Computer Architecture I

CS6133

Midterm

PROJECT REPORT

Sagar Panchal

N18051845

sap586

PART 1: .

MIPS REGISTERS

1. **Why is the instruction and data memory separated in MIPS SS v2 CPU?**

MIPS CPU follows Harvard architecture, i.e. it has separate room for Data Memory and Instruction Memory, rather than Von Neumann architecture i.e. keeping a single memory for Data and Instructions.

Each component in a CPU can be accessed only once in a single clock cycle. We cannot access a single memory multiple times in a single clock cycle. So, in order to access both the Instructions and Data, MIPS uses different memory for both of them, namely Instruction Memory and Data Memory.

Note that, modern CPU designers opt to use separate Instruction and Data Memories and thus making them more Harvard-like architecture.

2. **Explain the advantages and disadvantages of separate Instruction and Data Memory.**

Advantages:

- There are two or more data paths which allows simultaneous access to data and instructions in a single cycle.
- The CPU design is less complex as one set of address and data bus is used to perform read and write operations on data memory and another set of address and data bus is used to perform instruction fetch.
- Efficient pipelining i.e. operand and instructions can be fetched at a same time.
- It is safer because it does not allow random data to use as the instruction due to erroneous coding.
- There are separate code and data spaces, ex. address 0000 for instruction is separate from address 0000 for data. This separation prevents conflicts between accessing data and instructions.

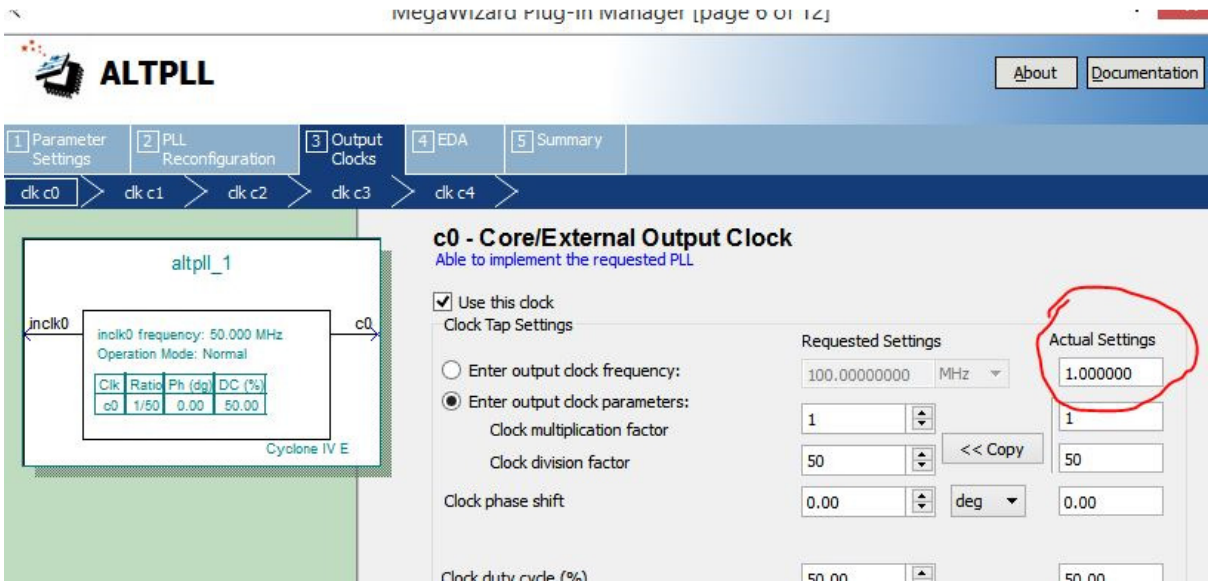
Disadvantages:

- It becomes expensive to use two different memories for storing Instructions and Data.
- Physical design becomes larger.

3. MIPS_SS_v2's clock rate.

1. What is PLL's clock rate or frequency ?

The PLL's clock rate can be seen by opening the ALTPLLC module in the design. The frequency is 1MHz as highlighted below.



2. What is the highest clock rate that could be achieved without any modifications?

	Fmax
1	40.72 MHz
2	47.31 MHz

The maximum frequency is shown in the screen shot above.

3. What limits the clock rate in the design?

Some factors affecting the clock frequency of the CPU includes the propagation delays of various modules in the CPU and temperature of operation of the device.

PART 2: .

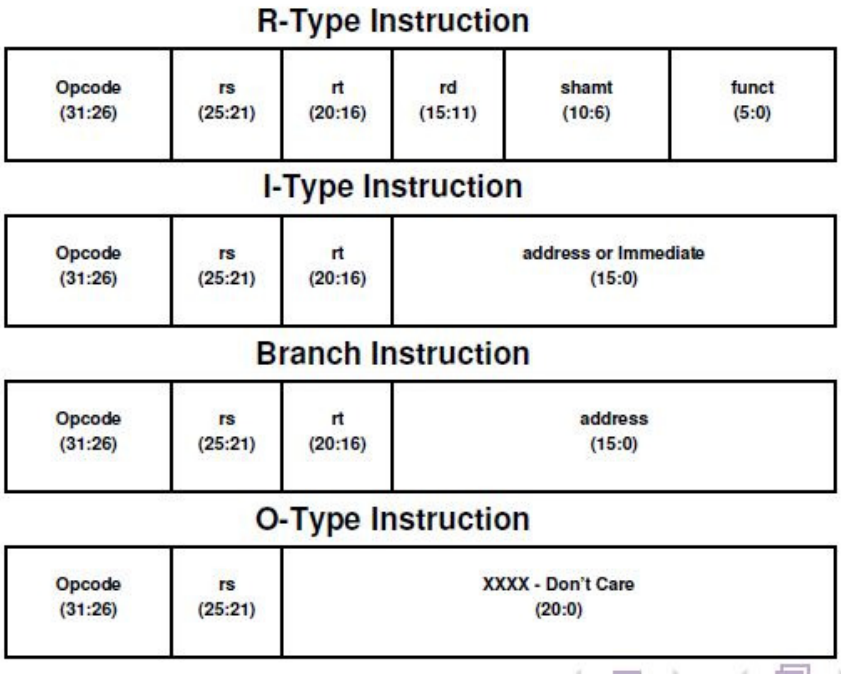
UNDERSTAINDING SIMPLE CPU

1. **Write a program that generates Triangular numbers up to n=20 and count backwards to 0.**

The triangular number counts the objects that can form an equilateral triangle. In part 2 of the project, we will be generating first 20 triangular numbers and the series would be like as shown below:

0, 1, 3, 5, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153, 171 & 190.

The image below shows the types of instructions in MIPS architecture. We will be using some instructions from all the below shown types to generate the triangular numbers.



In all, we will be using 6 different types of instructions. The instructions includes *add, addi, sub, ld, beq* and *out.*

The add and sub instructions used are of R-type, addi, beq and ld used are of I-type instructions and out is the O-type of instruction.

The following piece shows the assembly code for the triangular number generator.

```
000     ld  r1, 0
001     ld  r2, 4
002     ld  r3, 8
003     ld  r4, 0
004     ld  r5, 4
005     beq r3, r2, 0004
006     add r1, r2, r1
007     out r1
008     addi r2, r2, #1
009     beq r3, r3, FFFB
00A     out r4;
00B     sub r2, r2, r5
00C     sub r1, r1, r2
00D     beq r1, r4, 0003
00E     out r1
00F     sub r2, r2, r5
010     beq r1, r1, FFFB
011     out r4;
-----------------------------------
```

The first 5 instructions in the assembly code stores the predetermined value in the registers r1 through r5. The binary code for the above 5 lines is shown below.

000 : 100011,00000,00001,00000 00000 000000;

001 : 100011,00000,00010,00000 00000 000100;

002 : 100011,00000,00011,00000 00000 001000;

003 : 100011,00000,00100,00000 00000 000000;

004 : 100011,00000,00101,00000 00000 000100;

Note that the opcode for ld instruction is 100011. ld instruction will store the data in the memory location, addressed by 16 least significant bits of the instruction in binary form to the register addressed by the rt field. So by executing above statements, we are loading registers r1 through t5 with the data stored at the memory location 00, 04, 08, 00 and 04 respectively. Now if we look at the data memory, it has the following data stored inside the data memory.

Note that the data memory has the data stored in hexadecimal form and this is defined at the beginning of the dram.mif file as follows.

```
WIDTH=32;
DEPTH=1024;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
```

This will assign register r1=0, r2=1, r3=14h(20 in decimal, upper limit), r4=0 and r5=1.

```
000 : 00000000;
001 : 00000001;
002 : 00000014;
003 : 00000000;

END;
```

Then we begin our logic to move upwards in the triangular series. We begin with comparing r3 and r2, and if they are not equal, we will continue to add r2 to r1 and show r1 on the LEDs as output. We will also keep adding 1 to r2 using addi function. The assembly code for the above logic is shown below in the binary form.

```
005 :   000100,00011,00010,00000 00000 000100;
006 :   000000,00010,00001,00001,00000,100000;
007 :   101100,00001,00000 00000 00000 000000;
008 :   100000,00010,00010,00000 00000 000001;
009 :   000100,00011,00011,11111 11111 111011;
```

We will use add function to perform r1=r1+r2. Then after showing the r1 out on LEDs, we will add 1 to the value inside r2. We will keep doing this until the point arises when r2 reaches the value equal to the upper limit, which is 20. We keep this loop in repeating mode by using beq, to compare r3 with r3. This creates the condition similar to while(1), or an infinitely satisfying condition. I gave the offset by which we want to jump equal to FFFB, which is equal to 2's complement of -5. Note that while the instruction is executed, the value of PC is already added to the offset, to point the next instruction. So while calculating the offset, we need to calculate from the position of next instruction. Now, once the r2 reaches the upper limit, the condition to go out of the loop is matched, and thus we performed the jump by 4 instructions. This jump will bring to the PC to the instruction where only LEDs are made to blink. On the next cycle, the logic of subtraction. But,

before starting the decrement cycle, we need to subtract 1 from the value of r2 as it was added in the last cycle while incrementing the value. This is done by including 1 extra sub instruction before starting the actual decrementing operation.

The decrement logic when converted to binary form will look something like the following code.

```
00C  :   000000,00001,00010,00001,00000 100010;
00D  :   000100,00001,00100,00000 00000 000011;
00E  :   101100,00001,00000 00000 00000 000000;
00F  :   000000,00010,00101,00010,00000 100010;
010  :   000100,00001,00001,11111 11111 111011;
```

We begin the logic by subtracting 1 from the result. Then, we compare the resulting value with 0, which is store in register r4. Then, after displaying the result on LEDs, we subtract 1 from r2. We repeat this loop in until r1 becomes equal to 0, and once it becomes 0, we display the blinking LEDs before ending our program. This is done by the final line of the code as shown below.

```
011  :   101100,00100,00000 00000 00000 000000;
```

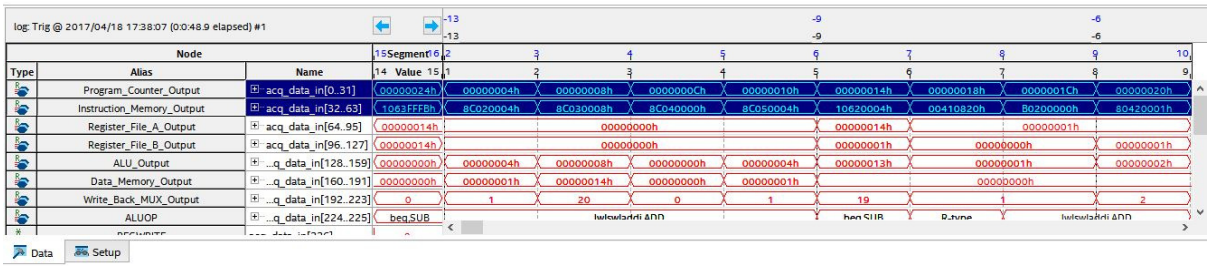
2. **Your program should display 8 least significant bits of each number you generate in the above part through LEDs.**

In the above code, first 20 triangular numbers are shown in series. There is a special instruction *out* in MIPS_ss_v2 CPU that is defined to show the output on the LEDs. The out function has the 6 bit opcode of 101100 and it has only one argument, the address of the register that is to be shown as output on LEDs. Rest of the other bits are not used and so they does not affect the way that *out* instruction works. We can notice from the code above that we have our calculated results stored in r1 register, and we are using *out* instruction to show that on LEDs. Note that the LEDs are able to show the output in binary form of any result.

3. **While counting back you should blink the LEDs whenever the count reaches 0 or 20.**

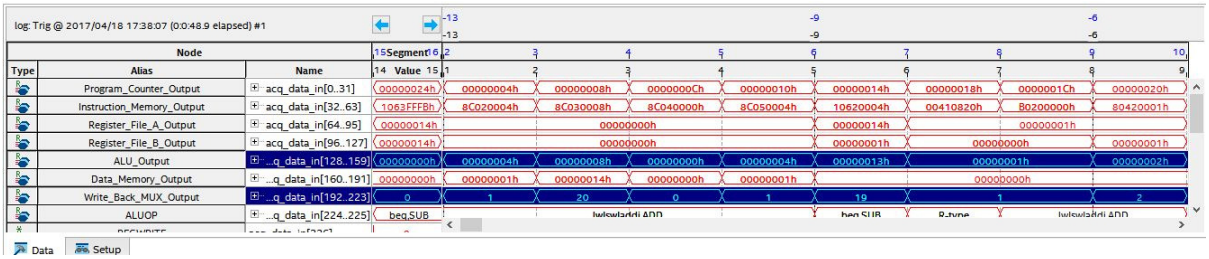
The blink of LEDs are achieved when there is the value ‘0’ in the registered that is used to show the value on output. We have used r4 register to show the blinking

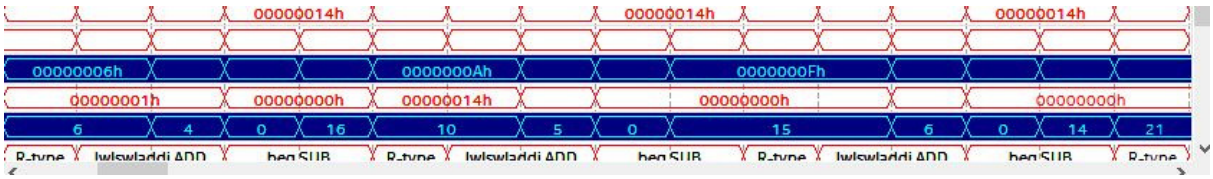
LEDs when i) our counter reaches the upper limit and ii) out counter reaches to zero again. We have just used the instruction *out r4* in both the cases and as r4 stores zero always, we are able to show the blinking LEDs when one of the two conditions are achieved.

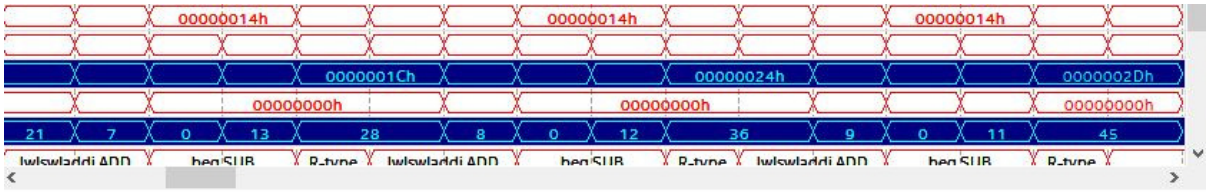


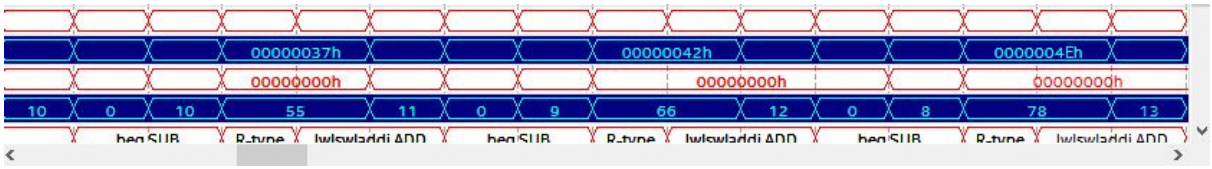
The above snap shot shows the signal tap for out triangular number generator. We can see the first row showing the value of Program Counter, which is incremented by 4 with every cycle. Right below Program Counter row, we can see the value of the instruction memory. It shows the current instruction being executed by the CPU.

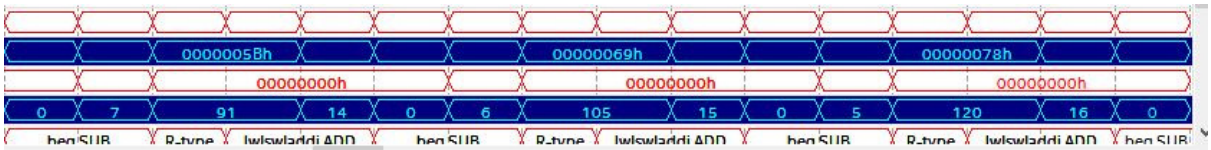
We can see the outputs as the calculated values shown in the triangular series in the signal tap. The below following snapshots were taken from the signal taps for the up and down count cycles.



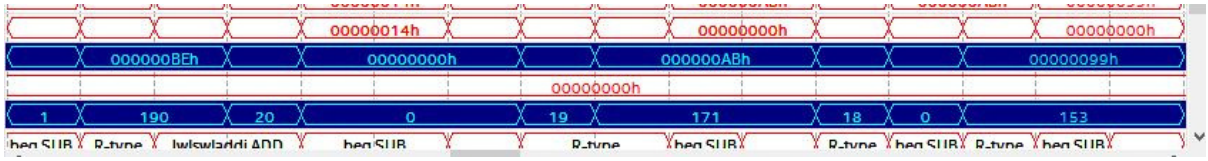


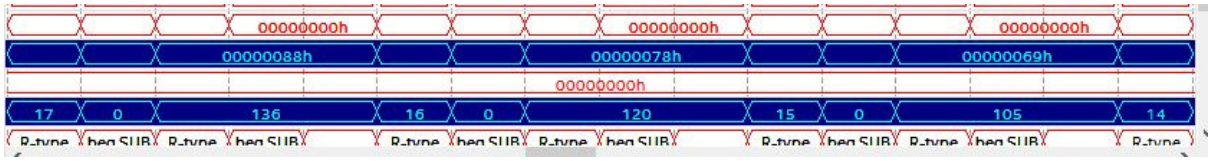


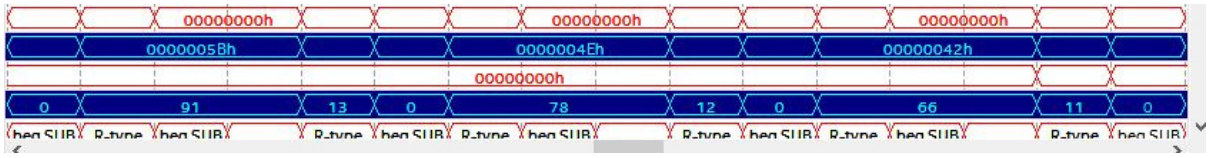


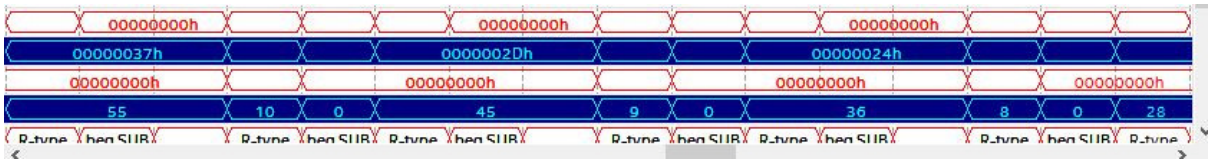

0000005Bh 00000069h 00000078h
00000000h 00000000h 00000000h
0 7 91 14 0 6 105 15 0 5 120 16 0
beq SUB R-type lwlswladdi ADD beq SUB R-type lwlswladdi ADD beq SUB R-type lwlswladdi ADD beq SUB

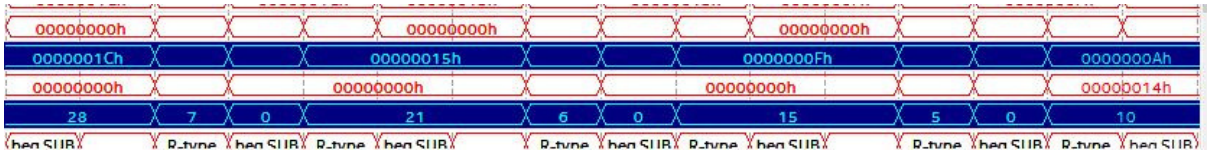
00000088h 00000099h 000000ABh
00000000h
4 136 17 0 3 153 18 0 2 171 19 0 1
beq SUB R-type lwlswladdi ADD beq SUB R-type lwlswladdi ADD beq SUB R-type lwlswladdi ADD beq SUB

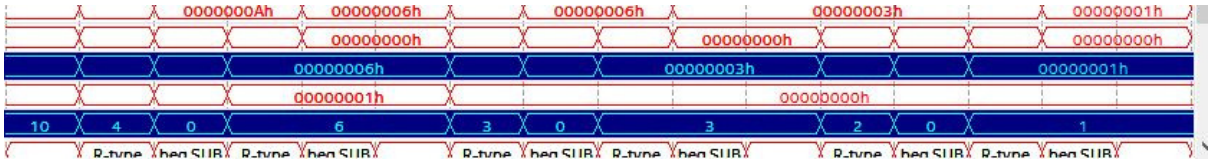

00000014h 00000000h 00000000h
000000BEh 00000000h 000000ABh 00000099h
00000000h
1 190 20 0 19 171 18 0 153
beq SUB R-type lwlswladdi ADD beq SUB R-type beq SUB R-type beq SUB R-type beq SUB


00000000h 00000000h 00000000h
00000088h 00000078h 00000069h
00000000h
17 0 136 16 0 120 15 0 105 14
R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type


00000000h 00000000h 00000000h
0000005Bh 0000004Eh 00000042h
00000000h
0 91 13 0 78 12 0 66 11 0
beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type


00000000h 00000000h 00000000h
00000037h 0000002Dh 00000024h
00000000h 00000000h 00000000h 00000000h
55 10 0 45 9 0 36 8 0 28
R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type


00000000h 00000000h 00000000h
0000001Ch 00000015h 0000000Fh 0000000Ah
00000000h 00000000h 00000000h 00000014h
28 7 0 21 6 0 15 5 0 10
beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB


0000000Ah 00000006h 00000006h 00000003h 00000001h
00000000h 00000000h 00000000h
00000006h 00000003h 00000001h
00000001h 00000000h
10 4 0 6 3 0 3 2 0 1
R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB

00000006h 00000003h 00000001h 00000001h 00000000h
00000000h 00000000h 00000001h 00000000h
00000003h 00000001h 00000000h
00000000h
3 0 3 2 0 1 0
R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB R-type beq SUB

PART 3: .

ADVANCED ADDRESSING MODES

Add following addressing modes to mips_ss_v2:

- **Displacement**
- **Register Indirect**

For third part, we have to implement 2 addressing modes in the MIPS CPU mips_ss_v2. Displacement and Register Indirect. Register Indirect addressing mode is defined by the following assembly code:

```
addri   rs, rt;
```

The above instruction will give the following output:

rt = rt + [rs]

This means that the operand in register rt will be added to another operand, which is pointed by the memory location, stored in the register rs. This is represented by showing square brackets around rs.

Displacement instruction is same as register indirect except one change. In displacement, an additional offset is added to the address stored inside the register rs. The offset is included as 16 bit immediate in the code. The assembly code for the displacement instruction is shown below:

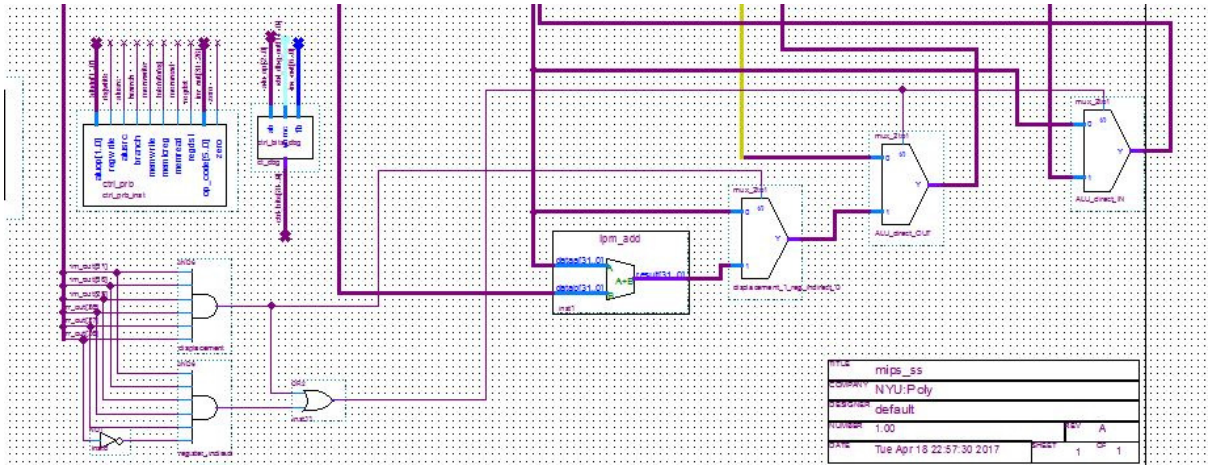
```
addd    rs, rt, offset;
```

The above instruction will give the following result:

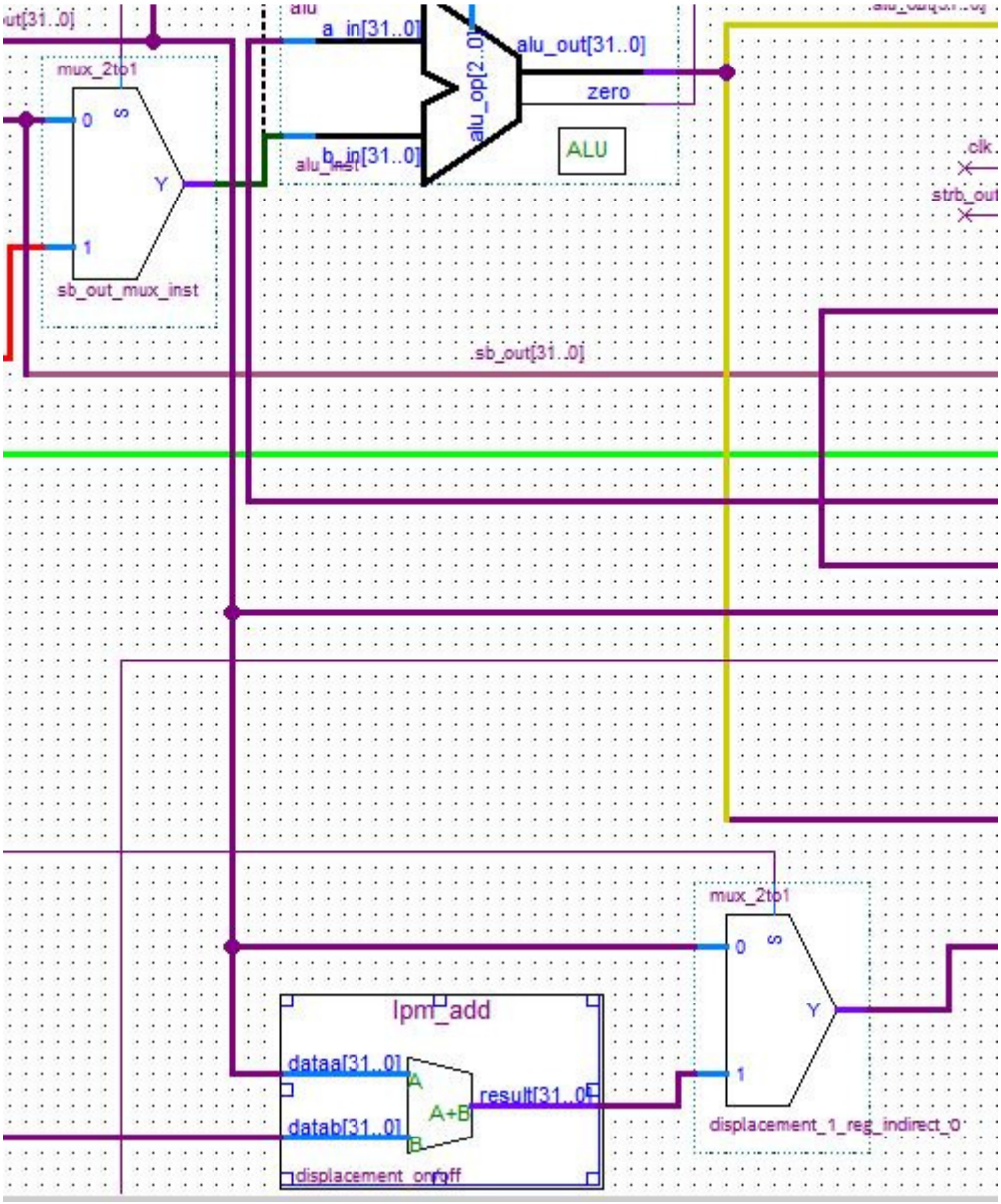
rt = rt + [rs + offset]

We can see that 'rs + offset' is stored inside the square brackets, which means that the addresses stored in the register rs will be added to the offset mentioned in the code, and the data in the resulting address will be added to the data stored in the register rt.

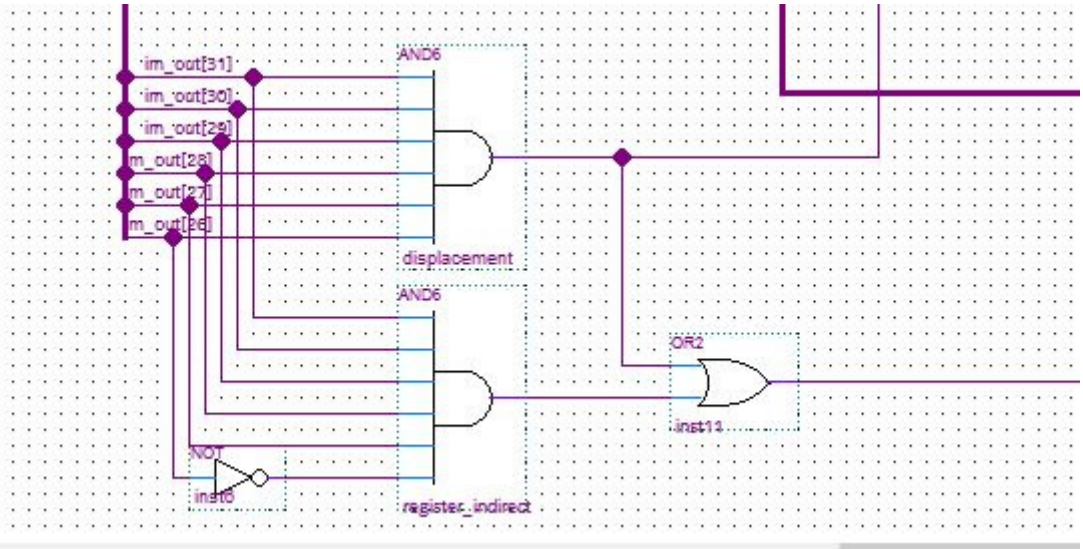
To implement the above two addressing modes, I have edited the design to some extent. The over view of the new design is shown below.



We can see that I have modified the data path of rs. The rs will either directly to data memory to point an address in the data memory (in case of register indirect) or it will be added to the offset represented by 16 bits in the code.

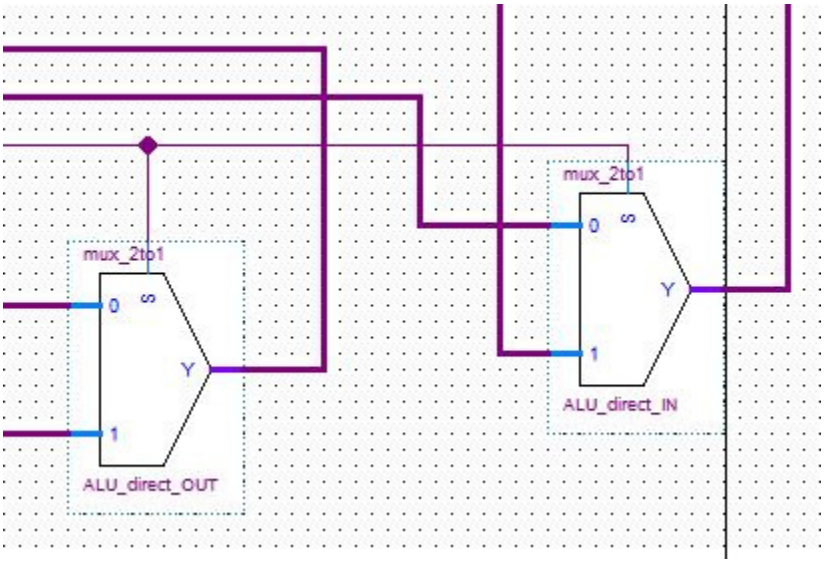


The addition is achieved by the addition module named ‘displacement_on/off’. The selection between the two addressing modes is done by implementing a 2x1 mux named ‘displacement_1_reg_indirect_0’. The name itself gives the idea of operation of the mux. When the control signal is 0, it means that the mode selected is register indirect and when it is 1, it means that the selected mode is displacement. We can see that the control signal for the above MUX is being controlled directly by the mode of opcode.

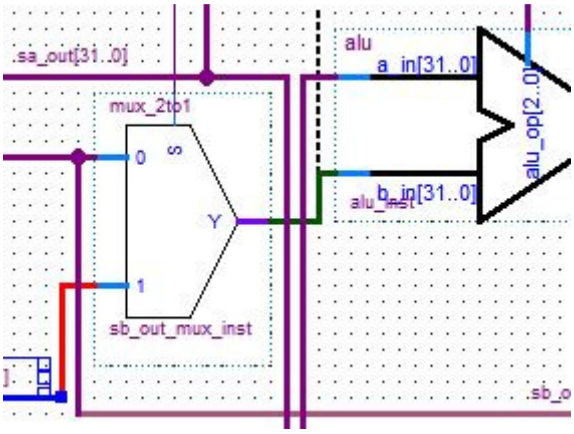


The opcode for the register indirect is defined as ‘111111’ and the opcode for displacement id defined as ‘111110’. So, the above mux is given the controlled signal of ‘1’ only when the addressing modes selected is displacement.

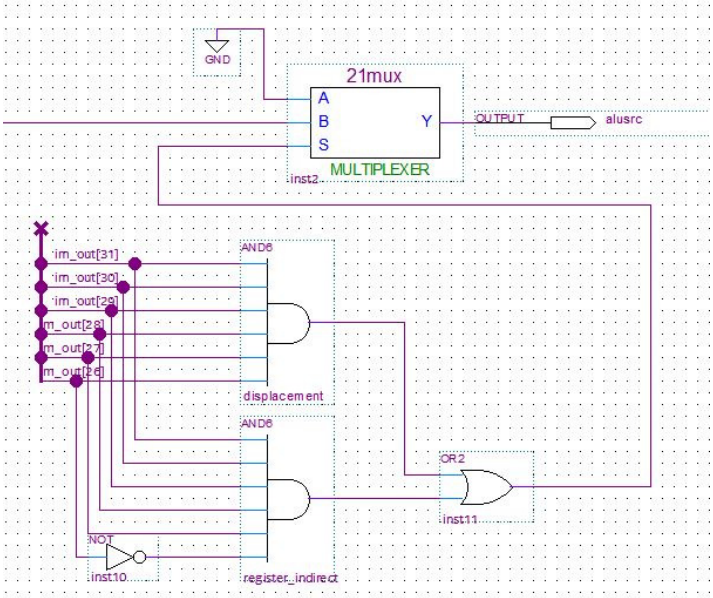
The resultant value is given to data memory and the selection between the above found result and ALU result is selected by implementing a MUX known as ‘ALU_direct_out’. Further, after getting the operand stored in the given data memory, it is sent to the ALU as one of the operand to perform the addition function with the first operand stored in the register rt. The choice of whether we want to send the rs register as it is to ALU or we want to send the data at the address pointed by the register is done by using two 2x1 MUXs, namely ALU_direct_out and ALU_direct_in. The Control signal of both of these MUXs are controlled by opcode directly and the value of the control signal is 1 only when either of the 2 newly implemented addressing modes are selected.



One problem of the design till now is that if we give an immediate value as the offset (in case of displacement addressing mode), the immediate value will be directly transferred as the second argument of the ALU unit.



This is being controlled by the control signal kown as ALUsrc which is generated as ‘1’ whenever we give immediate address in the code. To solve this problem, I did some modifications in the design of control unit as shown below. The generation of ALU_src signal is given a control case when either displacement or register indirect is selected. Whenever either of the two newly added addressing modes is selected, the ALU_src signal will become ‘0’. This will force the sb_out_mux_inst mux to allow the data inside ‘rt’ register to be to be transferred as the second input to ALU.

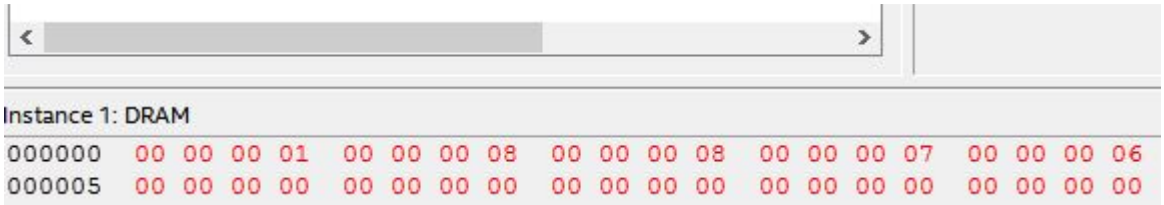


As the above alterations are shown to be done in the control unit, we can do all the changes in the control unit. This can give a better schematic to the CPU design.

Testing the newly implemented design:

- **REGISTER INDIRECT:**

Data memory:



The above screen shot shows the contents of Data memory. We will be testing the *addri* instruction for 3 different data, at the locations 08h, 0Ch and 10h.

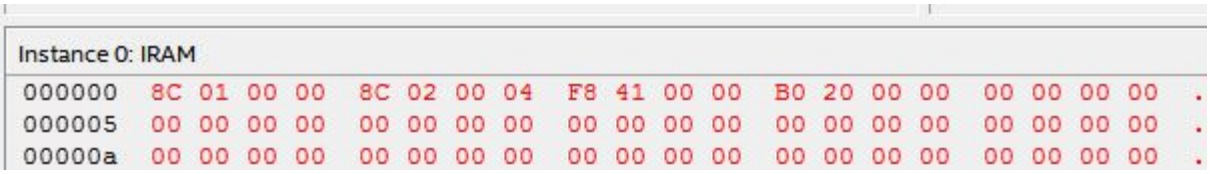
Instruction Memory:

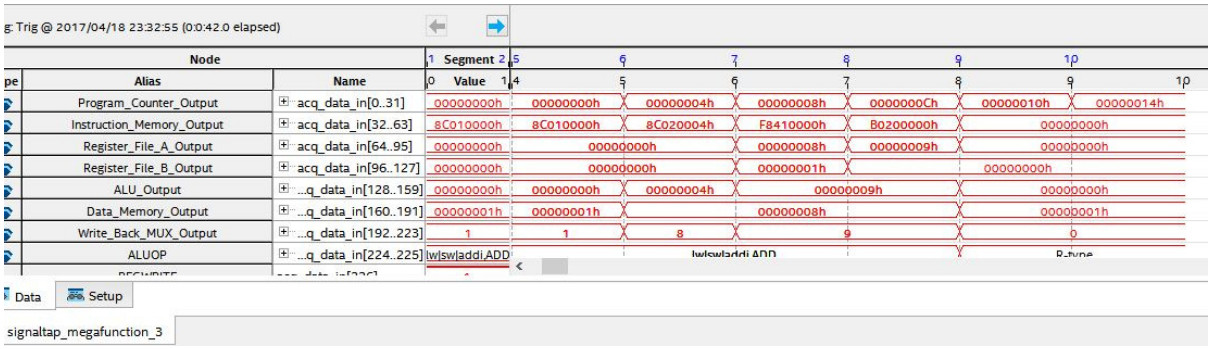
ld r1, 00; //r1=01 (contents of memory location 00)

ld r2, 04; //r2=08 (contents of memory location 04)

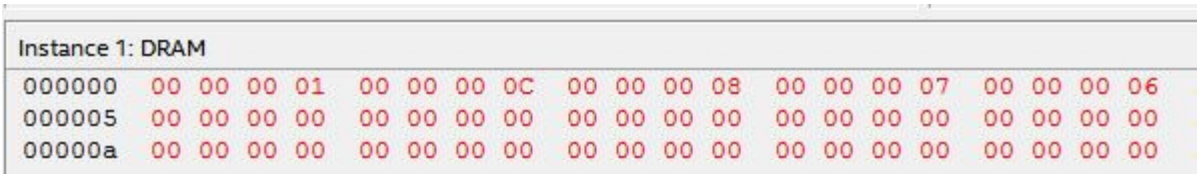
addri r2, r1; //r1=r1+[r2] (square brackets shows the memory contents)

out r1; //shows r1 on LEDs as output

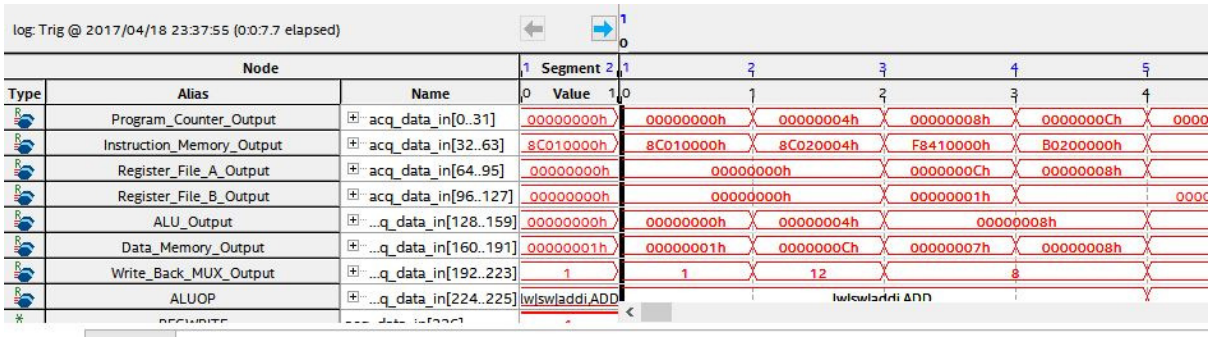




The contents of register r2 is 8 and the contents of 8th memory location is 8. So, the output will be equal to adding 1 to 8, which gives 9 as shown above.

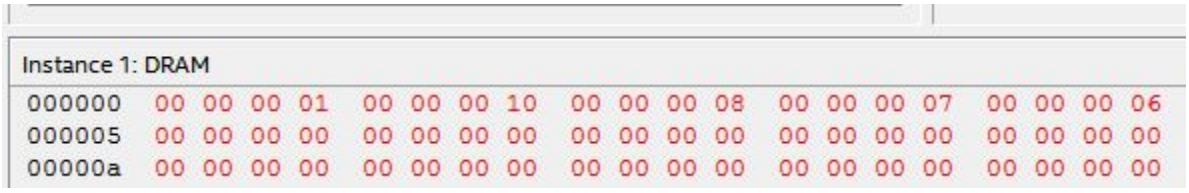


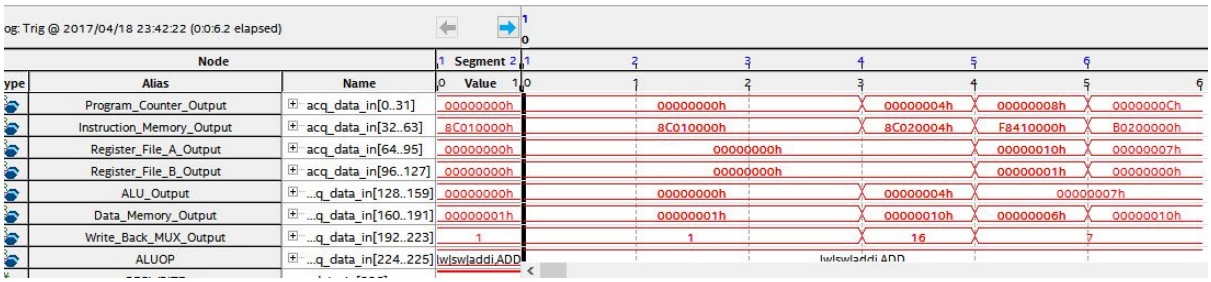
This time, we modified the value of DRAM's 4th memory location to '0C'. This will save 'C' in the register r2. C in r2 represents the 12th memory location and thus, we will be adding 07 (from 12th memory location) to 1.



The resulting value, 8 is shown on the LEDs and is shown in the signal-tap above.

Finally, we tested the design for 3rd case. This time, we stored the hex value '10' in the fourth memory location. This will store the decimal value 16 in the register r2. This will point to the memory location 16 which stores the value 06. Thus, we add 1 to the value 6 and the final result is shown on LEDs as well as captured in the signal-tap as shown above.



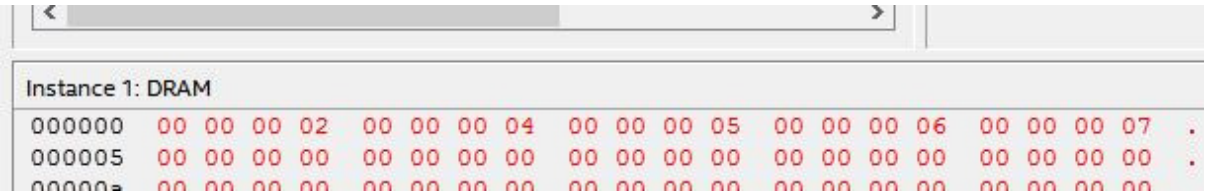


Alias	Name	Value
Program_Counter_Output	acq_data_in[0..31]	00000000h
Instruction_Memory_Output	acq_data_in[32..63]	8C010000h
Register_File_A_Output	acq_data_in[64..95]	00000000h
Register_File_B_Output	acq_data_in[96..127]	00000000h
ALU_Output	...q_data_in[128..159]	00000000h
Data_Memory_Output	...q_data_in[160..191]	00000001h
Write_Back_MUX_Output	...q_data_in[192..223]	1
ALUOP	...q_data_in[224..225]	lwlswladdi,ADD

- **DISPLACEMENT:**

The Displacement is added by including the offset in the immediate field in the instruction. We used the following codes for the test of the Displacement addressing mode.

Data Memory:



The above screen shot shows the DRAM, and we will be testing the displacement addressing mode for three different cases.

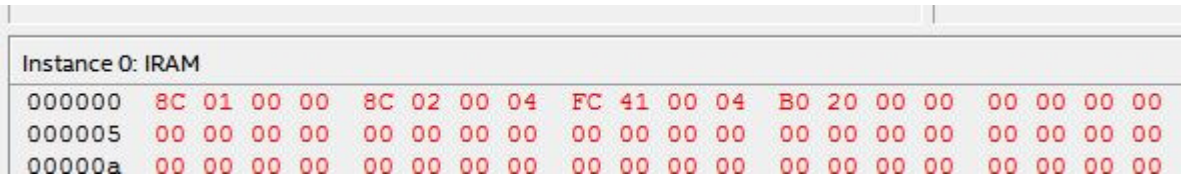
Instruction Memory:

ld r1, 00; //r1=02

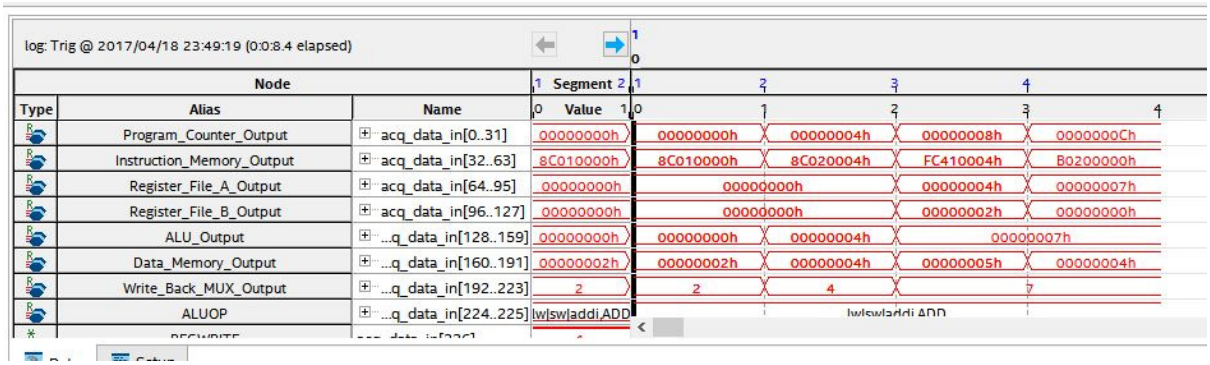
ld r2, 04; //r2=04

addd r2, r1, 04; //r1=r1+[r2+4] (adding an offset of 4 to the address in r1)

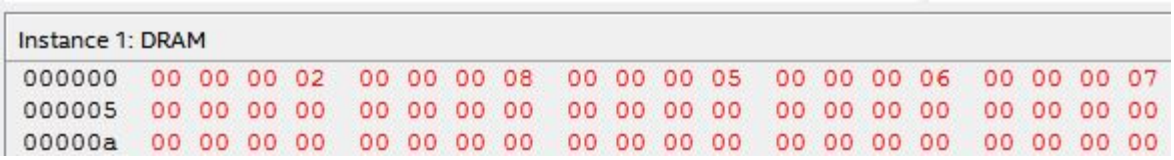
out r1; //show r1 on LEDs

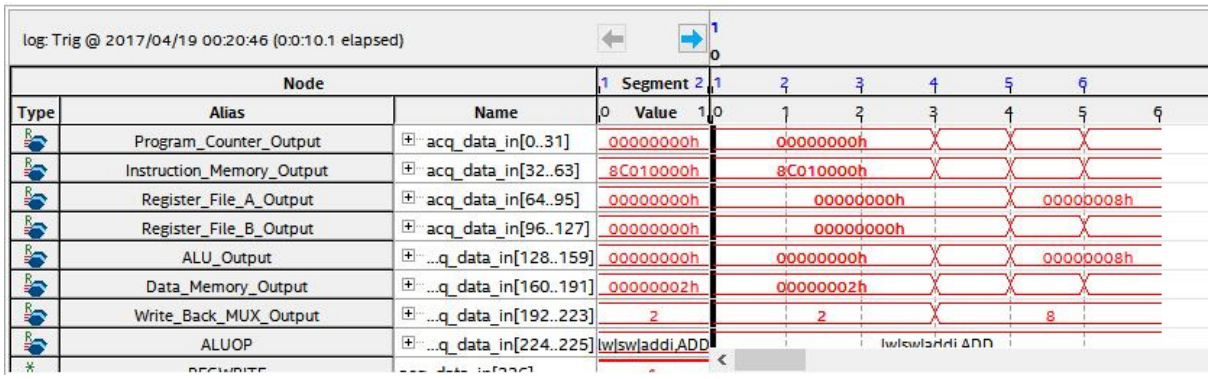


The value stored in the register r1 is added to the value pointed by the memory location in r2 + offset is stored in the register r1. As r2=4, it points to the fourth memory location, which is added to the offset of 4 and thus, the memory location to be considered in this case becomes 8. Now, in DRAM, the 8th memory location contains the value, the result 5+2 = 7 will be stored in the register r1 and shown on LEDs. Below is the screen shot of the signal-tap for the test case 1.

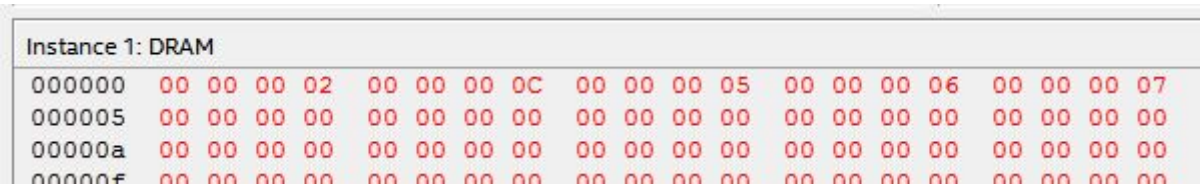


Then, I modified the value on 4th memory location in DRAM to 8. This will store 8 in the register r2, thus making r2 to point the 8th memory location in the DRAM.



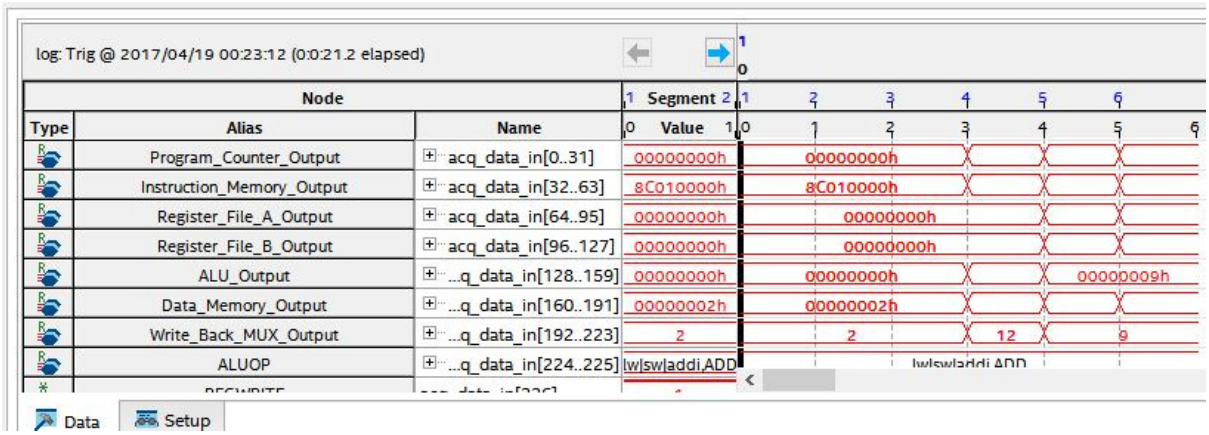


Moreover, the offset of 4 is added to the above value, and thus the memory location to be considered becomes 12. The 12th memory location in DRAM contains the value 6, so the result of 6 + 2, which is equal to 8 is now stored in the register r1.



Finally, the value on the 4th memory location is edited to '0C' in hexadecimal which equals to 12 in decimal. Now this 12 is stored in the register r2 and the offset given in the instruction is 4.

So, the data memory location to be considered becomes 16. On the 6th location in Data memory is the value 6. This 6 when added to 2 becomes equal to 8 and this 8 is stored in the register r1 and shown as output on the LEDs. Also, the output is captured in the signal-tap as shown below.



The above design thus becomes valid for both the newly added addressing modes, i.e. Register Indirect and Displacement. Thus, by adding some adders and MUXs into the design, we can make the design workable for the new instructions without disturbing the original functions of the CPU.