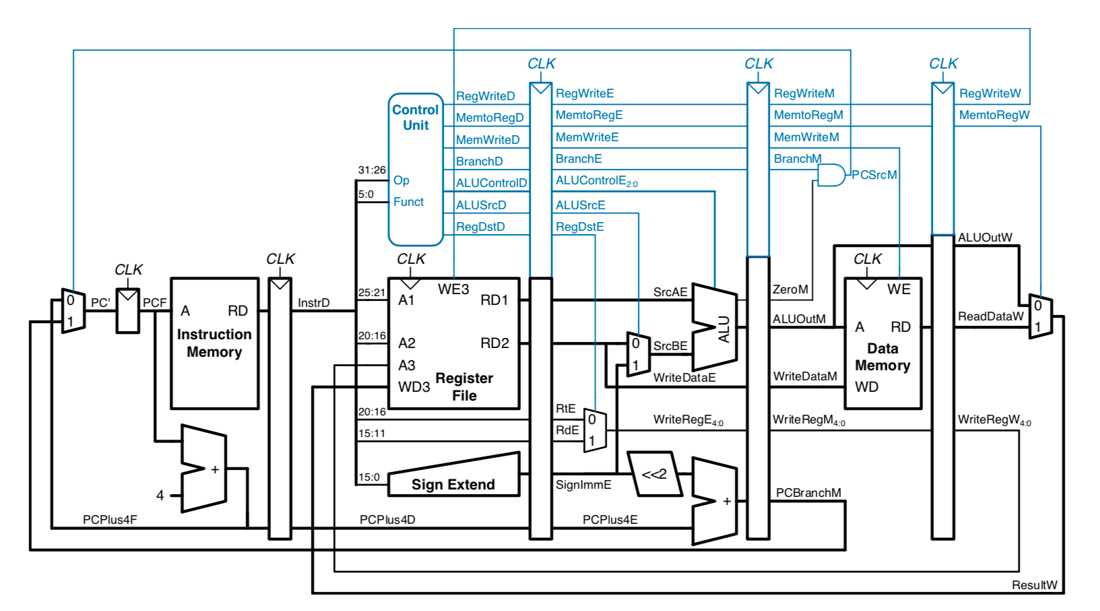
**Project Report**

**李华悦 118010138**

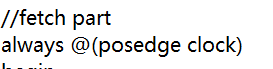
**1 understanding of this project:**

The purpose of this project is to write a MIPS pipeline processor which can execute MIPS instructions with Verilog. We need to design the pipeline for the CPU, which means we need to design the fetch part, decode part, execute part, memory part and writeback part for the CPU and every instruction will be executed in 5 clock. The testbench will first send the instruction and data into the CPU. Then the CPU will execute the instruction in 5 clock and finally put values in corresponding registers and show the output. And we also need to deal with the hazards problems for the pipeline processor.

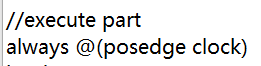
**2 block diagram:**

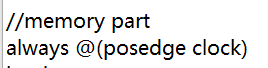


I have used five always in the CPU module to simulate the pipeline processor.



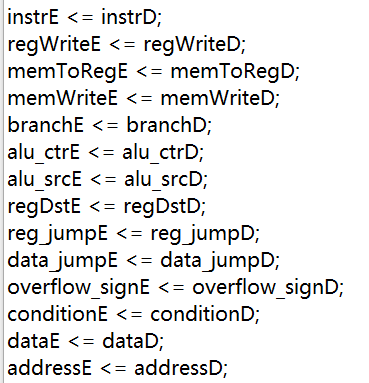




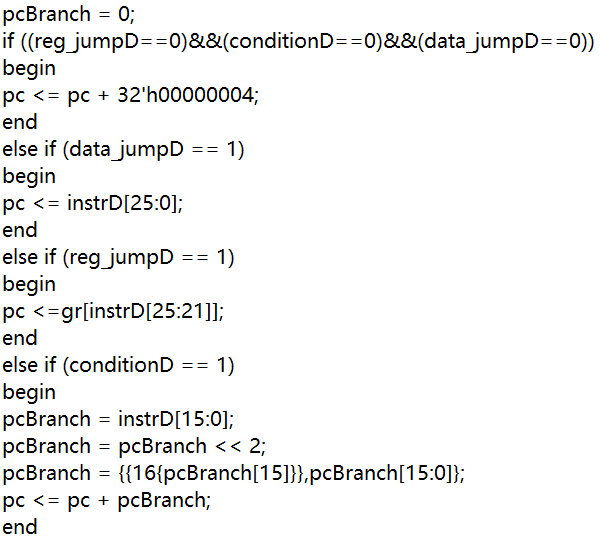




To simulate the pipeline processor, I have designed a lot of registers in the CPU module. I will not explain the functions of all the registers one by one because some registers have the same functions and their difference is just that they work in different stages. I use non-blocking assignments to give the values of registers of the last stage to the registers of the next stage at the end of the last stage like this:



To deal with the problems of hazards, I put 4 NOP instructions in the testbench after every “real” instruction because NOP instructions change nothing for all the values. And I have used my own solution to design the CPU to solve the problem of hazards of pc. I have put the change of pc from stage1 into stage2.



When you run the program and see the display result from terminal. The pc corresponding to every instruction is always the address of that instruction, which means the pc I have shown is the pcf. Therefore, the problem of hazards of pc has been solved.

**3 explanation of all required instructions:**

The requirements of this project are just as follows:

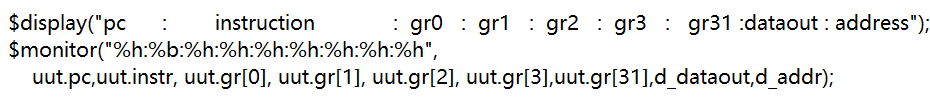
The values in corresponding registers can change rightly after an instruction is executed completely.

The value of pc can correspond to the instruction rightly.

The output of the CPU (d\_addr and d\_dataout) can show the right result for some instructions (lw and sw).

We don’t need to implement the instruction memory and data memory, but we need to show the input and output for the two memory. Therefore, I will use my own instructions as an example in the testbench to show the result of all the instructions.

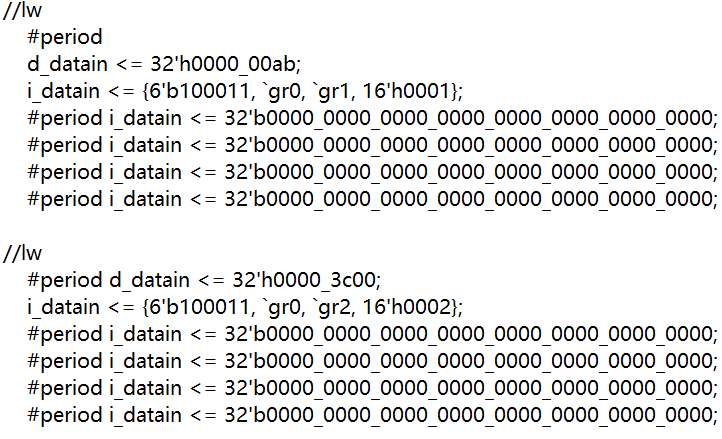
My display in the terminal is as follows:



The pc is always the address that corresponds to the instruction just put in the CPU. The instruction in the display is the instruction that is just put in the CPU. Because the length of the display is limited, I will just use five registers: gr0, gr1,gr2, gr3 and gr31 in my testbench and show the values in them. I will also show the output of the CPU (d\_dataout and d\_addr). To simulate the whole process of MIPS processor, some instructions in my example may repeat to appear. Because the limited length of the report, every kind of instruction will be explain one time and the repeat of that kind of instruction will just show the input and output and will not be explain again. My examples are as follows:

**lw:**

**Input:**



**Output:**

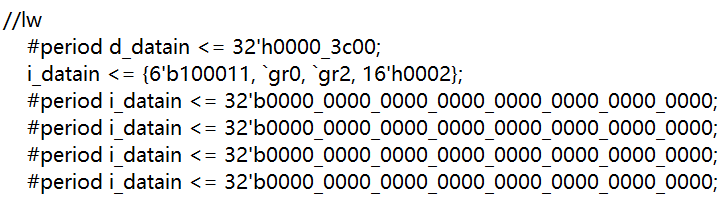




At the start of the program, address=0, so the pc of the first instruction “lw” is 0 and then for every instruction except branch and jump instructions, the pc will plus 4. Therefore, except branch and jump instructions, I will not explain one by one for the pc. The function of this lw instruction is to store the d\_datain in the gr1 register. It is obvious that after 5 clock the value of the gr1 is the d\_datain. And the lw also need to show the output. The d\_dataout is the same as the d\_datain and the d\_addr is equal to (0+1=1) 1. It is obvious that after five clock the d\_dataout and d\_addr show the right output. Therefore, the lw instruction works well in the CPU.

**lw:**

**Input:**



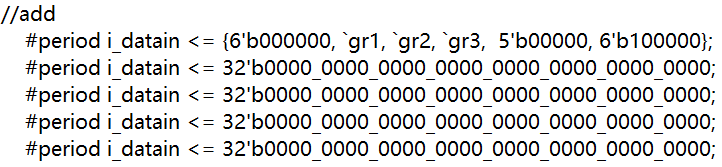
**Output:**





**add:**

**Input:**



**Output:**

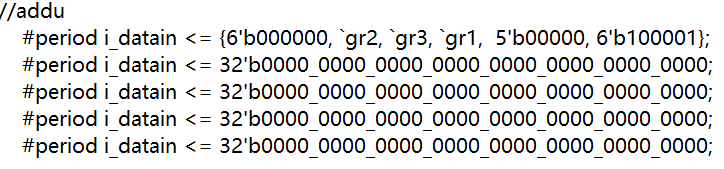




The function of this add instruction is to add the values in gr1 and gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes 3cab (ab + 3c00 = 3cab). It is obvious that the result is right. Therefore, the add instruction works well in the CPU.

**addu:**

**Input:**



**Output:**

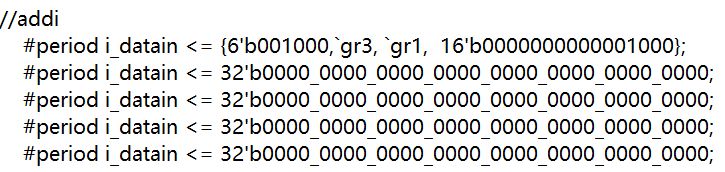




The function of this addu instruction is to add the values in gr2 and gr3 and put the result in gr1. After 5 clock, the value in gr1 becomes 78ab (3c00+3cab=78ab). It is obvious that the result is right. Therefore, the addu instruction works well in the CPU.

**addi:**

**Input:**



**Output:**

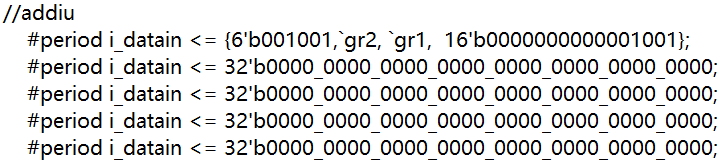




The function of this addi instruction is to add the values in gr3 and imm part of instruction and put the result in gr1. After 5 clock, the value in gr1 becomes 3cb3 (3cab+8=3cb3). It is obvious that the result is right. Therefore, the addi instruction works well in the CPU.

**addiu:**

**Input:**



**Output:**

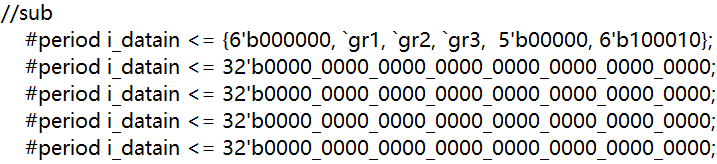


****

The function of this addiu instruction is to add the values in gr2 and imm part of instruction and put the result in gr1. After 5 clock, the value in gr1 becomes 3c09 (3c00+9=3c09). It is obvious that the result is right. Therefore, the addiu instruction works well in the CPU.

**sub:**

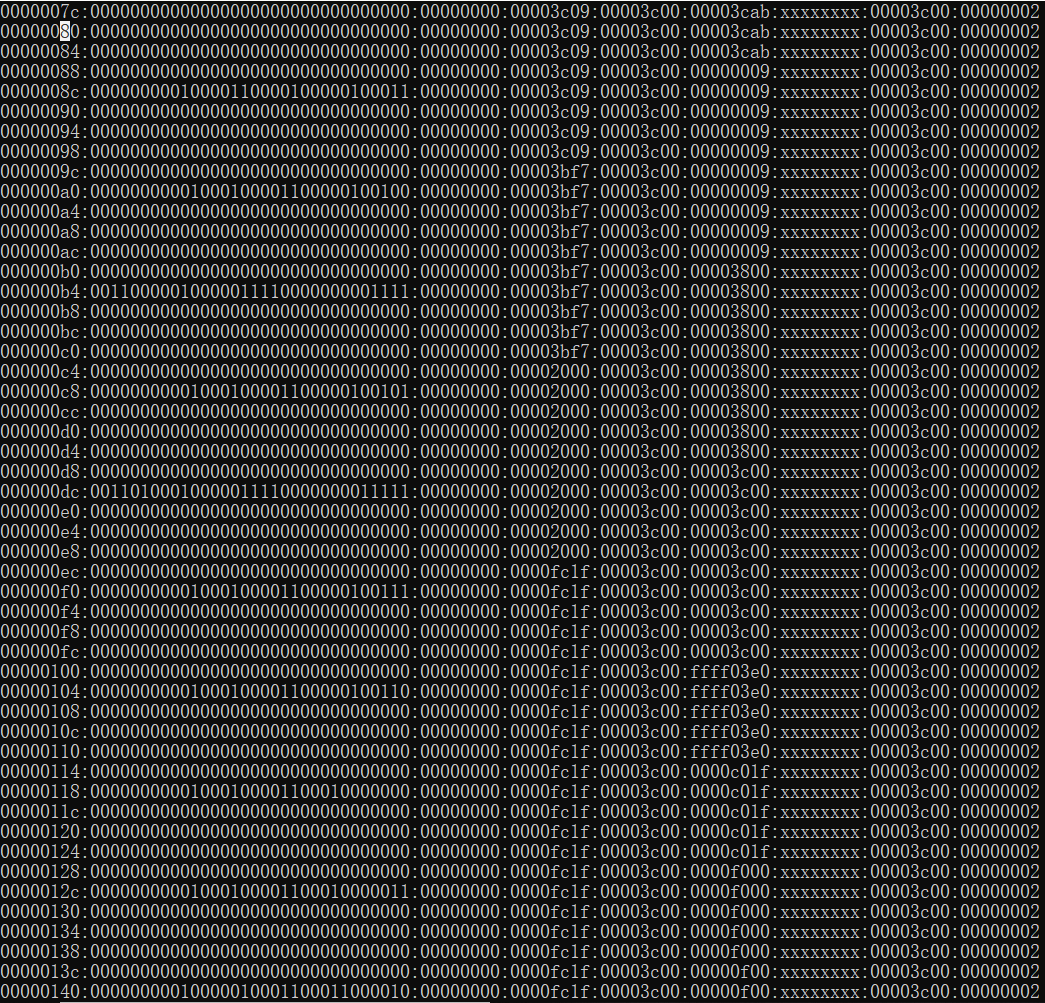
**Input:**



**Output:**



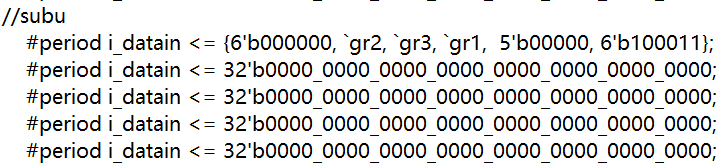
****

****

The function of this sub instruction is to do subtraction with the values in gr1 and gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes 9 (3c09-3c00=9). It is obvious that the result is right. Therefore, the sub instruction works well in the CPU.

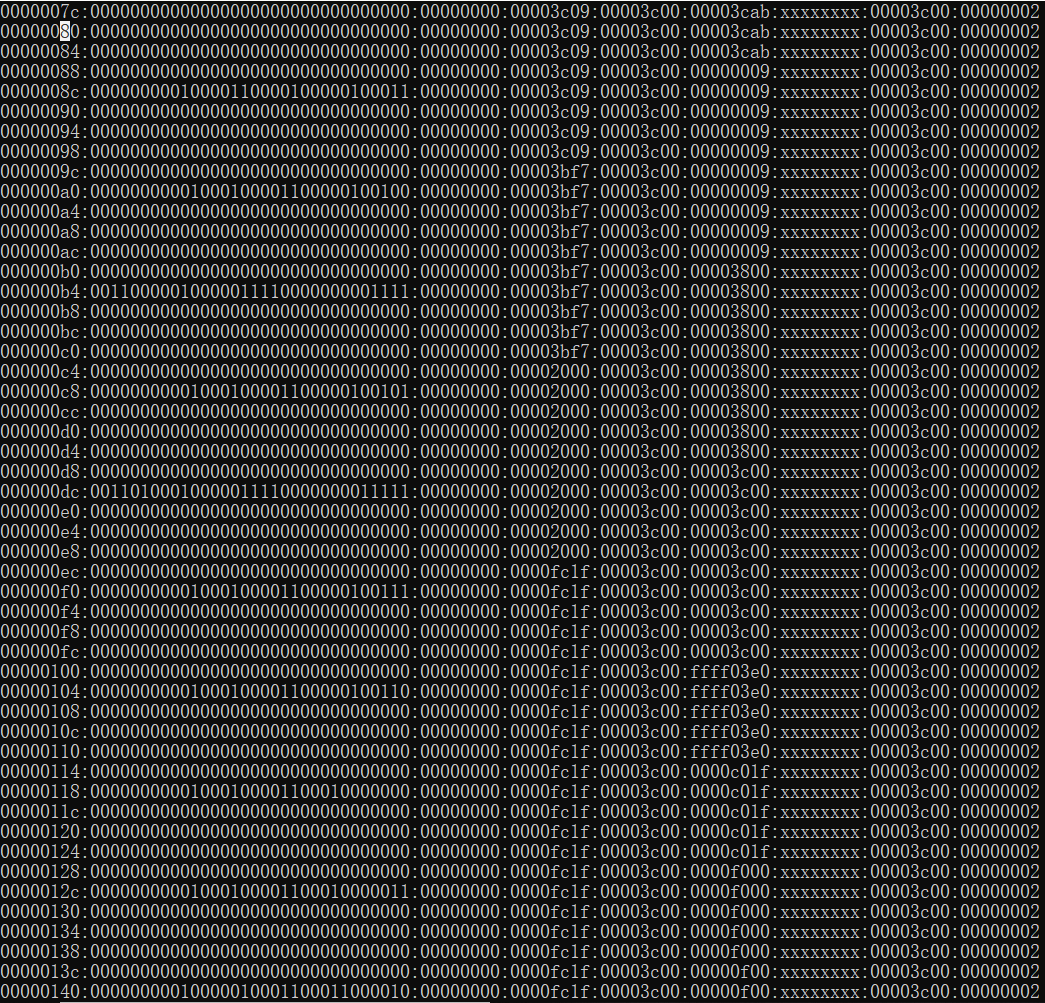
**subu:**

**Input:**



**Output:**

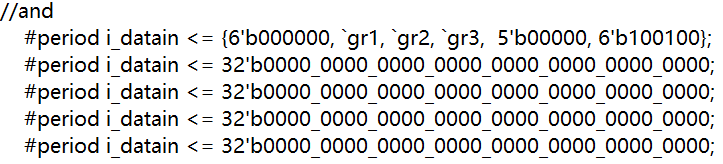


****

The function of this subu instruction is to do subtraction with the values in gr2 and gr3 and put the result in gr1. After 5 clock, the value in gr1 becomes 3bf7 (3c00-9=3bf7). It is obvious that the result is right. Therefore, the subu instruction works well in the CPU.

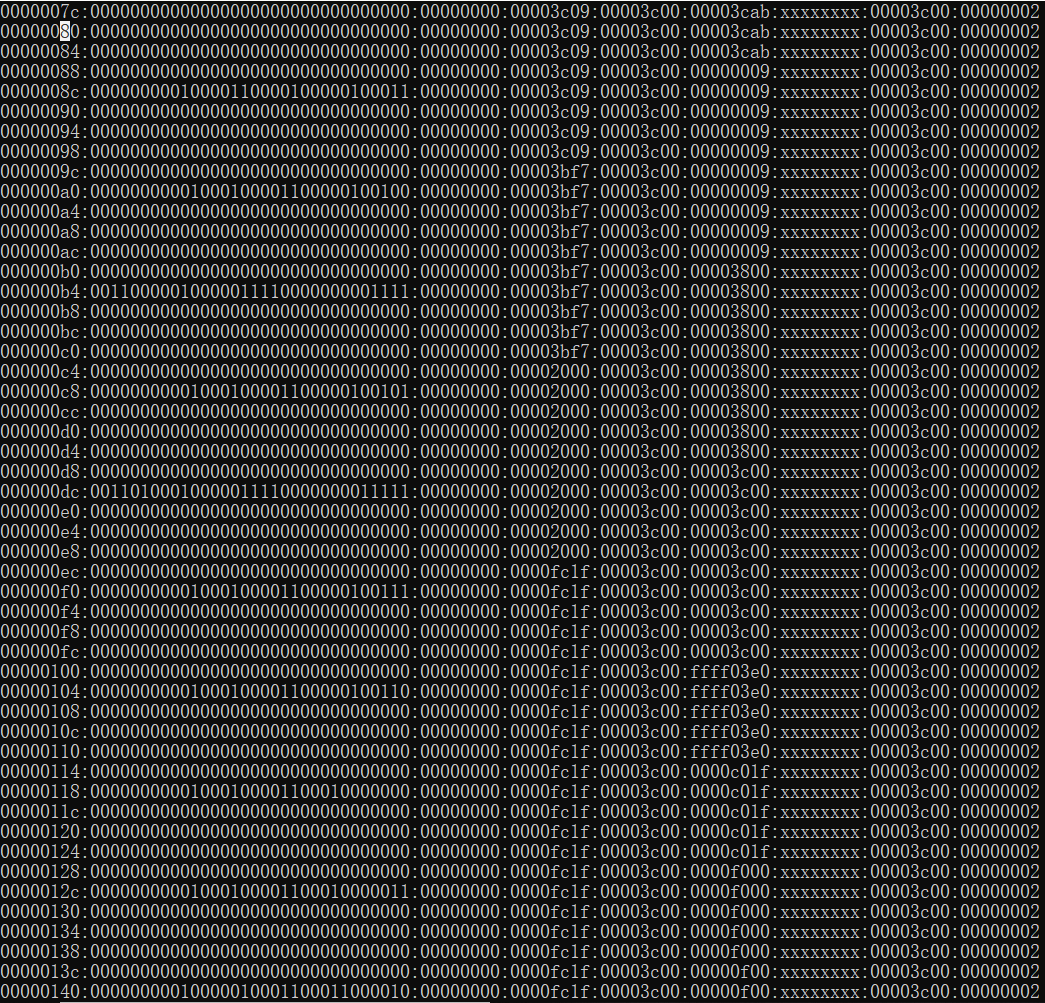
**and:**

**Input:**



**Output:**

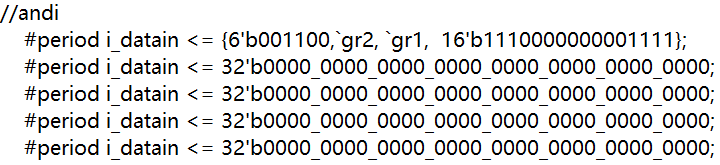


****

The function of this and instruction is to make “and” logic for every bit of the values in gr1 and gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes 3800 (3bf7 & 3c00=3800). It is obvious that the result is right. Therefore, the and instruction works well in the CPU.

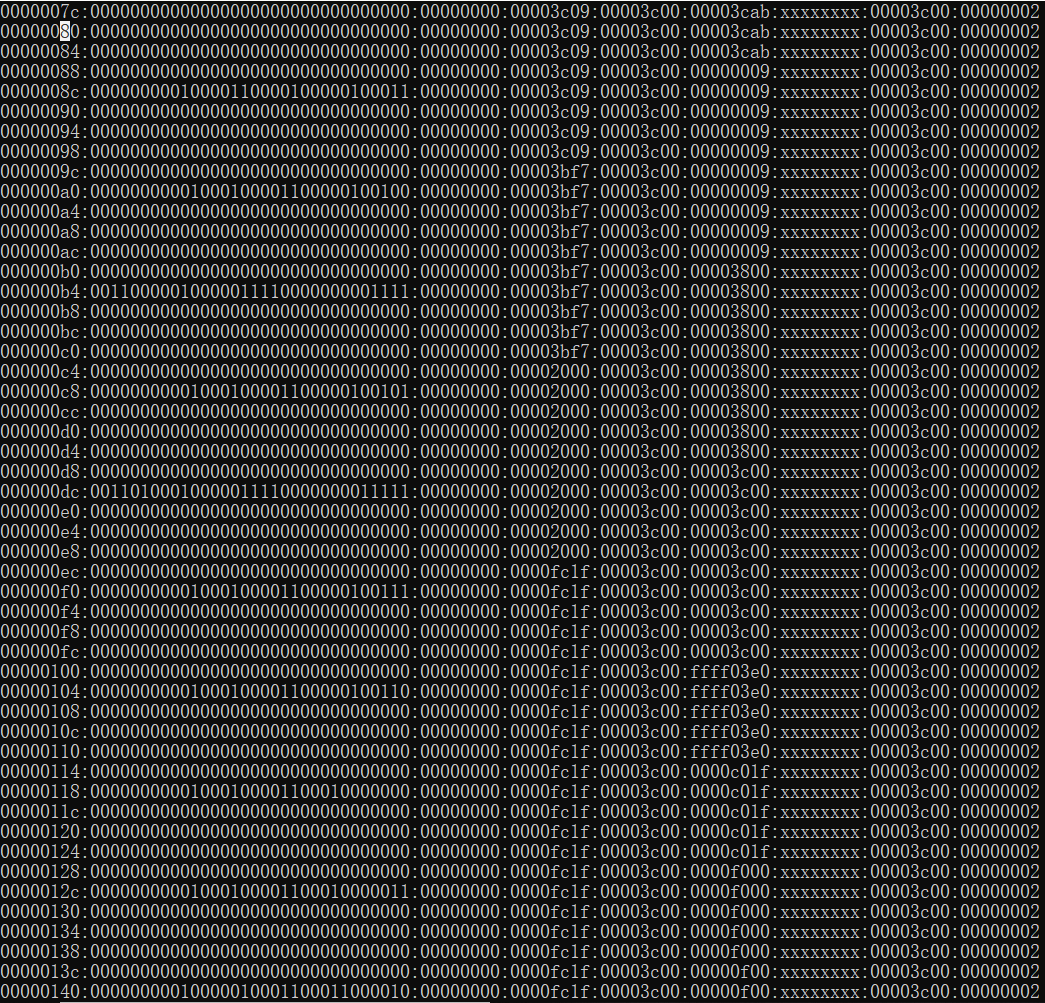
**andi:**

**Input:**



**Output:**

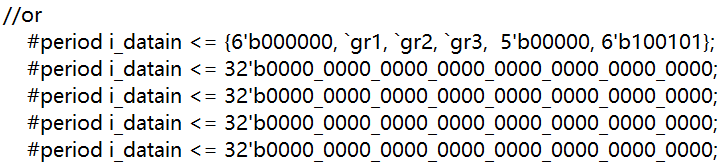


****

The function of this andi instruction is to make “and” logic for every bit of the values in the imm part of the instruction and gr2 and put the result in gr1. After 5 clock, the value in gr1 becomes 2000 (3c00 & f=2000). It is obvious that the result is right. Therefore, the andi instruction works well in the CPU.

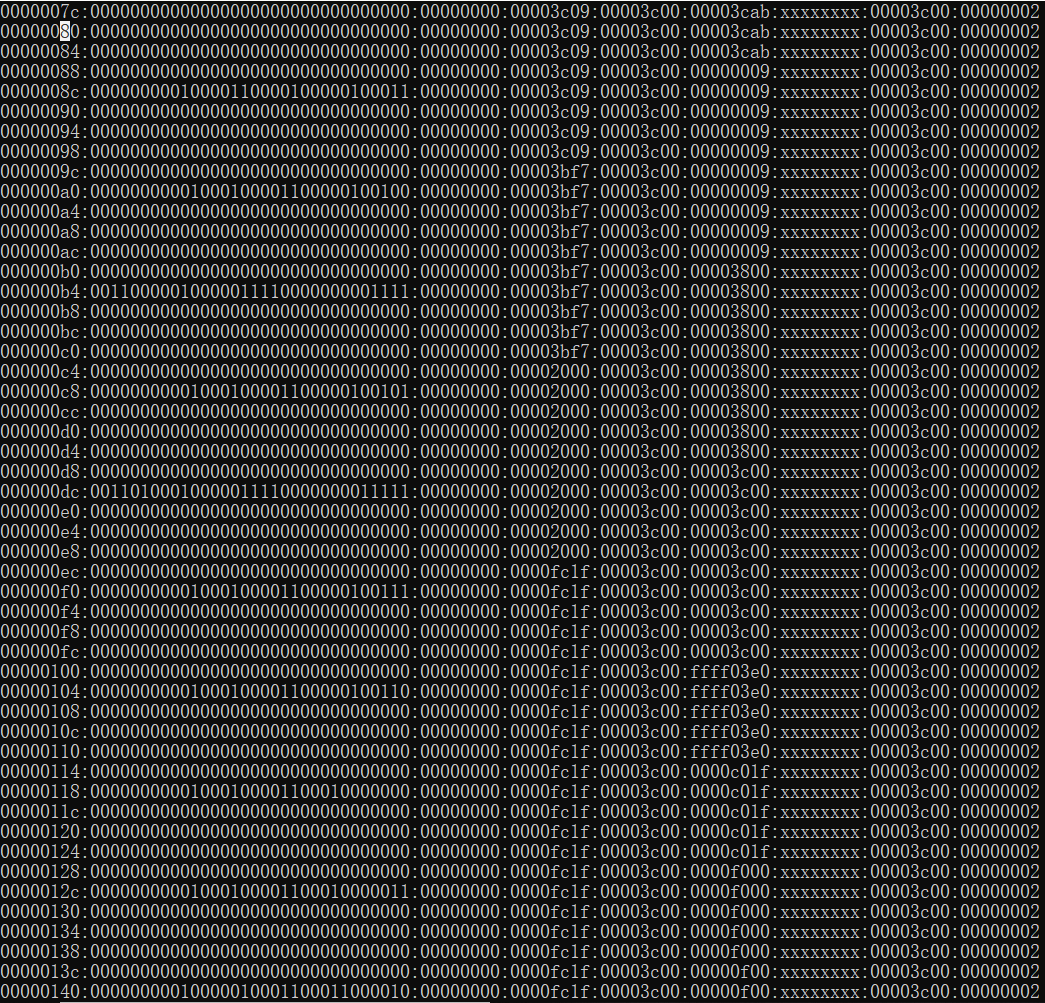
**or:**

**Input:**



**Output:**

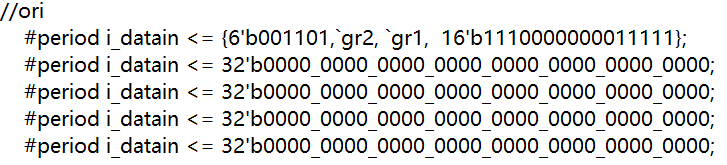


****

The function of this or instruction is to make “or” logic for every bit of the values in the gr1 and gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes 3c00 (2000 or 3c00=3c00). It is obvious that the result is right. Therefore, the or instruction works well in the CPU.

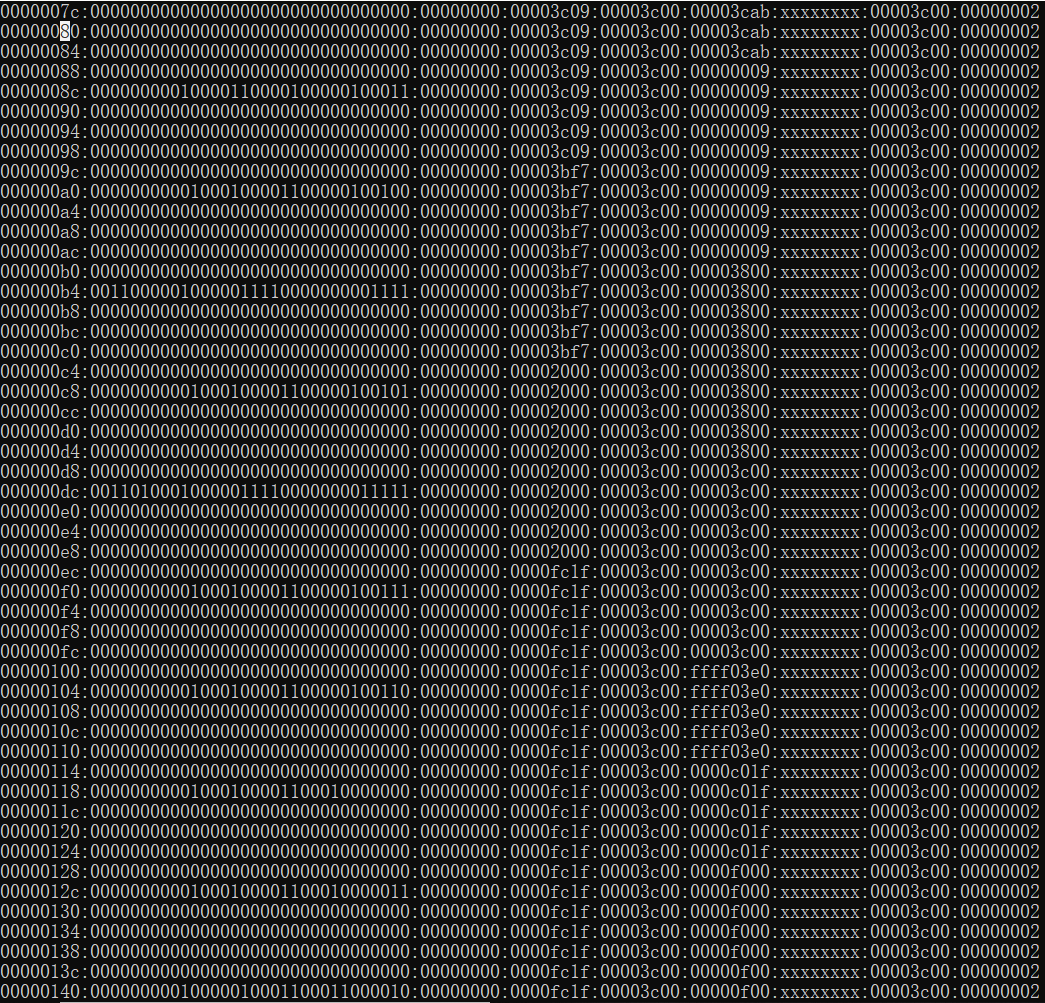
**ori:**

**Input:**

****

**Output:**

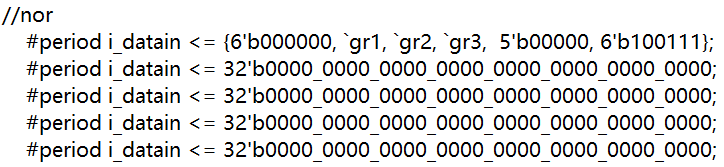


****

The function of this ori instruction is to make “or” logic for every bit of the values in the gr2 and the imm part of the instruction and put the result in gr1. After 5 clock, the value in gr1 becomes fc1f (3c00 or e01f=fc1f). It is obvious that the result is right. Therefore, the ori instruction works well in the CPU.

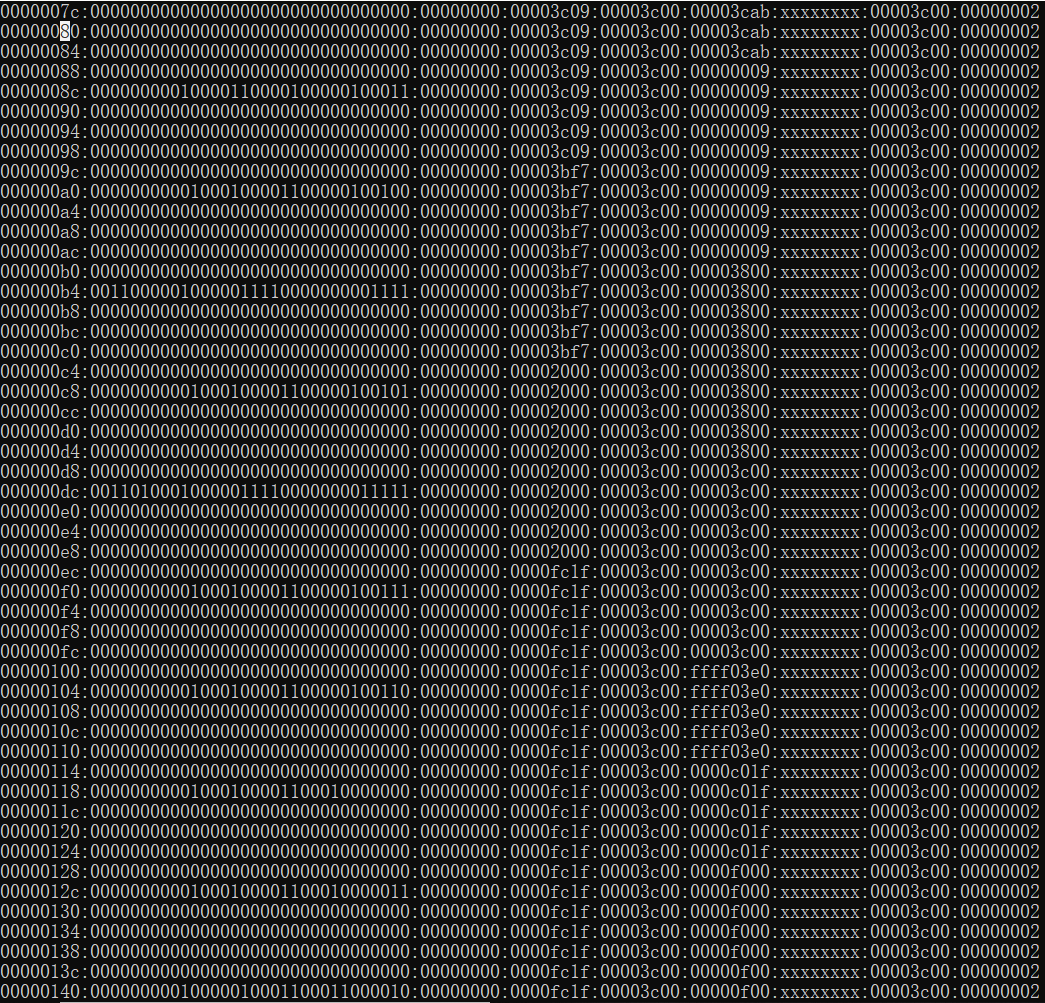
**nor:**

**Input:**



**Output:**

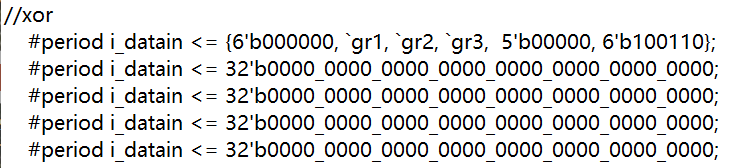


****

The function of this nor instruction is to make “nor” logic for every bit of the values in the gr1 and gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes ffff03e0 (fc1f nor 3c00=ffff03e0). It is obvious that the result is right. Therefore, the nor instruction works well in the CPU.

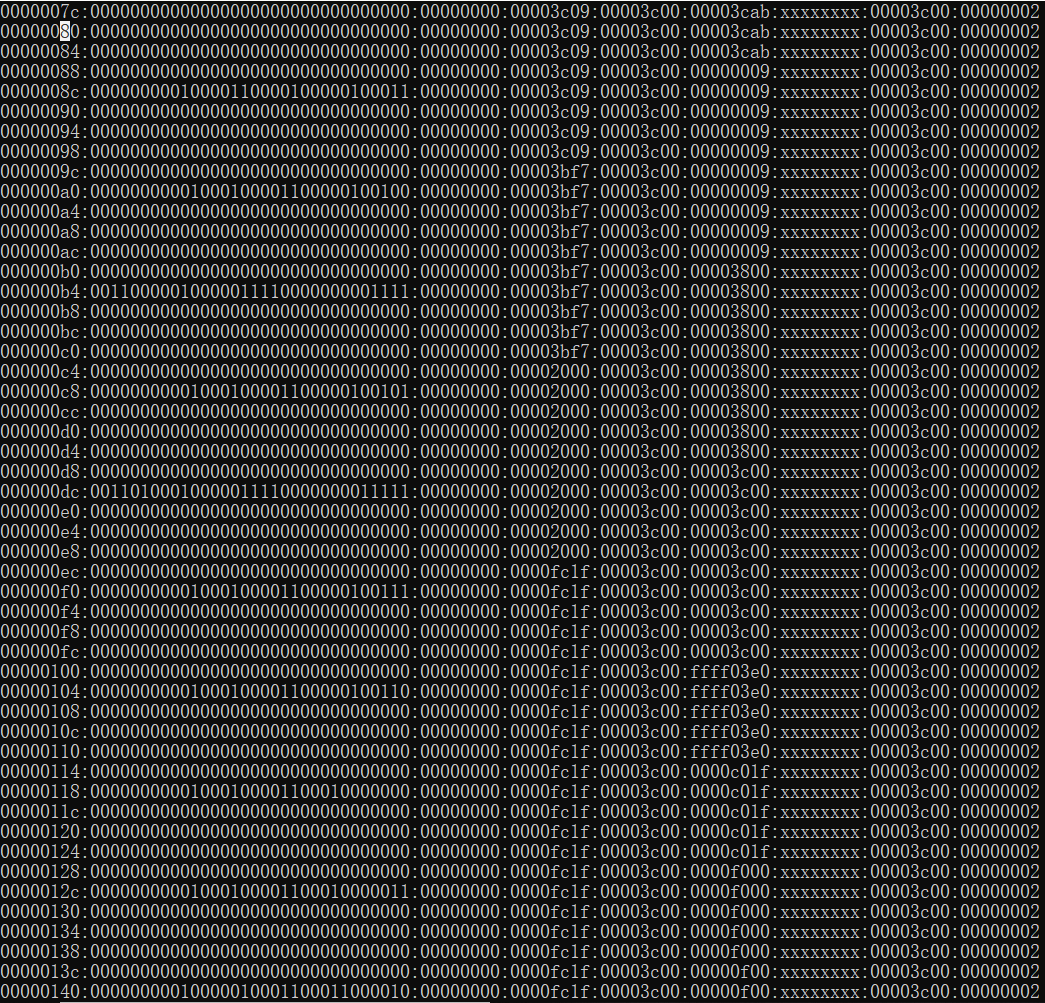
**xor:**

**Input:**



**Output:**

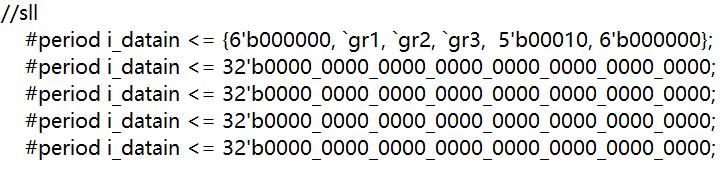


****

The function of this xor instruction is to make “xor” logic for every bit of the values in the gr1 and gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes c01f (fc1f xor 3c00=c01f). It is obvious that the result is right. Therefore, the xor instruction works well in the CPU.

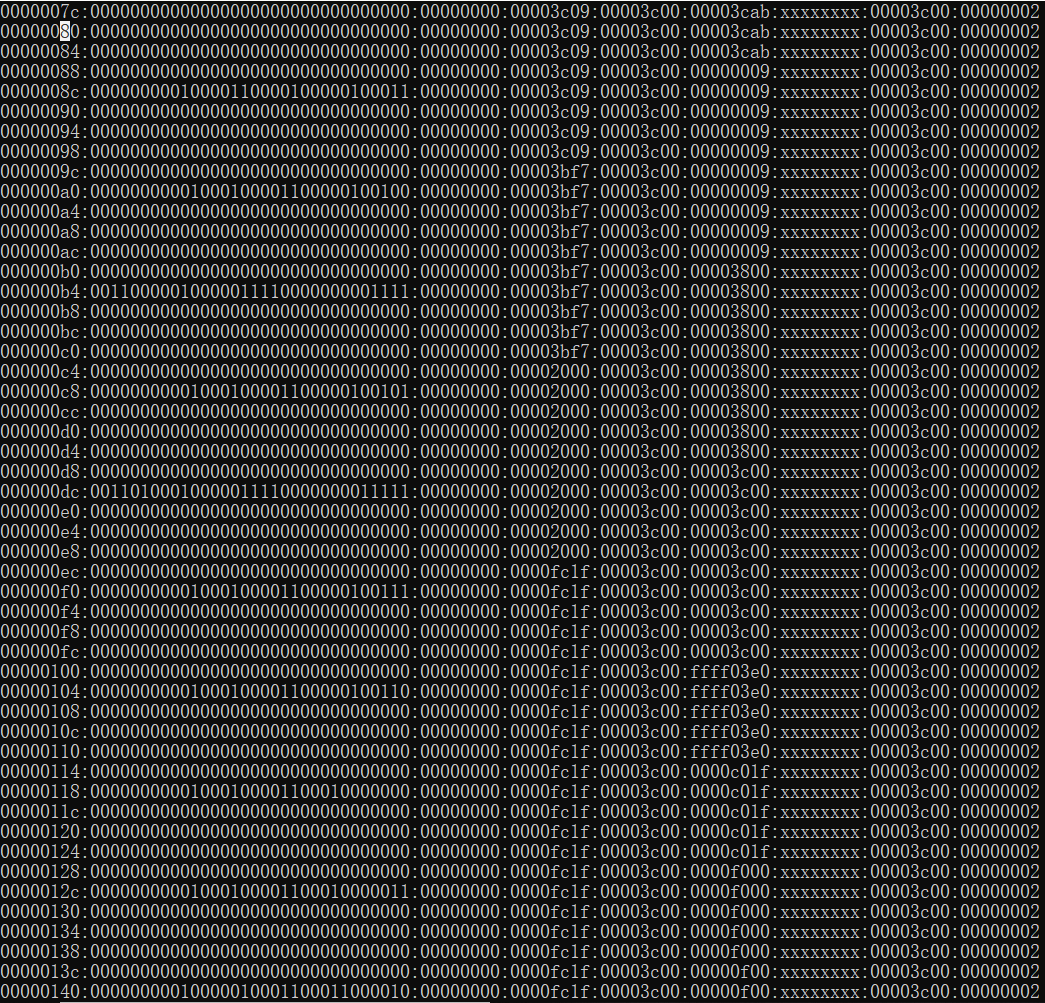
**sll:**

**Input:**



**Output:**

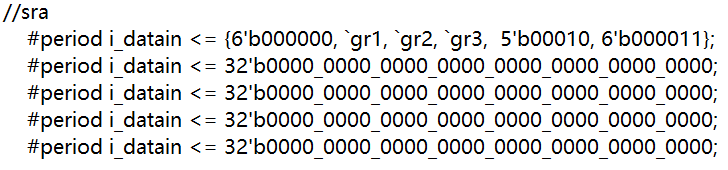


****

The function of this sll instruction is to logic left shift 2 bit of the value in gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes f000 (3c00<<2=f000). It is obvious that the result is right. Therefore, the sll instruction works well in the CPU.

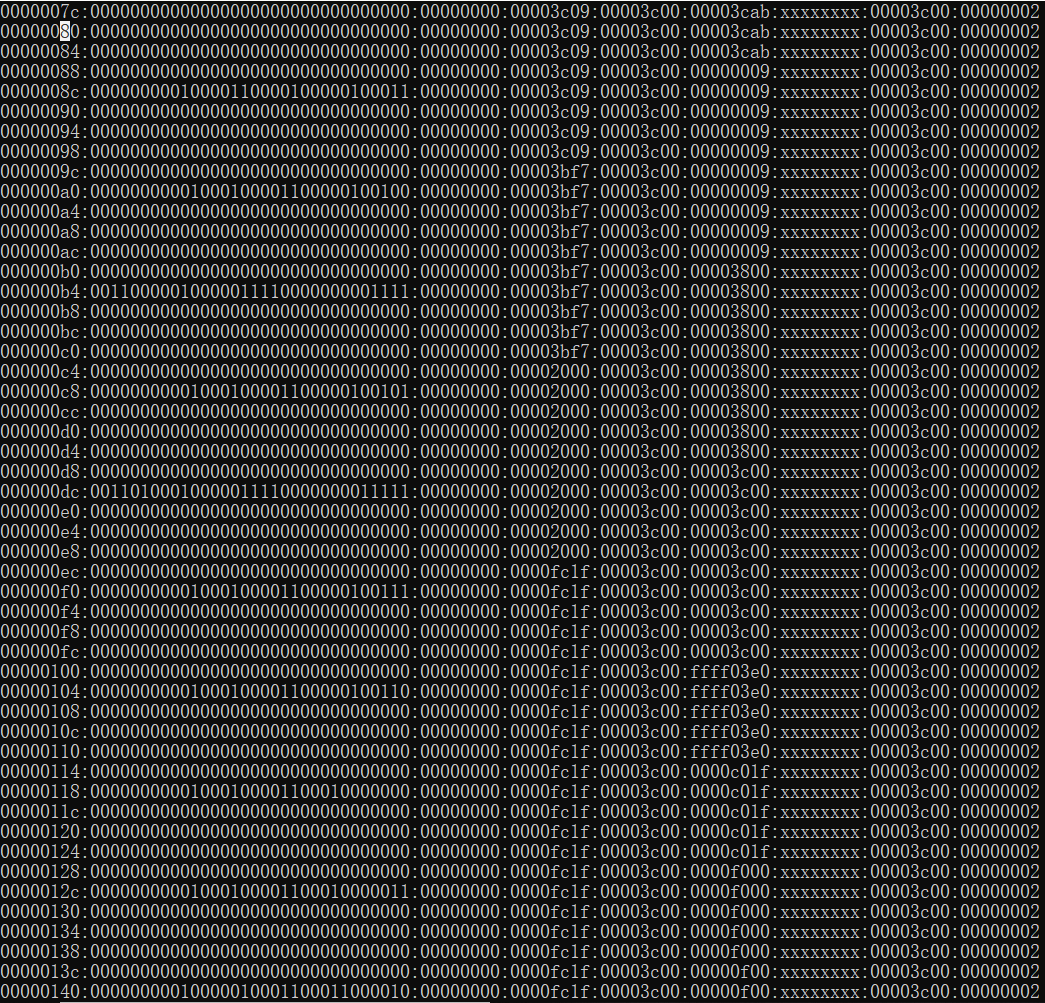
**sra:**

**Input:**



**Output:**

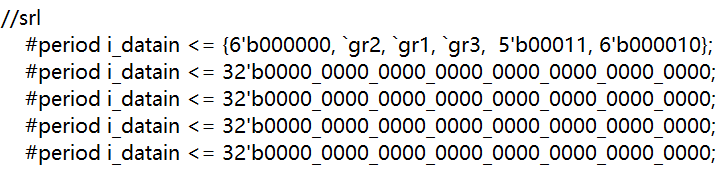


****

The function of this sra instruction is to arithmetic right shift 3 bit of the value in gr2 and put the result in gr3. After 5 clock, the value in gr3 becomes f00 (3c00>>3=f00). It is obvious that the result is right. Therefore, the sra instruction works well in the CPU.

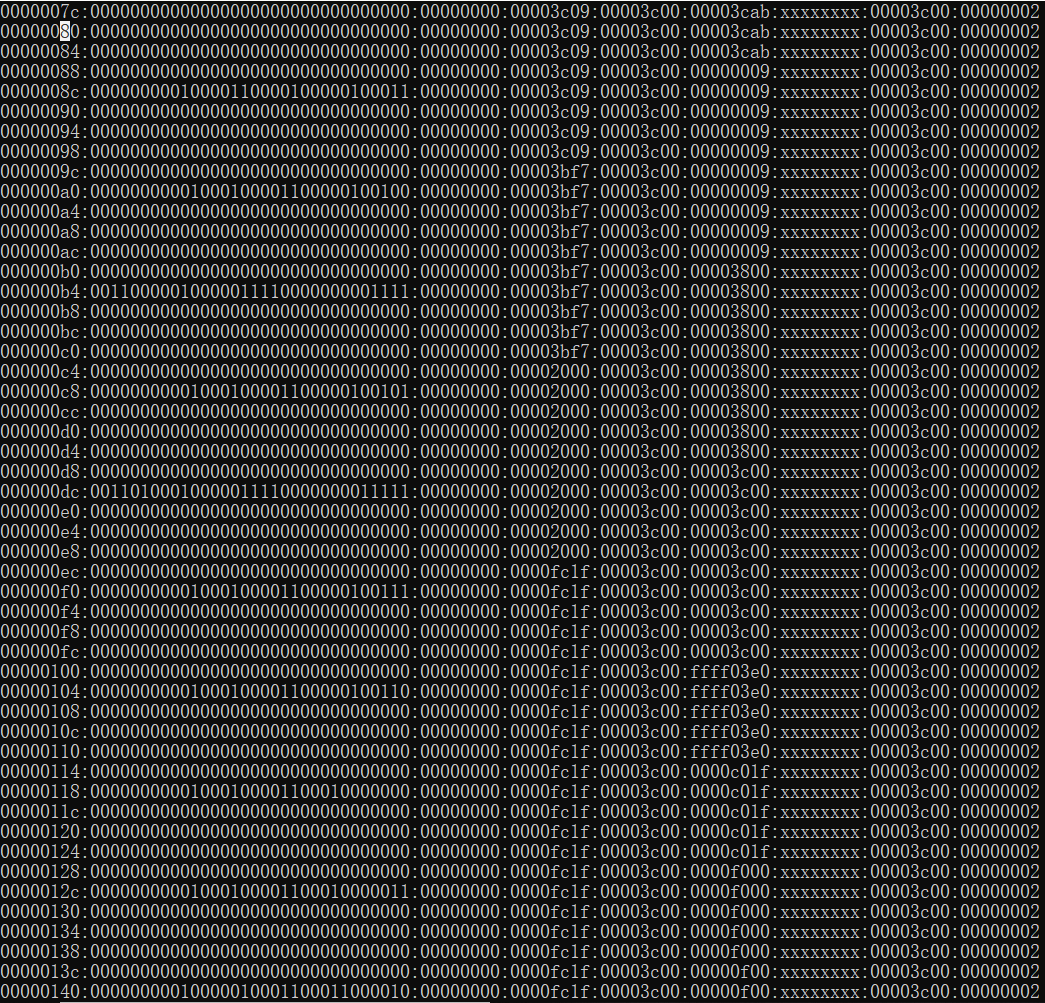
**srl:**

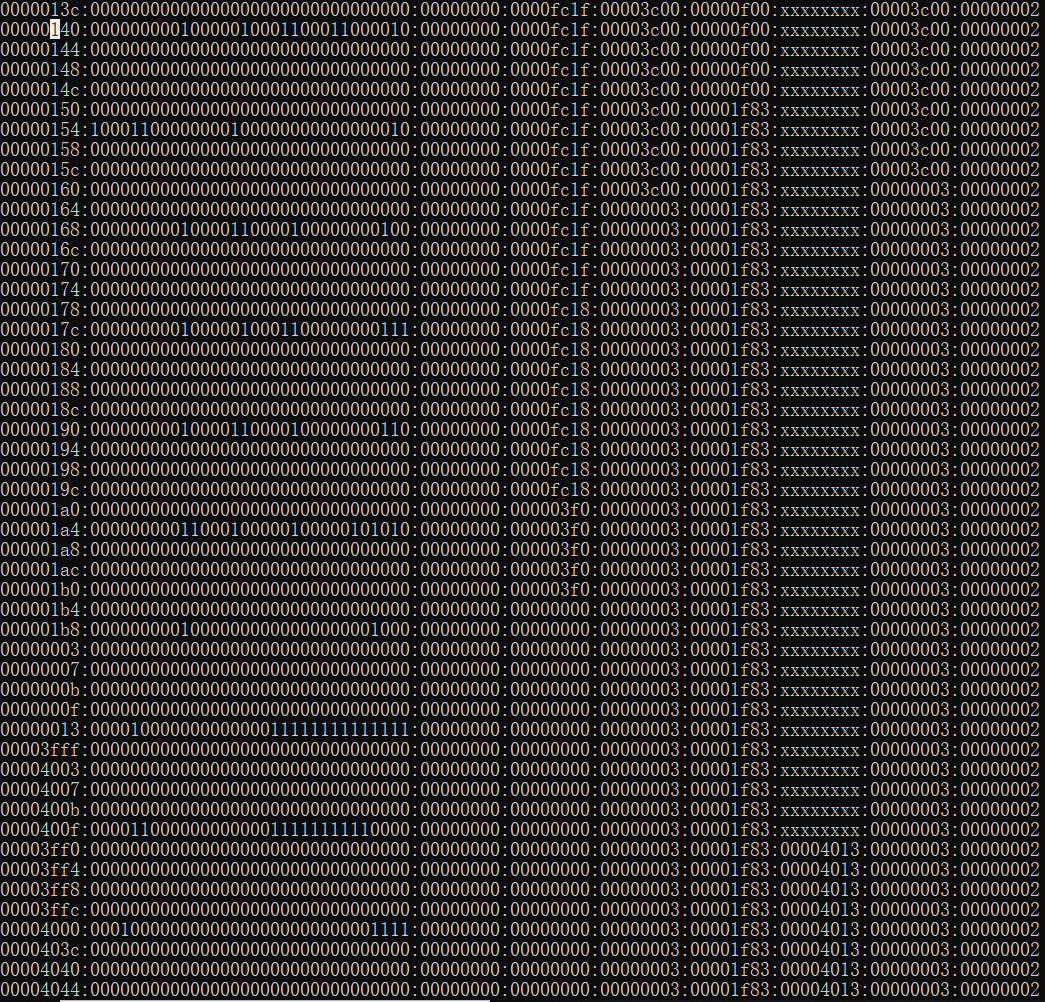
**Input:**



**Output:**



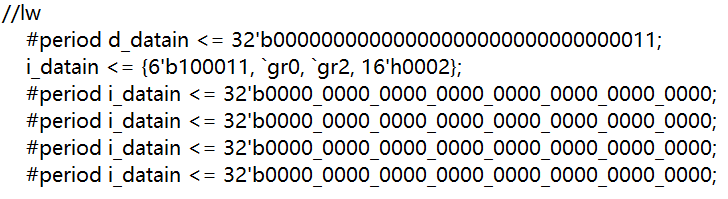
****

****

The function of this srl instruction is to logic right shift 2 bit of the value in gr1 and put the result in gr3. After 5 clock, the value in gr3 becomes 1f83 (fc1f>>2=1f83). It is obvious that the result is right. Therefore, the srl instruction works well in the CPU.

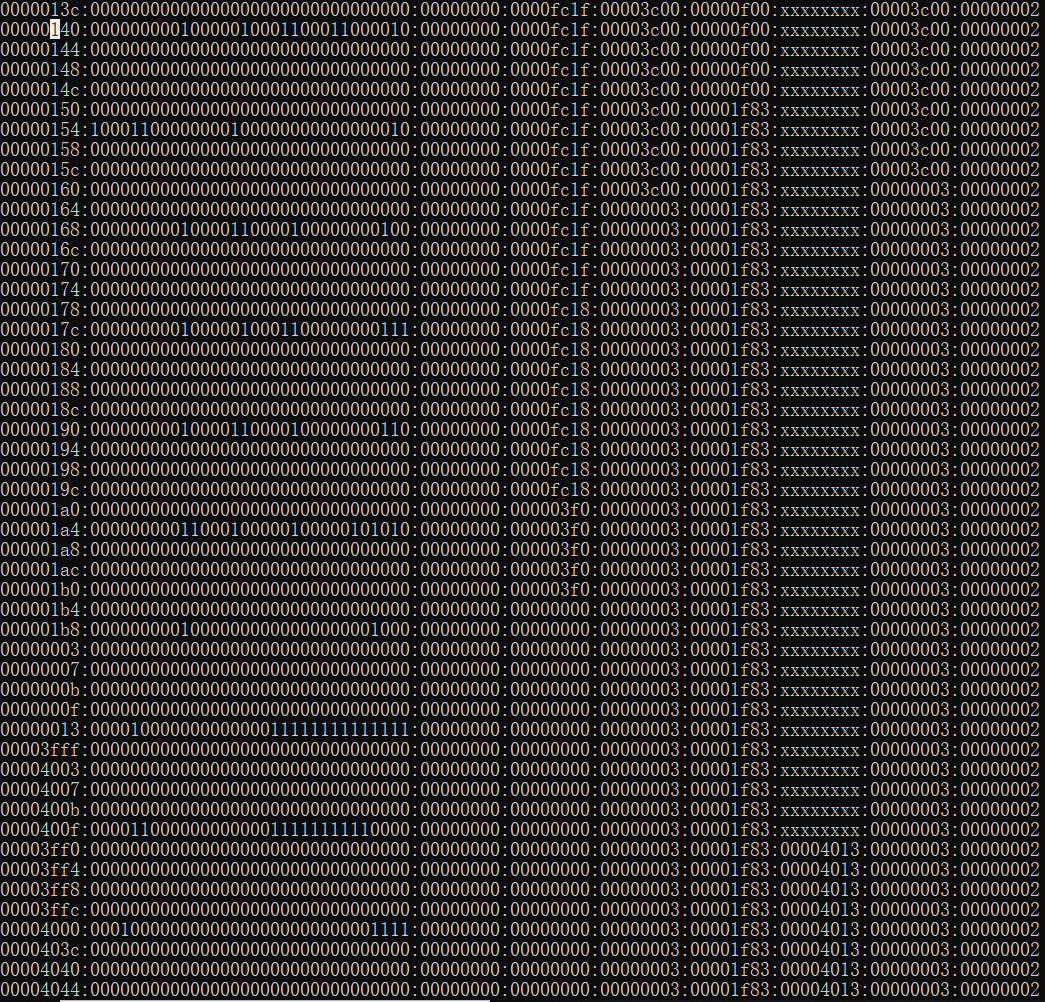
**lw:**

**Input:**



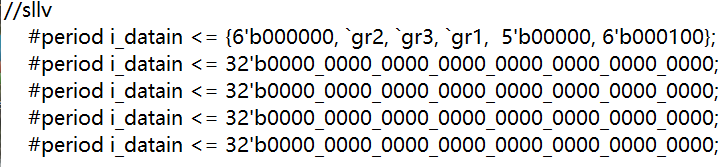
**Output:**



****

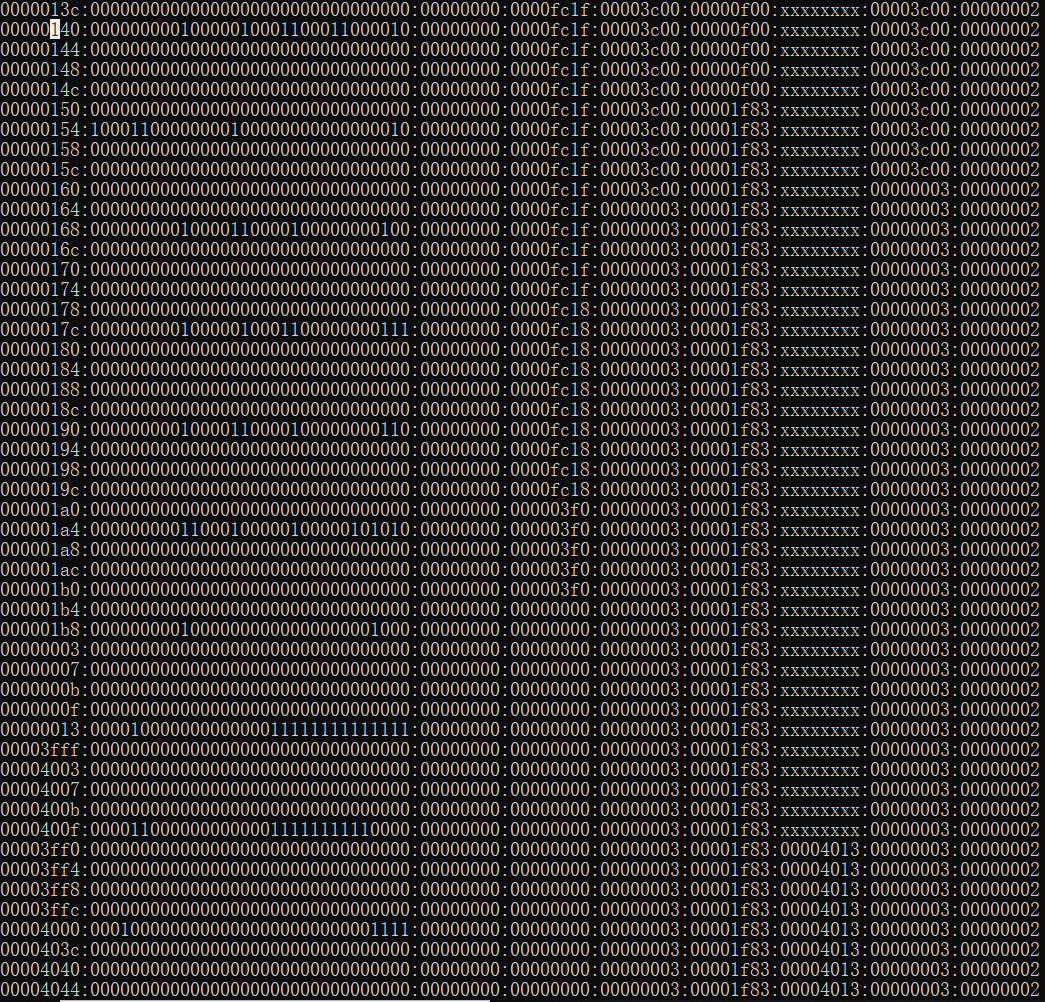
**sllv:**

**Input:**



**Output:**

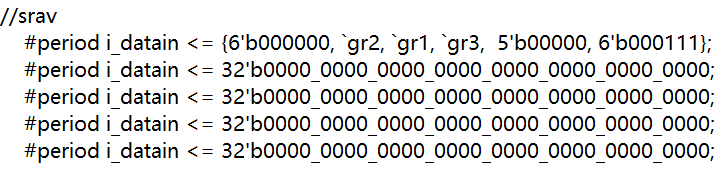


****

The function of this sllv instruction is to logic left shif value in gr2 of the value in gr3 and put the result in gr1. After 5 clock, the value in gr1 becomes fc18 (1f83<<3==fc18). It is obvious that the result is right. Therefore, the sllv instruction works well in the CPU.

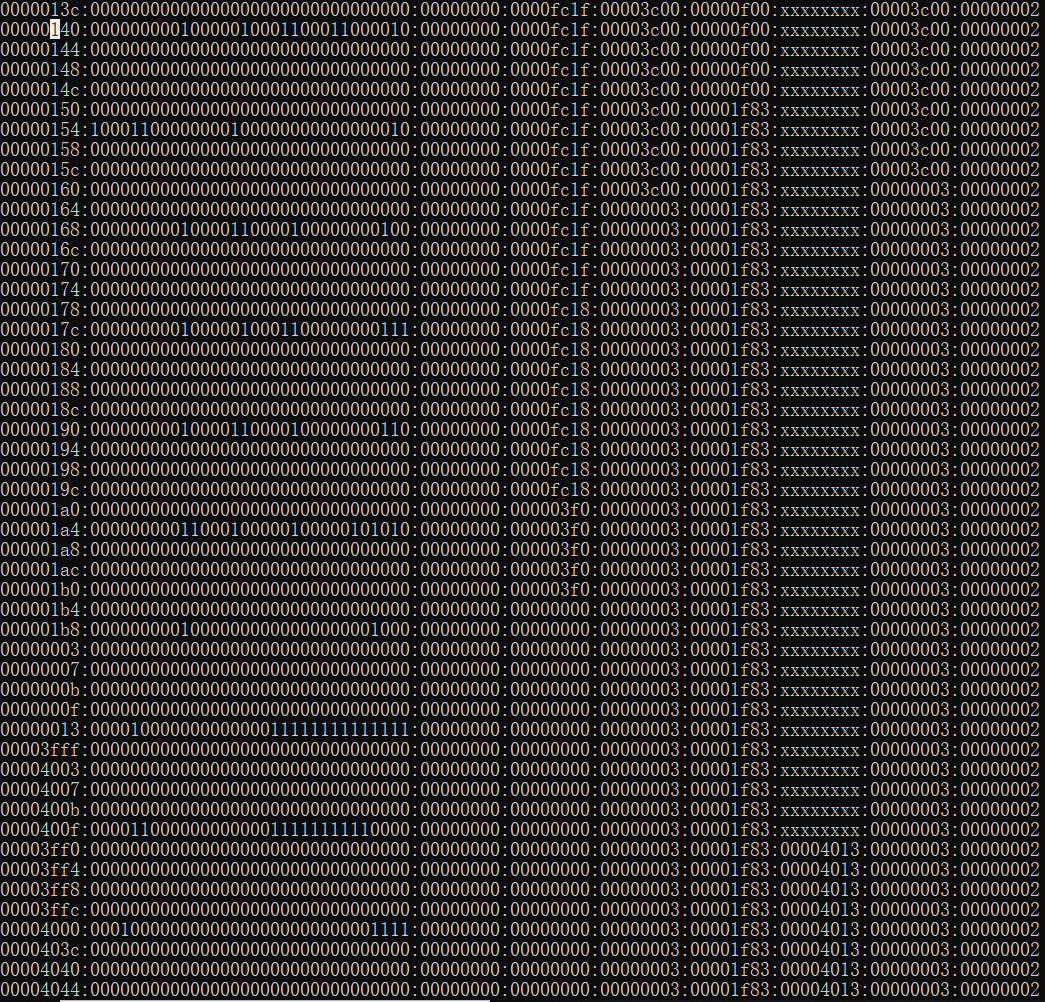
**srav:**

**Input:**



**Output:**

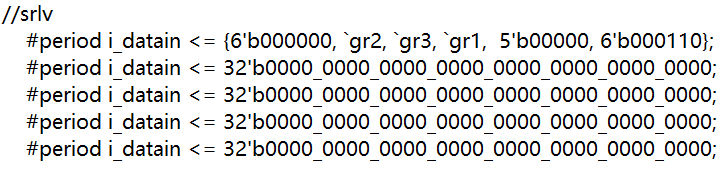


****

The function of this srav instruction is to arithmetic right shift value in gr2 of the value in gr1 and put the result in gr3. After 5 clock, the value in gr3 becomes 1f83 (fc18>>3==1f83). It is obvious that the result is right. Therefore, the srav instruction works well in the CPU.

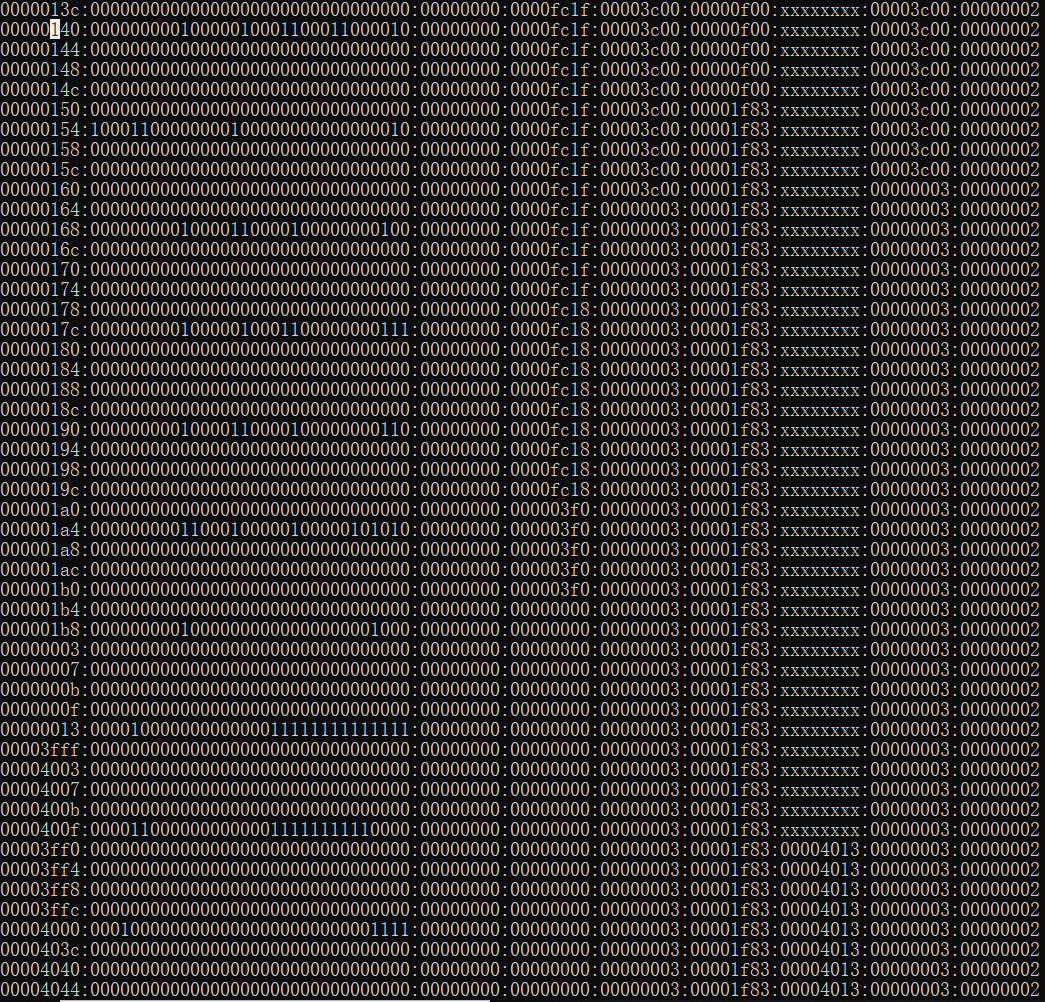
**srlv:**

**Input:**



**Output:**

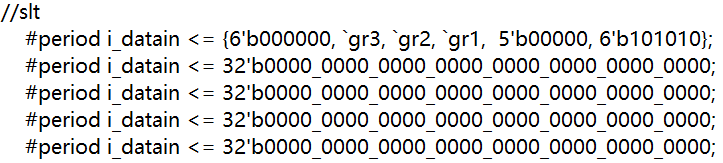


****

The function of this srlv instruction is to logic right shif value in gr2 of the value in gr3 and put the result in gr1. After 5 clock, the value in gr1 becomes 3f0 (1f83>>3==3f0). It is obvious that the result is right. Therefore, the srlv instruction works well in the CPU.

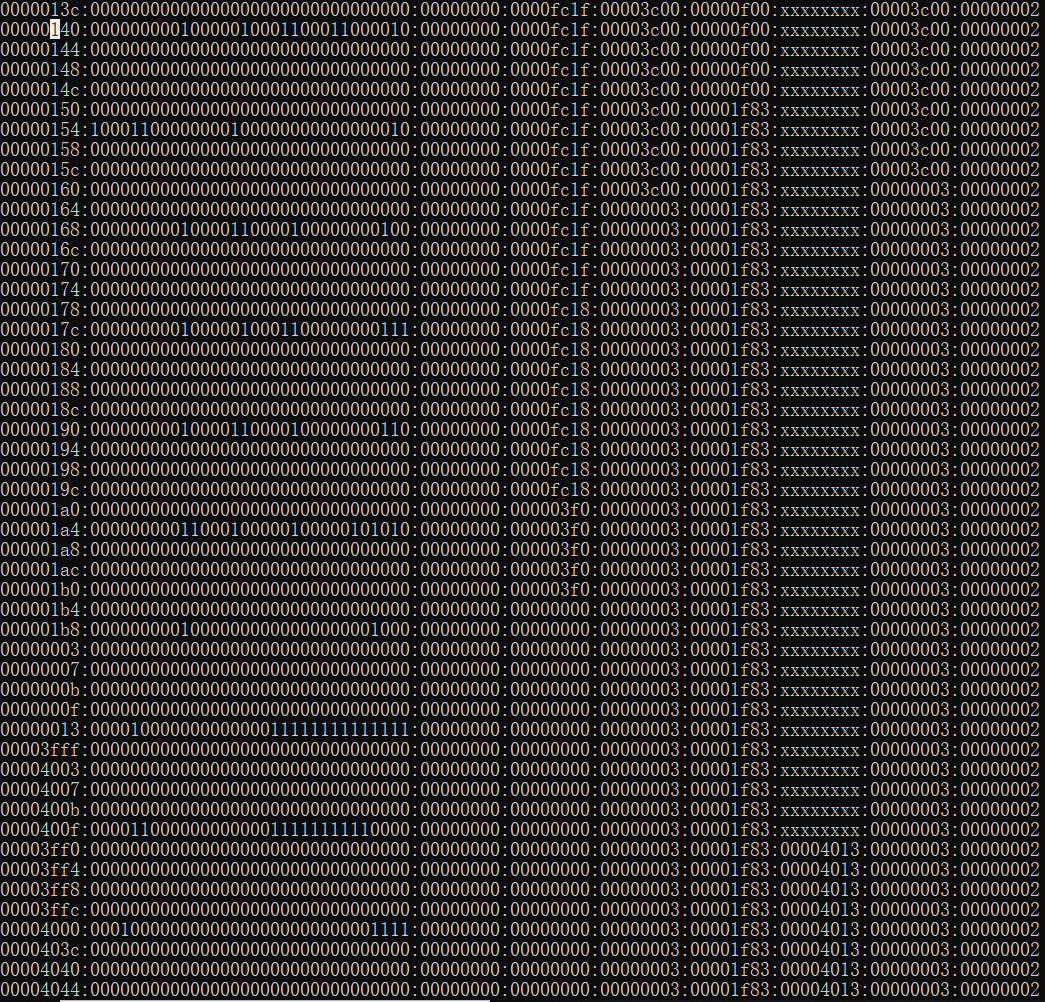
**slt:**

**Input:**



**Output:**

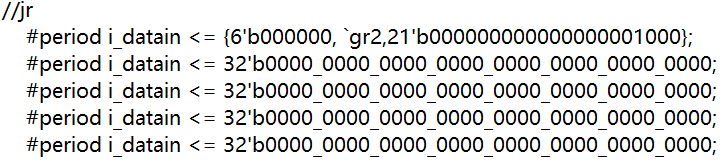


****

The function of this slt instruction is to judge whether the value in gr3 is less than the value in gr2 or not. If it is yes, the value in gr1 will change to 1. If it is no, the value in gr1 will change to 0. In this case, the value in gr3 is not less than the value in gr2 (1f83>3). After 5 clock, the value in gr1 changes to 0. It is obvious that the result is right. Therefore, the slt instruction works well in the CPU.

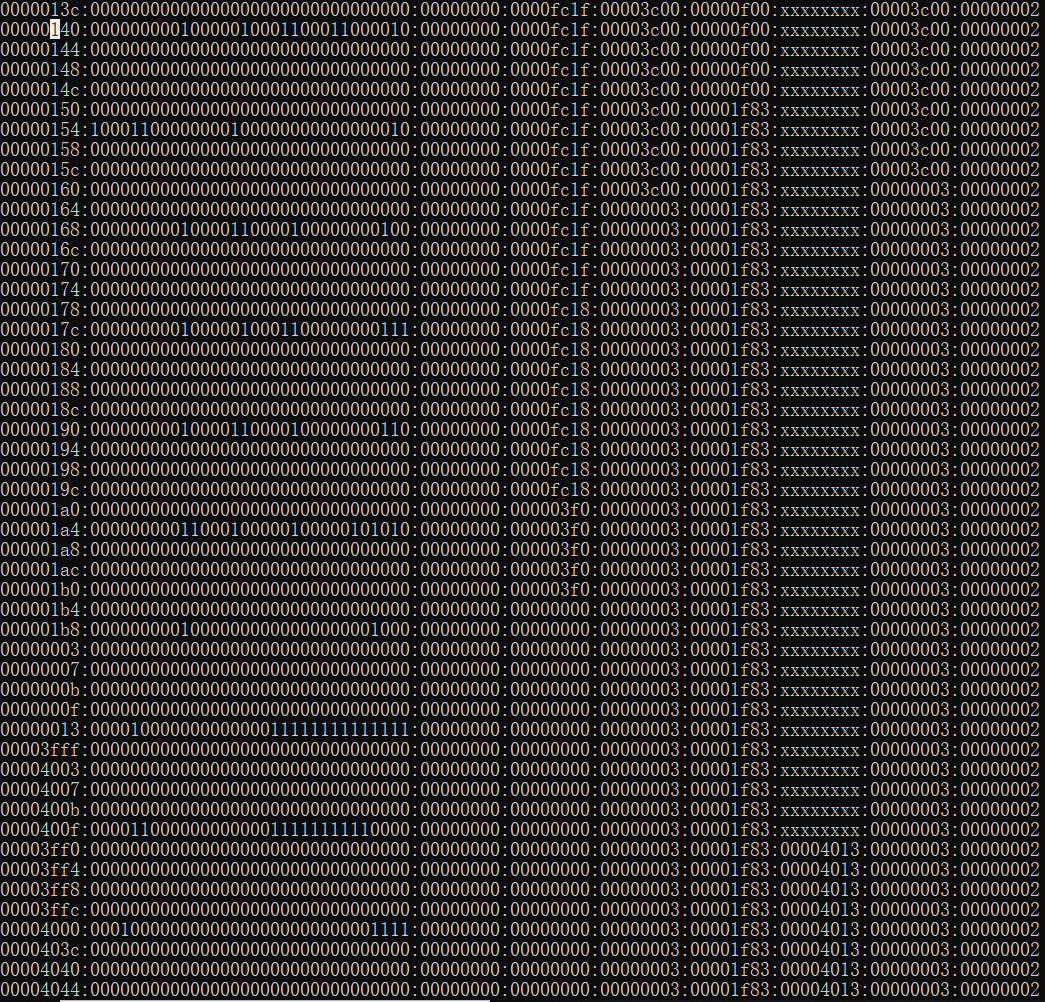
**jr:**

**Input:**



**Output:**

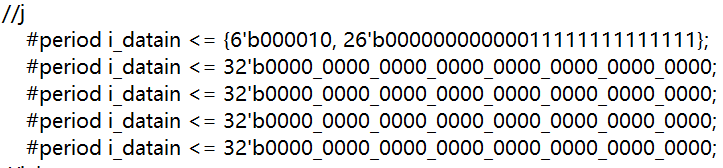


****

The function of this jr instruction is to change the pc to the address stored in gr2. The pc of the first NOP instruction becomes 3 which is exactly the value in gr2. It is obvious that the result is right. Therefore, the jr instruction works well in the CPU.

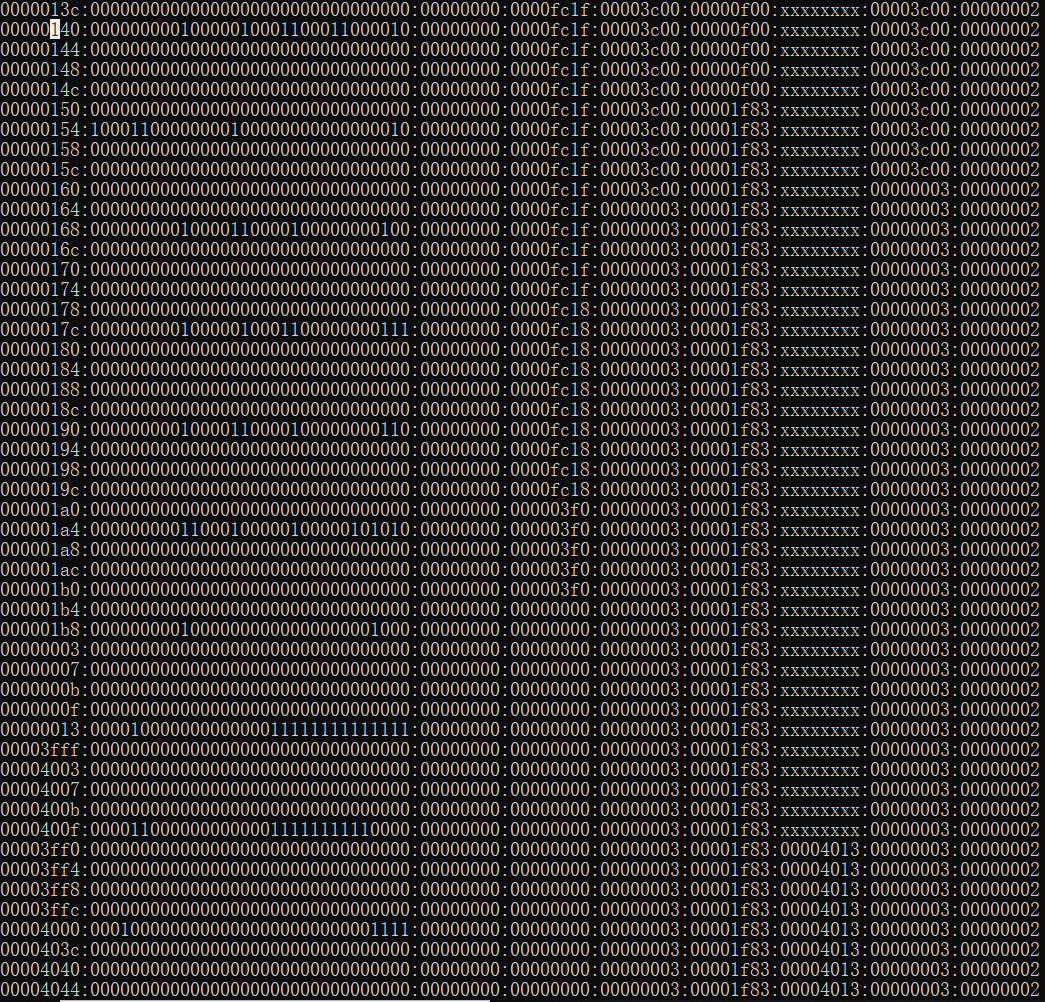
**j:**

**Input:**



**Output:**

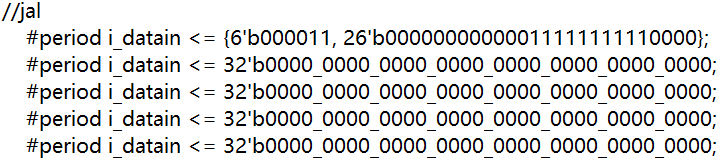


****

The function of this j instruction is to change the pc to the target part of the instruction. The pc of the first NOP instruction becomes 3fff which is exactly the value of the target. It is obvious that the result is right. Therefore, the j instruction works well in the CPU.

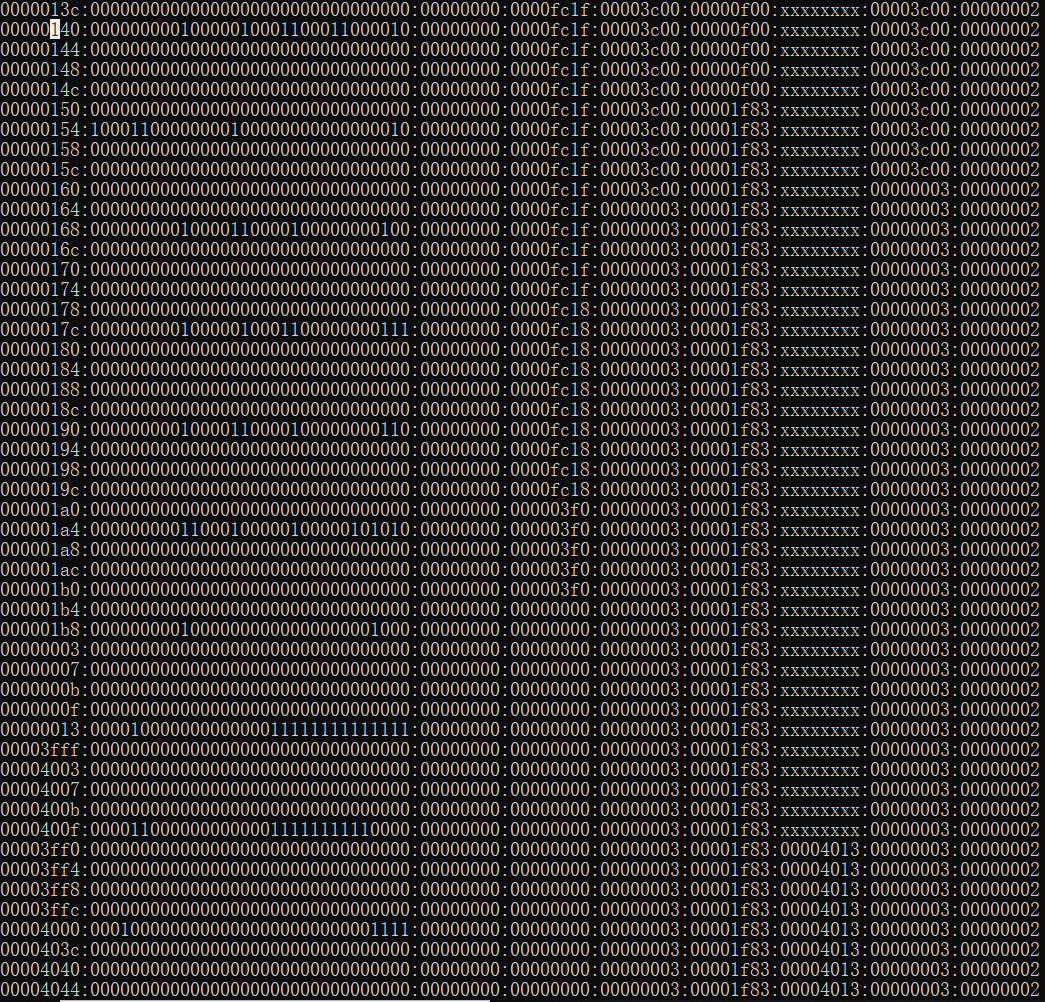
**jal:**

**Input:**



**Output:**

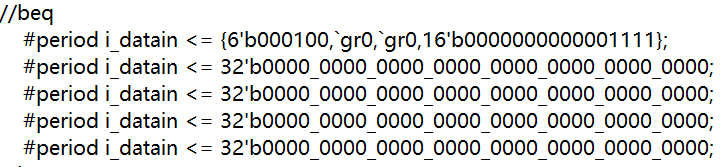


****

The function of this jal instruction is to change the pc to the target part of the instruction and store the original next address in the gr31 (ra) register. The pc of the first NOP instruction becomes 3ff0 which is exactly the value of the target and after 5 clock, the value in gr31 becomes 4013 which is exactly the original next address (400f+4=4013). It is obvious that the result is right. Therefore, the jal instruction works well in the CPU.

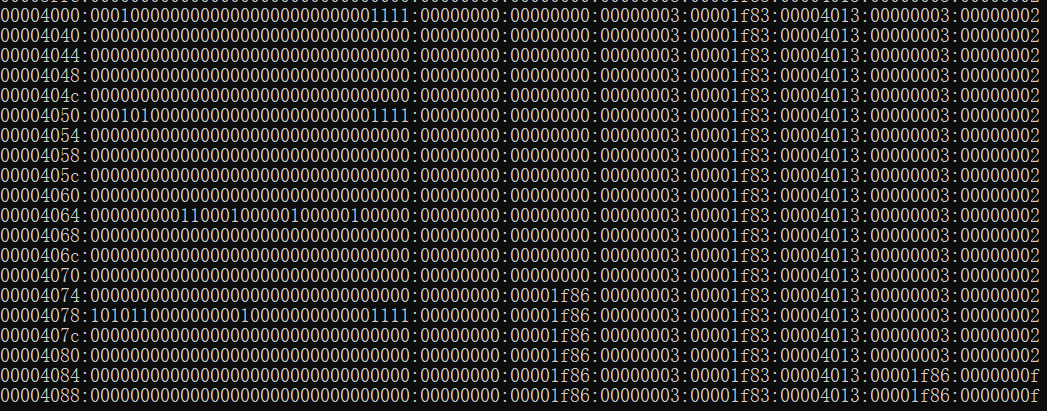
**beq:**

**Input:**



**Output:**

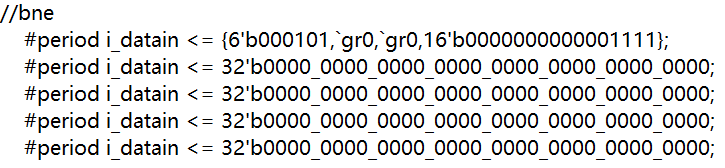


****

The function of this beq instruction is to judge whether the values in gr0 and gr0 are equal or not. If they are equal, the pc will change to . In this case, the values in gr0 and gr0 are equal. The pc of the first NOP instruction becomes 4040 . It is obvious that the result is right. Therefore, the beq instruction works well in the CPU.

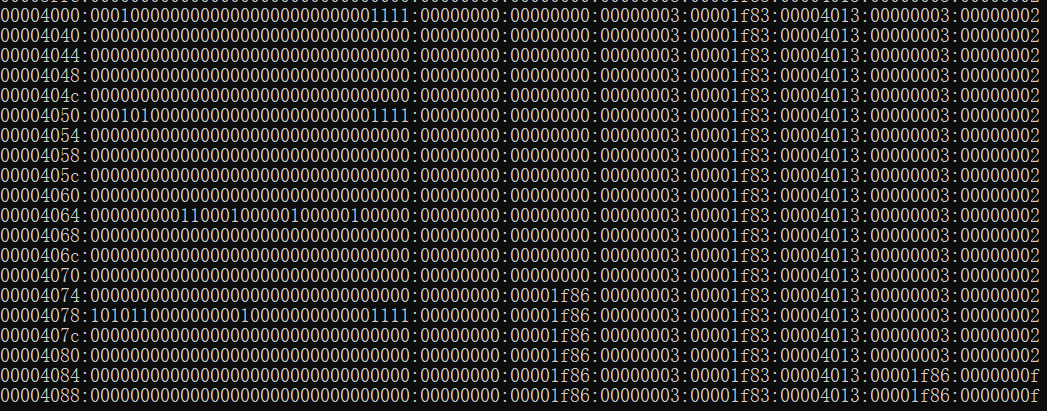
**bne:**

**Input:**



**Output:**

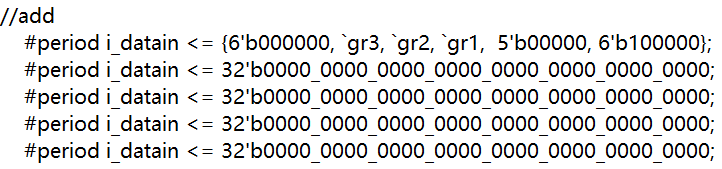


****

The function of this bne instruction is to judge whether the values in gr0 and gr0 are equal or not. If they are not equal, the pc will change to . In this case, the values in gr0 and gr0 are equal. Nothing will happen. The pc of the first NOP instruction becomes 4054 (4050+4=4054). It is obvious that the result is right. Therefore, the bne instruction works well in the CPU.

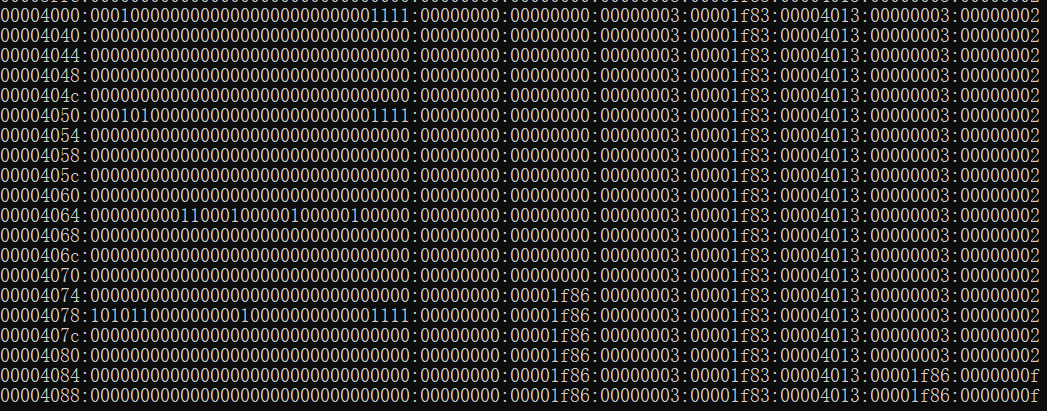
**add:**

**Input:**

****

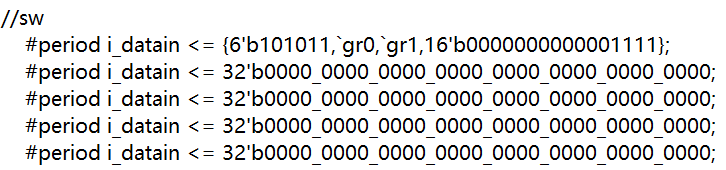
**Output:**



****

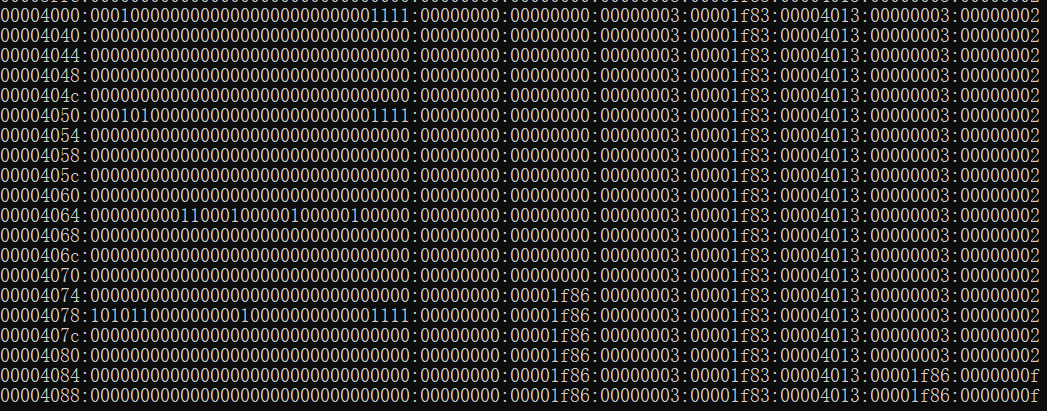
**sw:**

**Input:**



**Output:**



****

The function of this sw instruction is to get the value stored in gr1 and get the address from gr0 + offset. After 5 clock, the d\_dataout is 1f86 which is equal to the value in gr1 and the d\_addr is f which is equal to gr0 + offset (0+f=f). It is obvious that the result is right. Therefore, the sw instruction works well in the CPU.

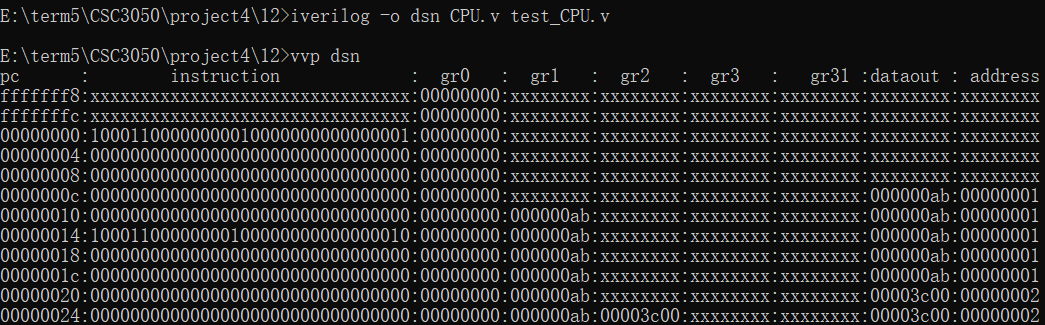
**4 explanation of the change of pc:**

In my program, my pc is the pcf, which means the pc is exact the address of the corresponding instruction but not the next instruction. For most of instructions required in this project, when the next instruction comes, we just need to make the pc plus 4. For jump instruction, we need to get the jump address and make the pc of the next instruction of the jump instruction be the jump address. For the branch instruction beq and bne, if they meet their branch condition, we just calculate the branch address (pc + offset\*4+4) and then make the pc of the next instruction of the branch instruction be the branch address.

**5 explanation of the output (d\_addr and d\_dataout):**

The output is only useful for lw and sw. For lw, the d\_dataout is the same as d\_datain and the d\_addr is equal to the value in rs plus offset. For sw, the d\_dataout is the same as the value in rt and the d\_addr is equal to the value in rs plus offset. For both lw and sw, the right d\_addr and d\_dataout will be shown after 5 clock (when the instruction executed completely). For other instructions, the output is not very important because it will not be used.

**6 how to run my code (in windows):**



**7 summary:**

I spent almost a whole week to write this project. I really try my best to finish this project. I have learnt a lot about Verilog and CPU from this project. With the help of TA and many students, I have overcome a lot of problems. I am grateful to them.