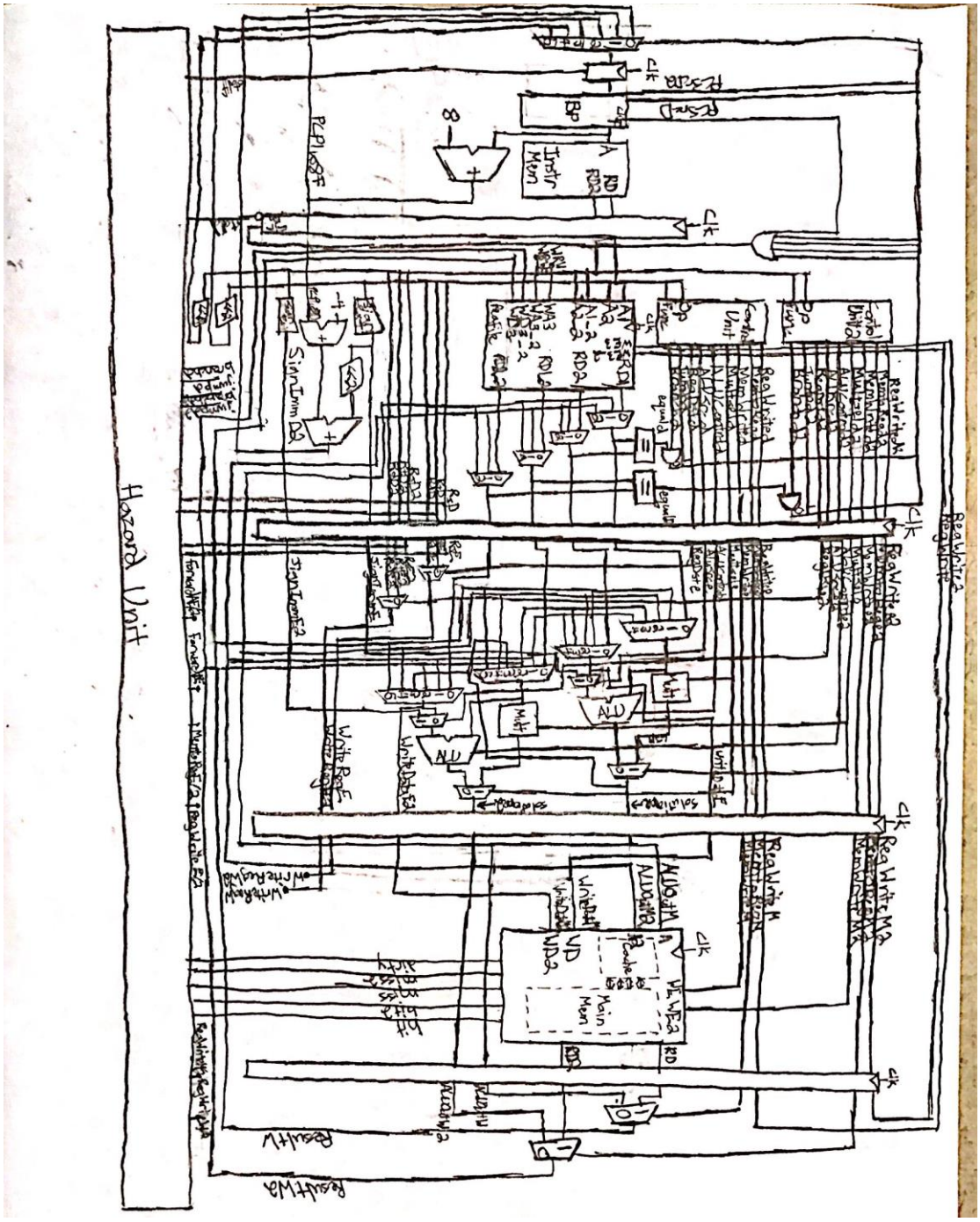
Final Report by Dylan and Yiming

Design Methodology



The whole processor we designed are shown above. We describe several parts here.

We began our superscalar processor by first perfecting our pipeline processor with our branch predictor and the cache-main memory system. We also made sure that our mult instruction implemented as we intended in to with proper signals before moving it to the superscalar system.

After verifying that all the signals worked as intended with proper hazard handling, we were able to contemplate what the entire superscalar processor would need to handle each instruction.

After much debating we decided every wire would need to double, the first set to handle the first instruction and the second to handle the second instruction. Since the instructions must come from a single instruction memory bank, we could not double every module. Rather, the branch predictor, instruction memory, register file, hazard unit, and memory system all needed to be updated to pass in with two lines of data instead of having two independent sets. We began by entering each module and doubling each wire for two sets of data to relay between each process, the only difference is that the program counter counts by eight rather than four, so we did not need to double this wire to perform these actions.

The first unit we updated was the branch predictor in which we needed to take in two branch data sets and store this information instead. For now, what we did is that we take care well of the first instruction. The only situation we can't deal with is the first predict is not taken and the second one is taken. That is a design flaw of our branch predictor.

One of the hardest parts of the lab was dealing with the data forwarding. Data forwarding occurred in two locations: one in the branch determination from the decode stage and the other from the forwarding in the execute stage. The execute stage needs to forward a total of five ways for the first instruction and six ways for the second instruction. The first instruction needs to forward from the memory and write-back stage looking both the first and second way instructions. The second instruction needs to do the same but also needs to forward from the first instruction's solution also in the execute stage if needed. Branch also needed forwarding as well. However, branch forward is difference because the value first needs to be executed then can be forwarded. Therefore, we needed to stall the processor for a cycle and allow an instruction to be executed, then forward from either the memory stage or the write-back stage. The part of code is showed below(wrote in hazard controller):

```
// forwarding sources to D stage (branch equality)

   always @( rsd, rsd2, rtd, rtd2, writeregm, regwritem, writeregm2, regwritem2 ) begin

      forwardad <= 2'b00; forwardbd <= 2'b00;

      forwardad2 <= 2'b00; forwardbd2 <= 2'b00;



      if (rsd !=0 & (rsd == writeregm) & regwritem)            // 1-->5

         forwardad <= 2'b01;

      else if (rtd !=0 & (rtd == writeregm) & regwritem)            // 1-->5

         forwardbd <= 2'b01;
```

```
    if (rsd !=0 & (rsd == writeregm2) & regwritem2)         // 1-->6
       forwardad <= 2'b10;
    else if (rtd !=0 & (rtd == writeregm2) & regwritem2)   // 1-->6
       forwardbd <= 2'b10;

    if (rsd2 !=0 & (rsd2 == writeregm) & regwritem)         // 2-->5
       forwardad2 <= 2'b10;
    else if (rtd2 !=0 & (rtd2 == writeregm) & regwritem)   // 2-->5
       forwardbd2 <= 2'b10;
    if (rsd2 !=0 & (rsd2 == writeregm2) & regwritem2)             // 2-->6
       forwardad2 <= 2'b01;
    else if (rtd2 !=0 & (rtd2 == writeregm2) & regwritem2)       // 2-->6
       forwardbd2 <= 2'b01;
  end

// forwarding sources to E stage (ALU)
always @( rse, rse2, rte, rte2, writeregm, regwritem, writeregw, regwritew, regwritee ) begin
      forwardae <= 2'b00; forwardbe <= 2'b00;
      forwardae2 <= 2'b00; forwardbe2 <= 2'b00;
      //if (rse != 0) begin
         if ((rse == writeregm2) & regwritem2)           // 4-->1
            forwardae    <= 3'b011;
         else if ((rse == writeregm)    & regwritem)     // 3-->1
            forwardae    <= 3'b010;
```

```
    else if ((rse == writeregw2) & regwritew2)     // 6-->1
        forwardae    <= 3'b100;
    else if ((rse == writeregw)    & regwritew)     // 5-->1
        forwardae    <= 3'b001;
    if ((rse2 == writerege) & regwritee)        // 1-->2
        forwardae2 <= 3'b011;
    else if ((rse2 == writeregm2) & regwritem2)   // 4-->2
        forwardae2 <= 3'b010;
    else if ((rse2 == writeregm)    & regwritem)    // 3-->2
        forwardae2    <= 3'b100;
    else if ((rse2 == writeregw2) & regwritew2)   // 6-->2
        forwardae2 <= 3'b001;
    else if ((rse2 == writeregw)    & regwritew)    // 5-->2
        forwardae2    <= 3'b101;
//end
//if (rte != 0) begin
    if ((rte == writeregm2) & regwritem2)           // 4-->1
        forwardbe    <= 3'b011;
    else if ((rte == writeregm)    & regwritem)     // 3-->1
        forwardbe    <= 3'b010;
    else if ((rte == writeregw2) & regwritew2)     // 6-->1
        forwardbe    <= 3'b100;
    else if ((rte == writeregw)    & regwritew)     // 5-->1
        forwardbe    <= 3'b001;
```

```
        if ((rte2 == writerege) & regwritee)        // 1-->2
            forwardbe2 <= 3'b011;
        else if ((rte2 == writeregm2) & regwritem2)   // 4-->2
            forwardbe2 <= 3'b010;
        else if ((rte2 == writeregm)    & regwritem)    // 3-->2
            forwardbe2    <= 3'b100;
        else if ((rte2 == writeregw2) & regwritew2)   // 6-->2
            forwardbe2 <= 3'b001;
        else if ((rte2 == writeregw)    & regwritew)    // 5-->2
            forwardbe2    <= 3'b101;
    //end
end
```

After solving all data forwarding cases, we resumed by updating the memory system to deal with the load word and store word instruction from both the first and second lines. A new hazard needed to be address if there was ever a load word and store word occurring parallel with each other. In this case, we stall the processor for the second line and execute one instruction line at a time. This saves us data write hazards that occur in superscalar processors like ours. We also modified the cache and main memory part. The cache has two input address and two input data now, it also has two read data which correspond to the two read instructions.

Once completing these tasks, we created tests to deal with our hazards and forwarding. These programs are described later below but include handling of forwarding, parallel store and load word, multiply instructions, and, most importantly, a mix of various R-type, I-type, taken and not taken branches, SW, and LW instructions. Overall, the goal of creating a superscalar processor was achieved and the results are described below.

Test Cases

We designed 3 tests to show how our processor works. The first one is designed for the original program showed below:

Cycle	reset	pc	Instr	branch	srca	srcb	aluresult	zero	pcsrc	Writedata	memwrite	read data
1	1	00	addi $2, $0, 5 20020005	0	0	5	5	0	0	X	0	X
2	0	04	addi $3, $0, 12 2003000c	0	0	C	C	0	0	X	0	X
3	0	08	addi $7, $3, -9 20067fff7	0	C	-9	3	0	0	X	0	X
4	0	0C	or $4, $7, $2 00e22025	0	3	5	7	0	0	X	0	X
5	0	10	and $5, $3, $4 00642824	0	C	7	4	0	0	X	0	X
6	0	14	add $5, $5, $4 00a42820	0	4	7	B	0	0	X	0	X
7	0	18	beq $5, $7, end 10a7000a	1	B	3	8	0	0	X	0	X
8	0	1C	slt $4, $3, $4 0064202a	0	C	7	0	1	0	X	0	X
9	0	20	beq $4, $0, around 10800001	1	0	0	0	1	1	X	0	X
10	0	24	slt $4, $7, $2 00e2202a	0	3	5	1	0	0	X	0	X
11	0	28	add $7, $4, $5 00853820	0	1	B	C	0	0	X	0	X
12	0	2C	sub $7, $7, $2 00e23822	0	C	5	7	0	0	X	0	X
13	0	30	sw $7, 68($3) ac670044	0	C	44	50	0	0	7	1	X
14	0	34	lw $2, 80($0) 8c020050	0	0	50	50	0	0	X	0	7
15	0	38	j end 08000011	1	0	0	0	1	1	X	0	X
16	0	3C	sw $2, 84($0) ac020054	0	0	54	54	0	0	7	1	X

The purpose for this program is to write 7 to memory and read the same value from the same position. The wave form is showed below: the instrd and instrd2 is parallelized and pcd is the pc counter we used. Srcae and srcbe are two input of the first ALU and srcae2, srcbe2 are used for the second ALU. Solutione and solutione2 is the output of the ALU. Every forward signal is shown in the waveform. Figure 1 shows the waveform from 1 to 100ns, figure 2 shows the waveform from 100ns to 180ns and figure 3 shows the waveform from 180ns to 205ns.

Figure1:

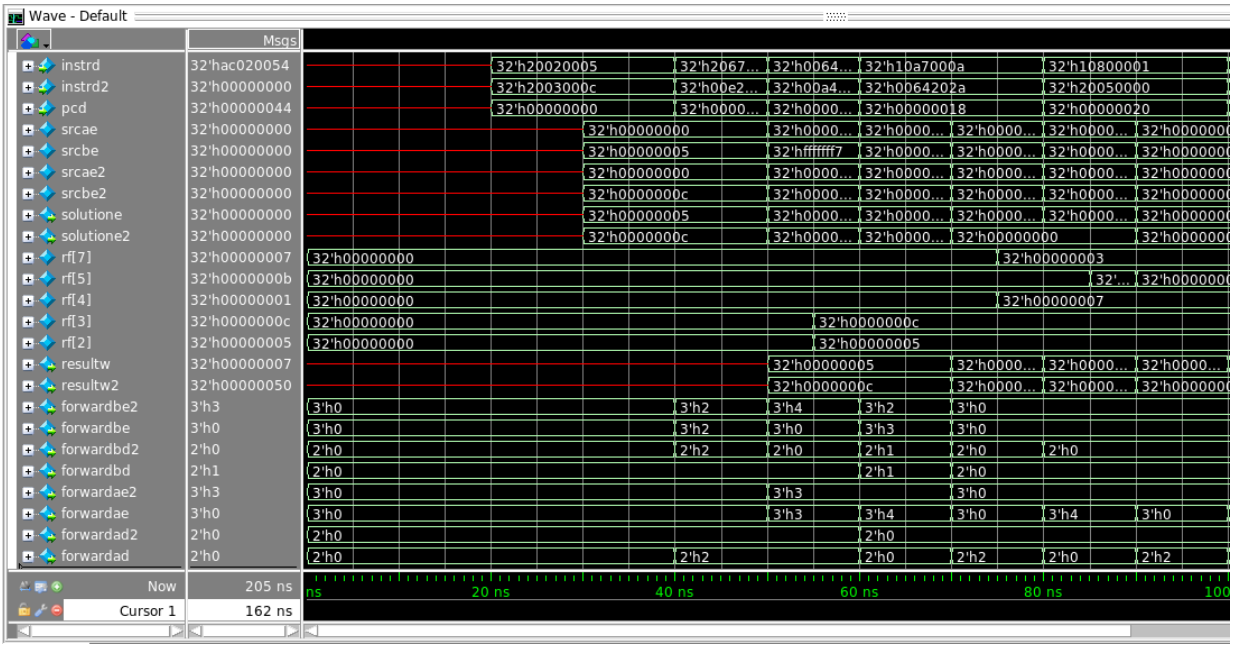


Figure 2:

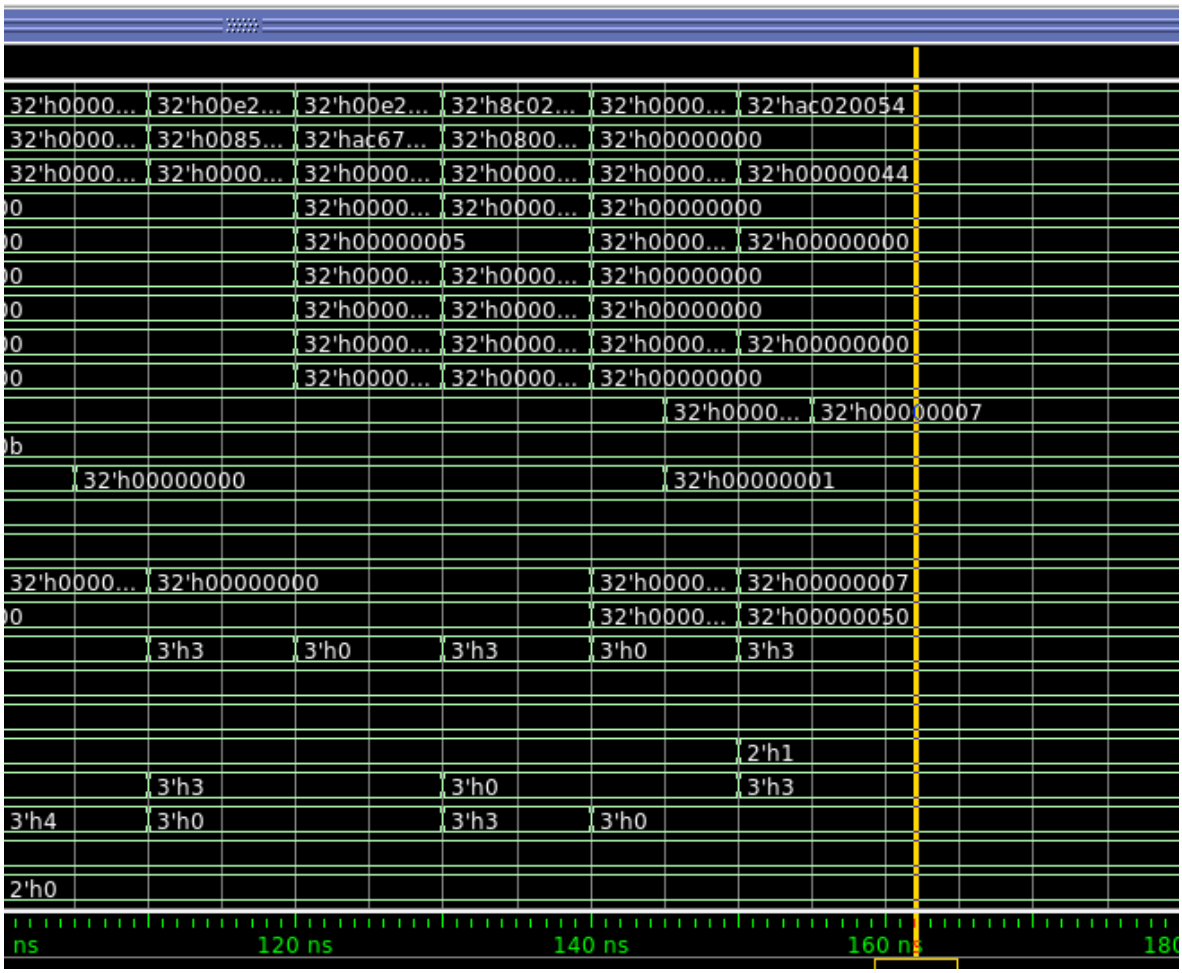
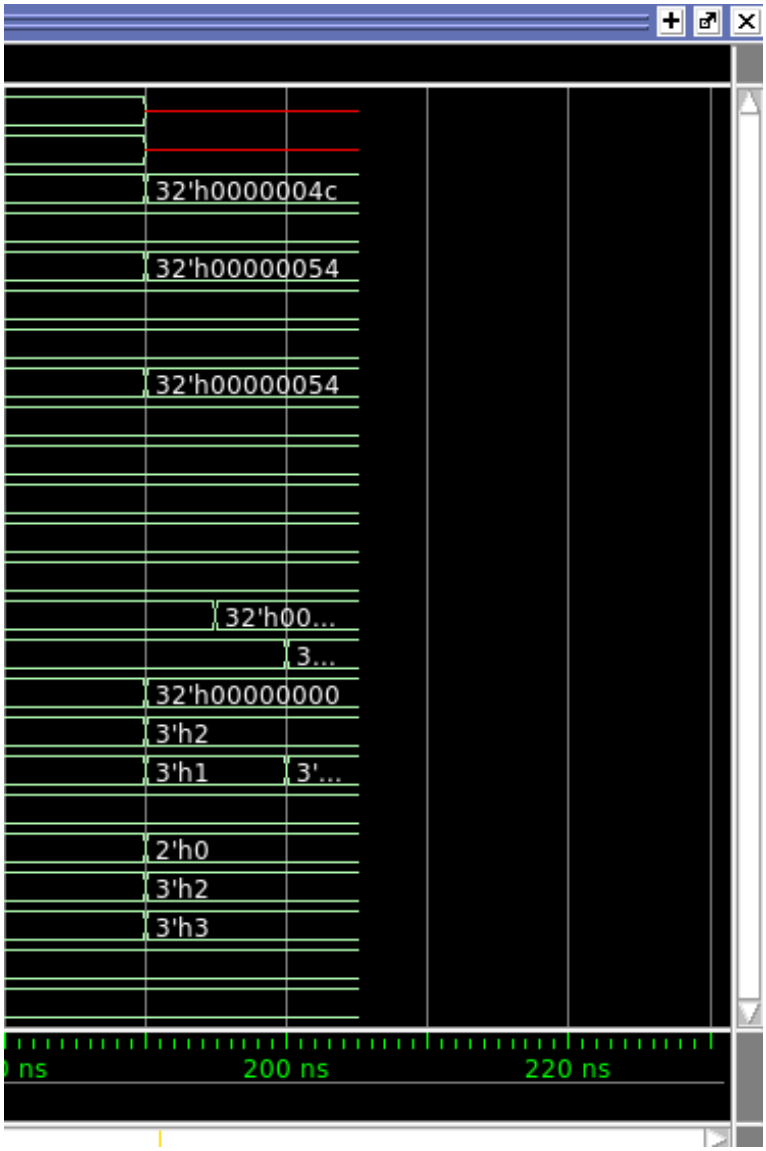


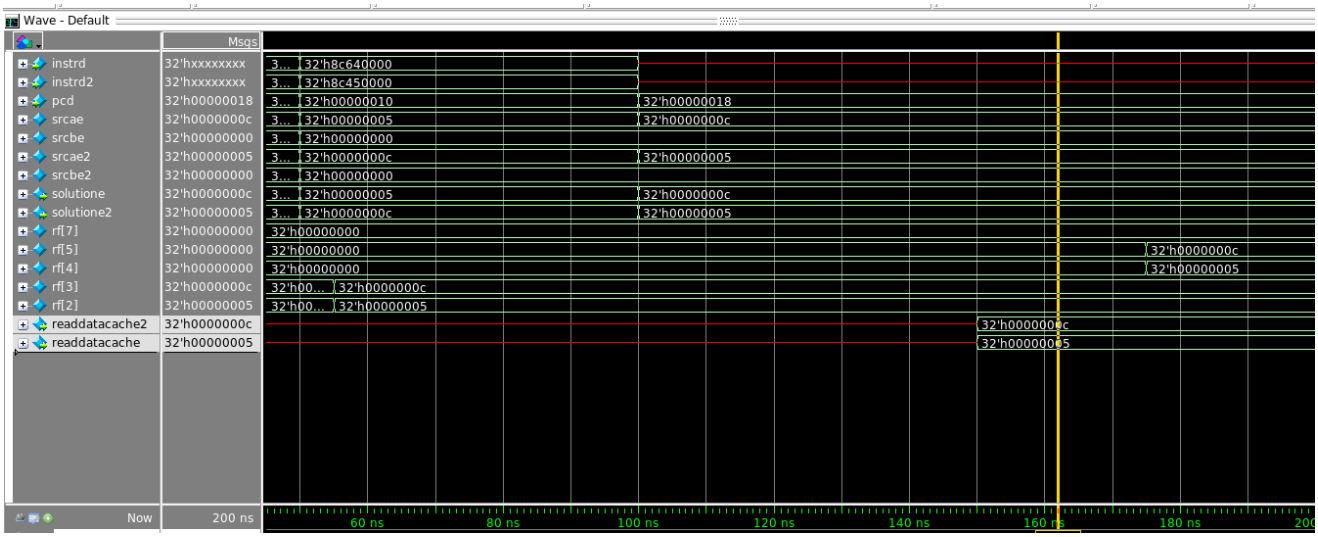
Figure 3:



For the second test, we designed a test for parallel memory(cache) access. The test we designed is showed below:

Hex	Instruction	Interpretation
20020005	addi $2 $0 5	$2=5
2003000c	addi $3 $0 12	$3=12
ac430000	sw $3 0($2)	[5]=12
ac620000	sw $2 0($3)	[12]=5
8c640000	lw $4 0($3)	$4=5
8c450000	lw $5 0($2)	$5=12

In this test, we designed two parallel store word and two parallel load word. After doing this, we can read two values from the cache. The figure is showed below:

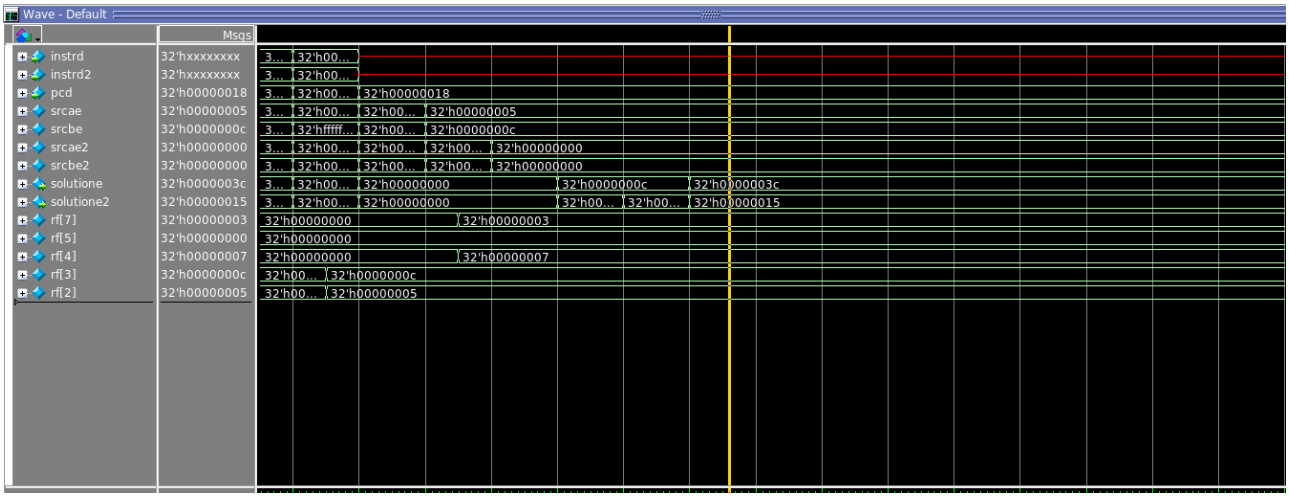


As it is shown in the figure above, the last two lines in the waveform show the read data from the cache which are the same as we wrote in.

For the third test, we designed a parallel multiplication to show our multiplier works. The instructions we use are shown below:

Hex	Instructions	Interpretation
20020005	addi $2 $0 5	$2=5
2003000c	addi $3 $0 12	$3=12
2067fff7	addi $7 $3 -9	$7=3
00e22025	or $4 $7 $2	$4=7
00430018	mult $2 $3	5*12=60->0x3c
00870018	mult $4 $7	7*3=21

Figure:



Conclusion

It took us about 36 hours to design the whole processor and the tests. We designed a really fancy processor. The biggest problem for us is the data hazard. We figure out that we will need to settle 9 forward down. We finally designed 3 tests to show the result of our processor.