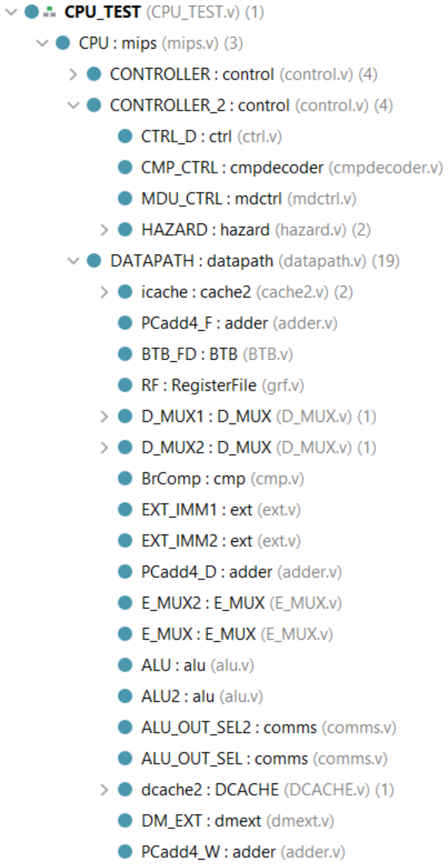
Project 04: Dual-Issue Superscalar

Procedure 1

We extend all our memory units with new ports for extra read and write operations. We can read two instructions from cache in one cycle and add one more controller to control another instruction. Also, we double ALU and can write back two results at the same time. However, according to the document, we try to simplified the program, data memory can only processor one instruction in a cycle. In this procedure, we do not consider any hazard.

Here is the whole project files we designed.

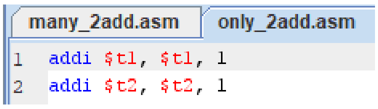


```
CPU_TEST (CPU_TEST.v) (1)
  CPU : mips (mips.v) (3)
    CONTROLLER : control (control.v) (4)
    CONTROLLER_2 : control (control.v) (4)
      CTRL_D : ctrl (ctrl.v)
      CMP_CTRL : cmpdecoder (cmpdecoder.v)
      MDU_CTRL : mdctrl (mdctrl.v)
      HAZARD : hazard (hazard.v) (2)
    DATAPATH : datapath (datapath.v) (19)
      icache : cache2 (cache2.v) (2)
      PCadd4_F : adder (adder.v)
      BTB_FD : BTB (BTB.v)
      RF : RegisterFile (grf.v)
      D_MUX1 : D_MUX (D_MUX.v) (1)
      D_MUX2 : D_MUX (D_MUX.v) (1)
      BrComp : cmp (cmp.v)
      EXT_IMM1 : ext (ext.v)
      EXT_IMM2 : ext (ext.v)
      PCadd4_D : adder (adder.v)
      E_MUX2 : E_MUX (E_MUX.v)
      E_MUX : E_MUX (E_MUX.v)
      ALU : alu (alu.v)
      ALU2 : alu (alu.v)
      ALU_OUT_SEL2 : comms (comms.v)
      ALU_OUT_SEL : comms (comms.v)
      dcache2 : DCACHE (DCACHE.v) (1)
      DM_EXT : dmext (dmext.v)
      PCadd4_W : adder (adder.v)
```

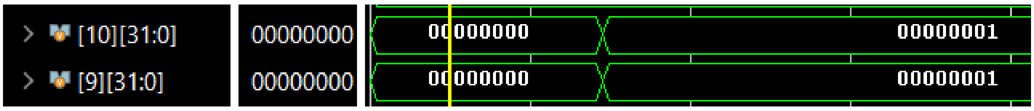
Procedure 2

Here is the two consecutive ADD instructions.

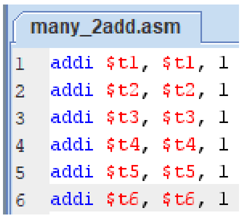
We set predict_PC to PC + 8, so we get the target PC and run the code successfully.



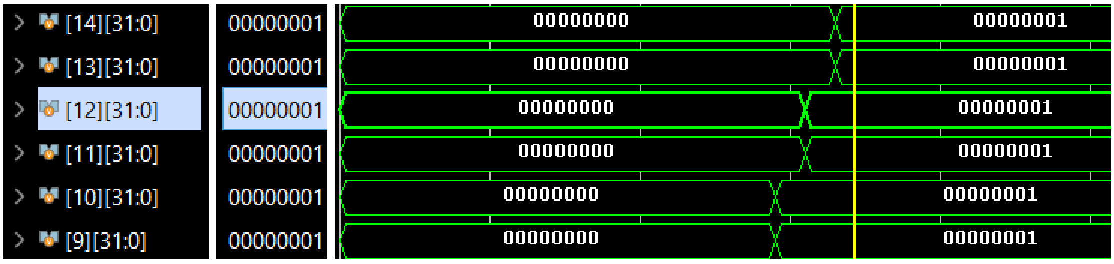
```
addi $t1, $t1, 1
addi $t2, $t2, 1
```



Here is the six consecutive ADD instructions.

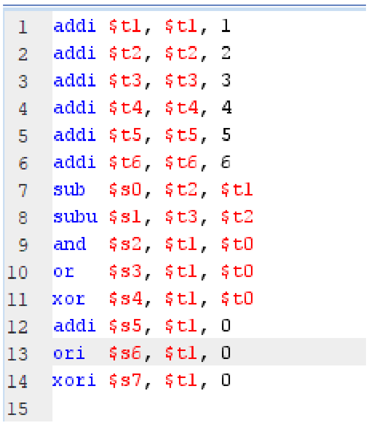


```
addi $t1, $t1, 1
addi $t2, $t2, 1
addi $t3, $t3, 1
addi $t4, $t4, 1
addi $t5, $t5, 1
addi $t6, $t6, 1
```

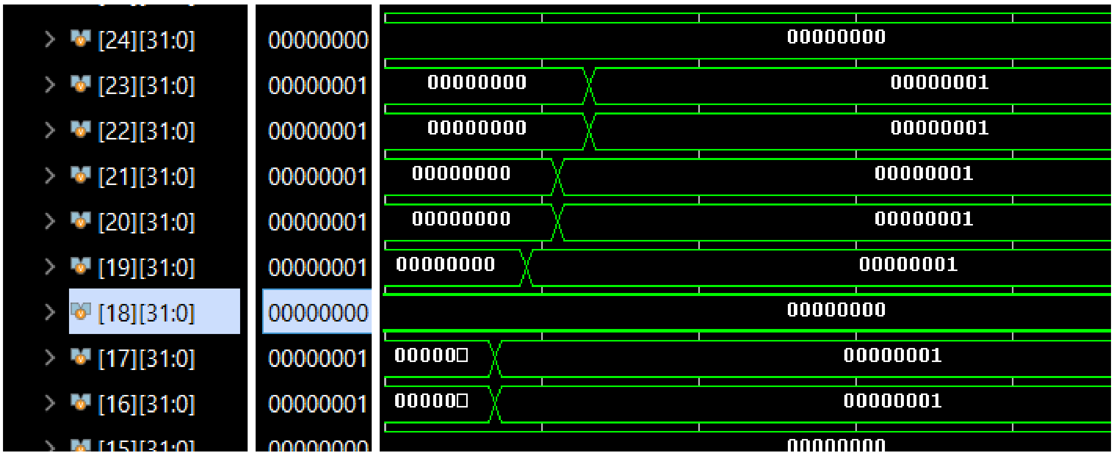


Procedure 3

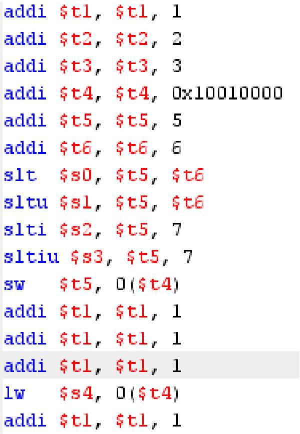
Now we test add, addu, addi, addiu, sub, subu, and, or, xor, xnor, andi, ori, xori.



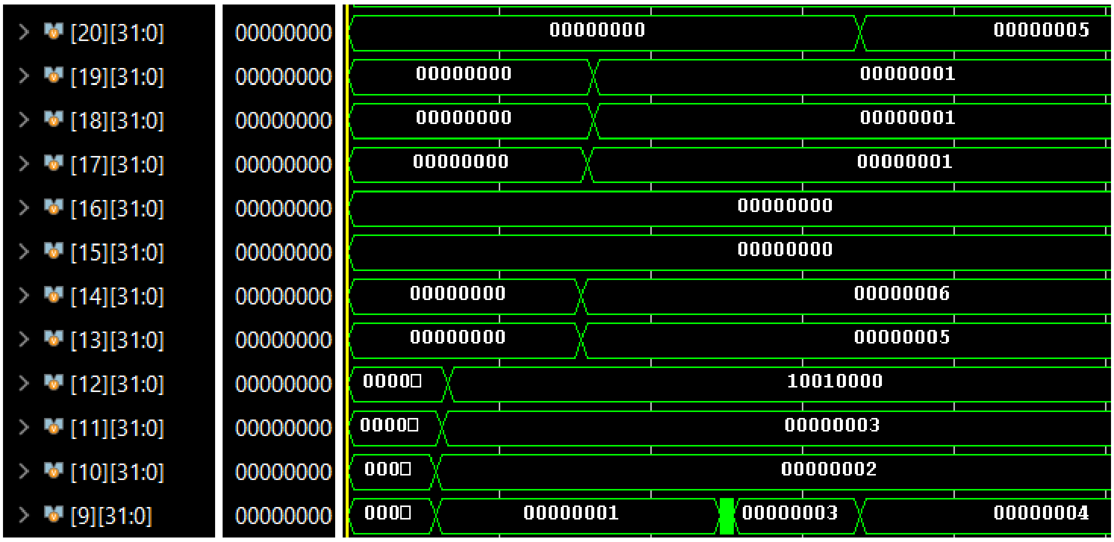
```
addi $t1, $t1, 1
addi $t2, $t2, 2
addi $t3, $t3, 3
addi $t4, $t4, 4
addi $t5, $t5, 5
addi $t6, $t6, 6
sub  $s0, $t2, $t1
subu $s1, $t3, $t2
and  $s2, $t1, $t0
or   $s3, $t1, $t0
xor  $s4, $t1, $t0
addi $s5, $t1, 0
ori  $s6, $t1, 0
xori $s7, $t1, 0

```



And now we test lw and sw. If there are two parallel lw or sw instructions, we will insert one circle stall to change them to sequence. And then, in the write back stage, we will recover to parallel instructions.



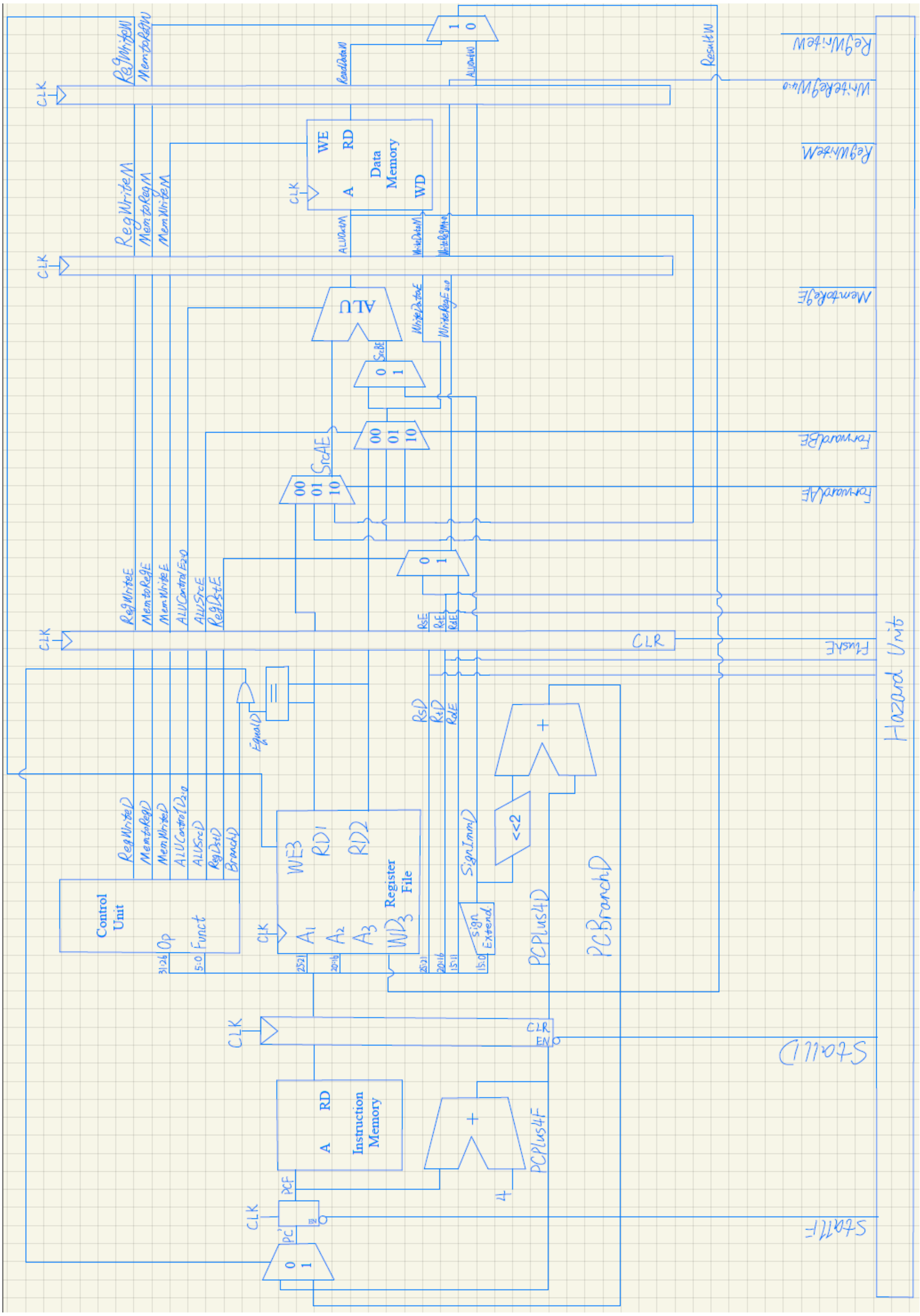
```
addi $t1, $t1, 1
addi $t2, $t2, 2
addi $t3, $t3, 3
addi $t4, $t4, 0x10010000
addi $t5, $t5, 5
addi $t6, $t6, 6
slt  $s0, $t5, $t6
sltu $s1, $t5, $t6
slti $s2, $t5, 7
sltiu $s3, $t5, 7
sw   $t5, 0($t4)
addi $t1, $t1, 1
addi $t1, $t1, 1
addi $t1, $t1, 1
lw   $s4, 0($t4)
addi $t1, $t1, 1
```



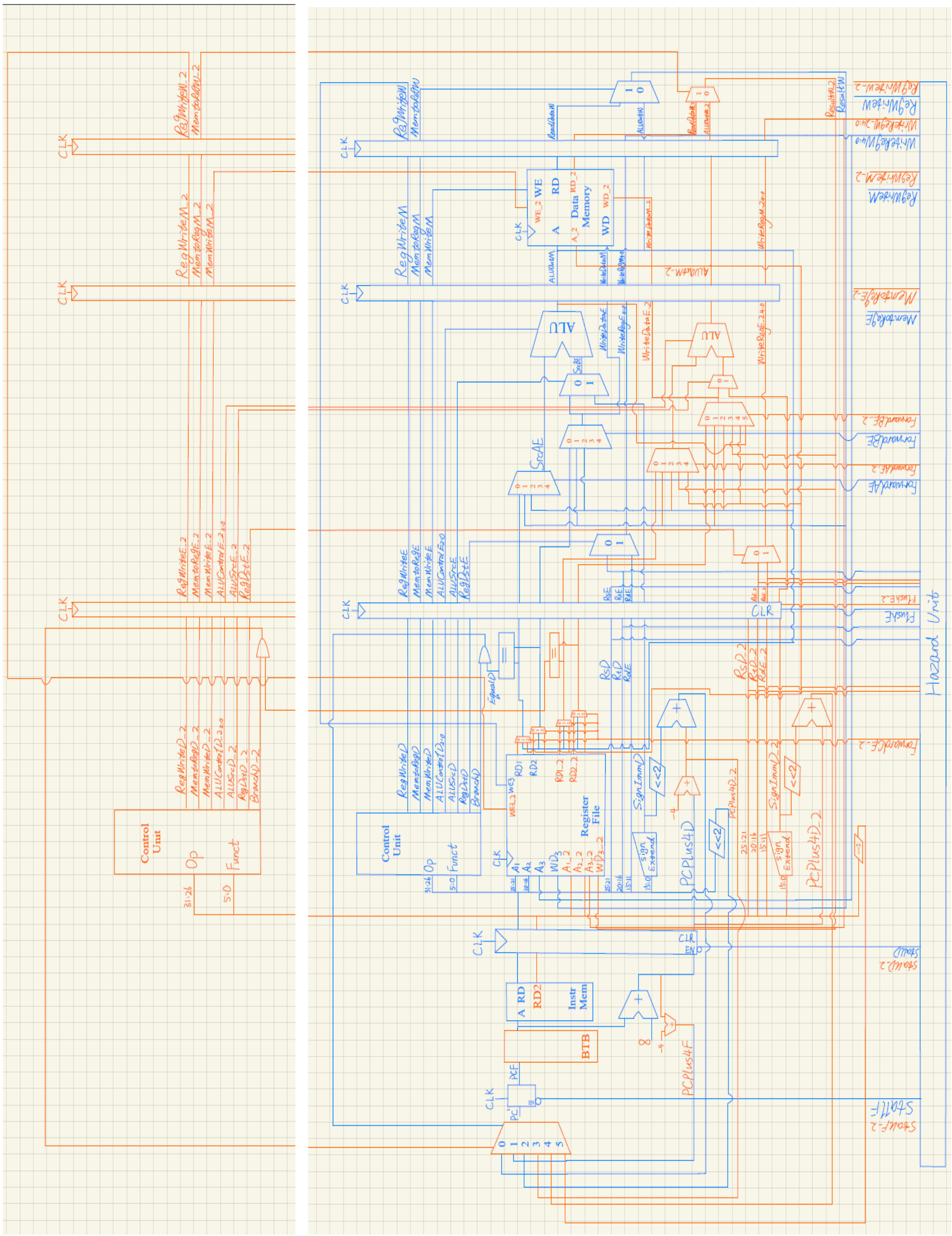
The register file $s4 is number [20] register, here is 5, so it is correct.

Procedure 4

Here is the diagram of the scalar five-stage MIPS pipeline.



Here is the diagram of the Dual-Issue Superscalar MIPS pipeline.



I am not sure if you can see the diagram clearly, I have uploaded the diagram.pdf in the project file.

Procedure 5

Now we are going to consider the hazards. We are assuming that we have six instructions.

1	2
3	4
5	6

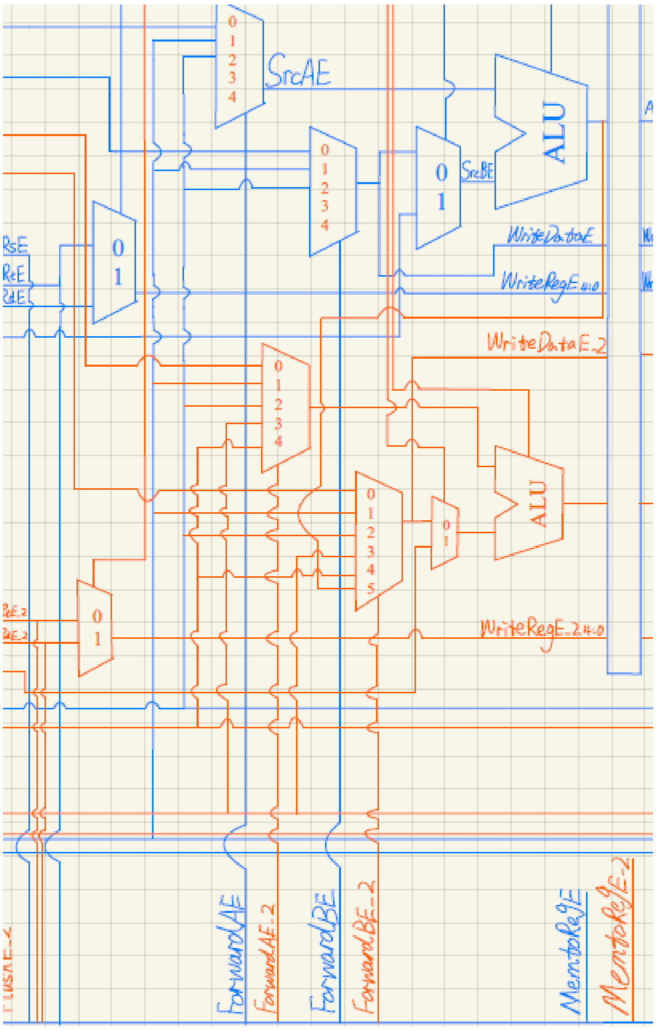
Every time, there will be two parallel instructions in the same stage.

1. 1 is write instruction and 2 is write instruction to the same register.

   The second parallel instruction has high level priority to write register.

2. 1 is write instruction and 2 is read instruction to the same register.

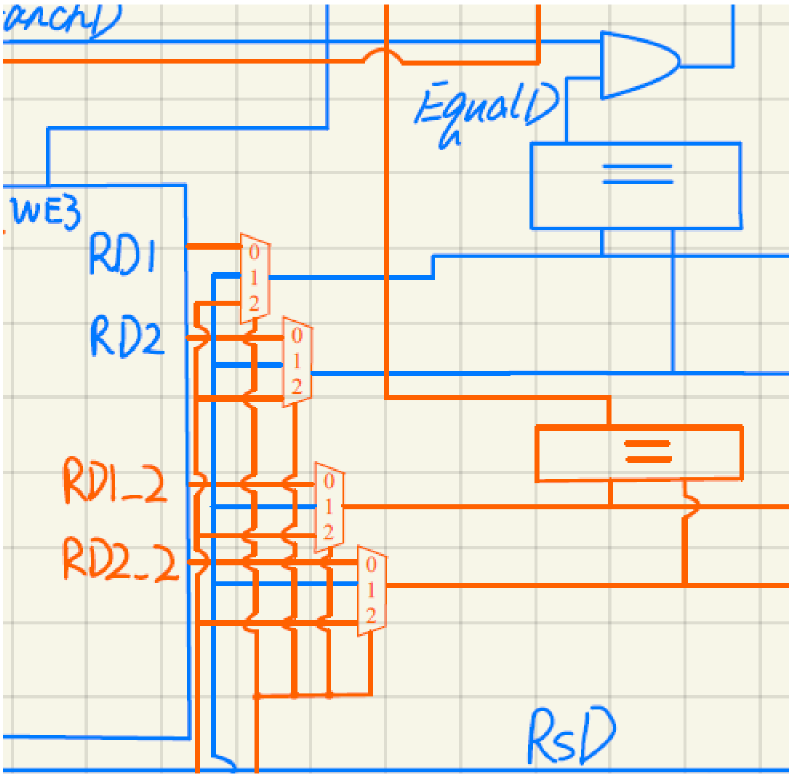
   Here is the situation that the signal ForwardBE_2 will be 5, the result of the above ALU will immediately transfer to the below ALU to calculate.



3. 1 is write instruction and 2 is branch instruction to the same register.

   1 is branch instruction and 2 is write instruction to the same register.

The branch instruction 2 will be stalled two circles and then it can read directly from the result of ALU or ALU_2 in execution stage.



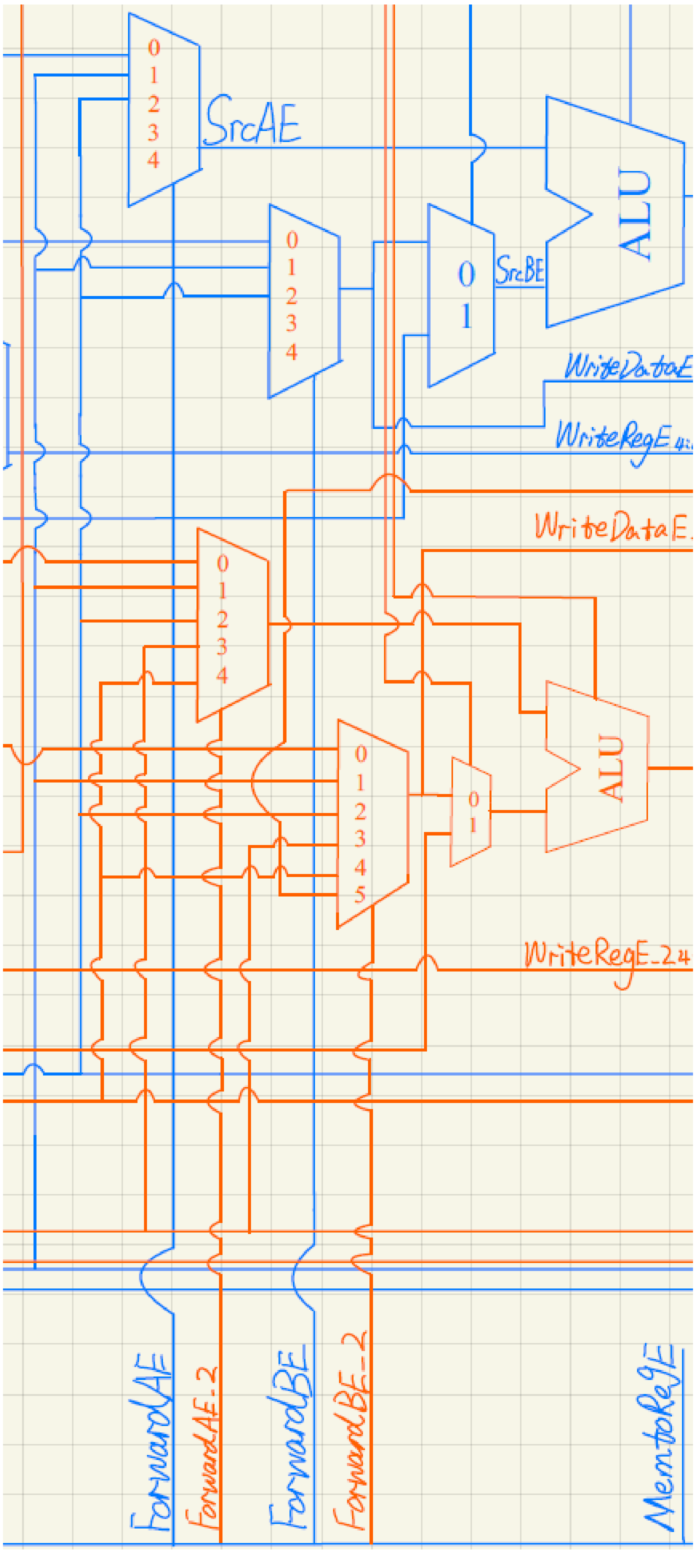
Above all are parallel instructions, below are the different combination.

4. Some instructions are write instructions to the same register file.

The newest write instruction has the highest-level priority to write register.

5. One of the six instructions is write instruction and one of the six instructions is read instruction to the same register and write instruction is always before read instruction.

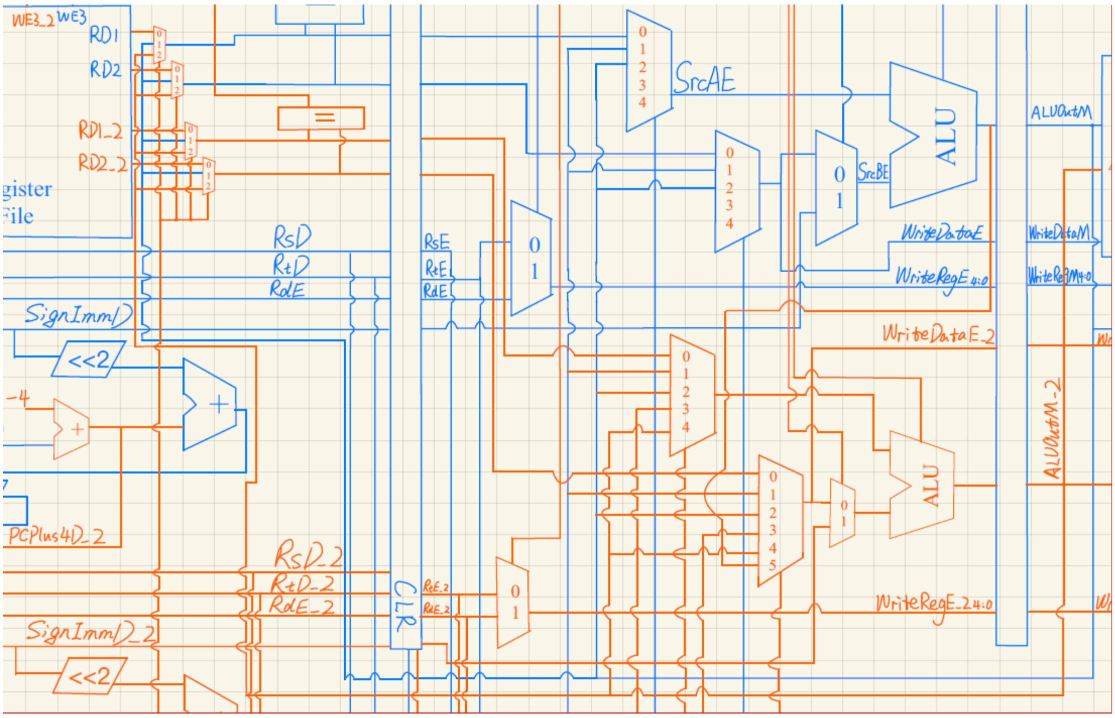
In this situation, we have signal ForwardAE, ForwardAE_2, ForwardBE and ForwardBE_2 to control these four MUXes.



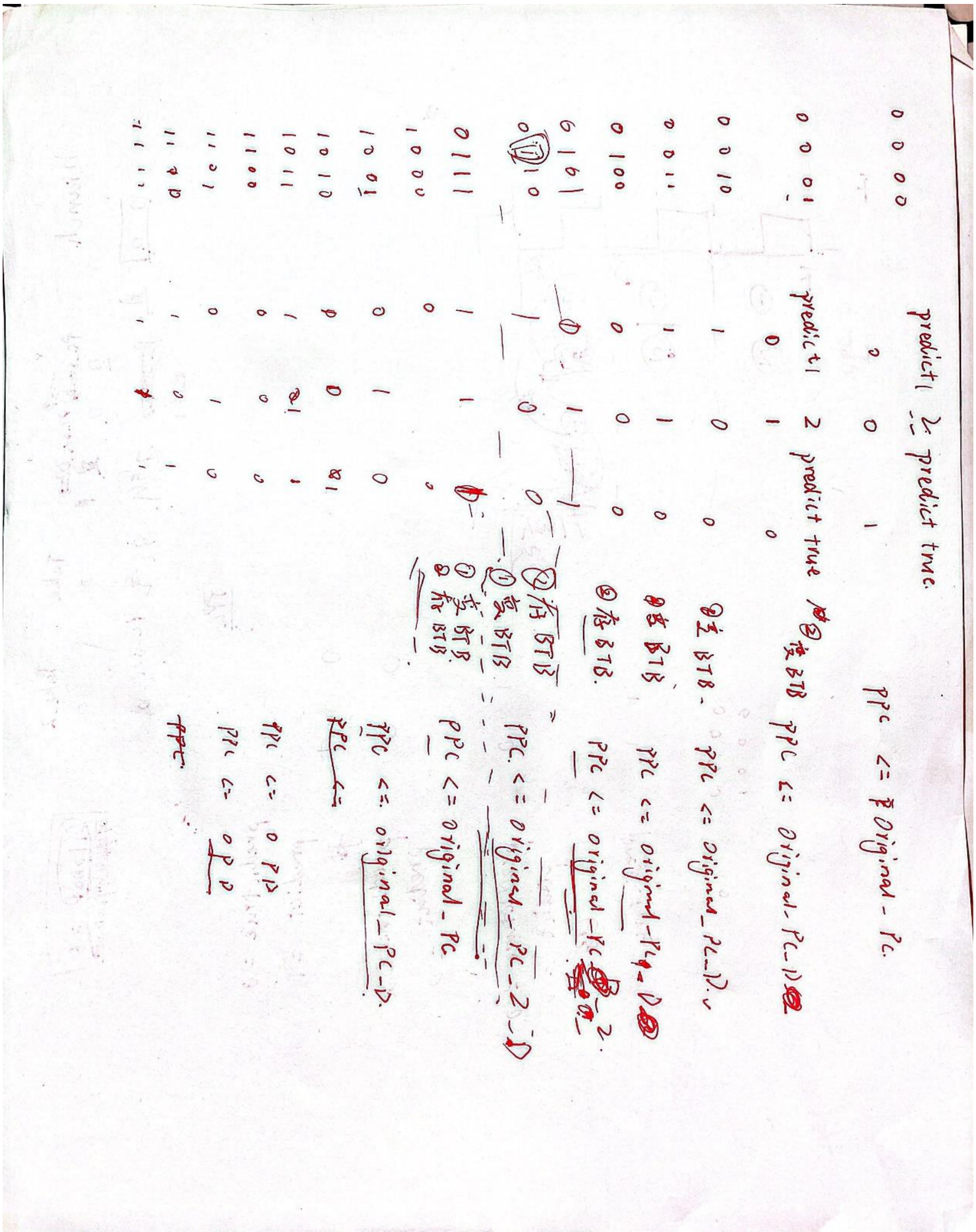
6. One of the six instructions is branch instruction and one of the six instructions is write instruction to the same register.

   In this situation, if the two instructions are parallel, and branch after write, we need to stall branch instruction to wait two circles to get the write result from ALU in execution stage.

   In the other situation, branch instructions need to stall one or two circles to forward the wanted result from ALU or ALU_2 from execution stage.



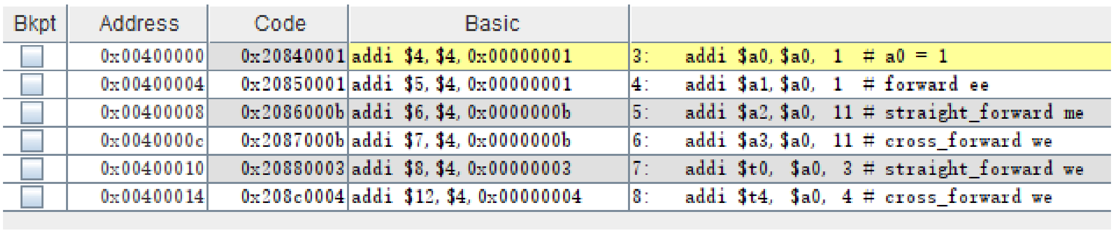
7. For the BTB, here is our solution. According to the value of the two PCsrcd and the value of the two predict signals, we can list a truth table. According to this truth table, we can determine the source of the ppc signal and decide to update the logic of the BTB table.



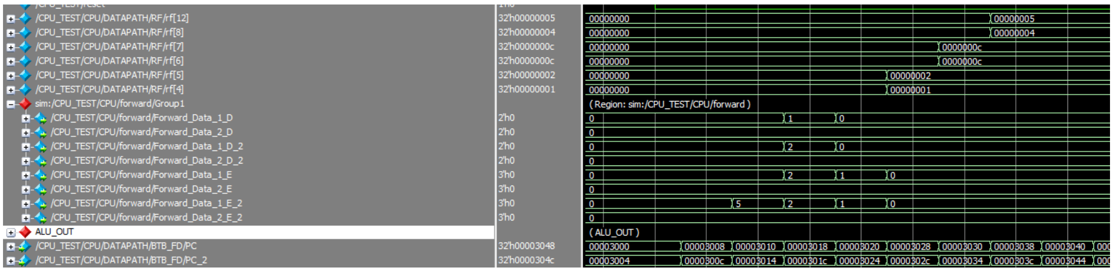
	predict1	2	predict true		
0 0 0 0	0	0	1		PPC <= Original_PC.
0 0 0 1	0	1	0	②变BTB	PPC <= Original_PC_D
0 0 1 0	1	0	0	①变BTB.	PPC <= Original_PC_D.
0 0 1 1	1	1	0	①变BTB	PPC <= original_PC_D
0 1 0 0	0	0	0	②存BTB.	PPC <= original_PC_2.
0 1 0 1	0	1	1		
0 1 1 0	1	0	0	②存BTB ①变BTB	PPC. <= original_PC_2_D
0 1 1 1	1	1	0	①变BTB ②存BTB	PPC <= original_PC
1 0 0 0	0		0		
1 0 0 1	0	1	0		PPC <= original_PC_D.
1 0 1 0	0	0	1		~~PPC <=~~
1 0 1 1	1	1	1		
1 1 0 0	0	0	0		PPC <= oPP
1 1 0 1	0	1	0		PPC <= oPP
1 1 1 0	1	0	1		~~PPC~~
1 1 1 1	1	1	1		

Test

The forward is assigned correctly, and the result of register file is correct.

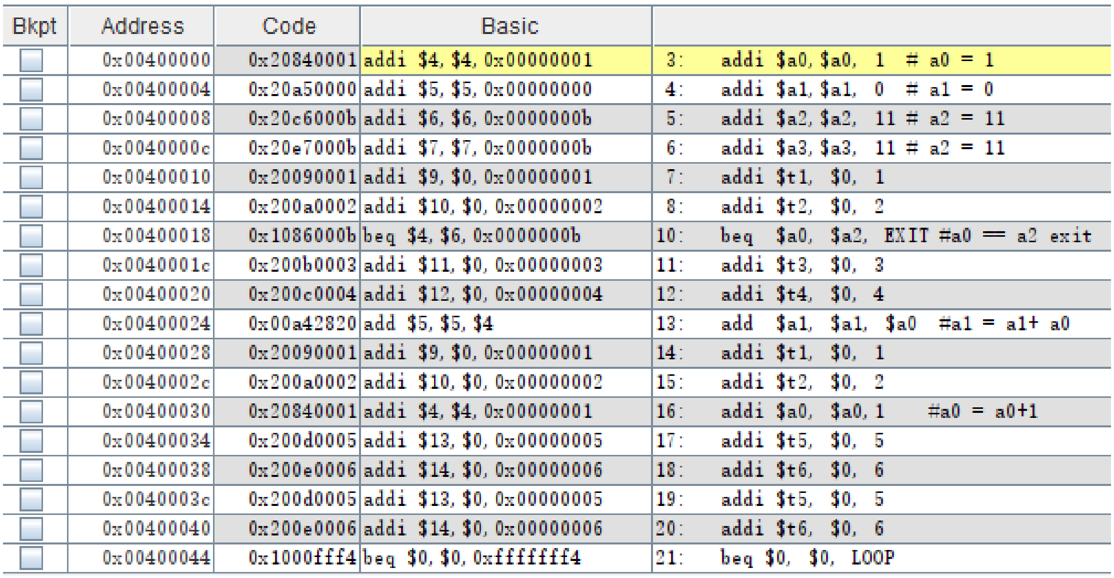


Bkpt	Address	Code	Basic	
□	0x00400000	0x20840001	addi $4, $4, 0x00000001	3: addi $a0, $a0, 1 # a0 = 1
□	0x00400004	0x20850001	addi $5, $4, 0x00000001	4: addi $a1, $a0, 1 # forward ee
□	0x00400008	0x2086000b	addi $6, $4, 0x0000000b	5: addi $a2, $a0, 11 # straight_forward me
□	0x0040000c	0x2087000b	addi $7, $4, 0x0000000b	6: addi $a3, $a0, 11 # cross_forward we
□	0x00400010	0x20880003	addi $8, $4, 0x00000003	7: addi $t0, $a0, 3 # straight_forward we
□	0x00400014	0x208c0004	addi $12, $4, 0x00000004	8: addi $t4, $a0, 4 # cross_forward we

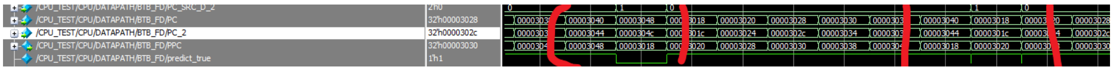


/CPU_TEST/CPU/DATAPATH/RF/rf[12]	32'h00000005
/CPU_TEST/CPU/DATAPATH/RF/rf[8]	32'h00000004
/CPU_TEST/CPU/DATAPATH/RF/rf[7]	32'h0000000c
/CPU_TEST/CPU/DATAPATH/RF/rf[6]	32'h0000000c
/CPU_TEST/CPU/DATAPATH/RF/rf[5]	32'h00000002
/CPU_TEST/CPU/DATAPATH/RF/rf[4]	32'h00000001
sim:/CPU_TEST/CPU/forward/Group1	
/CPU_TEST/CPU/forward/Forward_Data_1_D	2'h0
/CPU_TEST/CPU/forward/Forward_Data_2_D	2'h0
/CPU_TEST/CPU/forward/Forward_Data_1_D_2	2'h0
/CPU_TEST/CPU/forward/Forward_Data_2_D_2	2'h0
/CPU_TEST/CPU/forward/Forward_Data_1_E	3'h0
/CPU_TEST/CPU/forward/Forward_Data_2_E	3'h0
/CPU_TEST/CPU/forward/Forward_Data_1_E_2	3'h0
/CPU_TEST/CPU/forward/Forward_Data_2_E_2	3'h0
ALU_OUT	
/CPU_TEST/CPU/DATAPATH/BTB_FD/PC	32'h00003048
/CPU_TEST/CPU/DATAPATH/BTB_FD/PC_2	32'h0000304c

Implement BTB predictions



Bkpt	Address	Code	Basic	
□	0x00400000	0x20840001	addi $4, $4, 0x00000001	3: addi $a0, $a0, 1 # a0 = 1
□	0x00400004	0x20a50000	addi $5, $5, 0x00000000	4: addi $a1, $a1, 0 # a1 = 0
□	0x00400008	0x20c6000b	addi $6, $6, 0x0000000b	5: addi $a2, $a2, 11 # a2 = 11
□	0x0040000c	0x20e7000b	addi $7, $7, 0x0000000b	6: addi $a3, $a3, 11 # a2 = 11
□	0x00400010	0x20090001	addi $9, $0, 0x00000001	7: addi $t1, $0, 1
□	0x00400014	0x200a0002	addi $10, $0, 0x00000002	8: addi $t2, $0, 2
□	0x00400018	0x1086000b	beq $4, $6, 0x0000000b	10: beq $a0, $a2, EXIT #a0 == a2 exit
□	0x0040001c	0x200b0003	addi $11, $0, 0x00000003	11: addi $t3, $0, 3
□	0x00400020	0x200c0004	addi $12, $0, 0x00000004	12: addi $t4, $0, 4
□	0x00400024	0x00a42820	add $5, $5, $4	13: add $a1, $a1, $a0 #a1 = a1+ a0
□	0x00400028	0x20090001	addi $9, $0, 0x00000001	14: addi $t1, $0, 1
□	0x0040002c	0x200a0002	addi $10, $0, 0x00000002	15: addi $t2, $0, 2
□	0x00400030	0x20840001	addi $4, $4, 0x00000001	16: addi $a0, $a0, 1 #a0 = a0+1
□	0x00400034	0x200d0005	addi $13, $0, 0x00000005	17: addi $t5, $0, 5
□	0x00400038	0x200e0006	addi $14, $0, 0x00000006	18: addi $t6, $0, 6
□	0x0040003c	0x200d0005	addi $13, $0, 0x00000005	19: addi $t5, $0, 5
□	0x00400040	0x200e0006	addi $14, $0, 0x00000006	20: addi $t6, $0, 6
□	0x00400044	0x1000fff4	beq $0, $0, 0xfffffff4	21: beq $0, $0, LOOP



/CPU_TEST/CPU/DATAPATH/BTB_FD/PC_SRC_D_2	2'h0
/CPU_TEST/CPU/DATAPATH/BTB_FD/PC	32'h00003028
/CPU_TEST/CPU/DATAPATH/BTB_FD/PC_2	32'h0000302c
/CPU_TEST/CPU/DATAPATH/BTB_FD/PPC	32'h00003030
/CPU_TEST/CPU/DATAPATH/BTB_FD/predict_true	1'h1

To be improved

We still have several shortages in our design. In some situation we can't successfully run the code without insert several unrelated codes. In our consideration, maybe the reason is that we don't set the stall or the forward signal in the correct cycle, but we don't have enough time to fix such problem, and in most cases, we can run the code and obviously observe the performance increases.

Thank you so much for our TA George. He gave us so much help and time to finish our project step by step. Although we have take kind of same class in our university, but in 154B we still learned a lot and have a deep understanding of mips. It is really fortune for me and Can to take this class, Thanks again to professor Strukov and TA George!