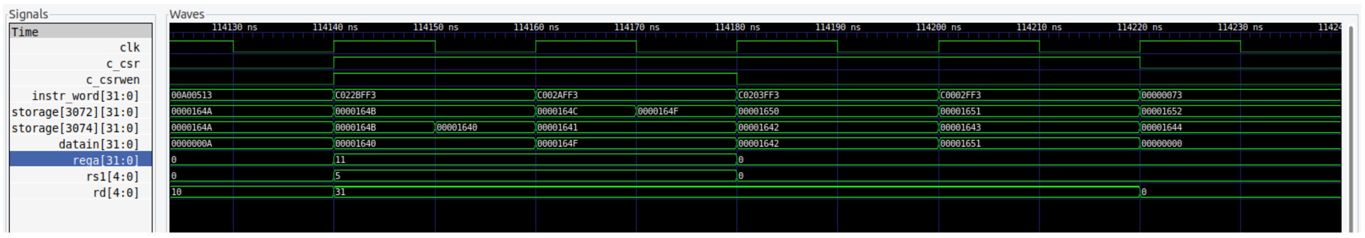
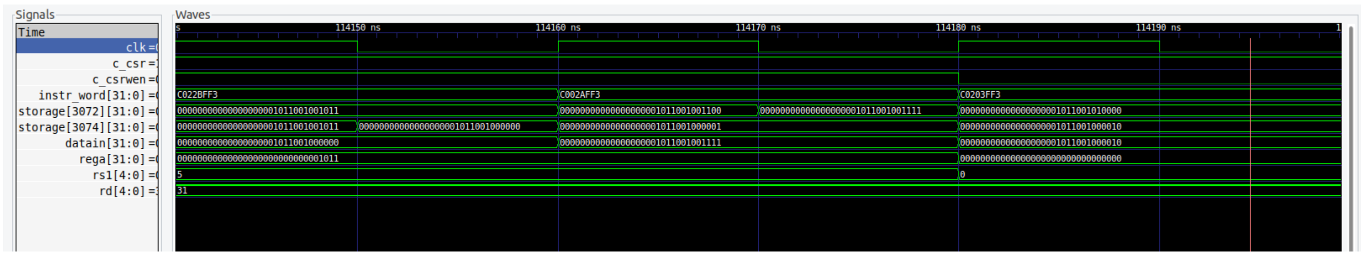
Testing CSR instructions - test_01

Note that the risc-v code is really long since it consists of the initial fibonacci series and 4 instructions added inside, I found it more convenient to refer to it in the test_01.s file

- Refer to the files in test_01 for further details
- I also added the ghw file for the below wave entiteled waveCSR0.ghw

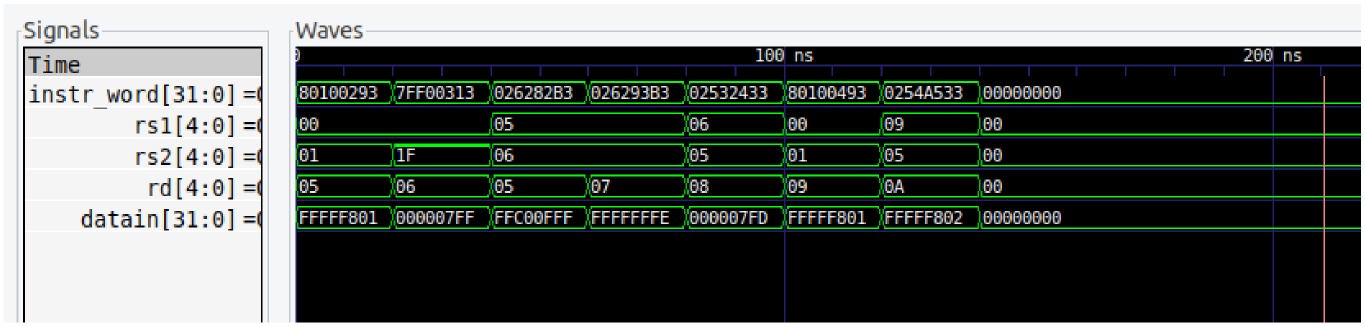




We can see in the image above that the 4 instructions get executed towards the end. We can check the correctness of the implementation through the following:

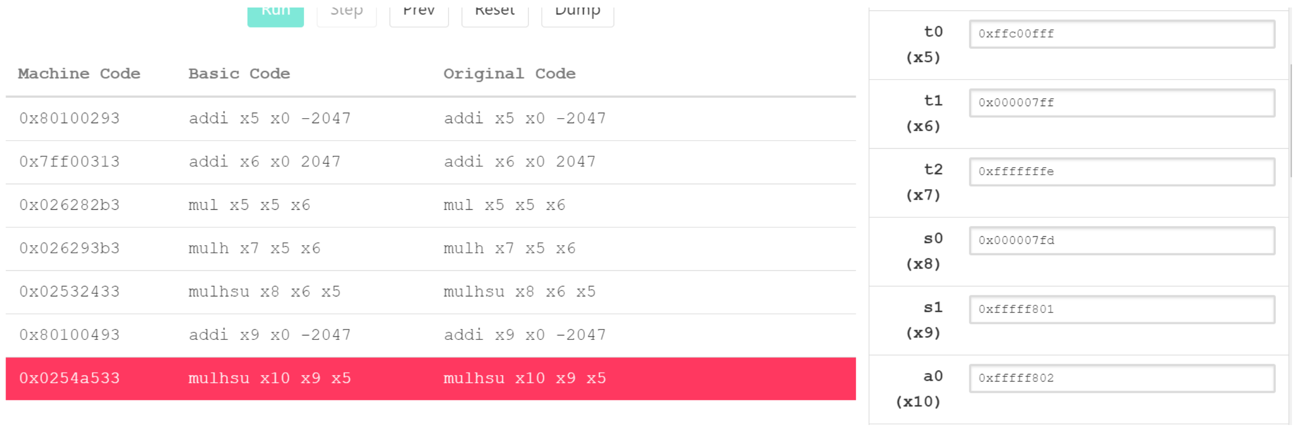
- The control signal that flags the CSR instructions got set to '1' for all 4 instructions
- The control signal controling the write back to the CSRfile got set to '1' for the first 2 instructions only
- Register rs1=5 during both instructions
- We can read the instruction machine code in the instr_word signal
- regA which is the value stored in rs1 and the mask that will be used is 1011 (which is 11 i decimal)
- For the CSRRC we can notice that the last 4 bits are 1011, but on the falling edge the correct bits got cleared and the last 4 bits became 0000
- For the CSRRS we can notice that the last 4 bits were 1100, but on the falling edge of the clock the correct bits got set and the last 4 bits became 1111
- We can also check that rd =31 and that the value being sent back to get stored in rd through datain is the correct, modified one for all 4 instructions

Testing the Multiply Instructions - test_02



```
Signals              Waves
Time                 0                        100 ns                        200 ns
instr_word[31:0] =   80100293 7FF00313 026282B3 026293B3 02532433 80100493 0254A533 00000000
rs1[4:0] =           00                05                06       00       09       00
rs2[4:0] =           01       1F       06                05       01       05       00
rd[4:0] =            05       06       05       07       08       09       0A       00
datain[31:0] =       FFFFF801 000007FF FFC00FFF FFFFFFFE 000007FD FFFFF801 FFFFF802 00000000
```

- We note that datain is the value being written back to the register file, ie the value being stored in rd
- We can test the correctness of the code by comparing it to the values obtained in ***venus*** simulator.



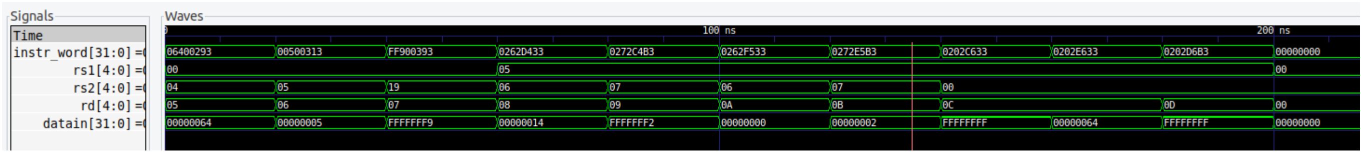
Machine Code	Basic Code	Original Code
0x80100293	addi x5 x0 -2047	addi x5 x0 -2047
0x7ff00313	addi x6 x0 2047	addi x6 x0 2047
0x026282b3	mul x5 x5 x6	mul x5 x5 x6
0x026293b3	mulh x7 x5 x6	mulh x7 x5 x6
0x02532433	mulhsu x8 x6 x5	mulhsu x8 x6 x5
0x80100493	addi x9 x0 -2047	addi x9 x0 -2047
0x0254a533	mulhsu x10 x9 x5	mulhsu x10 x9 x5

t0 (x5)	0xffc00fff
t1 (x6)	0x000007ff
t2 (x7)	0xfffffffe
s0 (x8)	0x000007fd
s1 (x9)	0xfffff801
a0 (x10)	0xfffff802

- We multiplied an unsigned number (stored in x6) by a negative signed number (stored in x5) and obtained a negative signed value (stored in x7) to test the mulhsu
- We multiplied an unsigned number (stored in x5) by a negative signed number (stored in x9) and obtained a negative signed value (stored in x10) to test the mulhsu with a negative number taken as an unsigned

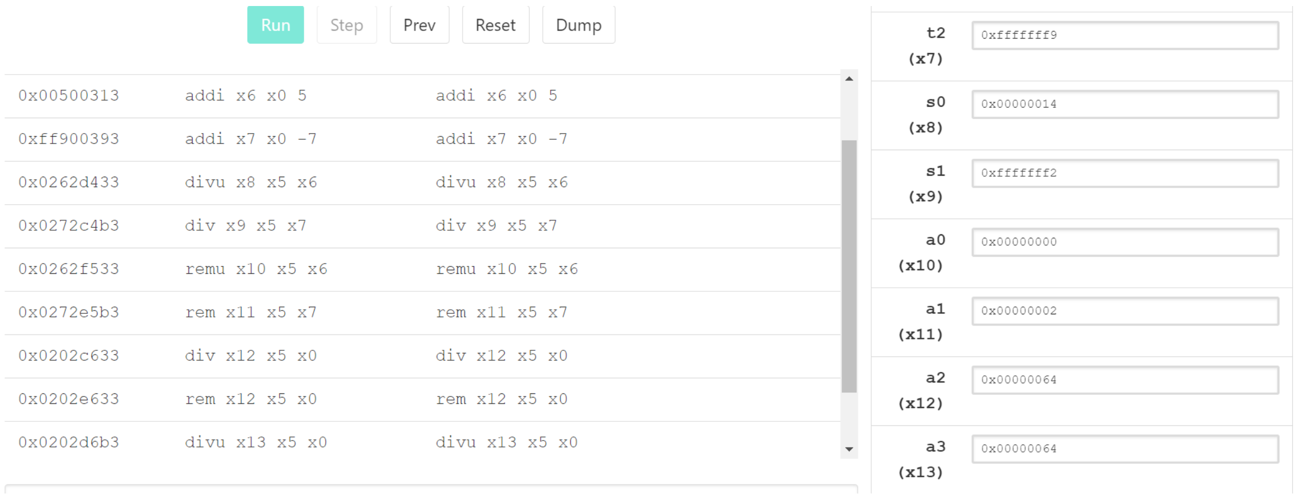
Testing the Divide and Remainder Instructions - test_03

→ In the code we first initialize registers x5,x6,x7 to 100,5 and -7 respectively. Next we implement basic divisions and remainder operations using those registers. Finally we implement divisions by 0 and observe the quotient and remainder.



```
Signals              Waves
Time                 0                                            100 ns                                      200 ns
instr_word[31:0] =   06400293 00500313 FF900393 0262D433 0272C4B3 0262F533 0272E5B3 0202C633 0202E633 0202D6B3 00000000
rs1[4:0] =           00                         05                                                              00
rs2[4:0] =           04       05       19       06       07       06       07       00
rd[4:0] =            05       06       07       08       09       0A       0B       0C                0D       00
datain[31:0] =       00000064 00000005 FFFFFFF9 00000014 FFFFFFF2 00000000 00000002 FFFFFFFF 00000064 FFFFFFFF 00000000
```

- We note that datain is the value being written back to the register file, ie the value being stored in rd
- We can test the correctness of the code by comparing it to the values obtained in ***venus*** simulator.


Run Step Prev Reset Dump

```
0x00500313    addi x6 x0 5       addi x6 x0 5
0xff900393    addi x7 x0 -7      addi x7 x0 -7
0x0262d433    divu x8 x5 x6      divu x8 x5 x6
0x0272c4b3    div x9 x5 x7       div x9 x5 x7
0x0262f533    remu x10 x5 x6     remu x10 x5 x6
0x0272e5b3    rem x11 x5 x7      rem x11 x5 x7
0x0202c633    div x12 x5 x0      div x12 x5 x0
0x0202e633    rem x12 x5 x0      rem x12 x5 x0
0x0202d6b3    divu x13 x5 x0     divu x13 x5 x0
```

```
t2 (x7)   0xfffffff9
s0 (x8)   0x00000014
s1 (x9)   0xfffffff2
a0 (x10)  0x00000000
a1 (x11)  0x00000002
a2 (x12)  0x00000064
a3 (x13)  0x00000064
```

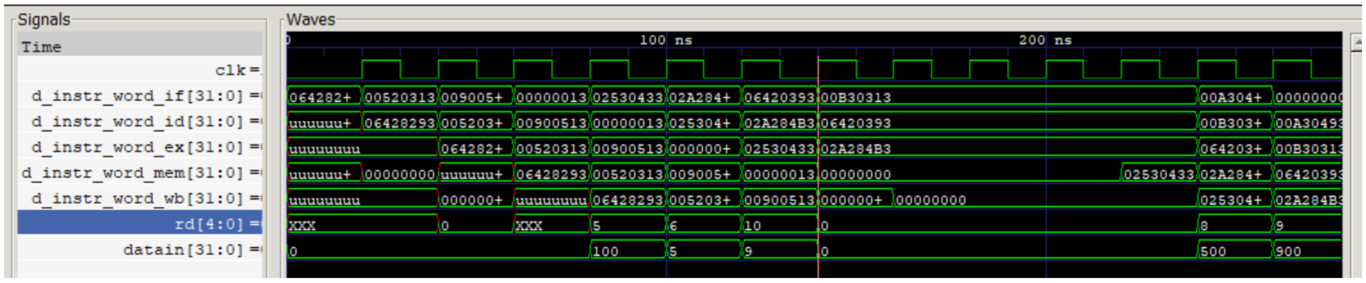
- We can mainly observe that the case were we are dividing by zero behaves as it should. Dividing by zero returns FFFFFFFF and the remainder is the value stored in rs1 which is X'64" or d"100".

Testing the multiply pipelined - test_04

The RISC-V code:

```
06428293 addi x5 x5 100
00520313 addi x6 x4 5
00900513 addi x10 x0 9
00000013 nop
02530433 mul x8 x6 x5
02a284b3 mul x9 x5 x10
06420393 addi x7 x4 100
00b30313 addi x6 x6 11
00a30493 addi x9 x6 10
```

Please take the red line in the picture below as the cursor reference for the explanation below



```
clk=
d_instr_word_if[31:0]=   064282+  00520313 009005+  00000013 02530433 02A284+  06420393 00B30313 00A304+ 0000000
d_instr_word_id[31:0]=   uuuuuu+  06428293 005203+  00900513 00000013 025304+  02A284B3 06420393 00B303+ 00A3049
d_instr_word_ex[31:0]=   uuuuuuuu 064282+  00520313 00900513 000000+  02530433 02A284B3 064203+ 00B3031
d_instr_word_mem[31:0]=  uuuuuu+  00000000 uuuuuu+  06428293 00520313 009005+  00000013 00000000 02530433 02A284+ 0642039
d_instr_word_wb[31:0]=   uuuuuuuu 000000+  uuuuuuuu 06428293 005203+  00900513 000000+  00000000 025304+ 02A284B
rd[4:0]=                 XXX 0 XXX 5 6 10 0 8 9
datain[31:0]=            0 100 5 9 0 500 900
```

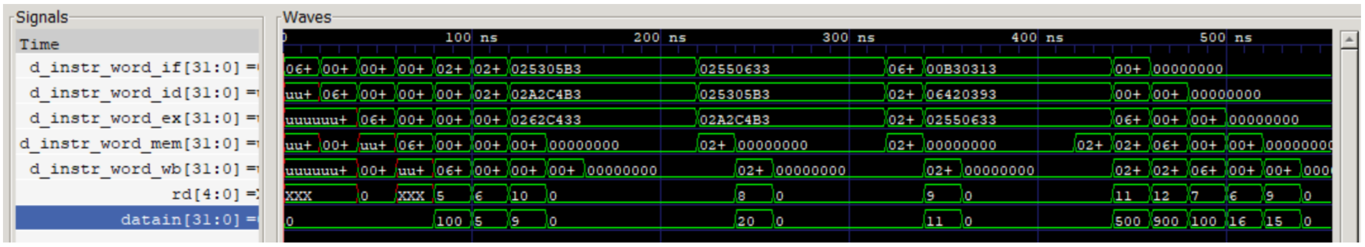
- We can first observe in the EX stage that the first multiply instuction enters but does not stall anything as the second multiply enters right after it
- After the second multiply we can observe everything being stalled for 5 cc.
- The addi that is after the second mult gets stalled in the ID and the addi that is after it gets stalled in the IF

- After the NOP passes through the mem stage, everything after it is just empty instructions (zeros) until the first multiply finishes executing, then it gets sent to mem and everyting continues normally
- Finally we can observe through rd and datain (these are the signals being sent to rf) that all instructions are being executed correctly, and all values are being written in the register file correctly

Testing the divide pipelined - test_05

The RISC-V code:

```
06428293 addi x5 x5 100
00520313 addi x6 x4 5
00900513 addi x10 x0 9
00000013 nop
0262c433 div x8 x5 x6
02a2c4b3 div x9 x5 x10
025305b3 mul x11 x6 x5
02550633 mul x12 x10 x5
06420393 addi x7 x4 100
00b30313 addi x6 x6 11
00a30493 addi x9 x6 10
```



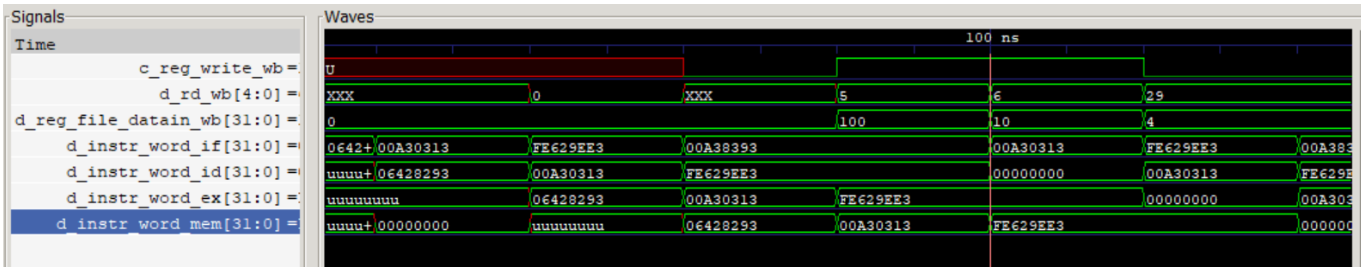
- The clearest way to observe that it works is through rd and datain
- Other than that, reading on the ex instruction we can observe that the first and second divide stay for five cc each, and then the first multiply enters and directly after it the second multiply enters.
- The instructions also come out to the wb stage correctly, with full zeros whenever there is a stall in the ex

Testing data forwarding for branch instructions - test_08

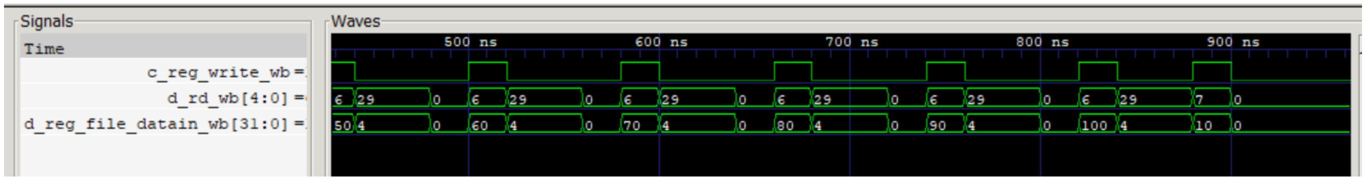
The RISC-V code:

```
06428293 addi x5 x5 100
                 loop:
00A30313 addi x6 x6 10
```

```
FE629EE3 bne x5 x6 loop
00A38393 addi x7 x7 10
```



- In the above picture we can observe first how the instruction after the branch got fetched, but as soon the branch got resolved as taken, the fetched instruction got flushed and did not make it to the ID stage (right side of the cursor)

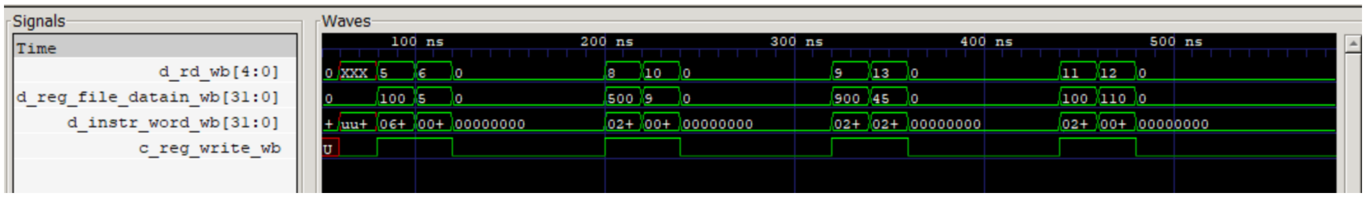


- We can first note that the instructions are being executed in a loop meaning the ranch is taken correctly
- Then we can notice that as soon as x6 becomes 100 no more instructions are taken meaning the branch got taken at the correct time
- And after it the last addi that comes right after the branch instruction got taken

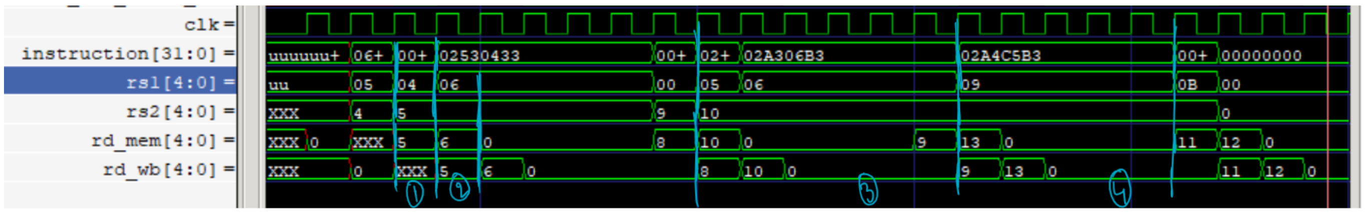
Testing data forwarding with mult and div - test_07

The RISC-V code:

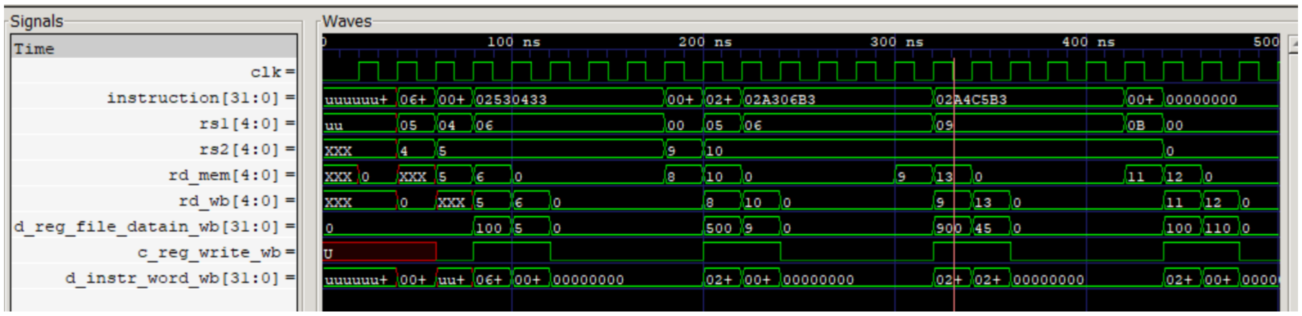
```
06428293 addi x5 x5 100
00520313 addi x6 x4 5
02530433 mul x8 x6 x5
00900513 addi x10 x0 9
02A284B3 mul x9 x5 x10
02A306B3 mul x13 x6 x10
02A4C5B3 div x11 x9 x10
00A58613 addi x12 x11 10
```



- We can observe all the values being stored correctly in the correct registers.
- Refer to the below explanation for further detail



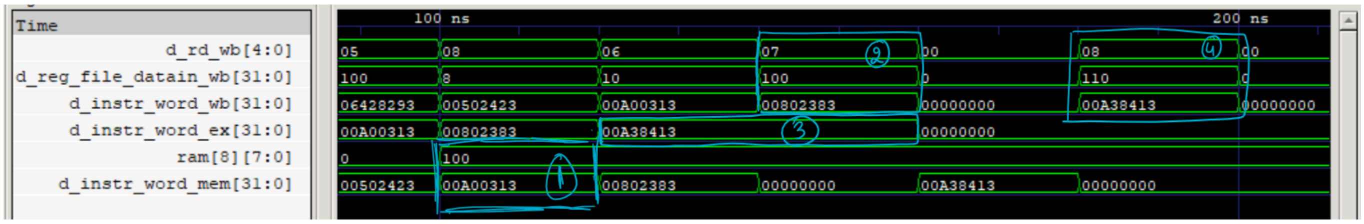
1. We can observe for the second addi that even though rs2 appears to be 5 and rd_mem is 5, it does not cause conflict because of the opcode condition stated, which takes into consideration that not all instructions have an rs2, some have immediate instead, even some instructions don't have rs1.
2. The first mul depends on both instructions before it. It takes regA from memory and regB from wb
3. Then we have 2 mul pipelined, they each successfully take values from wb and mem stages
4. We have the div whichwas stalled in the ID stage waiting for the mul to finish in the ex stage, then successfully took its regA value from the wb stage



Testing data forwarding with load - test_06

The RISC-V code:

```
06428293 addi x5, x5, 100
00502423 sw x5, 8(x0)
00A00313 addi x6, x0, 10
00802383 lw x7, 8(x0)
00A38413 addi x8, x7, 10
```



1. We can see that the correct value was stored in ram at location 8 during the phase where sw instruction was in mem stage, so normal data forwarding works correctly
2. We can see how the value was fetched from memory and loaded into x7
3. We can see how the last addi was stalled in EX to wait for the load to arrive to the MEM stage and fetch the value
4. We can see that the correct value was calculated and stored in x8 for the last addi