# COL 216 Assignment 2.7

Chinmay Mittal ( 2020CS10336 )

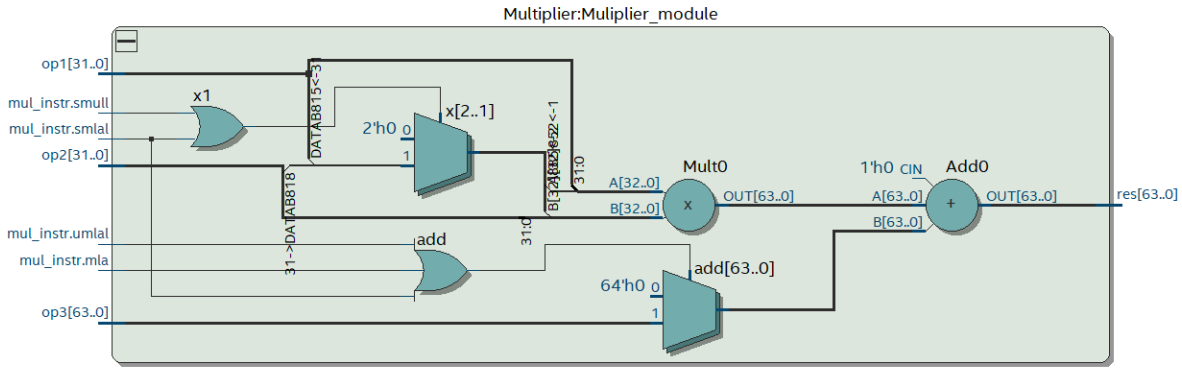
In this assignment, we add support for multiply functionality to the multicycle processor.
For this, I introduce a new component *Multiplier.*
I have added additional logic to the Decoder to determine the type of multiply instruction or if the instruction is not a multiply instruction for which a type is also added to *MyTypes*

```
    type mul_instr_type is ( mul, mla, umull, umlal, smull, smlal , nomul ) ;

-- new logic for figuring out type of multiplication instruction
mul_instr <= mul   when ( instruction( 7 downto 4 ) = "1001" and instruction( 27 downto 21 ) = "0000000") else
             mla   when ( instruction( 7 downto 4 ) = "1001" and instruction( 27 downto 21 ) = "0000001" ) else
             umull when ( instruction( 7 downto 4 ) = "1001" and instruction( 27 downto 21 ) = "0000100" ) else
             umlal when ( instruction( 7 downto 4 ) = "1001" and instruction( 27 downto 21 ) = "0000101" )else
             smull when ( instruction( 7 downto 4 ) = "1001" and instruction( 27 downto 21 ) = "0000110" ) else
             smlal when ( instruction( 7 downto 4 ) = "1001" and instruction( 27 downto 21 ) = "0000111" ) else
             nomul ;
```

The *Multiplier* takes in the type of the multiply instruction as input, two 32 bit operands and a 64-bit operand. The 32-bit operands are multiplied ( according to the type of the multiplication as specified in Lecture 9 ), and if the instruction is an accumulate instruction then the 64-bit number is added to the multiplication result ( in the case of MLA, the least significant 32 bits are important and are zero-extended to 64 bits before being fed to the multiplier)

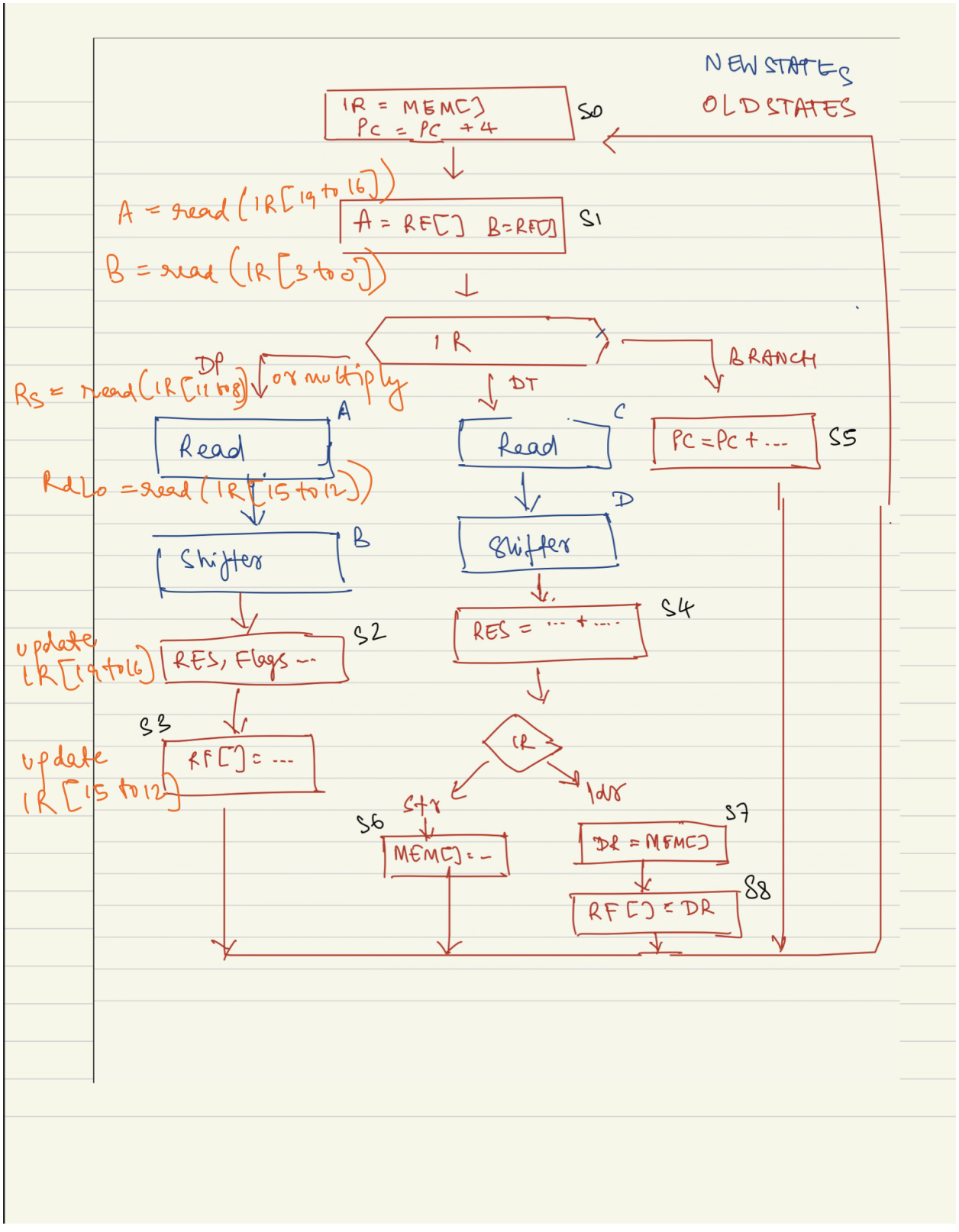
NETLIST output for multiplier



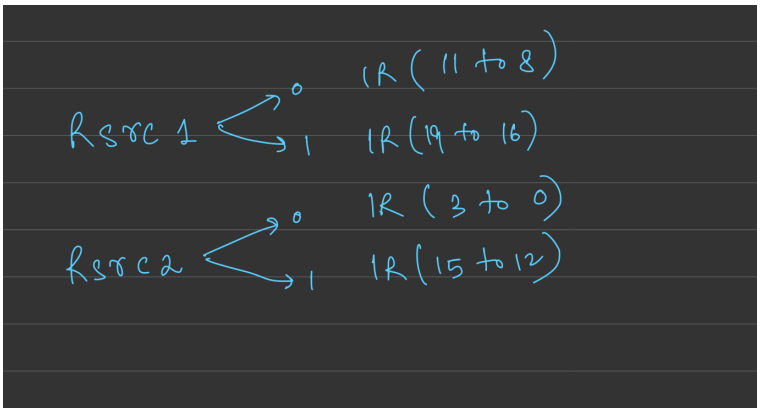

Resource utilization for multiplier

```
# Info: ************************************************************
# Info: Device Utilization for 7A100TCSG324
# Info: ************************************************************
# Info: Resource                          Used    Avail   Utilization
# Info: ------------------------------------------------------------
# Info: IOs                               199     210      94.76%
# Info: Global Buffers                    0       32        0.00%
# Info: LUTs                              65      63400     0.10%
# Info: CLB Slices                        16      15850     0.10%
# Info: Dffs or Latches                   0       126800    0.00%
# Info: Block RAMs                        0       135       0.00%
# Info: DSP48E1s                          4       240       1.67%
# Info: ------------------------------------------------------------
# Info: ************************************************************
```

At max four registers are required to be read for multiply and two registers are required to be written to. I do not introduce any new states and utilize the states involved in one cycle of DP instruction to perform all these operations as illustrated by my state transition diagram for the control FSM:



Rs was a register that stored the shift amount but since multiply instructions do not have any shift operation, we use the same register to store the value specified by IR( 11 to 8 ) used for multiply operations.
RdLo is a new register that is added along with its control signal and stores the value in the register specified by IR( 15 downto 12 ).
To enable the use states differently depending on whether the instruction is a DP instruction or a multiply instruction, we pass a signal *isMultiplication* and the type of multiply operations *mul_instr.*
We use these signals to decide what to do in states where both DP operations and multiply operations are possible.
Some additional control logic and control signals are required to access the register files and to write data to the Register File.
To access the register file the following control signals are used to specify the read addresses for both ports:



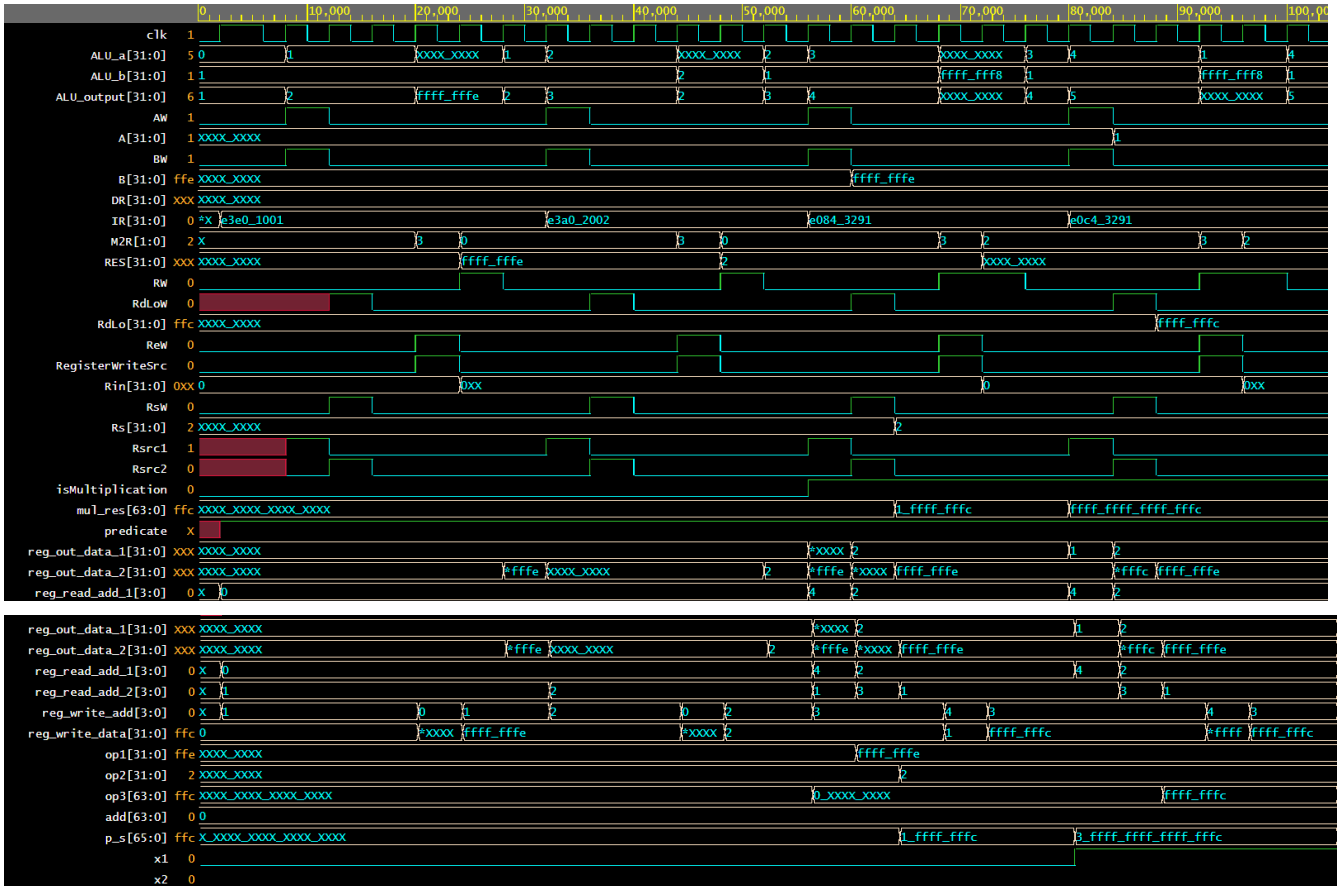

Previously M2R was used to control the write data for the Register File. M2R has now been made a 2-bit vector and further logic has been added to support write of data from ALU result / Memory / Output of multiplication of module.

```
-- additional logic added to write output of multiplier to registers
reg_write_data <= Rin when M2R = "01" else -- Rin comes from Pmconnect ( ie Memory )
                  RES when M2R = "00" else
                  mul_res( 31 downto 0 ) when M2R = "10" else
                  mul_res( 63 downto 32 ) when M2R = "11"  else
                  x"00000000" ;
```

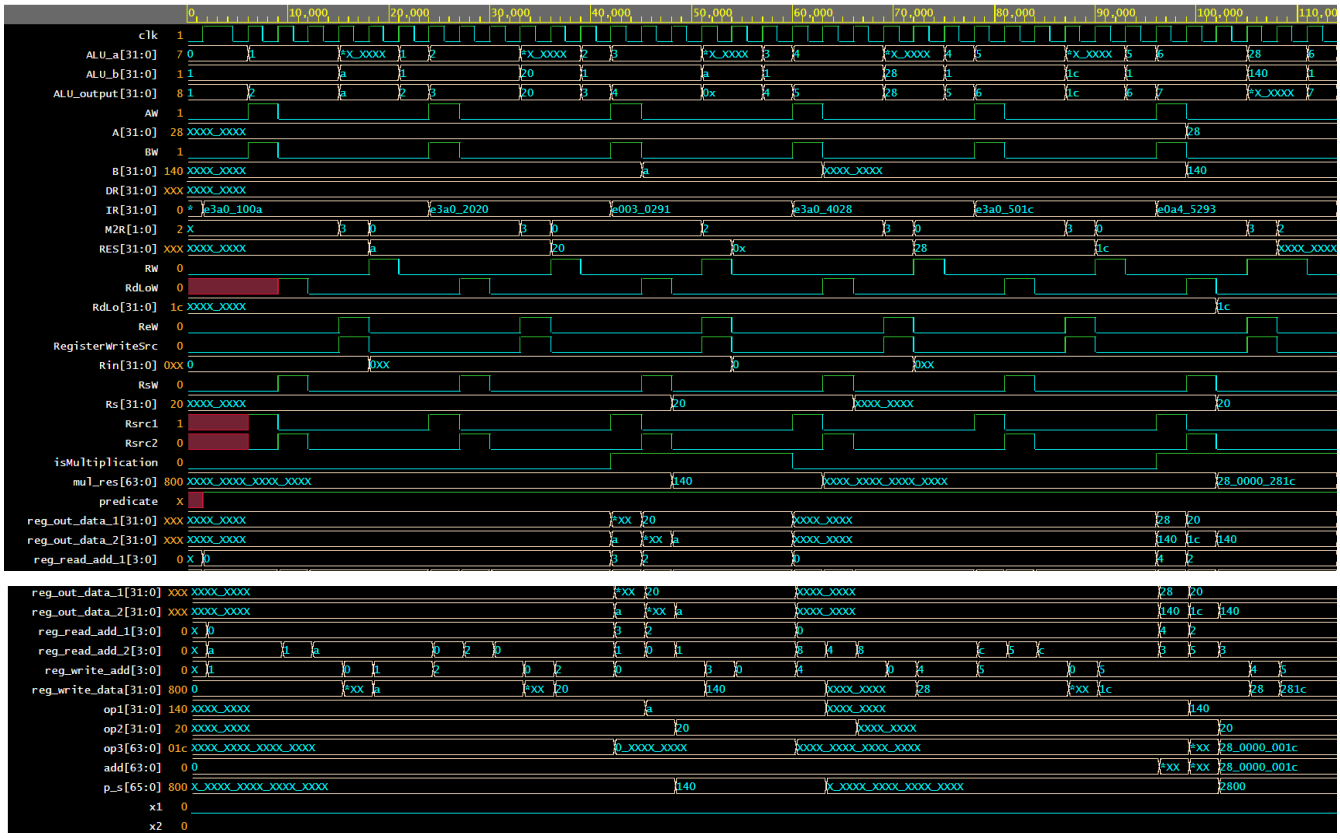
Test1.s


0
10,000
20,000
30,000
40,000
50,000
clk
ALU_a[31:0]
ALU_b[31:0]
ALU_output[31:0]
AW
A[31:0]
BW
B[31:0]
DR[31:0]
IR[31:0]
M2R[1:0]
RES[31:0]
RW
RdLoW
RdLo[31:0]
ReW
RegisterWriteSrc
Rin[31:0]
RsW
Rs[31:0]
Rsrc1
Rsrc2
isMultiplication
mul_res[63:0]
predicate
reg_out_data_1[31:0]
reg_out_data_2[31:0]
reg_read_add_1[3:0]
reg_read_add_2[3:0]
reg_write_add[3:0]
reg_write_data[31:0]
op1[31:0]
op2[31:0]
op3[63:0]
add[63:0]
p_s[65:0]
e3a0_1003
e3a0_2002
e3a0_3001
e024_3291

Test2.s


clk
ALU_a[31:0]
ALU_b[31:0]
ALU_output[31:0]
AW
A[31:0]
BW
B[31:0]
DR[31:0]
IR[31:0]
M2R[1:0]
RES[31:0]
RW
RdLoW
RdLo[31:0]
ReW
RegisterWriteSrc
Rin[31:0]
RsW
Rs[31:0]
Rsrc1
Rsrc2
isMultiplication
mul_res[63:0]
predicate
reg_out_data_1[31:0]
reg_out_data_2[31:0]
reg_read_add_1[3:0]
reg_read_add_2[3:0]
reg_write_add[3:0]
reg_write_data[31:0]
op1[31:0]
op2[31:0]
op3[63:0]
add[63:0]
p_s[65:0]
x1
x2

Test3.s


clk
ALU_a[31:0]
ALU_b[31:0]
ALU_output[31:0]
AW
A[31:0]
BW
B[31:0]
DR[31:0]
IR[31:0]
M2R[1:0]
RES[31:0]
RW
RdLoW
RdLo[31:0]
ReW
RegisterWriteSrc
Rin[31:0]
RsW
Rs[31:0]
Rsrc1
Rsrc2
isMultiplication
mul_res[63:0]
predicate
reg_out_data_1[31:0]
reg_out_data_2[31:0]
reg_read_add_1[3:0]
reg_out_data_1[31:0]
reg_out_data_2[31:0]
reg_read_add_1[3:0]
reg_read_add_2[3:0]
reg_write_add[3:0]
reg_write_data[31:0]
op1[31:0]
op2[31:0]
op3[63:0]
add[63:0]
p_s[65:0]
x1
x2