Group 1

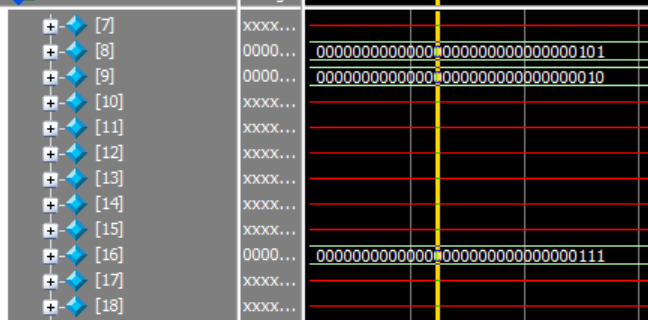
To implement add, I only had to change the control signals. These include regDst\_out, ALUSrc\_out, etc.

For the add test I put the following program in:

addi $t0, $zero, 5 #put 5 in $t0

addi $t1, $zero, 2 #put 2 in $t1

add $s0, $t0, $t1 #put 7 in $s0

And received 7 in register $16 = $s0, as expected:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| PC | Instruction | Opcode | Func | Write add | Write data | Write cntrl |
| 0x00400010 | xxxxxxxx | 6'h0 | 6'h20 | X | X | 0 |

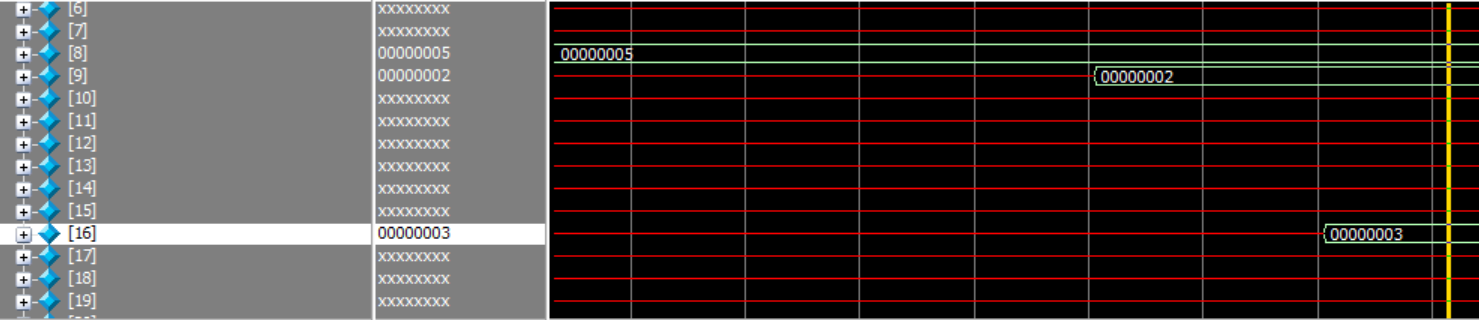
Sub had the same signals as add, except the alu opcode changed from 0010 to 0110 (subtract)

I tested it using:

addi $t0, $zero, 5 #put 5 in $t0

addi $t1, $zero, 2 #put 2 in $t1

sub $s0, $t0, $t1 #put 3 in $s0

As expected, I received 3 in $16=$s0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| PC | Instruction | Opcode | Func | Write add | Write data | Write cntrl |
| 0x0040000c | xxxxxxxx | 6'h0 | 6'h22 | X | X | 0 |

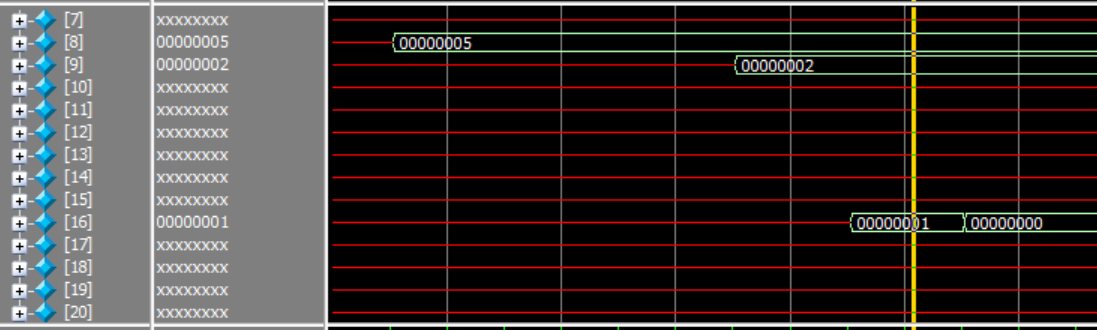
Slt was same, but had alu-op 0111 (set if less than).

I used the program:

addi $t0, $zero, 5 #put 5 in $t0

addi $t1, $zero, 2 #put 2 in $t1

slt $s0, $t1, $t0 #expect $s0 to get 1

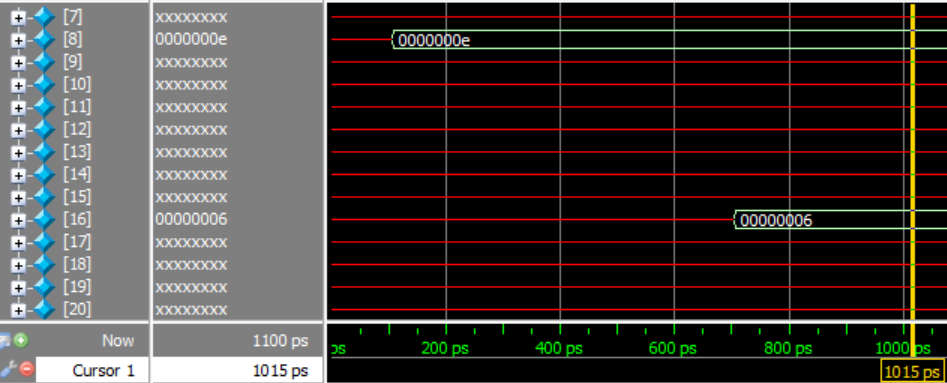
 slt $s0, $t0, $t1 #put $s0 back to 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| PC | Instruction | Opcode | Func | Write add | Write data | Write cntrl |
| 0x0040000c | xxxxxxxx | 6'h0 | 6'h2a | X | X | 0 |

andi:

Test program:

addi $t0, $zero, 14 #put 0b..0001110 = 14 in $t0

 andi $s0, $t0, 7 #expect 0b110 = 6 in $s0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| PC | Instruction | Opcode | Func | Write add | Write data | Write cntrl |
| 0x00400010 | xxxxxxxx | 6'h0 | 6'hx | X | X | 0 |

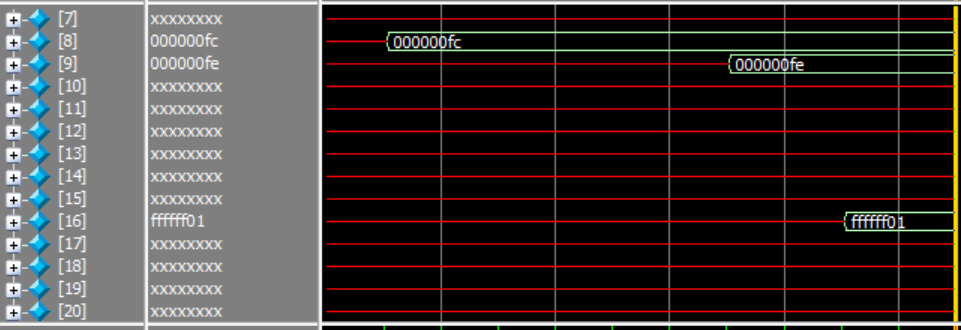
nor:

Test program:

addi $t0, $zero, 0xFc #put 0b00...1111 1100 in $t0

addi $t1, $zero, 0xFE #put 0b00...1111 1110 in $t1

nor $s0, $t0, $t1 #expect 0b11...0000 0001



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| PC | Instruction | Opcode | Func | Write add | Write data | Write cntrl |
| 0x00400010 | xxxxxxxx | 6'h0 | 6'h27 | X | X | 0 |

Group 2:

Testing of this group required examining the data-memory.v module. From the module description: Please note that we only model the first 32 words of memory. Thus, valid addresses to load and store data words are 0x10000000 to 0x1000007C.

I had the option to initialize the data memory with some data using data.txt. I chose instead to store a word in memory and then load it back to a register to verify sw and lw.

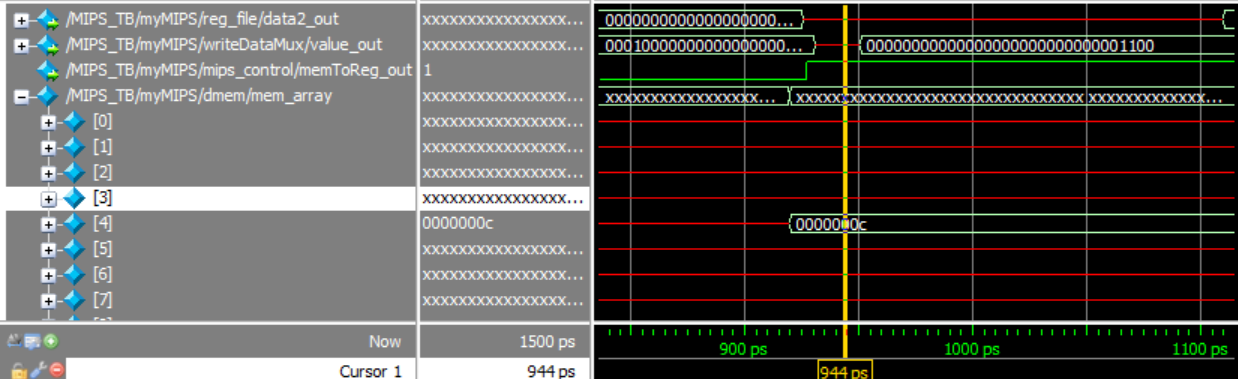
I used the following test program:

lui $s1, 0x1000 #choose 0x10000000

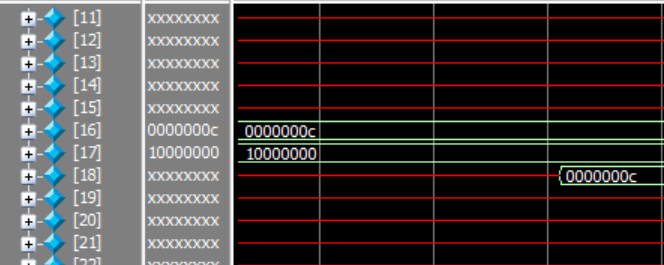
#user data segment default QTSpim: 0x10000000-0x10040000

addi $s0, $zero, 12 #store 12 in $s0

sw $s0, 16($s1) #store 12 in 0x10000010

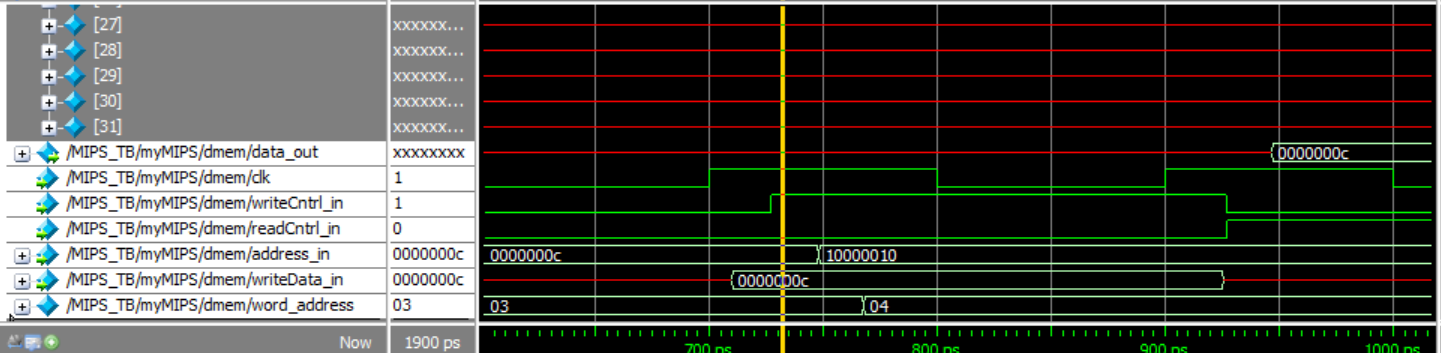
lw $s2, 16($s1) #retrieve 12 from memory

As can be seen, the 4th memory location receives 12 after the sw operation as expected.



Likewise, $s2=$18 receives 12 after the lw operation, as expected.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PC | Instruct | Imm | Write addrs/data/cntrl  For register file | Addrs  For data mem -> | Output data | Write data | Read control | Write control |
| 0x0040008 | 0xae300010 (sw $s0, 16($s1)) | 16 | 0x10/0xC/0 | 0xC | Xxxx | 0xC | 0 | 1 |
| 0x004000c | 0x8e320010  (lw $s2, 16($s1)) | 16 | 0x12/0xC/1 | 0x10000010 | 0xC | Xxxx | 1 | 0 |



**Group 3**

For branch, I first wanted to check if the branch signal goes to an and gate. Then check if the zero output from the ALU (1 if the inputs are equal) goes anywhere (should go to the same and gate). Output the and gate to a mux. I also added in the appropriate control signals.

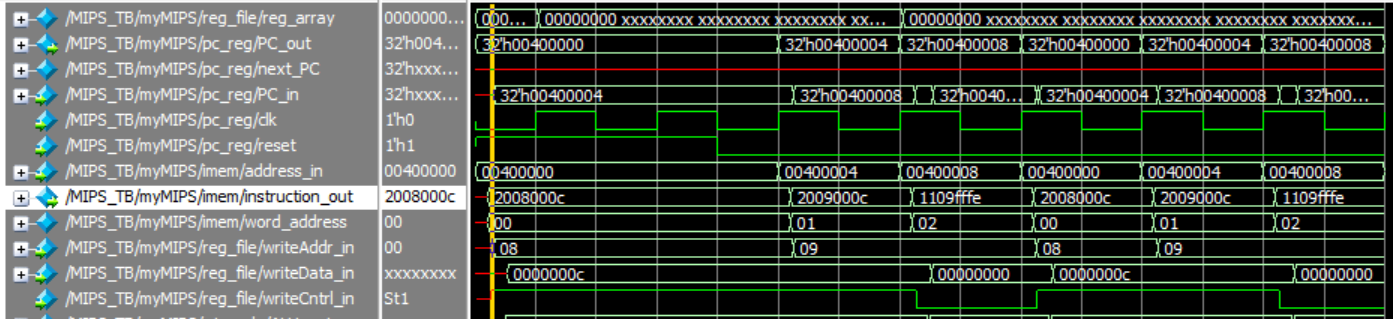
From the textbook: The beq instruction has three operands, two registers that are compared for equality, and a 16-bit offset used to compute the branch target address relative to the branch instruction address. Its form is beq $t1,$t2,offset. To implement this instruction, we must compute the branch target address by adding the sign-extended offset field of the instruction to the PC (see 277 in eText).

To implement this, I needed to add a shifter, and break connection to PC from simply PC+4 to the mux. I added another ALU which adds the sign/zero extended immediate (existing) which has been shifted left by two. Oddly, branch mux was spitting out one further PC than I wanted. Thus, I replaced my PC+4 input with just PC.

For the beq instruction, I used the following program:

START: addi $t0, $zero, 12 #set both $t0 and $t1 to 12

addi $t1, $zero, 12

 beq $t0, $t1, START #expect the PC to jump back to START after this instruction

As expected, this program loops forever, going back to the START address (0x0040 0000) every loop.

For bne, I added a new control signal. I also added an and gate, which takes the inverted output of zero output from the main alu, an inverter to invert this output, and an or gate. This or gate takes the outputs of the two branch AND gates. So, if beq and equal, 1. If bne and not equal, 1. The output of this or gate goes to the beq mux selector (breaking the original connection to just the beq AND). So if it’s 1, the mux chooses the branch adder result. Otherwise, chooses PC+4.

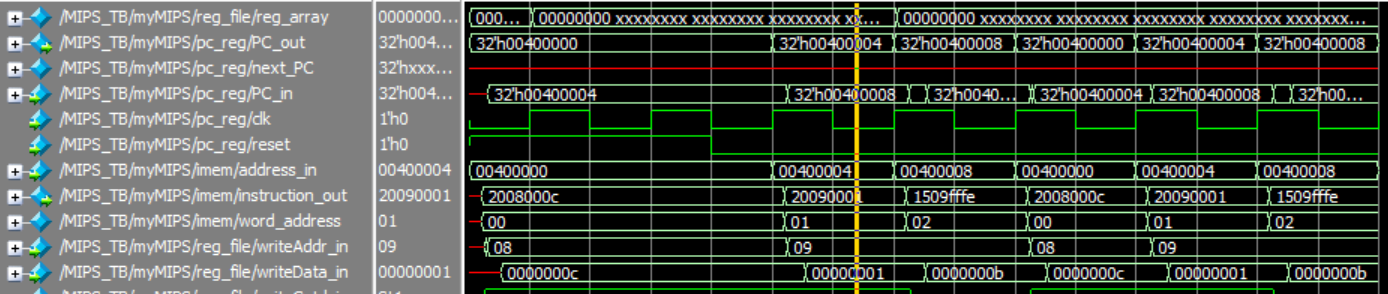
The program I used to test bne is:

START: addi $t0, $zero, 12 #set both $t0 and $t1 to 12

addi $t1, $zero, 1

bne $t0, $t1, START #expect the PC to jump back to START after this instruction

addi $t2, $zero, 4



As expected, this program loops forever, going back to the START address (0x0040 0000) every loop.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PC | Instruction | Zero signal | Jump | Branch |
| 0x0040 0008 | 0x1109fffe (beq) | 1 | 0 | 1 |
| 0x0040 0008 | 0x1509fffe (bne) | 0 | 0 | 0 |

**Group 4**

To implement j, jr, and jal, an additional mux needed to be added. The mux would be controlled by the control signal jump. This mux takes the output from the branch mux (0 input) and also the built jump address. It’s output would go to the PC (severing the original connection). The jump address would be built by taking the instruction address (Instruction[25:0]), left shifting it by 2, and concatenating this with PC+4[31:28].

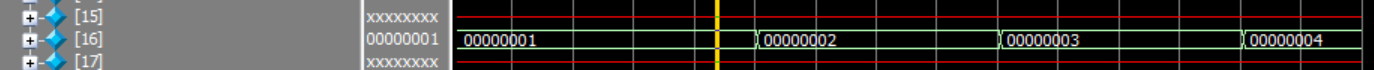
The program I used to check jump is as follows:

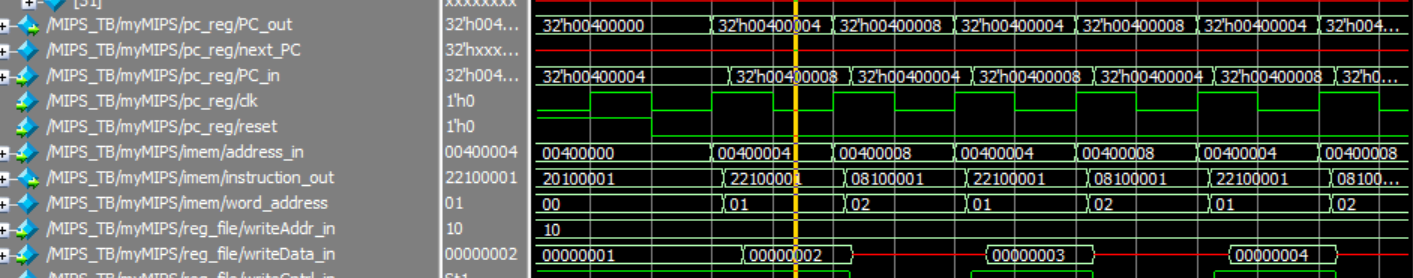
addi $s0, $zero, 1 #intialize $s0 to 1

STARTL: addi $s0, $s0, 1 #increment $s0

j STARTL #expect the PC to jump back to START after this instruction

addi $t2, $zero, 4

And I received the following output on $s0 = $16:

And a view of the PC and instruction\_out:

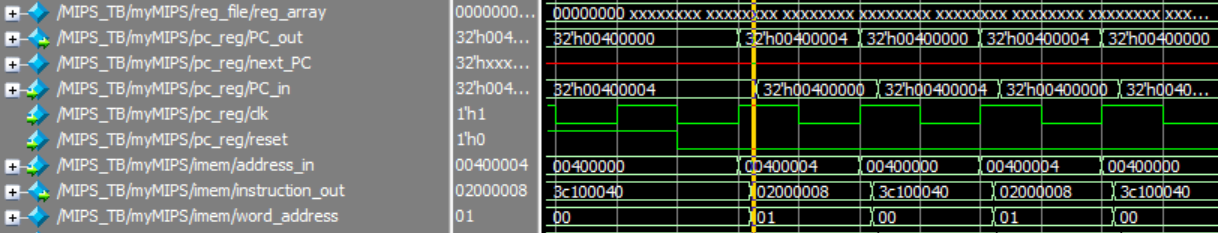
As expected, the jump instruction had the program jump back to the location of STARTL, 0x0040 0004.

Jump register (jr) would require yet another mux. I chose to put this mux after the jumpMuxOne. It would take jumpMuxOne\_out (0) and the register output (1), and be selected by a new control signal, jr. It’s output will go to PC. To get the address out of the register, we can pull it straight from the Read data 1 line on the register file.

The program I used to test jr is as follows: (test\_4\_2.s)

lui $s0, 0x0040 #intialize $s0 to starting instruction addresss

jr $s0 #jump to the starting address contained in $s0

I got the following output:

As expected, the jr instruction continues to jump up to the starting instruction address 0x0040 0000.

Finally, to implement jal I added another control signal, jal. When jal is called, I assert the jump signal as before, such that the pc jumps to the address. However, I also needed a way to update the $ra = #31 register.

In order to accomplish this, I modified the connections to two register file inputs: write register and write data. Write data now receives the output of a mux which chooses between the original connection (0) (the output of the mux which chooses between ALU result and Data memory) and PC+4 (1). It is selected by the jal control signal.

I also needed another mux to go to the write register input. This mux selects between the original input (0) (the mux which is selected by RegDst) and a constant (1) (31 = $ra address). It is also selected by the jal control signal.

Start by writing the jal control path, with alu opcode. Then, adding the jal control signal in. Then, create a write data mux. Then creating a write register mux.