuits:

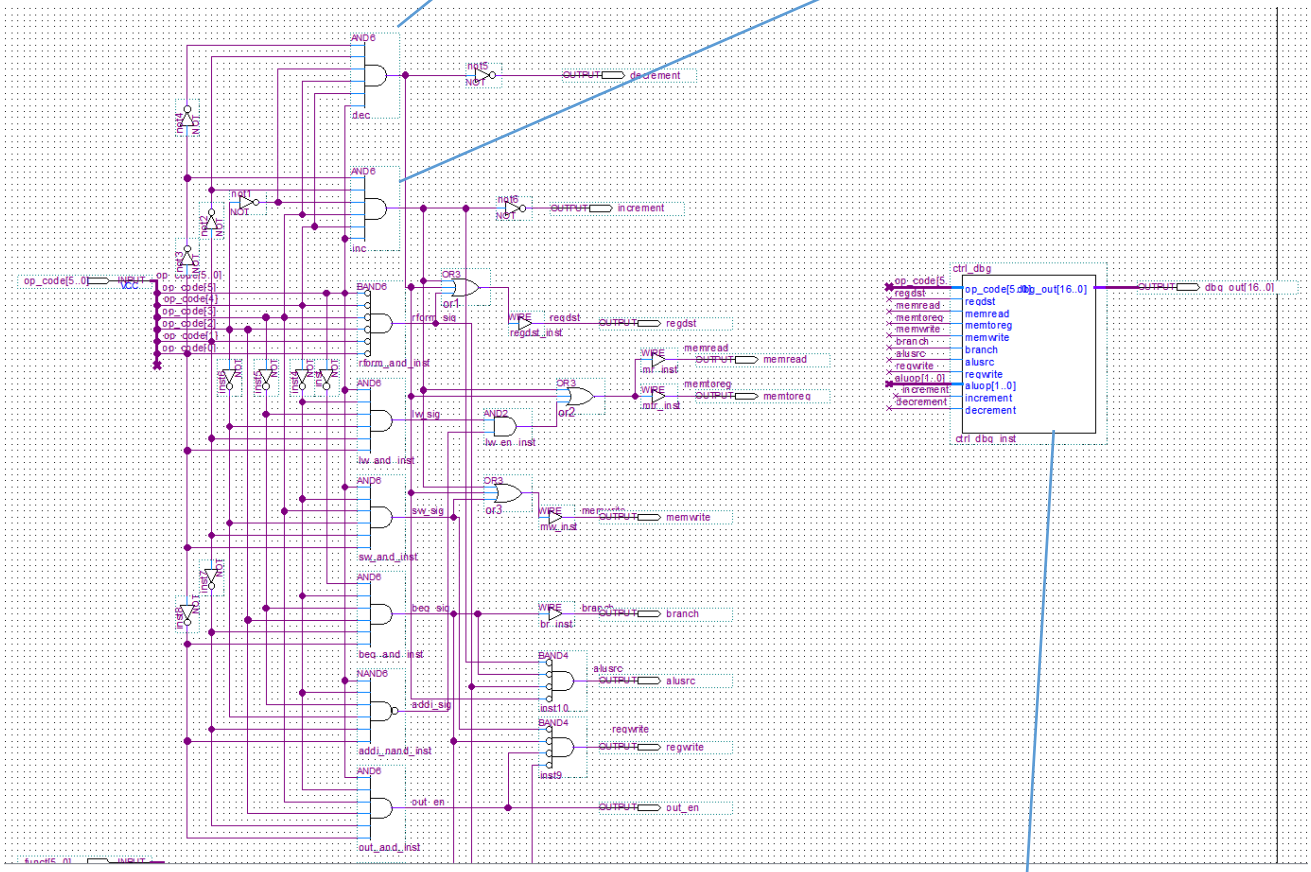
Schematic for autoincrement and autodecrement modes is as follows:



I have also changed the vhdl codes to match the correct sizes of dbg_out in main_ctrl block. The size changes due to the inclusion of two new opcodes viz. increment and decrement.

Increment opcode:000111

Decrement:100111



Updated the symbol to include increment and decrement opcodes