Non-Restoring Integer Square Root: A Case Study in Design by Principled Optimization

John O'Leary[1], Miriam Leeser[1], Jason Hickey[2], Mark Aagaard[1]

[1] School Of Electrical Engineering
Cornell University
Ithaca, NY 14853

[2] Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract. Theorem proving techniques are particularly well suited for reasoning about arithmetic above the bit level and for relating different levels of abstraction. In this paper we show how a non-restoring integer square root algorithm can be transformed to a very efficient hardware implementation. The top level is a Standard ML function that operates on unbounded integers. The bottom level is a structural description of the hardware consisting of an adder/subtracter, simple combinational logic and some registers. Looking at the hardware, it is not at all obvious what function the circuit implements. At the top level, we prove that the algorithm correctly implements the square root function. We then show a series of optimizing transformations that refine the top level algorithm into the hardware implementation. Each transformation can be verified, and in places the transformations are motivated by knowledge about the operands that we can guarantee through verification. By decomposing the verification effort into these transformations, we can show that the hardware design implements a square root. We have implemented the algorithm in hardware both as an Altera programmable device and in full-custom CMOS.

1 Introduction

In this paper we describe the design, implementation and verification of a subtractive, non-restoring integer square root algorithm. The top level description is a Standard ML function that implements the algorithm for unbounded integers. The bottom level is a highly optimized structural description of the hardware implementation. Due to the optimizations that have been applied, it is very difficult to directly relate the circuit to the algorithmic description and to prove that the hardware implements the function correctly. We show how the proof can be done by a series of transformations from the SML code to the optimized structural description.

At the top level, we have used the Nuprl proof development system [Lee92] to verify that the SML function correctly produces the square root of the input.

We then use Nuprl to verify that transformations to the implementation preserve the correctness of the initial algorithm.

Intermediate levels use Hardware ML [OLLA93], a hardware description language based on Standard ML. Starting from a straightforward translation of the SML function into HML, a series of transformations are applied to obtain the hardware implementation. Some of these transformations are expressly concerned with optimization and rely on knowledge of the algorithm; these transformations can be justified by proving properties of the top-level description.

The hardware implementation is highly optimized: the core of the design is a single adder/subtracter. The rest of the datapath is registers, shift registers and combinational logic. The square root of a $2n$ bit wide number requires n cycles through the datapath. We have two implementations of square root chips based on this algorithm. The first is done as a full-custom CMOS implementation; the second uses Altera EPLD technology. Both are based on a design previously published by Bannur and Varma [BV85]. Implementing and verifying the design from the paper required clearing up a number of errors in the paper and clarifying many details.

This is a good case study for theorem proving techniques. At the top level, we reason about arithmetic operations on unbounded integers, a task theorem provers are especially well suited for. Relating this to lower levels is easy to do using theorem proving based techniques. Many of the optimizations used are applicable only if very specific conditions are satisfied by the operands. Verifying that the conditions hold allows us to safely apply optimizations.

Automated techniques such as those based on BDDs and model checking are not well-suited for verifying this and similar arithmetic circuits. It is difficult to come up with a Boolean statement for the correctness of the outputs as a function of the inputs and to argue that this specification correctly describes the intended behavior of the design. Similarly, specifications required for model checkers are difficult to define for arithmetic circuits.

There have been several verifications of hardware designs which lift the reasoning about hardware to the level of integers, including the Sobel Image processing chip [NS88], and the factorial function [CGM86]. Our work differs from these and similar efforts in that we justify the optimizations done in order to realize the square root design. The DDD system [BJP93] is based on the idea of design by verified transformation, and was used to derive an implementation of the FM9001 microprocessor. High level transformations in DDD are not verified by explicit use of theorem proving techniques.

The most similar research is Verkest's proof of a non-restoring division algorithm [VCH94]. This proof was also done by transforming a design description to an implementation. The top level of the division proof involves consideration of several cases, while our top level proof is done with a single loop invariant. The two implementations vary as well: the division algorithm was implemented on an ALU, and the square root on custom hardware. The algorithms and implementations are sufficiently similar that it would be interesting to develop a single verified implementation that performs both divide and square root based on the research in these two papers.

The remainder of this paper is organized as follows. In section 2 we describe the top-level non-restoring square root algorithm and its verification in the Nuprl proof development system. We then transform this algorithm down to a level suitable for modelling with a hardware description language. Section 3 presents a series of five optimizing transformations that refine the register transfer level description of the algorithm to the final hardware implementation. In section 4 we summarize the lessons learned and our plans for future research.

2 The Non-Restoring Square Root Algorithm

An integer square root calculates $y = \sqrt{x}$ where x is the radicand, y is the root, and both x and y are integers. We define the *precise* square root (p) to be the real valued square root and the *correct* integer square root to be the floor of the precise root. We can write the specification for the integer square root as shown in Definition 1.

Definition 1 *Correct integer square root*
y is the *correct* integer square root of $x \hat{=} y^2 \leq x < (y+1)^2$

We have implemented a subtractive, non-restoring integer square root algorithm [BV85]. For radicands in the range $x = \{0..2^{2n} - 1\}$, subtractive methods begin with an initial guess of $y = 2^{(n-1)}$ and then iterate from $i = (n-1)\ldots 0$. In each iteration we square the partial root (y), subtract the squared partial root from the radicand and revise the partial root based on the sign of the result. There are two major classes of algorithms: restoring and non-restoring [Flo63]. In restoring algorithms, we begin with a partial root for $y = 0$ and at the end of each iteration, y is never greater than the precise root (p). Within each iteration (i), we set the i^{th} bit of y, and test if $x - y^2$ is negative; if it is, then setting the i^{th} bit made y too big, so we reset the i^{th} bit and proceed to the next iteration.

Non-restoring algorithms modify each bit position once rather than twice. Instead of setting the the i^{th} bit of y, testing if $x - y^2$ is positive, and then possibly resetting the bit; the non-restoring algorithms add or subtract a 1 in the i^{th} bit of y based on the sign of $x - y^2$ in the previous iteration. For binary arithmetic, the restoring algorithm is efficient to implement. However, most square root hardware implementations use a higher radix, non-restoring implementation. For higher radix implementations, non-restoring algorithms result in more efficient hardware implementations.

The results of the non-restoring algorithms do not satisfy our definition of *correct*, while restoring algorithms do satisfy our definition. The resulting value of y in the non-restoring algorithms may have an error in the last bit position. For the algorithm used here, we can show that the final value of y will always be either the precise root (for radicands which are perfect squares) or will be odd and be within one of the correct root. The error in non-restoring algorithms is easily be corrected in a cleanup phase following the algorithm.

Below we show how a binary, non-restoring algorithm runs on some values for n =3. Note that the result is either exact or odd.

$x = 18_{10} = 10100_2$	iterate 1	y = 100	$x - y^2 = +$
	iterate 2	y = 110	$x - y^2 = -$
	iterate 3	y = 101	
$x = 15_{10} = 01111_2$	iterate 1	y = 100	$x - y^2 = -$
	iterate 2	y = 010	$x - y^2 = +$
	iterate 3	y = 011	
$x = 16_{10} = 10000_2$	iterate 1	y = 100	$x - y^2 = 0$

In our description of the non-restoring square root algorithm, we define a datatype **state** that contains the state variables. The algorithm works by initializing the state using the function **init**, then updating the state on each iteration of the loop by calling the function **update**. We refine our program through several sets of transformations. At each level the definition of **state**, **init** and **update** may change. The SML code that calls the square root at each level is:

```
fun sqrt n radicand =  iterate update (n-1) (init n radicand)
```

The **iterate** function performs iteration by applying the function argument to the state argument, decrementing the count, and repeating until the count is zero.

```
fun iterate f n state =
    if n = 0 then state else iterate f (n - 1) (f state)
```

The top level ($\mathcal{L}_0$) is a straightforward implementation of the non-restoring square root algorithm. We represent the state of an iteration by the triple State{x,y,i} where **x** is the radicand, **y** is the partial root, and **i** is the iteration number. Our initial guess is $\mathtt{y} = 2^{(\mathtt{n}-1)}$. In **update**, **x** never changes and **i** is decremented from n-2 to 1, since the initial values take care of the $(\mathtt{n}-1)^{\mathtt{st}}$ iteration. At $\mathcal{L}_0$, **init** and **update** are:

```
fun init (n,radicand) =
  State{x = radicand,  y = 2 ** (n-1),  i = n-2 }

fun update (State{x, y, i}) =
     let
       val diffx  = x - (y**2)
       val y'     = if      diffx = 0   then y
                    else if diffx > 0   then (y + (2**i))
                    else (* diffx < 0 *)     (y - (2**i))
     in
      State{x = x,  y = y',  i = i-1 }
     end
```

In the next section we discuss the proof that this algorithm calculates the square root. Then we show how it can be refined to an implementation that requires significantly less hardware. We show how to prove that the refined algorithm also calculates the square root; in the absence of such a proof it is not at all obvious that the algorithms have identical results.

2.1 Verification of Level Zero Algorithm

All of the theorems in this section were verified using the Nuprl proof development system. Theorem 1 is the overall correctness theorem for the non-restoring square root code shown above. It states that after iterating through `update` for `n-1` times, the value of `y` is within one of the *correct* root of the radicand. We have proved this theorem by creating an invariant property and performing induction on the number of iterations of `update`. Remember that `n` is the number of bits in the result.

Theorem 1 *Correctness theorem for non-restoring square root algorithm*

$$
\begin{array}{l}
\vdash \forall n \,.\, \forall radicand : \{0..2^{2n}-1\}\,. \\
\quad \mathit{let} \\
\qquad \mathtt{State}\{x, y, i\} = \mathtt{iterate\ update}\ (n-1)\ (\mathtt{init}\ n\ radicand) \\
\quad \mathit{in} \\
\qquad (y-1)^2 \leq\ radicand < (y+1)^2 \\
\quad \mathit{end}
\end{array}
$$

The invariant states that in each iteration `y` increases in precision by $\mathtt{2}^{\mathtt{i}}$, and the lower bits in `y` are always zero. The formal statement ($\mathcal{I}$) of the invariant is shown in Definition 2.

Definition 2 *Loop invariant*

$$
\begin{array}{l}
\mathcal{I}(\mathtt{State}\{x, y, i\}) \mathrel{\hat{=}} \\
\quad ((y-2^i)^2 \leq\ radicand < (y+2^i)^2)\ \&\ (y \text{ rem } 2^i = 0)\ \&\ (x = radicand)
\end{array}
$$

In Theorems 2 and 3 we show that `init` and `update` are correct, in that for all legal values of `n` and `radicand`, `init` returns a legal state and for all legal input states, `update` will return a legal state and makes progress toward termination by decrementing `i` by 1. A legal state is one for which the loop invariant holds.

Theorem 2 *Correctness of initialization*

$$
\begin{array}{l}
\vdash \forall n \,.\, \forall radicand : \{0..2^{2n}-1\}.\forall x, y, i\,. \\
\quad \mathtt{State}\{x, y, i\} = \mathtt{init}\ n\ radicand \implies \\
\qquad \mathcal{I}(\mathtt{State}\{x, y, i\})\ \&\ (x = radicand)\ \&\ (y = 2^{n-1})\ \&\ (i = n-2)
\end{array}
$$

Theorem 3 *Correctness of update*

$$\vdash \forall x, y, i \,.\, \mathcal{I}((\texttt{State}\{x, y, i\})) \implies$$
$$\quad \forall x', y', i' \,.\, \texttt{State}\{x', y', i'\} = \texttt{update}\ (\texttt{State}\{x, y, i\}) \implies$$
$$\qquad \mathcal{I}(\texttt{State}\{x', y', i'\})\ \&\ (i' = i - 1)$$

The correctness of `init` is straightforward. The proof of Theorem 3 relies on Theorem 4, which describes the behavior of the update function. The body of `update` has three branches, so the proof of correctness of update has three parts, depending on whether $\texttt{x} - \texttt{y}^2$ is equal to zero, positive, or negative. Each case in Theorem 4 is straightforward to prove using ordinary arithmetic.

Theorem 4 *Update lemmas*

Case $x - y^2 = 0$
$$\vdash \forall x, y, i \,.\, \mathcal{I}((\texttt{State}\{x, y, i\})) \implies$$
$$\quad (x - y^2 = 0) \implies \mathcal{I}(\texttt{State}\{x, y, i - 1\})$$

Case $x - y^2 > 0$
$$\vdash \forall x, y, i \,.\, \mathcal{I}((\texttt{State}\{x, y, i\})) \implies$$
$$\quad (x - y^2 > 0) \implies \mathcal{I}(\texttt{State}\{x, y + 2^i, i - 1\})$$

Case $x + y^2 > 0$
$$\vdash \forall x, y, i \,.\, \mathcal{I}((\texttt{State}\{x, y, i\})) \implies$$
$$\quad (x + y^2 > 0) \implies \mathcal{I}(\texttt{State}\{x, y - 2^i, i - 1\})$$

We now prove that iterating `update` a total of `n-1` times will produce the correct final result. The proof is done by induction on n and makes use of Theorem 5 to describe one call to `iterate`. This allows us to prove that after iterating `update` a total of `n-1` times, our invariant holds and `i` is zero. This is sufficient to prove that square root is within one of the *correct* root.

Theorem 5 *Iterating a function*

$$\vdash \forall prop, n, f, s.$$
$$\quad prop\ n\ s \implies$$
$$\qquad (\forall n', s'. prop\ n'\ s' \implies prop\ (n' - 1)\ (f\ s')) \implies$$
$$\qquad\quad prop\ 0\ (\texttt{iterate}\ f\ n\ s)$$

2.2 Description of Level One Algorithm

The $\mathcal{L}_0$ SML code would be very expensive to directly implement in hardware. If the state were stored in three registers, `x` would be stored but would never change; the variable `i` would need to be decremented every loop and we would need to calculate $\texttt{y}^2$, $\texttt{x} - \texttt{y}^2$, $2^{\texttt{i}}$, and $\texttt{y} \pm 2^{\texttt{i}}$ in every iteration. All of these are expensive operations to implement in hardware. By restructuring the algorithm through a series of transformations, we preserve the correctness of our design and generate an implementation that uses very little hardware.

The key operations in each iteration are to compute $\mathtt{x} - \mathtt{y}^2$ and then update $\mathtt{y}$ using the new value $\mathtt{y}' = \mathtt{y} \pm\ 2^{\mathtt{i}}$, where $\pm$ is $+$ if $\mathtt{x} - \mathtt{y}^2 \geq \mathtt{0}$ and $-$ if $\mathtt{x} - \mathtt{y}^2 < \mathtt{0}$. The variable $\mathtt{x}$ is only used in the computation of $\mathtt{x} - \mathtt{y}^2$. In the $\mathcal{L}_1$ code we introduce the variable `diffx`, which stores the result of computing $\mathtt{x} - \mathtt{y}^2$. This has the advantage that we can incrementally update `diffx` based on its value in the previous iteration:

$$
\begin{aligned}
\mathtt{y'} &= \mathtt{y} \pm\ 2^{\mathtt{i}} \\
\mathtt{y}'^2 &= \mathtt{y}^2 \pm\ 2 * \mathtt{y} * 2^{\mathtt{i}} + (2^{\mathtt{i}})^2 \\
&= \mathtt{y}^2 \pm\ \mathtt{y} * 2^{\mathtt{i+1}} + 2^{2*\mathtt{i}}
\end{aligned}
\qquad
\begin{aligned}
\mathtt{diffx} &= \mathtt{x} - \mathtt{y}^2 \\
\mathtt{diffx'} &= \mathtt{x} - \mathtt{y}'^2 \\
&= \mathtt{x} - (\mathtt{y}^2 \pm\ \mathtt{y} * 2^{\mathtt{i+1}} + 2^{2*\mathtt{i}}) \\
&= (\mathtt{x} - \mathtt{y}^2) \mp\ \mathtt{y} * 2^{\mathtt{i+1}} - 2^{2*\mathtt{i}}
\end{aligned}
$$

The variable $\mathtt{i}$ is only used in the computations of $2^{2*\mathtt{i}}$ and $\mathtt{y} * 2^{\mathtt{i+1}}$, so we create a variable `b` that stores the value $2^{2*\mathtt{i}}$ and a variable `yshift` that stores $\mathtt{y} * 2^{\mathtt{i+1}}$. We update `b` as: `b' = b div 4`. This results in the following equations to update `yshift` and `diffx`:

$$
\begin{aligned}
\mathtt{y}' * 2^{\mathtt{i+1}} &= (\mathtt{y} \pm\ 2^{\mathtt{i}}) * 2^{\mathtt{i+1}} \\
\mathtt{yshift}' &= \mathtt{y} * 2^{\mathtt{i+1}} \pm\ 2^{\mathtt{i}} * 2^{\mathtt{i+1}} \\
&= \mathtt{yshift} \pm\ 2 * 2^{2*\mathtt{i}} \\
&= \mathtt{yshift} \pm\ 2 * \mathtt{b}
\end{aligned}
\qquad
\mathtt{diffx}' = \mathtt{diffx} \mp\ \mathtt{yshift} - \mathtt{b}
$$

The transformations from $\mathcal{L}_0$ to $\mathcal{L}_1$ can be summarized:

$$\mathtt{diffx} = \mathtt{x} - \mathtt{y}^2 \qquad \mathtt{yshift} = \mathtt{y} * 2^{\mathtt{i+1}} \qquad \mathtt{b} = 2^{2*\mathtt{i}}$$

The $\mathcal{L}_1$ versions of `init` and `update` are given below. Note that, although the optimizations are motivated by the fact that we are doing bit vector arithmetic, the algorithm is correct for unbounded integers. Also note that the most complex operations in the update loop are an addition and subtraction and only one of these two operations is executed each iteration. We have optimized away all exponentiation and any multiplication that cannot be implemented as a constant shift.

```
fun init1 (n,radicand) =
  let
    val b' = 2 ** (2*(n-1))
  in
   State{diffx  = radicand - b',
         yshift = b',
         b      = b' div 4
        }
  end
```

```
fun update1 (State{diffx, yshift, b}) =
  let
    val (diffx',yshift')  =
       if      diffx > 0 then (diffx - yshift - b, yshift + 2*b)
       else if diffx = 0 then (diffx             , yshift      )
       else (* diffx < 0 *)   (diffx + yshift - b, yshift - 2*b)
  in
   State{diffx  = diffx',
         yshift = yshift' div 2,
         b      = b  div 4
        }
  end
```

We could verify the $\mathcal{L}_1$ algorithm from scratch, but since it is a transformation of the $\mathcal{L}_0$ algorithm, we use the results from the earlier verification. We do this by defining a mapping function between the state variables in the two levels and then proving that the two levels return equal values for equal input states. The transformation is expressed as follows:

Definition 3 *State transformation*

$$\mathcal{T}(\texttt{State}\{x, y, i\}, \texttt{State}\{\mathit{diffx}, \mathit{yshift}, b\}) \hat{=}$$
$$(\mathit{diffx} = x - y^2 \ \& \ \mathit{yshift} = y * 2^{i+1} \ \& \ b = 2^{2*i})$$

All of the theorems in this section were proved with Nuprl. In Theorem 6 we say that for equal inputs `init1` returns an equivalent state to `init`. In Theorem 7 we say that for equivalent input states, `update1` and `update` return equivalent outputs.

Theorem 6 *Correctness of initialization*

$$\begin{aligned} &\vdash \forall n \,.\, \forall \mathit{radicand} \ : \ \{0..2^{2n} - 1\}. \\ &\quad \forall x, y, i \,.\, \texttt{State}\{x, y, i\} = \texttt{init}\ n\ \mathit{radicand} \implies \\ &\quad\quad \forall \mathit{diffx}, \mathit{yshift}, b \,.\, \texttt{State}\{\mathit{diffx}, \mathit{yshift}, b\} = \texttt{init1}\ n\ \mathit{radicand} \implies \\ &\quad\quad\quad \mathcal{T}(\texttt{State}\{x, y, i\}, \texttt{State}\{\mathit{diffx}, \mathit{yshift}, b\}) \end{aligned}$$

Theorem 7 *Correctness of update*

$$\begin{aligned} &\vdash \forall x, y, i, \mathit{diffx}, \mathit{yshift}, b \,. \\ &\quad \mathcal{T}(\texttt{State}\{x, y, i\}, \texttt{State}\{\mathit{diffx}, \mathit{yshift}, b\}) \implies \\ &\quad\quad \mathcal{T}(\texttt{update}(\texttt{State}\{x, y, i\}), \texttt{update1}(\texttt{State}\{\mathit{diffx}, \mathit{yshift}, b\})) \end{aligned}$$

Again, the initialization theorem has an easy proof, and the `update1` theorem is a case split on each of the three cases in the body of the update function, followed by ordinary arithmetic.

2.3 Description of Level Two Algorithm

To go from $\mathcal{L}_1$ to $\mathcal{L}_2$, we recognize that the operations in `init1` are very similar to those in `update1`. By carefully choosing our initial values for `diffx` and `y`, we increase the number of iterations from `n-1` to `n` and fold the computation of `radicand - b'` in `init` into the first iteration of `update`. This eliminates the need for special initialization hardware. The new initialize function is:

```
fun init2 (n,radicand) =
  State{diffx  = radicand,
        yshift = 0,
        b      = 2 ** (2*(n-1))}
```

The update function is unchanged from `update1`. The new calling function is:

```
fun sqrt n radicand =  iterate update1 n (init2 n radicand)
```

Showing the equivalence between `init2` and a loop that iterates `n` times and the $\mathcal{L}_1$ functions requires showing that the state in $\mathcal{L}_2$ has the same value after the first iteration that it did after `init1`. More formally, `init1 = update1` $\circ$ `init2`. We prove this using the observation that, after `init2`, `diffx` is guaranteed to be positive, so the first iteration of `update1` in $\mathcal{L}_2$ always executes the `diffx > 0` case. Using the state returned by `init2` and performing some simple algebraic manipulations, we see that after the first iteration in $\mathcal{L}_2$, `update1` stores the same values in the state variables as `init1` did. Because both $\mathcal{L}_1$ and $\mathcal{L}_2$ use the same update function, all subsequent iterations in the $\mathcal{L}_2$ algorithm are identical to the $\mathcal{L}_1$ algorithm.

To begin the calculation of a square root, a hardware implementation of the $\mathcal{L}_2$ algorithm clears the `yshift` register, load the radicand into the `diffx` register, and initializes the `b` register with a 1 in the correct location. The transformation from $\mathcal{L}_0$ to $\mathcal{L}_1$ simplified the operations done in the loop to shifts and an addition/subtraction. Remaining transformations will further optimize the hardware.

3 Transforming Behavior to Structure with HML

The goal of this section is to produce an efficient hardware implementation of the $\mathcal{L}_2$ algorithm. The first subsection introduces Hardware ML, our language for specifying the behavior and structure of hardware. Taking an HML version of the $\mathcal{L}_2$ algorithm as our starting point, we obtain a hardware implementation through a sequence of transformation steps.

1. Translate the $\mathcal{L}_2$ algorithm into Hardware ML. Provision must be made to initialize and detect termination, which is not required at the algorithm level.
2. Transform the HML version of $\mathcal{L}_2$ to a set of register assignments using syntactic transformations.

3. Introduce an internal state register `Exact` to simplify the computation, and "factor out" the condition `DiffX >= %0`.
4. Partition into functional blocks, again using syntactic transformations.
5. Substitute lower level modules for register and combinational assignments. Further optimizations in the implementation of the lower level modules are possible.

Each step can be verified formally. Several of these must be justified by properties of the algorithm that we can establish through theorem proving.

3.1 Hardware ML

We have implemented extensions to Standard ML that can be used to describe the behavior of digital hardware at the register transfer level. Earlier work has illustrated how Hardware ML can be used to describe the structure of hardware [OLLA93, OLLA92]. HML is based on SML and supports higher-order, polymorphic functions, allowing the concise description of regular structures such as arrays and trees. SML's powerful module system aids in creating parameterized designs and component libraries.

Hardware is modelled as a set of concurrently executing *behaviors* communicating through objects called *signals*. Signals have semantics appropriate for hardware modelling: whereas a Standard ML reference variable simply contains a value, a Hardware ML signal contains a list of time-value pairs representing a waveform. The current value on signal `a`, written `$a`, is computed from its waveform and the current time.

Two kinds of signal assignment operators are supported. *Combinational assignment*, written s `==` v, is intended to model the behavior of combinational logic under the assumption that gate delays are negligible. s `==` v causes the current value of the target signal s to become v. For example, we could model an exclusive-or gate as a behavior which assigns `true` to its output `c` whenever the current values on its inputs `a` and `b` are not equal:

```
fun Xor (a,b,c) = behavior (fn () => (c == ($a <> $b)))
```

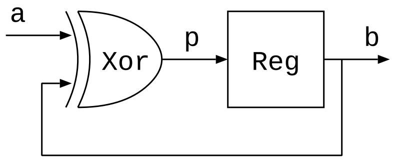
HML's `behavior` constructor creates objects of type `behavior`. Its argument is a function of type `unit -> unit` containing HML code – in this case, a combinational assignment.

Register assignment is intended to model the behavior of sequential circuit elements. If a register assignment s `<-` v is executed at time t the waveform of s is augmented with the pair $(v, t+1)$ indicating that s is to assume the value v at the next time step. For example, we could model a delay element as a behavior containing a register assignment:

```
fun Reg (a,b) = behavior (fn () => (b <- $a))
```

Behaviors can be composed to form more complex circuits. In the composition of b_1 and b_2, written b_1 `||` b_2, both behaviors execute concurrently. We compose our exclusive-or gate and register to build a parity circuit, which outputs `false` if and only if it has received an even number of `true` inputs:

```
fun Parity (a,b) =
 let
   val p = signal false
 in
   Xor (a,b,p) || Reg (p,b)
 end
```



The `val p =` ... declaration introduces an internal signal whose initial value is `false`.

Table 1 summarizes the behavioral constructs of HML. We have implemented a simple simulator which allows HML behavioral descriptions to be executed interactively – indeed, all the descriptions in this section were developed with the aid of the simulator.

$\alpha^=$ `signal`	Type of signals having values of equality type $\alpha^=$
`signal` v	A signal initially having the value v
`$` s	The current value of signal s
s `==` v	Combinational assignment
s `<-` v	Register assignment
`behavior`	Type of behaviors
`behavior` f	Behavior constructor
b_1 `||` b_2	Behavioral composition

Table 1. Hardware ML Constructs

3.2 Behavioral Description of the Level 2 Algorithm

It is straightforward to translate the $\mathcal{L}_2$ algorithm from Standard ML to Hardware ML. The state of the computation is maintained in the externally visible signal `YShift` and the internal signals `DiffX` and `B`. These signals correspond to `yshift`, `diffx` and `b` in the $\mathcal{L}_2$ algorithm and are capitalized to distinguish them as HML signals. For concreteness, we stipulate that we are computing an eight-bit root (`n` $= 8$ in the $\mathcal{L}_2$ algorithm), and so the `YShift` and `B` signals require 16 bits. `DiffX` requires 17 bits: 16 bits to hold the initial (positive) radicand and one bit to hold the sign, which is important in the intermediate calculations. We use HML's register assignment operator `<-` to explicitly update the state in each computation step. The `%` characters preceding numeric constants are constructors for an abstract data type `num`. The usual arithmetic operators are defined on this abstract type. We are thus able to simulate our description using unbounded integers, bounded integers, and bit vectors simply by providing appropriate implementations of the type and operators. The code below shows the HML representation of the $\mathcal{L}_2$ algorithm. Two internal signals are defined, and the behavior is packaged within a function declaration. In the future we will omit showing the function declaration and declarations of internal signals; it is understood that all signals except **Init**, **XIn**, **YShift**, and **Done** are internal.

```
val initb = %16384              (* 0x4000 *)

fun sqrt (Init, XIn, YShift, Done) =
    let
        (* Internal registers *)
        val DiffX = signal (%0)
        val B = signal (%0)
    in
        behavior
        (fn () =>
         (if $Init then
              (DiffX <- $XIn;
               YShift <- %0;
               B <- initb;
               Done <- false)
          else
              (if $DiffX = %0 then
                   (DiffX <- $DiffX;
                    YShift <- $YShift div %2;
                    B <- $B div %4)
               else if $DiffX > %0 then
                   (DiffX <- $DiffX - $YShift - $B;
                    YShift <- ($YShift + %2 * $B) div %2;
                    B <- $B div %4)
               else (* $DiffX < %0 *)
                   (DiffX <- $DiffX + $YShift - $B;
                    YShift <- ($YShift - %2 * $B) div %2;
                    B <- $B div %4));
               Done <- $B sub 0))
    end
```

In the SML description of the algorithm, some elements of the control state are implicit. In particular, the initiation of the algorithm (calling **sqrt**) and its termination (it returning a value) are handled by the SML interpreter. Because hardware is free running there is no built-in notion of initiation or termination of the algorithm. It is therefore necessary to make explicit provision to initialize the state registers at the beginning of the algorithm and to detect when the algorithm has terminated.

Initialization is easy: the **diffx**, **yshift**, and **b** registers are assigned their initial values when the **init** signal is high.

In computing an eight-bit root, the $\mathcal{L}_2$ algorithm terminates after seven iterations of its loop. An efficient way to detect termination of the hardware algorithm makes use of some knowledge of the high level algorithm. An informal analysis of the $\mathcal{L}_2$ algorithm reveals that **b** contains a single bit, shifted right two places in each cycle, and that the least significant bit of **b** is set during the execution of the last iteration. Consequently, the `done` signal is generated by testing whether the least significant bit of `b` is set (the expression `$b sub 0` selects the lsb of `b`) and delaying the result of the test by one cycle. `done` is therefore set during the clock cycle following the final iteration. To justify this analysis, we must formally prove that the following is an invariant of the $\mathcal{L}_2$ algorithm:

$$(\mathtt{i} = 1) \longleftrightarrow (\mathtt{b}_{\mathtt{0}} = 1)$$

3.3 Partitioning into Register Assignments

Our second transformation step is a simple one: we transform the HML version of the $\mathcal{L}_2$ algorithm into a set of register assignments. The goal of this transformation is to ensure that the control state of the algorithm is made completely explicit in terms of HML signals.

We make use of theorems about the semantics of HML to justify transformations of the `behavior` construct. First, `if` distributes over the sequential composition of signal assignment statements: if P and Q are sequences of signal assignments, then

```
if e then
   (s <- a; P)
else
   (s <- b; Q)
```

=

```
if e then
   s <- a
else
   s <- b;
if e then P else Q
```

We also use a rule which allows us to push `if` into the right hand side of an assignment:

```
if e then
   s <- a
else
   s <- b
```

=

```
s <- if e then a else b
```

Repeatedly applying these two rules allows us to decompose our HML code into a set of assignments to individual registers. The register assignments after this transformation are

```
DiffX  <- (if $Init then
                $XIn
            else if $DiffX = %0 then
                $DiffX
            else if $DiffX > %0 then
                $DiffX - $YShift - $B
            else (* $DiffX < %0 *)
                $DiffX + $YShift - $B);
```

```
YShift <- (if $Init then
              %0
           else if $DiffX = %0 then
              $YShift div %2
           else if $DiffX > %0 then
              ($YShift + %2 * $B) div %2
           else (* $DiffX < %0 *)
              ($YShift - %2 * $B) div %2);

B      <- (if $Init then initb else $B div %4);
Done   <- (if $Init then false else $B sub 0);
```

3.4 Simplifying the Computation

Our third transformation step simplifies the computation of `DiffX` and `YShift`. We begin by observing two facts about the $\mathcal{L}_2$ algorithm: if `DiffX` ever becomes zero the radicand has an exact root, and once `DiffX` becomes zero it remains zero for the rest of the computation. To simplify the computation of `DiffX` we introduce an internal signal `Exact` which is set if `DiffX` becomes zero in the course of the computation.

If `Exact` becomes set, the value of `DiffX` is not used in the computation of `YShift` (subsequent updates of `YShift` involve only division by 2). `DiffX` becomes a don't care, and we can merge the `$DiffX = %0` and `$DiffX > %0` branches. We also replace `$DiffX = %0` with `$Exact` in the assignment to `YShift`, and change the `$DiffX > %0` comparisons to `$DiffX >= %0` (note that this branch of the if is not executed when `$DiffX = %0`, because `Exact` is set in this case).

```
ExactReg <- (if $init then false else $Exact);
Exact    == ($DiffX = %0 orelse $ExactReg);

DiffX    <- (if $Init then
                $XIn
             else if $DiffX >= %0 then
                $DiffX - $YShift - $B
             else (* $DiffX < %0 *)
                $DiffX + $YShift - $B);

YShift   <- (if $Init then
                %0
             else if $Exact then
                $YShift div %2
             else if $DiffX >= %0 then
                ($YShift + %2 * $B) div %2
             else (* $DiffX < %0 *)
                ($YShift - %2 * $B) div %2);

B        <- (if $Init then initb else $B div %4);
Done     <- (if $Init then false else $B sub 0);
```

Simplifying the algorithm in this way requires proving a history property of the computation of `DiffX`. Using $\texttt{\$DiffX}(n)$ to denote the value of the signal `DiffX` in the n'th computation cycle, we state the property as:

$$\forall t : t \geq 0.(\texttt{\$DiffX}(t) = 0) \implies (\texttt{\$DiffX}(t+1) = 0)$$

Next, we note that the condition `$DiffX >= 0` can be detected by negating `DiffX`'s sign bit. We introduce the negated sign bit as the intermediate signal `ADD` (to specify that we are adding to `YShift` in the current iteration) and rewrite the conditions in the assignments to `DiffX` and `Yshift` to obtain:

```
ADD      == not ($DiffX sub 16);

ExactReg <- (if $init then false else $Exact);
Exact    == ($DiffX = %0 orelse $ExactReg);

DiffX    <- (if $Init then
                 $XIn
             else if $ADD then
                 $DiffX - $YShift - $B
             else $DiffX + $YShift - $B);

YShift   <- (if $Init then
                 %0
             else if $Exact then
                 $YShift div %2
             else if $ADD then
                 ($YShift + %2 * $B) div %2
             else ($YShift - %2 * $B) div %2);

B        <- (if $Init then initb else $B div %4);

Done     <- (if $Init then false else $B sub 0);
```

3.5 Partitioning into Functional Blocks

The fourth transformation separates those computations which can be performed combinationally from those which require sequential elements, and partitions the computations into simple functional units. The transformation is motivated by our desire to implement the algorithm by an interconnection of lower level blocks; the transformation process is guided by what primitives we have available in our library. For example, our library contains such primitives as registers, multiplexers, and adders, so it is sensible to transform

`s <- if x then a else b+c`	$\longrightarrow$	`s <- s';`
		`s' == if x then a else d;`
		`d == b + c`

This particular example is a consequence of some more general rules which are justified as before by appealing to the semantics of HML.

$$\boxed{s \texttt{ == if } e \texttt{ then } a \texttt{ else } b} \quad = \quad \boxed{\begin{array}{l} s \texttt{ == if } e \texttt{ then } s_1 \texttt{ else } s_2\texttt{;} \\ s_1 \texttt{ == } a\texttt{;} \\ s_2 \texttt{ == } b \end{array}}$$

The assignments resulting from this transformation are shown below. `DiffX'` can be computed by a multiplexer. `DiffXTmp` and `Delta` can be computed by adder/subtracters. The `YShift` register can be conveniently implemented as a shift register which shifts when `Exact`'s value is true, and loads otherwise. `YShift'` can be computed by a multiplexer; `YShiftTmp` by an adder/subtracter – multiplication and division by 2 are simply wired shifts.

```
ADD       == not ($DiffX sub 16);

ExactReg  <- (if $init then false else $Exact);
Exact     == ($EqZero orelse $ExactReg);
EqZero    == $DiffX = %0;

DiffX     <- $DiffX';
DiffX'    == (if $Init then $XIn else $DiffXTmp);
DiffXTmp  == (if $ADD then $DiffX - $Delta else $DiffX + $Delta);
Delta     == (if $ADD then $YShift + $B else $YShift - $B);

YShift    <- (if $Exact then $YShift div %2 else $YShift');
YShift'   == (if $Init then %0 else $YShiftTmp);
YShiftTmp == (if $ADD then
                  ($YShift + %2 * $B) div %2
              else
                  ($YShift - %2 * $B) div %2);

B         <- (if $Init then initb else $B div %4);

Done      <- (if $Init then false else $B sub 0);
```

3.6 Structural Description of the Level 2 Algorithm

The fifth, and final, transformation step is the substitution of lower-level modules for the register and combinational assignments; the result is a structural description of the integer square root algorithm which can readily be implemented in hardware, as shown in Figure 1.

`ShiftReg4` is a shift register which shifts its contents two bits per clock cycle; the `Done` signal is simply its shift output. `ShiftReg2`, `Mux`, `AddSub` and `Reg` are a shift register, multiplexer, adder/subtracter and register, respectively. `SubAdd` is an adder/subtracter, but the sense of its mode bit makes it the opposite of `AddSub`. The `Hold` element has the following substructure:

```
Hold      {exact=Exact, zero=EqZero, rst=Init}                   ||
IsZero    {out=EqZero, inp=DiffX}                                ||
Reg       {out=DiffX, inp=DiffX'}                                ||
Mux       {out=DiffX', in0=XIn, in1=DiffXTmp, ctl=Init}          ||
AddSub    {sum=DiffXTmp, in0=DiffX, in1=Delta, sub=ADD}          ||
Neg       {out=ADD, inp=$DiffX sub 16}                           ||
SubAdd    {out=Delta, in0=YShift, in1=B, add=ADD}                ||
ShiftReg2 {out=YShift, inp=YShift', shift=Exact}                 ||
Mux       {out=YShift', in0=signal (%0),
           in1=YShiftTmp div 2, ctl=Init}                        ||
SubAdd    {out=YShiftTmp, in0=YShift, in1=2 * B, sub=ADD}        ||
ShiftReg4 {out=B, shiftout=Done, in=signal initb, shiftbar=Init}
```

Fig. 1. Structural Description

```
fun Hold {exact=exact, zero=zero, rst=rst} =
    let
        val ExactReg = signal false
    in
        RegR {out=ExactReg, inp=exact, rst=rst}    ||
        Or2  {out=Exact, in0=zero, in1=ExactReg}
    end
```

There are further opportunities for performing optimization in the implementation of the lower level blocks. Analysis of the $\mathcal{L}_2$ algorithm reveals that `Delta` is always positive, so we do not need its sign bit. This property can be used to save one bit in the `AddSub` used to compute `Delta` – only 16 bits are now required. One bit can also be saved in the implementation of `SubAdd`; the value of the `ADD` signal can be shown to be identical to a latched version of the carry output of a 16-bit `SubAdd`, provided the latch initially holds the value `true`. Figure 2 shows a block diagram of the hardware to implement the square root algorithm, which includes this optimization.

A number of other optimizations are not visible in the figure. The `ShiftReg4` can be implemented by a `ShiftReg2` half its width if we note that every second bit of `B` is always zero. The two `SubAdd` blocks can each be implemented with only a few AND and OR gates per bit, rather than requiring subtract/add modules, if we make use of some results concerning the contents of the `B` and `YShift` registers.

To justify these optimizations we are obliged to prove that every second bit of `B` is always zero:

$$\forall i : 0 \leq i < n : \neg \mathtt{B}_{2i+1}$$

and that the corresponding bits of `B` and `YShift` cannot both be set:

$$\forall i : 0 \leq i < 2n . \neg(\mathtt{B}_i \wedge \mathtt{YShift}_i)$$

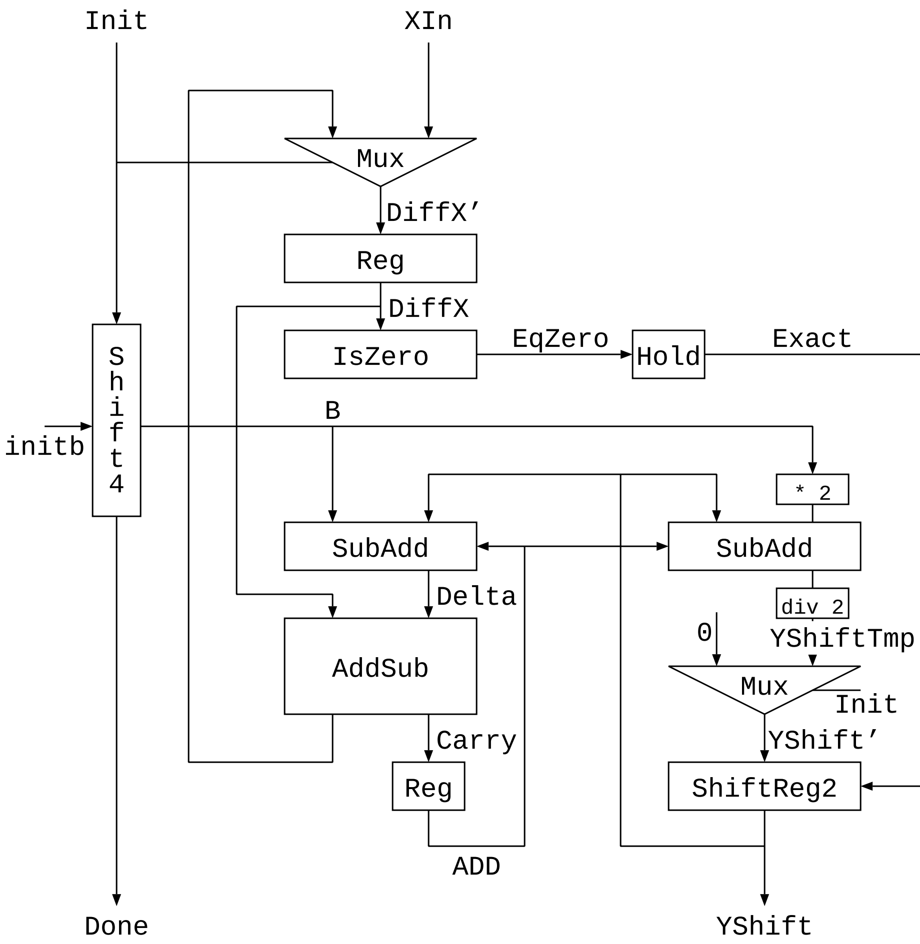


Fig. 2. Square Root Hardware

and that the corresponding bits of `$B * %2` and `YShift` cannot both be set:

$$\forall i : 1 \leq i < 2n . \neg(\mathtt{B}_{i-1} \wedge \mathtt{YShift}_i)$$

We have produced two implementations of the square root algorithm which incorporate all these optimizations. The first was a full-custom CMOS layout fabricated by Mosis, the second used Altera programmable logic parts. In the latter case, the structural description was first translated to Altera's AHDL language and then passed through Altera's synthesis software.

4 Discussion

We have described how to design and verify a subtractive, non-restoring integer square root circuit by refining an abstract algorithmic specification through several intermediate levels to yield a highly optimized hardware implementation. We have proved using Nuprl that the $\mathcal{L}_0$ algorithm performs the square root function, and we have also used Nuprl to show how proving the first few levels of refinement ($\mathcal{L}_0$ to $\mathcal{L}_1$, $\mathcal{L}_1$ to $\mathcal{L}_2$) can be accomplished by transforming the top level proof in a way that preserves its validity.

This case study illustrates that rigorous reasoning about the high-level description of an algorithm can establish properties which are useful even for bit-level optimization. Theorem provers provide a means of formally proving the desired properties; a transformational approach to partitioning and optimization ensures that the properties remain relevant at the structural level. Each of the steps identified in this paper can be mechanized with reasonable effort. At the bottom level, we have a library of verified hardware modules that correspond to the modules in the HML structural description [AL94].

In many cases the transformations we applied depend for their justification upon non-trivial properties of the square root algorithm: we are currently working on formally proving these obligations. Some of our other transformations are purely syntactic in nature and rely upon HML's semantics for their justification. We have not considered semantic reasoning in this paper – this is a current research topic.

The algorithm we describe computes the integer square root. The algorithm and its implementation are of general interest because most of the algorithms used in hardware implementations of floating-point square root are based on the algorithm presented here. One difference is that most floating-point implementations use a higher radix representation of operators. In the future, we will investigate incorporating higher radix floating-point operations. We believe much of the reasoning presented here will be applicable to higher radix implementations of square root as well.

Many of the techniques demonstrated in this case study are applicable to hardware verification in general. Proof development systems are especially well suited for reasoning at high levels of abstraction and for relating multiple levels of abstraction. Both of these techniques must be exploited in order to make it feasible to apply formal methods to large scale highly optimized hardware systems. Top level specifications must be concise and intuitively capture the designers' natural notions of correctness (for example, arithmetic operations on unbounded integers), while the low level implementation must be easy to relate to the final implementation (for example, operations on bit-vectors). By applying a transformational style of verification as a design progresses from an abstract algorithm to a concrete implementation, theorem proving based verification can be integrated into existing design practices.

Acknowledgements

This research is supported in part by the National Science Foundation under contracts CCR-9257280 and CCR-9058180 and by donations from Altera Corporation. John O'Leary is supported by a Fellowship from Bell-Northern Research Ltd. Miriam Leeser is supported in part by an NSF Young Investigator Award. Jason Hickey is an employee of Bellcore. Mark Aagaard is supported by a Fellowship from Digital Equipment Corporation. We would like to thank Peter Soderquist for his help in understanding the algorithm and its implementations, Mark Hayden for his work on the proof of the algorithm, Shee Huat Chow and Ser Yen Lee for implementing the VLSI version of this chip, and Michael Bertone and Johanna deGroot for the Altera implementation.

References

[AL94] Mark D. Aagaard and Miriam E. Leeser. A methodology for reusable hardware proofs. *Formal Methods in System Design*, 1994. To appear.

[BJP93] Bhaskar Bose, Steve D. Johnson, and Shyamsundar Pullela. Integrating Boolean verification with formal derivation. In David Agnew, Luc Claesen, and Raul Camposano, editors, *Computer Hardware Description Languages and their Applications*, IFIP Transactions A-32. Elsevier, North-Holland, 1993.

[BV85] J. Bannur and A. Varma. A VLSI implementation of a square root algorithm. In *IEEE Symp. on Comp. Arithmetic*, pages 159–165. IEEE Comp. Soc. Press, Washington D.C., 1985.

[CGM86] Alberto Camilleri, Mike Gordon, and Tom Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs.* Elsevier, September 1986.

[Flo63] Ivan Flores. *The Logic of Computer Arithmetic.* Prentice Hall, Englewood Cliffs, NJ, 1963.

[Lee92] Miriam E. Leeser. Using Nuprl for the verification and synthesis of hardware. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design.* Prentice-Hall International Series on Computer Science, 1992.

[NS88] Paliath Narendran and Jonathan Stillman. Formal verification of the sobel image processing chip. In *DAC*, pages 211–217. IEEE Comp. Soc. Press, Washington D.C., 1988.

[OLLA92] John O'Leary, Mark Linderman, Miriam Leeser, and Mark Aagaard. HML: A hardware description language based on Standard ML. Technical Report EE-CEG-92-7, Cornell School of Electrical Engineering, October 1992.

[OLLA93] John O'Leary, Mark Linderman, Miriam Leeser, and Mark Aagaard. HML: A hardware description language based on SML. In David Agnew, Luc Claesen, and Raul Camposano, editors, *Computer Hardware Description Languages and their Applications*, IFIP Transactions A-32. Elsevier, North-Holland, 1993.

[VCH94] D. Verkest, L. Claesen, and H. De Man A proof of the nonrestoring division algorithm and its implementation on an ALU. *Formal Methods in System Design*, 4(1):5–31, January 1994.