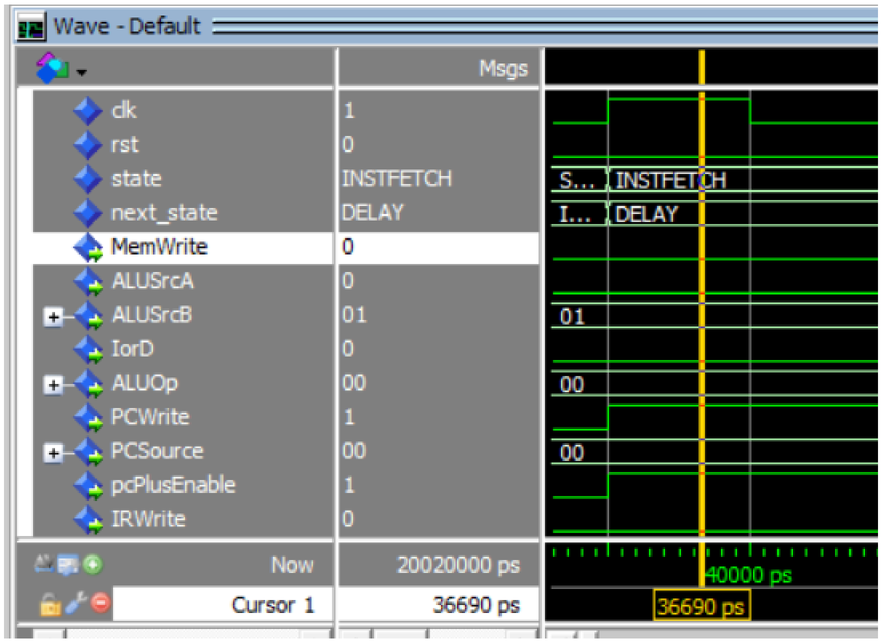
Gary Quilligan

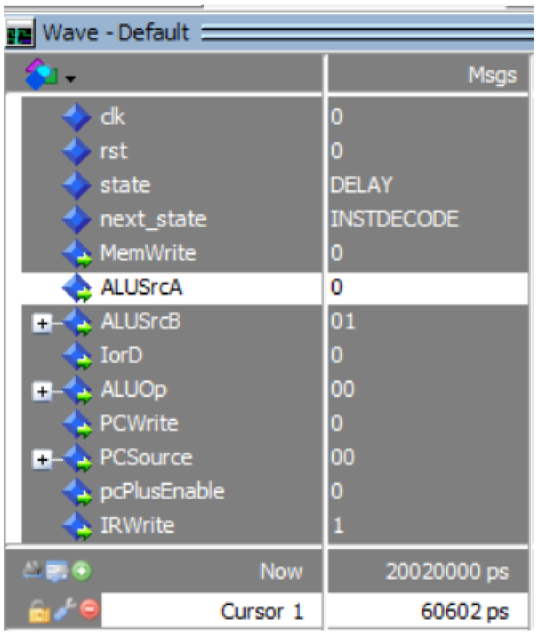
Test Case Simulation Annotations

InstFetch, Delay, and InstDecode



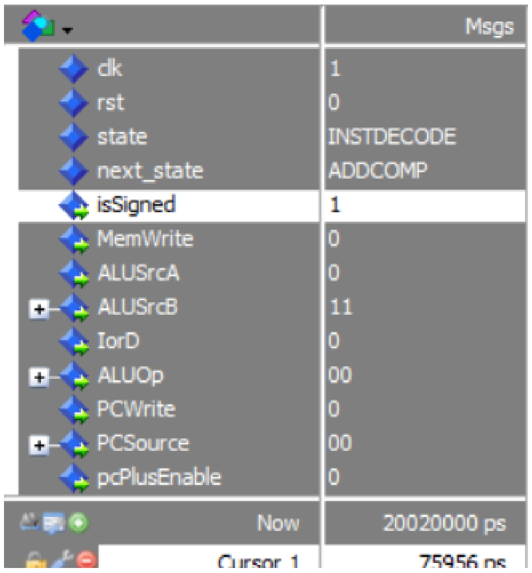
	Msgs
clk	1
rst	0
state	INSTFETCH
next_state	DELAY
MemWrite	0
ALUSrcA	0
ALUSrcB	01
IorD	0
ALUOp	00
PCWrite	1
PCSource	00
pcPlusEnable	1
IRWrite	0
Now	20020000 ps
Cursor 1	36690 ps

In the INSTFETCH state we want to read the next instruction from memory so MemWrite = ‘0’. ALUSrcA = ‘0’ and ALUSrcB = “01” to add 4 to the PC. IorD = ‘0’ because we want instruction memory not data memory. ALUOp = “00” in my implementation just tells alu_control to do addition. PCWrite = ‘1’ to write PC+4, and I made a register coming out of the alu to store the current pc+4 value, and this is only enabled on INSTFETCH (for possible JAL instruction).



	Msgs
clk	0
rst	0
state	DELAY
next_state	INSTDECODE
MemWrite	0
ALUSrcA	0
ALUSrcB	01
IorD	0
ALUOp	00
PCWrite	0
PCSource	00
pcPlusEnable	0
IRWrite	1
Now	20020000 ps
Cursor 1	60602 ps

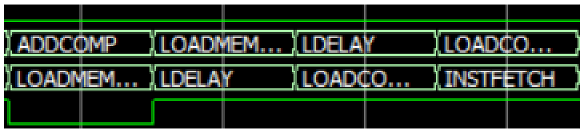
We only write to the instruction register after a clock cycle has passed so I put a delay state between instfetch and instdecode. It can be seen that IRWrite now = ‘1’.



	Msgs
clk	1
rst	0
state	INSTDECODE
next_state	ADDCOMP
isSigned	1
MemWrite	0
ALUSrcA	0
ALUSrcB	11
IorD	0
ALUOp	00
PCWrite	0
PCSource	00
pcPlusEnable	0
Now	20020000 ps
Cursor 1	75956 ps

This is the instdecode state, in case of a branch instruction in advance we calculate the branch address in relation to PC. ALUSrcB = “11” indicates that the branch address is coming from the sign-extend and 2-left-shifted offset. isSigned = ‘1’ because it is signExtended. This result is stored in the ALUOUT register in case of branch instruction.

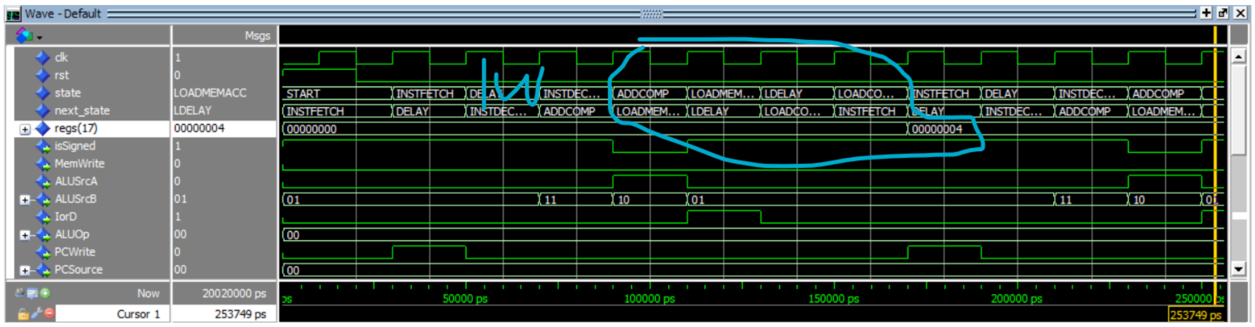
Test Case 1



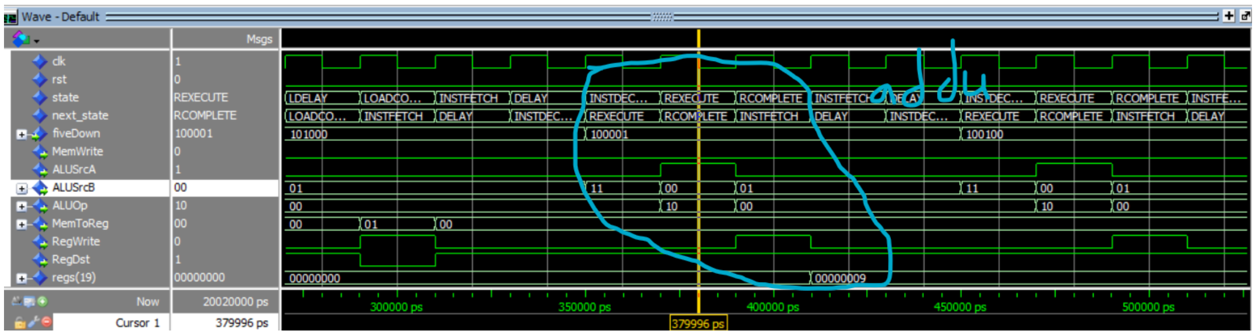
Load structure is address computation -> load memory -> delay -> load complete.

```
when ADDCOMP =>
    ALUOp <= "00";
    ALUSrcA <= '1';
    ALUSrcB <= "10";
    isSigned <= '0';
    if(opCode = "101011")then
        next_state <= STOREMEMACC;
    else
        next_state <= LOADMEMACC;
    end if;
when STOREMEMACC =>
    IorD <= '1';
    MemWrite <= '1';
    next_state <= INSTFETCH;
when LOADMEMACC =>
    IorD <= '1';
    MemWrite <= '0';
    next_state <= LDELAY;
when LDELAY =>
    next_state <= LOADCOMPLETE;
when LOADCOMPLETE =>
    MemToReg <= "01";
    RegDst <= '0';
    RegWrite <= '1';
    next_state <= INSTFETCH;
end case;
```

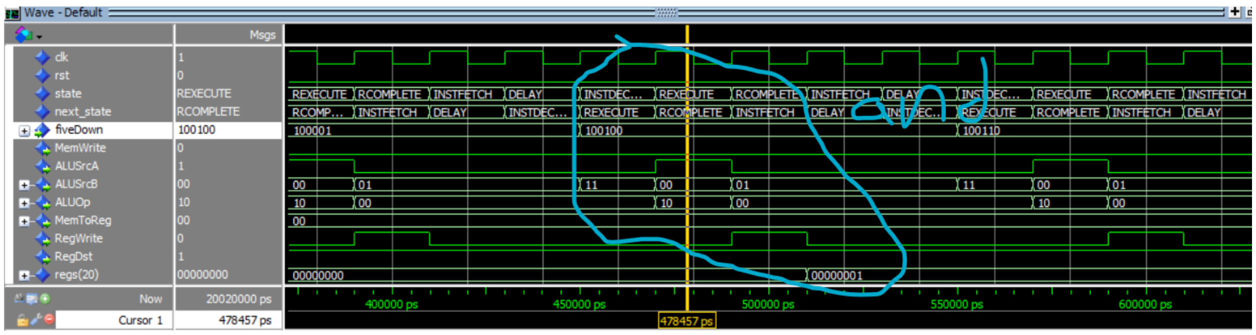
This is logic I used for lw and sw. First you zero-extend offset add to pc and then read from this place in memory using IorD = ‘1’ and MemWrite = ‘0’. You then delay because the RAM has a synchronous read, and then complete the load in LOADCOMPLETE.



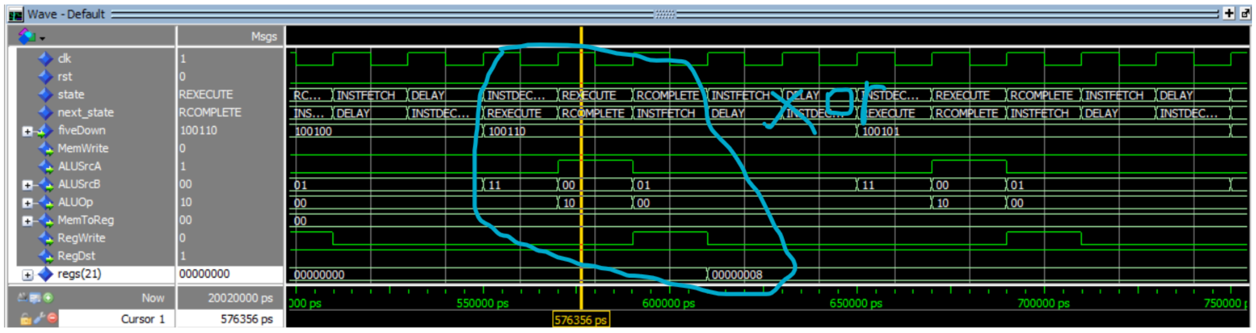
This is lw. We are loading into 0x4 into r17 which is circled.



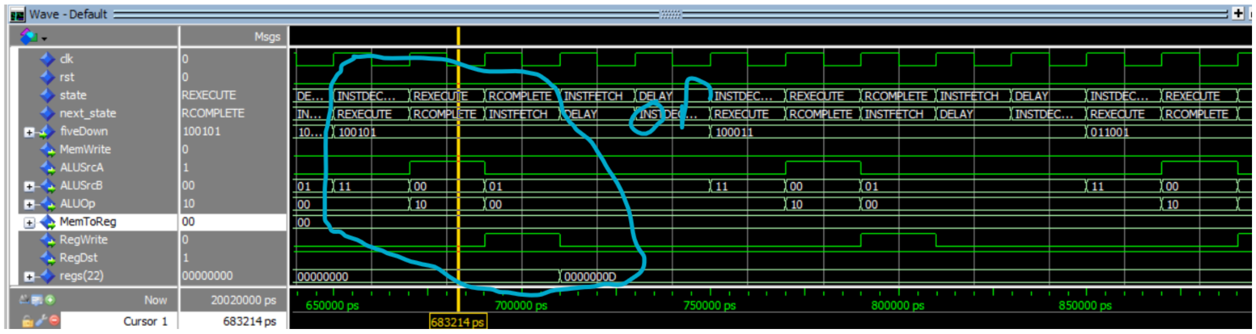
This is addu and fiveDown indicates it = “100001”, we are adding registers so ALUSrcA = ‘1’ and ALUSrcB = “00”, ALUOp = “10” for r-operations, and alu_control uses bottom five bits to determine which operation to do.RegDst = ‘1’ for R-operations because we are saving to rd. we are adding 5 and 4 and we get 9 in r19 as seen at bottom of blue circle.



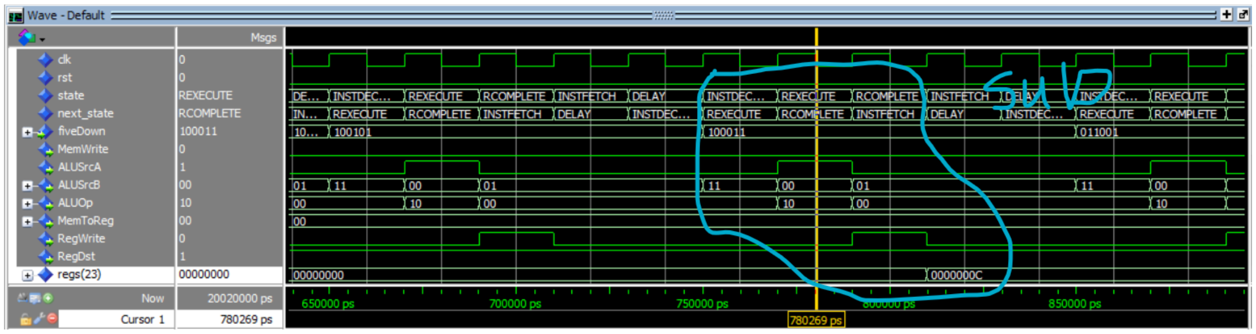
This is and, fiveDown = “100100”. We are anding 1001 and 0101 and get 1 in r20 register seen at bottom of circle.



This is xor, fiveDown = “100100”. We are xoring 1001 and 0001 and get 8 in r21 register seen at bottom.

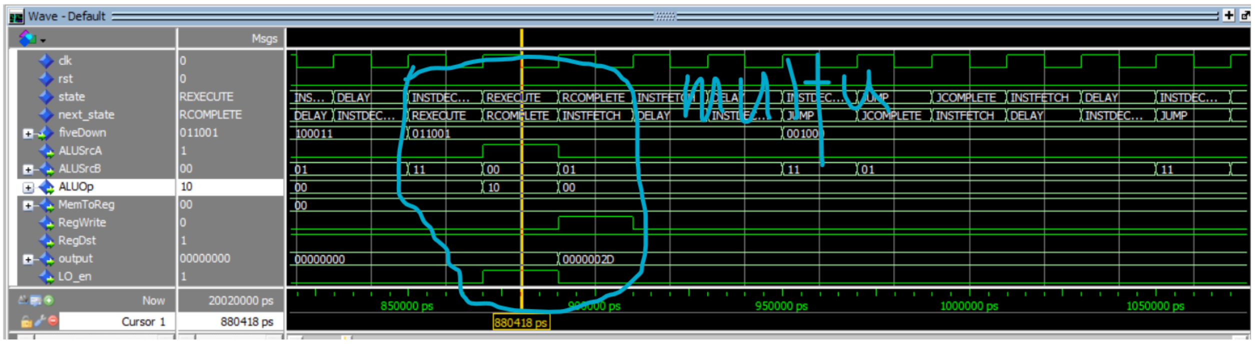


This is or, fiveDown = “100101”. We are oring 1001 and 0100 which gives 1101 which is 0xD which is stored in r22.



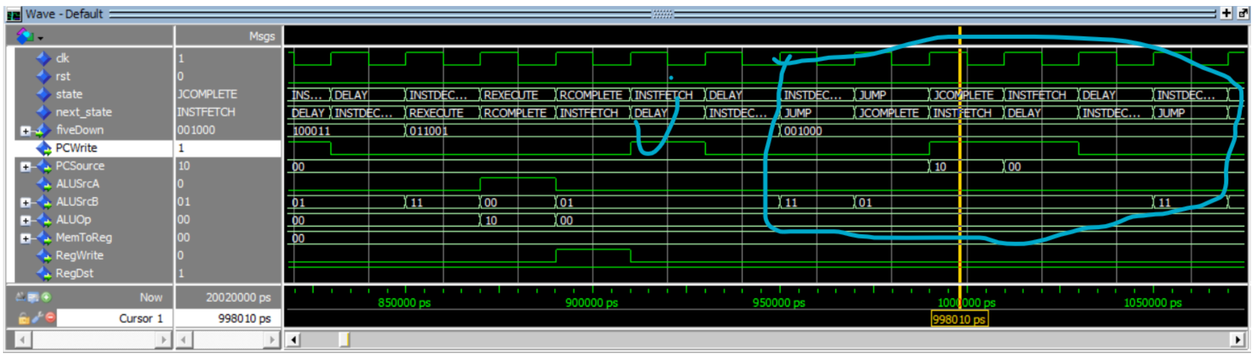
Signal	Msgs
clk	0
rst	0
state	REXECUTE
next_state	RCOMPLETE
fiveDown	100011
MemWrite	0
ALUSrcA	1
ALUSrcB	00
ALUOp	10
MemToReg	00
RegWrite	0
RegDst	1
regs(23)	00000000
Now	20020000 ps
Cursor 1	780269 ps

This is sub, fiveDown = “100011”. We are subtracting 1 from 13 which is 0xC which is stored in r23.



Signal	Msgs
clk	0
rst	0
state	REXECUTE
next_state	RCOMPLETE
fiveDown	011001
ALUSrcA	1
ALUSrcB	00
ALUOp	10
MemToReg	00
RegWrite	0
RegDst	1
output	00000000
LO_en	1
Now	20020000 ps
Cursor 1	880418 ps

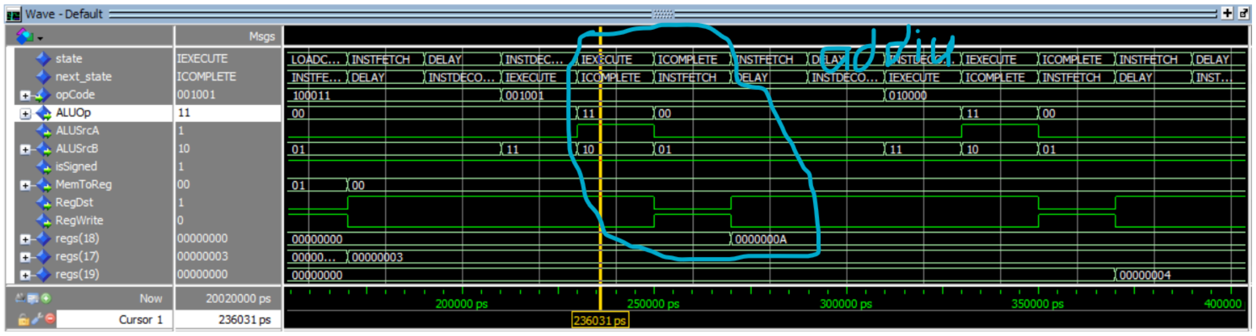
This is multu, fiveDown = “011001”, we are multiplying 9 and 5 which is 45 or 0x2D which is stored in the LO register. It can be seen that LO_en = ‘1’ which happens only in multiply operations.



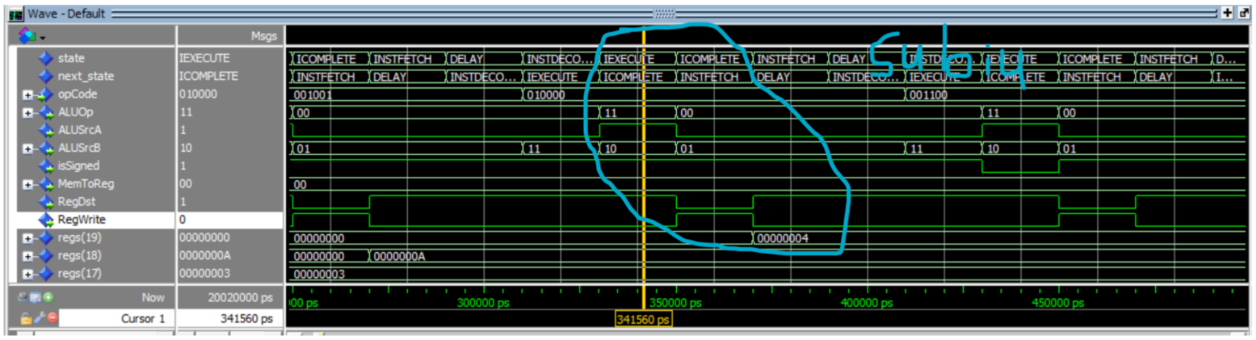
Signal	Msgs
clk	1
rst	0
state	JCOMPLETE
next_state	INSTFETCH
fiveDown	001000
PCWrite	1
PCSource	10
ALUSrcA	0
ALUSrcB	01
ALUOp	00
MemToReg	00
RegWrite	0
RegDst	1
Now	20020000 ps
Cursor 1	998010 ps

This is j, fiveDown = “001000”, PCSource = “10” which indicates that IR[25-0] shifted left two and with PC[31-28] concatenated on top is fed into the PC source mux and then PCWrite = ‘1’ to write this to the program counter. This executes a jump. At very right of image you can see that the next_state is jump, and we are continuing to jump endlessly.

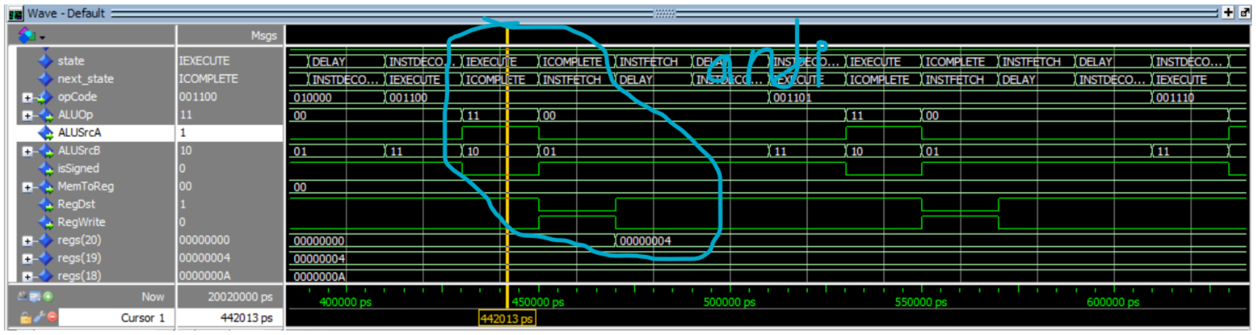
Test Case 2



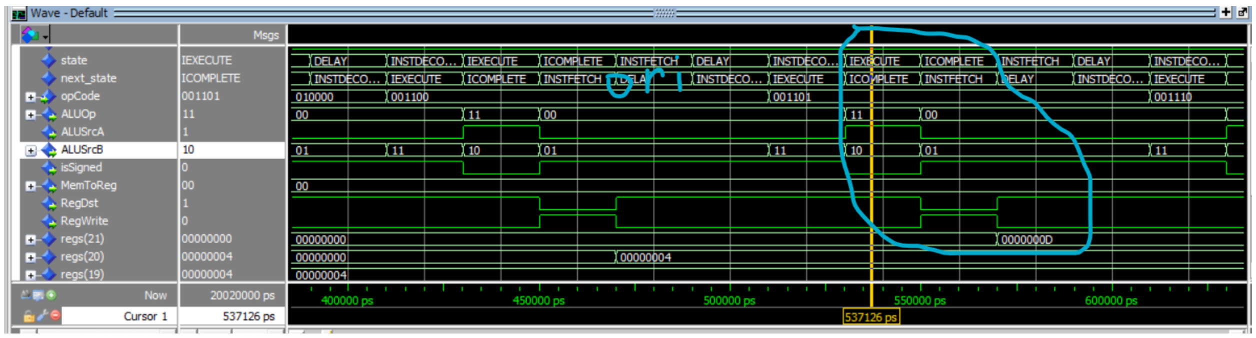
This is an addiu command adding 3+7. In the IEXECUTE stage, ALUSrcA = ‘1’ to add register rs with ALUSrcB = “10” which is the signExtended immediate. isSigned = ‘1’ because on addiu we signExtend, ALUOp = “11” for I-operations, and the ALUControl uses opCode on I-operations to choose which operation. In the ICOMPLETE stage, MemToReg = “00” and regDST = ‘0’ to signify that rt is the destination register rather than rd. It can be seen that r18 is updated with the sum 0xA.



This is a subiu command using 10-6. In the IEXECUTE stage, ALUSrcA = ‘1’ to subtract register rs with ALUSrcB = “10” which is the signExtended immediate. isSigned = ‘1’ because on subiu we signExtend. In the ICOMPLETE stage, MemToReg = “00” and regDST = ‘0’ to signify that rt is the destination register rather than rd. It can be seen that r19 is updated with the difference 0x4.

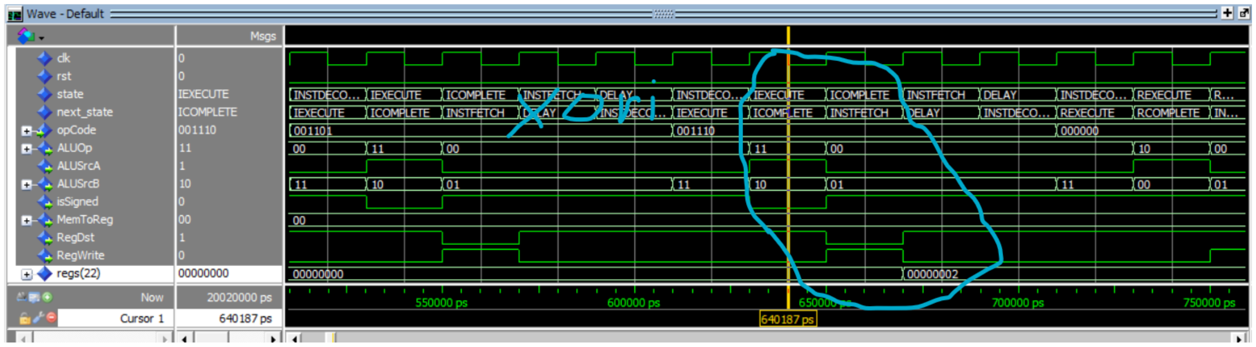


This is the andi command anding 4 and 5 (line 4 of .mif). The immediate is zero-extended in this case so isSigned = ‘0’. Besides this it is just like other immediate operations. In bottom of blue circle, it can be seen that r20 is updated with 4 (since 100 and 101 = 100).



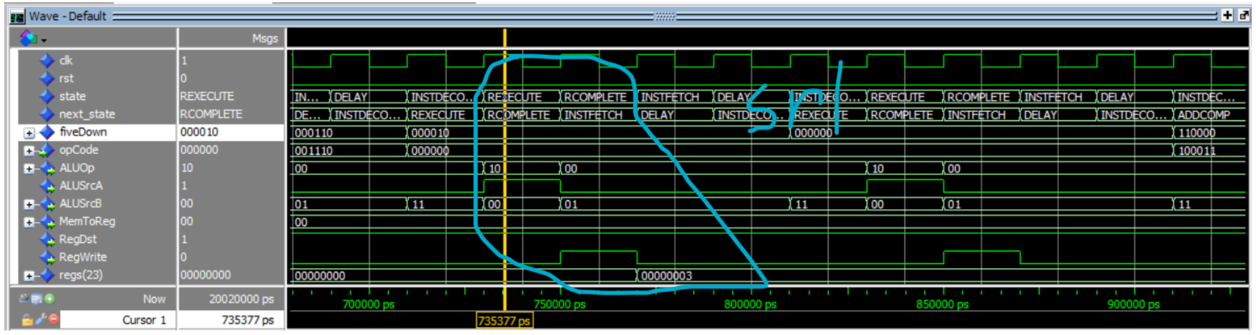
```
                Msgs
state           IEXECUTE
next_state      ICOMPLETE
opCode          001101
ALUOp           11
ALUSrcA         1
ALUSrcB         10
isSigned        0
MemToReg        00
RegDst          1
RegWrite        0
regs(21)        00000000
regs(20)        00000004
regs(19)        00000004
Now             20020000 ps
Cursor 1        537126 ps
```

This is the ori command which has same control signals as andi commands and is also zero-extended, only difference is that alu_control uses opCode to choose which immediate operation to do. We are oring values 1001 and 0100 which gives 1101 or 0xD which r21 is updated with after ICOMPLETE.



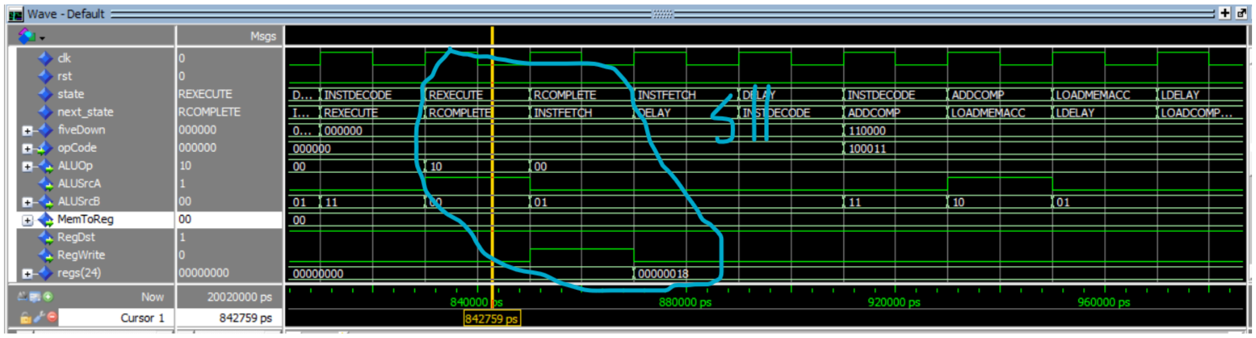
```
                Msgs
clk             0
rst             0
state           IEXECUTE
next_state      ICOMPLETE
opCode          001110
ALUOp           11
ALUSrcA         1
ALUSrcB         10
isSigned        0
MemToReg        00
RegDst          1
RegWrite        0
regs(22)        00000000
Now             20020000 ps
Cursor 1        640187 ps
```

This is the xori command, just like andi and ori is zero-extended. We are xoring values 0100 and 0110 which gives 0010 or 0x2 which r22 is updated with after the ICOMPLETE stage.



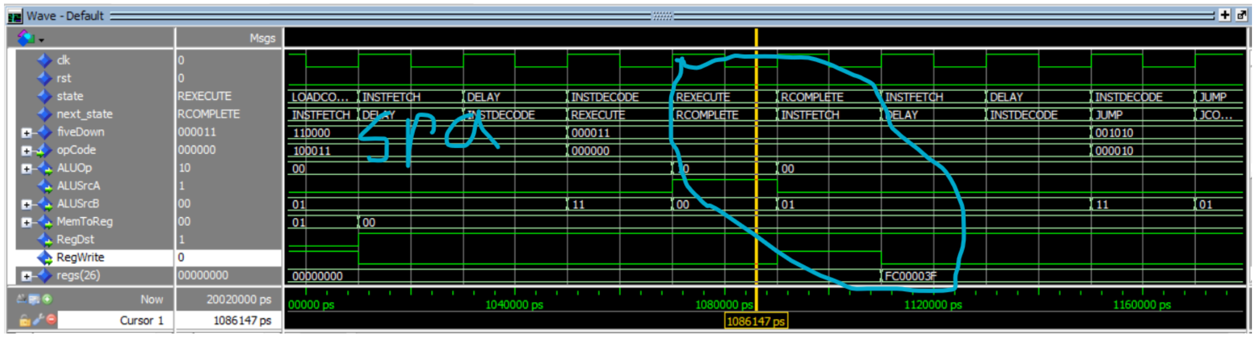
```
                Msgs
clk             1
rst             0
state           REXECUTE
next_state      RCOMPLETE
fiveDown        000010
opCode          000000
ALUOp           10
ALUSrcA         1
ALUSrcB         00
MemToReg        00
RegDst          1
RegWrite        0
regs(23)        00000000
Now             20020000 ps
Cursor 1        735377 ps
```

This is the srl command, we are operating on two registers so ALUSrcA = ‘1’ and ALUSrcB = “00”. The RegDst = ‘1’ now because rd is the destination register for r-operations. We are logically shifting 1101 right two times, so 0011 or 0x3 should be saved to r23, which it is next cycle.



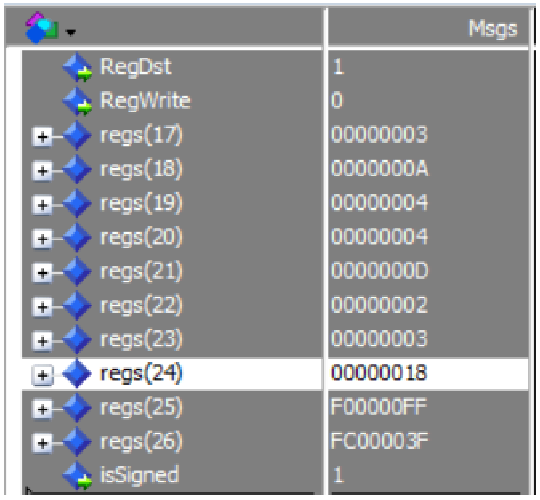
	Msgs
clk	0
rst	0
state	REXECUTE
next_state	RCOMPLETE
fiveDown	000000
opCode	000000
ALUOp	10
ALUSrcA	1
ALUSrcB	00
MemToReg	00
RegDst	1
RegWrite	0
regs(24)	00000000
Now	20020000 ps
Cursor 1	842759 ps

This is the sll command. We are logically left shifting 3 3 times. So 0011 -> 11000 = 0x18 which is stored into r24 on the next cycle.



	Msgs
clk	0
rst	0
state	REXECUTE
next_state	RCOMPLETE
fiveDown	000011
opCode	000000
ALUOp	10
ALUSrcA	1
ALUSrcB	00
MemToReg	00
RegDst	1
RegWrite	0
regs(26)	00000000
Now	20020000 ps
Cursor 1	1086147 ps

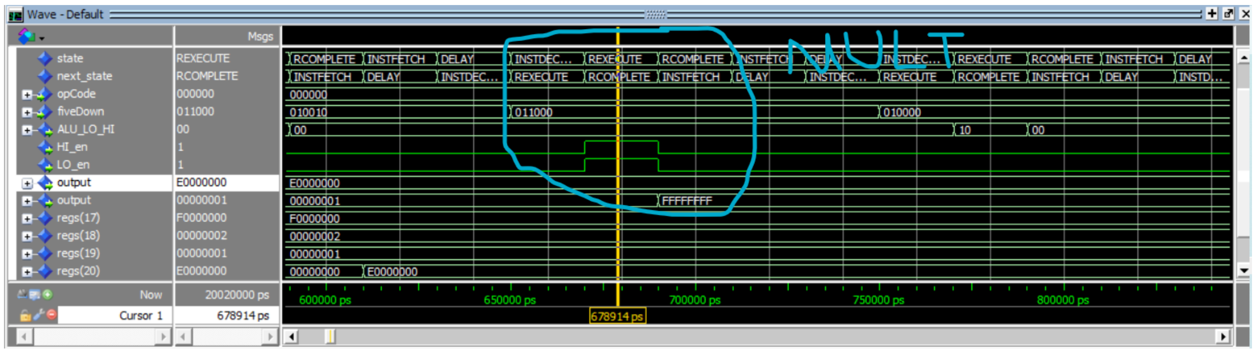
This is the sra command. We are arithmetically right shifting F00000FF 2 times, so we should get FC00003F which appears in r26 a cycle later in bottom of blue circle.



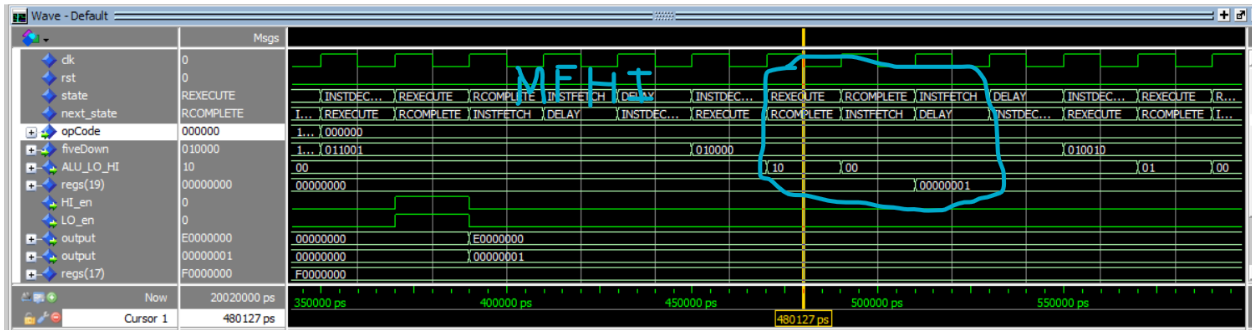
	Msgs
RegDst	1
RegWrite	0
regs(17)	00000003
regs(18)	0000000A
regs(19)	00000004
regs(20)	00000004
regs(21)	0000000D
regs(22)	00000002
regs(23)	00000003
regs(24)	00000018
regs(25)	F00000FF
regs(26)	FC00003F
isSigned	1

Final values of registers for test case 2.

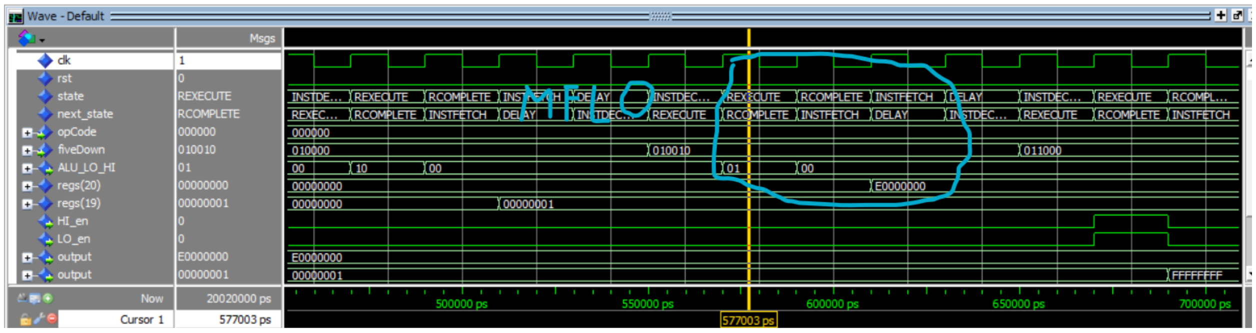
Test Case 4



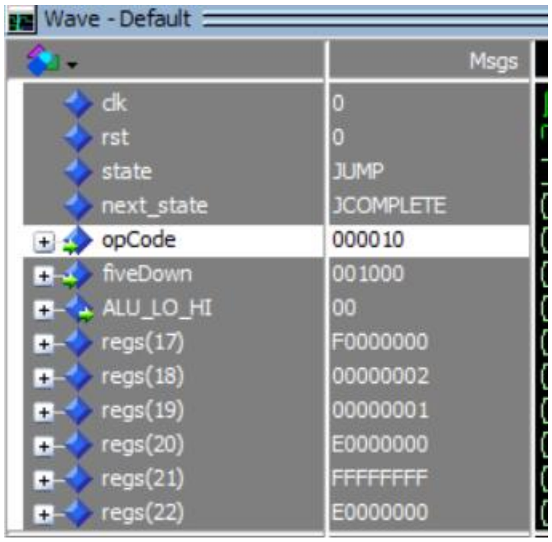
This is the MULT command of F0000000 and 00000002, which has a result of FFFFFFFF in the HI register and E0000000 in the LO register. It can be seen that Hi_en and Lo_en are ‘1’ with this operation.



This is the mfhi command(IR[5-0] = “010000”), we are meant to move the value 1 into r19(line 4 of .mif). ALU_LO_HI is set to “10” to select the HI_REG output to go to register data memory. In the next clock cycle r19 is updated with 1.



This is the mflo command(IR[5-0] = “010010”), we are meant to move the value E0000000 into r20(line 5 of .mif). ALU_LO_HI is set to “01” to select the LO_REG output to got to register data memory. In the next clock cycle r20 is updated with E0000000.

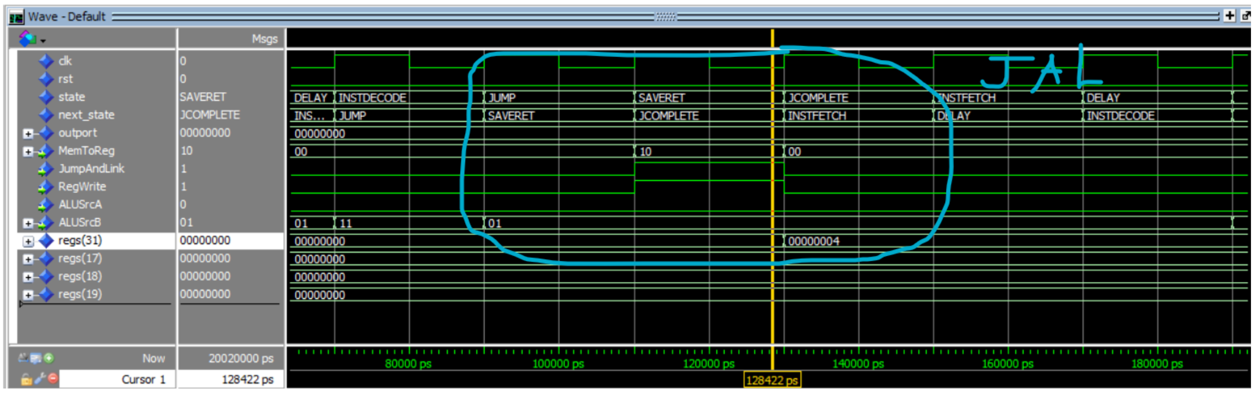


Wave - Default

	Msgs
clk	0
rst	0
state	JUMP
next_state	JCOMPLETE
opCode	000010
fiveDown	001000
ALU_LO_HI	00
regs(17)	F0000000
regs(18)	00000002
regs(19)	00000001
regs(20)	E0000000
regs(21)	FFFFFFFF
regs(22)	E0000000

Final register values for test 4.

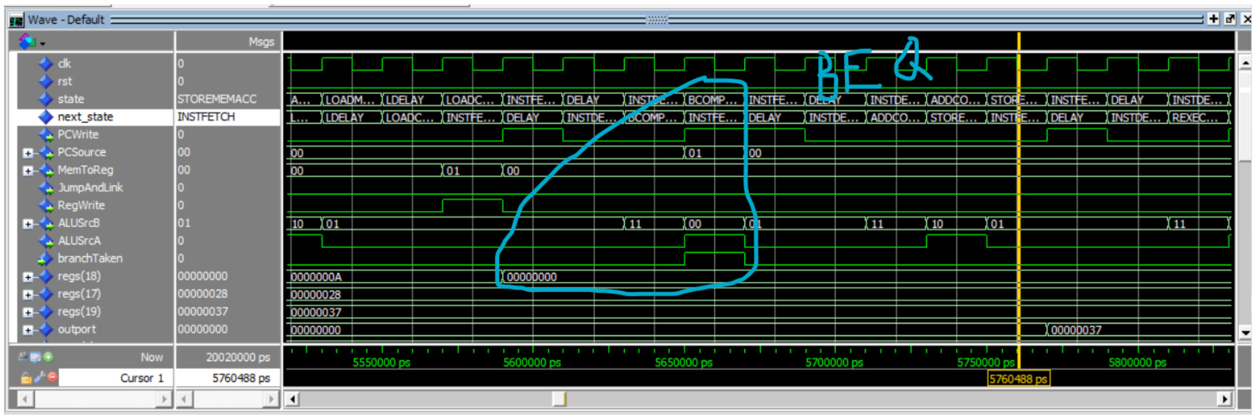
Test Case 7



Wave - Default

	Msgs
clk	0
rst	0
state	SAVERET
next_state	JCOMPLETE
outport	00000000
MemToReg	10
JumpAndLink	1
RegWrite	1
ALUSrcA	0
ALUSrcB	01
regs(31)	00000000
regs(17)	00000000
regs(18)	00000000
regs(19)	00000000
Now	20020000 ps
Cursor 1	128422 ps

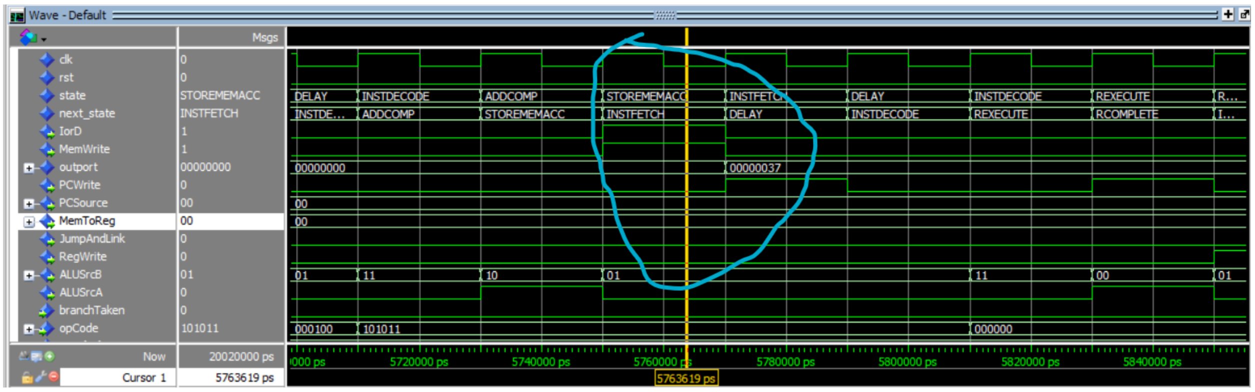
This is the JAL command, in the Jump state, after the INSTDECODE state, we check if the OPCODE matches the JAL command, if it does we go to the save return address state. In the SAVERET state it can be seen that the jumpAndLink signal changes to 1 and the regWrite signal changes to one, it can also be seen that r31 is updated with the value 4. I also used a 3x1 mux for the memToReg register which has one of its inputs wired to a register that stores the current PC+4 value.



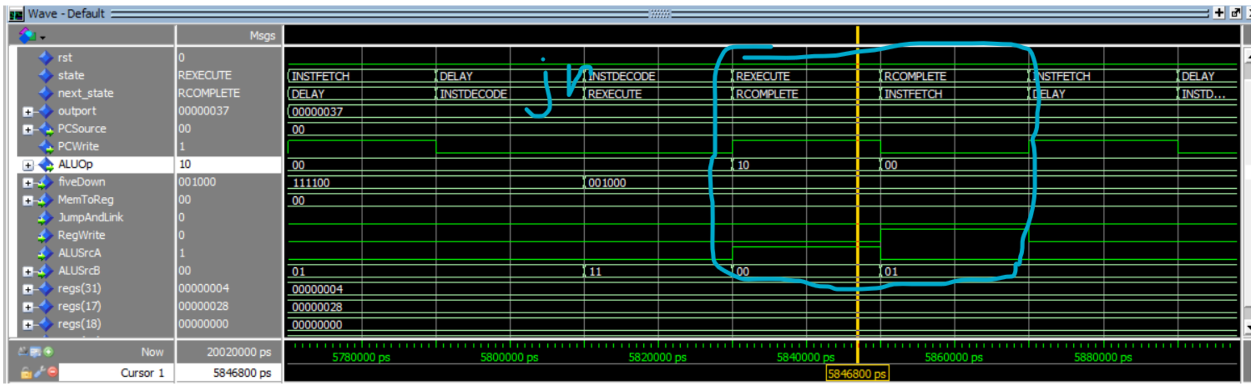
Wave - Default

	Msgs
clk	0
rst	0
state	STOREMEMACC
next_state	INSTFETCH
PCWrite	0
PCSource	00
MemToReg	00
JumpAndLink	0
RegWrite	0
ALUSrcB	01
ALUSrcA	0
branchTaken	0
regs(18)	00000000
regs(17)	00000028
regs(19)	00000037
outport	00000000
Now	20020000 ps
Cursor 1	5760488 ps

This is the BEQ command. In TestCase7 the line " beq $s2, $zero, 8 "says that when r18 equals 0 branch to address 8. It can be seen that PCSource = "01" and PCWrite = '1' indicating a branch to a different address, and the branchTaken = '1' allowing the PC to be written to. ALUSrcA = '1' and ALUSrcB = "00" because this is a comparison of two registers.



This is the SW command. It can be seen that IorD is set to 1 to allow data rather than an instruction to go to memory, and memWrite is set to '1' to indicate that the data is being written. The next clock cycle outport is written with the sum of 1-10 which is 0x37 or 55 in decimal.



This is the JR command it is encoded as a R-operation with opcode "10", but I send IR[5-0] to the controller to let it know this is the JR command and that it should write the register value to the Program counter. It can be seen that PCSource = "00" and PCWrite = '1'.