16 GCD - Greatest Common Divisor

Although being a small and worn out example, this circuit reveals some interesting verification problems: data dependent loops and data abstraction from bits and bit vectors to natural numbers. Moreover, if a more abstract specification is used (see Section 16.2.2 “High-Level Specification”) then even simple *program* verification tasks arise.

16.1 Introduction

The circuit simply computes the greatest common divider (GCD)of two natural numbers, given as bit vectors of fixed bit width.

16.2 Specification

In the following, two different specifications are given. The first “low level” version only requires an implementation verification, namely that the given algorithm to compute the GCD is correctly implemented. The second specification is based on the mathematical definition of the GCD and hence requires some sort of “program verification”, i.e. is has to be shown that the implemented algorithm really computes the GCD.

The circuit is based on a simple handshake protocol: It starts running after the signal **start** has been set to true, signalling that new input values are available at the inputs **A** and **B**. The latter carry binary encoded natural numbers. The **stop** signal indicates the termination of a computation.

16.2.1 Low-Level Specification

Given two natural numbers A and B (of fixed bit width). Compute the natural number *GCD* according to the algorithm of table 16-1.

```
X1:= Max(A,B);
X2:= Min(A,B);
repeat
M:= X1 MOD X2;
if (M ≠ 0) then
begin
X1:= X2;
X2:= M;
end
until (M=0)
GCD:= X2
```

Table 16-1: Computation of the GCD of two numbers A and B

16.2.2 High-Level Specification

Given two natural numbers A and B (of fixed bit width). Compute the natural number *GCD* with the following property:

$$\mathrm{Max}(\{g \mid ((g \in N) \text{and } g \text{ divides } A \text{ without rest and } G \text{ divides } B \text{ without rest})\})$$

16.3 Implementation

The implementation follows the specified algorithm (table 16-1). It is divided into control and data path. The data path computes (without controller interaction) the maximum of two n-bit vectors, to be interpreted as unsigned integers. Afterwards, the iteration of the algorithm is performed.

There are 4 signals between controller and data path. **StoreComp** activates the storage of the minimum/maximum sorted values in the data path. The sorting is performed by a comparator (Fig. 16-1).

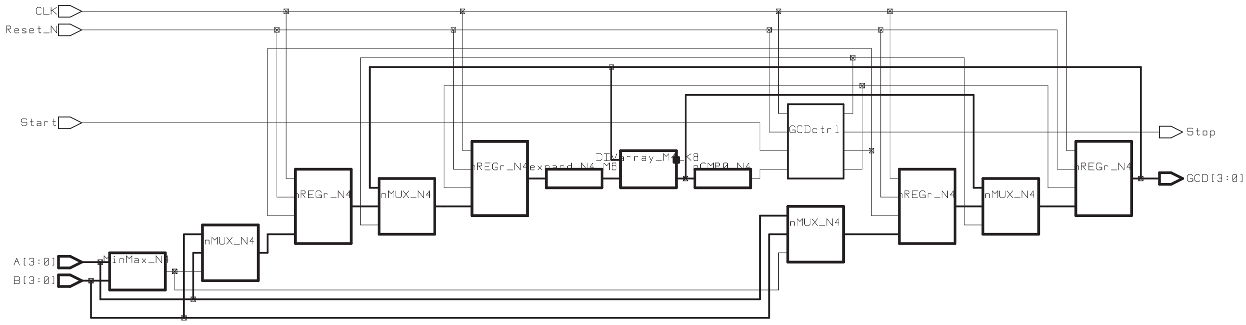

Figure 16-1: Controller and data path

SelectLoopInit determines, if there will be an additional loop computation or if new values are used for the loop. **StoreLoop** activates a register assignment, if the computation has not finished. The end of a computation is determined by **eq0**. All lines are active high.

16.3.1 Controller

The controller is specified by table 16-2. **Start** resets the controller, **Stop** signals the end of a computation. The FSM uses 3 states: 00 is the starting state after **Start** = 1. In state 01 the values of X1 and X2 are stored. Dependent on **eq0**, the FSM remains in 01 or jumps to the final state 10. States are stored in 2 D-flip-flops (q_0, q_1). State 11 is unreachable.

Start	eq0	state $(q_1q_0)^t$	next state $(q_1q_0)^{t+1}$	StoreComp	StoreLoop	SelectLoopInit	Stop
0	-[a]	00	01	0	1	0	0
0	0	01	01	-	1	0	0
0	1	01	10	-	0	0	1
0	-	10	10	-	0	0	1
1	-	--	00	1	1	1	0
0	-	11	10	-	0	0	1

Table 16-2: Controller transition table

a. “-” denotes don’t cares

Using table 16-2, the transition functions of table 16-3 result.

$$q_1^{t+1} = d_1^t = (\mathbf{Start}\ q1) \vee (\mathbf{Start}\ \mathbf{eq0}\ q_0)$$

$$q_0^{t+1} = d_0^t = (\mathbf{Start}\ \overline{q_1}\overline{q_0}) \vee (\mathbf{Start}\ \mathbf{eq0}\ \overline{q_1})$$

StoreComp = **Start**

Table 16-3: FSM transition functions

StoreLoop =	**Start** $\vee d_0^t$
SelectLoopInit =	**Start**
Stop =	d_1^t

Table 16-3: FSM transition functions

An implementation of the controller is given in Fig. 16-2.

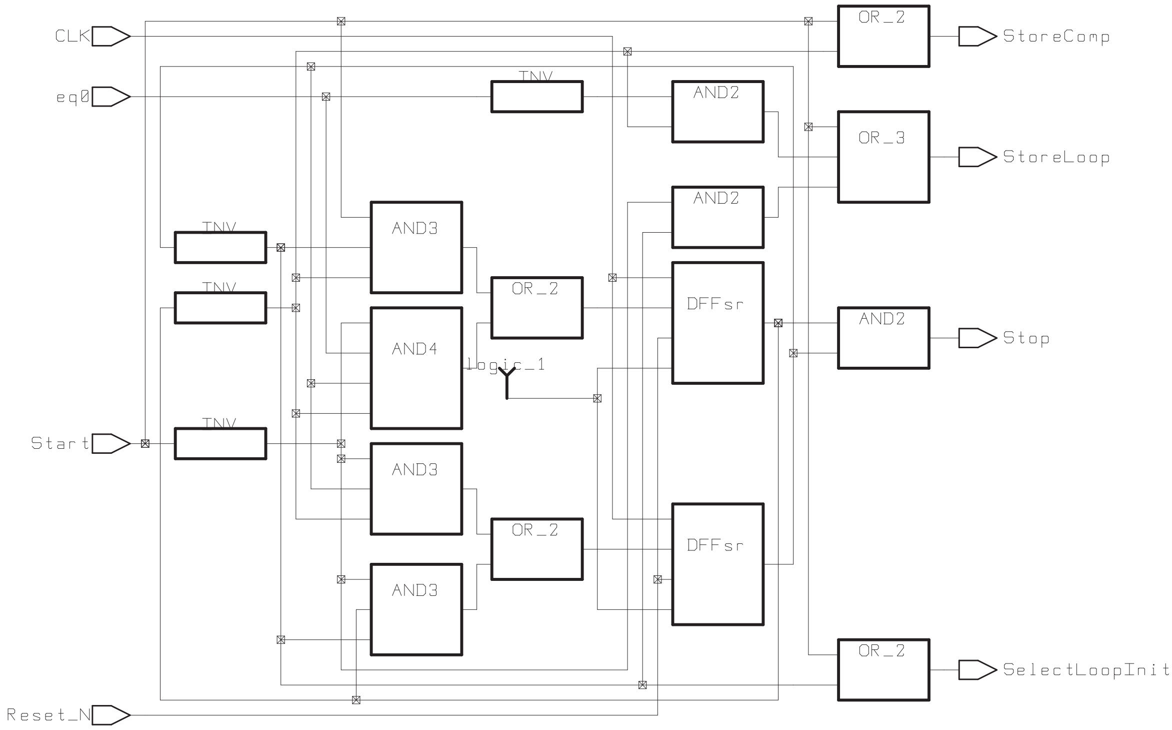

Figure 16-2: Controller realization

16.3.2 Data Path

The realization of the data path need the following modules: 4 n-bit multiplexer, 4 n-bit registers, 1 n-bit comparator, 1 n-bit divider and 1 n-bit “zero”-tester. These are connected as shown in (Fig. 16-1).

16.3.3 Simulations

In Fig. 16-3 two example computations are given. The GCD of 15_{10} (F_{16}) and 2 (result 1) and the GCD of 14_{10} (E_{16}) and 10_{10} (A_{16}) with result 2 are computed. Busses named GCD<3:0> and Div<7:0> carry divisor and dividend. Loop iterations occur on value transitions on these busses. The first computation needs 4 and the second computation needs 3 iterations.

To initialize the circuits, the Start and input signals have to be applied for a minimum duration of 2 clock cycles.

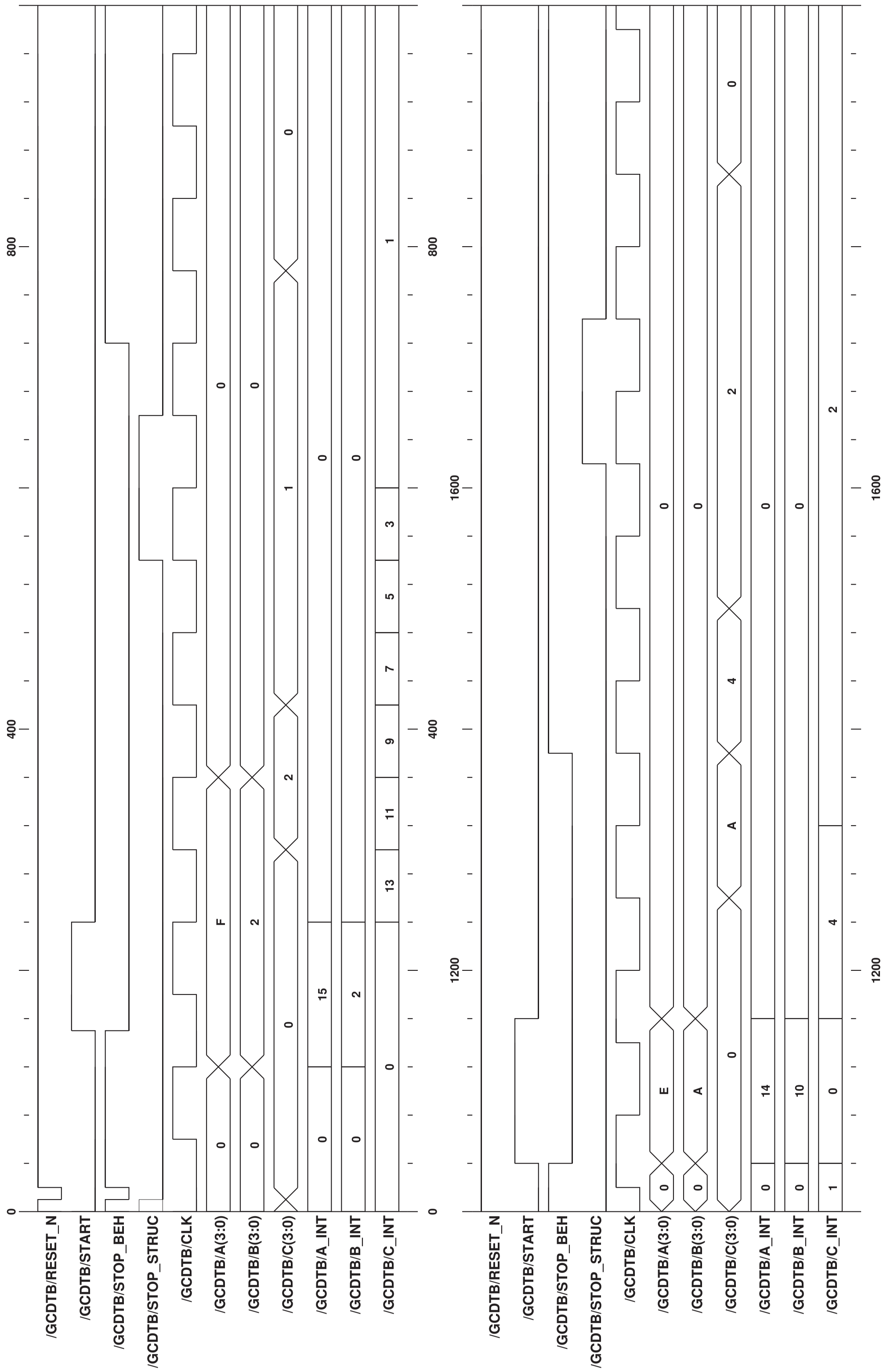

Figure 16-3: Waveforms for two GCD computations

16.4 Status and Acknowledgments

The GCD circuit is a variant of a high-level synthesis benchmark example [VRMK91]. The circuit presented here uses a ‘modulo” operation instead of repeated subtractions in order to put more emphasis on arithmetic. However, for uniformity reasons in a future release the algorithm may be adapted to the one used in [VRMK91].

Thanks to H.-P. Eich and C.-J. Thomas for designing the circuit in a commercial design system.

16.5 Literature

[VRMK91] R. Vemuri, J. Roy, P. Mamtora, and N. Kumar. Benchmarks for high-level synthesis. Technical Report ECE-DDE-91-11, Laboratory for Digital Design Environments, ECE Dept., University of Cincinnati, Ohio, USA, November 1991.