**ELEC 5200 CPU DESIGN REPORT**

**PART 5**

**Travis Keller**

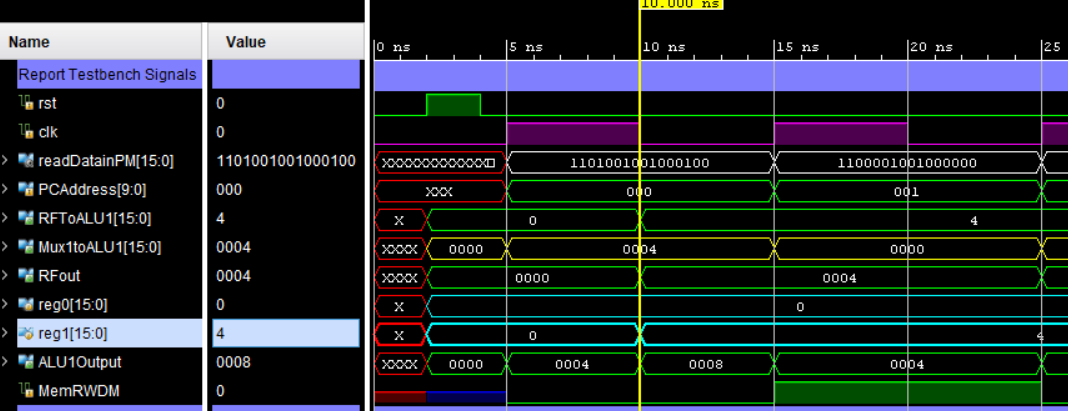
**Assembly Program**

|  |
| --- |
| **1101001001000100; //plusi $r1,$r1,000100 add 4 to whatever is in R1  1100001001000000; //sw r1,000000($r1) store whatever is in R1 into memory at the address in 000000($R1)  1101001000000010; //plusi r0,r1,2 add 2 to whatever value is in register 1 and store it in r0 1011000001000000; //lw r0,000000($r1) load whatever value of whatever address is in R0 into R1 0100001001000000; //plus $r1,$r1,$r0 add whatever is in $r1 to itself and store it in $r0 8 + 8 = 16  0101000001000000; //min $r0,$r1,$r0 subtract that back out 16 - 8 = 8  0110000001000000; //and $r0,$r1,$r0 and whatever is in r1 and r0 and store it in r0 8 & 8 = 8  0111000001000000; //or $r0,$r1,$r0 or whatever is in r1 and r0 and store it in r0 8 | 8 = 8  0000000001000010; //fkeq $r0,$r1,$r0 fork to PC + 1 + 2 if $r0 and $r1 equal. fork to 11 bc 8 and 8 equal  0001000001000010; //fkne $r0,$r1,$r0 fork to PC + 1 + 2 if $r0 and $r1 equal. don't fork bc 8 and 8 equal  0010000001000010; //fklt $r0,$r1,$r0 fork to PC + 1 + 2 if $r0 < $r1. don't fork bc 8 and 8 equal  0011000001000010; //fkle $r0,$r1,$r0 fork to PC + 1 + 2 if $r0 <= $r1. fork to 16 bc 8 and 8 equal  1000000001000010; //fkgt $r0,$r1,$r0 fork to PC + 1 + 2 if $r0 > $r1. don't fork bc 8 and 8 equal  1001000001000010; //fkge $r0,$r1,$r0 fork to PC + 1 + 2 if $r0 >= $r1. fork to 20 bc 8 and 8 equal  1101001001000001; //plusi $r1,$r0,$000001 add 1 to $r1 and store in $r1  0000000001000010; //fkeq $r0,$r1,$r0 fork to PC + 1 + 2 if $r0 and $r1 equal. don't fork first time. go to 22. second time fork to 24  1101001001111111; //plusi $r1,$r1,111111 subtract 1 from $r1 to make it equal to $r2 again.  1010000000010101; //jump 0000010101 jump back two instructions to 21. Now $r0 and $r1 are equal again so instruction 21 will fork to storage[24] where the HALT instruction is  1110000000000000; //HALT the program** |

**SIMULATION RESULTS AND EXPLANATION**

**1101001001000100; //plusi $r1,$r1,0010 add 4 to whatever is in $r1 and store it in $r1**

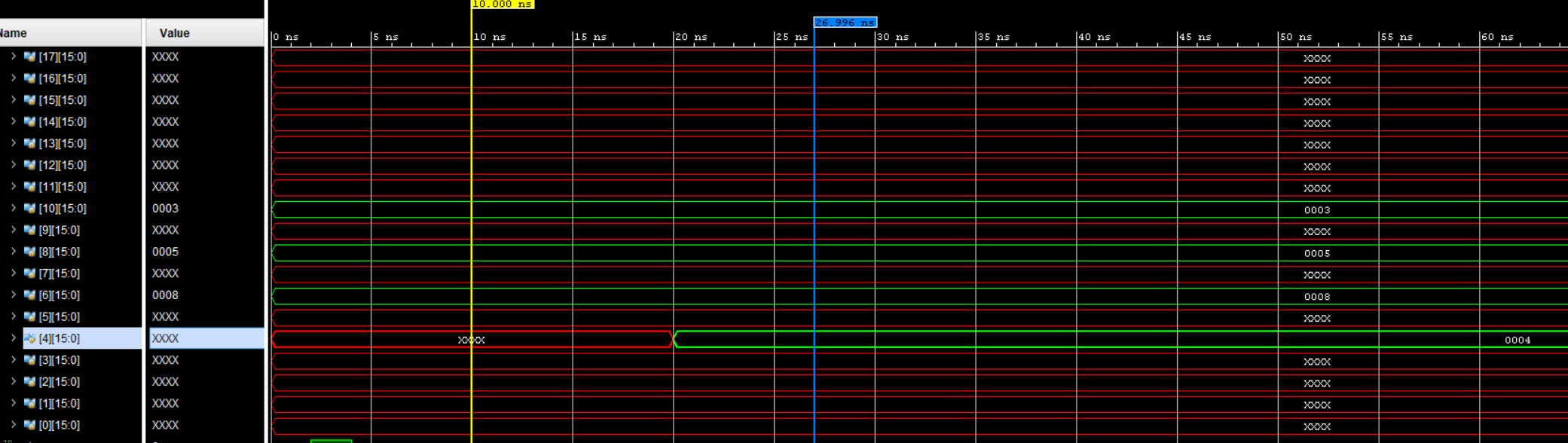
All registers are initialized to zero after ‘rst’ is activated at clk = 2; Then 4 is added to 0 and the result 4 is stored in $r1. Then the PC is incremented 1. In my design I write registers on the negative edge of the clock. The PC is written on the positive edge of the clock. It can be seen in the figure below that the value in register 1 becomes 4 on the negative edge of the 1st clock cycle (see far right side of figure). It can also be seen that the inputs of ALU1 are 0 and 4 and the ALU1 Output is 4 which matches the expectation for a plusi instructiona adding a constant of 4 to a register value of 0. Also readDatainPM is the instruction in binary form.



**1101001001000100; //plusi $r1,$r1,0010**

**1100001001000000; //sw r1,000000($r1) store whatever is in R1 into memory at the address in 000000($R1)**

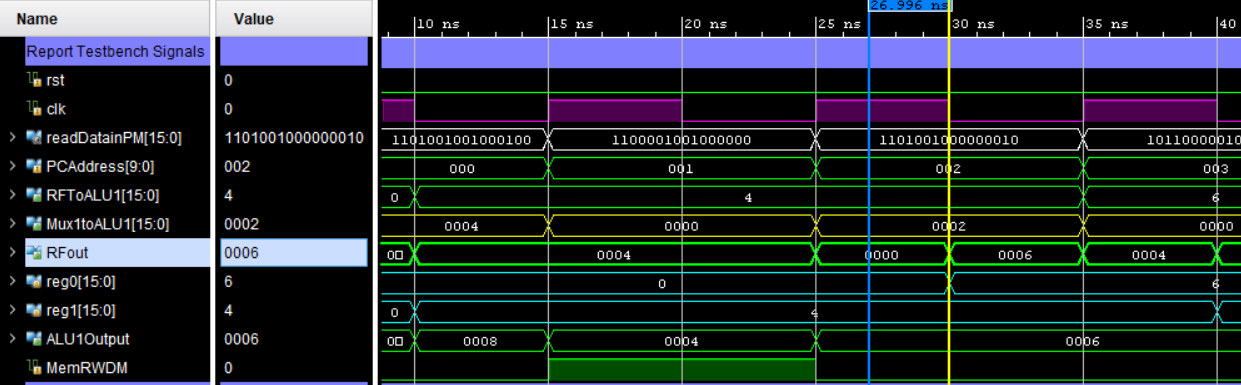
This instruction stores the value in r1 (4) at the data memory address in r1 (also 4). The register values for this instruction can be seen in the previos figure for the last instruction. Below is a figure showing this value being written to the address 4 in data memory. Since there are 1024 register addresses, it was not possible to capture each of them in this screenshot. You can however see the time the value is written (negative edge of clock) at the top of the figure. This corresponds to the negative edge of the second clock cycle which is as expected. The previous figure also shows the data memory read write signal go high and then low again at the next instruction. High is write.



**1100001001000000 //sw r1,000000($r1)**

**1101001000000010; //plusi r0,r1,2 add 2 to whatever value is in register 1 and store it in r0**

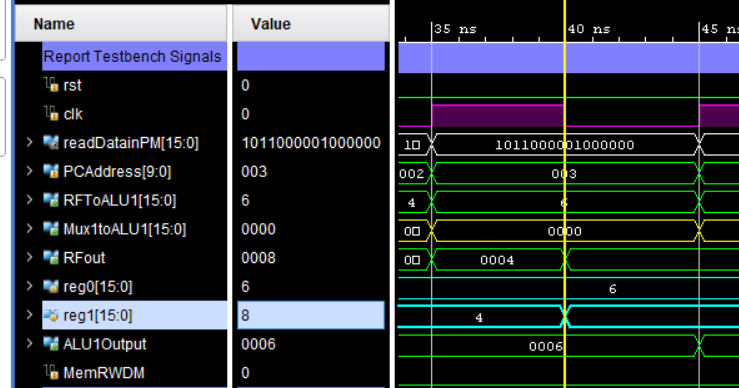
This is another plusi instruction. It adds 2 to R1 and stores it in R0 as can be seen in the figure below. R1 is 4 so R0 becomes 6.



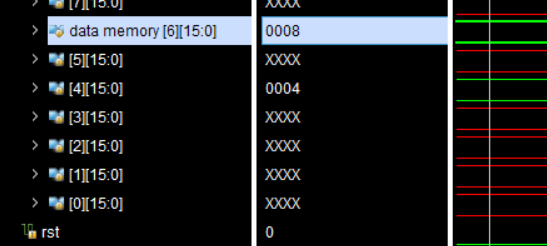
**1101001000000010 //plusi r0,r1,2**

**1011000001000000; //lw r0,000000($r1) load whatever value of whatever address is in R0 into R1**

This instruction will get the value of whatever is in the address located in register R0 and write it into the register R1. Since the address in R0 is 6, this will load from the data memory address 6, and grab the value from there. I have not yet written to address 6 (only address 4) but I preloaded address 6 with the value 8 using an initial block. It can be seen from the figure below that the value of 8 is loaded into register R1. The next figure shows the contents at address 6 in memory.



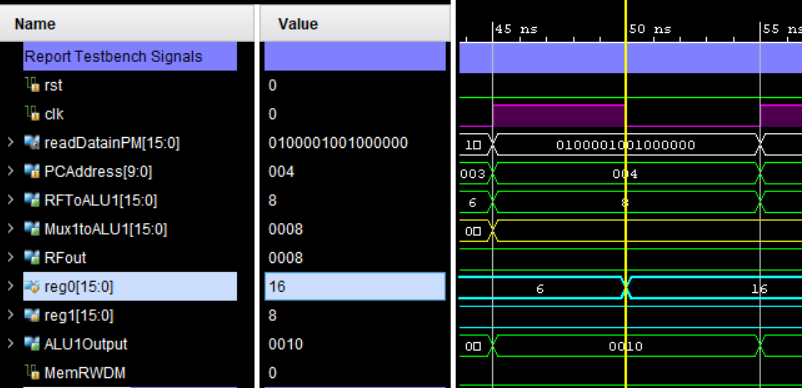
**1011000001000000; //lw r0,000000($r1)**

****

**1011000001000000; //lw r0,000000($r1)**

**0100001001000000; //plus $r1,$r1,$r0 add whatever is in $r1 to itself and store it in $r0 8 + 8 = 16**

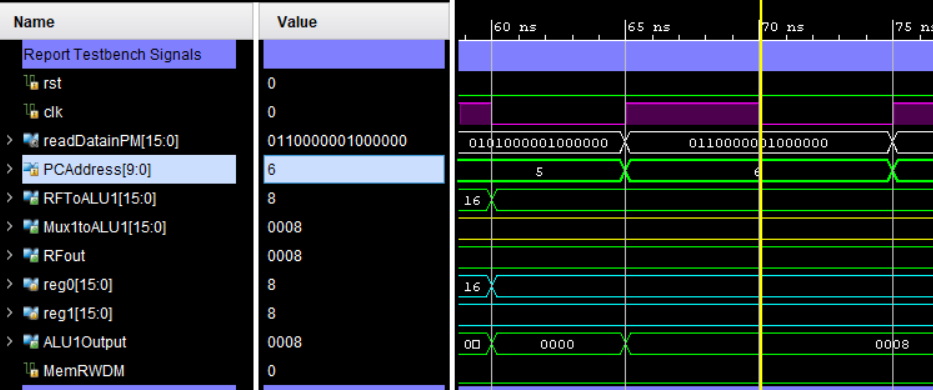
This is a simple addition ALU operation instruction. It adds r1 to itself and stores the value in r0. In this case, it adds 8 + 8 and stores the result in r0. The figure below shows the register file outputs, the ALU inputs and outputs, and the program counter values.



**0100001001000000; //plus $r1,$r1,$r0**

**0101000001000000; //min $r0,$r1,$r0 subtract that back out 16 - 8 = 8**

This is another simple arithmetic operation that substracts the value in r1 from the value in r0 and stores it in R0. Since I previously added r1 to itself and stored that in r0. I am now just subtracting that same value back out of r0. So r0 is now 8.



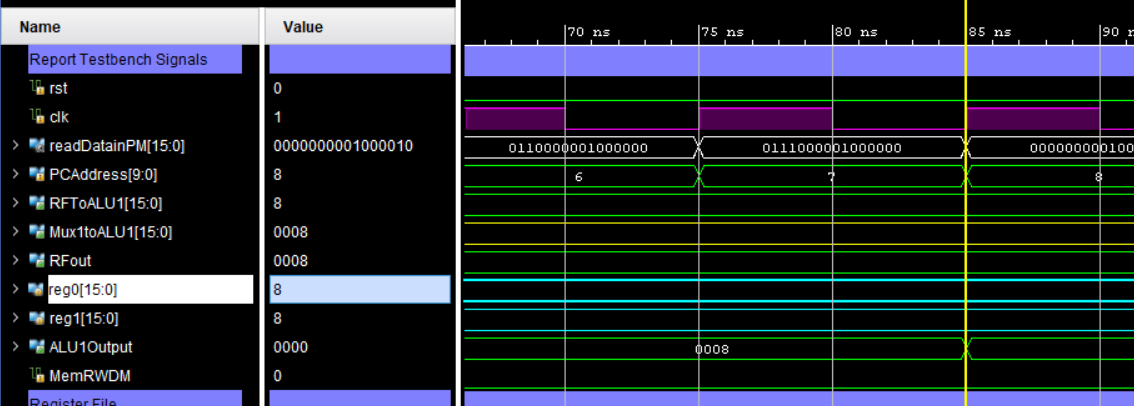
**0101000001000000; //min $r0,$r1,$r0**

**0110000001000000; //and $r0,$r1,$r0 and whatever is in r1 and r0 and store it in r0 8 & 8 = 8**

AND

**0111000001000000; //or $r0,$r1,$r0 or whatever is in r1 and r0 and store it in r0 8 | 8 = 8**

This instruction simply performs a bitwise AND instruction on the values in r1 and r0 and stores that in r0. Since the values in r1 and r0 are both 8, there is no change to the value of the registers. This is the same for the OR instruction also included in the below figure. The AND and OR instructions precede the vertical yellow line in the figure.



**0110000001000000; //and $r0,$r1,$r0**

**0111000001000000; //or $r0,$r1,$r0**

**0000000001000010; //fkeq $r0,$r1,$r0 fork to PC + 1 + 2 if $r0 and $r1 equal. fork to 11 bc 8 and 8 equal**

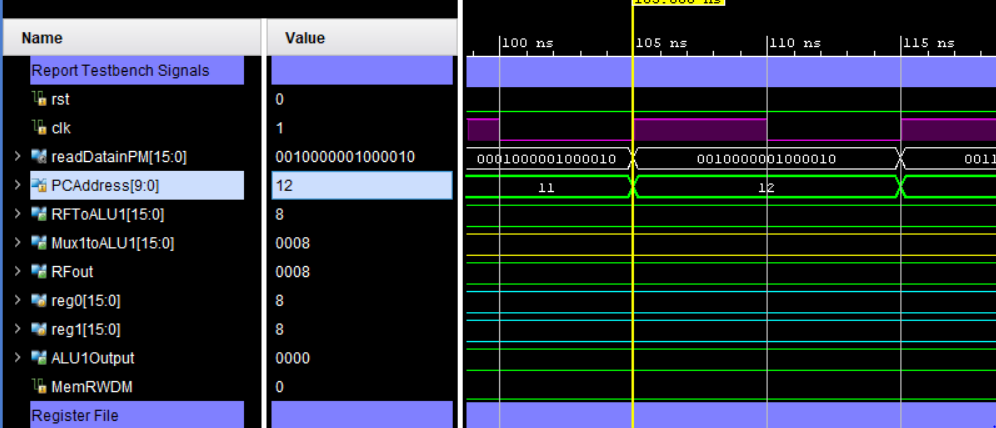
FKEQ will branch to the address in the last 6 digits of the instruction plus the PC + 1 if the values in the two registers (r0 and r1 in this case) are equal. In this case, the values in both registers are 8 and the branch address is 2, so the Program Counter Branches from its current address (8) to 3 addresses forward, which is 11. This can be seen on the PCAddress signal in the figure below. Once again the Program Counter address is written on the positive edge of the clock.



**0000000001000010; //fkeq $r0,$r1,$r0**

**0001000001000010; //fkne $r0,$r1,$r0 fork to PC + 1 + 2 if $r0 and $r1 equal. don't fork bc 8 and 8 equal**

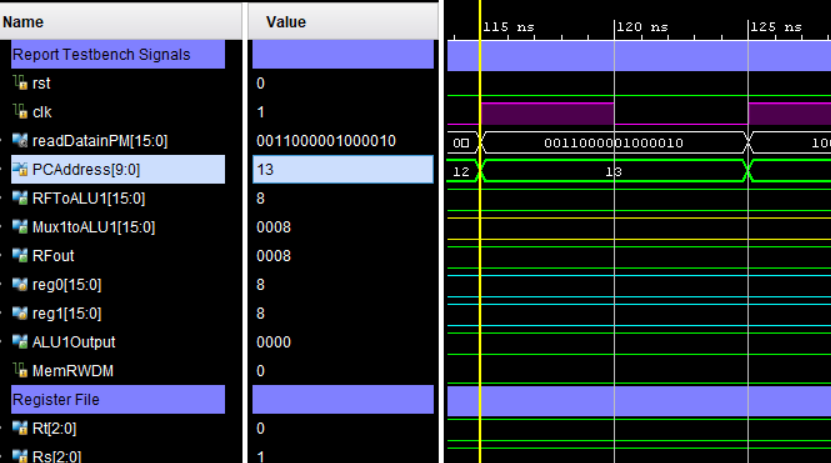
The FKNE instruction has the exact opposite behavior of the previous FKEQ instruction. The register values have not changed (they are still both 8), and thus the instruction will NOT cause a branch. So the Program Counter will only advance by one address as usual. This is seen in the figure below.



**0001000001000010; //fkne $r0,$r1,$r0**

**0010000001000010; //fklt $r0,$r1,$r0 fork to PC + 1 + 2 if $r0 < $r1. don't fork bc 8 and 8 equal**

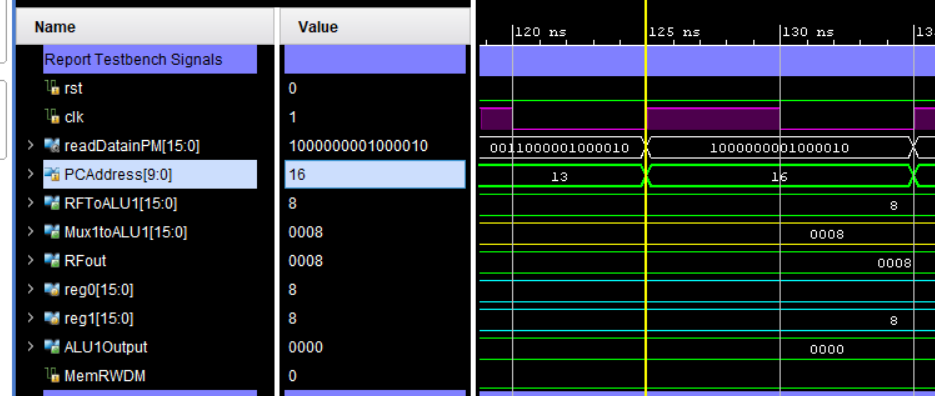
Fklt just branches if the source register is less than the target register (r1 in this case). Again, r0 and r1 are both 8, so neither is less than the other, and thus the branch is NOT taken. Therefore the Program Counter only increments by 1 as usual.



**0010000001000010; //fklt $r0,$r1,$r0**

**0011000001000010; //fkle $r0,$r1,$r0 fork to PC + 1 + 2 if $r0 <= $r1. fork to 16 bc 8 and 8 equal**

This instruction is the same as fklt, except the branch will also be taken if the register values are equal. They are equal and thus the PC is advanced 2 + PC + 1 to get to a PC address of 16.

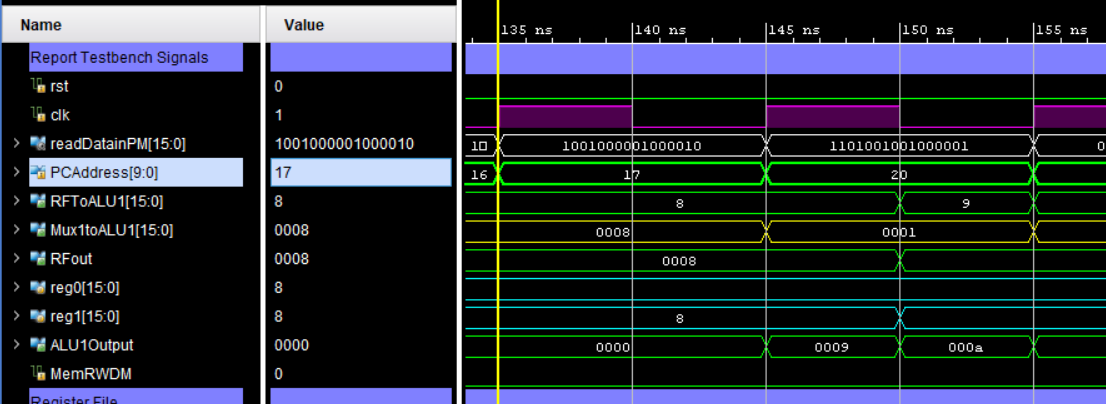
****

**0011000001000010; //fkle $r0,$r1,$r0**

**1000000001000010; //fkgt $r0,$r1,$r0 fork to PC + 1 + 2 if $r0 > $r1. don't fork bc 8 and 8 equal**

AND **1001000001000010; //fkge $r0,$r1,$r0 fork to PC + 1 + 2 if $r0 >= $r1. fork to 20 bc 8 and 8 equal**

These instructions have the exact opposite function of fklt and fkle. Fkgt branches if the source register is greater than the target register and Fkge is the same but also branches if the register values are equal. Since the values in registers r0 and r1 are stil both 8, the branch will NOT be taken when fkgt is called and WILL be taken when fkge is called. The operation of these instructions can be seen in the figure below. FKGE causes the program counter to be icnremented by 3 just as the FKLE instruction previously. Once the instructions are complete, the Program Counter will advance to address 21 upon the next positive edge of the instruction.



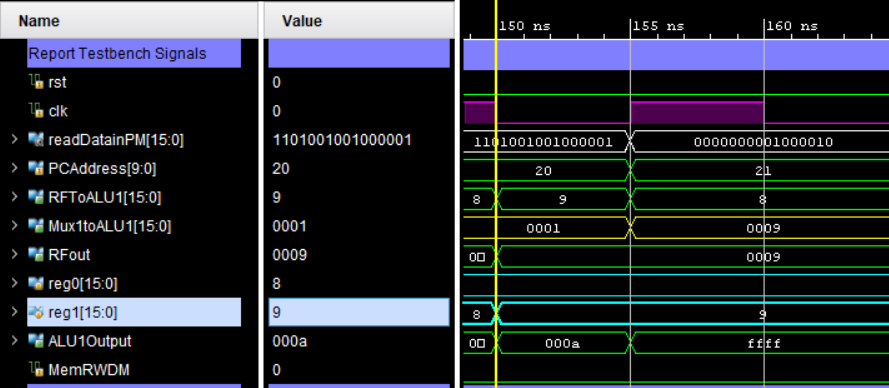
**1000000001000010; //fkgt $r0,$r1,$r0**

AND

**1001000001000010; //fkge $r0,$r1,$r0**

**1101001001000001; //plusi $r1,$r1,000001 add 1 to $r1 and store in $r1**

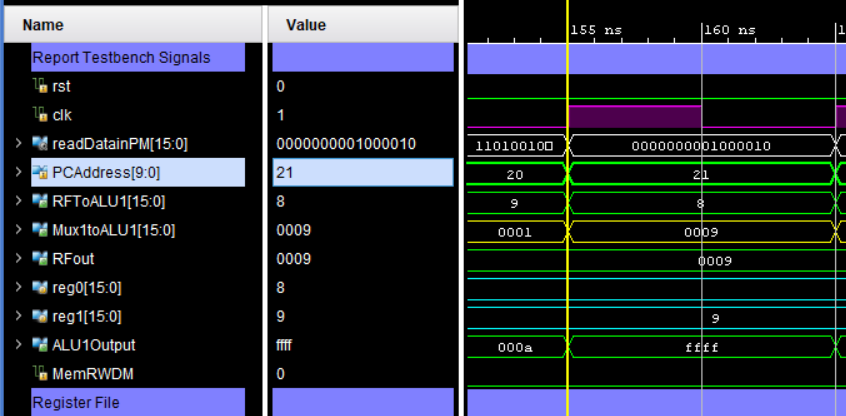
This is simply another plusi instruction. It has the same operation as the 2 described previously. In this case it adds 1 to register r1 and stores it in r1. So register r1 now has a value of 9. This will be used to activate the branch in the next instruction which is the beginning of a loop.



**1101001001000001; //plusi $r1,$r1,000001**

**0000000001000010; //fkeq $r0,$r1,$r0 fork to PC + 1 + 2 if $r0 and $r1 equal. don't fork first time. go to 22. second time fork to 24**

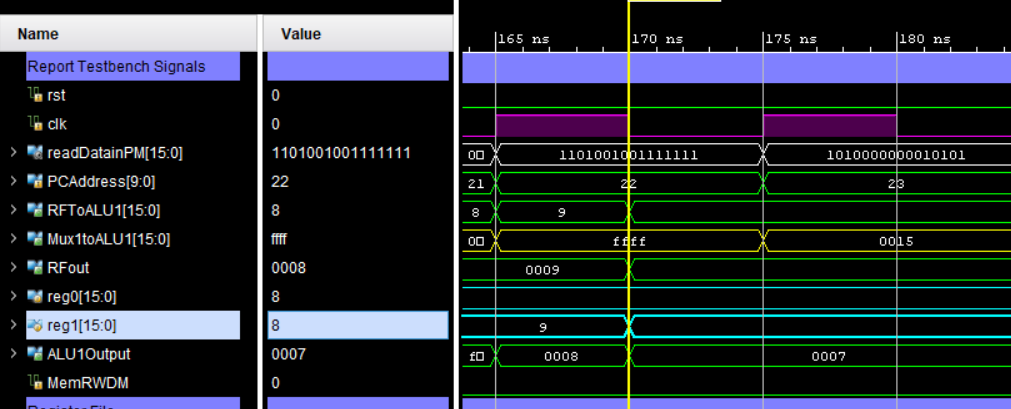
This instruction serves as the sentinel value for a loop. It checks if the values in r1 and r0 are equal and branches if they are. Since during the first iteration, they are 8 and 9, the branch will NOT be taken. The PC address only increments by 1 and thus becomes 21 upon the instruction’s positive clock edge.



**0000000001000010; //fkeq $r0,$r1,$r0**

**1101001001111111; //plusi $r1,$r1,111111 subtract 1 from $r1 to make it equal to $r2 again.**

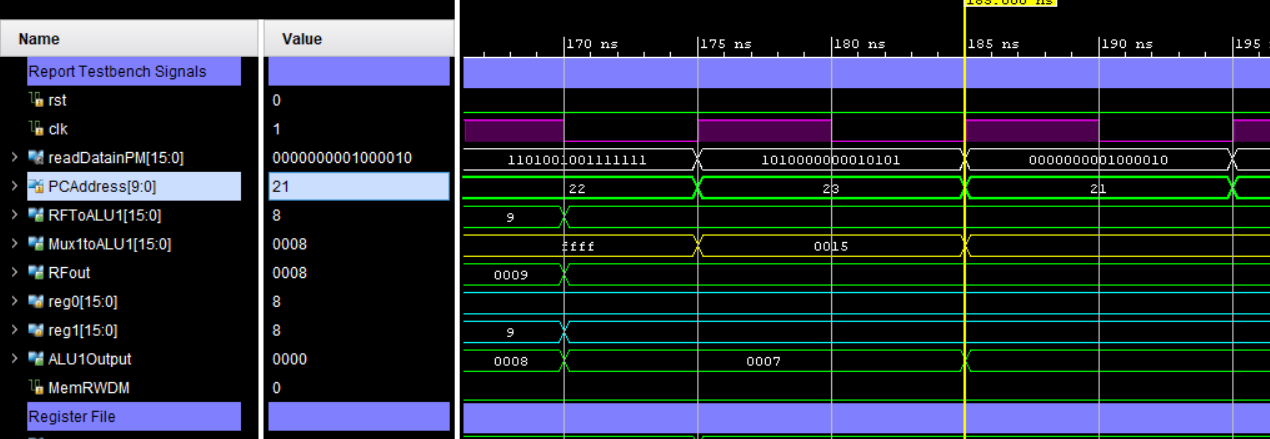
Here is another plusi instruction. This only simples adds -1 to the value of r1 (currently 9) to get it back to 8 so that when the loop returns to instrction 21, the branch will be taken because the value of both r0 and r1 will be equal.



**1101001001111111; //plusi $r1,$r1,111111**

**1010000000010101; //jump 0000010101 jump back two instructions to 21. Now $r0 and $r1 are equal again so instruction 21 will fork to instruction 24 where the HALT instruction is**

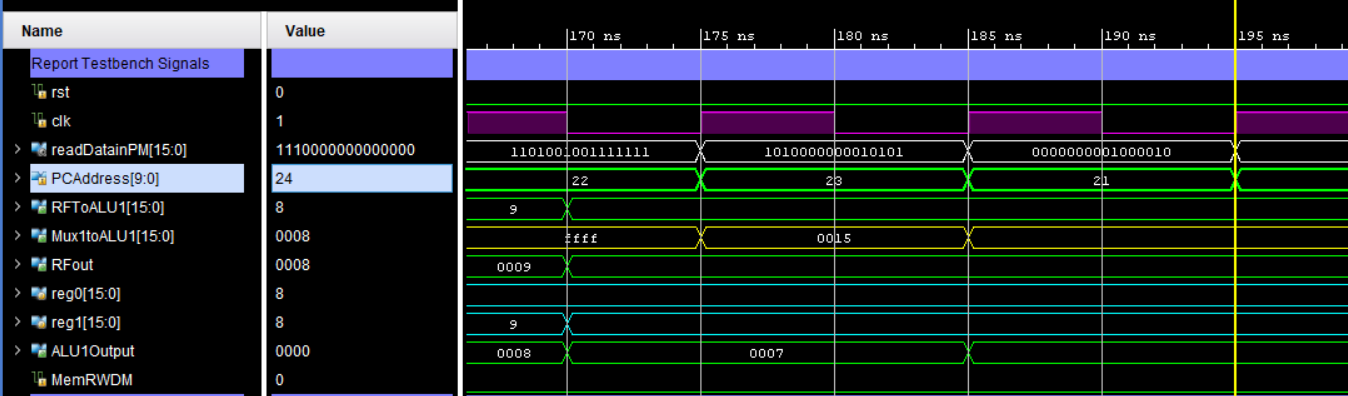
This is a jump instruction. In my program, when a Jump instruction is called it will set the program counter directly to whatever address is included in the last 10 digits of the instruction code. In this case that equals decimal 21. So the Program Counter address jumps back to instruction 21. It can be seen that the PC address after this instruction is 21 again.



**1010000000010101; //jump 0000010101**

**0000000001000010; //fkeq $r0,$r1,$r0 fork to PC + 1 + 2 if $r0 and $r1 equal. don't fork first time. go to 22. second time fork to 24**

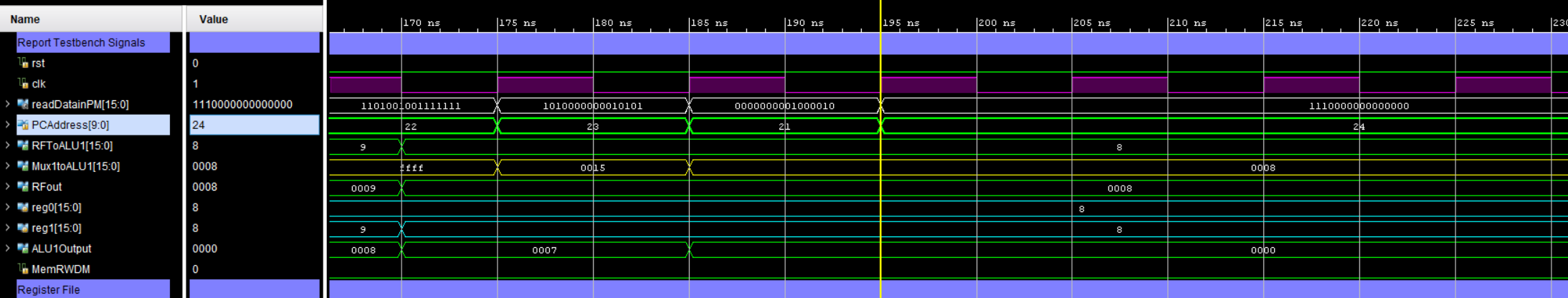
This is simply instruction 21, except this time the values in r0 and r1 are now equal, and thus the branch is taken. Since the branch address is 2 again, the Program Counter will be advanced by 3 to instruction 24, which is where the final HALT instruction is located. It can be seen that Program Counter has now advanced to address 24.



**0000000001000010; //fkeq $r0,$r1,$r0**

**1110000000000000; //HALT the program**

This is the HALT instruction. When this instruction is called, the last mux that sends the Program Counter its new value to be written on the next positive clock edge does not advance, Its value is set to latch throughout each clock cycle and thus the Program Counter never advances. This completes execution and demonstration of the assembly program.



**1110000000000000; //HALT the program**