Verification of a $GF(2^m)$ Multiplier-Circuit for Digital Signal Processing*

Dirk W. Hoffmann, Thomas Kropf

Universität Karlsruhe
Institut für Rechnerstrukturen und Fehlertoleranz
Prof. Dr.-Ing. D. Schmid
76128 Karlsruhe, Germany
hoff@ira.uka.de kropf@ira.uka.de
http://goethe.ira.uka.de/hvg

Abstract

Hard-wired solutions for Finite Field Arithmetic have become increasingly important in recent years and are mostly part of domain specific Digital Signal Processors (DSPs). We have specified and verified a real-life example of an array-type multiplier for Finite Field multiplication in $GF(2^m)$ [1]. The multiplier has been specified in higher-order logic and correctness has been proven using the HOL98 theorem prover. Since our model is generic, the correctness results hold for arbitrary scaled circuits.

1 Introduction

Finite Field Arithmetic has various applications in telecommunication, i.e, coding theory and cryptography. In the past, different approaches for performing multiplication in Finite Fields have been presented [7, 4, 8], mostly based on sequential algorithms (shift-register type multipliers). Since the application domain for Finite Field Arithmetic is steadily increasing, "hard-wired", non-sequential solutions have been proposed. Most of them are realized in form of domain specific Digital Signal Processors (DSPs). Due to their lack in speed, sequential algorithms are not well suited for DSP integration where efficiency is by far the most important issue. Since it has become possible to integrate large circuits on a single chip without having an explosion in production costs, hard-wired non-sequential algorithms become more and more important.

In [1], a DSP multiplier unit has been proposed based on a two dimensional array structure. The circuit is user programmable and can therefore be applied to a much broader range of applications at the same time. This is in contrast to shift-register type multipliers that can only perform multiplications based on a fixed primitive irreducible polynomial (see Section 2).

*This work is supported by the ESPRIT LTR Project 26241

In this paper, we have completely specified the multiplier-circuit as described in [1] using higher-order logic. The circuit description is generic and the proved theorems hold for multipliers of arbitrary size. All proofs have been derived using the HOL theorem prover [3]. While constructing the proofs, we have discovered an error in the circuit architecture. The error was due to a missing gate in the layout for a single multiplier-cell (Fig. 3 in [1]).

Our paper is organized as follows: In Section 2, we give a brief introduction to Finite Field Theory, Section 3 presents the multiplier architecture and Section 4 describes how verification has been performed. We close our paper with a summary in Section 5 and some remarks about further research.

2 Finite Field Theory

Every field containing a finite number of elements is called a Finite Field or Galois Field. Let $\mathcal{F}$ denote an arbitrary Finite Field. It can be shown that

$$|\mathcal{F}| = p^m \tag{1}$$

holds for some prime number p and integer $m, m \geq 1$. Moreover, given any prime p and integer $m, m \geq 1$, there exists a Finite Field with exactly p^m elements.

Two Finite Fields $\mathcal{F}$ and $\mathcal{G}$ with $|\mathcal{F}| = |\mathcal{G}|$ can always proven to be isomorphic. Thus, we define $GF(p^m)$ to be *the* Finite Field with p^m elements.

If $m = 1$, $GF(p^m)$ is isomorphic to $(\{1, \ldots, p\}, \oplus, \otimes)$ with

$$a \oplus b = \quad (a + b) \mod p \tag{2}$$

$$a \otimes b = \quad (a \cdot b) \mod p \tag{3}$$

However, for $m \geq 2$, the set $\{1, \ldots, p^m\}$ does no longer form a field with standard addition and multiplication modulo p^m. In particular, the multiplicative group of $GF(p^m)$ turns out to become slightly harder to characterize.

It can be shown [5] that $GF(p^m)$ is an extension field of $GF(p)$ and

$$GF(p^m) \cong GF(p)[x]/f \tag{4}$$

where $f \in GF(p)[x]$ is a primitive irreducible polynomial of the form

$$f(x) = x^m + f_{m-1}x^{m-1} + \ldots + f_1 x + f_0 \tag{5}$$

Thus, $GF(p^m)$ is isomorphic to the set of polynomials over $GF(p)$ reduced modulo f. Addition and multiplication are the standard operators for polynomials.

Moreover, it turns out that every root α of f has the property that it *generates* $GF(p^m)$. Thus, every element of $GF(p^m)$ can be represented as a power of α. Using the fact that $f(\alpha) = 0$, we get

$$\alpha^m = (-f_{m-1})\alpha^{m-1} + \ldots + (-f_1)\alpha + (-f_0) \tag{6}$$

Therefore, we can represent every element of $GF(p^m)$ with a polynomial in α having a degree less than m. Successively applying equation (6) to a polynomial

of degree greater or equal m finally yields in a polynomial of degree less than m which is element of $GF(p^m)$. In other words, equation (6) can be exploited to perform modulo computation.

For the rest of this paper, we restrict ourselves to Galois Fields $GF(2^m)$ (extension fields of $GF(2)$) which are mostly used in digital systems.

Let

$$\begin{aligned} a &= a_0 + a_1\alpha + \ldots + a_{m-1}\alpha^{m-1} \\ b &= b_0 + b_1\alpha + \ldots + b_{m-1}\alpha^{m-1} \end{aligned}$$

denote two elements of $GF(2^m)$. Then, we get

$$a + b = \sum_{i=0}^{m-1} a_i\alpha^i + \sum_{i=0}^{m-1} b_i\alpha^i \tag{7}$$

$$= \sum_{i=0}^{m-1} (a_i + b_i)\alpha^i = \sum_{i=0}^{m-1} (a_i \text{ XOR } b_i)\alpha^i \tag{8}$$

Hence, addition in $GF(2^m)$ is equivalent to component-wise application of the XOR-operation and can be implemented by a chain of independent XOR-gates.

In the more complicated case of multiplication, we get

$$a \cdot b = \left(a \cdot \sum_{i=0}^{m-1} b_i\alpha^i\right) \bmod f = \sum_{i=0}^{m-1} \left((a \cdot b_i)\alpha^i\right) \bmod f \tag{9}$$

$$= (((a \cdot b_{m-1})\,\alpha \bmod f + a \cdot b_{m-2})\,\alpha \bmod f + \ldots)\,\alpha \bmod f + a \cdot b_0 \tag{10}$$

In (10), multiplication in $GF(2^m)$ has been reduced to a formula only involving multiplication of a polynomial with α, multiplication with a scalar value, and modulo computation.

Multiplication with α is just a left shift and scalar multiplication with $v \in GF(2)$ can be computed as follows:

$$a \cdot v = \left(\sum_{i=0}^{m-1} a_i\alpha^i\right) v = \sum_{i=0}^{m-1} (a_i \cdot v)\,\alpha^i = \sum_{i=0}^{m-1} (a_i \wedge v)\,\alpha^i \tag{11}$$

According to (11), scalar multiplication is nothing else than component-wise conjunction with v and can be performed with a chain of independent AND-gates.

A closer look to equation (10) also shows that the involved modulo-operation is only applied to polynomials with degree less or equal m. In the former case, the polynomial is already reduced. If the degree equals m, we can perform modulo computation according to equation (6) as shown below:

$$\left(\alpha^m + \sum_{i=0}^{m-1} a_i\alpha^i\right) \bmod f = \sum_{i=0}^{m-1} f_i\alpha^i + \sum_{i=0}^{m-1} a_i\alpha^i \tag{12}$$

$$= \sum_{i=0}^{m-1} (f_0 + a_i)\,\alpha^i = \sum_{i=0}^{m-1} (f_0 \text{ XOR } a_i)\,\alpha^i \tag{13}$$

Again, modulo computation of polynomials of degree m can be achieved by simply adding the minimal irreducible polynomial f.

For a more detailed introduction to Finite Field Arithmetic, see [5] or [2].

3 Circuit Architecture

The circuit architecture presented in [1] can be viewed as a straightforward realization of the theoretical results presented in Section 2. The circuit is shown in Figure 1.

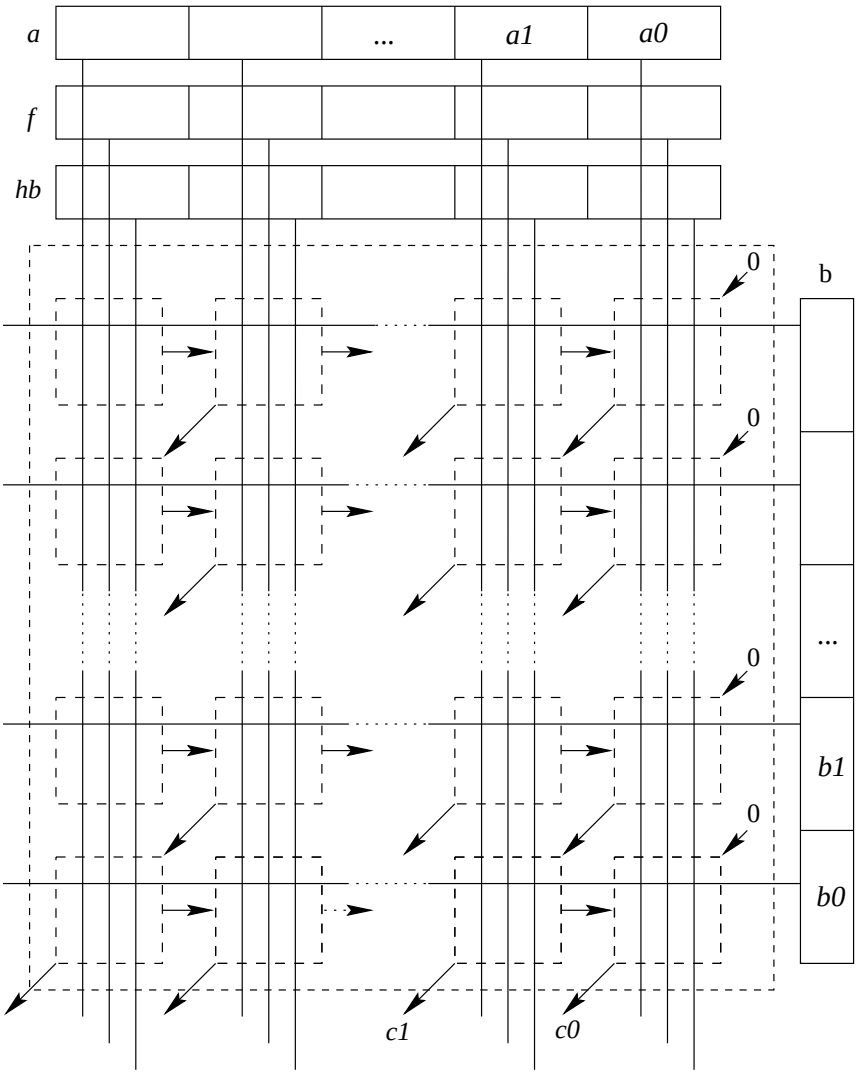

Figure 1: Architecture of the array multiplier

The multiplier consists of input vectors a, f, hb, b, and output vector c which contains the computed result. a and b store the operands to be multiplied while the rightmost cell in a contains coefficient a_0 and the uppermost cell of b contains b_{m-1}. f contains the coefficients of the primitive irreducible polynomial and hb is functioning as a highest-bit-locator. If f has degree m, hb is true in column m and false everywhere else. Finally, vector c returns the coefficients of the product $a \cdot b$.

The main component of the multiplier is the two dimensional cell-array

in the middle of Fig. 1. Each cell has five input wires a, b, d, hb, yi, and two output wires yo and c. Fig. 2 shows the architecture of a single cell in more detail. The two XOR-gates and the AND-gate in the lower right corner perform component-wise multiplication and addition, respectively. The gates in the upper left corner form an overflow detector and determine when the modulo operation has to be applied. Once an overflow has been detected, the output yo propagates the signal through all other cells in the same row.

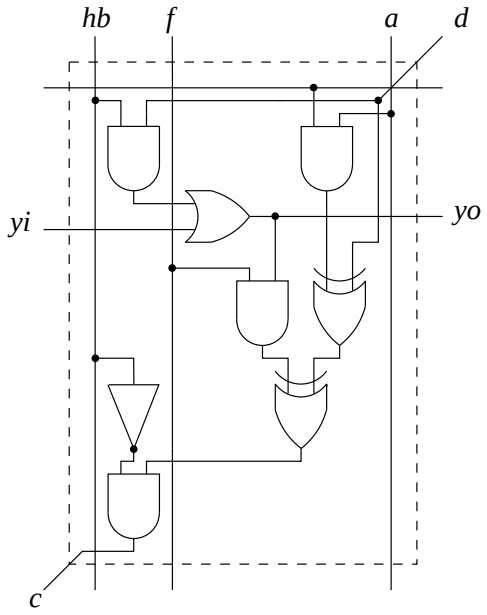

Figure 2: A single multiplier cell

4 Specification and Verification

In this section, we first describe how the circuit proposed in Section 3 has been specified in higher-order logic. Then, we formulate the specification stating the behavioral correctness of the circuit and briefly explain how correctness has been proven using the HOL98 theorem prover.

4.1 Implementation Description

In HOL, we represent elements of $GF(2)$ by variables of type `bool` while elements of $GF(2^m)$ are modeled with type `num->bool`. The circuit is described in a modular way (see Fig. 3) according to the architecture definition in Section 3.

We define three predicates

- `CELL`,
- `ROW`, and

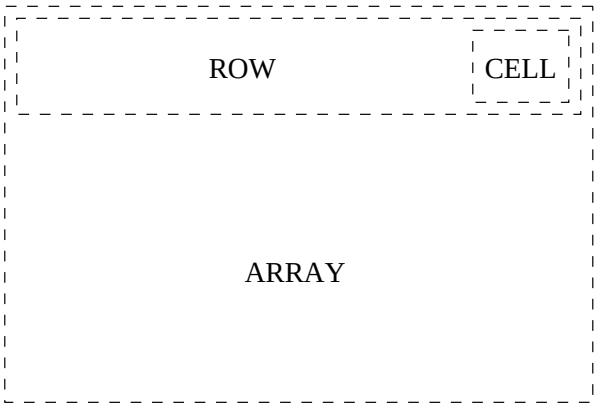

Figure 3: Hierarchy of the array multiplier

- `ARRAY`

stating the relationship between input and output values of the specific component.

4.1.1 Cell:

The `CELL`-predicate is defined as follows:

```
CELL (a:bool) (b:bool) (d:bool) (f : bool) (hb : bool)
                                (yi:bool) (yo:bool) (c:bool)
=  ?s1 s2 s3 s4 s5 s6.(
                          (AND_GATE hb d s1)  /\
                          (OR_GATE yi s1 yo)  /\
                          (AND_GATE a b s2)   /\
                          (XOR_GATE s2 d s3)  /\
                          (AND_GATE f yo s4)  /\
                          (XOR_GATE s3 s4 s5) /\
                          (NOT_GATE hb s6)    /\
                          (AND_GATE s6 s5 c)
                       )
```

The definition exactly mirrors the circuit layout given in Fig. 2. The existentially quantified variables `s1` to `s6` represent intermediate signals connecting the different gates. The predicates `NOT_GATE`, `AND_GATE`, `OR_GATE`, `XOR_GATE` are defined as usual, e.g., `XOR_GATE` is defined as

```
(XOR_GATE X Y Z) = (Z = (X /\ ~Y) \/ (~X /\ Y))
```

4.1.2 Row:

The `ROW`-predicate characterizes one single horizontal chain of cells and states the relationship between coefficient vectors a, d, f, hb, c and single coefficient b_i:

```
ROW (a:num->bool) (b: bool) (d:num->bool) (f:num->bool)
                                  (hb:num->bool) (c:num->bool)
= ?(y:num->bool). !(n:num).
   (CELL (a n) b (d n) (f n) (hb n) (y (SUC n)) (y n) (c n))
```

The existentially quantified variable `y` represents the intermediate signals connecting output yo of cell $n+1$ with input yi of cell n.

4.1.3 Array:

The `ARRAY`-predicate represents the whole multiplier array according to Fig. 1:

```
(array 0 a b f hb c
     = ROW a (b 0) (\n.F) f hb c) /\

(array (SUC n) a b f hb c
     = ?(c2:num->bool) (c3:num->bool).
         (array n a b f hb c2) /\
         (LEFT_SHIFT c2 c3) /\
         (ROW a (b (SUC n)) c3 f hb c))
```

`ARRAY` is recursively defined with recursion variable n measuring the number of rows in the circuit. The base case $n = 0$ corresponds to a multiplier array containing one single row of cells. Predicate `LEFT_SHIFT` is used to specify the connecting wires between outputs of row n and inputs of row $n+1$.

4.2 Specification Description

To specify the correct behavior of the multiplier circuit, we define four functions

```
GFadd         : (num->bool)->(num->bool)->(num->bool)
GFscalar_mult : (num->bool)->bool->(num->bool)
GFmultX       : (num->bool)->(num->bool)
GFmodf        : (num->bool) (num->bool) (num)
```

performing addition, multiplication with scalar values, multiplication with α, and modulo computation, respectively. Function `GFmodf` takes an additional variable of type `num` containing the degree of f. All function definitions are fairly straightforward and based on the derived equations in Section 2. For example, addition in $GF(2^m)$ corresponds to a bitwise XOR-operation according to equation (8). In HOL, we define

```
val GFadd = new_definition
  ("GFadd",
   --‘GFadd (X:num->bool) (Y:num->bool)
        = \(n:num). (((X n) /\ ~(Y n)) \/ (~(X n) /\ (Y n)))‘--);
```

Multiplication with a scalar value, multiplication with x, and modulo computation are expressed similarly.

The main theorem we want to prove about the multiplier is

Theorem 1
Let c^m denote the output of row number m, $m \geq 0$. Then,

$$c^m = ((((a \cdot b_{m-1})\,\alpha \bmod f + a \cdot b_{m-2})\,\alpha \bmod f + \ldots)\,\alpha \bmod f + a \cdot b_0 \quad (14)$$

In terms of HOL-logic, the right side of formula (14) can be expressed recursively as shown below:

```
(GFproduct 0
             (a:num->bool) (b:num->bool) (f:num->bool) (m:num)
     = (GFscalar_mult a (b 0))) /\

(GFproduct (SUC n)
             (a:num->bool) (b:num->bool) (f:num->bool) (m:num)
     = (GFadd (GFmodf (GFmultX (GFproduct n a b f m)) f m)
                               (GFscalar_mult a (b (SUC n)))))
```

Hence, theorem 1 can be stated as

```
!n a b f hb m. ASS ==> array n a b f hb (GFproduct n a b f m)
```

where `ASS` is a list of assumptions we have to make about the input vectors a, f, and hb. More precisely, for computations in $GF(2^m)$, we need to postulate the following assumptions:

- the highest coefficient of f is m
- hb_n equals 1 if and only if $n = m$
- $a_n = 0$ for $n \geq m$

Using HOL, the assumptions are written as

```
(f m = T) /\
(!n. ( (n > m) ==> (f n = F))) /\
(hb m = T) /\
(!n. n > m ==> (hb n = F)) /\
(!n. n < m ==> (hb n = F)) /\
(a m = F) /\
(!n. n > m ==> (a m = F))
```

Theorem 1 can be proven by induction on parameter n. We get two major subgoals:

```
!a b f hb m. ASS ==> array 0 a b f hb (GFproduct 0 a b f m)
```

and

```
(!a b f hb m. ASS ==> array n a b f hb (GFproduct n a b f m))
  ==> (!a b f hb m. ASS ==>
          array (SUC n) a b f hb (GFproduct (SUC n) a b f m))
```

For both the base case and the induction step, we make use of an intermediate lemma about the `ROW`-predicate:

Lemma 2
For each row with inputs $a, f, hb \in GF(2^m)$, $b \in GF(2)$ *and output* c,

$$c = (d \ mod \ f) + a \cdot b \tag{15}$$

Lemma 2 is equivalent to the following HOL theorem:

```
!a b d f hb m. ASS ==>
   ROW a b d f hb (GFadd (GFmodf d f m) (GFscalar_mult a b))
```

The proof of lemma 2 mainly consists of term rewrites and two user guided case splits. At the end, most subgoals are statements about the CELL predicate that can be solved automatically by applying a user defined HOL tactic.

While working on the proofs, we have captured an error in the circuit design due a missing NOT-gate in the cell-layout[1].

If we have a closer look at the assumption list, we note that `(a m)` is assumed to store `F`. In theory, for computations in $GF(2^m)$, only the coefficients a_0 to a_{m-1} have to be taken into account since a is considered to be a reduced polynomial. However, given the cell-layout in Fig. 2, the contents of a_m influences the computed result and the assumption `(a m = F)` is indispensable for the correct circuit behavior. Otherwise, theorem 1 does no longer hold.

An important property of our specification is its generic nature. The circuit has been specified for arbitrary size, i.e., even for infinite input vectors a.

We have made the observation that the unconstraint size of the array has considerably simplified the proof instead of making it more complicated. Similar experiences with other theorem provers have been reported in [6].

5 Summary

In this paper, we have formally specified and proven a real-life DSP multiplier circuit presented in [1] performing multiplication in Finite Fields $GF(2^m)$. Unlike conventional shift-register type multipliers the circuit is user programmable and well suited for DSP integration due to its very high efficiency.

The circuit has been specified in higher-order logic using the HOL98 theorem prover. While proving the behavioral correctness, we have captured a missing gate in the circuit architecture.

Our specification is generic and the correctness results hold for multipliers of arbitrary size.

Although a lot of subgoals can be proven automatically because of their propositional nature, a fairly high amount of user guidance is still needed. The PROSPER project[2] aims on the integration of different proof tools in a higher-order logic environment to achieve a higher degree of automation. The

[1] Fig. 2 already shows the fixed cell architecture with the missing NOT-gate added
[2] http://www.dcs.gla.ac.uk/prosper/

multiplier circuit has been chosen to serve as one of the benchmark examples for PROSPER to evaluate its practical strength.

Furthermore, we plan to apply our verification approach to more complex circuits for Finite Field Arithmetic like inversion, division, or GCD-computation.

References

[1] Wolfram Drescher and Gerhard Fettweis. VLSI architectures for multiplication in $GF(2^m)$ for application tailored digital signal processors. In *Workshop on VLSI Signal Processing IX, San Francisco / CA*, 1996.

[2] R.J. Mc Eliece. *Finite Fields for Computer Scientists and Engineers.* Kluwer Academic Publishers, Boston, MA, 1987.

[3] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[4] S. Lin and D.J. Costello, Jr. *Error Control Coding: Fundamentals and Applications.* Prentice Hall, 1983.

[5] J. Lipson, editor. *Elements of Algebra and Algebraic Computing.* The Benjamin/Cummings Publishing Company, Inc., 1981.

[6] J.S. Moore. Ongoing commercial applications of the ACL2 theorem prover. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Scienc*, pages 357–368. Springer-Verlag, August 1998.

[7] C.C. Wang, T.K. Troung, H.M. Shao, L.J. Deutsch, and I.S.Reed. VLSI architectures for computing multiplications and inverses in $GF(2^m)$. *IEEE Transactions on Computers*, c-34:709–717, August 1985.

[8] C.S. Yeh, I.S. Reed, and T.K. Troung. Systolic multipliers for finite fields $GF(2^m)$. *IEEE Transactions on Computers*, c-33:pp. 357–360, April 1984.

A Appendix: HOL Proof-Script

```
(* ------------------------------------------------------------ *)
(* Verification of an multiplier-circuit                        *)
(* for finite fields GF(2^m)                                    *)
(*                                                              *)
(* Dirk Hoffmann                                                *)
(* Institut fuer Rechnerentwurf und Fehlertoleranz              *)
(* Universitaet Karlsruhe                                       *)
(* hoff@ira.uka.de                                              *)
(* http://goethe.ira.uka.de/hvg                                 *)
(*                                                              *)
(* 31.7.98                                                      *)
(*                                                              *)
(* requires HOL98                                               *)
(* ------------------------------------------------------------ *)

app load ["decisionLib","tautLib"];

open decisionLib;
open tautLib;

val num_Axiom = prim_recTheory.num_Axiom;
val INDUCT_TAC = INDUCT_THEN numTheory.INDUCTION ASSUME_TAC;


new_theory "GFmult";


(* ------------------------------------------------------------ *)
(* Circuit description                                          *)
(* ------------------------------------------------------------ *)

val NOT_GATE = new_definition
  ("NOT_GATE",
  --`(NOT_GATE X Y) = (Y = ~X)`--);

val AND_GATE = new_definition
  ("AND_GATE",
   --`(AND_GATE X Y Z) = (Z = X /\ Y)`--);

val OR_GATE = new_definition
  ("OR_GATE",
   --`(OR_GATE X Y Z) = (Z = X \/ Y)`--);

val XOR_GATE = new_definition
  ("XOR_GATE",
```

```
   --‘(XOR_GATE X Y Z) = (Z = (X /\ ~Y) \/ (~X /\ Y))‘--);

val LEFT_SHIFT = new_definition
  ("LEFT_SHIFT",
  --‘(LEFT_SHIFT (X:num->bool) (Y:num->bool)) =
       !n. ((Y 0 = F) /\ (Y (SUC n) = X n))‘--);

val CELL = new_definition
  ("CELL",
   --‘CELL (a:bool) (b:bool) (d:bool) (f : bool)
              (hb : bool) (yi:bool) (yo:bool) (c:bool)
   =    (?s1 s2 s3 s4 s5 s6.((AND_GATE hb d s1) /\
                             (OR_GATE yi s1 yo) /\
                             (AND_GATE a b s2)  /\
                             (XOR_GATE s2 d s3) /\
                             (AND_GATE f yo s4) /\
                             (XOR_GATE s3 s4 s5) /\
                             (NOT_GATE hb s6) /\
                             (AND_GATE s6 s5 c)))‘--);

val ROW = new_definition
  ("ROW",
   --‘ROW (a:num->bool) (b: bool) (d:num->bool)
             (f:num->bool) (hb:num->bool) (c:num->bool)
   = ?(y:num->bool). !(n:num).
     (CELL (a n) b (d n) (f n) (hb n) (y (SUC n))
                                            (y n) (c n))‘--);

val array = new_recursive_definition
  {name="array",
   fixity=Prefix,
   rec_axiom= num_Axiom,
   def =
   --‘(array 0 a b f hb c = ROW a (b 0) (\n.F) f hb c) /\
      (array (SUC n) a b f hb c
        = ?(c2:num->bool) (c3:num->bool).
            (array n a b f hb c2) /\
            (LEFT_SHIFT c2 c3) /\
            (ROW a (b (SUC n)) c3 f hb c))‘--};

(* ------------------------------------------------------------ *)
(* Specification                                                *)
(* ------------------------------------------------------------ *)


(* A R I T H M E T I C   F U N C T I O N S                      *)
```

```
(* Addition in GF(2^n) = component-wise XOR                    *)

val GFadd = new_definition
  ("GFadd",
   --‘GFadd (X:num->bool) (Y:num->bool)
        = \(n:num). (((X n) /\ ~(Y n)) \/ (~(X n) /\ (Y n)))‘--);

(* Scalar multiplication in GF(2^n) = component-wise AND       *)

val GFscalar_mult = new_definition
  ("GFscalar_mult",
   --‘GFscalar_mult (X:num->bool) (scalar : bool)
        = \(n:num). ((X n) /\ scalar)‘--);

(* Multiplication with 'x' = left shift                        *)

val GFmultX = new_definition
  ("GFmultX",
  --‘GFmultX (A:num->bool)
       = \(n:num). (n = 0) => F | (A (PRE n))‘--);

val GFmodf = new_definition
  ("GFmodf",
  --‘GFmodf (X:num->bool) (f:num->bool) (degree_f:num)
       = ((X degree_f) => (GFadd X f) | X)‘--);


val GFproduct = new_recursive_definition
  {name="GFproduct",
   fixity = Prefix,
   rec_axiom=num_Axiom,
   def = --‘(GFproduct 0 (a:num->bool) (b:num->bool)
                                       (f:num->bool) (m:num)
        = (GFscalar_mult a (b 0))) /\
     (GFproduct (SUC n) (a:num->bool) (b:num->bool)
                                       (f:num->bool) (m:num)
        = (GFadd (GFmodf (GFmultX (GFproduct n a b f m)) f m)
                          (GFscalar_mult a (b (SUC n)))))‘--};


(* ---------------------------------------------------------- *)
(* A S S U M P T I O N S                                      *)
(* ---------------------------------------------------------- *)

val asm = --‘(f m = T) /\
             (!n. ( (n > m) ==> (f n = F))) /\
```

```
            (hb m = T) /\
            (!n. n > m ==> (hb n = F)) /\
            (!n. n < m ==> (hb n = F)) /\
            (a m = F) /\
            (!n. n > m ==> (a n = F))‘--;


(* ------------------------------------------------------------ *)
(* T H E O R E M S                                              *)
(* ------------------------------------------------------------ *)

val mod0 = prove(--‘!f m. (GFmodf (\n.F) f m = (\n.F))‘--,
                REWRITE_TAC[GFmodf]);

val add0 = prove(--‘!(x:num->bool). (GFadd (\n.F) x = x)‘--,
     STRIP_TAC
     THEN REWRITE_TAC[GFadd,
                      (ETA_CONV(Term ‘\n.(x:num->bool) n‘))]);

val LEFT_SHIFT_lemma =
  let
      val stdrw = DECIDE (--‘(!n. (PRE (SUC n) = n)) /\
                             (!n. (((SUC n) = 0) = F))‘--);
  in
      prove (--‘!(X:num->bool). LEFT_SHIFT X (GFmultX X)‘--,
             REWRITE_TAC[LEFT_SHIFT,GFmultX]
             THEN BETA_TAC
             THEN REWRITE_TAC [stdrw])
  end;

fun MY_SPLIT_TAC cond =
    let
        val cond_thm = DECIDE cond;
    in
    (ASSUME_TAC cond_thm)
       THEN UNDISCH_TAC cond
       THEN STRIP_TAC
    end;

fun CELL_TAC a b d f hb yi =
  let
       val XOR = new_definition
           ("XOR", --‘XOR a b = (a /\ ~b) \/ (~a /\ b)‘--);
  in

  REWRITE_TAC[CELL]
    THEN EXISTS_TAC (Term ‘^hb /\ ^d‘)
```

```
    THEN EXISTS_TAC (Term `^a /\ ^b`)
    THEN EXISTS_TAC (Term `XOR (^a /\ ^b) ^d`)
    THEN EXISTS_TAC (Term `^f /\ (^yi \/ (^d /\ ^hb))`)
    THEN EXISTS_TAC (Term `XOR (XOR (^a /\ ^b) ^d)
                               (^f /\ (^yi \/ (^d /\ ^hb)))`)
    THEN EXISTS_TAC (Term `~(^hb)`)
    THEN REWRITE_TAC[XOR,NOT_GATE,AND_GATE,XOR_GATE,OR_GATE]
    THEN TAUT_TAC
  end;


val ROW_lemma =
  let
      val t = --`T`--;
      val f = --`F`--;
      val b = --`b:bool`--;
      val a_n = --`(a:num->bool) n`--;
      val f_n = --`(f:num->bool) n`--;
      val d_n = --`(d:num->bool) n`--;

      val witness = --`\(n:num). ((n > m) => F | (d m))`--;

      val num1 = prove (--`!n m. n < m ==> ((SUC n > m) = F)`--,
                  DECIDE_TAC);
      val num2 = prove (--`!n m. n < m ==> ((n > m) = F)`--,
                  DECIDE_TAC);
      val num3 = prove (--`!n m. n > m ==> ((SUC n) > m)`--,
                  DECIDE_TAC);

      val std_rw = prove (--`(SUC n > n = T) /\
                             (n > n = F) /\
                             (n < n = F)`--,DECIDE_TAC);
  in
  store_thm ("ROW_lemma",
  --`!a b d f hb m. (^asm ==> ROW a b d f hb
                 (GFadd (GFmodf d f m) (GFscalar_mult a b)))`--,

  (REPEAT STRIP_TAC
     THEN REWRITE_TAC[ROW]
     THEN EXISTS_TAC witness
     THEN STRIP_TAC
     THEN (MY_SPLIT_TAC
             (Term `(n = m) \/ (n < m) \/ (n > m)`))
     THEN (MY_SPLIT_TAC
             (Term `(d (m:num) = T) \/ (d (m:num) = F)`))
     THEN ASM_REWRITE_TAC[std_rw,GFmodf,GFscalar_mult,GFadd]
     THEN BETA_TAC
```

```
    THEN ASM_REWRITE_TAC[std_rw]
    THEN ASSUME_TAC num1
    THEN ASSUME_TAC num2
    THEN ASSUME_TAC num3
    THEN RES_TAC
    THEN ASM_REWRITE_TAC[]
    THENL [(CELL_TAC f b t t t f),
           (CELL_TAC f b f t t f),
           (CELL_TAC a_n b d_n f_n f t),
           (CELL_TAC a_n b d_n f_n f f),
           (CELL_TAC f b d_n f f f),
           (CELL_TAC f b d_n f f f)]))
  end;

val MAIN_THM =
  let
      val ROW_lem_spec1 = PURE_REWRITE_RULE [mod0,add0]
                           (SPECL [(Term `a:num->bool`),
                             (Term `((b:num->bool) 0)`),
                             (Term `\(n:num).F`),
                             (Term `f:num->bool`),
                             (Term `hb:num->bool`),
                             (Term `m:num`)]
                             ROW_lemma);
      val ROW_lem_spec2 = SPECL [(Term `a:num->bool`),
                             (Term `((b:num->bool) (SUC n))`),
                             (Term `(GFmultX
                                     (GFproduct n a b f m))`),
                             (Term `f:num->bool`),
                             (Term `hb:num->bool`),
                             (Term `m:num`)]
                             ROW_lemma;

      val witness1 = --`GFproduct n a b f m`--;
      val witness2 = --`GFmultX (GFproduct n a b f m)`--;

  in
  store_thm("MAIN_THM",
  --`!n a b f hb m.
        ^asm ==> array n a b f hb (GFproduct n a b f m)`--,

  INDUCT_TAC
     THEN REPEAT STRIP_TAC
     THEN REWRITE_TAC[array,GFproduct]
     THENL [(ASSUME_TAC ROW_lem_spec1)
               THEN RES_TAC,
            (EXISTS_TAC witness1)
```

```
              THEN (EXISTS_TAC witness2)
              THEN REPEAT STRIP_TAC]
     THENL [RES_TAC,
            (ASSUME_TAC LEFT_SHIFT_lemma),
            ASSUME_TAC ROW_lem_spec2
              THEN RES_TAC]
     THEN ASM_REWRITE_TAC[])
  end;


print_theory ();
export_theory ();
```