# Lab Assignment 2, Stage 7: Implementing all multiply group instructions

*Kushal Kumar Gupta, 2020CS10355*

Implemented instructions `MUL, MLA, UMULL, UMLAL, SMULL, SMLAL`.

## Files, entities and Architectures:

Changes from stage 6-

`I. Multiplier_stage7.vhd:`

```
entity Multiplier is
port(
    op1: in std_logic_vector(31 downto 0); --Inputs: operands 1, 2 and 3 if accumulate, type of
multiply instruction
    op2: in std_logic_vector(31 downto 0);
    op3: in std_logic_vector(63 downto 0);
    MULT_instr: MULT_instr_type;
    res: out std_logic_vector(63 downto 0) --Outputs: result of operation, carry out
    );
end Multiplier;
```

An independent multiply-accumulate module with 64-bit result designed for implementing all the six instructions of multiply group.

Implementation considerations -

- As described in the lecture slides, operands 1 and 2 are signed or unsigned extended to 33 bits and then signed multliplication of this is calculated. From the temporary 66 bit result, the lower 64 bits give the appropriate required result of the multiplication. In this, the accumulator value is added if required using a 64 bit adder.
- Even for MUL and MLA, the 64 bit result is calculated. The accumulate value (operand 3) is also 64 bit, and hence 0-extended for 32 bit accumulate values in MLA.
- It is a combinational circuit.

For more details regarding the implementation of `Multiplier`, see the architecture `Multiplier_beh`.

II.Processor_stage7.vhd:

Included component of Multiplier, and appropriate glue logic followed to incorporate this module. Also added another state of FSM for long multiply instructions. FlagUpdater update the Z and N flag for multiply instruction. New registers Y and ResHi added whose purpose is described below.

Implementation considerations -

- Multiply instructions go to state 10 from state 2. In state 10, register X is loaded with RF[IR[11-8]] as before and also register Y is loaded with RF[IR[15-12]].
- From state 3, multiply instructions transition to state 3. Here, if the instruction was DP, as earlier Res is loaded with alu_res. Now, if the instruction is multiply, Res is loaded with lower 32 bits of Multiplier_res and register ResHi is loaded with higher 32 bits of Multiplier_res.
- In state 6, Res is stored in register file appropriately. For DP instructions (logic and arithmetic which require writing back in register file), Res is stored in RF[IR[15-12]]. Also, for long multiply instructions Res has lower half of result and hence Res is stored in RF[IR[15-12]]. For short multiply instructions Res has the 32 bit required result and hence Res is stored in RF[IR[19-16]].
- DP and short multiply instructions transition to state 1 from state 6 but Long multiply instructions transition to the new state 12 from state 6 to write the higher 32 bits of result stored in ResHi to RF[IR[19-16]].
- Accumulate value (operand 3) for Multiplier has been given as 64 bits, hence for MLA it is zero extended to 64 bits.
- FlagUpdater is provided Multiplier_res also to update the Z and N flags appropritely for multiply instructions when S bit (bit 20) is set in the instruction.

For more details regarding the implementation of processor, see the architecture processor_beh.

III.mytypes_stage7.vhd:

Changed MUL to MULT in instr_class_type

```
type instr_class_type is (DP, DTtype00, DTtype01, MULT, BRN, none);
```

Introduced type MULT_instr_type

```
type MULT_instr_type is (MUL, MLA, UMULL, UMLAL, SMULL, SMLAL, invalidMULT);
```

IV.Decoder_stage7.vhd:

```
entity Decoder is
Port (
    instruction : in word;
    instr_class : out instr_class_type;
    operation : out optype;
    DP_subclass : out DP_subclass_type;
     DT_instr : out DT_instr_type;
     MULT_instr: out MULT_instr_type;
    DP_operand_src : out src_type;
    load_store : out load_store_type;
     pre_post : out pre_post_type;
     DT_write_back : out write_back_type;
    DT_offset_sign : out DT_offset_sign_type;
    DT_offset_src : out src_type;
    reg_shift_type : out shift_rotate_op_type;
    reg_shift_src : out src_type
);
end Decoder;
```

For 00 F field, instr_class is determined using

- DP if either bit 25 is 1 (immediate op2) or bit 25 is 0 and at least one of bit 4 and 7 is 0
- MULT when both bit 4 and 7 are 1 and SH is 00
- DT when both bit 4 and 7 are 1 and SH is 01, 10 or 11

`V.FlagUpdater_stage7.vhd`:

```
entity FlagUpdater is
Port (
    CLK: in bit;
    instr_class: in instr_class_type;
   DP_subclass : in DP_subclass_type;
    MULT_instr : in MULT_instr_type;
    Fset : in std_logic; --if this is 1 then at the rising edge of the clock, flags are updated
    carry_ALU, carry_shifter, MSBop1, MSBop2: in std_logic; --carry bit generated by ALU, carry
bit generated by shifter, MSB bits of operands of ALU (after considering + operation for arith
and comp subclass, for calculating V flag)
    res_ALU: in word; --result of ALU
    res_Multiplier : in std_logic_vector(63 downto 0);
    Z, V, C, N: out std_logic := '0'
);
end FlagUpdater;
```

Included support for multiply instructions, updates `Z` and `N` flags, decided by either the lower 32 bits of result if short multiply or the whole 64 bit result if long multiply.

# How to use:

On edaplayground.com, upload testbench.vhd and run.do in the left column, and ALU_stage3.vhd, RegFile_stage1.vhd, Mem_stage3.vhd, mytypes_stage7.vhd , decoder_stage7.vhd, FlagUpdater_stage7.vhd, conditionChecker_stage4.vhd, programCounter_stage3.vhd, processor_stage7.vhd, shifter_stage5.vhd, PMconnect_stage6.vhd, Multiplier_stage7.vhd. Copy contents in design.vhd section as given in the design.vhd file. Select Testbench + Design as VHDL. Then, for-

**1.)Simulation**

Type testbench in the Top entity. Select Aldec Riviera Pro 2020.04 simulator to simulate the design. Set the run time accordingly and select the EPWave option. Then Save and Run the simulation to get waves of the signals defined in these modules.

**2.)Synthesis**

Copy the VHDL file of the component you want to synthesise into design.vhd. (Only design.vhd entities are synthesised)

Then, select Mentor Precision 2021.1 to synthesise. Select netlist option to view the Verilog description. Then Save and Run to get the synthesis result. We get the report containing the resource table specifying number of IOs, LUTs, CLB slices, ports, nets, etc. used to implement this module in the given FPGA specified by run.do.

## Results of EPWave (Simulation):

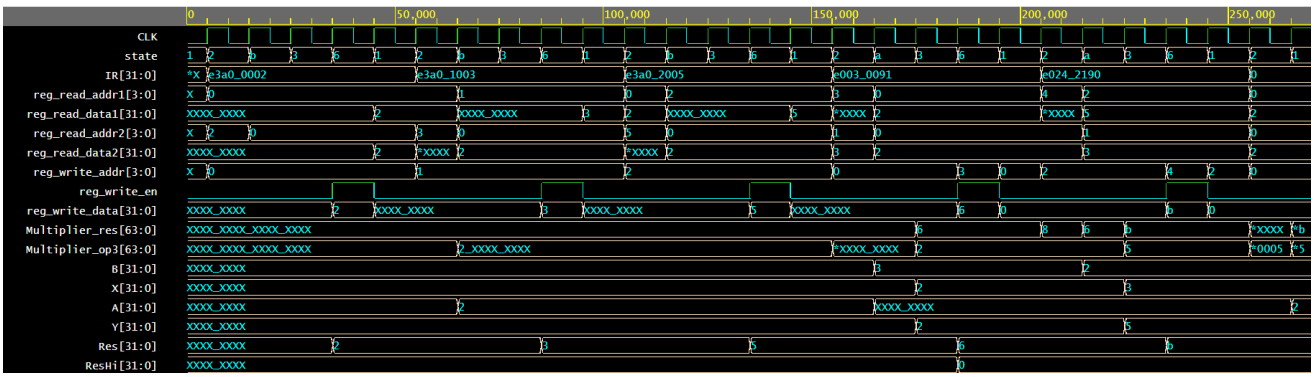
*NOTE: Please ensure that Program memory is from address 0 to 63, data memory is from 64 to 127*

Can see the input and output signals of processor against the clock.

Memory initialisation with different programs and EPWave-

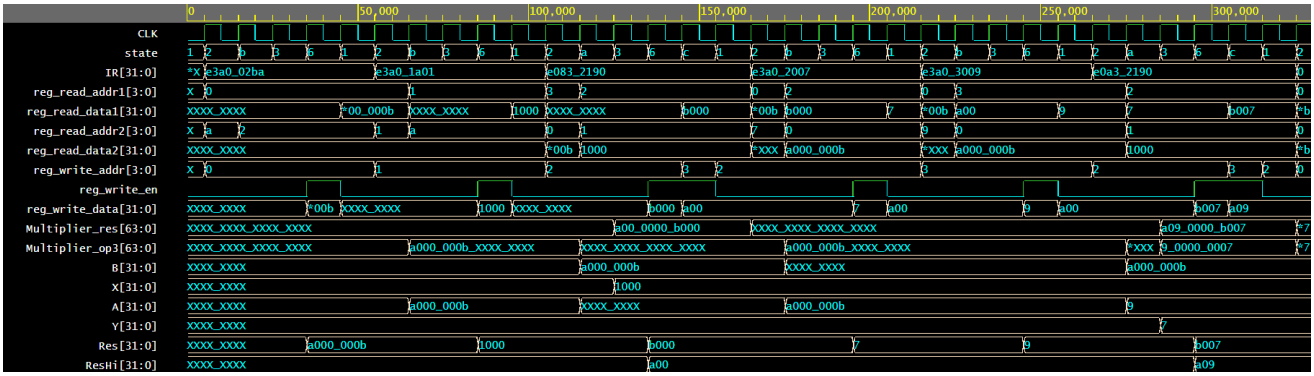
1.

```
signal Mem_space : type_mem :=
                              ( 0 => X"E3A00002",
                                1 => X"E3A01003",
                                2 => X"E3A02005",
                                3 => X"E0030091",
                                4 => X"E0242190",
                                others => X"00000000"
                                );

-- mov r0, #2
-- mov r1, #3
-- mov r2, #5
-- mul r3, r1, r0
-- mla r4, r0, r1, r2
```



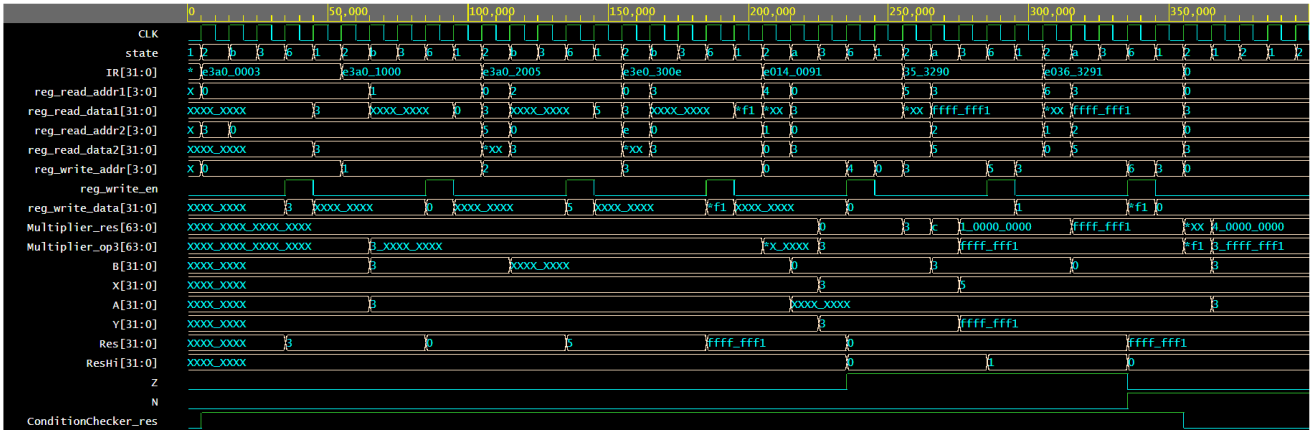
2.

```
        signal Mem_space : type_mem :=
                                  (  0 => X"E3A002BA",
                                     1 => X"E3A01A01",
                                     2=> X"E0832190",
                                     3 => X"E3A02007",
                                     4 => X"E3A03009",
                                     5 => X"E0A32190",
                                     others => X"00000000"
                                     );

    mov r0, #0xA000000B
    mov r1, #0x1000
    umull r2, r3, r0, r1 @result should be r3 = 00000A00 r2 = 0000B000
    mov r2, #7
    mov r3, #9
    umlal r2, r3, r0, r1 @result should be 00000A09 0000B007
```



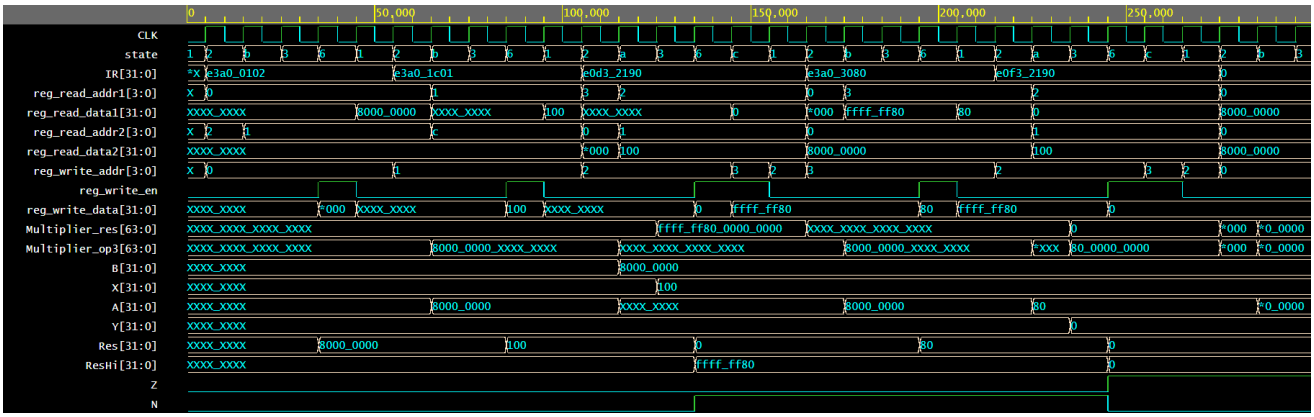
3.

```
    signal Mem_space : type_mem :=
                                ( 0 => X"E3A00003",
                                  1 => X"E3A01000",
                                  2 => X"E3A02005",
                                  3 => X"E3E0300E",
                                  4 => X"E0140091",
                                  5 => X"00353290",
                                  6 => X"E0363291",
                                  others => X"00000000"
                                  );

--  mov r0, #3
--  mov r1, #0
--  mov r2, #5
--  mov r3, #-15
--  muls r4, r1, r0
--  mlaeqs r5, r0, r2, r3
--  mlas r6, r1, r2, r3
```



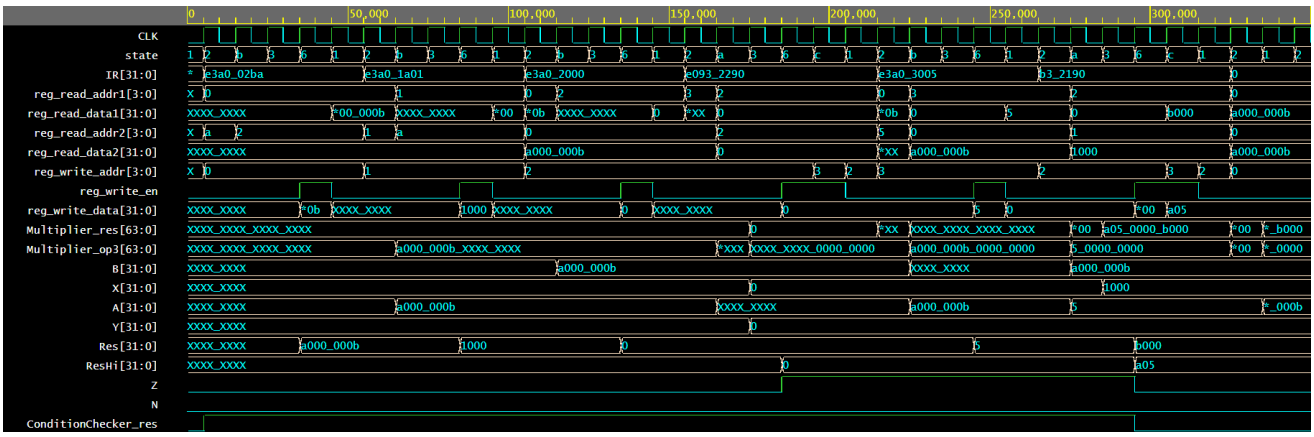
4.

```
    signal Mem_space : type_mem :=
                                ( 0 => X"E3A00102",
                                  1 => X"E3A01C01",
                                  2 => X"E0D32190",
                                  3 => X"E3A03080",
                                  4 => X"E0F32190",
                                  others => X"00000000"
                                  );

--  mov r0, #-0x80000000
--  mov r1, #0x100
--  smulls r2, r3, r0, r1 @result is -2^31 * 256 = FFFF FF80 00000000
--  mov r3, #-0xFFFFFF80
--  smlals r2, r3, r0, r1 @ result is 0
```



5.

```
    signal Mem_space : type_mem :=
                                ( 0 => X"E3A002BA",
                                  1 => X"E3A01A01",
                                  2=> X"E3A02000",
                                  3 => X"E0932290",
                                  4 => X"E3A03005",
                                  5 => X"00B32190",
                                  others => X"00000000"
                                  );

--  mov r0, #0xA000000B
--  mov r1, #0x1000
--  mov r2, #0
--  umulls r2, r3, r0, r2 @result should be 0
--  mov r3, #5
--  umlaleqs r2, r3, r0, r1 @result should be r3 = 00000A05 r2 = 0000B000
```



**2.) Resource Table (Synthesis)**

Can see the resources used by the different module implementations on given FPGA.

For example, for Multiplier -

```
# Info: Resource                          Used    Avail   Utilization
# Info: ---------------------------------------------------------------
# Info: IOs                               199     210      94.76%
# Info: Global Buffers                    0       32        0.00%
# Info: LUTs                              131     63400     0.21%
# Info: CLB Slices                        33      15850     0.21%
# Info: Dffs or Latches                   0       126800    0.00%
# Info: Block RAMs                        0       135       0.00%
# Info: DSP48E1s                          4       240       1.67%
# Info: ---------------------------------------------------------------
# Info: ************************************************************************
# Info: Library: work    Cell: Multiplier    View: Multiplier_beh
# Info: ************************************************************************
# Info:  Number of ports :                          199
# Info:  Number of nets :                           838
# Info:  Number of instances :                      463
# Info:  Number of references to this view :          0
# Info: Total accumulated area :
# Info:  Number of DSP48E1s :                         4
# Info:  Number of LUTs :                           131
# Info:  Number of Primitive LUTs :                 131
# Info:  Number of MUX CARRYs :                      63
# Info:  Number of accumulated instances :          463
# Info: ****************************
```