# EE 224: Implementation of a GCD circuit Case Study

Madhav P. Desai

March 20, 2017

## 1 Objective

- Design an unsigned integer divider with the following interface

```
entity unsigned_divider is
    port(clk: in bit;
       reset: in bit;
       -- the two inputs
       dividend: in bit_vector(15 downto 0);
       divisor : in bit_vector(15 downto 0);
       -- the next two implement a ready-ready
       -- protocol to start the division
       inputs_ready: in bit;
       divider_ready : out bit;
       -- the two outputs
       quotient  : out bit_vector(15 downto 0);
       remainder : out bit_vector(15 downto 0);
       -- the output ready-ready handshake
       output_ready: out bit;
       output_accept: in bit;
       );
end entity unsigned_divider;
```

  The input side information is exchanged using the inputs_ready/divider_ready pair, and the output side information is exchanged using the output_ready/output_accept pair. You are NOT ALLOWED to use the built in divide functions present in any of the ieee packages.

- Design a GCD circuit which accepts 16 input numbers and outputs their GCD. The system interface should be as follows:

```
entity System is
   port ( din: in bit_vector(15 downto 0);
          dout: out bit_vector(15 downto 0);
          start: in bit;
          done: out bit;
          erdy: in bit;
          srdy: out bit;
          clk: in bit;
          reset: in bit);
end entity;
```

The input data is accepted using the erdy/srdy pair. The system starts when the start bit is asserted, and indicates completion by asserting the done bit. When the done bit is asserted, the information at dout must be valid data.

## 2 Algorithm

The GCD of two numbers $a, b$ can be calculated by the following procedure:

```
uint16_t gcd2(uint16_t a, uint16_t b)
{
   uint16_t L = (a < b ? a : b);
   uint16_t H = (a < b ? b : a);

   uint16_t r;
   uint16_t q;
   while(L != 0)
   {
      r = L;
      uint16_t tmp = L;
      q = udiv16(H,L,&L);
      H = tmp;
   }
   return(r);
}
```

In this procedure, *udiv*16 is a 16-bit divide function. The *gcd*2 procedure can be used to construct a *gcd* function which works on an array of numbers as follows:

```
// size is the number of elements in N.
uint16_t gcd(uint16_t* N, uint16_t size)
{
   if(size <= 0)
     return(0);
   else if(size == 1)
     return(N[0]);
   uint16_t i;
   uint16_t g = gcd2(N[0],N[1]);
   for(i = 2; i < size; i++)
   {
     g = gcd2(g,N[i]);
   }

   return(g);
}
```

## 3 Implementation of the divider

We can use the long division algorithm:

```
// long division algorithm.
uint16_t udiv16(uint16_t dividend, uint16_t divisor, uint16_t* remainder)
{

   uint16_t quotient = 0xffff;
   *remainder = 0;
   if(divisor == 0)
   {
     return(quotient);
   }
   else if(divisor > dividend)
   {
     quotient = 0;
     *remainder = dividend;
     return(quotient);
   }
   else
   {
      quotient = 0;
      while(dividend >= divisor)
      {
        uint16_t curr_quotient = 1;
        uint16_t dividend_by_2 = (dividend >> 1);
        uint16_t shifted_divisor = divisor;
        while(1)
        {
           if(shifted_divisor < dividend_by_2)
           {
              shifted_divisor = (shifted_divisor << 1);
              curr_quotient = (curr_quotient << 1);
           }
           else
```

```
                break;
        }

        quotient += curr_quotient;
        dividend -= shifted_divisor;
        }

        *remainder = dividend;
   }
   return(quotient);
}
```

## 4 The implementation

We can implement the GCD algorithm using the structure shown in the C program. We will implement four threads: the main thread, the gcd thread, the gcd2 thread and the divider thread.

### 4.1 The GCD RTL algorithm in pseudo-code

```
// Notes:  there are four threads described here
// The call structure is
//    main --> gcd --> gcd2 --> udiv16
//
// The top-level thread main reads in the 16 numbers
// and puts them in a memory.
//
// main then calls gcd to compute the gcd of the 16
// numbers.
//
// gcd in turn uses gcd2 to compute the gcd of two numbers.
//
// gcd uses the divider thread udiv16 to compute the
// quotient and remainder.
//

// Thread divider
// registers = { dividend, divisor, quotient, remainder,
//                  curr_quotient, shifted_divisor, dividend_by_2 }
// input control signals from "outside" = { div_start }
// output status signals to "outside" = { div_done  }
div_reset:
       if div_start then
          goto div_outerloop /
          {remainder <- dividend ,
           quotient <- 0}
       else
          goto div_reset
       endif

div_outerloop:
       if (divisor <= remainder) then
          goto div_innerloop /
          { curr_quotient <- "0000000000000001" ,
            dividend_by_2 <- (remainder >> 1) ,
            shifted_divisor <- divisor}
       else
          goto div_completed
       endif

div_innerloop:
       if (shifted_divisor < dividend_by_2) then
          goto div_innerloop /
          { shifted_divisor <- (shifted_divisor << 1) ,
            curr_quotient <- (curr_quotient << 1) }
       else
          goto div_update
       endif

div_update:
       goto div_outerloop/
          { quotient <- (quotient | curr_quotient) ,
            remainder <- (remainder - shifted_divisor)}

div_completed:
       goto div_reset / {div_done = 0}


// Thread gcd2
// registers   = {gcd2_A, gcd2_B, remainder,
//                  gcd2_result, dividend, divisor}
// E = { gcd2_start, div_done }
// G = { gcd2_done, div_start }
gcd2_reset: if gcd2_start then
          if( gcd2_A > gcd2_B) then
              goto gcd2_loop /
              {dividend <- gcd2_A ,
               divisor <- gcd2_B }
          else
              goto gcd2_loop /
              { dividend <- gcd2_B ,
                divisor <- gcd2_A}
       else
          goto gcd2_reset
       endif

gcd2_loop: if (divisor == 0) then
           goto gcd2_reset / {gcd2_done = 1}
      else
           goto gcd2_wait_div /
           { div_start = 1 ,
             gcd2_result <- divisor }


gcd2_wait_div: if(div_done) then
               goto gcd2_loop /
               { dividend <- divisor ,
                 divisor <- remainder}
           else
               goto gcd2_wait_for_div


// thread gcd
// registers = {gcd2_result, gcd_result,
//                 gcd2_A, gcd2_B, i}
// Memory = {N[0:15]}
// E = { gcd_start, gcd2_done }
// G = { gcd_done, gcd2_start }
gcd_reset:
       if gcd_start then
           goto M1/I <- 0
       else
          goto gcd_reset
       endif

// single ported memory..  one access per cycle.
// gcd2_A will be the memory read data register.
// I will be the memory address register.
gcd_M1:
       {gcd2_A <- N[I] , I <- I+1 }
gcd_M2:
       {gcd2_A <- N[I] , gcd2_B <- gcd2_A , I <- I+1}
gcd_gcd2:
       {gcd2_start = 1}
gcd_gcd2_wait:
       if gcd2_done then
          if(I==16) then
```

```
              goto gcd_done /
              {gcd_result <- gcd2_result}
           else
              goto gcd_gcd2 /
              { gcd2_A <- N[I] ,
                gcd2_b <- gcd2_result ,
                I <- (I+1) }
           endif
        else
           goto gcd_gcd2_wait
        endif

gcd_done:
        goto gcd_reset / {done = 1}



// Thread main
// Inputs = {din}
// O = {dout}
// Registers = {gcd_result, Count}
// Mem = {N[0:15]}
// E = { main_start, gcd_start, main_rdy}
// G = { main_done , gcd_done ,  env_rdy}
main_reset:
        if main_start then
            goto main_read / { Count <- 0}
        else
            goto main_reset
        endif

main_read:
        if Count == 16 then
            goto main_gcd/ {gcd_start = 1}
        else
            {main_rdy = 1}
            if(env_rdy) then
               goto main_read /
               {N[Count] <- din , Count <- Count+1}
            else
               goto main_Read
            endif
        endif

main_gcd:
        if (gcd_done) then
```

```
          goto main_completed /
           {dout <- gcd_result }
       endif

main_completed:
      goto main_reset / {main_done = 1}
```

## 4.2 The divider RTL pseudo-code

A direct implementation of the divider algorithm using long-division.

```
// Thread divider
//
// inputs = { dividend, divisor }
//      inputs are held constant throughout
//      the division operation.
//
// registers = { dividend, divisor, quotient,
//               remainder, curr_quotient,
//               shifted_divisor, dividend_by_2 }
//
// outputs = { quotient, remainder}
//        these are also registers.
//
// input control signals from "outside" = { div_start }
// output status signals to "outside" = { div_done  }
div_reset:
       if div_start then
          goto div_outerloop /
           {remainder <- dividend , quotient <- 0}
       else
          goto div_reset
       endif

div_outerloop:
       if (divisor <= remainder) then
          goto div_innerloop /
             { curr_quotient <- "0000000000000001" ,
               dividend_by_2 <- (remainder >> 1) ,
               shifted_divisor <- divisor}
       else
          goto div_completed
       endif

div_innerloop:
       if (shifted_divisor < dividend_by_2) then
```

```
          goto div_innerloop /
             { shifted_divisor <- (shifted_divisor << 1) ,
                curr_quotient <- (curr_quotient << 1) }
       else
          goto div_update
       endif

div_update:
       goto div_outerloop/
          {quotient <- (quotient | curr_quotient) ,
           remainder <- (remainder - shifted_divisor)}

div_completed:
       goto div_reset / {div_done = 1}
```