Verilog Project

Verilog-Implementation-Of-32-Bit-Signed-Divider

By : Piyush Gautam

Problem Statement

A Divider for signed (2’s complement) binary numbers that divides a 32-bit dividend by a 16-bit divisor to give a 16-bit quotient.

Description

Here we have developed a signed divider by using a successive shift and subtraction operations.We complement the dividend if they are negative, and after completion of division we change the sign of quotient. We need to subtract divisor from dividend so when we have negative divisor, we don’t have to complement. For overflow check we first left shift the dividend and then compare first 16 bits if they are greater or equal to divisor we would give overflow because quotient wouldn’t able to fit in 16 bits.

Binary division for signed numbers can be done using complex algorithms that operate directly on signed values. However, such designs require intricate control logic. In this project, we simplify the process by converting the operation into an unsigned division problem:

- If the dividend or divisor is negative, we take its 2’s complement before division.
- At the end, the quotient is complemented if the signs of the original operands differ.

Theory & Background

Key idea:

Signed division is performed by:

1. Storing and managing operand signs separately.
2. Performing an unsigned shift-subtract division.
3. Correcting the sign of the quotient at the end.

Overflow detection method:

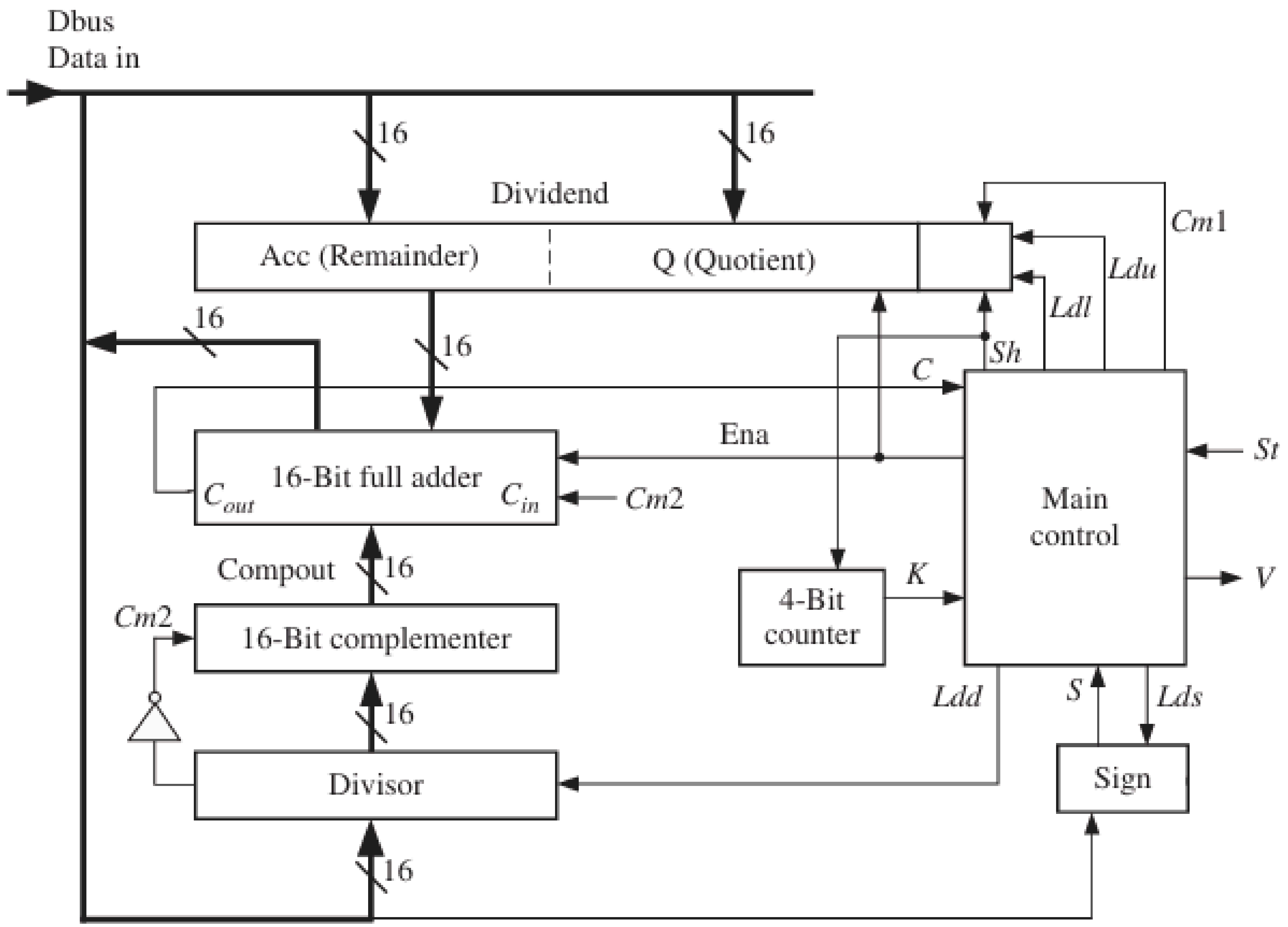
Overflow occurs when the quotient magnitude exceeds 7FFFh (most positive 16-bit signed value).

Detection is done by:

- Shifting the dividend left by one place.
- Comparing the upper half of the dividend with the divisor.
- If divu ≥ divisor, overflow is asserted.

Special case: dividend = 80000000h with divisor = FFFFh requires extra handling if generating 8000h is desired.

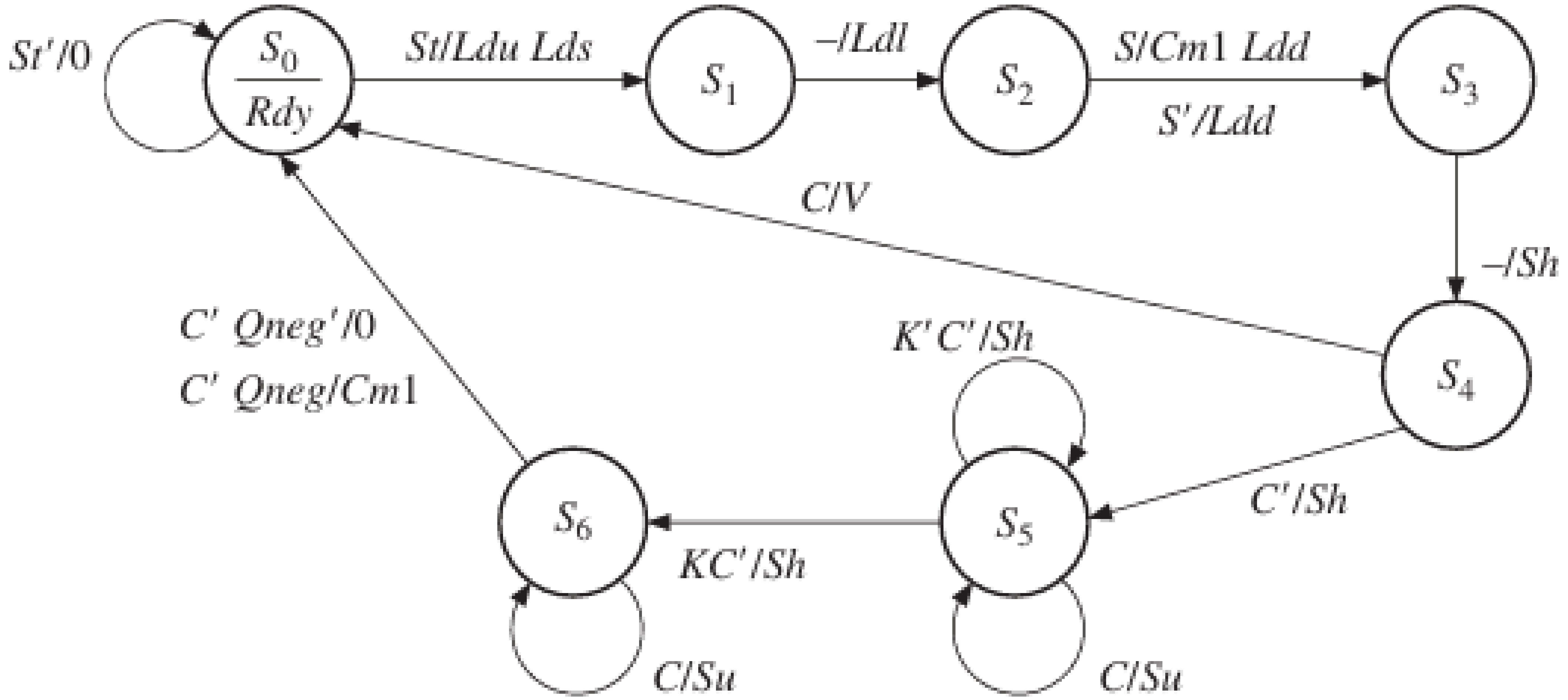
Block Diagram for Signed Divider



Control signals:

- LdU – Load upper half of dividend from bus.
- LdL – Load lower half of dividend from bus.
- Lds – Load sign of dividend into sign flip-flop.
- S – Sign of dividend.
- Cm1 – Complement dividend register (2's complement).
- Ldd – Load divisor from bus.
- Su – Enable adder output onto bus (Ena), and load upper half of dividend from bus.
- Cm2 – Enable complementer (Cm2 equals the complement of the sign bit of the divisor, so a positive divisor is complemented and a negative divisor is not).
- Sh – Shift the dividend register left one place and increment the counter.
- C – Carry output from adder (if C = 1, the divisor can be subtracted from the upper dividend).
- St – Start.
- V – Overflow.
- Qneg – Quotient will be negative (Qneg = 1 when the sign of the dividend and divisor are different).

State Diagram



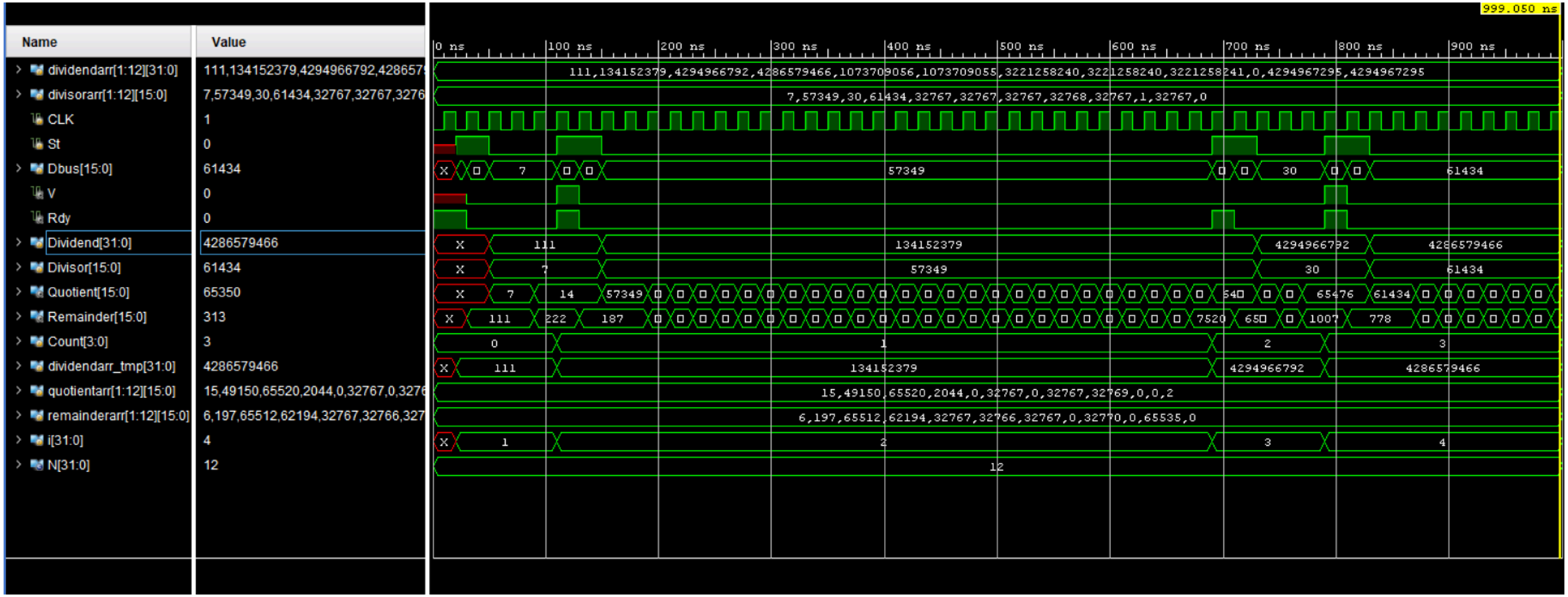
Algorithm/ Pseudo code

1. Load upper half of dividend; store sign bit.
2. Load lower half of dividend.
3. Load divisor.
4. Complement dividend if negative.
5. Test for overflow; if true, go to done state.
6. Perform shift-subtract division for 15 cycles.
7. Complement quotient if original signs differ.

Verilog Implementation : Github Link

Testing & Simulation : Github Link

Simulation Results :

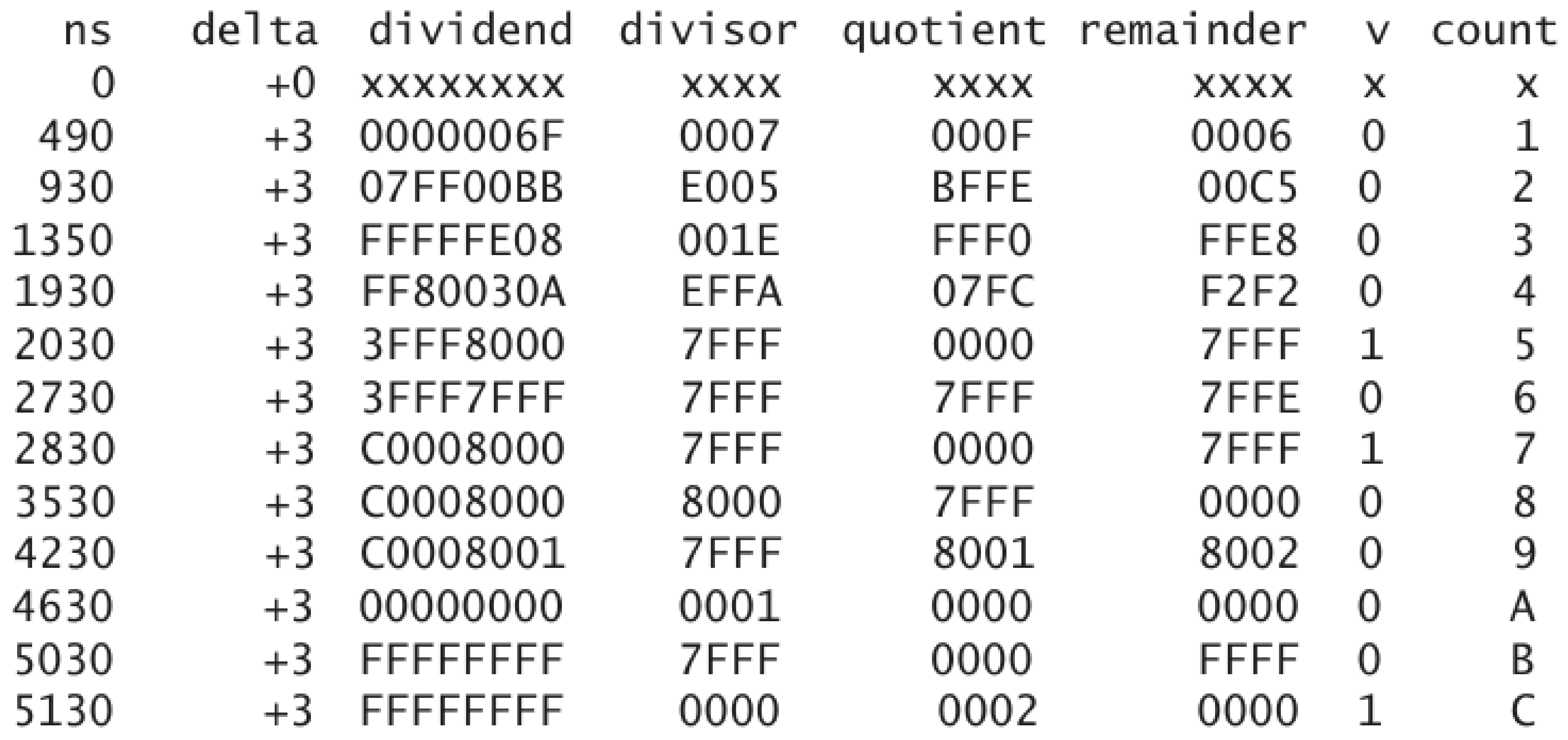


Name	Value
dividendarr[1:12][31:0]	111,134152379,4294966792,428657
divisorarr[1:12][15:0]	7,57349,30,61434,32767,32767,3276
CLK	1
St	0
Dbus[15:0]	61434
V	0
Rdy	0
Dividend[31:0]	4286579466
Divisor[15:0]	61434
Quotient[15:0]	65350
Remainder[15:0]	313
Count[3:0]	3
dividendarr_tmp[31:0]	4286579466
quotientarr[1:12][15:0]	15,49150,65520,2044,0,32767,0,3276
remainderarr[1:12][15:0]	6,197,65512,62194,32767,32766,327
i[31:0]	4
N[31:0]	12

Console Output :

```
# KERNEL: quotient[1] is correct
# KERNEL: remainder[1] is correct
# KERNEL: quotient[2] is correct
# KERNEL: remainder[2] is correct
# KERNEL: quotient[3] is correct
# KERNEL: remainder[3] is correct
# KERNEL: quotient[4] is correct
# KERNEL: remainder[4] is correct
# KERNEL: quotient[5] is correct
# KERNEL: remainder[5] is correct
# KERNEL: quotient[6] is correct
# KERNEL: remainder[6] is correct
# KERNEL: quotient[7] is correct
# KERNEL: remainder[7] is correct
# KERNEL: quotient[8] is correct
# KERNEL: remainder[8] is correct
# KERNEL: quotient[9] is correct
# KERNEL: remainder[9] is correct
# KERNEL: quotient[10] is correct
# KERNEL: remainder[10] is correct
# KERNEL: quotient[11] is correct
# KERNEL: remainder[11] is correct
# KERNEL: quotient[12] is correct
# KERNEL: remainder[12] is correct
# KERNEL: TESTS DONE
```

Console Output :



```
  ns   delta  dividend  divisor  quotient remainder  v  count
   0     +0  xxxxxxxx     xxxx      xxxx      xxxx  x      x
 490     +3  0000006F     0007      000F      0006  0      1
 930     +3  07FF00BB     E005      BFFE      00C5  0      2
1350     +3  FFFFFE08     001E      FFF0      FFE8  0      3
1930     +3  FF80030A     EFFA      07FC      F2F2  0      4
2030     +3  3FFF8000     7FFF      0000      7FFF  1      5
2730     +3  3FFF7FFF     7FFF      7FFF      7FFE  0      6
2830     +3  C0008000     7FFF      0000      7FFF  1      7
3530     +3  C0008000     8000      7FFF      0000  0      8
4230     +3  C0008001     7FFF      8001      8002  0      9
4630     +3  00000000     0001      0000      0000  0      A
5030     +3  FFFFFFFF     7FFF      0000      FFFF  0      B
5130     +3  FFFFFFFF     0000      0002      0000  1      C
```

Conclusion :

The signed binary divider successfully performs 32-bit ÷ 16-bit signed division with overflow detection.
While most cases are handled correctly, certain edge cases require refinement for complete
IEEE-style signed arithmetic compatibility. Future improvements could include:

- Special handling for the 8000h quotient case.

Detecting overflow in signed division is more complex than in unsigned division due to the presence of sign bits and the need to correctly handle edge cases. By analyzing the maximum allowable values of the divisor, quotient, and remainder, we find that the maximum dividend without overflow is 3FFF7FFFh. To detect overflow, the dividend is first shifted left once and compared to the divisor. If the upper half of the shifted dividend is greater than or equal to the divisor, an overflow will occur. This method is reliable for both positive and negative numbers, except for the special edge case of the minimum negative value (8000000h), which requires additional handling.