CS6133
Midterm-Project
Yi Qin
N18068309

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)

1.1 what is purpose of having a 32 registers? Could RF be reduced to 16 or 8? (2.5)

The purpose to have 32 register is to store some value temporarily so that they will do some arithmetical operation conveniently. In assembly language, we always have R0-R3 to store the argument and R4-R11 to store local variable or do calculation.

MIPS is a "RISC" or "load-store" architecture.
RAM used to be as fast as CPUs. So people would write programs that would use RAM as intermediate or temporary storage. Early CPUs only had a few registers due to this (i.e. 6502, Z80 - the 6502 only had 3 general purpose registers. Some CPUs like the TMS9900 actually used RAM as registers). This made the CPUs use less transistors which means they were cheaper, easier to get good yields from, easier to develop (no CAD-based chip design in the 70's...) RAM being as fast as the CPU stopped being true around 1985 or so, and has only gotten worse. RISC came to be partly to address this (this was before CPU cache was common or large like it is today) - by having a bunch of registers, slow RAM can be avoided a lot of time for intermediate calculation results and such. Reducing the registers available means it has to go to slower RAM more often for this purpose. Not all processor architectures stopped at 32 registers. Almost all the RISC architectures that have 32 registers exposed in the instruction set actually have 32 integer registers and 32 more floating point registers (so 64). (Floating point "add" uses different registers than integer "add".) The SPARC architecture has register windows. On the SPARC you can only access 32 integer registers at a time, but the registers act like a stack and you can push and pop new registers 16 at a time. The Itanium architecture from HP/Intel had 128 integer and 128 floating point registers exposed in the instruction set. Modern GPUs from NVidia, AMD, Intel, ARM and Imagination Technologies, all expose massive numbers of registers in their register files. (I know this to be true of the NVidia and Intel architectures, I am not very familiar with the AMD, ARM and Imagination instruction sets, but I think the register files are large there too.) most modern microprocessors implement register renaming to eliminate unnecessary serialization caused by needing to reuse resources, so the underlying physical register files can be larger (96, 128 or 192 registers on some

machines.) This (and dynamic scheduling) eliminates some of the need for the compiler to generate so many unique register names, while still providing a larger register file to the scheduler.

Because no matter in R type or I type, we have the bit number for temporary register is 5(i.e. 4..0). The size is relevant to the encoding of instruction. In mips, we always have address of register 00000 - 11111. Because the address size of register is 5…0. So it can have 32

And I think normally we can not reduce the size of register file to 16 or 8 in mips except that we do not need so many general useful register. So, this may refer to some more complexed CPU. And more temporary register will increase the computing speed. So, it may be slow in 16 or 8 register mode.

1.2 What is impact and benefits of reducing the size of the RF? (2.5)

There are two impacts or benefits that we decrease the number of registers exposed in the instruction set. First, you need to be able to specify the register identifiers in each instruction. 32 registers requires a 5 bit register specifier, so 3-address instructions (common on RISC architectures) spend 15 of the 32 instruction bits just to specify the registers. If you increased that to 6 or 7 bits, then you would have less space to specify opcodes and constants. GPUs and Itanium have *much* larger instructions. Larger instructions comes at a cost: you need to use more instruction memory, so your instruction cache behavior is less ideal.
The second reason is access time. The larger you make a memory the slower it is to access data from it. (Just in terms of basic physics: the data is stored in 2-dimensional space, so if you are storing n bits, the average distance to a specific bit is sqrt(n). A register file is just a small multi-ported memory, and one of the constraints on making it larger is that eventually you would need to start clocking your machine slower to accommodate the larger register file. Usually in terms of total performance this is a lose.

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

Keeping instruction memory writable is not done because:

1. The advantages of self modifying or self writing code can be achieved through other means.
2. It is a security risk, as is allowing any data memory to be executed.
3. Debugging programs that are dynamically written is nightmarishly complex.
4. CPU designers assume this will not happen because it is both rare and greatly complicates the design of a pipelined CPU.

Let me address these points one by one.

To the first point: why would we want to write to instruction memory, aside from when code is first loaded? One case would be that we have some "modular" code that we wish to load on the fly; however, most modern operating systems provide facilities to do this at runtime. You don't need to know the code that will be run when you compile because you can simply load it later. But maybe if you know more about the state of the system when you're running the program, you can do optimizations which weren't possible when you compiled it. You necessarily have equivalent or greater information about your system at runtime as opposed to compile time. In most cases, you could still use some form of JIT or dynamic code generation, which is then loaded as a module, to accomplish this task. While this requires the instruction memory to be writable during the loading, the loader should not set the memory as writable and executable at the same time. Now, for argument's sake, let's assume that you have some magical algorithm that, given input as you look at it, can generate the fastest program possible. Let's further assume that this generated program is so markedly faster with respect to any alternative as to be significant. This will need to be addressed by the remaining three points.

To the second point: by allowing your program memory to be written, you expose yourself to the risk that input that is beyond your control (that is, input from an outside source) may be executed as code. Assuming that a person may re-write code in your program memory, perhaps through some bug in your program, is quite similar to having a buffer overflow in your program where the attacker may execute data memory. In the case of modifiable program memory, the attacker would re-write your program. In the case of executing programs from data memory, the attacker would write the desired program into data memory and then jump to it. In either case, an untrusted party may be able to make your program

perform whatever actions they desire. This is bad, and you don't want this to happen.

Now, let's talk about trying to debug a program that re-writes its own code. This actually turns out to be fairly hard. Writing a sane debugger for optimized programs is hard. For programs that write their own code on the fly, even more so. In general, there is a great deal of information that is included in program output to assist the debugger, like: I have this variable -- where is it in memory? I have this line of code -- what instructions correspond to it? Self writing code will contain none of this information. Dealing with code optimization and program variables is actually quite complicated ... what do you do if your variable is turned into some intermediate state of a processor instruction ... what do you do if many lines of code have been mixed (or intermixed) into one or more processor instructions? There is often not a clear answer. This is just for code optimized by a compiler ... now assume your program code may be changed arbitrarily. If it is supported, you might be able to tell your debugger "Hey, start disassembling instructions from this address, assuming they're all valid instructions." In any case, you will, most likely, be stuck debugging your program at a level which is far lower than you are accustomed to (and comfortable with).

The final reason why this does not happen is that it makes CPU implementation more complicated. Because of the extra complexity for something that programmers rarely do (for the above three reasons) there is little support in CPUs for this working predictably or quickly. Let's look at how you might do it. First of all, let's define a way we'd want this to work. CPUs appear as if they are executing instructions in a serial stream. Let's say for some instruction B, if any instruction, up to the execution of B, re-writes B, then B should execute as the new instruction. While this seems fairly simple, the fact that modern CPUs are executed in an instruction pipelining and often with out of order execution complicates things. Usually, while one instruction is being decoded, the one before it may be getting values from registers, the one before it may be accessing memory, the one before it may be using the ALU, and so on. In modern CPUs, there can be hundreds of instructions in various stages of execution at any one time. The fact that instruction B has changed may not be available by the time that instruction B is being decoded. This creates what is called a "pipeline hazard" and CPU designers don't like these. In effect, you would have to monitor if any of the instructions in the pipeline before B have written to where instruction B is stored. If it does write instruction B, then instruction B and all instructions after it will have to be canceled and re-executed to ensure that the correct result is obtained. For data, techniques like

"forwarding" the data to other stages of the pipeline is done, but for instructions this is harder, as all actions performed previously in the pipeline may need to change. No CPU that I know of includes support for detecting the modification of code memory in this way, and it is unlikely, even if you had permission to write instruction memory, that your result would end up in your code cache. You may end up having to flush either the cache line the instruction is in or the entire cache just to see the new instruction. This is slow and usually will negate any speed advantage the re-written code would have.

Advantage:

you don't need to have the whole program in addressable memory and you don't need far jumps. Swapping functions in and out of near memory works just fine. But swapping is expensive if most of the code is the same and conditionals are a waste of space if changes in logic are extremely rare. Replacing one or two instructions is a better option, in such cases. But the combination of precisely those circumstances is rare.In practice, genetic algorithms (which must be written to) and polymorphic viruses (which can change their form) are the main areas of use.C programmers will use dopen() and dclose() to load code in dynamically, but that isn't the same as changing it in the fly.

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.
2 Your program should display 8 least significant bits of each number you generate in the above part through the LEDs.
3 While counting back you should blink the LEDs whenever the count value is equal to a Fibonacci number.

1.1 Design and conceive

For this part, I divide this question into two parts. The adding process to generate the Fibonacci number and the decreasing process by 1 until it reach 0.
The addition part we should consider is to show result through LED.
For LED part, I modify the schematic to achieve this function and show result by instruction: out. we should blink LED when we come across a Fibonacci number. In my design, I will output a 0 to LED which will cause blinking result.

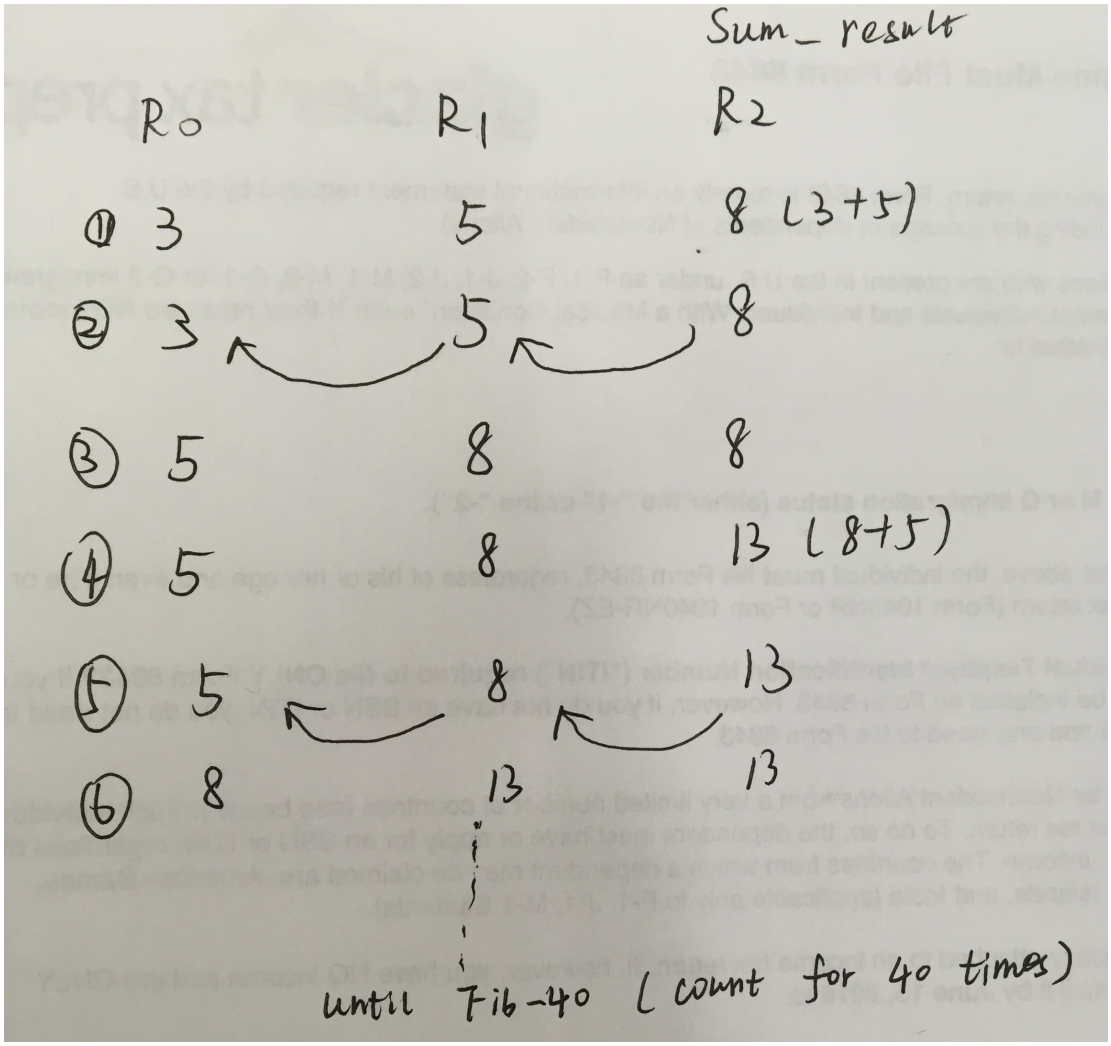
Design for the mif file to implement the process:

Here is the graph to describe this process.
Firstly, we increase the number by Fibonacci number until n=25. So, the number should be added to 325. When it counts back, I will increase the counter by 2 to calculate whether the number has decreased to a Fibonacci number. For example,

Adding Process:

I use R2 to store the addition result of previous two Fibonacci number. R0 stores the value of previous Fibonacci number. R1 stores the value of previous previous Fibonacci number.

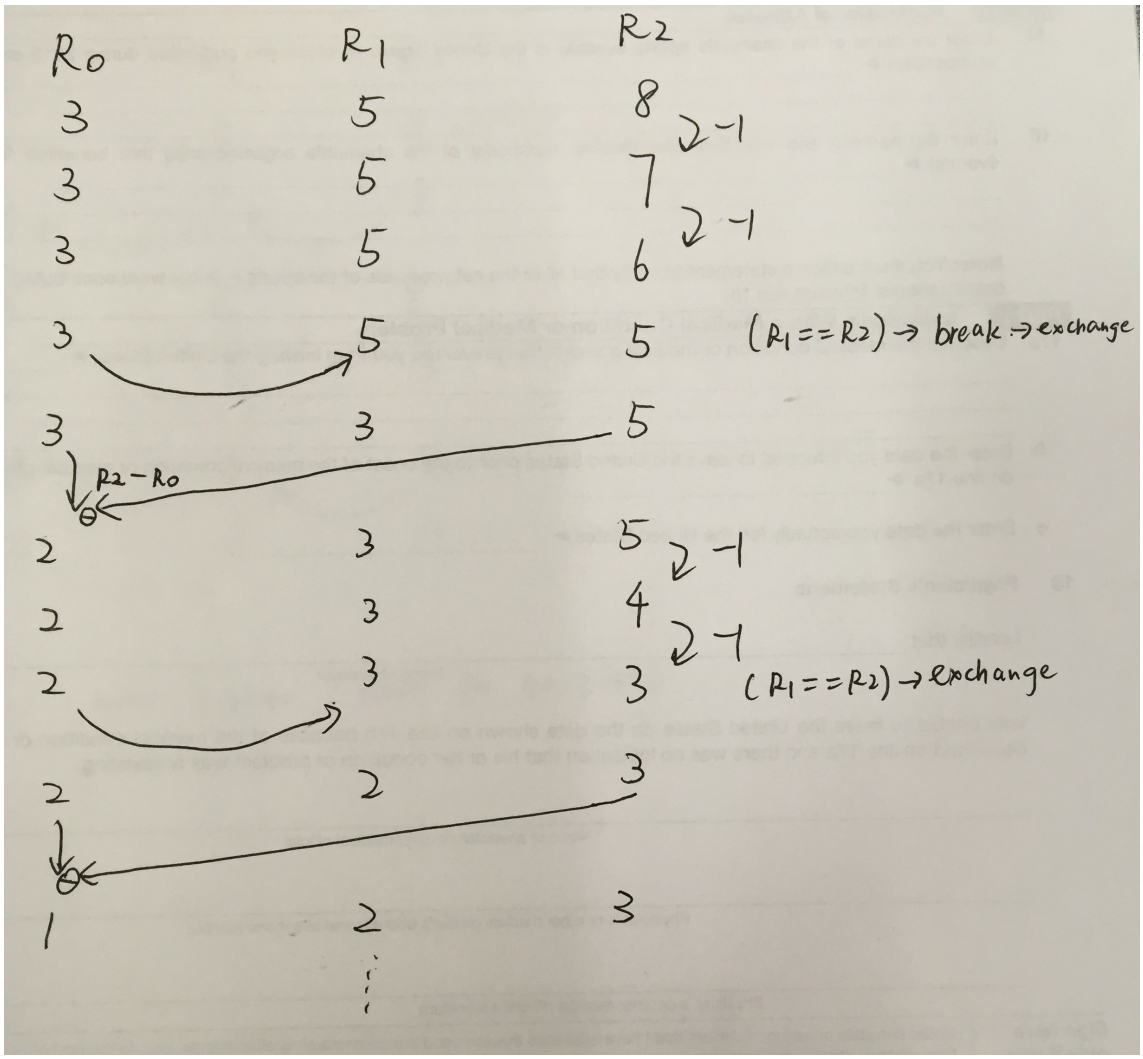
1. Add previous Fibonacci number and store the result to R2
2. Exchange the value: assign the value from R1 to R0 and assign the value from R2 to R1. In this way R0, R1 can always hold the value of closed previous two Fibonacci number
3. Add until we have count to the 40th Fibonacci number Fib-40.



Subtraction Process:

The R2 will store the current decrement value. R0 stores the value of previous Fibonacci number. R1 stores the value of previous previous Fibonacci number.

1. Decrease the R2 by 1 until come across the Fibonacci number
2. If R2 come across the Fibonacci number, which means (R2 == R1), we will assign the value of R1 from R0 and then assign R0 with the subtraction value of R2, R0.
3. Decrease until R0 equals to 0.



This is the **pseudo code** for the whole process of Fibonacci numbers generator: (Based on C)

The procedure of our assembly code: (pseudo code)	

Add the value to Fibonacci number	do{
	new Fib_num = previous Fib 1 + previous previous Fib 2;
	out new Fib_num; (show LEDS)
	count_times ++ ;
	}
	while(count_times != 39)
	
	new Fib_num = previous Fib 1 + previous previous Fib 2;

Subtract the value by 1 before we come across or pass by the first Fibonacci number	do{
	Fib_num = Fib_num - 1;
	out Fib_num;
	}
	while(Fib_num != previous Fib_num)

if we come across the Fibonacci number, we should blink LED and do some operation	if (Fib_num == previous Fib_num){
	blink LED;
	previous Fib_num 1 = previous previous Fib_num 2;
	Fib_num 2 = current Fib_num - Fib_num 2;
	}

If we have decrease all the Fibonacci number, we should restart the process.	if (previous previous Fib_num 2 == 0){
	restart;
	}

There is the table for register referring to variable:

register	variable
R0	increment/decrement parameter Fib1
R1	increment/decrement parameter Fib2
R2	increment/decrement result
R3	count increment times
R4	store value 0 for blinking LEDS
R5	store 1
R6	store 39

1.2 Coding and coding for mif file

1) In order to accomplish this exercise, I used a total of 3 space in DRam, as described here:

Location	Value(Binary)	Value(Decimal)
0	00000000000000000000000000000000	0
4	00000000000000000000000000000001	1
8	00000000000000000000000000101000	39

Based on the pseudo code, I write the assembly code as below:

Location	Pseudo-Assembly	Comment
0	lw R0, 0	Load the initial value to register (see previous descriptions)
1	lw R1, 4	
2	lw R2, 0	
3	lw R3, 4	
4	lw R4, 0	
5	lw R5, 4	
6	lw R6, 8	
7	add R2, R0 ,R1	add the fibonacci number to sum
8	out R2	show result in LEDS
9	add R0, R1, R4	R0 <- R1
10	add R1, R2, R4	R1 <- R2
11	add R3, R3, R5	count_incre + 1
12	beq R3, R6, 1	we have 39 fibonacci number?
13	beq R4, R4, -7	if not, continue addition process
14	add R2, R0 ,R1	to get the 40th fibonacci number
15	sub R2, R2, R5	subtract the number by 1
16	out R2	show result in LED
17	beq R2, R1, 1	if sub_num == para2?
18	beq R4, R4, -4	if not, continue subtracting
19	add R1, R0, R4	R1 <- R0
20	sub R0, R2, R0	R0 <- R2 - R0
21	out, R4	blink LEDS
22	beq R0, R4, -23	num==0? restart
23	beq R4,R4, -9	continue subtracting
24	NOP	No operation

LED part modification:

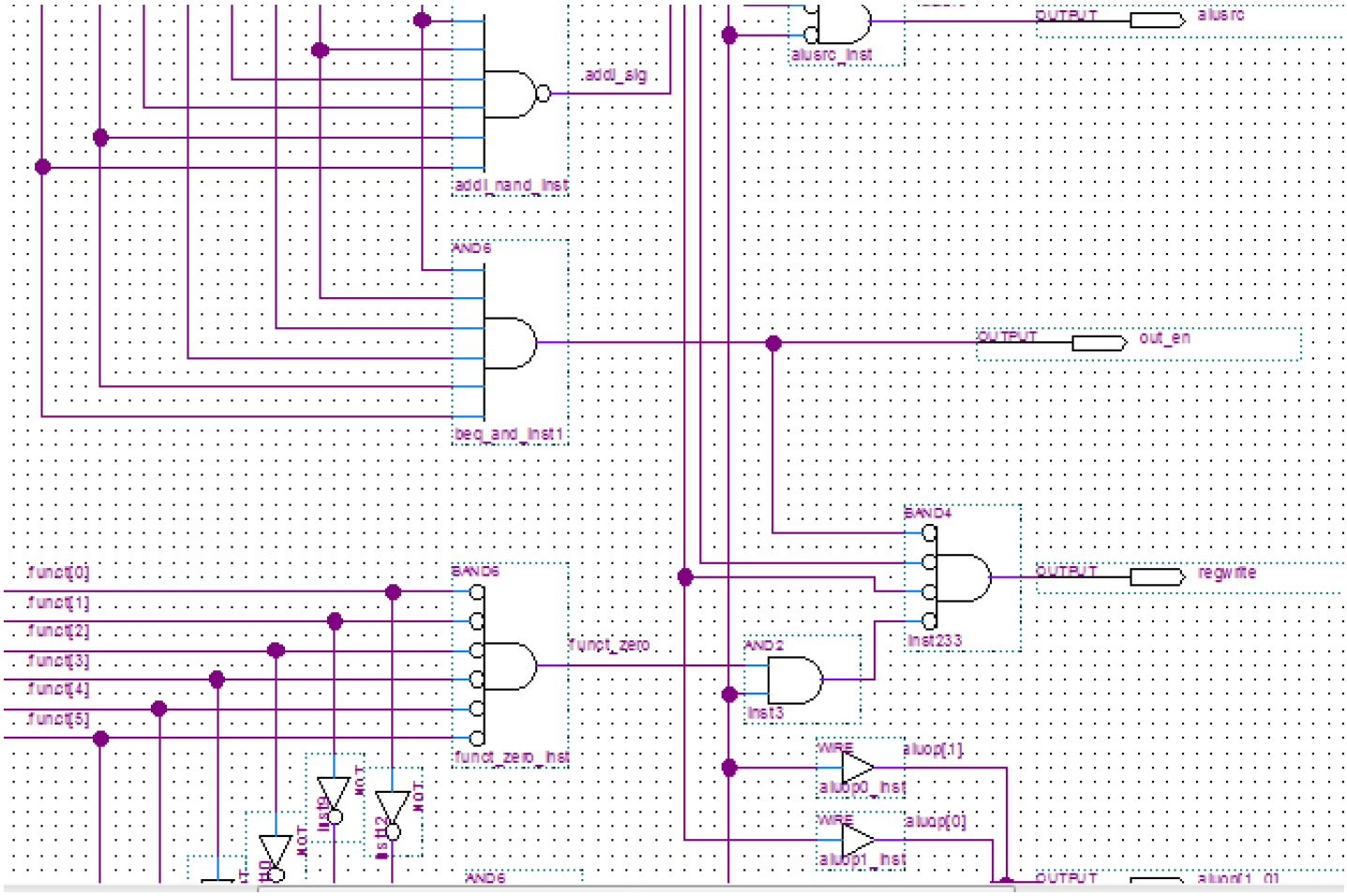
To implement show the results of addition and subtraction result, we should modify the circuit.

I should modify the control unit to add an enable_LED signal to control the outside LED part. When, the enable_LED signal is 1, I will output the result from register. The below graph shows the details of control signal of LED.

Packet format:

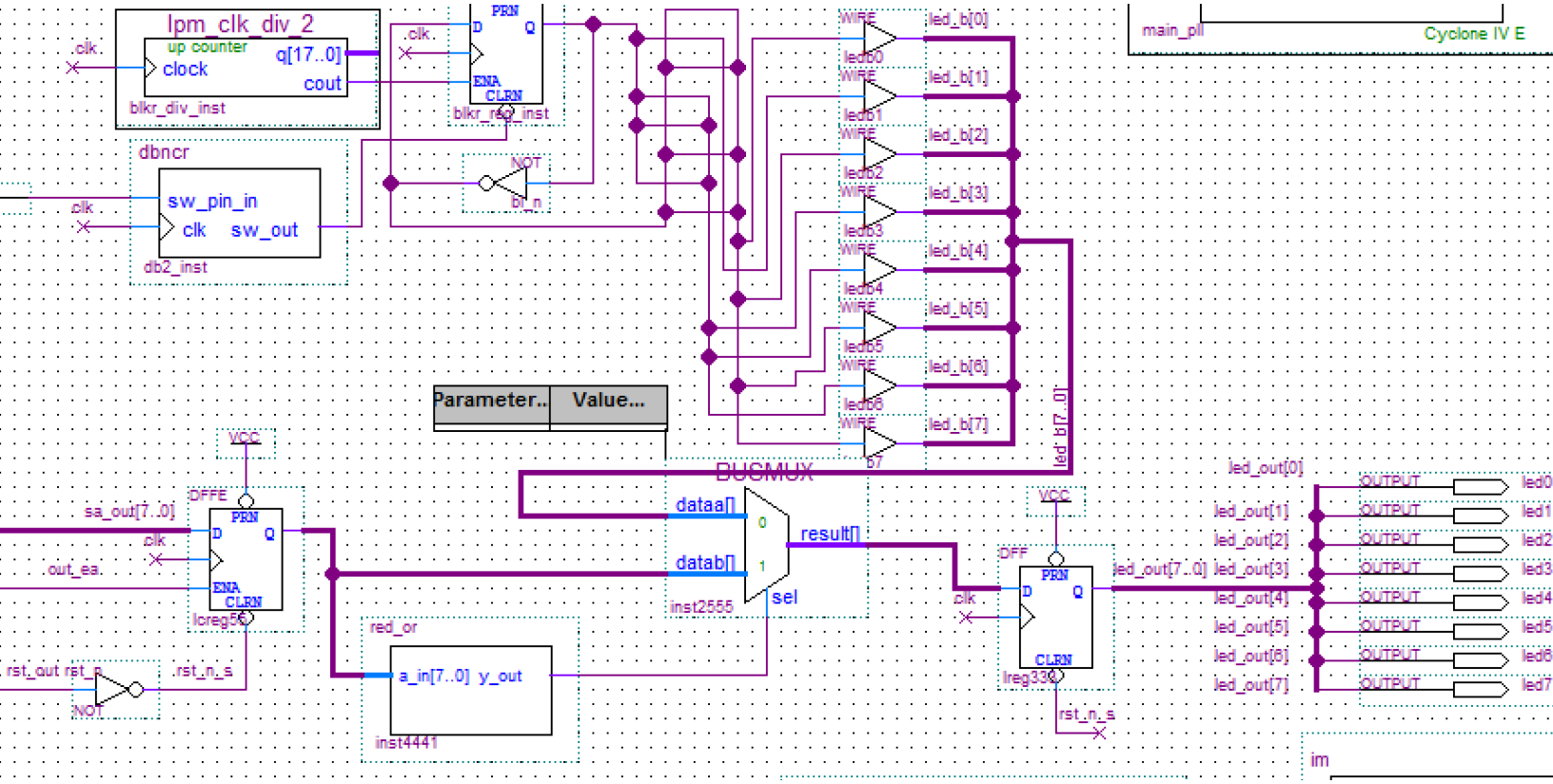
opcode	rs	no influence			
101100	Rs	XXXX	XXXX	XXXXX	XXXXXX

Only register a_out will influence the result of LED. Because I will only connect the a_out to LED part.



Details of LED_enable signal (out_en in graph)

At top level:
Here is the graph how I modify the LED part. Just use the 8 bit or gate and a 2to1_MUX can implement this process. The 8 bit or gate is to verify if the result from register is all zero. If there is all zero result, the output of 8 bit or gate will be 1, otherwise there will be 0.
And the 2to1 mux can cooperate with the 8 bit or gate. The output of 8 bit or gate is connected to 2to1 mux. If the output of 8 bit or gate is 1, which means the output from register is not all 0, the select will be set to 1. So, the result will pass to LED and be shown in LED. On the other hand, if the output of 8 bit or gate is 0, which means the output from register is all 0, the select will be set to 0. So, the above pass will be allowed to pass. So, the LED will blink.



Details of LED part

1.3 The mif file

```
-- NYU:Poly CS6133 Computer Architecture
-- Instructor: Vikram Padman
--
--
-- Modified by Yi Qin
--
-- Instrucion Format:
--
-- R-Type: <6-bit Opcode>,<5-bit rs>,<5-bit rt>,<5-bit rd>,<5-bit
shamt>,<6-bit funct>
--   bits   (31-26)       (25-21)     (20-16)    (15-11)     (10-6)
(5-0)
--
-- I-Type: <6-bit Opcode>,<5-bit rs>,<5-bit rt>,<16-bit Address>
--   bits     (31-26)      (25-21)   (20-16)     (15-0)
--
--
--File format:
-- Hex Address 3 hex nibbles (12 bits) : bit31 ..... bit0;

WIDTH=32;
DEPTH=1024;

ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;

CONTENT BEGIN

-----------------------------------------------------------------
-----
--  Overall program structure:
--  r0 is to store increment/decrement parameter Fib1
--  r1 is to store increment/decrement parameter Fib2
--  r2 is to store increment/decrement result
--  r3 is to store count increment times
--  r4 is 0
--  r5 is 1
--  r6 is 39/27h
```

```
--  Addr        Mnemonics
--  000         ld r0,0         -- Load data memory 0 -> r0
--  001         ld r1,4         -- Load data memory 4 -> r1
--  002         ld r2,0         -- Load data memory 8 -> r2
--  003         ld r3,4         -- Load data memory 0 -> r3
--  004         ld r4,0         -- Load data memory 4 -> r4
--  005     ld r5,4     -- Load data memory 12 -> r5
--  006         ld r6,8         -- Load data memory 0 -> r6
--
--  007     add R2, R0 ,R1  -- add the fibonacci number to sum
--  008     out R2          -- show result in LED
--  009     add R0, R1, R4  -- R0 <- R1
--  00A     add R1, R2, R4  -- R1 <- R2
--  00B     add R3, R3, R5  --count_incre + 1
--  00C     beq R3, R6, 1   --we have 39 fibonacci number?
--  00D     beq R4, R4, -7  --if not, continue counting
--  00E     add R2, R0 ,R1  --to get the 40th fibonacci number
--  00F     sub R2, R2, R5  --subtract the number by 1
--  010     out R2          --show result in LED
--  011     beq R2, R1, 1   --if sub_num == para2?
--  012     beq R4, R4, -4  --if not, continue subtracting
--  013     add R1, R0, R4  --R1 <- R0
--  014     sub R0, R2, R0  --R0 <- R2 - R0
--  015     out, R4         --blink LEDS
--  016     beq R2, R4, -23 --num==0? restart
--  017     beq R4,R4, -9   -- continue subtracting
--  018     NOP             --No operation

-------------------------------------------------------------------------------
--Hex Address :   bit31...........bit15.........bit0;
--   |             |               |               |
    000       :    10001100000000000000000000000000;
--                 |____||___||___||_______________|
--                   |     |    |         |
--                  op=ld, rs=0, rt=0,    addr=0
--  000         ld r0,0         -- Load data memory 0 -> r0
-------------------------------------------------------------------------------
--Hex Address :   bit31...........bit15.........bit0;
--   |             |               |               |
    001       :    10001100000000010000000000000100;
--                 |____||___||___||_______________|
--                   |     |    |         |
--                  op=ld, rs=0, rt=1,    addr=4
--  001         ld r1,1         -- Load data memory 1 -> r1
-------------------------------------------------------------------------------
--Hex Address :   bit31...........bit15.........bit0;
--   |             |               |               |
    002       :    10001100000000100000000000000000;
--                 |____||___||___||_______________|
--                   |     |    |         |
--                  op=ld, rs=0, rt=2,    addr=0
--  002         ld r2,0         -- Load data memory 0 -> r2
```

```
--Hex Address :   bit31...........bit15.........bit0;
--   |              |                |              |
    003       :    10001100000000110000000000000100;
--                 |____||___||___||_______________|
--                   |     |    |           |
--                 op=ld, rs=0, rt=3,    addr=0
--  003          ld r3,1          -- Load data memory 1 -> r3

-------------------------------------------------------------------------------
--Hex Address :   bit31...........bit15.........bit0;
--   |              |                |              |
    004       :    10001100000001000000000000000000;
--                 |____||___||___||_______________|
--                   |     |    |           |
--                 op=ld, rs=0, rt=4,    addr=4
--  004          ld r4,0          -- Load data memory 0 -> r4

-------------------------------------------------------------------------------

--Hex Address :   bit31...........bit15.........bit0;
--   |              |                |              |
    005       :    10001100000001010000000000000100;
--                 |____||___||___||_______________|
--                   |     |    |           |
--                 op=ld, rs=0, rt=5,    addr=4
--  005          ld r5,1          -- Load data memory 1 -> r5

-------------------------------------------------------------------------------

--Hex Address :   bit31...........bit15.........bit0;
--   |              |                |              |
    006       :    10001100000001100000000000001000;
--                 |____||___||___||_______________|
--                   |     |    |           |
--                 op=ld, rs=0, rt=6,    addr=8
--  006          ld r6,40   -- Load data memory 40 -> r6

-------------------------------------------------------------------------------



-- Add process:


--Hex Address :   bit31...........bit15.........bit0;
--   |              |                |              |
    007       :    00000000000000001000100000100000;
--                 |____||___||___||___||___||____|
--                   |     |    |    |    |    |
--                 R-type,rs=0,rt=1,rd=2,---,f=add
--  007          add r2,r0,r1     -- add and load r2<-r0+r1
```

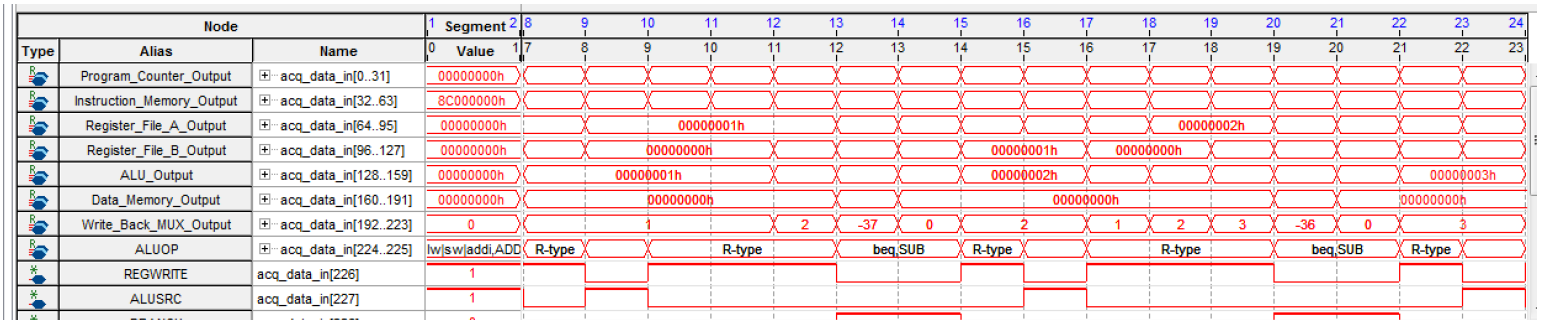
```
--Hex Address :   bit31...........bit15.........bit0;
--   |              |                |              |
    008       :    10110000010000000000000000000000;
--                 |____||___||____________________|
--                   |     |          |
--                    out, rs=2, XXXXXXXXXXXXXXXXXX
-- This is a out instruction
--  008          out r2

-------------------------------------------------------------------------
--Hex Address :   bit31...........bit15.........bit0;
--   |              |                |              |
    009       :    00000000001001000000000000100000;
--                 |____||___||___||___||___||____|
--                   |     |    |    |    |    |
--                  R-type,rs=1,rt=4,rd=0,---,f=add
--  009          add R0, R1, R4  -- R0 <- R1

-------------------------------------------------------------------------
--Hex Address :   bit31...........bit15.........bit0;
--   |              |                |              |
    00A       :    00000000010001000000100000100000;
--                 |____||___||___||___||___||____|
--                   |     |    |    |    |    |
--                  R-type,rs=2,rt=4,rd=1,---,f=add
--  00A          add add R1, R2, R4    -- R1 <- R2

-------------------------------------------------------------------------
--Hex Address :   bit31...........bit15.........bit0;
--   |              |                |              |
    00B       :    00000000011001010001100000100000;
--                 |____||___||___||___||___||____|
--                   |     |    |    |    |    |
--                  R-type,rs=3,rt=5,rd=3,---,f=add
--  00B          add R3, R3, R5  -- R1 <- R2
-------------------------------------------------------------------------
--Hex Address :   bit31...........bit15.........bit0;
--   |              |                |              |
    00C       :    00010000011001100000000000000001;
--                 |____||___||___||________________|
--                   |     |    |          |
--                    beq, rs=3, rt=6,   offset=1
-- This is a beq, branch if equal instruction
--  00C          beq R3, R6, 1   -- Branch to 00D
-------------------------------------------------------------------------
--Hex Address :   bit31...........bit15.........bit0;
--   |              |                |              |
    00D       :    00010000100001001111111111111001;
--                 |____||___||___||________________|
--                   |     |    |          |
--                    beq, rs=4, rt=4,   offset=-7
--  00D          beq r0,r0,007   -- Branch to 007
```

```
--Hex Address :   bit31...........bit15..........bit0;
--   |              |                 |               |
    00E       :    00000000000000010001000000100000;
--                 |____||___||___||___||___||____|
--                   |     |    |    |    |    |
--                  R-type,rs=0,rt=1,rd=2,---,f=add
--  00E          add r2,r0,r1    -- add and load r2<-r0+r1

-------------------------------------------------------------------------------

-- subtract process

--Hex Address :   bit31...........bit15..........bit0;
--   |              |                 |               |
    00F       :    00000000010001010001000000100010;
--                 |____||___||___||___||___||____|
--                   |     |    |    |    |    |
--                  R-type,rs=2,rt=5,rd=2,---,f=sub
--  00F          sub r2,r2,r5    -- subtract the number by 1

-------------------------------------------------------------------------------
--Hex Address :   bit31...........bit15..........bit0;
--   |              |                 |               |
    010       :    10110000010000000000000000000000;
--                 |____||___||____________________|
--                   |     |           |
--                   out, rs=2, XXXXXXXXXXXXXXXXXX
-- This is a out instruction
--  010          out r2    -- show result in LED

-------------------------------------------------------------------------------
--Hex Address :   bit31...........bit15..........bit0;
--   |              |                 |               |
    011       :    00010000010000010000000000000001;
--                 |____||___||___||________________|
--                   |     |    |          |
--                   beq, rs=2, rt=1,   offset=1
-- This is a beq, branch if equal instruction
--  011          beq R2, R1, 1   -- Branch to 012

-------------------------------------------------------------------------------
--Hex Address :   bit31...........bit15..........bit0;
--   |              |                 |               |
    012       :    00010000100001001111111111111100;
--                 |____||___||___||________________|
--                   |     |    |          |
--                   beq, rs=4, rt=4,   offset=-4
-- This is a beq, branch if equal instruction
--  012          beq r4,r4,00E   -- Branch to 00E
```

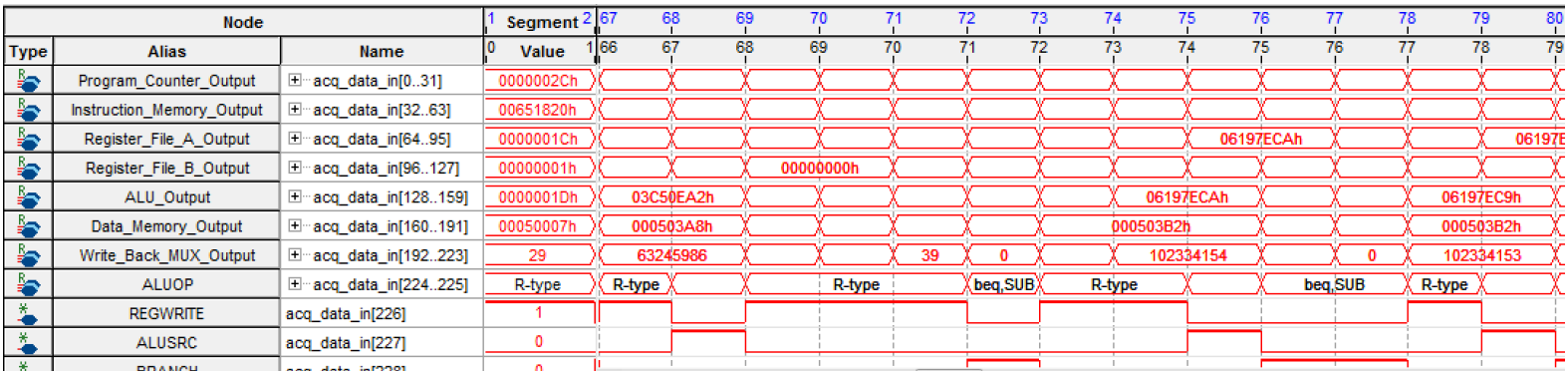
```
--Hex Address :   bit31...........bit15..........bit0;
--   |              |                 |               |
    013       :    00000000000010000010000100000;
--                 |____||___||___||___||___||____|
--                   |     |    |    |    |    |
--                  R-type,rs=0,rt=4,rd=1,---,f=add
--  013          add r1,r0,r4     -- R1 <- R0
-------------------------------------------------------------------------
--Hex Address :   bit31...........bit15..........bit0;
--   |              |                 |               |
    014       :    00000000100000000000000100010;
--                 |____||___||___||___||___||____|
--                   |     |    |    |    |    |
--                  R-type,rs=2,rt=0,rd=0,---,f=sub
--  014          sub r2,r2,r5     -- R0 <- R2 - R0
-------------------------------------------------------------------------
--Hex Address :   bit31...........bit15..........bit0;
--   |              |                 |               |
    015       :    10110000100000000000000000000000;
--                 |____||___||____________________|
--                   |     |          |
--                   out, rs=4, XXXXXXXXXXXXXXXXXX
-- This is a out instruction
--  015          out r4          -- blink LEDS
-------------------------------------------------------------------------
--Hex Address :   bit31...........bit15..........bit0;
--   |              |                 |               |
    016       :    00010000000010011111111111101001;
--                 |____||___||___||_______________|
--                   |     |    |          |
--                    beq, rs=0, rt=4,   offset=-23
-- This is a beq, branch if equal instruction
--  016          beq r2,r4,000   -- Branch to begining
-------------------------------------------------------------------------
--Hex Address :   bit31...........bit15..........bit0;
--   |              |                 |               |
    017       :    00010000100001001111111111110111;
--                 |____||___||___||_______________|
--                   |     |    |          |
--                    beq, rs=4, rt=4,   offset=-9
-- This is a beq, branch if equal instruction
--  017          beq r4,r4,00E   -- Branch to 00E
-------------------------------------------------------------------------
--Hex Address :   bit31...........bit15..........bit0;
--   |              |                 |               |
    018       :    00000000000000000000000000000000;
--                 |____||___||___||___||___||____|
--                   |     |    |    |    |    |
--                  R-type,XXXX,XXXX,XXXX,XXX,f=NOP
--  018          This is a NOP instruction
```

1.4 Testing and verification through Signal Tap II and LEDS

1) Increasing to Fib-40

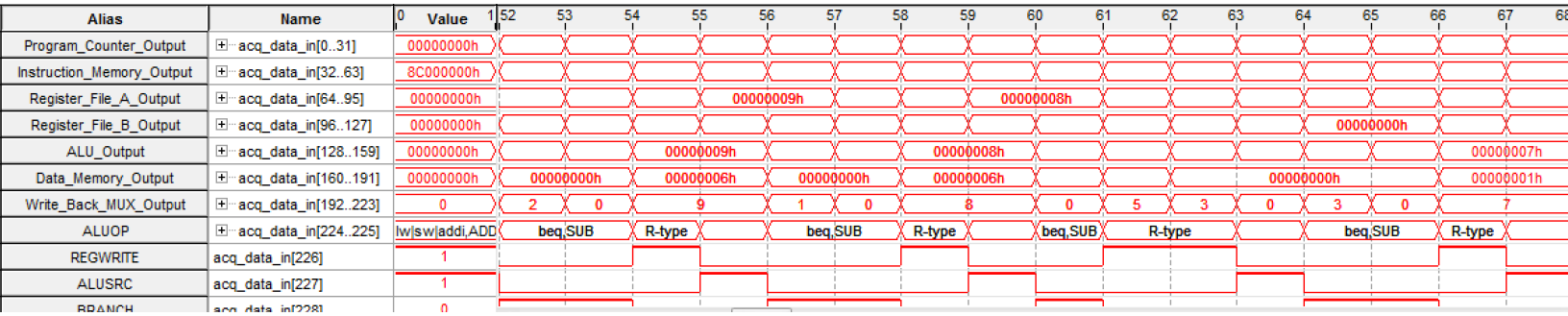


Increase the Fibonacci number R3 = R1 + R2. We can see the result is 1 (0+1), 2 (1+1), 3(1+2).



As we can see through above graph, the number is increased to 102334154 by previous Fibonacci number and stop by counter = 40. Then, begin to decrease by 1, such as 102334153, 102334152….

2) Decreasing until come across or pass by the Fibonacci number (example is number 8):



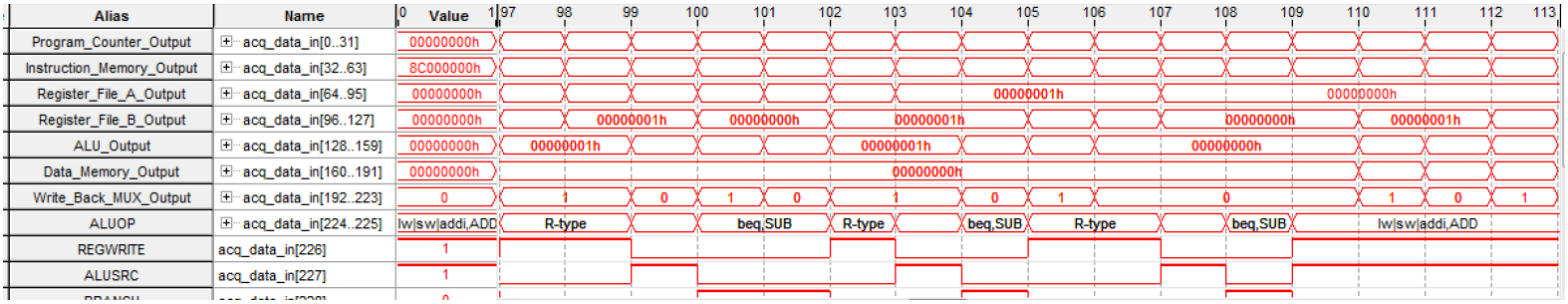
Because the Fib-40 is so big, I will type many times clk in Signal tap. I add to Fib-6 to test the decrease process.

As we can see through above graph, as we reach 8, it is a Fibonacci number. So, it will enter to the special operation stage.

Stage	Verification of operation
58	Fib_num decrease to 8
59	out the result at LED
60	it is a Fib_num? —> Yes and branch
61	R1 <- R0, R1 = 5
62	R0 <- R2 - R0, R0 = 8 - 5 = 3
63	blink LEDS, out 000000
64	num==0? no
65	The next decrease loop

So, the process is right.

3) Decreasing when come across the final Fibonacci number, return the beginning and restart



Stage	Verification of operation
102	Fib_num decrease to 1
103	out the result at LED
104	it is a Fib_num? —> Yes and branch
105	R1 <- R0, R1 = 1
106	R0 <- R2 - R0, R0 = 1 - 1 = 0
107	blink LEDS, out 000000
108	num==0? yes and restart
109	Restart the system and load value to R0

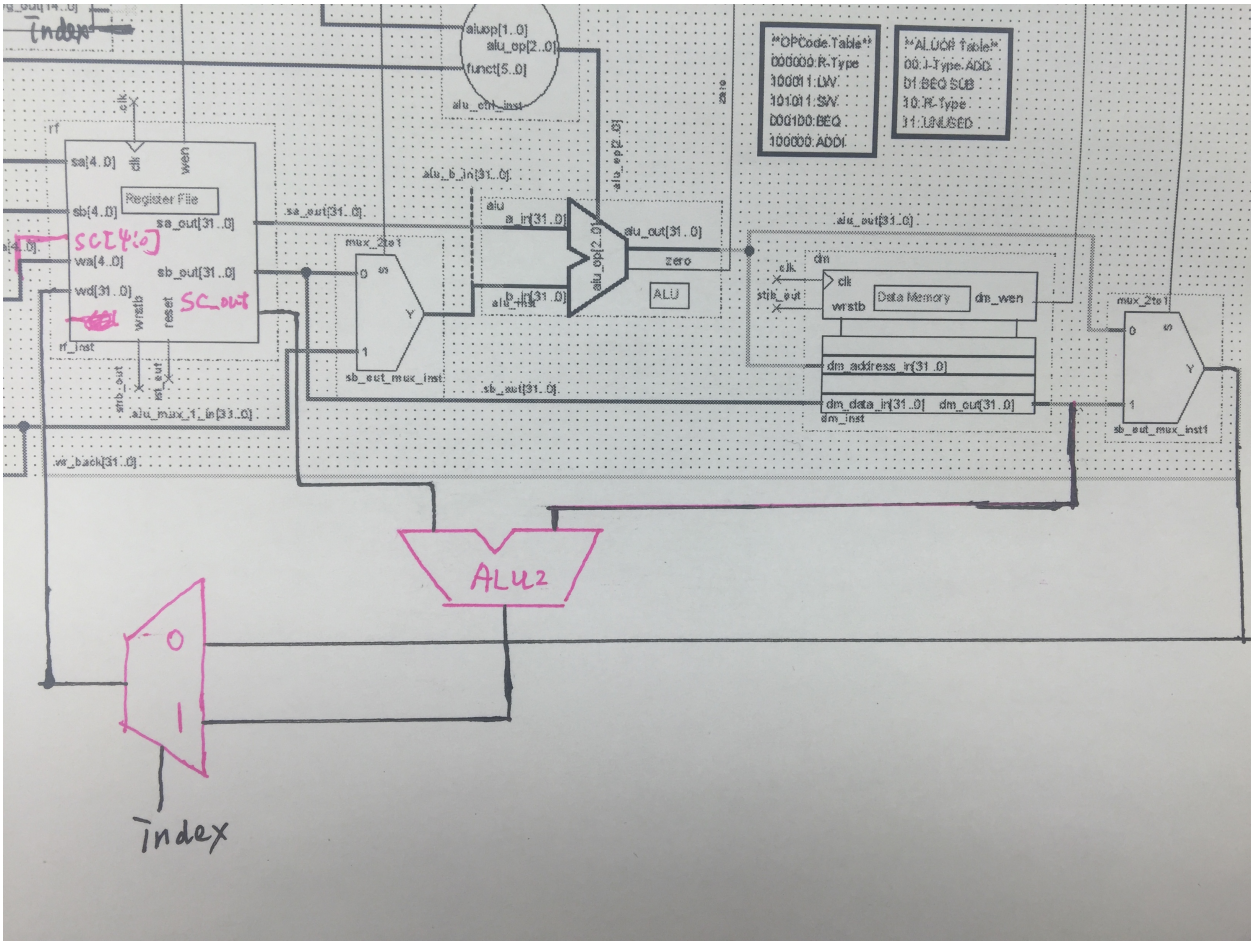
Part III – New Instruction:

Mode: Indexed, auto increment, auto decrement

1.Indexed addressing:

[Rd] <- [Rd] + [Rs + Rt] The value to load is from data memory and the address of this data is the addition result of Rs and Rt

This is the overall schematic of this process:



As shown above, this is the design graph.

This is the format of index mode.

opcode	rs	rt	rd	—-	index
000000	—-	—-	——	00000	110000

In the control signal, there are two signal I should modify:

the memory read should be 1 and to generate the index control signal to control the memory read signal. This means we want to use the output of data memory. And the address of data memory is from the output of ALU. Because the last 4 bit of function number is 0000, the ALU will operate in increasing mode automatically. And I also increase a register output from register file to get the Rd value.

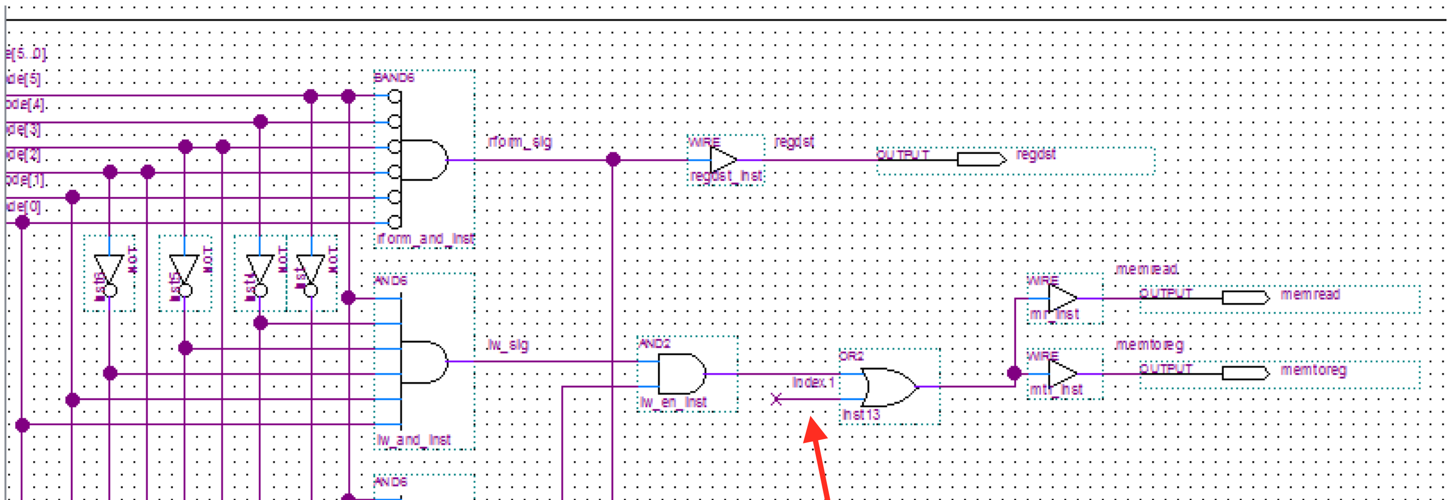
At the control unit: I add a control signal index to control the other module.

And at the register file: we should add a sc_address[4:0] and the sc_out[31:0] to let the Rd value out and do some operations.

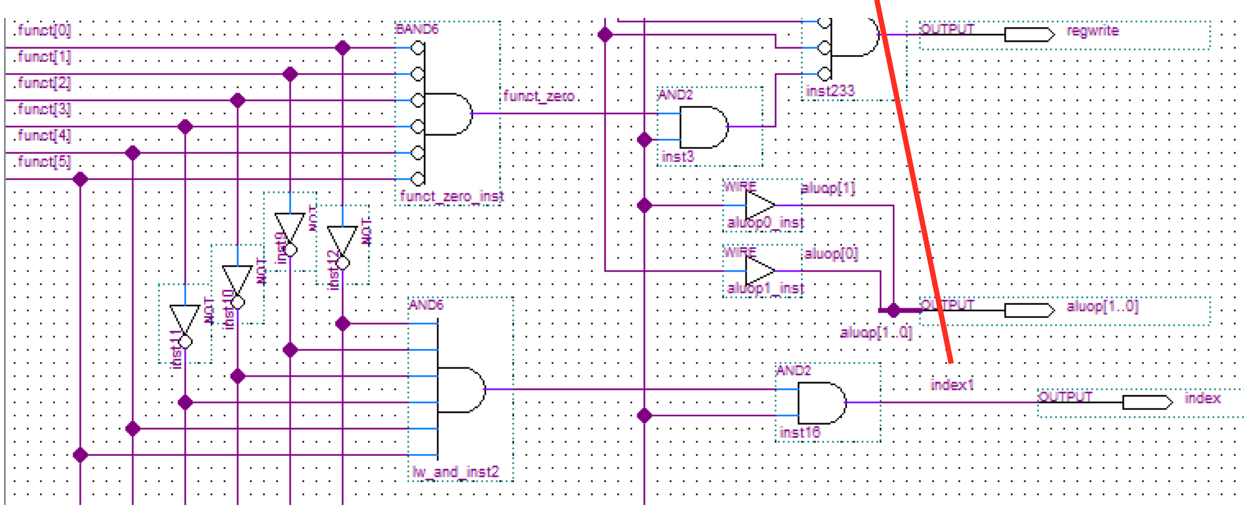
Finally, at the top level: I add an new ALU to add the value of Rd with memory out. And add a mux to choose whether the data in of register file is from the normal path or the new ALU output.

In control unit module:

The memory read signal:

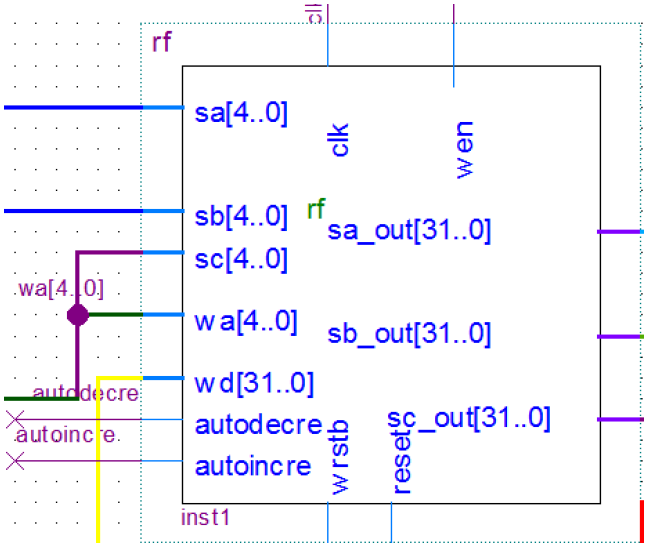


The indexed mode signal:



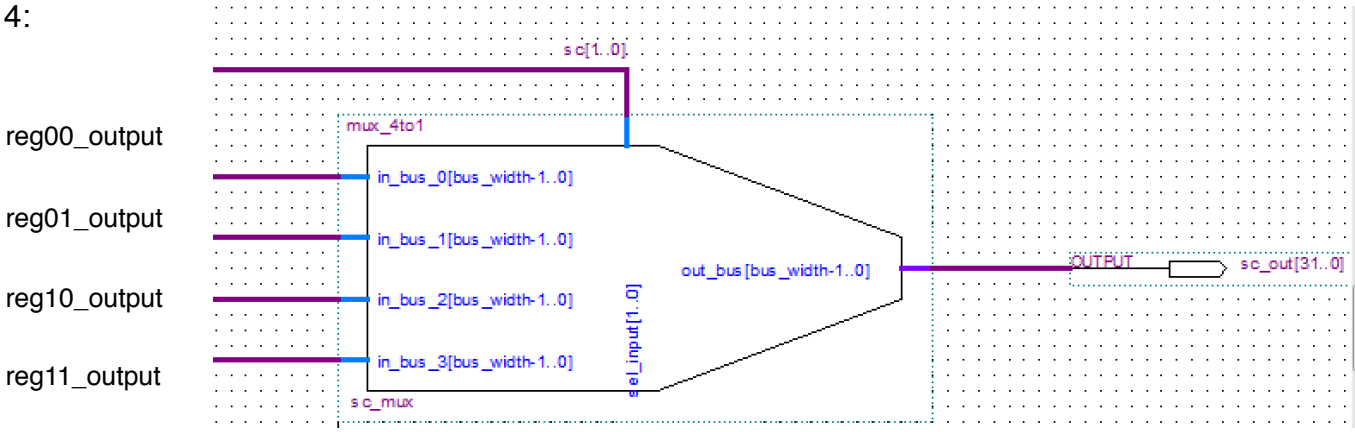
For the function number 110000, I add the index signal to control this mode and the memory read should also be one to get the value from data memory.

In the register file:



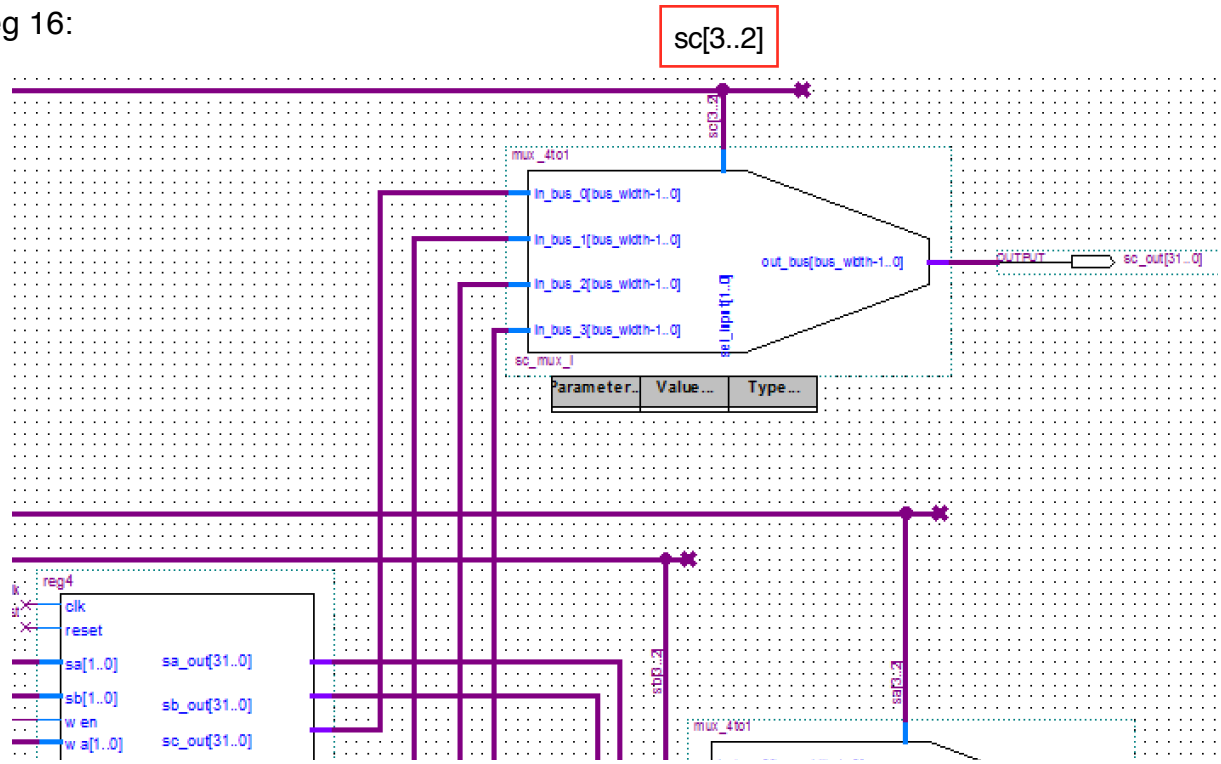
We should add a sc_address[4:0] for address of Rd and the sc_out[31:0] to let the Rd value out and do some operations. And connect the sc address with the write address, because in index add mode, we want to Add Rd to Rd. Rd = Rd + [memory]. So select Rd out for adding (sc) and write back to Rd (wa).

in reg 4:

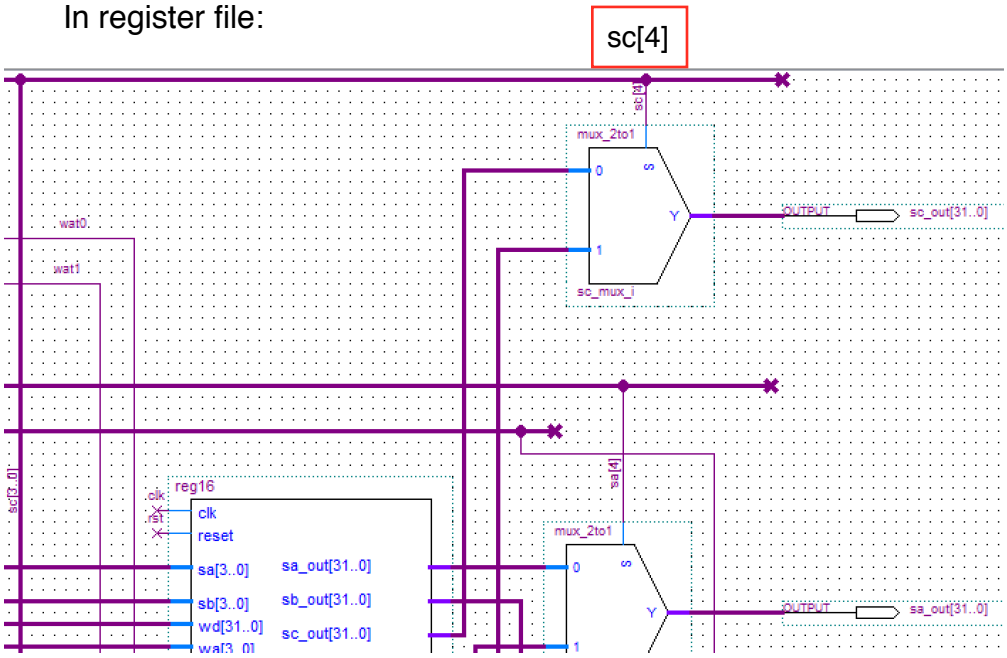


I add a mux 4 to 1 to select the Rd for output. Because we want to add the [Rd] with memory out, we should select the Rd for output.

In reg 16:

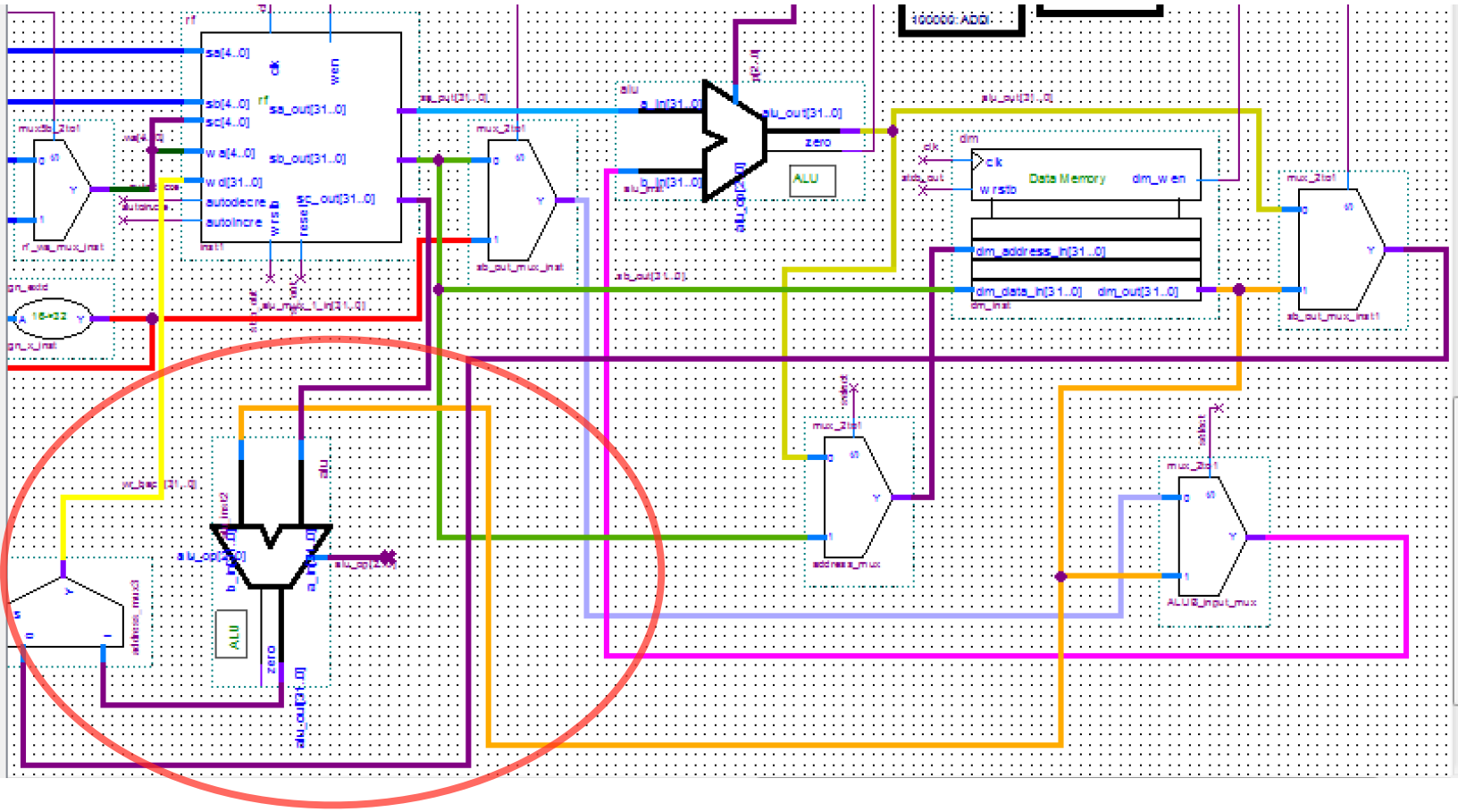


In register file:



Same to the reg 4, we should add a 4 to 1 mux in reg 16 and 2 to 1 mix in register file to select the Rd for output.

At the top level:



I add an new ALU to add the value of Rd with memory out. And add a mux to choose whether the data in of register file is from the normal path or the new ALU output.

Testing of indexed mode:

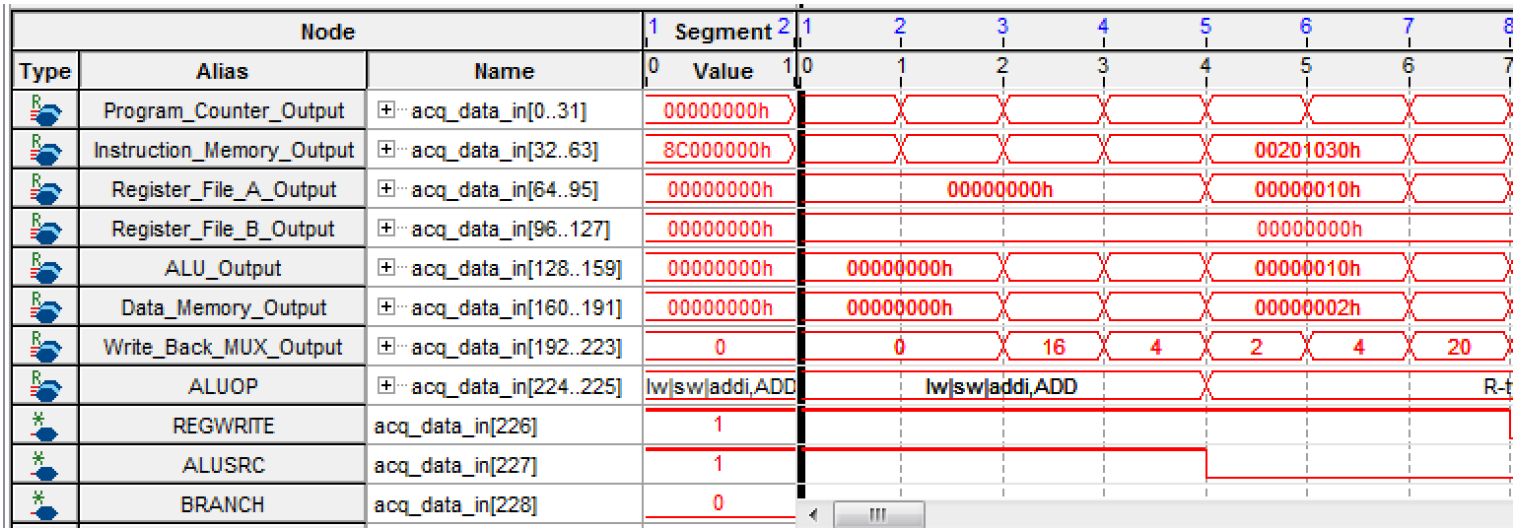
DRAM: path: midterm_YiQin -> Part3_testing_file -> index -> dram_index.mif

```
CONTENT BEGIN
    000        :    00000000000000000000000000000000;
    001        :    00000000000000000000000000010000;
    002        :    00000000000000000000000000000100;
    003        :    00000000000000000000000000000011;
    004        :    00000000000000000000000000000010;
END;
```

IRAM: path: midterm_YiQin -> Part3_testing_file -> index -> Iram_index.mif

```
    000       :    10001100000000000000000000000000;
--                 |____||___||___||_______________|
--                   |     |    |          |
--                   lw, rs=0, rt=0,   offset=0
--  000         load 0 to r0
-----------------------------------------------------------------
--
    001       :    10001100000000100000000000000000;
--                 |____||___||___||_______________|
--                   |     |    |          |
--                   lw, rs=0, rt=2,   offset=0
--  001         load 0 to r2
-----------------------------------------------------------------

    002       :    10001100000000010000000000000100;
--                 |____||___||___||_______________|
--                   |     |    |          |
--                   lw, rs=0, rt=1,   offset=4
--  002         load 16 to r1
-----------------------------------------------------------------
    003       :    10001100000000110000000000001000;
--                 |____||___||___||_______________|
--                   |     |    |          |
--                   lw, rs=0, rt=3,   offset=8
--  003         load 4 to r3

-- index:
-----------------------------------------------------------------
--Hex Address :   bit31..........bit15.........bit0;
--   |             |              |              |
    004       :    00000000001000000001000000110000;
--                 |____||___||___||___||___||____|
--                   |     |    |    |    |    |
--                  R-type,rs=1,rt=0,rd=2,---,f=index
--  004     index_add r1,r0,r2  --  [Rd] =[Rd] + M [Rs + Rt]

-----------------------------------------------------------------
--Hex Address :   bit31..........bit15.........bit0;
--   |             |              |              |
    005       :    00000000001000000001000000110000;
--                 |____||___||___||___||___||____|
--                   |     |    |    |    |    |
--                  R-type,rs=1,rt=0,rd=2,---,f=index
--  005     index_add r1,r0,r2  --  [Rd] =[Rd] + M [Rs + Rt]
-----------------------------------------------------------------
--Hex Address :   bit31..........bit15.........bit0;
--   |             |              |              |
    006       :    00000000011000000001000000110000;
--                 |____||___||___||___||___||____|
--                   |     |    |    |    |    |
--                  R-type,rs=3,rt=0,rd=2,---,f=index
--  006     index_add r3,r0,r2  --  [Rd] =[Rd] + M [Rs + Rt]
```

The result from signal tap:



CLK	operation	write_back_result
0	R0=0	0
1	R2=0	0
2	R1=16	16
3	R3=4	4
4	R2 = R2 + M[R0+R2] (16)	2
5	R2 = R2 + M[R0+R2] (16)	4
6	R2 = R2 + M[R0+R3] (4)	20

We look the last three step is the index add. The initial value of R0 is 0, and it add 2 because the data memory at address 16 (R0+R2) is 2. So, it increase by 2 at CLK 4, 5. And at CLK 6, the data memory address is 4 (R0+R3). And the value in data ram at address 4 is 16. So 4 + 16 = 20.

2. Auto-increment Mode

There are many instructions in this mode, such as add, subtraction, load and so forth. I just take the auto-increment add as example and others are same in register file, just different in opcode in control signal.

```
Add R1,(R2)+:
R1 <- R1 +M[R2]
R2 <- R2 + 4
```

Firstly, add the r1 with data memory output and the address of data memory output is

from R2, then add 4 to R2 automatically.

The instruction format is:

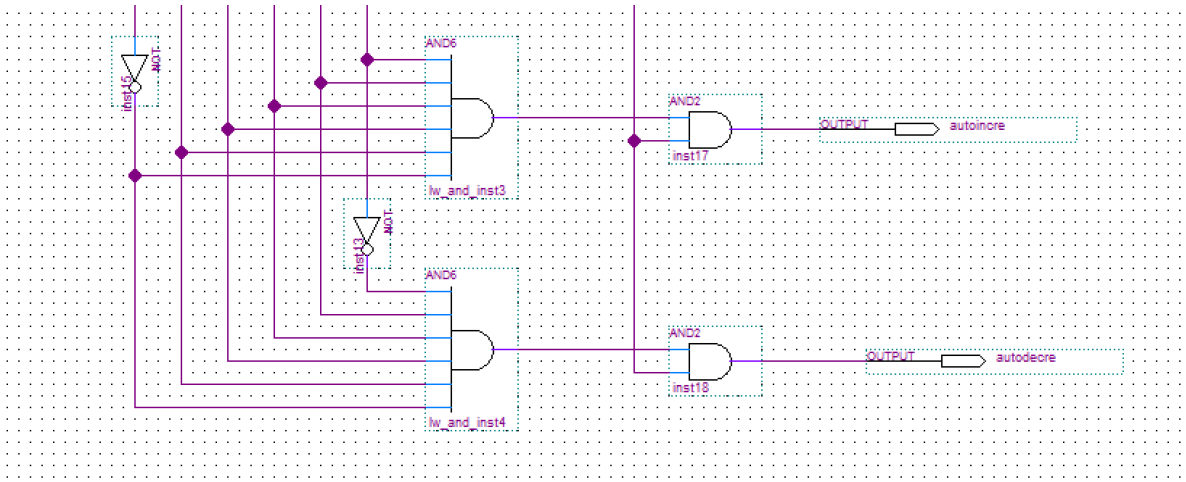
opcode	rs	rt	rd	—-	auto increment
000000	—-	—-	——	00000	010000

I set 010000 as the function code and the format is R type for the least 4 bit of function

is all 0, it will increment automatically.

Firstly, we should design the select signal based on the format and function number. For

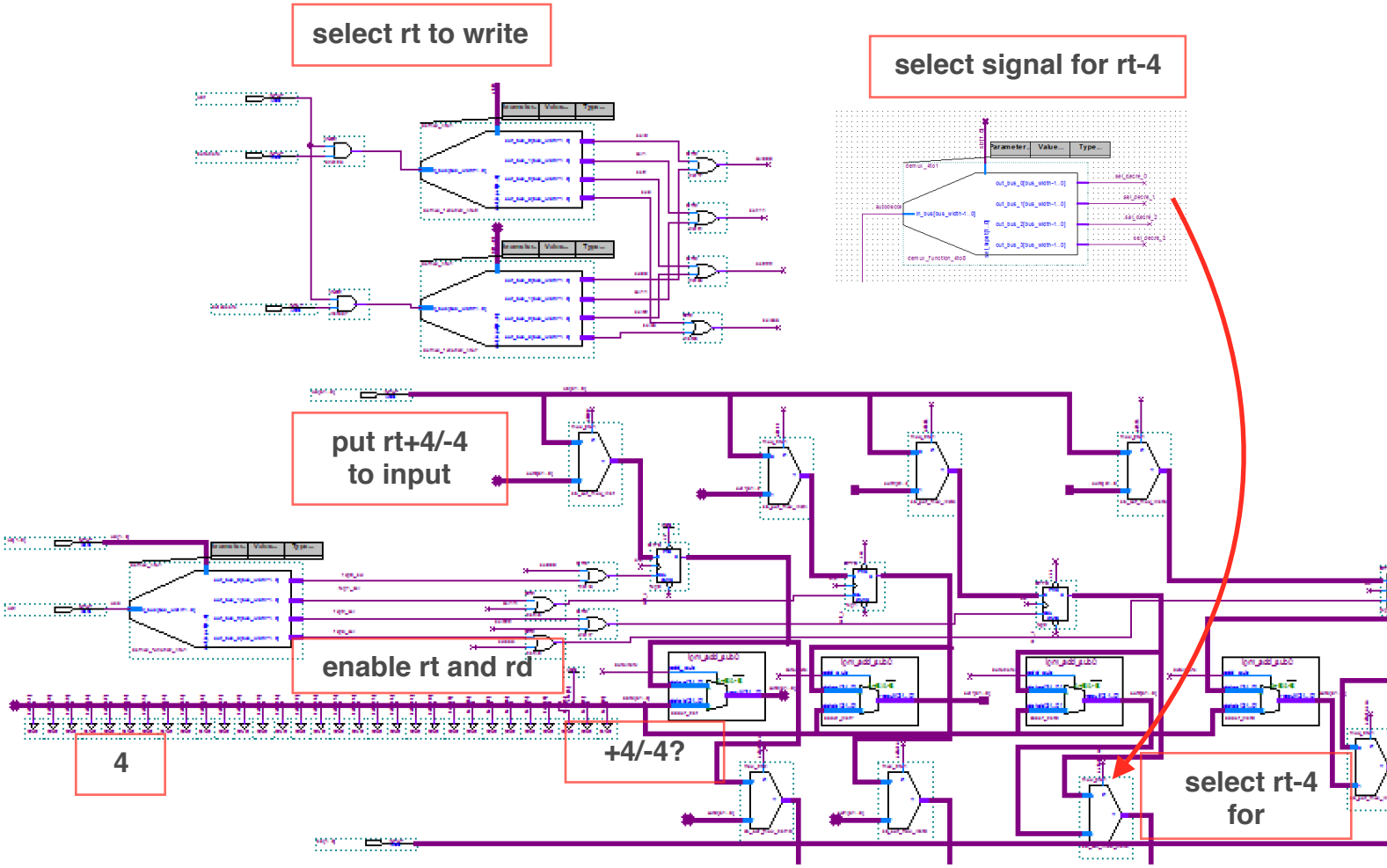
the select signal, we produce it in the control unit.



As we can see through above picture, the signal is based on the function.

We do not need to set other signals because they are all satisfied.

in reg4 module:



This is the overall graph of reg4 module. I will explain each module individually.

In this module, I do the

Rt <- Rt + 4 and Rs <- Rs + input data

Firstly, I should choose which two register will be assigned value. Rs and Rt

For Rt, we should add 4 and load the addition result to Rt itself.

And for Rs, will do the process normally same to add, which means load the addition result to corresponding register.

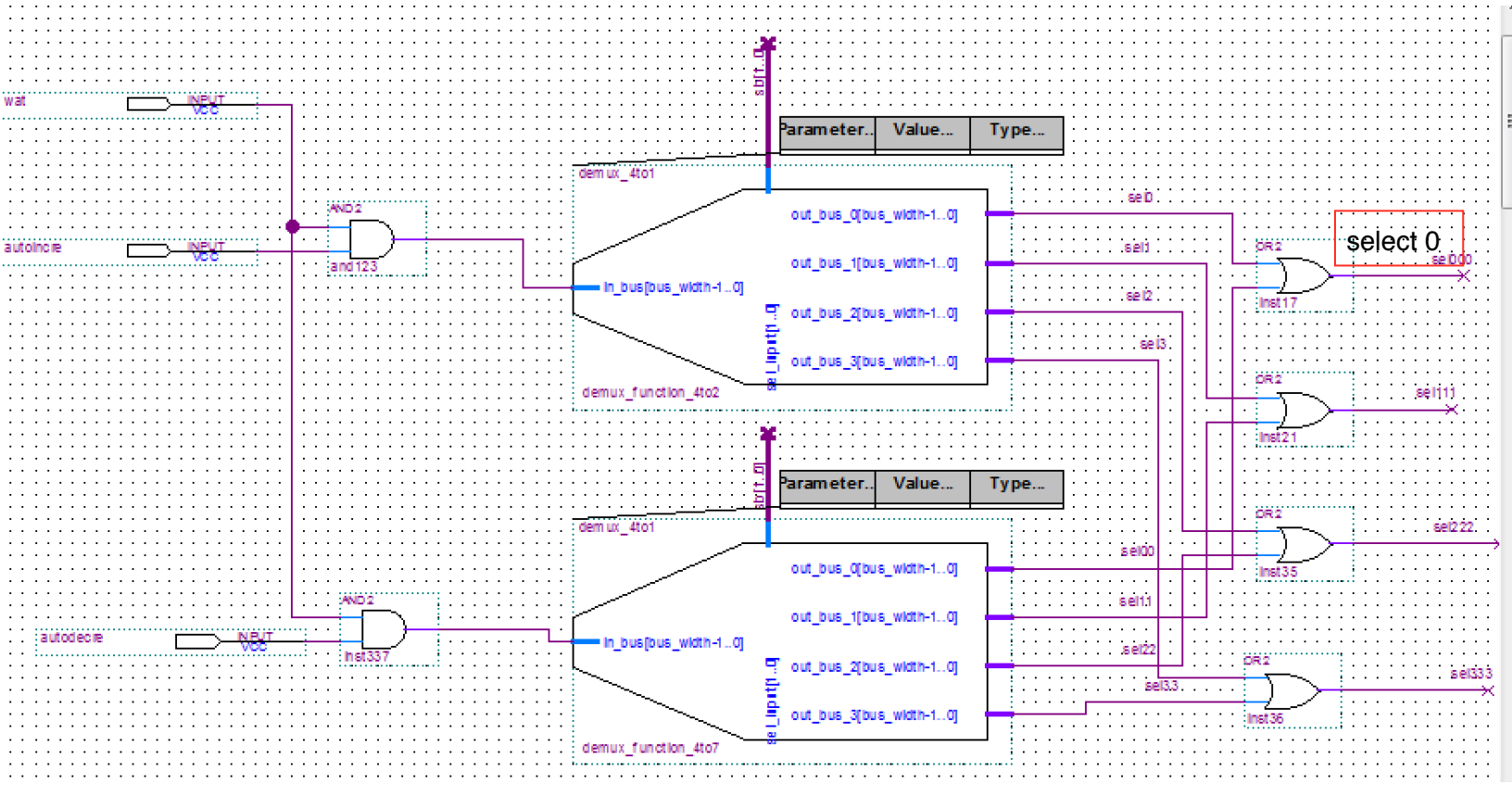
So, firstly we should design the choose signal to choose which register will be choose to add 4 or subtract 4.

We only do this process in the auto-increment or auto-decrement mode. So, we use an encoder to implement this select signal.

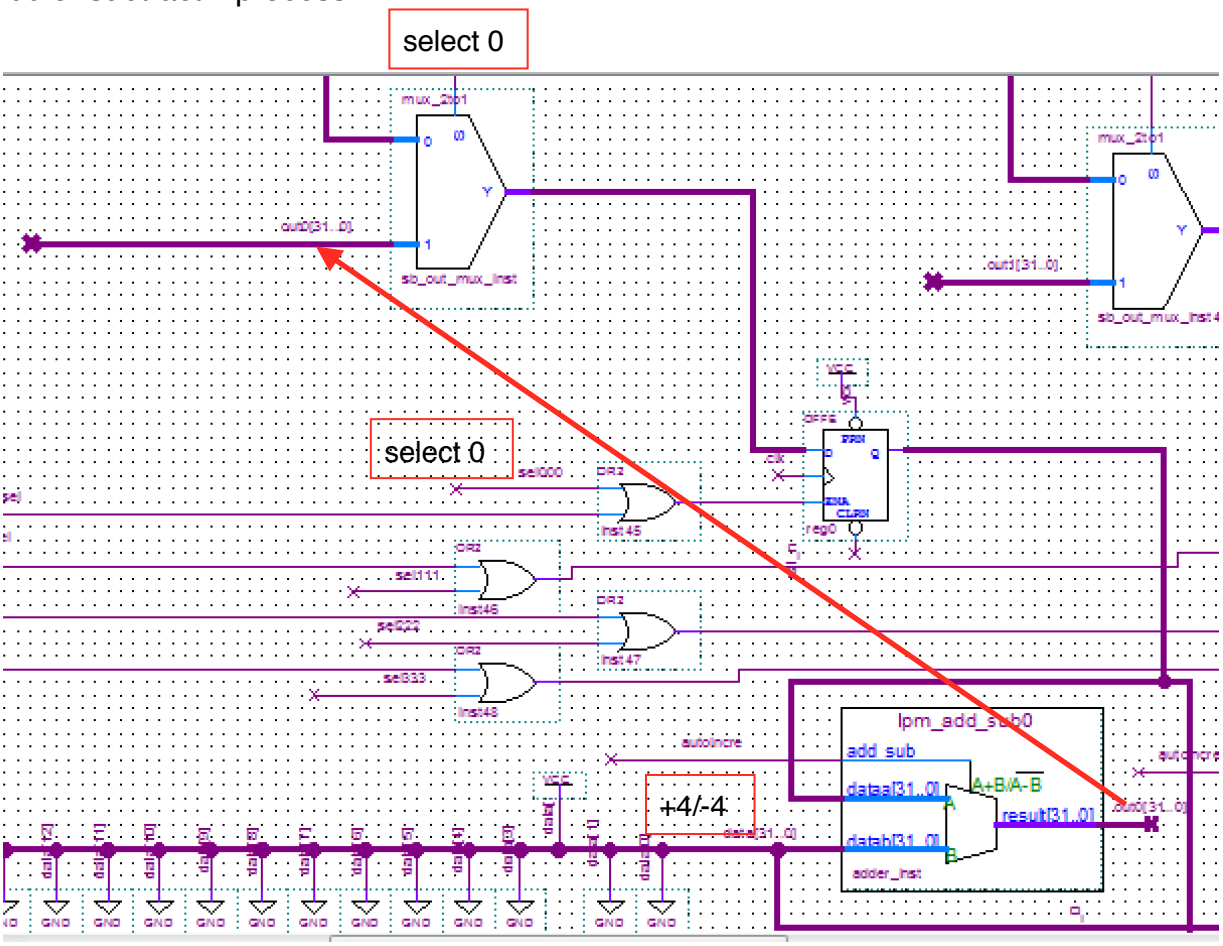
wen_rt && auto-increase && rt [1..0] | | wen_rt && auto-decrease && rt [1..0]

wen_rt is from the last level, which is to encode based on the rt address to select rt.

The result is select0, select1, select2, select3.



Add or subtract 4 process:



As we can see through above picture, the addition result and subtraction result will be back to mux_2_to_1. And select_0, select_1, select_2, select_3 will select whether the addition, subtraction result or the data_in will be chosen. Also, because we want to assign Rs and Rt value, we should enable two register to let data in. We have data_enable, rt_enable (select 0), which I add to encode rt address.

For Rt path, Rt -> out from d flipflop

Rt = Rt + 4 based on select signal and enable signal

for the RS path: [Rs] -> out from d flipflop

data_in —> Rs register

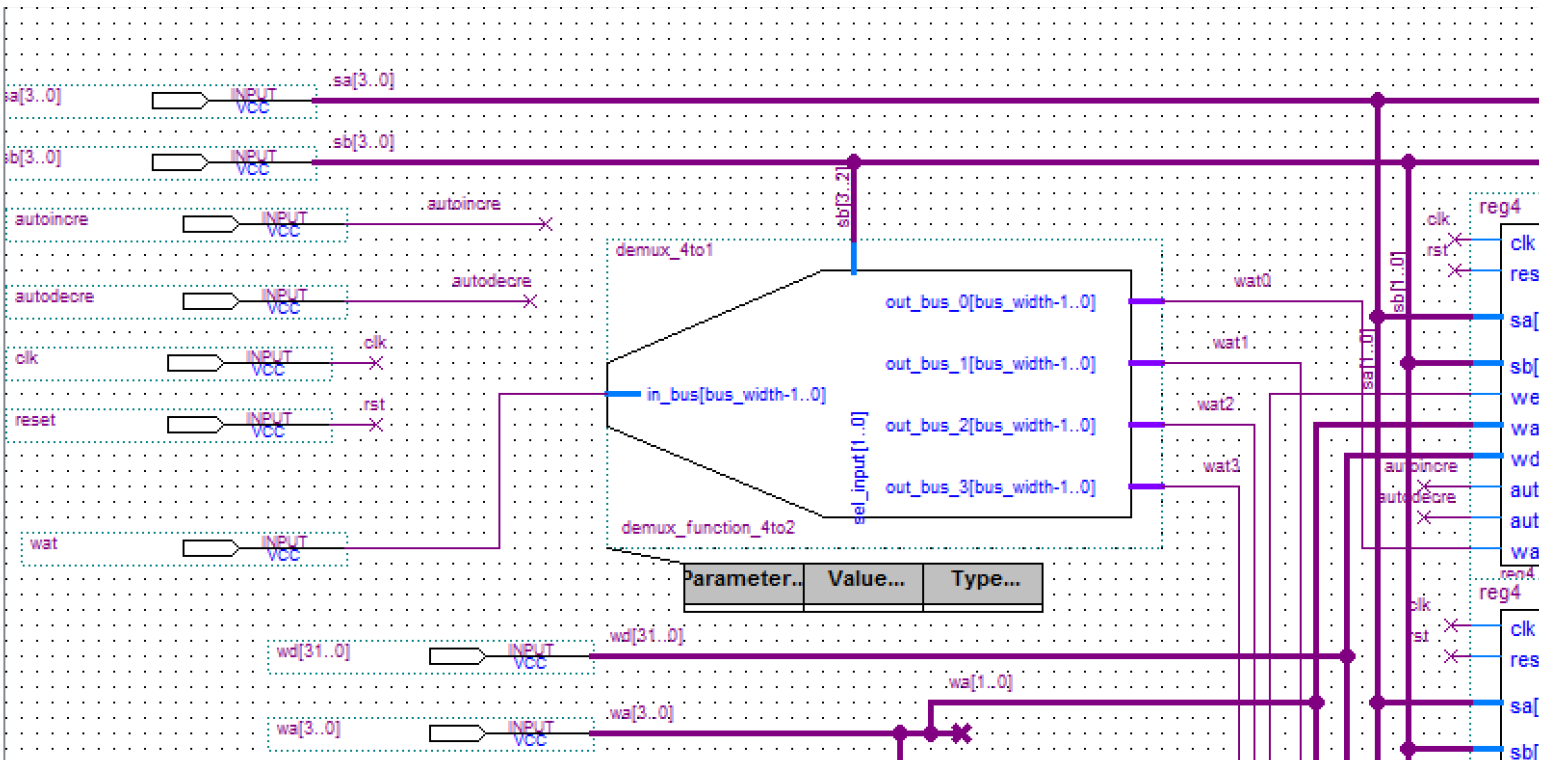
For the lpm_add_sub0 module, I add the auto increase signal to add_sub input for the correct adding and subtracting process.

In reg 16: encode the Rt to reg4

We should add an enable signal with the Rt address because we want to write to Rt.

And if enable signal is 1, this means the the address is matched in the lest level reg 16.

This is same to the process of writing to Rd.

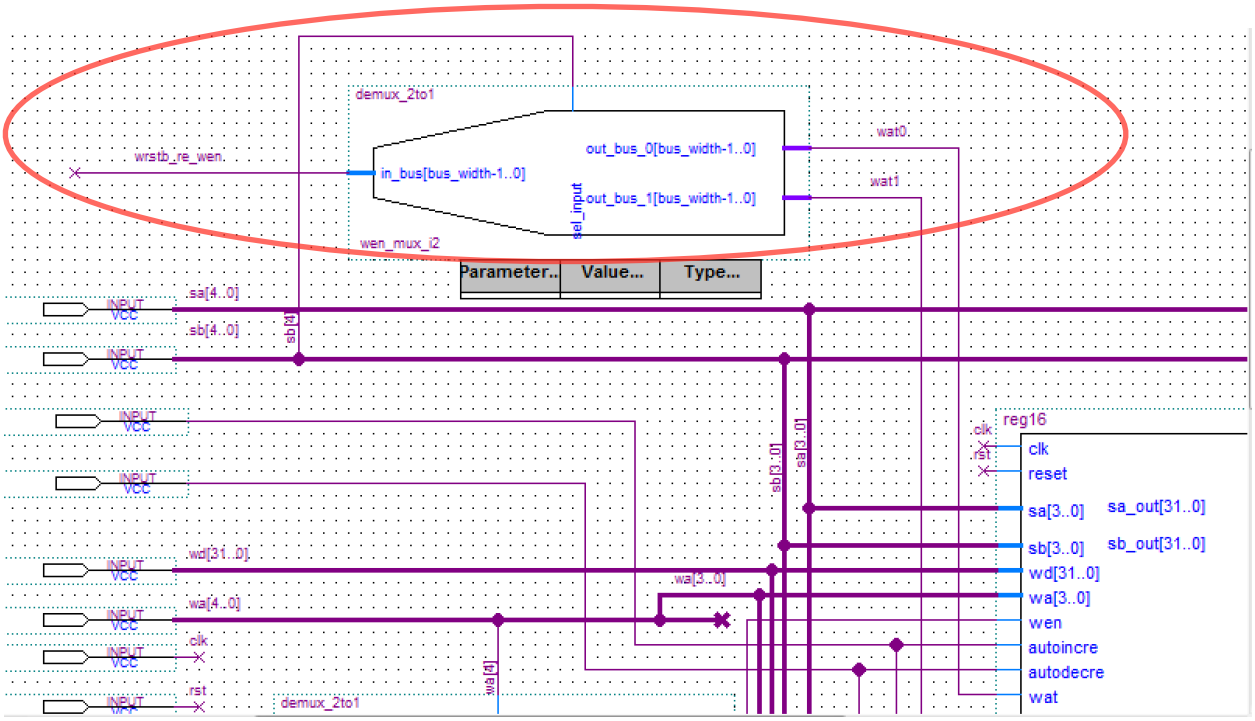


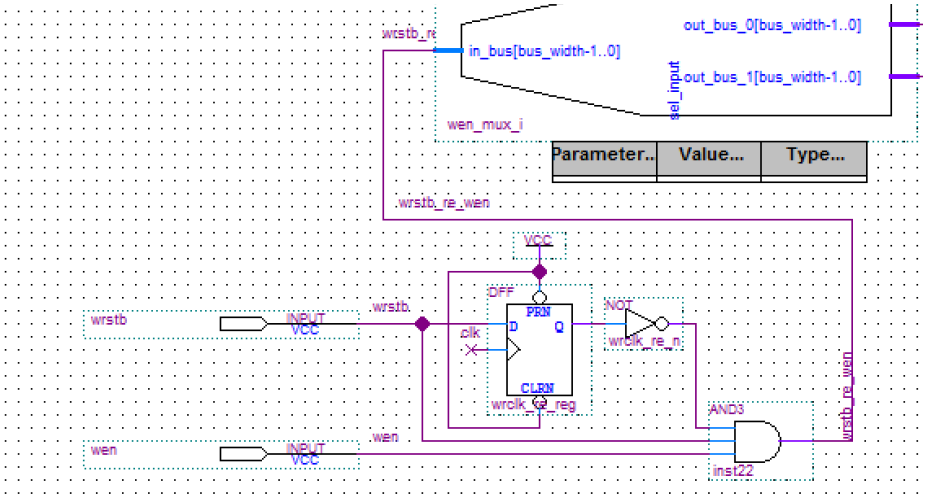
In the register file:

We should add an enable signal with the Rt address because we want to write to Rt.

This is same to the process of writing to Rd.

And we should connect the in_bus pole to wrsb_re_wen to ensure that the signal I write in this clk will be wrote to the register in the next clk.

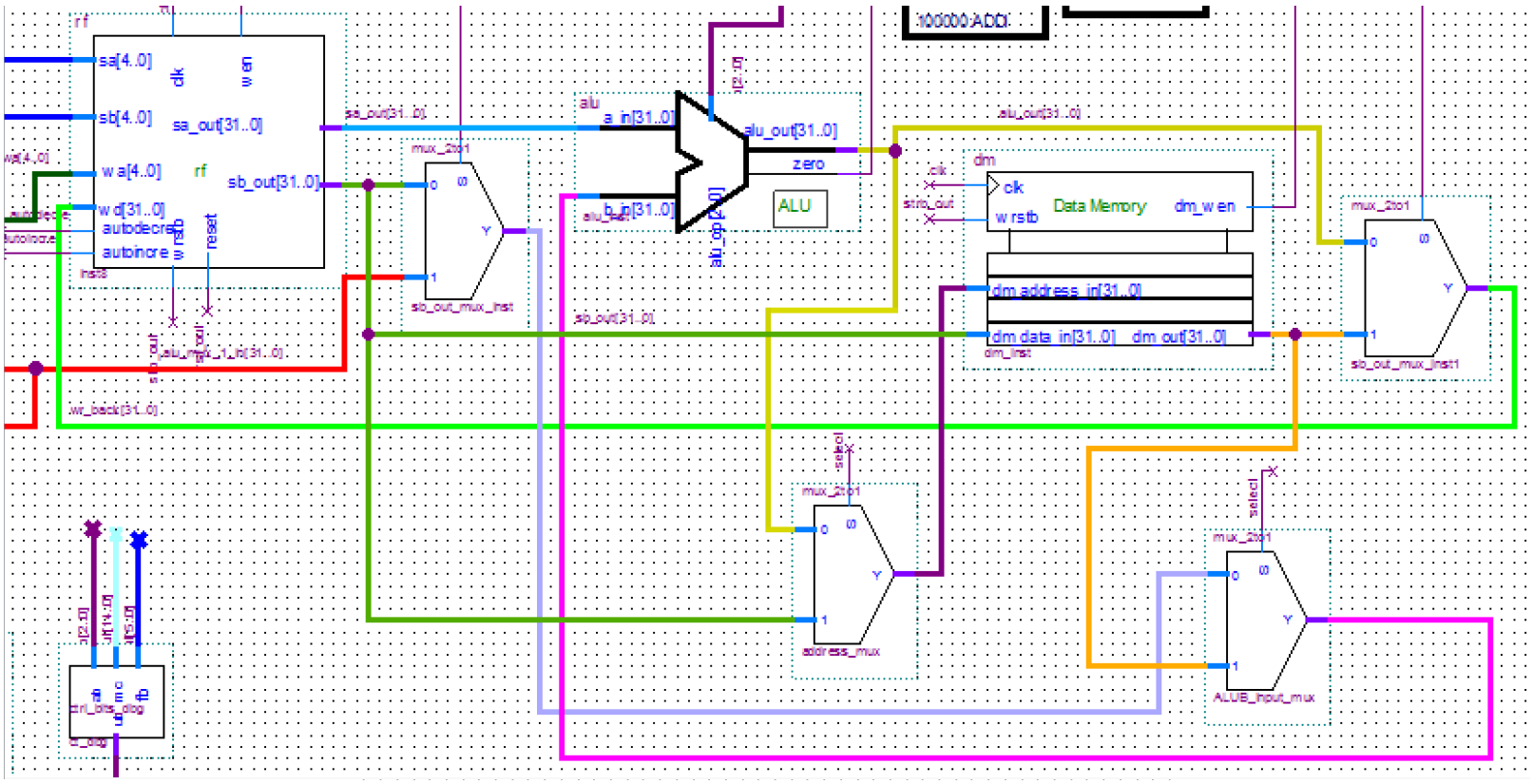


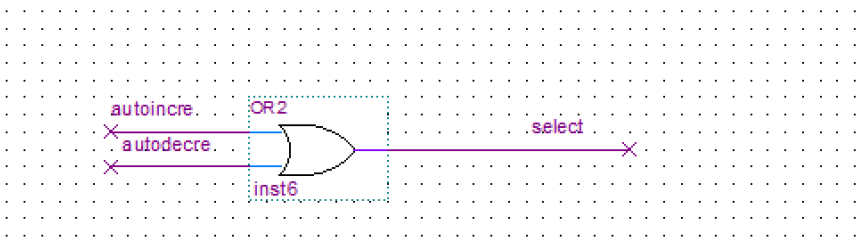


At the top level, we can see that the output of sb_out will pass by the mux2to1. The select signal is autoincre I autodecre. Because they are all load data memory to ALU and return to register file. The select signal will select whether pass by the sb_out to data memory address. And select signal also choose whether the input of ALU_B will be from data memory out or sb_out. At the auto increment and auto decrement mode, we should all set select to 1 to finish R1 <- R1 +M[R2].

The path is:

sb_out -> data memory address -> data memory output -> ALU output -> register file





Testing of auto increment mode:

DRAM: path: midterm_YiQin -> Part3_testing_file -> autoincre -> dram_autoincre.mif

```
CONTENT BEGIN
    000       :    00000000000000000000000000000000;
    001       :    00000000000000000000000000001000;
    002       :    00000000000000000000000000000100;
    003       :    00000000000000000000000000000011;
    004       :    00000000000000000000000000000010;
END;
```

IRAM: Path: midterm_YiQin -> Part3_testing_file -> autoincre -> iram_autoincre.mif

```
CONTENT BEGIN
--Hex Address :   bit31.........................bit0;
--   |             |                             |
    000       :    10001100000000000000000000000000;
--                 |____||___||___||_____________|
--                   |     |    |          |
--                   lw, rs=0, rt=0,   offset=0
-- This is a lw instructioni. It loads r0 with data from
-- data memory location 0. Data memory location 0 is
-- preloaded with 0, see DRAM.mif.
--
    001       :    10001100000000010000000000000100;
--                 |____||___||___||_____________|
--                   |     |    |          |
--                   lw, rs=0, rt=1,   offset=4
-- This is a lw instructioni. It loads r1 with data from
-- data memory location 1. Data memory location 1 is
-- preloaded with 8, see DRAM.mif.

-------------------------------------------------------------
--Hex Address :   bit31..........bit15.........bit0;
--   |             |                |              |
    002       :    00000000000000010000000000010000;
--                 |____||___||___||___||___||____|
--                   |     |    |    |    |    |
--                  R-type,rs=0,rt=1,rd=0,---,f=autoincre
--  002     R1 = R1 +[R0], autoincre R0 <- 4 + 4

-------------------------------------------------------------
--Hex Address :   bit31..........bit15.........bit0;
--   |             |                |              |
    003       :    00000000000000010000000000010000;
--                 |____||___||___||___||___||____|
--                   |     |    |    |    |    |
--                  R-type,rs=0,rt=1,rd=0,---,f=autoincre
--  003     R1 = R1 +[R0], autoincre R0 <- 4 + 4 + 4
-------------------------------------------------------------
--Hex Address :   bit31..........bit15.........bit0;
--   |             |                |              |
    004       :    00000000000000010000000000010000;
--                 |____||___||___||___||___||____|
--                   |     |    |    |    |    |
--                  R-type,rs=0,rt=1,rd=0,---,f=autoincre
--  004     R1 = R1 +[R0], autoincre R0 <- 4 + 4 + 4 + 4

END;
```

The result through **signal tap testing:**

Node			Segment	0	1	2	3	4	5	6
Type	Alias	Name	Value	0	1	2	3	4	5	
	Program_Counter_Output	acq_data_in[0..31]	00000000h	00000000h	00000004h	00000008h	0000000Ch	00000010h		
	Instruction_Memory_Output	acq_data_in[32..63]	8C000000h	8C000000h	8C010004h		00010010h			
	Register_File_A_Output	acq_data_in[64..95]	00000000h	00000000h			00000004h	00000007h		
	Register_File_B_Output	acq_data_in[96..127]	00000000h	00000000h		00000008h	0000000Ch	00000010h		
	ALU_Output	acq_data_in[128..159]	00000000h	00000000h	00000004h		00000007h	00000009h		
	Data_Memory_Output	acq_data_in[160..191]	00000000h	00000000h	00000008h	00000004h	00000003h			
	Write_Back_MUX_Output	acq_data_in[192..223]	0	0	8	4	7	9		
	ALUOP	acq_data_in[224..225]	lw|sw|addi,ADD	lw|sw|addi,ADD					R-t	
	REGWRITE	acq_data_in[226]	1							
	ALUSRC	acq_data_in[227]	1							
	BRANCH	acq_data_in[228]	0							

Data | Setup

The initial value of rt is 008h and will be incremented by 4, you can see the register B output. And the rs will increased by the rs + Memory[rt], Memory[rt+4], Memory[rt+8]. The initial value of rs is 8.

I store 4,3,2 in dram address 002, 003, 004.

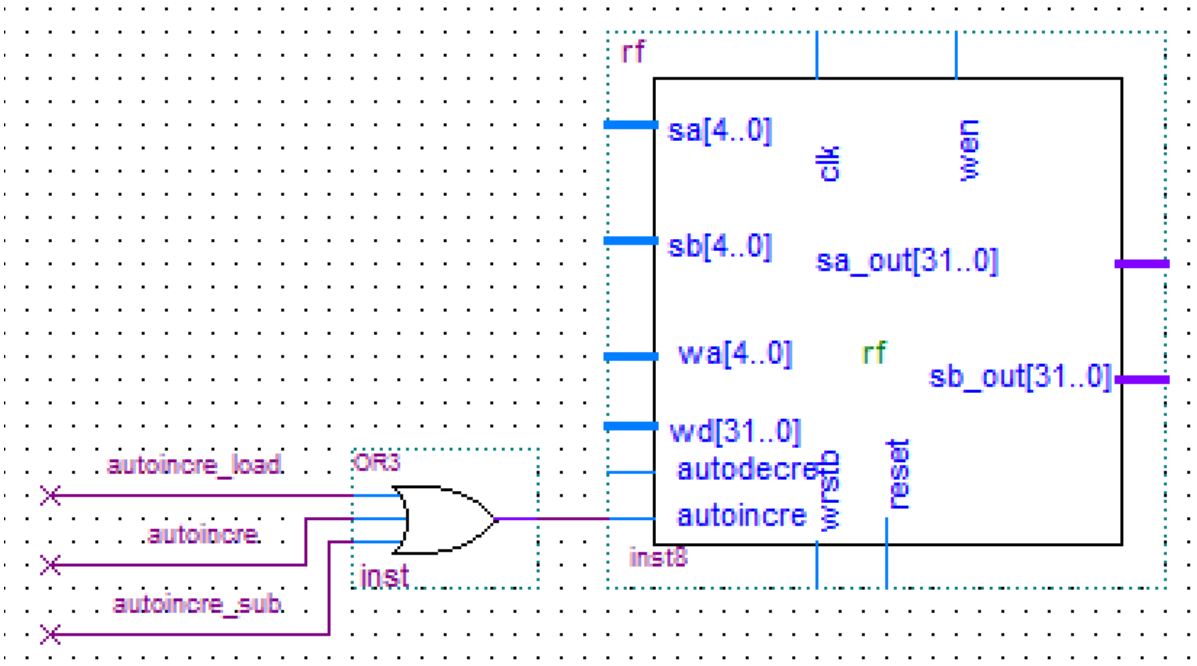
So, you can see the result is 0 -> 0+4 (002)=4 -> 4 +3 (003)=7 -> 7+2 (004)= 9

The result is correct

Other instruction in this mode:

For other instruction, such as subtract, load and so forth, we just modify the control signal because of different opcode or function number[5:0]. And the operation except the register file is same to the instruction in normal register mode.

For the register file, we just add an or gate, to do the operation Rt = Rt + 4, which means Rt will add 4 to itself as long as it is in auto-increment mode



3. Auto-decrement Mode

Also, I take the add instruction in this mode as example and others are almost same in register file, just control unit is different for opcode

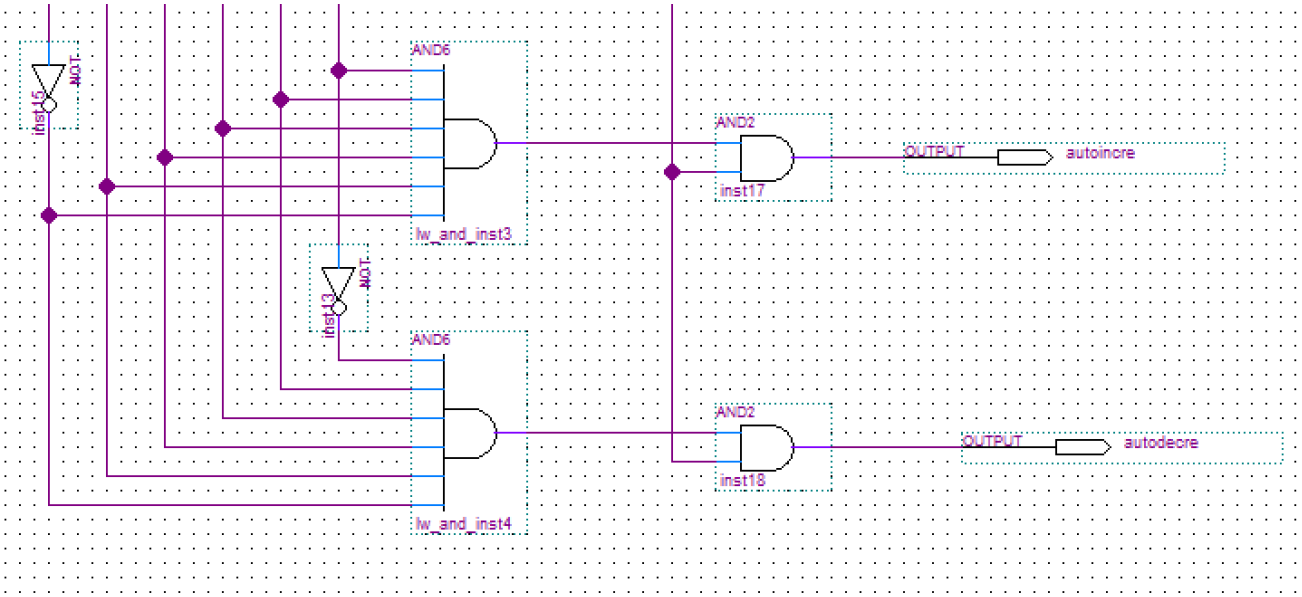
```
Add R1,-(R2):
R2 <- R2 - 4
R1 <- R1 +M[R2]
```

Firstly, decrease R2 by 4, then add the r1 with data memory output and the address of data memory output is [R2+4]

opcode	rs	rt	rd	—-	index
000000	—-	—-	——	00000	010001

The design is almost same to auto increment but some different.

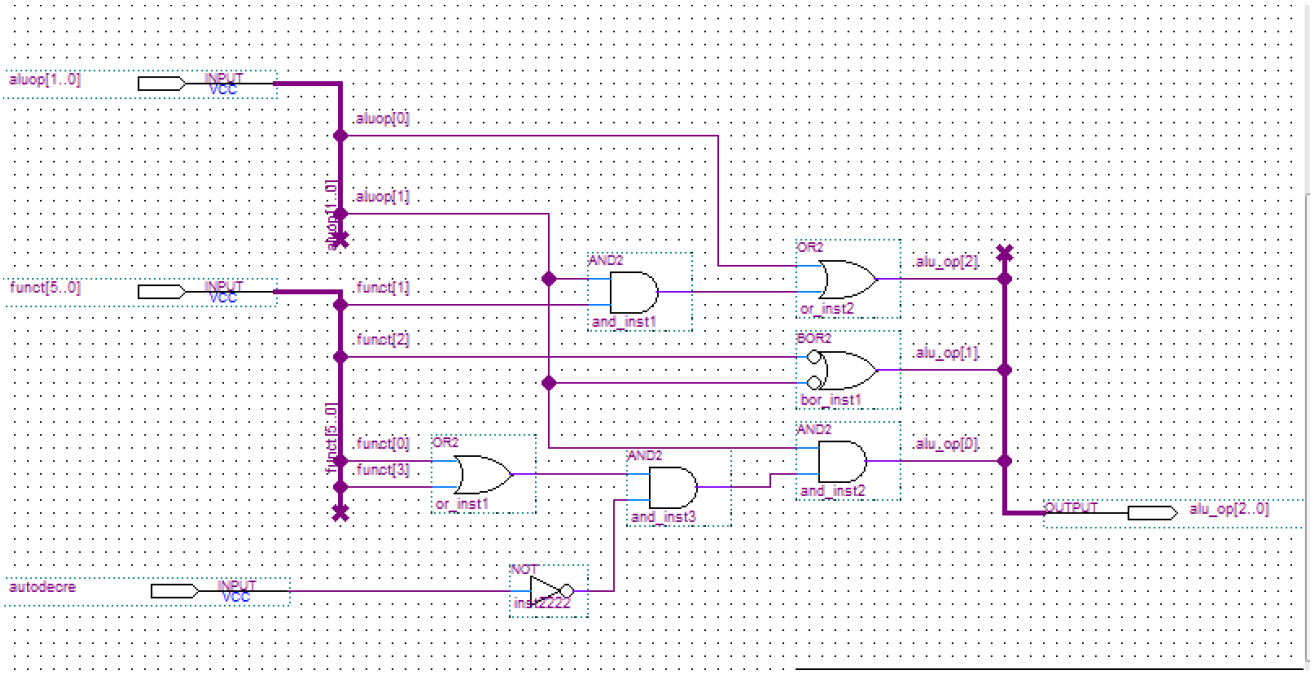
Control Unit:



Because the function number is 010001, the auto decrement control signal should be designed based on this

For the lest 4 significant function code are not all 0, we should add a nor gate at ALU

ALU_Ctrl module:



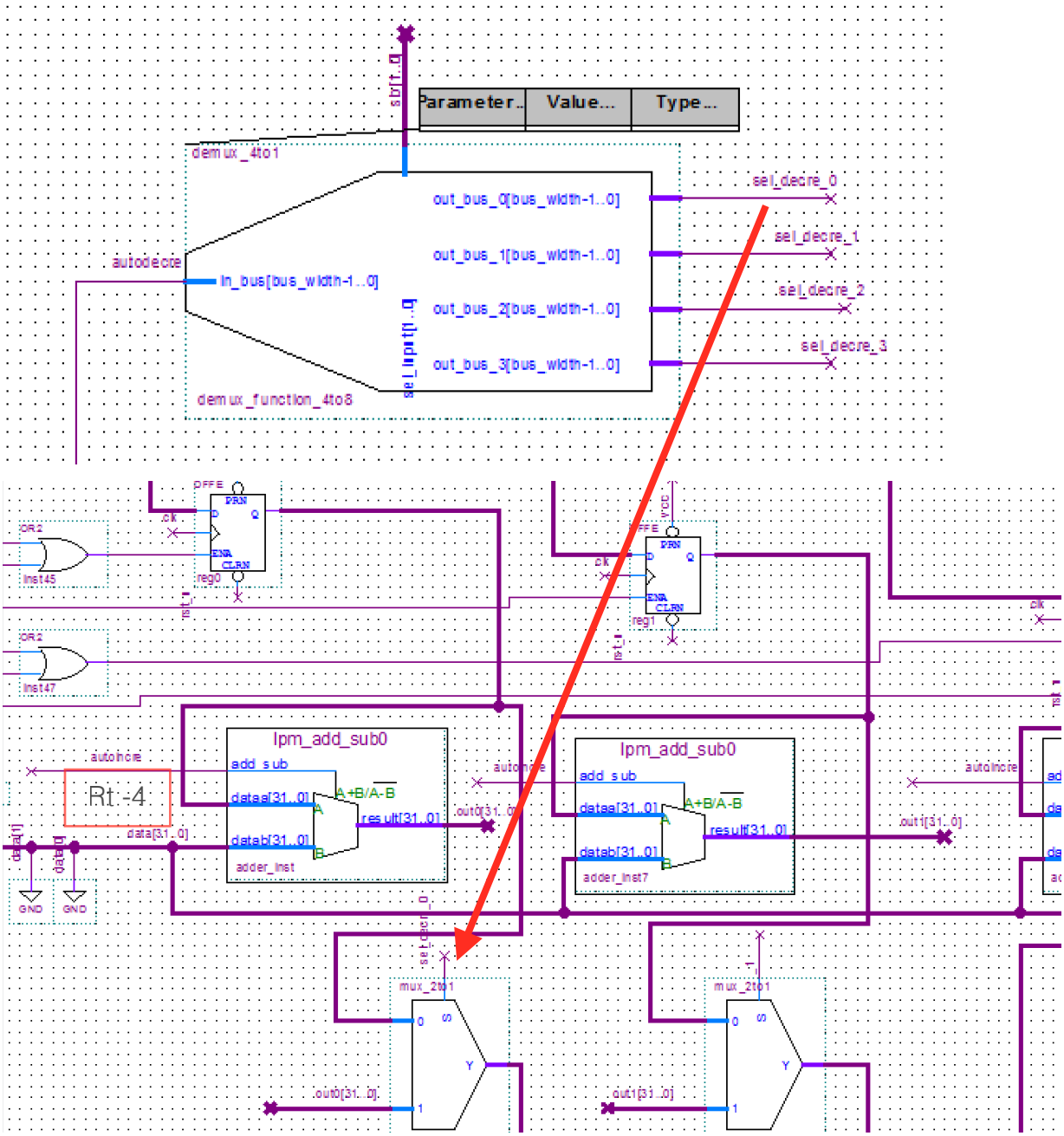
Add a nor gate to verify the ALU is in adding mode. I make sure that the least bit of function number will not influence the adding mode.

The other parts are all same to increment module, only in reg4, there are some difference.

At reg4 module:

We add signal select_decre_0, select_decre_1, select_decre_2, select_decre_3, which is produced by the auto decrement signal from control unity and the address of Rt address. Why it do not need to pass by enable?

Because the output should of register of Rt should be decreased by 4 immediately, the data memory will get the [Rt] - 4 value not the [Rt]. Because the input of 4to1 mix is the least two bit of Rt and the auto decrease signal, which are not passed by the wrstb structure. So, the mux should be selected to 1 immediately in auto decrement mode.



So, we add a mux2to1 after the output of register. And another input is the subtraction result. Select signal is auto decrease from control unit. So, when the auto decrease is 1, R2 <- R2 - 4 and the output of sb_out will be R2-4

The other parts are same to auto increment process.

Testing of auto decrement:

IRAM: path: midterm_YiQin -> Part3_testing_file -> autodecre -> iram_autodecre.mif

```
CONTENT BEGIN
--Hex Address :   bit31.........................bit0;
--   |             |                             |
    000       :    10001100000000000000000000000000;
--                |____||___||___||_______________|
--                  |     |    |          |
--                  lw, rs=0, rt=0,   offset=0
--
--                  load 0 to r0
-------------------------------------------------------------
    001       :    10001100000000010000000000000100;
--                |____||___||___||_______________|
--                  |     |    |          |
--                  lw, rs=0, rt=1,   offset=4
--
--                  load 16 to r1
-------------------------------------------------------------
--Hex Address :   bit31...........bit15.........bit0;
--   |             |                 |              |
    002       :    00000000000000010000000000010001;
--                |____||___||___||___||___||____|
--                  |     |    |    |    |    |
--                 R-type,rs=0,rt=1,rd=0,---,f=autodecre
--  002     R0 = R0 -4, R1 = R1 + [R0 -4]

-------------------------------------------------------------
--Hex Address :   bit31...........bit15.........bit0;
--   |             |                 |              |
    003       :    00000000000000010000000000010001;
--                |____||___||___||___||___||____|
--                  |     |    |    |    |    |
--                 R-type,rs=0,rt=1,rd=0,---,f=autodecre
--  003     R0 = R0 -4, R1 = R1 + [R0 -4]
-------------------------------------------------------------
--Hex Address :   bit31...........bit15.........bit0;
--   |             |                 |              |
    004       :    00000000000000010000000000010001;
--                |____||___||___||___||___||____|
--                  |     |    |    |    |    |
--                 R-type,rs=0,rt=1,rd=0,---,f=autodecre
--  004     R0 = R0 -4, R1 = R1 + [R0 -4]
```

DRAM: path: midterm_YiQin -> Part3_testing_file -> autodecre -> dram_autodecre.mif

```
CONTENT BEGIN
    000       :    00000000000000000000000000000000;
    001       :    00000000000000000000000000010000;
    002       :    00000000000000000000000000000100;
    003       :    00000000000000000000000000000011;
    004       :    00000000000000000000000000000010;
END;
```

The result shows auto decrement process is right.

Node			Segment						
Type	Alias	Name	Value						
	Program_Counter_Output	acq_data_in[0..31]	00000000h	00000000h	00000004h	00000008h	0000000Ch	00000010h	
	Instruction_Memory_Output	acq_data_in[32..63]	8C000000h	8C000000h	8C010004h		00010011h		
	Register_File_A_Output	acq_data_in[64..95]	00000000h		00000000h		00000003h	00000007h	
	Register_File_B_Output	acq_data_in[96..127]	00000000h	00000000h		0000000Ch	00000008h	00000004h	
	ALU_Output	acq_data_in[128..159]	00000000h	00000000h	00000004h	00000003h	00000007h	00000017h	
	Data_Memory_Output	acq_data_in[160..191]	00000000h	00000000h	00000010h	00000003h	00000004h	00000010h	
	Write_Back_MUX_Output	acq_data_in[192..223]	0	0	16	3	7	23	
	ALUOP	acq_data_in[224..225]	lw|sw|addi,ADD	lw|sw|addi,ADD					R-t
	REGWRITE	acq_data_in[226]	1						
	ALUSRC	acq_data_in[227]	1						
	BRANCH	acq_data_in[228]	0						

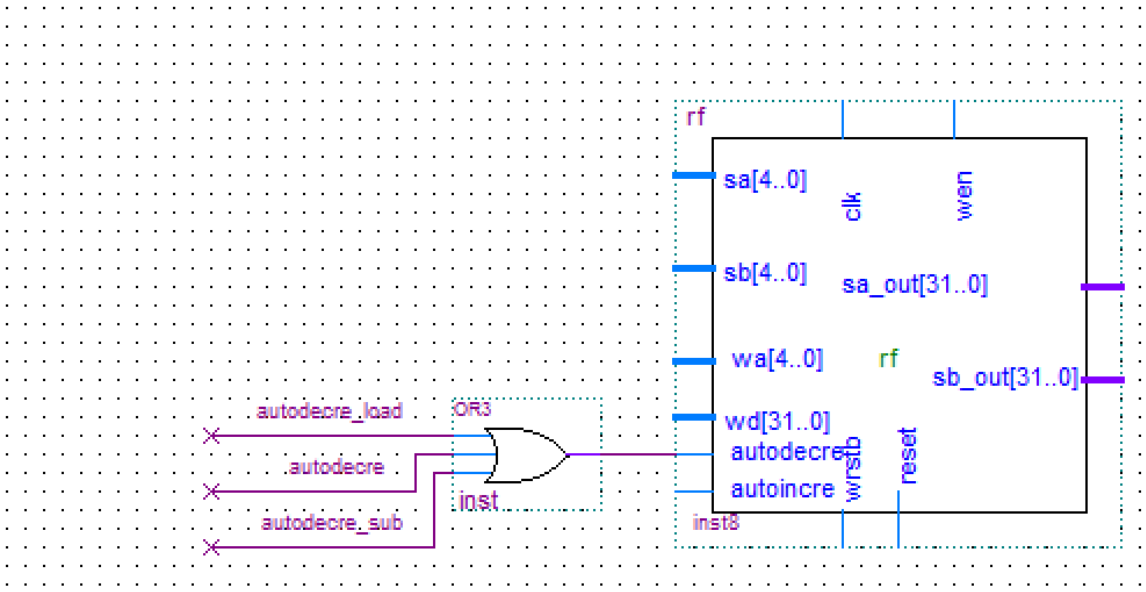
The [Rs] = 16, it will increase by M[Rt-4], M[Rt-8],M[Rt-12], so the value is

0 -> 0+3 (003)=3 -> 3+4 (002) = 7 -> 7+16 (001) = 23, which I store in data ram.

Other instruction in this mode:

For other instruction, such as subtract, load and so forth, we just modify the control signal because of different opcode or function number[5:0]. And the operation except the register file is same to the instruction in normal register mode.

For the register file, we just add an or gate, to do the operation Rt = Rt - 4, which means Rt will subtract 4 to itself as long as it is in auto-decrement mode



Finally, the timeQuest Timing Analyzer is satisfied the frequency requirement.