COL216 Assignment 2

Stage 4

Rishabh Dhiman
2020CS10837

6 March 2022

1 Objective

Test the processor designed in previous stages using ARM assembly codes.

2 Technical Details

- The VHDL code was analyzed, and simulated using GHDL 1.0.0.
- The waveform viewer used is GTKWave Analyzer v3.3.104.

3 Documentation

The submission contains a single `arm` directory along with a `makefile`.

The `arm` directory contains three folders, `scripts`, `assembly` and `binary`.

- `scripts` directory contains two bash scripts `to-bin.sh` and `create-mem.sh`.
- `assembly` directory contains three assembly codes, `fact.s`, `range-sum.s`, and `neg-fib.s`.
- `binary` directory contains three text files `fact.txt`, `range-sum.txt`, and `neg-fib.txt`.

4 Testing Procdure

The code was analyzed, and simulated using GHDL. It was synthesized using Quartus 21.1.

You can simulate it yourself by copying the files to the folder containing the stage 3 submission.

1. Ensure that you are using a Linux machine with Make and GHDL installed.
2. Create a directory `simulation/arm-tests/` exists (if it doesn't already exist).
3. In the commandline, run `make testarm INPUT=` X where $X \in \{$`fact`, `range-sum`, `neg-fib`$\}$.
4. A X`.ghw` waveform file will be created in the `simulation/arm-tests` directory. Note that the earlier file will be overwritten.

5 Script Description

The script, `to-bin.sh` takes as argument the path to an ARM file and outputs the corresponding binary instruction code. It assumes that `arm-linux-gnueabi` toolchain is installed, along with some common linux binaries.

The script, `create-mem.sh` takes as argument the path to an ARM file and outputs the corresponding VHDL memory file with the ARM code hardcoded into it.

6 Program Descriptions

The description of the three programs follow,

6.1 `fact`

This computes 3! and stores the result in `r0`. The code and the corresponding binary-encoded instructions are,

```
@ Code for storing 3! in r0
.text
mov r0, #1
mov r1, #7
mov r2, #1
mov r3, #0
fact_loop:

@ this bit between comments does
@ mul r0, r2
mov r4, r0
mov r0, #0
mov r5, #0
mul_loop:
cmp r5, r2
beq terminate_mul
add r0, r4
add r5, #1
b mul_loop
mov r0, r4
terminate_mul:
@ by adding r0, r2 times

add r2, #1
cmp r1, r2
bne fact_loop
.end
```

```
e3a00001
e3a01007
e3a02001
e3a03000
e1a04000
e3a00000
e3a05000
e1550002
0a000003
e0800004
e2855001
eaffffa
e1a00004
e2822001
e1510002
1affff3
```

6.2 `range-sum`

This computes and stores $0 + 1 + \cdots + i$ at memory location $64 + i$, for $0 \le i < 5$. This computation is done as $a[i] = a[i-1] + i$, $a[0] = 0$. So that `ldr` and `str` are properly tested.

```
e3a000ff
e2800001
e3a01000
e5801000
e5901000
e2811001
e5801004
e5901004
e2811002
e5801008
e5901008
e2811003
e580100c
e590100c
e2811004
e5801010
```

```
.text
mov r0, #255
add r0, #1
@ r0 = 4 * base

@ in base, base + 1, ..., base + 5 - 1, where base = 64
@ store 0, 0 + 1, 0 + 1 + 2, 0 + 1 + 2 + 3, ...
@
@ we do this as follows,
@ a[0] = 0;
@ for (i = 1; i < 5; ++i)
@       a[i] = a[i - 1] + i;
@
@ we implement it like this,
@ a[0] = 0;
@ for (i = 1; i < 5; ++i) {
@       x = a[i - 1];
@       x += i;
@       a[i] = x;
@ }
@
@ we manually unroll the loop to test the immediate offsets of
@   ldr and str

mov r1, #0
str r1, [r0]

ldr r1, [r0, #0]
add r1, #1
str r1, [r0, #4]

ldr r1, [r0, #4]
add r1, #2
str r1, [r0, #8]

ldr r1, [r0, #8]
add r1, #3
str r1, [r0, #12]

ldr r1, [r0, #12]
add r1, #4
str r1, [r0, #16]
.end
```

6.3 `neg-fib`

This computes the negative Fibonacci numbers, F_{-i}. If $a_i = F_{-i}$, then

$$F_{i+2} = F_{i+1} + F_i \implies F_i = F_{i+2} - F_{i+1} \implies a_i = F_{-i} = F_{-i+2} - F_{-i+1} = a_{i-2} - a_{i-1}.$$

F_{-i} is stored in memory location, $64 + i$, for $0 \le i \le 8$.

```
e3a000ff
e2800001
e3a01000
e5801000
e3a01001
e5801004
e3a01008
e2800004
e3a02001
e5103004
e5904000
e0433004
e5803004
e2800004
e2822001
e1520001
1afffff7
```

```
@ compute negative fibonacci as a[i] = a[i - 2] - a[i - 1]
@  where a = mem[64], a[0] = 0, a[1] = 1

mov r0, #255
add r0, #1
@ location of a

mov r1, #0
str r1, [r0]
mov r1, #1
str r1, [r0, #4]
@ set a[0] = 0, a[1] = 1

mov r1, #8
@ ie,
@ a[0] = 0
@ a[1] = 1
@ for (i = 1; i <= 8; ++i)
@   a[i + 1] = a[i - 1] - a[i]

add r0, #4
mov r2, #1 @ index variable
loop:
@ r3 = a[r0 - 1]
@ r4 = a[r0]
@ r4 -= r3
@ a[r0 + 1] = r4

ldr r3, [r0, #-4]
ldr r4, [r0]
sub r3, r4
str r3, [r0, #4]

@ update things
add r0, #4
add r2, #1
cmp r2, r1
bne loop
```

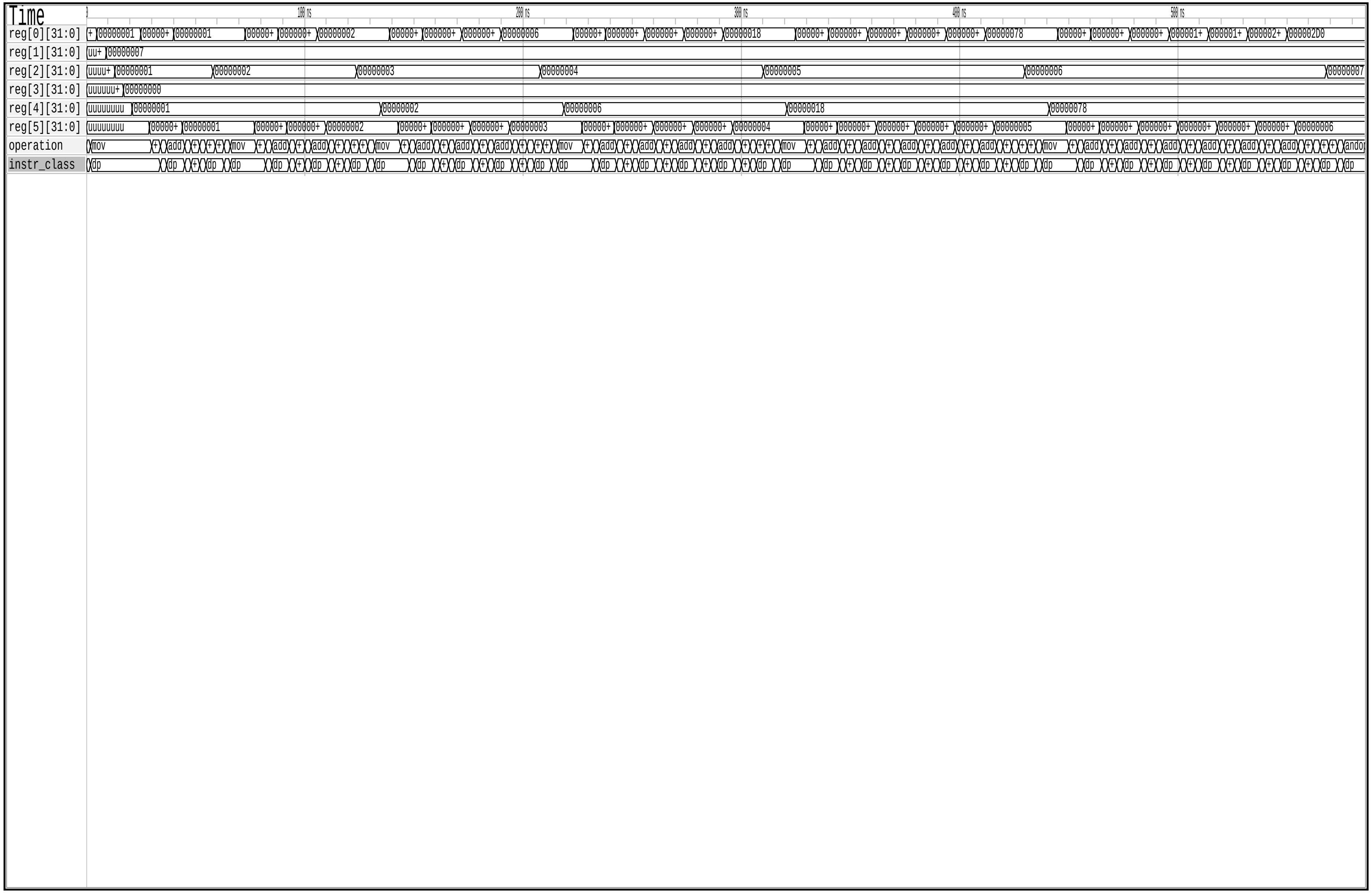
7 Results

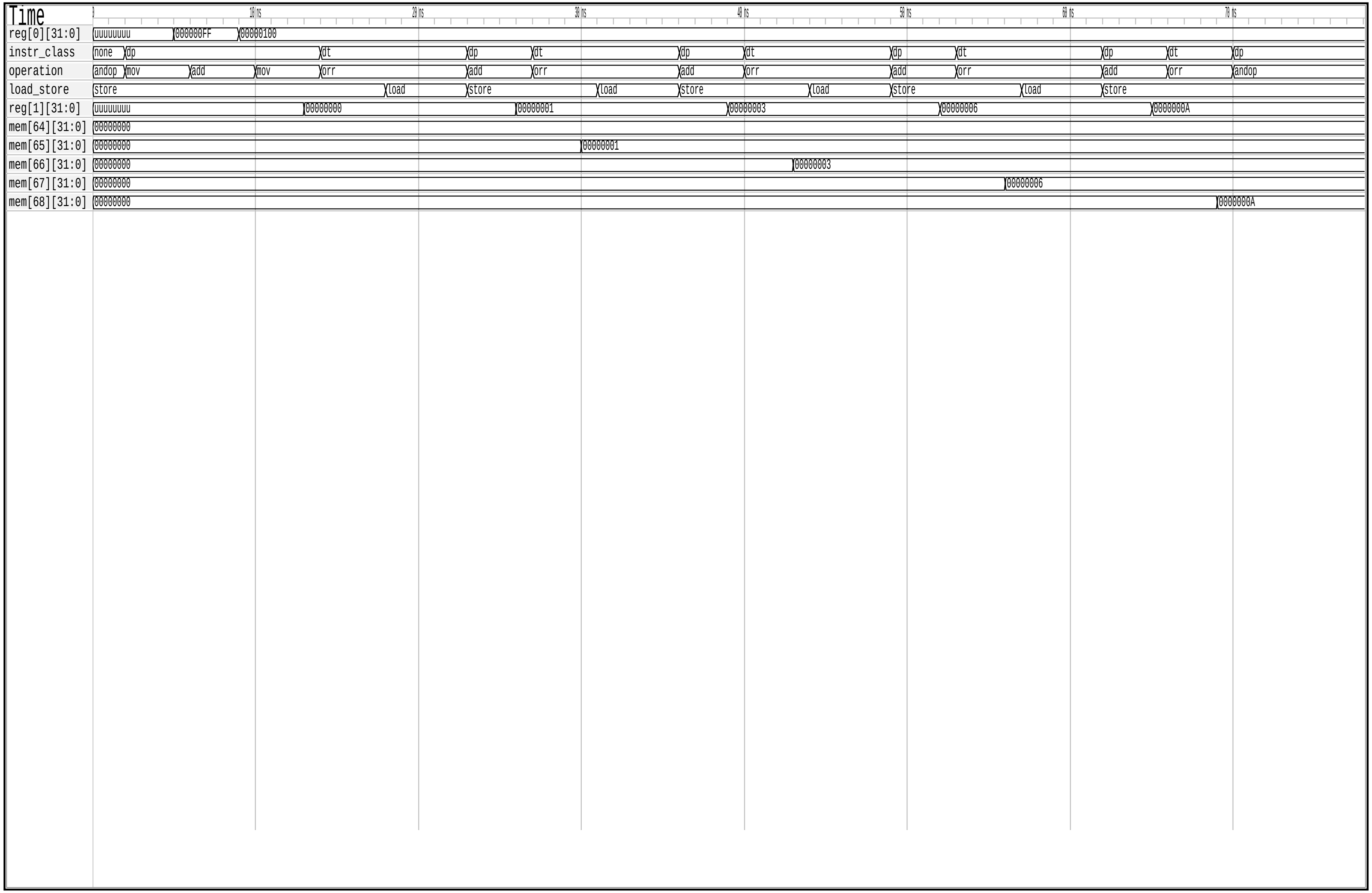
All the testbenches gave the expected results.

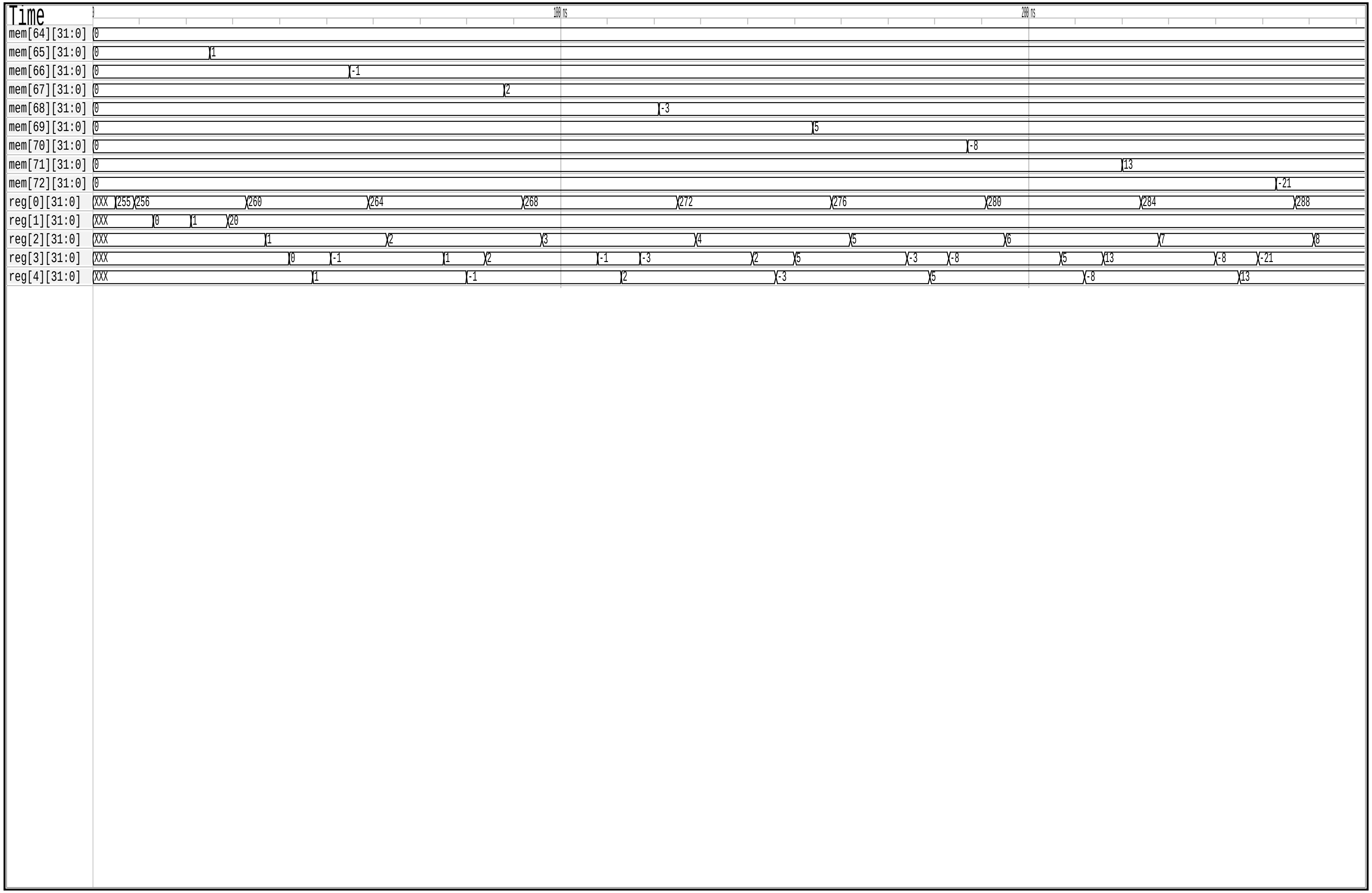
The output waveform of the testbench results can be viewed in .ghw and .pdf files in the output folder.

The pdf files of the simulation are also attached at the end of this report.

arm-tests/fact.pdf




Time
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
70 ns
reg[0][31:0]
instr_class
operation
load_store
reg[1][31:0]
mem[64][31:0]
mem[65][31:0]
mem[66][31:0]
mem[67][31:0]
mem[68][31:0]
uuuuuuuu 000000FF 00000100
none dp dt dp dt dp dt dp dt dp dt dp
andop mov add mov orr add orr add orr add orr add orr andop
store load store load store load store load store
uuuuuuuu 00000000 00000001 00000003 00000006 0000000A
00000000
00000000 00000001
00000000 00000003
00000000 00000006
00000000 0000000A


Time
100 ns
200 ns
mem[64][31:0] 0
mem[65][31:0] 0 1
mem[66][31:0] 0 -1
mem[67][31:0] 0 2
mem[68][31:0] 0 -3
mem[69][31:0] 0 5
mem[70][31:0] 0 -8
mem[71][31:0] 0 13
mem[72][31:0] 0 -21
reg[0][31:0] XXX 255 256 260 264 268 272 276 280 284 288
reg[1][31:0] XXX 0 1 20
reg[2][31:0] XXX 1 2 3 4 5 6 7 8
reg[3][31:0] XXX 0 -1 1 2 -1 -3 2 5 -3 -8 5 13 -8 -21
reg[4][31:0] XXX 1 -1 2 -3 5 -8 13