GCD Custom Instruction

System Verilog:

```
module gcd_ci(
    input logic clock,
    input logic reset,
    input logic clock_en,
    input logic start,
    input logic [31:0] dataa,
    input logic [31:0] datab,
    output logic [31:0] result,
    output logic done
);
    // Control-status registers:
    //     aReg: operand A
    //     bReg: operand B
    //     sReg: status register (contains 0 when computation is done)

    logic [31:0] aReg, bReg;
    logic sReg;

    always_ff @(posedge clock)
    if(clock_en) //operations proceed if clock_en is high
      begin
        if(reset)
          begin
            bReg <= {32{1'b0}};
            aReg <= {32{1'b0}};
            done <= 0;
            sReg <= 0;
```

```
        end
      else
        begin
          if(start)
            begin
              aReg <= dataa;
              bReg <= datab;
              sReg <= 1'b1; //control for calculation
            end
          if(sReg) //calculation begins
            begin
              if (aReg > bReg) aReg <= aReg - bReg;
            else if (bReg > aReg) bReg <= bReg - aReg;
            else begin //assigning done and result after calculation
                        done <= 1'b1;
                        result <= aReg;
                        sReg <= 0; //turn off the control
              end
            end
          end
        end
    else
    begin
       done <= 1'b0;
    end
endmodule
```

Software Implementation:

```
/*******************************************************************************
 * This program calculates GCD of two 32/64 bit unsigned integer
 *
 * It performs the following:
 *      1. Gets the bit length and gets two non-zero operands
 *      2. performs GCD of them in SW and HW component usign a custom instruction
 *      3. shows the number of clock cycle taken for calculations
 ******************************************************************************/

#include "address_map_nios2.h"
#include <stdio.h>
#include <string.h>
#include <stdint.h>

#define TIMER_BASE  0xFF202000

//Custom Instruction Definition
#define INS_GCDCI_ID 0x00
#define INS_GCDCI(x,y) __builtin_custom_inii(INS_GCDCI_ID, x, y)

int main(void) {
   /* Declare volatile pointers to I/O registers (volatile means that IO load
      and store instructions will be used to access these pointer locations,
      instead of regular memory loads and stores) */

  volatile int *   timer_ctrl  = (int *)(TIMER_BASE+4);
  volatile int *   timer_start_down  = (int *)(TIMER_BASE+8);
  volatile int *   timer_start_up  = (int *)(TIMER_BASE+12);
  volatile int *   timer_snap_down  = (int *)(TIMER_BASE+16);
```

```
volatile int *   timer_snap_up  = (int *)(TIMER_BASE+20);

unsigned long int a_32=0, b_32=0, clock_start=0xFFFFFFFF;

printf("\nThis is a GCD calculator ");

while(1) {

    // Get input A
    printf("\nEnter the first non-zero unsigned integer (A): ");
    scanf("%lu", &a_32);
    printf("\nA is = %lu\n", a_32);

    // Get input B
    printf("\nEnter the second non-zero unsigned integer (B): ");
    scanf("%lu", &b_32);
    printf("\nB is = %lu\n", b_32);

    // Check for zero values
    if(a_32 == 0 || b_32 == 0) {
        printf("\nError: Inputs must be non-zero.\n");
        continue;
    }

     unsigned long int res_32=0, res_ci=0;

   *(timer_start_down) = 0x0000FFFF; //load the counter
   *(timer_start_up) = 0x0000FFFF;   //load the counter
   *(timer_ctrl) = 0x00000004;       //start the counter

    res_ci = INS_GCDCI(a_32, b_32);        //get the gcd using custom instruction
```

```
*(timer_snap_down) = 0; //again take snapshot
 unsigned long int clock_end_d_ci = *(timer_snap_down); //read the snap value
unsigned long int clock_end_u_ci = *(timer_snap_up); //read the snap value
*(timer_ctrl) = 0x00000008;       //stop the counter
clock_end_u_ci = clock_end_u_ci << 16;
unsigned long int clock_end_ci = clock_end_u_ci | clock_end_d_ci; //combine two snapshots into 32 bit value
printf("\nGCD of the two integer by usign custom GCD_CI is = %lu\n", res_ci);
float time_ci = 0.02*(clock_start-clock_end_ci); //Timer runs at 50MHz
printf ("\n Time taken for SW execution is = %f micro secs\n", time_ci);
*(timer_start_down) = 0x0000FFFF; //load the counter
*(timer_start_up) = 0x0000FFFF;   //load the counter
*(timer_ctrl) = 0x00000004;       //start the counter
 //  GCD Calculation
      while (b_32 != 0) {
          if (a_32>b_32) {
                    a_32 = a_32-b_32;
         }
         else {
            b_32 = b_32-a_32;
         }
      }
      res_32=a_32;
  *(timer_snap_down) = 0; //again take snapshot
  unsigned long int clock_end_d = *(timer_snap_down); //read the snap value
  unsigned long int clock_end_u = *(timer_snap_up); //read the snap value
  *(timer_ctrl) = 0x00000008;       //stop the counter
  clock_end_u = clock_end_u << 16;
  unsigned long int clock_end = clock_end_u | clock_end_d; //combine two snapshots into 32 bit value
  printf("\nGCD of the two integer of software execution is = %lu\n", res_32);
  float time = 0.02*(clock_start-clock_end);
  printf ("\n Time taken for SW execution is = %f micro secs\n", time);
  printf ("\n Speed-up/slow down achieved = %.2f x\n", (time/time_ci));
```

```
    }
}
```

Output:

```
Enter the first non-zero unsigned integer (A): 1
A is = 1
Enter the second non-zero unsigned integer (B): 1
B is = 1
GCD of the two integer by usign custom GCD_CI is = 1
 Time taken for SW execution is = 1.040000 micro secs
GCD of the two integer of software execution is = 1
 Time taken for SW execution is = 2.440000 micro secs
 Speed-up/slow down achieved = 2.35 x
```

I

```
Enter the first non-zero unsigned integer (A): 55555333
A is = 55555333
Enter the second non-zero unsigned integer (B): 55555553
B is = 55555553
GCD of the two integer by usign custom GCD_CI is = 1
 Time taken for SW execution is = 5051.500000 micro secs
GCD of the two integer of software execution is = 1
 Time taken for SW execution is = 35358.558594 micro secs
 Speed-up/slow down achieved = 7.00 x
```

II

```
Enter the first non-zero unsigned integer (A): 8048
A is = 8048
Enter the second non-zero unsigned integer (B): 1024
B is = 1024
GCD of the two integer by usign custom GCD_CI is = 16
 Time taken for SW execution is = 1.200000 micro secs
GCD of the two integer of software execution is = 16
 Time taken for SW execution is = 5.800000 micro secs
 Speed-up/slow down achieved = 4.83 x
```

III

```
Enter the first non-zero unsigned integer (A): 24488822
A is = 24488822
Enter the second non-zero unsigned integer (B): 3
B is = 3
GCD of the two integer by usign custom GCD_CI is = 1
 Time taken for SW execution is = 163259.562500 micro secs
GCD of the two integer of software execution is = 1
 Time taken for SW execution is = 1142814.250000 micro secs
 Speed-up/slow down achieved = 7.00 x
```

IV

Case I: Trivial Inputs but still we can observe the performance gain is around 2.35x.

Case II: 8 digit prime number inputs and we can observe a performance gain around 7x.

Case III: 4 digit number inputs and we can observe a performance gain around 4.83x.

Case IV: One input is 8 digit and other is single digit. Here also the performance gain is around 7x.

Thus we can conclude that our custom instruction for GCD calculation is definitely beneficial and will lead to an advantageous design if opted for. For larger numbers, where the number loops will be higher, it will give more performance gain.

Demo:

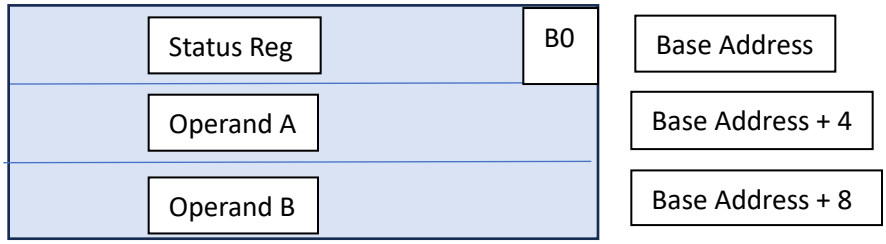
https://youtu.be/RpuvJjMNdts?si=R3-Zetu8IBYQdHRy

Hardware Accelerator for GCD Calculation

⊟ **GcdCalculator32bit_0**	GcdCalculator32bit				
reset	Reset Input	*Double-click to*	[clock_s...		
csr	Avalon Memory Mapped ...	*Double-click to*	[clock_s...	0x0000 5000	0x0000_500f
clock_sink	Clock Input	*Double-click to*	**Syste...**		
⊟ **GcdCalculator64bit_0**	GcdCalculator64bit				
reset	Reset Input	*Double-click to*	[clock_s...		
csr	Avalon Memory Mapped ...	***Double-click to***	[clock_s...	0x0000 5080	0x0000_509f
clock_sink	Clock Input	*Double-click to*	**Syste...**		

Memory Map in ARM

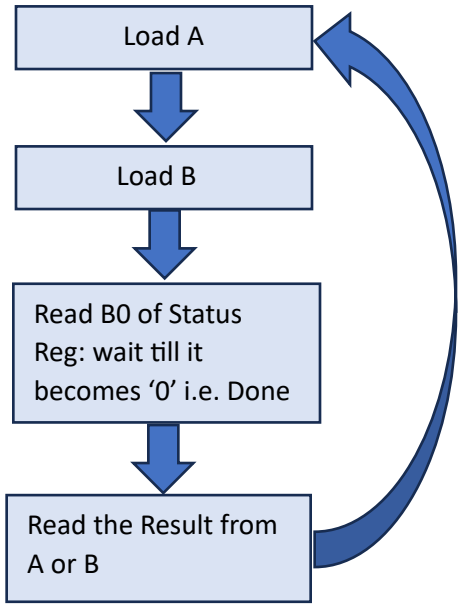
Programmer's View of 32bit GCD Hardware Component:



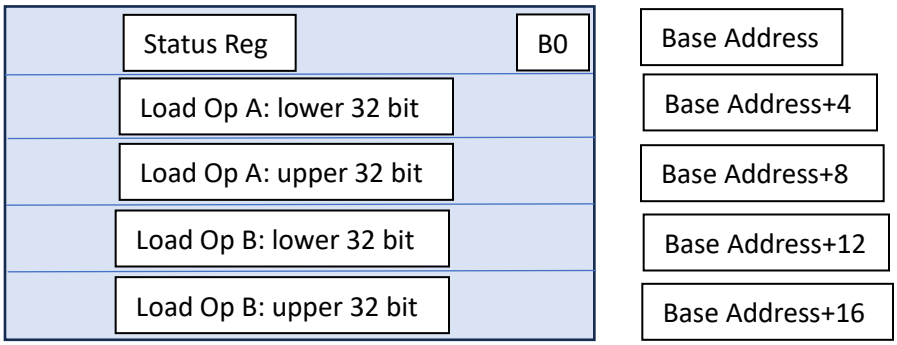
B31 B1 B0

Fig: Register Architecture of 32bit GCD Hardware Component

Programming Flow Requirement:



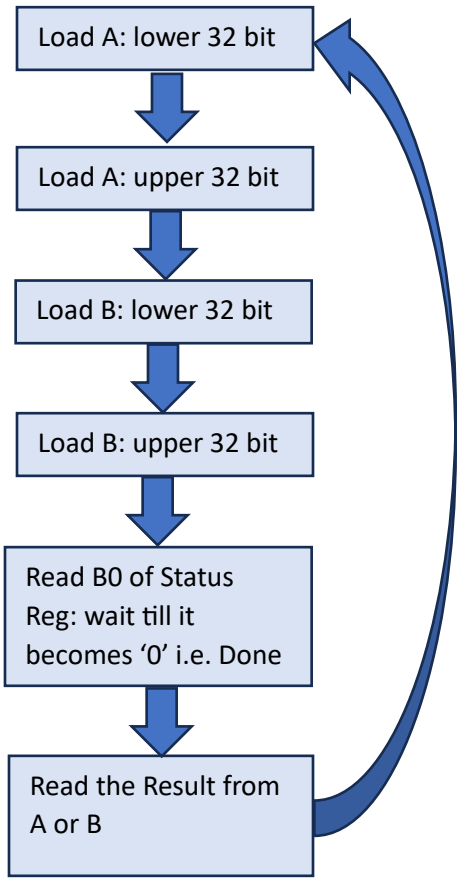
Programmers View of 64bit GCD Hardware Component:



B31……..……..B1 B0

Fig: Register Architecture of 64bit GCD Hardware Component

Programming Flow Requirements:



System Verilog Code for 32bit GCD Component:

```
module GcdCalculator32bit(
    input logic clock,
    input logic reset,

    // Avalon-MM slave interface for accessing control-status registers
    input logic csr_read,
    input logic csr_write,
    input logic [1:0] csr_address,
    output logic [31:0] csr_readdata,
    input logic [31:0] csr_writedata
);


    // Control-status registers:
    //     aReg: operand A
    //     bReg: operand B
    //     sReg: status register (contains 0 when computation is done)

    logic [31:0] aReg, bReg, sReg;

    always_ff @(posedge clock)
    begin
        if (reset) begin
            sReg <= 1'b0;
            aReg <= 32'b0;
            bReg <= 32'b0;
```

```
    end

    else begin
      // While sReg is set, keep doing computations
      if (sReg) begin
        if (aReg > bReg) aReg <= aReg - bReg;
        else if (bReg > aReg) bReg <= bReg - aReg;
        else sReg <= 1'b0;
      end

      // Write control-status registers
      // When computation is going, write is disabled
      else if (csr_write) begin
        case(csr_address)
          2'b01: aReg <= csr_writedata;
          2'b10: begin
            bReg <= csr_writedata;
            sReg <= 1'b1;
          end
        endcase
      end

      // Read is always possible
      if (csr_read) begin
        case(csr_address)
          2'b00: csr_readdata <= sReg;
          2'b01: csr_readdata <= aReg;
          2'b10: csr_readdata <= bReg;
        endcase
      end
    end
  end
```

```
endmodule
```

System Verilog Code for 64 bit GCD Component:

```
module GcdCalculator64bit(
    input logic clock,
    input logic reset,

    // Avalon-MM slave interface for accessing control-status registers
    input logic csr_read,
    input logic csr_write,
    input logic [2:0] csr_address,
    output logic [31:0] csr_readdata,
    input logic [31:0] csr_writedata
);


    // Control-status registers:
    //     aReg: operand A
    //     bReg: operand B
    //     sReg: status register (contains 0 when computation is done)

    logic [31:0] sReg;
    logic [63:0] aReg, bReg;

    always_ff @(posedge clock)
    begin
        if (reset) begin
            sReg <= 1'b0;
            aReg <= 64'b0;
```

```
    bReg <= 64'b0;
end

else begin
    // While sReg is set, keep doing computations
    if (sReg) begin
        if (aReg > bReg) aReg <= aReg - bReg;
        else if (bReg > aReg) bReg <= bReg - aReg;
        else sReg <= 1'b0;
    end

    // Write control-status registers
    // When computation is going, write is disabled
   else if (csr_write) begin
        case(csr_address)
            // Move data to aReg
            3'b001: aReg[31:0] <= csr_writedata;
            3'b010: aReg[63:32] <= csr_writedata;

            // Move data to bReg
            3'b011: bReg[31:0] <= csr_writedata;
            3'b100: begin
                bReg[63:32] <= csr_writedata;

                // Writing to the higher part of
                // bReg starts the computation process
                sReg <= 1'b1;
            end
        endcase
    end

    // Read is always possible
    if (csr_read) begin
```

```
            case(csr_address)
               3'b000: csr_readdata <= sReg;
               3'b001: csr_readdata <= aReg[31:0];
               3'b010: csr_readdata <= aReg[63:32];
               3'b011: csr_readdata <= bReg[31:0];
               3'b100: csr_readdata <= bReg[63:32];
            endcase
         end
      end
   end

endmodule
```

Software Implementation of the Application:

```
/*************************************************************************
 * This program calculates GCD of two 32/64 bit unsigned integer
 *
 * It performs the following:
 *                                  1. Gets the bit length and gets two non-zero operands
 *                                  2. performs GCD of them in SW and HW component
 *                                  3. shows the number of clock cycle taken for calculations
*************************************************************************/
#include "address_map_arm.h"
#include <stdio.h>
#include <time.h>

#define TIMER_BASE  0xFF202000
#define ARM_BRIDGE 0xFF200000
#define GCD_64_BASE 0x00005080
#define GCD_32_BASE 0x00005000

int main(void) {
   /* Declare volatile pointers to I/O registers (volatile means that IO load
      and store instructions will be used to access these pointer locations,
      instead of regular memory loads and stores) */
  volatile int *   hwd64_done  = (int *)(GCD_64_BASE+ARM_BRIDGE+0);
  volatile int *   hwd64_a_d  = (int *)(GCD_64_BASE+ARM_BRIDGE+4);
  volatile int *   hwd64_a_u  = (int *)(GCD_64_BASE+ARM_BRIDGE+8);
  volatile int *   hwd64_b_d  = (int *)(GCD_64_BASE+ARM_BRIDGE+12);
  volatile int *   hwd64_b_u  = (int *)(GCD_64_BASE+ARM_BRIDGE+16);

  volatile int *   hwd32_done  = (int *)(GCD_32_BASE+ARM_BRIDGE+0);
  volatile int *   hwd32_a  = (int *)(GCD_32_BASE+ARM_BRIDGE+4);
```

```
volatile int *   hwd32_b  = (int *)(GCD_32_BASE+ARM_BRIDGE+8);

volatile int *   timer_ctrl  = (int *)(TIMER_BASE+4);
volatile int *   timer_start_down  = (int *)(TIMER_BASE+8);
volatile int *   timer_start_up  = (int *)(TIMER_BASE+12);
volatile int *   timer_snap_down  = (int *)(TIMER_BASE+16);
volatile int *   timer_snap_up  = (int *)(TIMER_BASE+20);

unsigned long long int a=0, b=0;
unsigned long int len=0, a_32=0, b_32=0, clock_start=0xFFFFFFFF;

printf("\nThis is a GCD calculator ");

while(1) {
 printf("\nEnter the bit length of your operands (32 or 64): ");
 scanf("%lu", &len);
 printf("%lu\n", len);


 if (len==64) {
    printf("\nEnter the first non-zero unsigned integer (A): ");
    scanf("%llu", &a);
    printf("\nA is = %llu\n", a);

    // Get input B
    printf("\nEnter the second non-zero unsigned integer (B): ");
    scanf("%llu", &b);
    printf("\nB is = %llu\n", b);

    // Check for zero values
    if(a == 0 || b == 0) {
        printf("\nError: Inputs must be non-zero.\n");
        continue;
```

```
}
unsigned long long int res=0, a_h=a, b_h=b;
*(timer_start_down) = 0x0000FFFF; //load the counter
*(timer_start_up) = 0x0000FFFF;   //load the counter
*(timer_ctrl) = 0x00000004;       //start the counter


//  GCD Calculation
while (b != 0) {
        if (a>b) {
                a = a-b;
        }
        else {
                b = b-a;
        }
}

res=a;



*(timer_snap_down) = 0; //again take snapshot
unsigned long int clock_end_d = *(timer_snap_down); //read the snap value
unsigned long int clock_end_u = *(timer_snap_up); //read the snap value
*(timer_ctrl) = 0x00000008;       //stop the counter

clock_end_u = clock_end_u << 16;
unsigned long int clock_end = clock_end_u | clock_end_d;
printf("\nGCD of the two integer of software execution is = %llu\n", res);
//Timer Works at 100MHz
float time = 0.01*(clock_start-clock_end);
printf ("\n Time taken for SW execution is = %f micro secs\n", time);
unsigned long long int r_h_d = 0, r_h_u = 0;
```

```
unsigned long long int a_u = (a_h & 0xFFFFFFFF00000000);
a_u = a_u >> 32;
unsigned long long int a_d = (a_h & 0x00000000FFFFFFFF);
unsigned long long int b_u = (b_h & 0xFFFFFFFF00000000);
b_u = b_u >> 32;
unsigned long long int b_d = (b_h & 0x00000000FFFFFFFF);
*(timer_start_down) = 0x0000FFFF; //load the counter
*(timer_start_up) = 0x0000FFFF;   //load the counter
*(timer_ctrl) = 0x00000004;       //start the counter

*(hwd64_a_d) = a_d;
*(hwd64_a_u) = a_u;
*(hwd64_b_d) = b_d;
*(hwd64_b_u) = b_u;

 while (*(hwd64_done) != 0) { //wait while not done
 }
 //get the result
 r_h_d = *(hwd64_a_d);
 r_h_u = *(hwd64_a_u);

*(timer_snap_down) = 0; //again take snapshot
 unsigned long int clock_end_d_h = *(timer_snap_down); //read the snap value
 unsigned long int clock_end_u_h = *(timer_snap_up); //read the snap value
 *(timer_ctrl) = 0x00000008;       //stop the counter

 clock_end_u_h = clock_end_u_h << 16;
 unsigned long int clock_end_h = clock_end_u_h | clock_end_d_h; //convert 16bit snapshots to one 32 bit value
 r_h_u = r_h_u << 32;

unsigned long long int r_h64 = r_h_u | r_h_d; //convert the 32bit results to one 64bit numbers
printf("\nGCD of the two integer of hardware execution is = %llu\n", r_h64);
//Timer works at 100MHz
```

```
    float time_h = 0.01*(clock_start-clock_end_h);
    printf ("\n Time taken for SW execution is = %f micro secs\n", time_h);
    printf ("\n Speed-up/slow down achieved = %.2f x\n", (time/time_h));
}
if (len==32) {
    // Get input A
    printf("\nEnter the first non-zero unsigned integer (A): ");
    scanf("%lu", &a_32);
    printf("\nA is = %lu\n", a_32);

    // Get input B
    printf("\nEnter the second non-zero unsigned integer (B): ");
    scanf("%lu", &b_32);
    printf("\nB is = %lu\n", b_32);

    // Check for zero values
    if(a_32 == 0 || b_32 == 0) {
        printf("\nError: Inputs must be non-zero.\n");
        continue;
    }

   unsigned long int res_32=0, a_32_h=a_32, b_32_h=b_32;

   *(timer_start_down) = 0x0000FFFF; //load the counter
   *(timer_start_up) = 0x0000FFFF;   //load the counter
   *(timer_ctrl) = 0x00000004;       //start the counter

   //  GCD Calculation

   while (b_32 != 0) {
          if (a_32>b_32) {
                a_32 = a_32-b_32;
          }
```

```
        else {
                b_32 = b_32-a_32;
        }
}

res_32=a_32;

 *(timer_snap_down) = 0; //again take snapshot
 unsigned long int clock_end_d = *(timer_snap_down); //read the snap value
 unsigned long int clock_end_u = *(timer_snap_up); //read the snap value
 *(timer_ctrl) = 0x00000008;       //stop the counter
 clock_end_u = clock_end_u << 16;
 unsigned long int clock_end = clock_end_u | clock_end_d;
 printf("\nGCD of the two integer of software execution is = %lu\n", res_32);
 //Timer Works at 100MHz
 float time = 0.01*(clock_start-clock_end);
printf ("\n Time taken for SW execution is = %f micro secs\n", time);
unsigned long int r_h = 0;
*(timer_start_down) = 0x0000FFFF; //load the counter
*(timer_start_up) = 0x0000FFFF;   //load the counter
*(timer_ctrl) = 0x00000004;       //start the counter

*(hwd32_a) = a_32_h;
*(hwd32_b) = b_32_h;

while (*(hwd32_done) != 0) { //wait till done
 }

 r_h = *(hwd32_a); //get the result

 *(timer_snap_down) = 0; //again take snapshot
 unsigned long int clock_end_d_h = *(timer_snap_down); //read the snap value
 unsigned long int clock_end_u_h = *(timer_snap_up); //read the snap value
```

```
    *(timer_ctrl) = 0x00000008;       //stop the counter

    clock_end_u_h = clock_end_u_h << 16;
    unsigned long int clock_end_h = clock_end_u_h | clock_end_d_h;
    printf("\nGCD of the two integer of hardware execution is = %lu\n", r_h);
    //user logic works at 50MHz and Timer works at 100MHz
    float time_h = 0.01*(clock_start-clock_end_h);
    printf ("\n Time taken for SW execution is = %f micro secs\n", time_h);
    printf ("\n Speed-up/slow down achieved = %.2f x\n", (time/time_h));
   }
 }
}
```

Output:

```
Enter the bit length of your operands (32 or 64): 64
64

Enter the first non-zero unsigned integer (A): 1071054436673

A is = 1071054436673

Enter the second non-zero unsigned integer (B): 2619679884701

B is = 2619679884701

GCD of the two integer of software execution is = 101

 Time taken for SW execution is = 2.050000 micro secs

GCD of the two integer of hardware execution is = 101

 Time taken for SW execution is = 3.300000 micro secs

 Speed-up/slow down achieved = 0.62 x
```

I

```
Enter the bit length of your operands (32 or 64): 22
22

Enter the bit length of your operands (32 or 64): 32
32

Enter the first non-zero unsigned integer (A): 1

A is = 1

Enter the second non-zero unsigned integer (B): 0

B is = 0

Error: Inputs must be non-zero.
```

II

```
Enter the bit length of your operands (32 or 64): 64
64

Enter the first non-zero unsigned integer (A): 1071054436673

A is = 1071054436673

Enter the second non-zero unsigned integer (B): 256888

B is = 256888

GCD of the two integer of software execution is = 1

 Time taken for SW execution is = 46907.210938 micro secs

GCD of the two integer of hardware execution is = 1

 Time taken for SW execution is = 41697.179688 micro secs

 Speed-up/slow down achieved = 1.12 x
```

```
Enter the bit length of your operands (32 or 64): 64
64

Enter the first non-zero unsigned integer (A): 22222222444444

A is = 22222222444444

Enter the second non-zero unsigned integer (B): 66666666644444

B is = 66666666644444

GCD of the two integer of software execution is = 4

 Time taken for SW execution is = 322582.625000 micro secs

GCD of the two integer of hardware execution is = 4

 Time taken for SW execution is = 322584.218750 micro secs

 Speed-up/slow down achieved = 1.00 x
```

III

```
Enter the bit length of your operands (32 or 64): 32
32

Enter the first non-zero unsigned integer (A): 1

A is = 1

Enter the second non-zero unsigned integer (B): 1

B is = 1

GCD of the two integer of software execution is = 1

 Time taken for SW execution is = 0.670000 micro secs

GCD of the two integer of hardware execution is = 1

 Time taken for SW execution is = 1.460000 micro secs

 Speed-up achieved = 0.46 x
```

IV

```
Enter the bit length of your operands (32 or 64): 32
32

Enter the first non-zero unsigned integer (A): 55555333

A is = 55555333

Enter the second non-zero unsigned integer (B): 55555553

B is = 55555553

GCD of the two integer of software execution is = 1

 Time taken for SW execution is = 1052.930054 micro secs

GCD of the two integer of hardware execution is = 1

 Time taken for SW execution is = 2526.810059 micro secs

 Speed-up achieved = 0.42 x
```

V

VI

64 Bit operation:

Case I: 13-digit number inputs and we can observe performance gain is less than 1.

Case II: Shows if invalid option and inputs are provided.

Case III & IV: Large even and odd number inputs with different number of digits. We can observe performance is around 1-1.1x.

32 Bit operation:

Case V: Trivial inputs are provided, and we can observe a performance gain is less than 1.

Case VI: Large 8-digit prime number inputs and we can observe a performance gain is less than 1.

The performance gain is very less significant as the ARM processor runs at 800MHz and the user logic runs at 50MHz in the FPGA. There is no significant performance gain in terms of absolute time and actually in most cases the custom component is slower than the software but for the skewed inputs where one of the inputs are very large compare to other and for some very large inputs with very small GCD shows some performance gain.

Demo:

https://youtu.be/_SlGN1rbahw?si=xn1k48It95TieQcj