1: ALU

a FULL_ADDER

Description

To implement my FULL_ADDER, I connect two half_adders. The full_adder has three one bit inputs "A", "B", "Cin", and two one bit outputs "Sum" and "Carry". "Sum" is the result of two bits addition, and "Carry" is the carry out.

Waveform

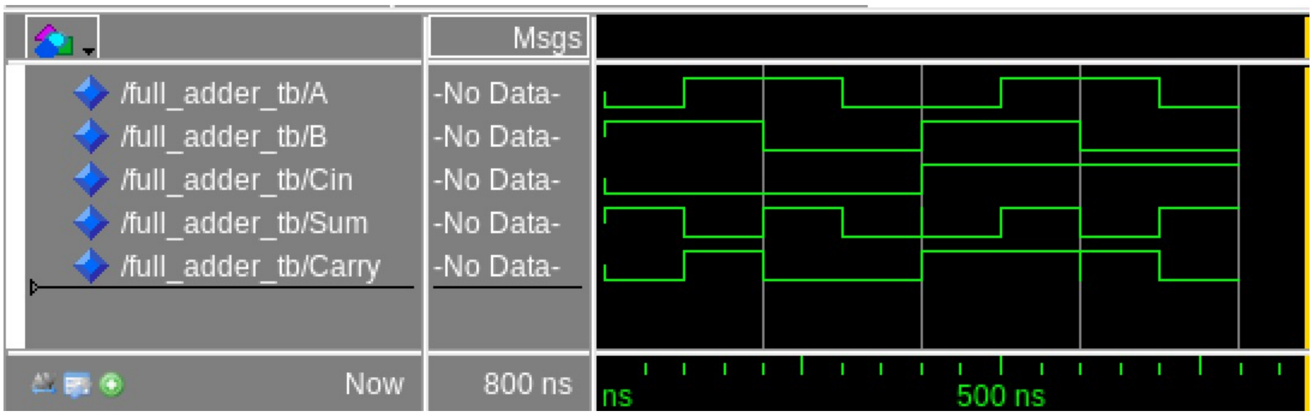


Figure 1: Simulation Waveform of AND2

A	B	Cin	Sum	Carry
0	1	0	1	0
1	1	0	0	1
1	0	0	1	0
0	0	0	0	0
0	1	1	0	1
1	1	1	1	1
1	0	1	0	1
0	0	1	1	0

Table 1: FULL_ADDER output table

From the above figure and table, we can see all the addition operations are working correctly.

b 64 bits adder

Description

To implement my 64 bits adder, I connect 64 full adders into a 64 bits carry ripple adder. The carry ripple adder has an one bit input "Cin", two 64 bits inputs "A" and "B", an one bit output "Cout", and one 64 bits output "Sum". "Sum" is the result of our addition, and "Cout" is the carry out.

Waveform

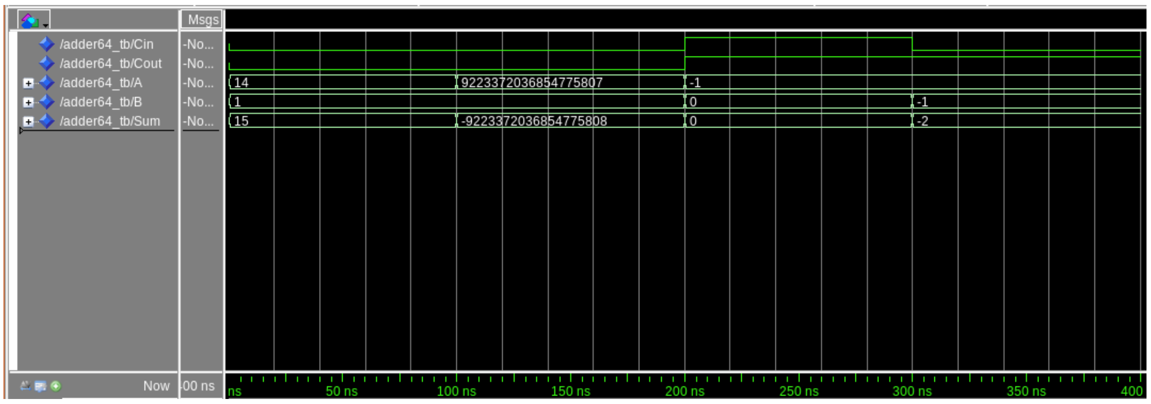


Figure 2: Simulation Waveform of 64bits carry ripple adder

In the waveform, I did 4 additions. 64 bits signals are shown in decimal radix. For those cases do not have overflow issue, all the results are correct. Although we do not care carry out in these cases, the carry out signals are all correct if we treat these signals as binary.

c ALU

Description

ALU runs different arithmetic logic operation according to the operation code. ALU has two 64 bits inputs, when 'operation' = "0000", it do "A and B". When 'operation' = "0001", it does "A or B". When 'operation' = "0010", it does signed addition "A + B". And I use my 64 bits carry ripple adder to implement this addition operation. When 'operation' = "0110", it does signed subtraction "A - B".
If 'operation' equals to other value, currently ALU does nothing.
'zero' is used to check the result, if result equals to '0', then 'zero' is set to '1', otherwise 'zero' is '0'. 'overflow' is '1' when overflow happens.

'AND' 'OR' Waveform

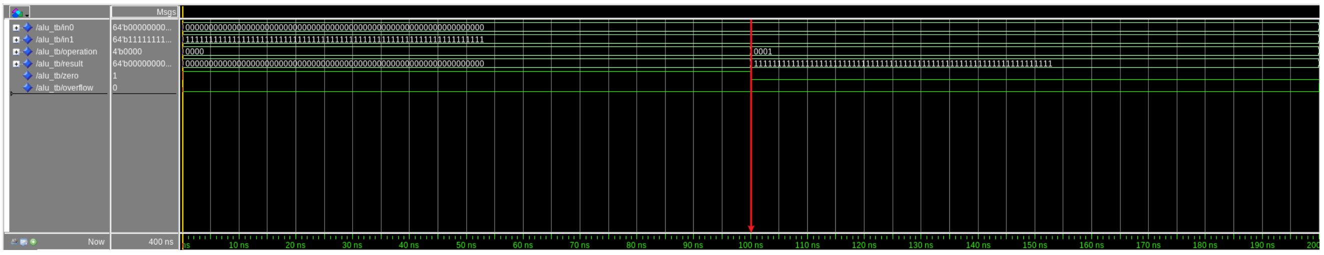


Figure 3: Simulation Waveform of 'AND' and 'OR'(include one zero case)

In the above waveform, I set A to 64 bits '0', B to 64 bits '1'. During 0 to 100 ns, the operation is "0000" which means "AND" function. The output of "A AND B" is 64 bits '0'. Since the output is zero, the "zero" is set to '1'. During 100 to 200 ns, the operation is "0001" which means "OR" function. The output of "A OR B" is 64 bits '1'.

'add' 'subtract' Waveform

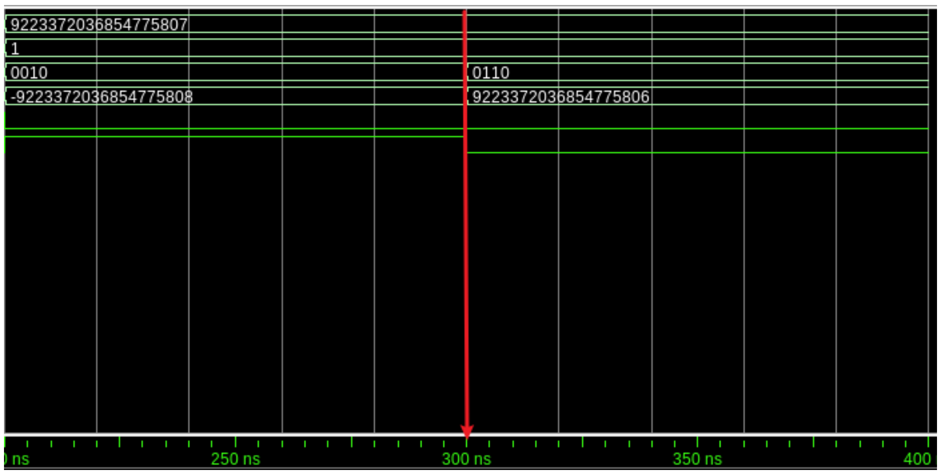


Figure 4: Simulation Waveform of 'ADD' and 'SUB'

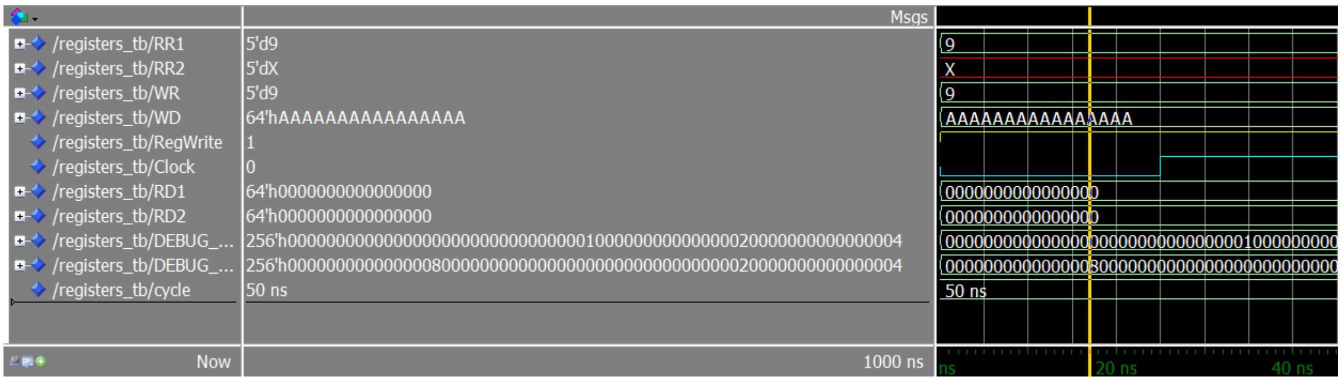
In the above waveform, I set A to '1'(63 bits '0')=9223372036854775807, B to (63 bits '0')'1'=1. During 200 to 300 ns, the operation is "0010" which means "ADD" function. The output of "A + B" is -9223372036854775808. Overflow occurs here, and the "overflow" signal is '1'. During 300 to 400 ns, the operation is "0110" which means "SUBTRACT" function. The output of "A - B" is 9223372036854775806. All the outputs are correct.

2: Registers

Description

We can write data into registers, and we can read data from registers. These are the two functions of registers. Register has three 5 bits inputs to declare the addresses of registers to be read/written. 'RR1', and 'RR2' are the addresses of the registers to be read, 'WR' is the address of registers to be written.
Write operation only happens on the negative edge of 'Clock' and 'RegWrite' = '1'. The 'Clock' cycle is 50 ns in my testbench. Registers get the write in data from 'WD', and write it into register at address 'WR'.
Read operation doesn't depend on 'Clock', it reads all the time. The value of 'RR1' and 'RR2' are correspondingly saved into registers at addresses 'RD1' and 'RD2'.
Since the length of register address is 5 bits, there are thirty-two 64 bits registers. For the initialization, I set reg(9), reg(10), reg(11), reg(12), reg(13), reg(14), reg(15), reg(19), reg(20), reg(21), reg(22), reg(23), reg(24), reg(25), reg(26), and reg(27) are set as required. reg(31), XZR, and all the other registers are set to 0.
Attempting to write register XZR is not allowed, there will be an error "Invalid register for writing".

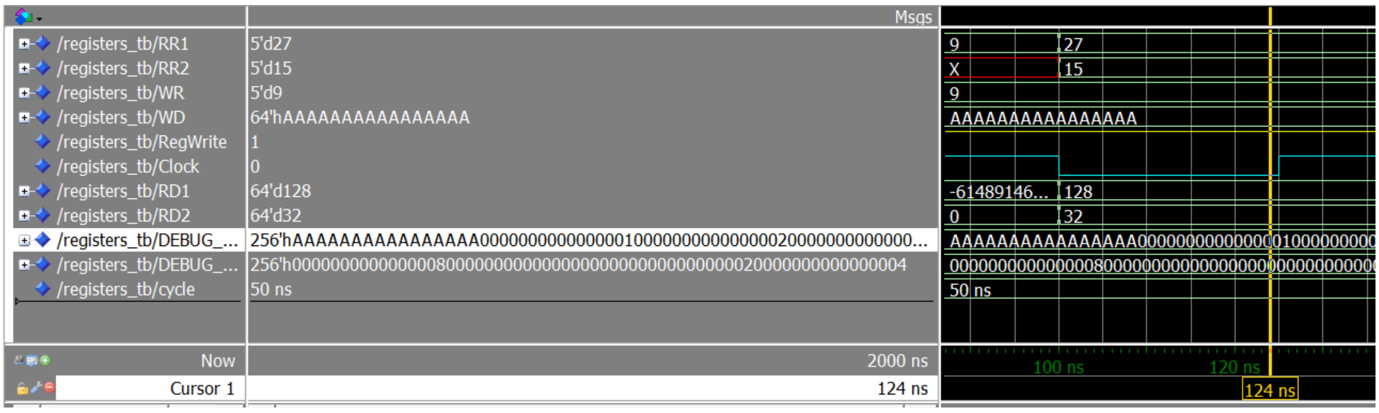
Waveforms



```
/registers_tb/RR1        5'd9
/registers_tb/RR2        5'dX
/registers_tb/WR         5'd9
/registers_tb/WD         64'hAAAAAAAAAAAAAAAA
/registers_tb/RegWrite   1
/registers_tb/Clock      0
/registers_tb/RD1        64'h0000000000000000
/registers_tb/RD2        64'h0000000000000000
/registers_tb/DEBUG_...  256'h0000000000000000000000000000000100000000000000020000000000000004
/registers_tb/DEBUG_...  256'h0000000000000000800000000000000000000000000000020000000000000004
/registers_tb/cycle      50 ns
Now                      1000 ns
```

Figure 5: Simulation Waveform of register initialization

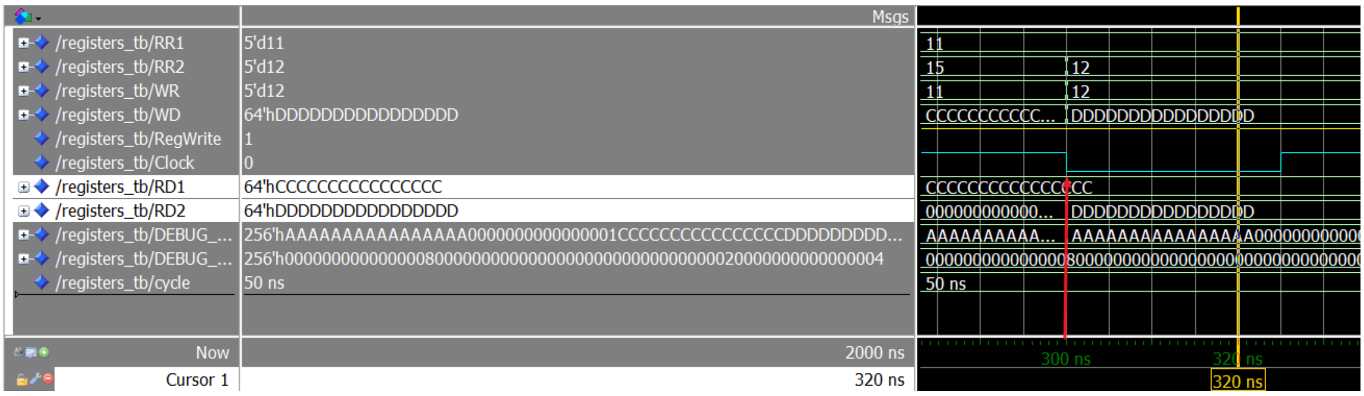
After initialization, I checked two DEBUG signals, they are correct.



	Msgs
/registers_tb/RR1	5'd27
/registers_tb/RR2	5'd15
/registers_tb/WR	5'd9
/registers_tb/WD	64'hAAAAAAAAAAAAAAAA
/registers_tb/RegWrite	1
/registers_tb/Clock	0
/registers_tb/RD1	64'd128
/registers_tb/RD2	64'd32
/registers_tb/DEBUG_...	256'hAAAAAAAAAAAAAAAA0000000000000001000000000000000200000000000000...
/registers_tb/DEBUG_...	256'h0000000000000000800000000000000000000000000000002000000000000004
/registers_tb/cycle	50 ns
Now	2000 ns
Cursor 1	124 ns

Figure 6: Simulation Waveform of register read

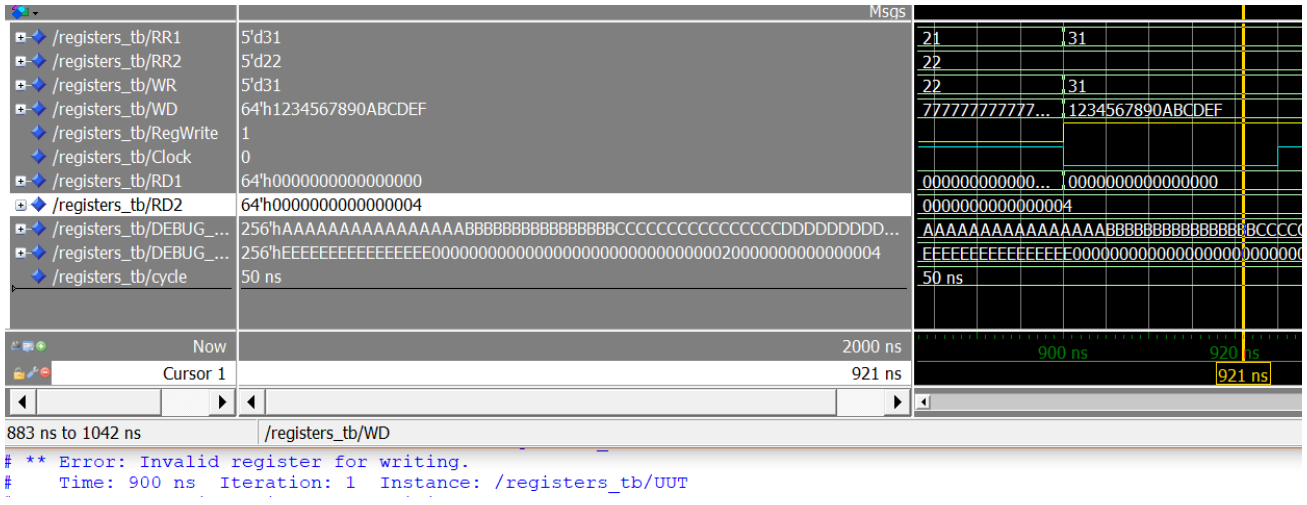
At 100 ns, I set RR1 = "11011", RR2 = "01111". And we get RD1 = \$X27 = 128, RD2 = \$X15 = 32 instantly.



	Msgs
/registers_tb/RR1	5'd11
/registers_tb/RR2	5'd12
/registers_tb/WR	5'd12
/registers_tb/WD	64'hDDDDDDDDDDDDDDDD
/registers_tb/RegWrite	1
/registers_tb/Clock	0
/registers_tb/RD1	64'hCCCCCCCCCCCCCCCC
/registers_tb/RD2	64'hDDDDDDDDDDDDDDDD
/registers_tb/DEBUG_...	256'hAAAAAAAAAAAAAAAA0000000000000001CCCCCCCCCCCCCCCCDDDDDDDDD...
/registers_tb/DEBUG_...	256'h0000000000000000800000000000000000000000000000002000000000000004
/registers_tb/cycle	50 ns
Now	2000 ns
Cursor 1	320 ns

Figure 7: Simulation Waveform of register write

At 300 ns, this is a falling edge of 'Clock', and RegWrite = '1'. I set WR = "01100" = 12, RR2 = "01100" = 12, WD = all 'D's. And we can see the value of \$X12 is changed to "DDDDDDDDDDDDDDDD" at this negative edge of 'Clock'.



```
                              Msgs
/registers_tb/RR1          5'd31                     21          31
/registers_tb/RR2          5'd22                     22
/registers_tb/WR           5'd31                     22          31
/registers_tb/WD           64'h1234567890ABCDEF      777777777777...  1234567890ABCDEF
/registers_tb/RegWrite     1
/registers_tb/Clock        0
/registers_tb/RD1          64'h0000000000000000      000000000000...  0000000000000000
/registers_tb/RD2          64'h0000000000000004      0000000000000004
/registers_tb/DEBUG_...    256'hAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCDDDDDDDDD...   AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBCCCCC
/registers_tb/DEBUG_...    256'hEEEEEEEEEEEEEEEE00000000000000000000000000000000200000000000000004   EEEEEEEEEEEEEEEE000000000000000000000000
/registers_tb/cycle        50 ns                     50 ns

Now                        2000 ns                   900 ns          920 ns
Cursor 1                   921 ns                                    921 ns

883 ns to 1042 ns          /registers_tb/WD
```

```
# ** Error: Invalid register for writing.
#    Time: 900 ns  Iteration: 1  Instance: /registers_tb/UUT
```

Figure 8: Simulation Waveform of register write to XZR

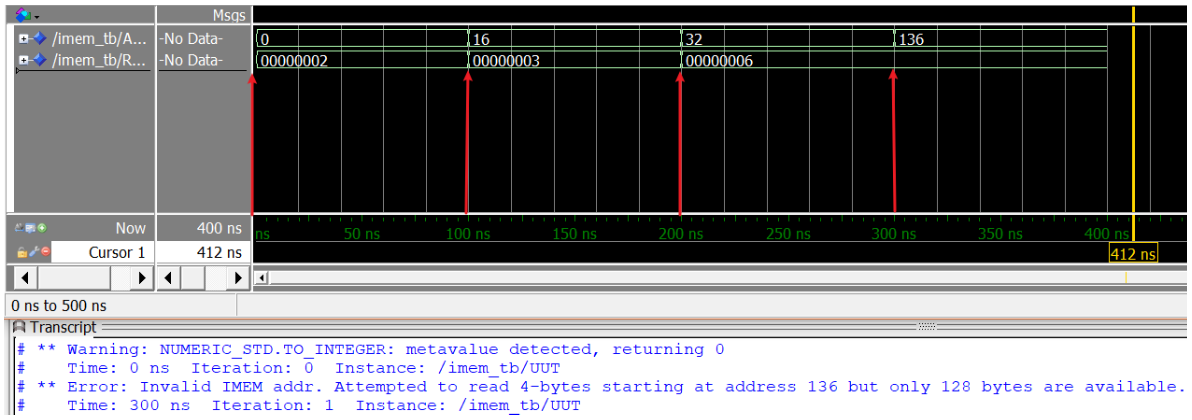
At 900 ns, I tried to write "0x12345667890ABCDEF" to $X31, and this is a falling edge of 'Clock', and RegWrite = '1'. But the value of $X31 stay at 0. And there is an error report "Invalid register for writing" at 900 ns.

3: IMEM

Description

Instruction memory continuously reads instructions, and outputs the read data. IMEM reads from 'Address' and 'ReadData' shows the four bytes data start from that address. I've set imemBytes(0) = "00000010", imemBytes(16) = "00000011", imemBytes(32) = "00000110", and all the other bytes are 8 bits '0'.

Waveforms



```
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
#    Time: 0 ns  Iteration: 0  Instance: /imem_tb/UUT
# ** Error: Invalid IMEM addr. Attempted to read 4-bytes starting at address 136 but only 128 bytes are available.
#    Time: 300 ns  Iteration: 1  Instance: /imem_tb/UUT
```

Figure 9: Simulation Waveform of instruction memory

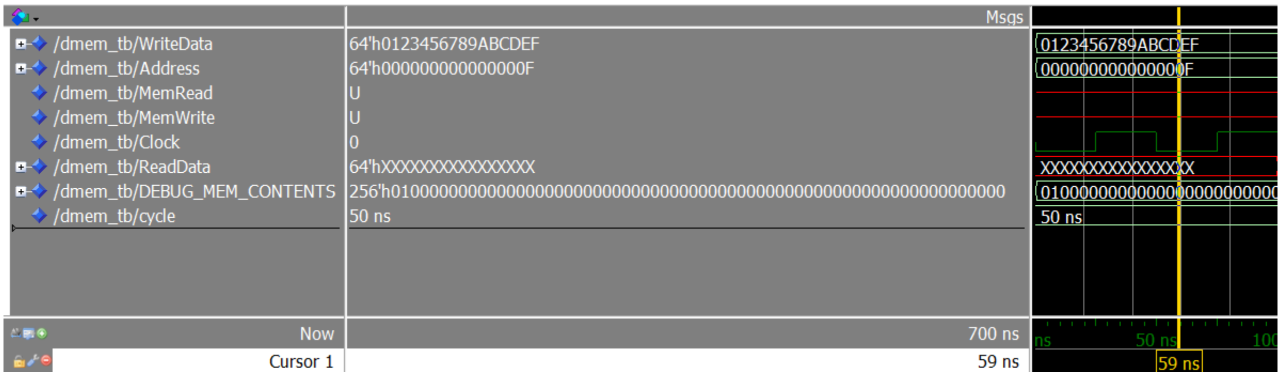
At 0 ns, IMEM reads 4 bytes start from byte(0). The 'ReadData' is "0x00000002". At 100 ns, IMEM reads 4 bytes start from byte(16). The 'ReadData' is "0x00000003". At 200 ns, IMEM reads 4 bytes start from byte(32). The 'ReadData' is "0x00000006". These three cases match my initialization. At 300 ns, IMEM tries to read 4 bytes start from byte(136), we got an error. It fails, because the length of my IMEM is 128 bytes, and byte(136) is not available.

4: DMEM

Description

Data memory can do two operations, read and write. The size of my data memory is 1024KB. It does read operation when ’MemRead’ = ’1’, ’MemWrite’ = ’0’. It does write operation when ’MemRead’ = ’0’, ’MemWrite’ = ’1’, and rising edge of ’Clock’. For initialization, only Byte(0) = ’1’, Byte(41) = ’1’, and all the other Bytes are ’0’.

Waveforms

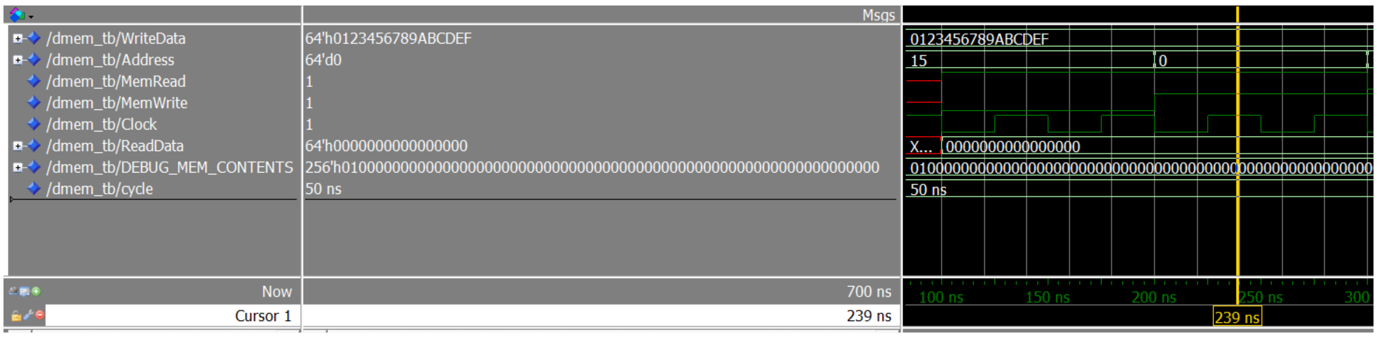


```
                                Msgs
/dmem_tb/WriteData              64'h0123456789ABCDEF
/dmem_tb/Address                64'h000000000000000F
/dmem_tb/MemRead                U
/dmem_tb/MemWrite               U
/dmem_tb/Clock                  0
/dmem_tb/ReadData               64'hXXXXXXXXXXXXXXXX
/dmem_tb/DEBUG_MEM_CONTENTS     256'h0100000000000000000000000000000000000000000000000000000000000000
/dmem_tb/cycle                  50 ns

Now       700 ns
Cursor 1  59 ns
```

Figure 10: Simulation Waveform of data memory initialization

Data memory is initialized, ’MemRead’ and ’MemWrite’ have no value, DMEM didn’t do anything. The DEBUG_MEM_CONTENTS matches the initialization.

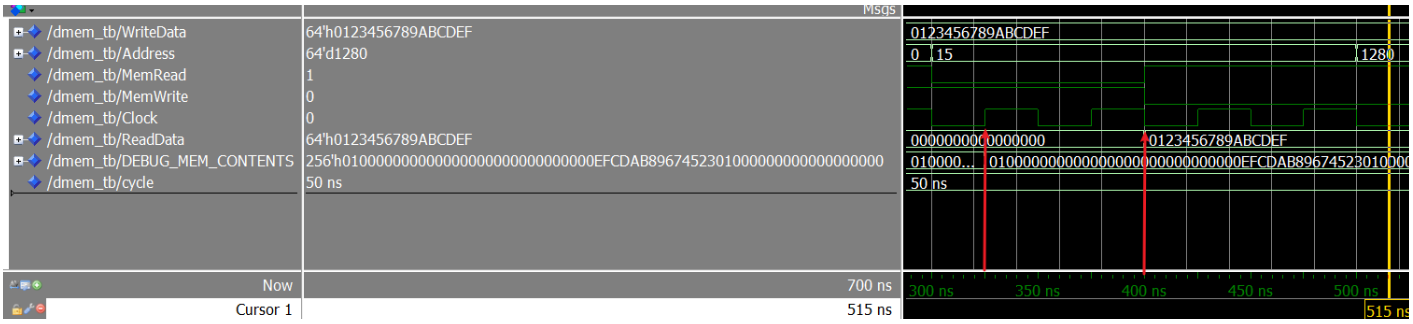


```
                                Msgs
/dmem_tb/WriteData              64'h0123456789ABCDEF
/dmem_tb/Address                64'd0
/dmem_tb/MemRead                1
/dmem_tb/MemWrite               1
/dmem_tb/Clock                  1
/dmem_tb/ReadData               64'h0000000000000000
/dmem_tb/DEBUG_MEM_CONTENTS     256'h0100000000000000000000000000000000000000000000000000000000000000
/dmem_tb/cycle                  50 ns

Now       700 ns
Cursor 1  239 ns
```

Figure 11: Simulation Waveform of data memory read

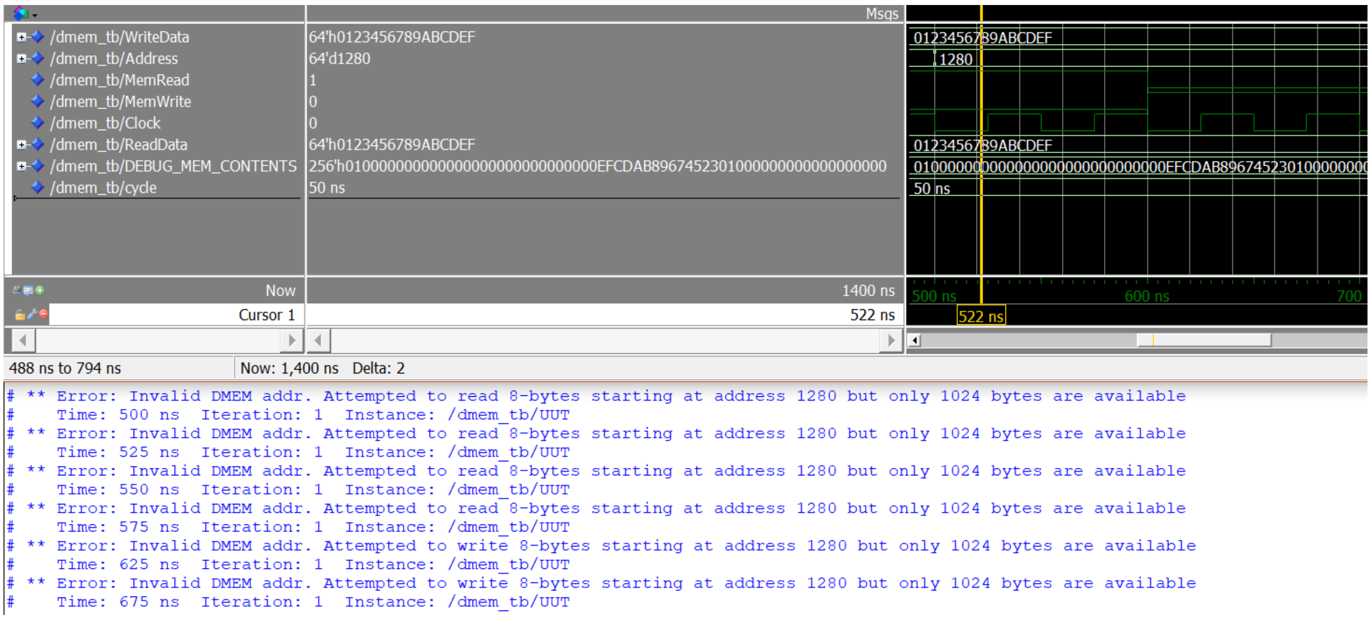
At 100 ns, ’MemRead’ = ’1’, ’MemWrite’ = ’0’, ’Address’ = 15. DMEM reads data from Bytes(15). At 200 ns, ’MemRead’ = ’1’, ’MemWrite’ = ’1’, ’Address’ = 0. If read operation is successful, we should get ”0x0000000000000001” from ’ReadData’. DMEM correctly does nothing when both ’MemRead’ and ’MemWrite’ = ’1’.



```
/dmem_tb/WriteData            64'h0123456789ABCDEF
/dmem_tb/Address              64'd1280
/dmem_tb/MemRead              1
/dmem_tb/MemWrite             0
/dmem_tb/Clock                0
/dmem_tb/ReadData             64'h0123456789ABCDEF
/dmem_tb/DEBUG_MEM_CONTENTS   256'h010000000000000000000000000000EFCDAB89674523010000000000000000
/dmem_tb/cycle                50 ns
Now                           700 ns
Cursor 1                      515 ns
```

Figure 12: Simulation Waveform of data memory write

At 350 ns, 'MemRead' = '0', 'MemWrite' = '1', 'Address' = 15, and there is a rising edge of Clock. From 'DEBUG_MEM_CONTENTS', we can tell data are correctly written into Byte(15) instantly. At 400 ns, 'MemRead' = '1', 'MemWrite' = '0', 'Address' = 15, 'ReadData' correctly give us the data we just wrote in.



```
/dmem_tb/WriteData            64'h0123456789ABCDEF
/dmem_tb/Address              64'd1280
/dmem_tb/MemRead              1
/dmem_tb/MemWrite             0
/dmem_tb/Clock                0
/dmem_tb/ReadData             64'h0123456789ABCDEF
/dmem_tb/DEBUG_MEM_CONTENTS   256'h010000000000000000000000000000EFCDAB89674523010000000000000000
/dmem_tb/cycle                50 ns
Now                           1400 ns
Cursor 1                      522 ns
488 ns to 794 ns      Now: 1,400 ns   Delta: 2
```

```
# ** Error: Invalid DMEM addr. Attempted to read 8-bytes starting at address 1280 but only 1024 bytes are available
#    Time: 500 ns  Iteration: 1  Instance: /dmem_tb/UUT
# ** Error: Invalid DMEM addr. Attempted to read 8-bytes starting at address 1280 but only 1024 bytes are available
#    Time: 525 ns  Iteration: 1  Instance: /dmem_tb/UUT
# ** Error: Invalid DMEM addr. Attempted to read 8-bytes starting at address 1280 but only 1024 bytes are available
#    Time: 550 ns  Iteration: 1  Instance: /dmem_tb/UUT
# ** Error: Invalid DMEM addr. Attempted to read 8-bytes starting at address 1280 but only 1024 bytes are available
#    Time: 575 ns  Iteration: 1  Instance: /dmem_tb/UUT
# ** Error: Invalid DMEM addr. Attempted to write 8-bytes starting at address 1280 but only 1024 bytes are available
#    Time: 625 ns  Iteration: 1  Instance: /dmem_tb/UUT
# ** Error: Invalid DMEM addr. Attempted to write 8-bytes starting at address 1280 but only 1024 bytes are available
#    Time: 675 ns  Iteration: 1  Instance: /dmem_tb/UUT
```

Figure 13: Simulation Waveform of data memory invalid address

At 500 ns, 'MemRead' = '1', 'MemWrite' = '0', 'Address' = 1280. We tried to read data from Byte(1280), which is not available. And we got error infromation, and the 'ReadData' didn't change. During 600 to 700 ns, 'MemRead' = '0', 'MemWrite' = '1', 'Address' = 1280, and there are rising edges at 625 ns and 675 ns. We tried to write data into Byte(1280), which is not available. And we got error information at the rising edge of 'Clock'.

5: Resubmission

ALU Control

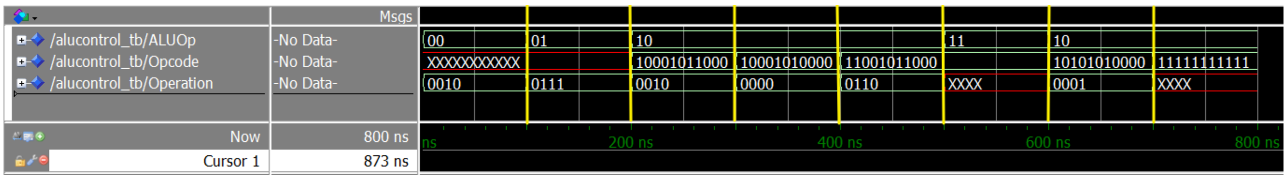

Msgs
/alucontrol_tb/ALUOp -No Data- 00 01 10 11 10
/alucontrol_tb/Opcode -No Data- XXXXXXXXXXX 10001011000 10001010000 11001011000 10101010000 11111111111
/alucontrol_tb/Operation -No Data- 0010 0111 0010 0000 0110 XXXX 0001 XXXX
Now 800 ns
Cursor 1 873 ns
ns 200 ns 400 ns 600 ns 800 ns

Figure 14: Simulation Waveform of ALU Control

From 0 to 100 ns, ALUOp is ”00”, it’s LDUR instruction and we don’t care the Opcode. The operation code is ”0010” for add action.
From 100 to 200 ns, ALUOp is ”01”, it’s STUR instruction and we don’t care Opcode. The operation code is ”0010” for add action.
From 200 to 500 ns, ALUOp is ”10”, they are R-type instructions and according to Opcode. Correspondingly, the operation codes are ”0010” for add action from 200 to 300 ns, ”0000” for and action from 300 to 400 ns, and ”0110” for subtract action from 400 to 500 ns.
From 500 to 600 ns, ALUOp is ”11”, this is an invalid ALUOp, so the operation code is ”XXXX”.
From 600 to 800 ns, ALUOp is ”10” they are R-type instructions and according to Opcode. Correspondingly, the operation codes are ”0001” for or action from 600 to 700 ns, and ”XXXX” for invalid action from 700 to 800 ns.

Design Decisions For register, I change the logic to make it only could be wrote in when the write address is not ”31”.
For ALUControl, when we get invalid opcode or ALUOp, the output operation code would be ”XXXX”.
For add part, I first connect two half adders and implement a full adder. Then I connect 64 full adders into a 64 bits carry ripple adder. And I use the 64 bits carry ripple adder to implement my add action.
For invalid instructions, the CPU Control signals would be Reg2Loc = ’0’, CBranch = ’0’, MemRead = ’0’, MemtoReg = ’0’, MemWrite = ’0’, ALUSrc = ’0’, RegWrite = ’0’, UBranch = ’0’, ALUOp = ”XX”. NOTZERO = ’0’.

Compile Sequence xorgate.vhd half_adder.vhd full_adder.vhd adder64.vhd add.vhd alu.vhd alucontrol.vhd cpucontrol.vhd dmem.vhd imem.vhd registers.vhd