

## Report of Lab 2.7

Lalit Meena,2019CS50439

For this Lab 2 we are building microarchitecture for subset of arm instructions.

This lab involves putting together some stage 1 modules, to form a simple processor which can execute the following subset of ARM instructions with limited variants/features. {add, sub, cmp, mov, ldr, str, beq, bne, b}. Then in stage 3, we extend single cycle datapath(stage 2) to flexible and efficient clock period multicycle path.

In stage 4, we have tested our design exhaustively and reported waves/test-cases for most DP instructions. Then in stage 5, we are adding one more features to our implementation which supports shift operations on register and immediate operand variations. In stage 6, we are supporting more DT instructions features. It includes byte and half word transfers (signed and unsigned), auto increment/decrement with option of pre/post indexing.

Now in stage 7 we are including all variations of multiply instructions with mul\_acc module

### Zip folder name – L2.7\_2019CS50439

Submission folder, L2.7 2019CS50439 contains-

1. **alu.vhd** – alu module design file
2. **data\_mem.vhd** - data memory 128x32 module design file
3. **rfile.vhd** - design file for module register file 16x32
4. **others.vhd**-contains additional modules required for stage-2(Instruction Decoder, Condition Checker)
5. **flags.vhd**- design file for module flags (**one line change**)
6. **multicycle.vhd**- SIMPLE multi CYCLE PROCESSOR
7. **shifter.vhd**- Shifter which supports 4 shift types-LSL,LSR,ASR,ROR
8. **PM.vhd**- combination circuit path between the processor and memory
9. **Mul\_acc.vhd**- module to perform multiply -accumulator computation
- 10.**package.vhd** -package to declare some types.-etc (**states added**)**run.do**

### testbench files-

**testbench\_multicycle.vhd** - testbench file for module simple multicycle processor

**testbench\_mulac.vhd** - testbench file for mul\_acc module-**test\_mulacc.jpg**

**mul\_acc TESTCASES EPWAVE PICS-mtestcase-1,2,3,4,5,6....**

**mul\_acc TESTCASES files- mtestcasen.s, n is 1,2,.....**

**Report of Lab 2.7**-lab 2 stage 7 description and test cases(also screenshots )

**More details are in next pages of each module-**

## Mul\_acc.vhd – multiply-accumulator module

### explanation-

It is combination circuit that performs multiply -accumulator computation. My implementation is based on this another approach discussed in class.-

### Another way

```
signal op1, op2: std_logic_vector (31 downto 0);
signal result: std_logic_vector (63 downto 0);
signal p_s: signed (65 downto 0);
signal x1, x2: std_logic;
x1 <= op1(31) when instr = smull else '0';
x2 <= op2(31) when instr = smull else '0';
p_s <= signed (x1 & op1) * signed (x2 & op2);
result <= std_logic_vector(p_s (63 downto 0));
-- uses 4 DSP48E1 !
```

This is purely combinational circuit, related as  $r = a * b (+c)$

```
entity mulacc is
port(
  a: in word ;
  b: in word;
  c: in std_logic_vector (63 downto 0); --word
  mopc: in std_logic_vector (2 downto 0);
  r: out std_logic_vector (63 downto 0));
```

6 types of multiply and accumulate operation possible , -

a) mul- "000" multiply with 32-bit output  $32 \times 32 \Rightarrow 32$

m1a- "001" multiply-accumulate with 32-bit output  $32 \times 32 + 32 \Rightarrow 32$

b) umull- "100" unsigned multiply with 64-bit output  $32 \times 32 \Rightarrow 64$

umlal- "101" unsigned multiply-accumulate with 64-bit output  $32 \times 32 + 64 \Rightarrow 64$

smull- "110" signed multiply with 64-bit output  $32 \times 32 \Rightarrow 64$

smlal- "111" signed multiply-accumulate with 64-bit output  $32 \times 32 + 64 \Rightarrow 64$

### simulated with help of testbench\_mulac.vhd-

Here we divided testing into two parts for short and long instructions so,each part can also be run seperately.For comparison among these 6 types of variations(varying mopc signal) we have kept a,b,c input to same complex value,covering possible variation.

Result =  $a * b (+c)$

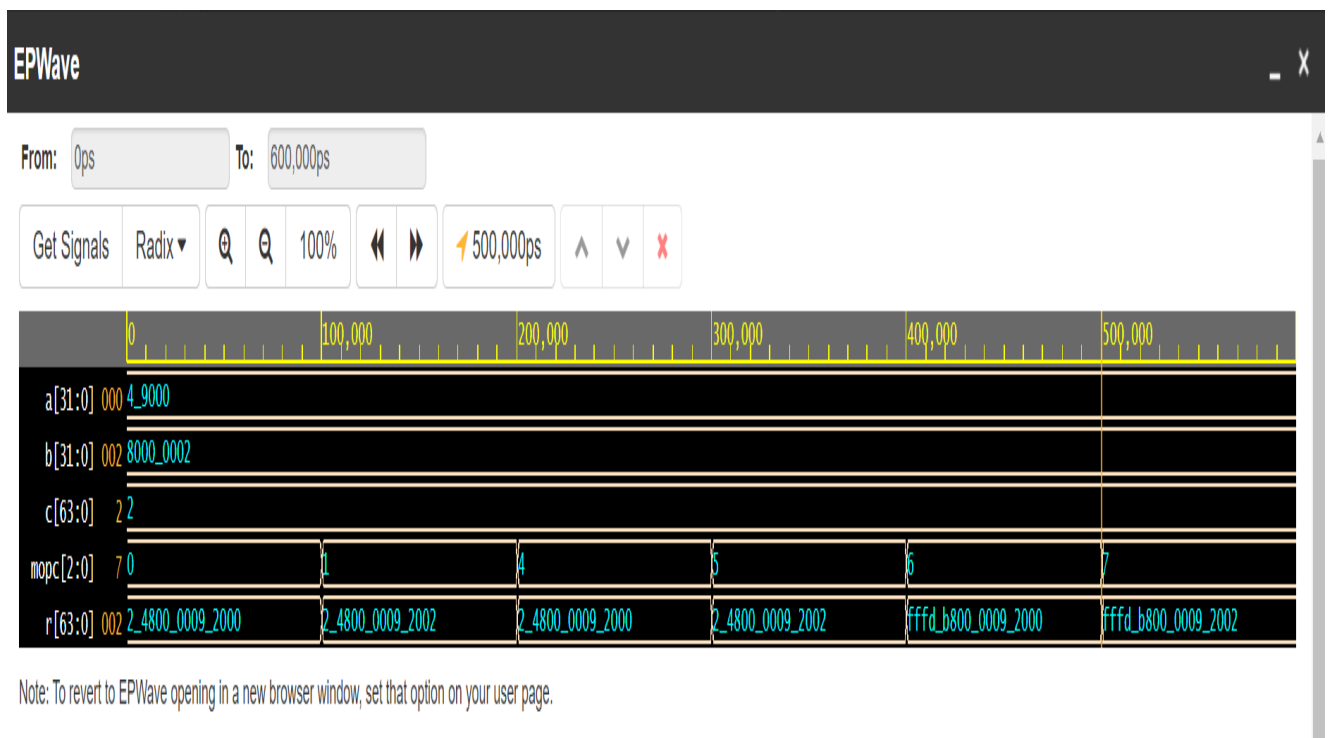
a-#0X00049000

b-#0X80000002

c-#2

We have introduced mopc signal to track currently which of 6 types of variations is running. It is 3 bit signal corresponding to mopc\_type: (MUL,MLA,UMULL,UMLAL,SMULL,SMLAL);

**test\_MULACC**-here all six types of variations are tested.



## Merging stage 7 in main multicycle processor

MULACC module merge require appropriate adjustment,signals to match its port values to main programme signals.

Number of states used inside main clocked process is states-

(fetch,read\_AB,arithm,MSTR,MLDR,w2RF,READC,RSHIFT,WB2RF,MULAC,MW2RF).Also used state signal in testbench to show at which state running currently.

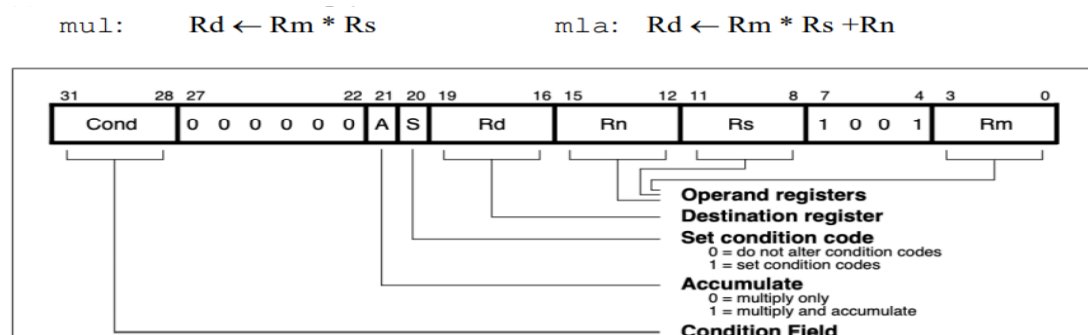
At this stage our design have only 11 states ,here we HAVE added two new states –

**MULAC**-to perform multiply and accumulation computation and then settings flags.storing result in 64 bit MLA register.

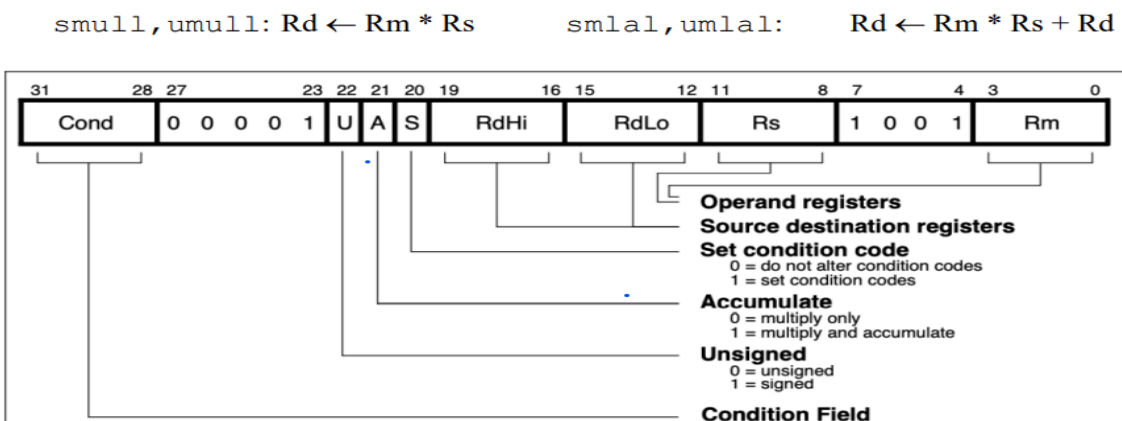
**MW2RF**-for simplicity to write agin Register file ,this is used to write RDHIGH register when instruction is of long -64 bit type.here need to indrodue new state to access RF again .

For flags ,we take advantage of wise implementation to result input-MLA(MSB 32) or MLA(lsb)

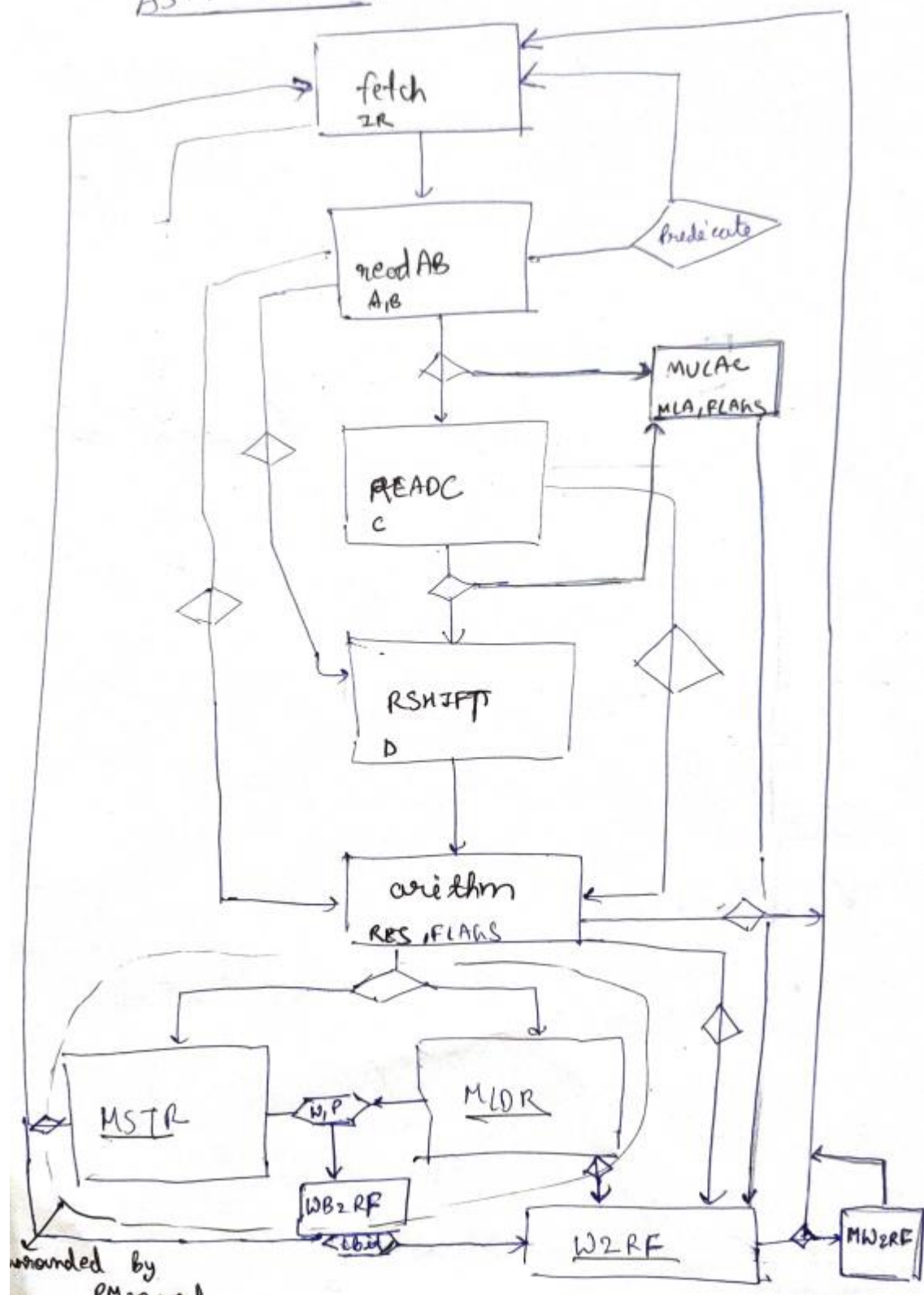
**Format of mul\_acc Instructions in ARM according to IR(23)-long instruction or not-**



Here long='1'-



# ASM CHART.



# Testcases

## simulated with help of testbench multicycle.vhd-

state change like flags and registers at end of program or step can be checked by signal like A,B,C,D,FLAGS, IR,PC,RES,state,minstype ,MWetc. Here main things to look - mopc signal and MLA 64 bit result which then further checked by moiv instruction(B signal).

Correctness can be justified by values of registers read after instructions,etcAlso added extra mov instructions to read and show value of B,RES OR 2<sup>nd</sup> operand.

We can observe number of cycles etc,also used state signal to show which state currently And also which type of mopc variation.

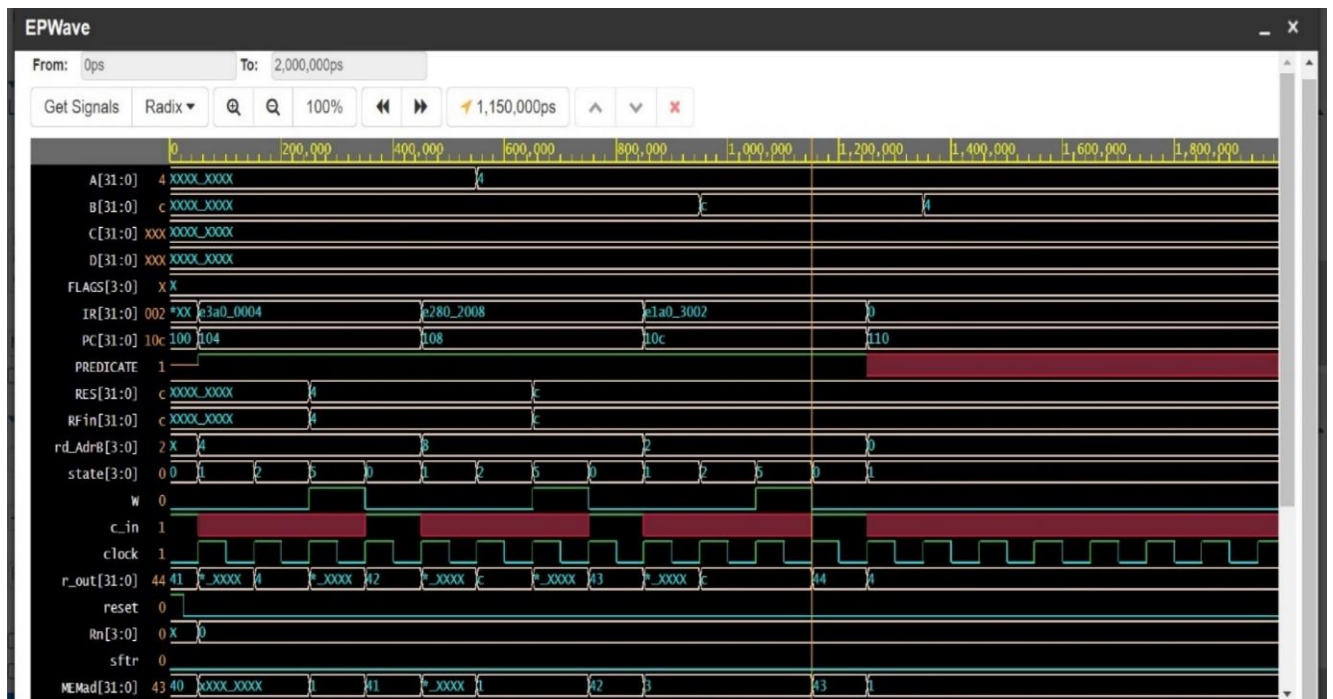
**Below are MUL exhaustive related testcases-which covers all variants multiply and accumulate instruction possible in MUL.**

We can identify MUL\_ACC related parameters in middle-end -mopc(type),MLA(64 bit result) ,etc.

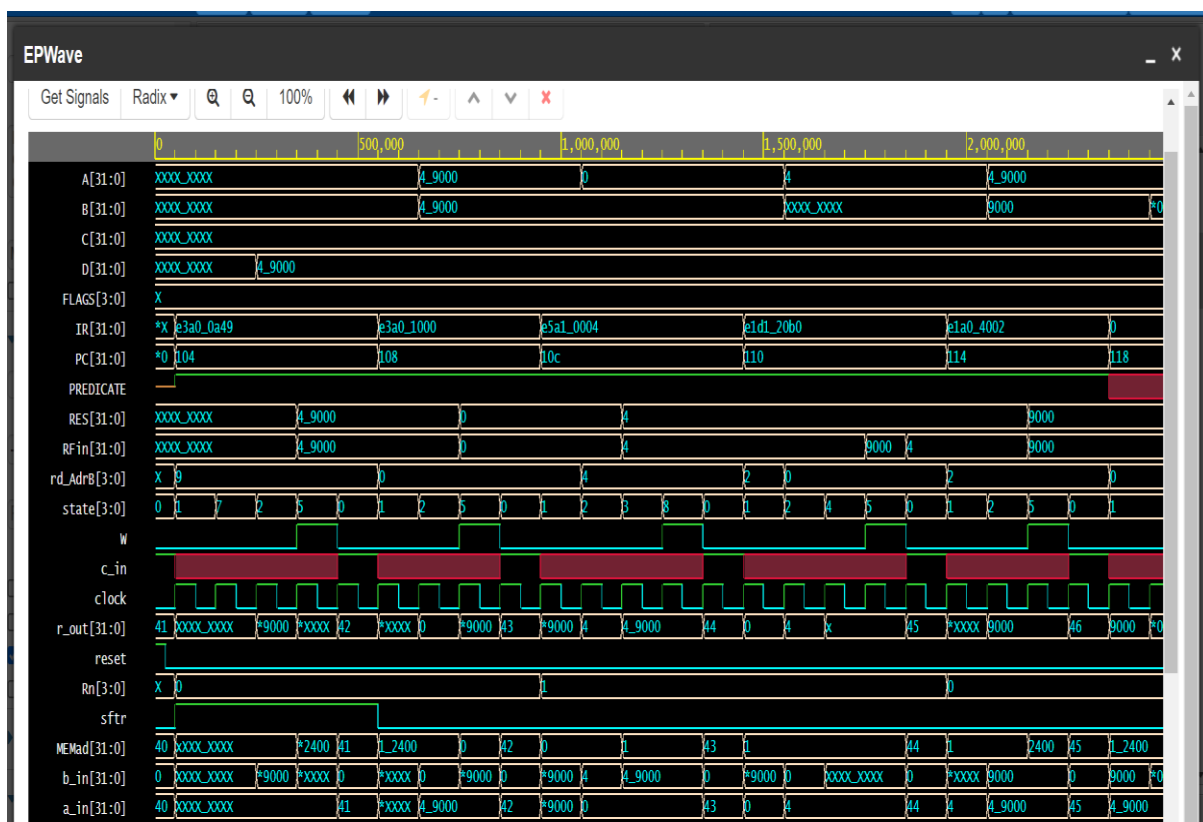
Take help of **run time limit** on simulator for taking required PC increments.For clock related testbench signals generation and this limit can sometime generate error like pipe broken(resolve by changing).On checking again runtime limit canbe set to epwave top listed value.

**TESTCASES SEPARATE EPWAVE PICS and assembly files CAN also BE FOUND IN SUBMITTED FOLDER.**

Check with previous testcase-2,found that previous implementation is working same for previous testcases.



Also working same for previous pmconnect stage pmtestcase1



## Mtestcase1

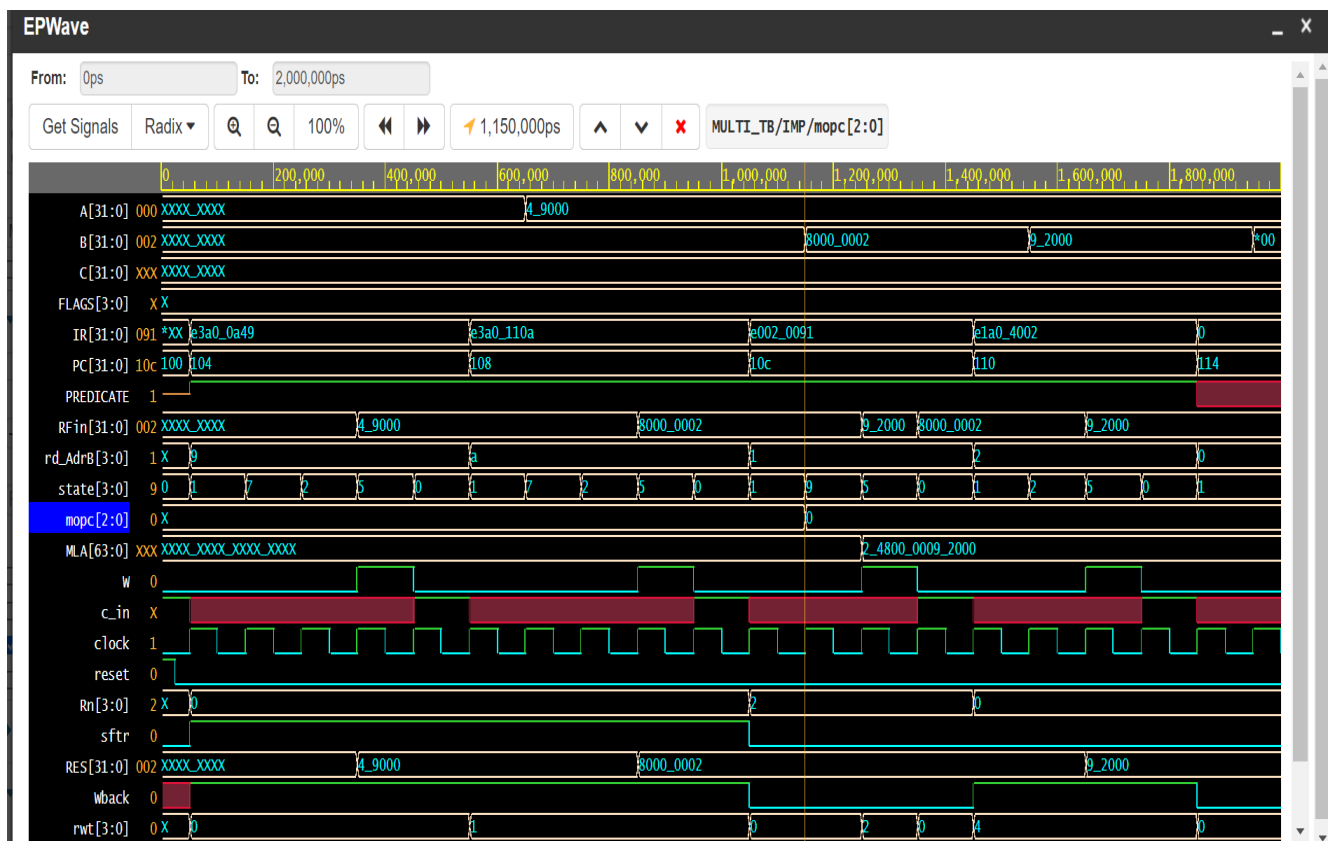
The screenshot shows a debugger interface with two main windows: **RegistersView** and **CodeView**.

**RegistersView:** Displays the state of various registers. The **General Purpose** tab is selected, showing registers R0 through R15. R2 and R15 are highlighted in red. R15 (PC) is at address 0000100c.

Register	Value
R0	:00049000
R1	:80000002
R2	:00092000
R3	:00000000
R4	:00000000
R5	:00000000
R6	:00000000
R7	:00000000
R8	:00000000
R9	:00000000
R10 (s1)	:00000000
R11 (fp)	:00000000
R12 (ip)	:00000000
R13 (sp)	:00011400
R14 (lr)	:00000000
R15 (pc)	:0000100c

**CodeView:** Displays the assembly code for **mtestcase1.o**. The code is in the **.text** section.

```
.text
00001000:E3A00A49  mov r0,#0X00049000
00001004:E3A0110A  mov r1,#0X80000002
00001008:E0020091  mul r2,r1,r0
0000100C:E1A04002  mov r4,r2
.end...
```





## Mtestcase2

RegistersView

General Purpose

Floating Point

Hexadecimal

Unsigned Decimal

Signed Decimal

R0	: 00049000
R1	: 80000002
R2	: 00000002
R3	: 00092002
R4	: 00000000
R5	: 00000000
R6	: 00000000
R7	: 00000000
R8	: 00000000
R9	: 00000000
R10 (s1)	: 00000000
R11 (fp)	: 00000000
R12 (ip)	: 00000000
R13 (sp)	: 00011400
R14 (lr)	: 00000000
R15 (pc)	: 00001010

CodeView

mtestcase2.o

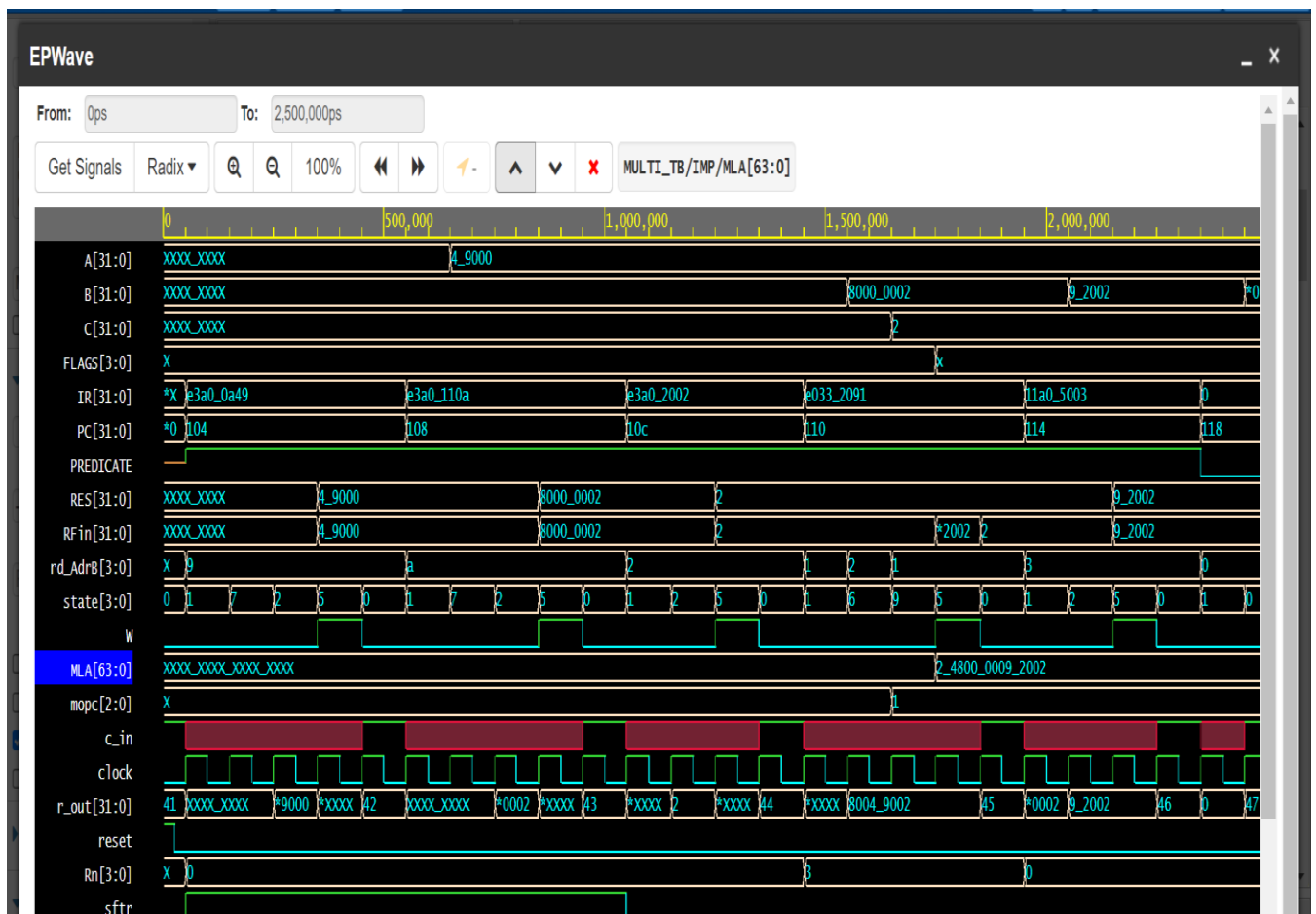
```

.text
00001000:E3A00A49  mov r0,#0X00049000
00001004:E3A0110A  mov r1,#0X80000002
00001008:E3A02002  mov r2,#2
0000100C:E0332091  mlas r3,r1,r0,r2
00001010:11A05003  movne r5,r3
.end...

```

OutputView

WatchView



## Mtestcase3

RegistersView

General Purpose

Floating Point

Hexadecimal

Unsigned Decimal

Signed Decimal

R0

: 00049000

R1

: 80000002

R2

: 00092000

R3

: 00024800

R4

: 00092000

R5

: 00024800

R6

: 00000000

R7

: 00000000

R8

: 00000000

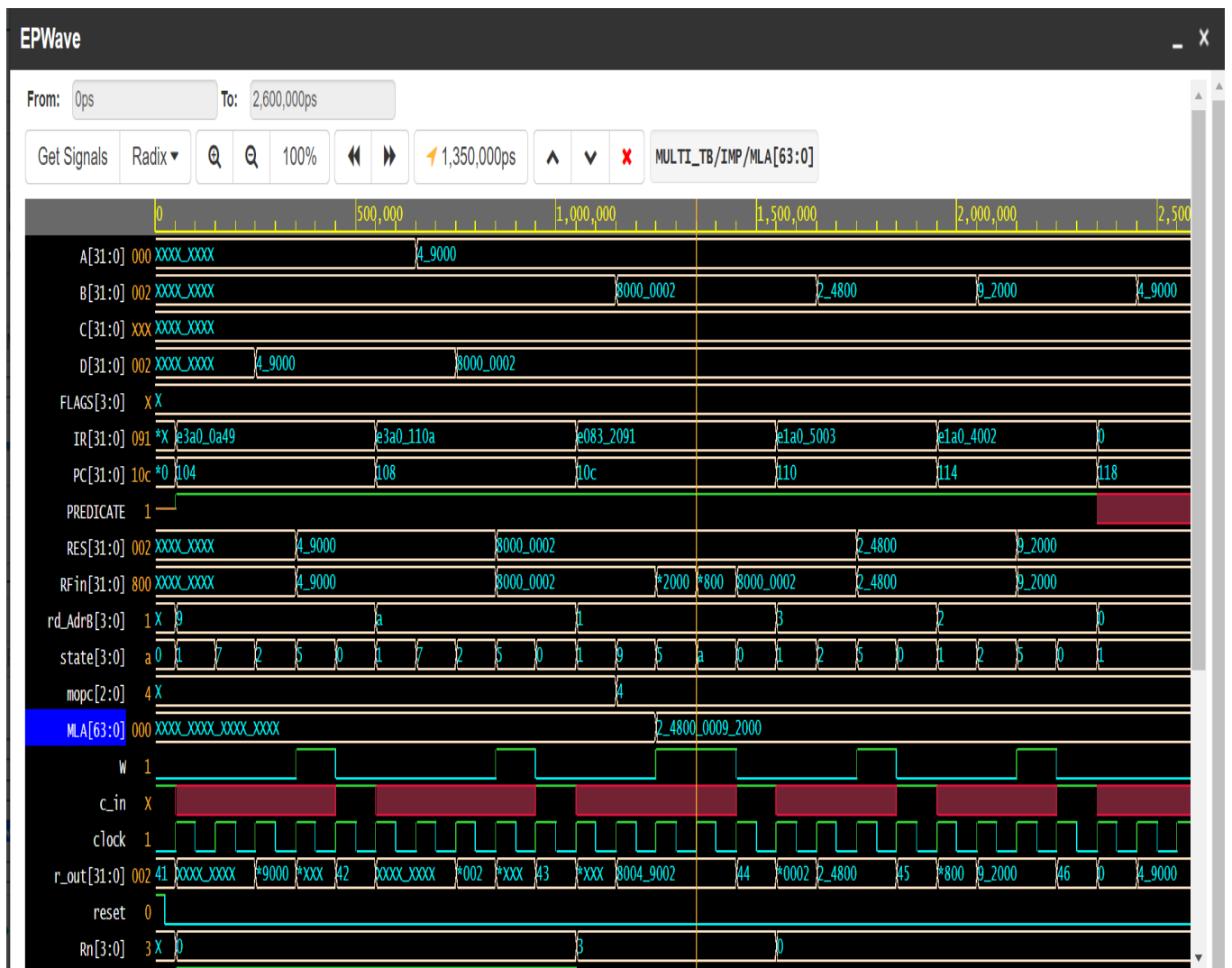
CodeView

mtestcase3.o

```

.text
00001000:E3A00A49    mov r0,#0X00049000
00001004:E3A0110A    mov r1,#0X80000002
00001008:E0832091    umull r2,r3,r1,r0
0000100C:E1A05003    mov r5,r3
00001010:E1A04002    mov r4,r2
                                .end...

```



## Mtestcase4

RegistersView

General Purpose

Floating Point

Hexadecimal

Unsigned Decimal

Signed Decimal

R0 : 00049000  
R1 : 80000002  
R2 : 00092002  
R3 : 00024800  
**R4 : 00092002**  
R5 : 00024800  
R6 : 00000000  
R7 : 00000000  
R8 : 00000000  
R9 : 00000000  
R10 (s1) : 00000000  
R11 (fp) : 00000000

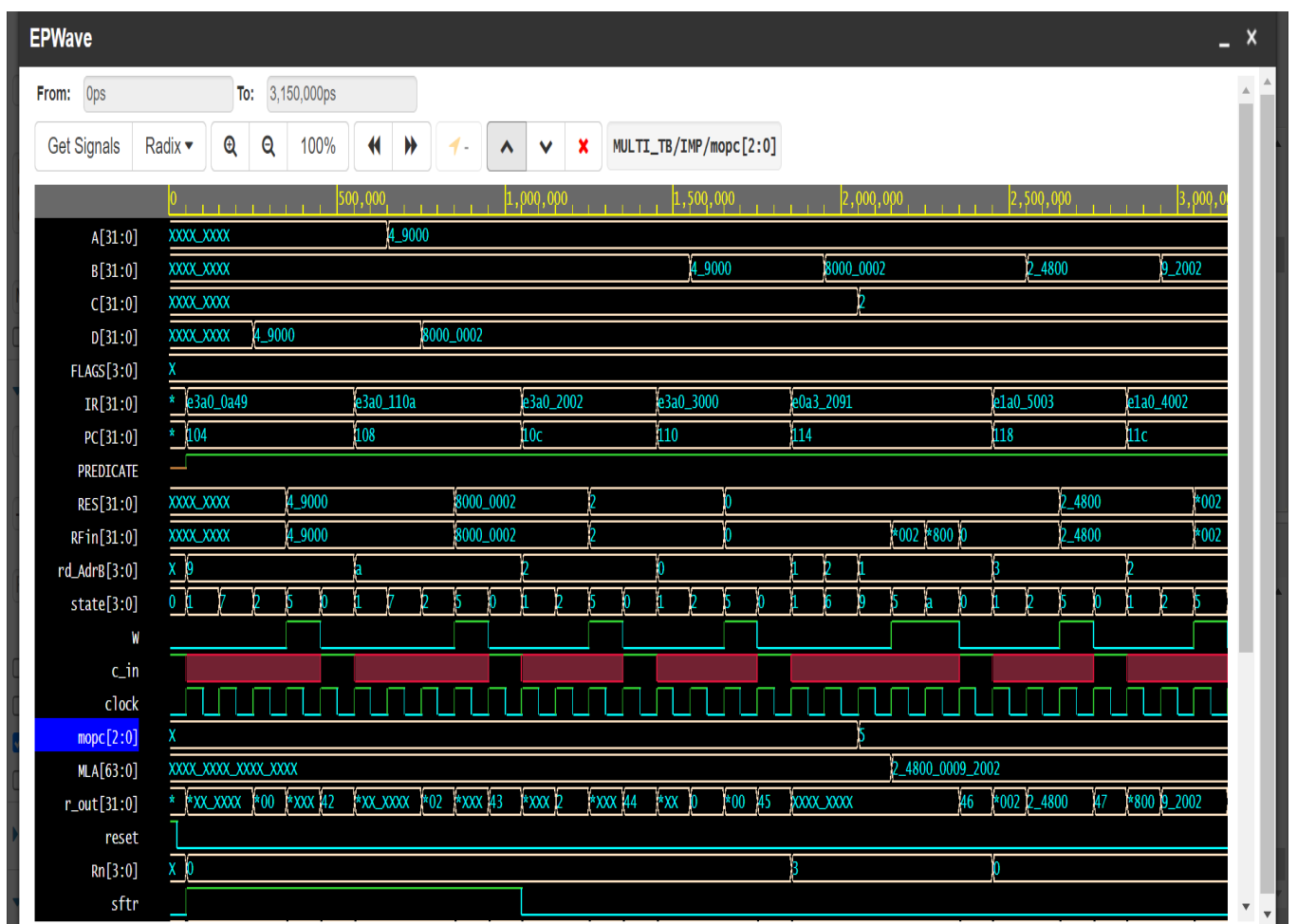
CodeView

mtestcase4.o

```

.text
00001000:E3A00A49    mov r0,#0X00049000
00001004:E3A0110A    mov r1,#0X80000002
00001008:E3A02002    mov r2,#2
0000100C:E3A03000    mov r3,#0
00001010:E0A32091    umlal r2,r3,r1,r0
00001014:E1A05003    mov r5,r3
00001018:E1A04002    mov r4,r2
.end

```



## Mtestcase5

RegistersView

General Purpose
Floating Point

Hexadecimal
Unsigned Decimal
Signed Decimal

R0 : 00049000  
R1 : 80000002  
R2 : 00092000  
R3 : fffdb800  
**R4 : 00092000**  
R5 : fffdb800  
R6 : 00000000  
R7 : 00000000  
R8 : 00000000  
R9 : 00000000  
R10 (s1) : 00000000  
R11 (fp) : 00000000

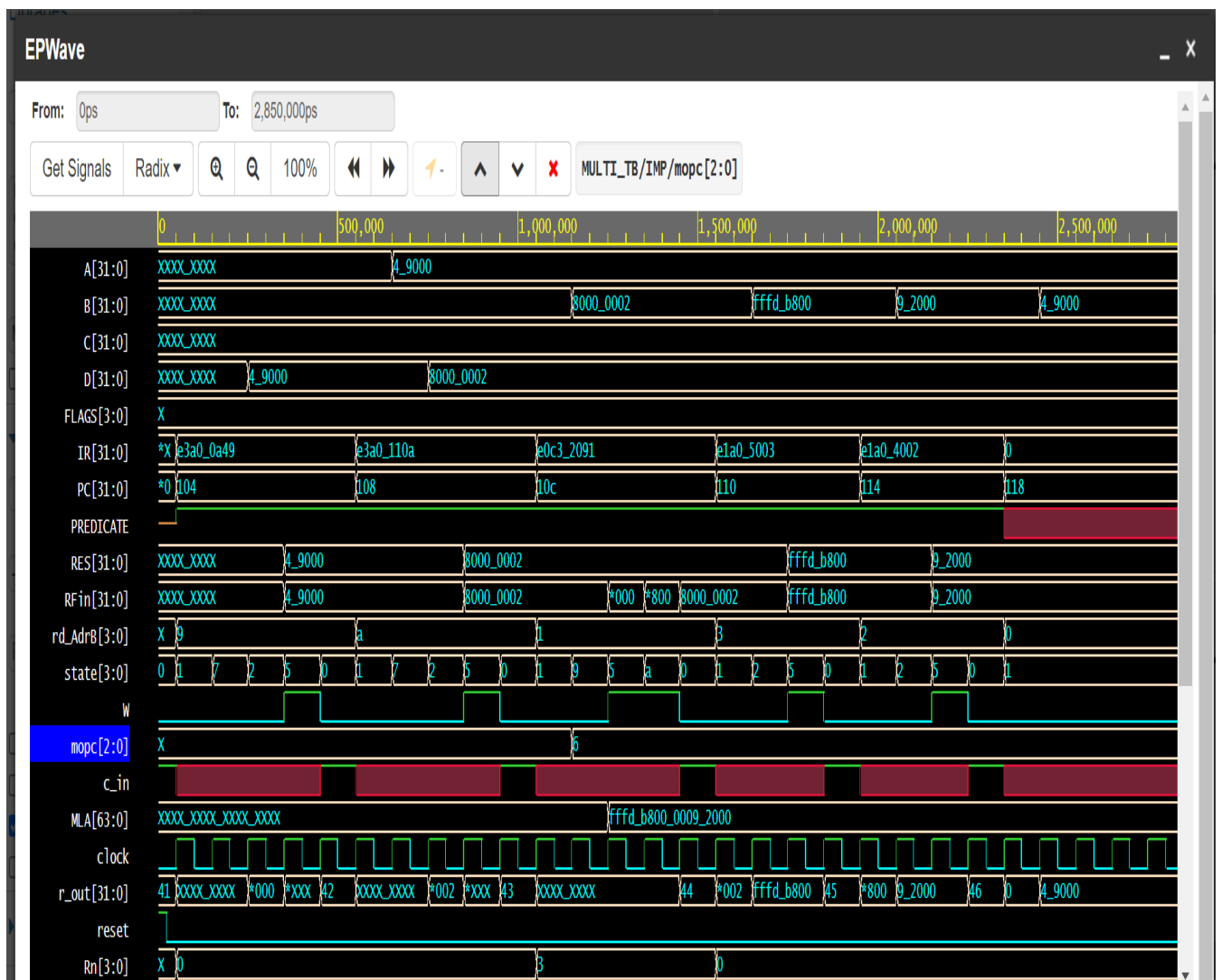
CodeView

mtestcase5.o

```

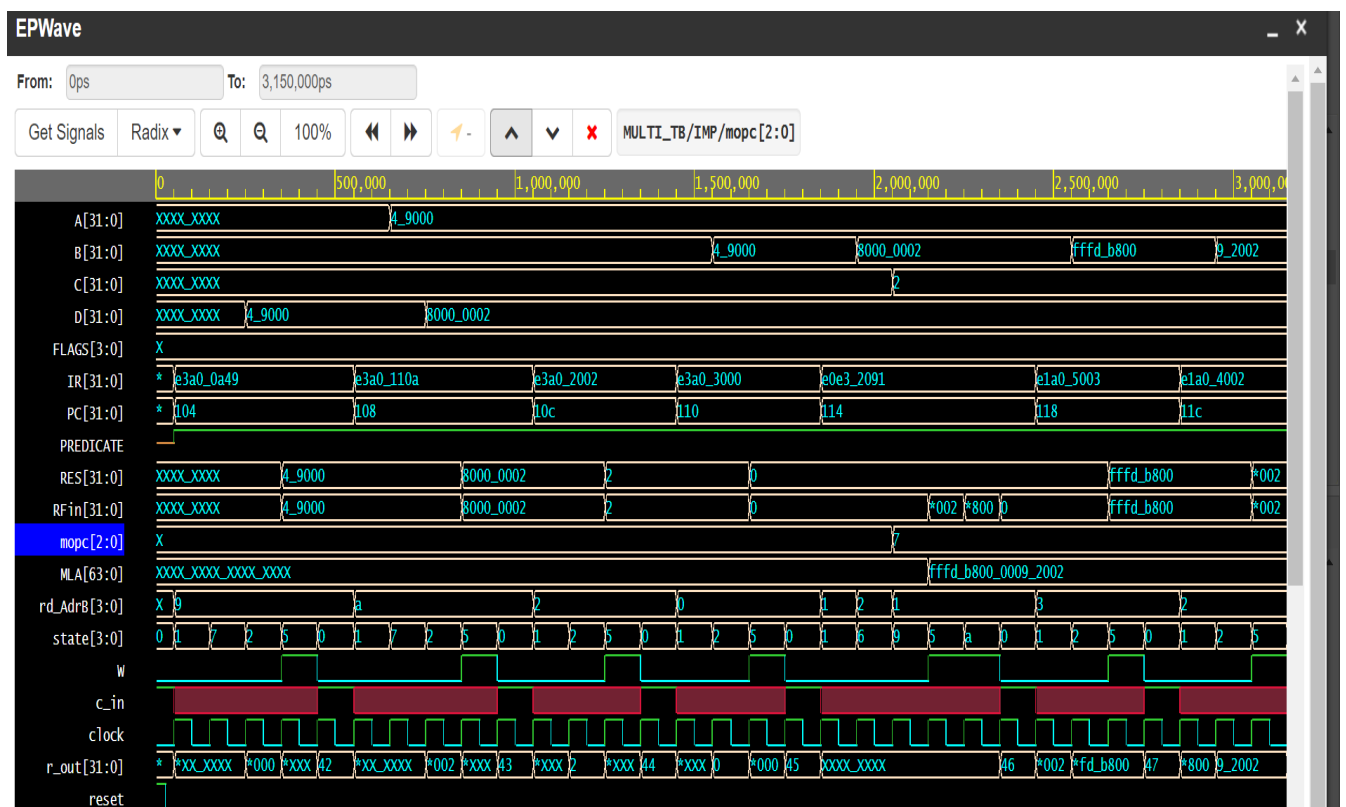
.text
00001000:E3A00A49  mov r0,#0X00049000
00001004:E3A0110A  mov r1,#0X80000002
00001008:E0C32091  smull r2,r3,r1,r0
0000100C:E1A05003  mov r5,r3
00001010:E1A04002  mov r4,r2
.end...

```



### Mtestcase6

The screenshot displays the Keil uVision IDE interface. On the left, the 'RegistersView' window is open, showing a list of registers. Register R4 is highlighted in red, indicating it is the current register of interest, and its value is 00092002. Other registers like R0, R1, R2, etc., are listed with their respective values. On the right, the 'CodeView' window is open, showing assembly code for a file named 'mtestcase6.o'. The instruction 'mov r4, r2' at memory address 00001018 is highlighted in blue, corresponding to the value in register R4. The code includes a '.text' section and several 'mov' and 'smlal' instructions.



For each module, work and analysis I have put files in edaplayground like below-

Used for simulation- Aldec riviera pro 2020.04

simulation settings For eg mtestcase1-

The screenshot displays the edaplayground web interface. The top navigation bar includes a logo, a 'Run' button, a 'Save' button, a 'Copy' button, and a message: 'If this page reloads when you click "Run" please read this.' The interface is divided into several sections:

- Libraries:** A sidebar on the left with sections for 'Testbench + Design' (VHDL), 'Libraries' (None, OVL 2.8.1, OSVVM), 'Top entity' (MULTI\_TB), 'Tools & Simulators' (Aldec Riviera Pro 2020.04), 'Compile Options' (-2019 -o), 'Run Options' (Run Options), 'Run Time' (2800ns), and 'Examples'.
- Design Files:** A central area showing a list of files: testbench.vhd, run.do, design.vhd, package.vhd, alu.vhd, data\_mem.vhd, flags.vhd, others.vhd, rfile.vhd, shifter.vhd, PM.vhd, and mul\_acc.vhd. The 'testbench.vhd' file is selected and its content is displayed in the 'VHDL Testbench' window.
- VHDL Testbench:** A code editor showing the following VHDL code:

```
273 CLK<='1';
274 WAIT FOR 50 ns;
275 CLK<='0';
276 WAIT FOR 50 ns;
277 CLK<='1';
278 WAIT FOR 50 ns;
279 CLK<='0';
280 WAIT FOR 50 ns;
281
282 CLK<='1';
283 WAIT FOR 50 ns;
284 CLK<='0';
285 WAIT FOR 50 ns;
286 CLK<='1';
287 WAIT FOR 50 ns;
288 CLK<='0';
289 WAIT FOR 50 ns;
290
291
```
- VHDL Design:** A code editor showing the following VHDL code:

```
23 ARCHITECTURE dm OF datam IS
24
25 --SIGNAL MEMORY : DMEM;
26 SIGNAL TEMP :word;
27 SIGNAL ADDR : INTEGER RANGE 0 TO 127;--PGM incorporated
28
29 --Mtestcase 1--
30 SIGNAL MEMORY : DMEM:=
31 (64 => X"E3A00A49",
32 65 => X"E3A0110A",
33 66 => X"E0020091",
34 67 => X"E1A04002",
35 others => X"00000000"
36 );
37 -- --Mtestcase 2--
38 -- SIGNAL MEMORY : DMEM:=
```
- Log:** A window showing the simulation log with the following text:

```
# KERNEL: Time: 0 ps, Iteration: 0, Instance: /MULTI_TB/IMP/DTM, Process: line__25/.
# KERNEL: WARNING: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
# KERNEL: Time: 0 ps, Iteration: 0, Instance: /MULTI_TB/IMP/DCD, Process: line__33.
# KERNEL: WARNING: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
# KERNEL: Time: 50 ns, Iteration: 3, Instance: /MULTI_TB/IMP/DTM, Process: line__257.
# KERNEL: stopped at time: 2800 ns
# VSIM: Simulation has finished.
Finding VCD file...
./dump.vcd
[2022-03-27 06:21:37 UTC] Opening EPWave...
Done
```
- EPWave:** A window at the bottom right showing the simulation results.

For more testcases and live running ,edaplayground can be played in demo for this stage and public link of it can be shared.