Verifying a Self-Timed Divider

Tarik Ono-Tesfaye   Christoph Kern   Mark R. Greenstreet*
Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4
Canada
{tesfaye,ckern,mrg}@cs.ubc.ca

Abstract

This paper presents an approach to verifying timed designs based on refinement: first, correctness is established for a speed-independent model; then, the timed design is shown to be a refinement of this model. Although this approach is less automatic than methods based on timed state space enumeration, it is tractable for larger designs. Our method is implemented using a proof checker with a built-in model checker for verifying properties of high-level models, a tautology checker for establishing refinement, and a graph-based timing verification procedure for showing timing properties of transistor level models. We demonstrate the method by proving the timing correctness of Williams' self-timed divider.

1 Introduction

Self-timed designs operate without clocks, and timing of events is determined by data encodings and/or matched delays. This provides many opportunities for optimization. For example, data dependent computation times can be exploited. Further optimization are based on the observation that similar circuits will have delays that track over variations in process parameters, power supply voltage, temperature, etc. While timing based optimizations may be essential to achieve competitive performance, they impose heavy demands on design verification techniques. On the one hand, there is an increased demand for verification because it becomes less "obvious" that the design is correct. On the other hand, verification becomes more difficult because optimized designs may not strictly follow a clear design methodology, timing and functionality issues often become intertwined, and the number of cases to be considered can grow dramatically.

For example, in speed-independent designs, correct handshaking between components is an invariant that does not depend on the relative delays of components. For such a design, functionality and performance are separable concerns: functionality can be verified without considering component delays; and performance can be optimized by reducing delays without fear of introducing errors in functionality. However, it is often possibility to increase the performance of a design by exploiting timing relationships between components. With such optimizations, functionality and performance are no longer separate concerns: correct functionality depends on the ordering of events which depends on both the discrete operations of components and their delays.

Many researchers have recognized the need for verification of designs that have critical delay dependencies. Timed automata provide a theoretical framework for modeling such designs, and in principle, many properties of such automata are decidable [11]. Several tools have been implemented based on various timed automata models [15, 2, 4, 26]. Although fully automatic, the high computational complexity of these automata based models have limited their application to models much simpler than typical self-timed designs. Hulgaard et. al. [12] avoid some of this state space explosion by using a task graph where dependencies are independent of data values. This allows them to verify timing properties of abstract models of sizable designs, but they cannot model delays that depend on data values.

Our approach is based on the observation that many timed designs are derived from a design that was originally speed independent. It is natural to view the optimized design as a refinement of the original design.

*This work was supported in part by NSERC research grant OGP-0138501, a UBC graduate fellowship, and a BC Advanced Systems Institute faculty fellowship.

Key properties of the design can be verified starting with the simple, unoptimized design. Often, highly automated methods such as model-checking can be used at this level. Next, the designer provides a mapping between states of the original, speed-independent design and the optimized, timed design. Verifying the optimized design does not require directly computing the entire state space of the optimized design. Instead, single state transitions or short sequences of state transitions are shown to correspond to actions of the abstract model. In the first case, correspondence can often be established using a tautology checker. In the second, decision procedures based on simple timing verification techniques can be employed. We combine these results using a proof checker to establish the correctness of the optimized design.

To demonstrate our approach, we consider the three-stage version of Williams' self-timed divider [28]. This design was chosen because includes several low-level optimizations including timing assumptions and extensive use of pre-charged logic. It is also the basis of several subsequent divider designs including a five-stage design [29] and a design based on static logic for lower power consumption [16].

In the remainder of this paper, the divider design is described in section 2, and section 3 outlines our verification strategy for the divider. Our proofs were developed in a proof checker. This tool ensures that the proofs are complete and provides the logical glue for combining results from various verification tools. The proof checker is described in section 4, and the proofs are presented in section 5.

2 The Divider

The divider that we verified [31] implements the radix-2 SRT algorithm [8]. This algorithm is similar to the binary version of the traditional "paper-and-pencil" algorithm except that quotient bits are chosen from the set $\{-1, 0, 1\}$ instead of the more traditional $\{0, 1\}$. This set of digits is redundant in the sense that the same number can be represented with several different encodings. For example, the decimal integer 5 has five possible four-bit encodings:

```
( 0)(+1)( 0)(+1)   ( 0)(+1)(+1)(-1)
(+1)(-1)( 0)(+1)   (+1)(-1)(+1)(-1)
(+1)( 0)(-1)(-1)
```

This redundancy allows quotient bits to be selected without first computing the exact value of the partial remainder. Instead, only the top few bits of the partial remainder are needed, and there may be pending carries from the lower bits. After the complete quotient has been computed, the traditional, non-redundant representation can be obtained by performing a single subtraction operation. For floating point applications, the mantissa of the divisor can be assumed to be normalized; therefore it lies in the interval $[0.5, 1)$. For the radix-2 algorithm, quotient digits can be selected based on the value of the partial remainder without considering the value of the divisor.

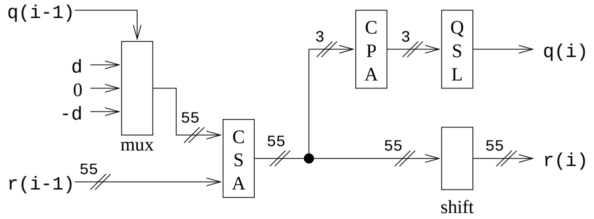

Figure 1. A Divider Stage

The divider chip that we verified operates on 55 bit mantissas. Figure 1 shows the hardware that implements a single step of the division algorithm. The quotient bit from the previous stage, `q(i)`, is used to select whether `d`, 0, or `-d` will be added in a carry-save adder to the previous partial remainder, `r(i-1)` (`d` is the divisor). The output of the carry-save adder is then shifted one bit to the left (the shift is just a relabeling of wires) to produce the new partial remainder, `r(i)`. The sum of the three most-significant bits of the carry-save adder is determined using a carry-propagate adder, CPA. Its output is then used by the quotient selection logic to select the next quotient digit. Because SRT division allows the use of carry-save arithmetic, the time required for a division step is roughly independent of the number of bits in the operands.

The SRT algorithm computes a sequence of partial remainders and quotient bits. This leads naturally to an iterative implementation, and the divider chip we verify uses the self-timed ring [30, 23] shown in figure 2. Data values are encoded using a dual-rail code [20] except for the quotient bits which are encoded using a three-wire, "one-hot" scheme [17]. Precharged logic is

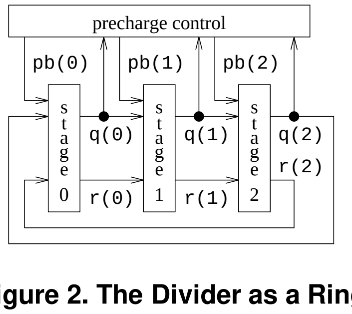

Figure 2. The Divider as a Ring

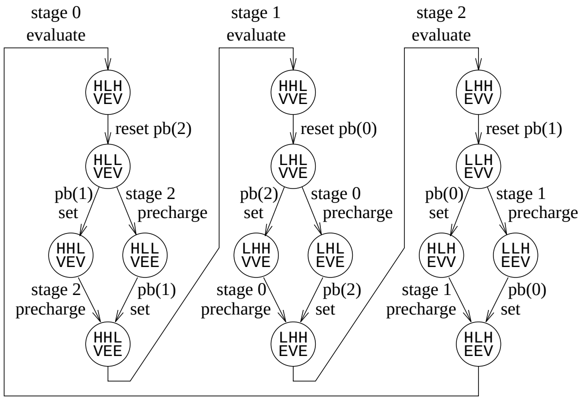

Figure 3. A Speed-Independent Implementation

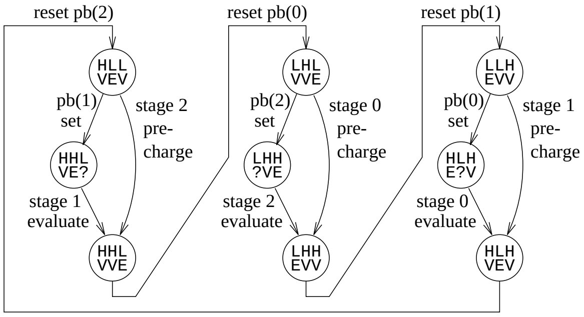

Figure 4. A Timed Implementation

used throughout the design. Each stage can be in one of three states: precharge, evaluate, or hold. Precharging leads to a state where every dual-rail signal produced by the stage has the "empty" value. Evaluation leads to a state where every signal has a "valid" value. A stage in the holding state leaves its outputs unchanged so that its successor change can use them to compute the next partial remainder and quotient bit.

The operation of each stage is controlled by the precharge control block. We first consider the speed-independent variation of the divider that is depicted in figure 3. The "precharge bar" signal for stage `i` is `pb(i)`. When `pb(i)` is low, stage i is precharging. The upper line of each state label gives the value (high or low) of `pb(0)`, `pb(1)`, and `pb(2)`. The lower line gives the state of the outputs of stages 0, 1, and 2 (empty or valid). This design is speed-independent as is apparent from the figure 3. For good measure, we have also verified speed independence using the model checker in our proof checker.

The actual divider is not speed independent; it exploits timing relationships between various components to achieve higher performance. To determine the completion status of the entire output word of a divider stage in a speed-independent manner, a large C-element tree would be required. This would significantly increase the area of the design and would add substantial delay to the critical timing paths. Because the path from the inputs of a stage to the quotient bit output is longer than the path to any of the partial remainder bits, it seems reasonable that the status of the quotient bit output can be used to determine the status of all bits. This observation is verified in section 5. Using the quotient bit to indicate stage completion eliminates the need for a large completion tree.

The second optimization is based on the observation that all logic elements in a stage can be precharged in parallel. Evaluation requires data to flow through a sequence of logic elements and can be expected to take longer than precharging. Like the observation about completion detection, this assumption is verified in section 5. In the optimized design, stage `i` evaluates in parallel with the precharging of stage `i+1`. This effectively removes the time for precharging from the computation. A state transition diagram for the optimized divider is shown in figure 4.

With these optimizations, the divider is no longer speed-independent. To show that the divider continues to operate correctly for all component delays within specified bounds, requires timing analysis that would not be required for a speed-independent design. This timing analysis must establish relative orderings of events in the operation of the divider. Accordingly, the analysis depends on the sequences of states traversed by the divider. Thus, verification of timing and verification of functionality are not separable for this design.

To close this section, we include a few remarks about our model for the divider. Our design is based primarily on the description in [28]. Many details of the design are omitted from that paper, and we had to fill them by following the general design style guided by some earlier conversations with the designer. Although our design is not an exact replica of the chip described in [28], we believe that the two designs are very similar. For simplicity, we assumed the same upper and lower bounds on the delays of all logic elements. These bounds appear as variables in our proof, with the assumption that the minimum gate delay is no less than a certain percentage of the maximum delay. Our verification algorithms could easily be generalized to specific delay bounds for different classes of logic elements.

3 Proof Strategy

To verify the divider, we model the design with a hierarchy of Synchronized Transitions (henceforth called ST) programs. The most abstract program in this hierarchy describes only the SRT algorithm, details of handshakes and timing are omitted. The most detailed program models the design at the transistor level. Safety properties are verified for each program, establishing the correct function and timing of the design. Programs at the lower levels of the hierarchy are shown to be refinements of those at higher levels. As a consequence safety properties of the more abstract programs are inherited by the more detailed ones. The most detailed program maintains all of the properties verified for any of the programs. This approach to verification is described in greater detail in the remainder of this section.

3.1 Synchronized Transitions

In ST, programs are collections of guarded commands called **transitions** [24]. For example,

$$\ll \texttt{x > y} \rightarrow \texttt{x, y := y, x} \gg$$

is a transition that is enabled to swap `x` and `y` when `x` is greater than `y`. Two or more transitions may be combined using the asynchronous combinator, $||$. Consider the program

$$t_1||t_2||\ldots||t_n$$

Program execution consists of repeatedly selecting a transition, testing its guard, and, if the guard is satisfied, performing the multi-assignment. The order in which transitions are selected is unspecified: this nondeterminism corresponds to arbitrary delays in a speed-independent model.

To model bounded delays, we introduce a real-valued variable, τ, whose value corresponds to the current time (see [1]). If $\texttt{x.v}$ is the value of `x`, and $\texttt{x.}\tau$ is the time at which `x` acquired this value, then we can express a minimum delay of $\delta_{\min}$ for the swap transition by strengthening its guard as shown below:

$$\begin{array}{ll} \ll & (\texttt{x.v > y.v}) \wedge (\max(\texttt{x.}\tau, \texttt{y.}\tau) \geq \delta_{\min}) \\ \rightarrow & \texttt{x.v, y.v, x.}\tau\texttt{, y.}\tau \texttt{ := y.v, x.v, } \tau\texttt{, } \tau \\ \gg & \end{array}$$

The passage of time is modeled by a **protocol** [25] that describes the behaviour of the circuit's environment. In a protocol, $\texttt{x}.\mathit{pre}$ denotes the value of `x` before an environment action, and $\texttt{x}.\mathit{post}$ denotes the value of `x` afterwards. Figure 5 shows a protocol that asserts that time increases monotonically, the transition that swaps `x` and `y` has a maximum delay of $\delta_{\max}$, and environment actions leave `x` and `y` unchanged.

$$\begin{array}{l} \texttt{PROTOCOL} \\ \quad (\tau.\mathit{post} \geq \tau.\mathit{pre}) \\ \quad \wedge\ ((\texttt{x.v} > \texttt{y.v}) \Rightarrow (\tau.\mathit{post} < \max(\texttt{x.}\tau, \texttt{y.}\tau) + \delta_{\max})) \\ \quad \wedge\ (\texttt{x}.\mathit{post} = \texttt{x}.\mathit{pre}) \wedge (\texttt{y}.\mathit{post} = \texttt{y}.\mathit{pre}) \end{array}$$

Figure 5. A simple protocol

ST has two other operators for composing transitions and more elaborate protocol mechanisms that aren't used in this paper and are not described here. In the remainder of this paper, an ST program is an initial state predicate, a collection of transitions, and a protocol.

3.2 Semantics

The range of a type is the set of allowed values for a variable of that type. The state space of a program is the cross-product of the ranges of the types of the variables of the program. A state is an element of the state space.

Many properties of ST programs can be easily formulated using a *wp* semantics [9]. If P is a program and Q is a predicate, then $wp(P, Q)$ is the weakest condition that must hold such that Q is guaranteed to hold after any single action allowed by P is performed. Consider a transition $\ll G \rightarrow M \gg$: the guard, G, denotes a function from program states to the Booleans; the multi-assignment, M, denotes a function from states to states. In other words, if performing M from state s_1 leads to state s_2, then $M(s_1) = s_2$. A protocol is a predicate over pairs of states: $\mathit{proto}(s_1, s_2)$ is true if the protocol admits an environment action that starts in state s_1 and ends in state s_2. A *wp* semantics of ST is

$$\begin{array}{rcl} wp(\ll G \rightarrow M \gg, Q) & = & G \Rightarrow Q \circ M \\ wp(t_1||t_2||\ldots||t_n, Q) & = & \bigwedge_{i=1}^{n} wp(t_i, Q) \\ wp(\mathit{proto}, Q) & = & \lambda s_1.\ \forall s_2. \\ & & \quad \mathit{proto}(s_1, s_2) \Rightarrow Q(s_2) \end{array}$$

where $\circ$ denotes function composition.

We make extensive use of **invariants**. A predicate, I, is an invariant if I holding in some state implies that I will hold in all possible subsequent states of the program. A simple induction argument shows that I is an invariant of P *iff* $I \Rightarrow wp(P, I)$. Invariants are useful because they allow a property to be established for all possible program executions by considering a single step of the program.

A predicate Q is a **safety property** of P if Q holds in all states reachable in any execution of P. Let Q_0 be the initial state predicate of P. It can be shown (see [14]), that Q is a safety property of P if and only if there is an invariant I such that $Q_0 \Rightarrow I$ and $I \Rightarrow Q$. Often, finding an invariant requires the insight of the designer, and this approach is taken for some of the proofs in this paper.

In some cases, invariants can be found automatically by **model-checking** [7]. Let $win(P,Q)$ [14] be the least fixpoint of $wp(P)$ starting from Q, i.e. apply the wp operator until it converges. If the state space of P is finite, this computation will converge in a finite number of steps. The predicates in this fixpoint computation can be represented either explicitly or by symbolic means such as Binary Decision Diagrams [5, 6]. When model checking is used in this paper, we use a symbolic approach. The predicate Q is a safety property of P *iff* $Q_0 \Rightarrow win(P,Q)$.

Intuitively, program P' is a refinement of P if every state transition that P' can make corresponds to a move of P. More formally, let S' be the state space of P', S be the state space of P, and A be a mapping from S' to S. A is called an **abstraction mapping**; for simplicity, we assume that A is a function. Let Q_0, T and $proto$ be the initial state predicate, set of transitions, and protocol for P and Q'_0, T' and $proto'$ the same for P'.

Let s'_1 be a state of P'. We say that the transitions of P' are matched by the transitions of P at state s'_1 *iff* for every state s'_2 that is reachable by performing a single transition of P', there is a transition of P that effects a move from $A(s'_1)$ to $A(s'_2)$. Stuttering actions (where $A(s'_1) = A(s'_2)$) are exempted. We write $match_{T,A}(T',T)(s')$ to denote that the transitions of P' are matched by those of P at state s', with

$$\begin{array}{l} match_{T,A}(T',T)(s') \quad = \\ \quad \forall \ll G' \rightarrow M' \gg \in T'.\ G'(s') \Rightarrow \\ \qquad\qquad (A \circ M')(s') = A(s') \\ \qquad \vee \quad \exists \ll G \rightarrow M \gg \in T. \\ \qquad\qquad\quad (G \circ A)(s') \wedge ((A \circ M')(s') = (M \circ A)(s')) \end{array}$$

We define $match_{proto,A}$ in a similar fashion to indicate matching of protocol actions, again exempting stuttering actions:

$$\begin{array}{l} match_{proto,A}(proto', proto)(s'_1) \quad = \\ \quad \forall s'_2 \in S'. \\ \qquad\qquad proto'(s'_1, s'_2) \Rightarrow proto(A(s'_1), A(s'_2)) \\ \qquad \vee \quad A(s'_1) = A(s'_2) \end{array}$$

All actions of P' are matched by P at state s' if both transition actions and protocol actions are matched:

$$\begin{array}{rcl} Q_{P' \preceq_A P}(s') & = & match_{T,A}(T',T)(s') \\ & \wedge & match_{proto,A}(proto', proto)(s') \end{array}$$

Noting that $Q_{P' \preceq_A P}(s')$ is a predicate over states of P', we say that P' is a refinement of P under abstraction mapping A if $Q'_0 \Rightarrow Q_0 \circ A$ and $Q_{P' \preceq_A P}(s')$ is a safety property of P'. We write $P' \preceq_A P$ to denote this.

There are many useful relationships between refinement and safety properties. Two that are used extensively in this paper are described below. The first theorem states that safety properties of a more abstract program are inherited by refinements of the program.

Theorem 1 *Given programs P and P', an abstraction function A such that $P' \preceq_A P$, and a predicate Q such that Q is a safety property of P. Then $Q \circ A$ is a safety property of P'.*

This theorem is easily proven by induction over traces of P'.

The second theorem describes how safety properties of P can be used to show that P' is a refinement. This often reduces the problem of showing refinement to one of automatic tautology checking.

Theorem 2 *Given programs P and P' with initial state predicates Q_0 and Q'_0, an abstraction function A, and a predicate Q such that Q is a safety property of P. If $Q'_0 \Rightarrow Q_0 \circ A$ and $(Q \circ A) \Rightarrow Q_{P' \preceq_A P}$, then $P' \preceq_A P$,*

A simple induction over traces of P' shows that $Q_{P' \preceq_A P}(s')$ is a safety property of P' and establishes the claim.

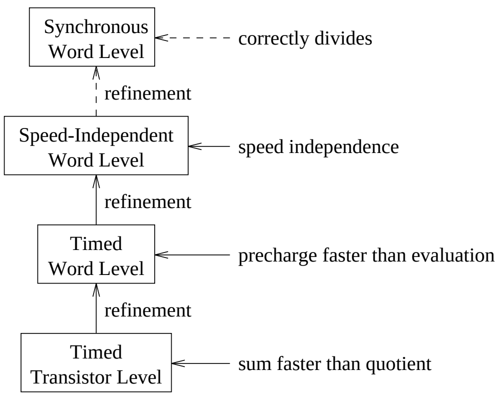

Figure 6. Verification Hierarchy

3.3 A Refinement Hierarchy for the Divider

Figure 6 shows the hierarchy of models that we are using to verify the divider. Each box is labeled with a description of the model at that level. Arrows indicate verification obligations: solid arrows indicate proofs that we have completed at the time of writing, dashed arrows indicate pending proofs. Vertical arrows correspond to refinement proofs. Horizontal arrows indicate other properties that either establish correctness or assist in the refinement proofs.

In the synchronous model, steps of the model correspond to steps of the SRT algorithm. We have an ST model for this level and have tested the model by compiling and executing the ST code. We need to formally verify the correspondence with the SRT algorithm, and then verify our description of the algorithm by showing that the result is indeed the quotient of the operands within appropriate rounding criteria.

The speed-independent model implements the handshaking shown in figure 3. We have verified speed independence of this model using model checking. We plan to verify that this model refines the synchronous model using methods similar to those described in [27].

In the timed, word-level model, bounds are given on the ratio of precharge time to evaluation time. This model implements the handshake protocol used in the actual chip as shown in figure 4.

The lowest-level model corresponds directly to our transistor-level implementation of the divider chip. At this level, we must show that the quotient bit output by a stage serves as a completion signal for the entire stage. In particular, we must show that it is not set until all outputs of the adder are valid. This requires an argument about the timing of events as data values propagate through the logic elements within the stage. The safety property that captures these timings is more detailed than any property that can be inherited from the word level model; accordingly, this refinement proof can not be performed by the tautology checking approach described earlier. Instead, we extended traditional methods of timing verification for combinational logic to apply to the precharged logic of the divider stages, as is described in the next section.

3.4 Timing Verification

As described in section 2, the dual-rail encoded values and precharged logic are used throughout the divider's data-path. Our timing verification techniques are adapted for this design style. To illustrate this approach to design, figure 7 shows the implementation of a logic element that computes $\mathtt{a} \wedge (\mathtt{b} \vee \neg \mathtt{c})$.

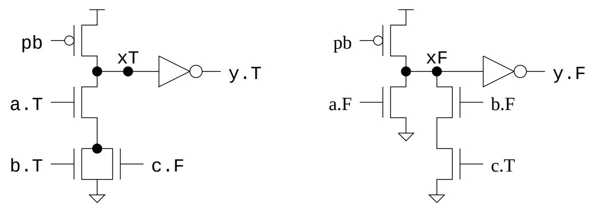

Figure 7. Implementation of $\mathtt{y} \leftarrow \mathtt{a} \wedge (\mathtt{b} \vee \neg \mathtt{c})$

When the "precharge bar" signal, `pb`, is low, internal signals `xT` and `xF` are precharged high which in turn brings outputs `y.T` and `y.F` low. Thus, precharging brings the logic element's output to a dual-rail empty value. When `pb` is high, the circuit can evaluate its function. If the `a`, `b`, and `c` inputs have values satisfying $\mathtt{a} \wedge (\mathtt{b} \vee \neg \mathtt{c})$ then `xT` will be pulled low which brings output `y.T` high. Signals `xF` and `y.F` will remain unchanged. Thus, the circuit will output the dual-rail encoding for "true" as required. Conversely, if the values of `a`, `b`, and `c` are dual-rail valid but do not satisfy $\mathtt{a} \wedge (\mathtt{b} \vee \neg \mathtt{c})$ the circuit will output a dual-rail false (i.e. `y.T` will be low and `y.F` will be high). The circuit is designed so that the pull-up transistors for precharging can overpower the corresponding pull-down networks. Thus, precharging takes precedence over evaluation. This circuit can be modeled by the four transitions below:

$$\begin{array}{ll}
 & \ll \mathtt{pb} \rightarrow \mathtt{y.T} \ \mathtt{:=}\ \mathtt{FALSE} \gg \\
\| & \ll \quad \neg\mathtt{pb} \wedge (\mathtt{a.T} \wedge (\mathtt{b.T} \vee \mathtt{c.F})) \\
 & \quad \rightarrow \mathtt{y.T} \ \mathtt{:=}\ \mathtt{TRUE} \\
 & \gg \\
\| & \ll \mathtt{pb} \rightarrow \mathtt{y.F} \ \mathtt{:=}\ \mathtt{FALSE} \gg \\
\| & \ll \quad \neg\mathtt{pb} \wedge (\mathtt{a.F} \vee (\mathtt{b.F} \wedge \mathtt{c.T})) \\
 & \quad \rightarrow \mathtt{y.F} \ \mathtt{:=}\ \mathtt{TRUE} \\
 & \gg
\end{array}$$

This example illustrated the design of a particular logical operation. Other operations are implemented in a similar manner by substituting appropriate pull-down networks for the ones used here.

In general, a precharged logic element has two outputs, `y.T` and `y.F` in the previous example. Any element can be modeled in ST by four transitions: one to set `y.T` false when `pb` is false, one to set `y.T` true when `pb` is true and the other inputs enable the pulldown network for `xT`, and two similar transitions for `y.F`. If the pull-down networks are series-parallel networks, then they can be transliterated into the guards for setting `y.T` and `y.F` true with conjunction representing series connections, and disjunction representing parallel connections. More general networks can be modeled by extensions of this approach. This allows the transistor level structure to be represented directly in the ST source code.

We model the timing of events within a stage using a directed graph. Let $\mathcal{T}$ be a set of transitions that are either "precharge" transitions of the form

$$\ll \neg\texttt{pb} \rightarrow \texttt{y := FALSE} \gg$$

or "eval" transitions of the form

$$\ll \texttt{pb} \wedge g_{\texttt{y}}(\texttt{x}_1,\ldots,\texttt{x}_k) \rightarrow \texttt{y := TRUE} \gg.$$

Here, g_y is a negation-free Boolean expression in the gate's inputs $\texttt{x}_1,\ldots,\texttt{x}_k$ containing only "and" and "or" connectives. We refer to g_y as the gate's **pull-down guard**.

The set of vertices $N_{\mathcal{T}}$ of the graph correspond to the signals appearing in the transitions of $\mathcal{T}$ (e.g. a.T or y.F). For signals u and v, there is an edge from u to v *iff* u is connected to the gate of a transistor in the pull-down network for v, i.e. u appears in $g_{\texttt{v}}$. Let $I_{\mathcal{T}}$ denote the set of **primary inputs** of a stage, i.e. those signals corresponding to graph nodes without incoming edges.

Evaluation in a stage starts when the pb signal becomes true and lasts until all outputs are valid. Let *eval* be a predicate that characterizes this evaluation mode; for the divider we use $eval(i) = \texttt{pb}(i) \wedge \neg\texttt{q}(i)$. For each vertex in the graph, we determine the earliest and latest time at which the signal corresponding to that vertex can become high. Our analysis requires that the following conditions are satisfied:

1. The graph for the stage and acyclic.

2. The stage is precharged when evaluation starts, i.e. all non-input signals of the stage (signals in $N_{\mathcal{T}} \setminus I_{\mathcal{T}}$) are *false* when pb is enabled to rise.

3. The "rest of the program", i.e. its protocol and the transitions not in $\mathcal{T}$, do not interfere with the evaluation, i.e. neither one is enabled to change the signals in $N_{\mathcal{T}}$ while *eval* holds.

4. The *eval* predicate and the pb signal relate in the following manner: $eval \Rightarrow \texttt{pb}$, rising edges of *eval* are due to rising edges of pb, $\texttt{pb}.\tau = \tau$ at the rising edge of pb, and $\texttt{pb}.\tau$ remains unchanged while *eval* holds.

The introduction of the *eval* predicate and the rather technical last assumption are necessary because pb being high is generally not strong enough to establish the non-interference condition. In case of the divider, if $\texttt{pb} \wedge \texttt{q}$ for a stage (i.e. this stage is in hold mode) then its predecessor may precharge and thus modify the stage's inputs.

Our verification of the divider includes a proof that these four conditions are satisfied.

We employ a simple timing model where for each signal, there is a minimum delay, δ_{min}, and a maximum delay, δ_{max} between when the signal is enabled to change and when it actually changes. Note that the structure of the pull-down guards ensures that the output of each gate is monotonic in its inputs. To compute the latest time at which a signal can change, we add the maximum delay of the gate to the latest settling time of any of the inputs to the gate. From the example of figure 7:

$$\begin{aligned} &latest^{\uparrow}(\texttt{y.T}) \;=\\ &\quad \max(latest^{\uparrow}(\texttt{a.T}), latest^{\uparrow}(\texttt{b.T}), latest^{\uparrow}(\texttt{c.F})) + \delta_{max}\\ &latest^{\uparrow}(\texttt{y.F}) \;=\\ &\quad \max(latest^{\uparrow}(\texttt{a.F}), latest^{\uparrow}(\texttt{b.F}), latest^{\uparrow}(\texttt{c.T})) + \delta_{max}\end{aligned}$$

It is easily shown that this estimate provides a conservative upper bound on the actual time of change (for primary inputs, we set $latest^{\uparrow}(\texttt{y}) = \texttt{pb}.\tau$ based on the assumption that the inputs are already settled when evaluation starts).

We could compute the earliest changing time in a similar manner. However, this would be overly conservative: from the example of figure 7, the earliest time at which y.T is enabled is no earlier than the earliest time at which a.T can change no matter how early b.T and c.F can change. Thus, we compute the earliest time at which an expression can be satisfied by the following rules:

1. If the expression is a signal y, then the time at which it can be satisfied is the earliest changing time of the signal $earliest^{\uparrow}(\texttt{y})$, or $\texttt{pb}.\tau$ if y is a primary input.

2. If the expression, e, is $e_1 \wedge e_2$, then the earliest time at which e can be satisfied is the maximum of the earliest times for e_1 and e_2.

3. If the expression, e, is $e_1 \vee e_2$, then the earliest time at which e can be satisfied is the minimum of the earliest times for e_1 and e_2.

These rules exploit the fact that our guard expressions are transliterations of the circuit structure. It is easily shown that this provides a conservative lower bound on the actual time at which the guard expression is satisfied. The earliest time at which the signal can change is the earliest time that the guard is satisfied plus the minimum gate delay, δ_{min}. From the example of figure 7:

$$\begin{aligned}
&\mathit{earliest}^{\uparrow}(\mathtt{y.T}) \;\; = \\
&\quad \max(\mathit{earliest}^{\uparrow}(\mathtt{a.T}), \\
&\qquad \min(\mathit{earliest}^{\uparrow}(\mathtt{b.T}), \mathit{earliest}^{\uparrow}(\mathtt{c.F}))) + \delta_{min} \\
&\mathit{earliest}^{\uparrow}(\mathtt{y.F}) \;\; = \\
&\quad \min(\mathit{earliest}^{\uparrow}(\mathtt{a.F}), \\
&\qquad \max(\mathit{earliest}^{\uparrow}(\mathtt{b.F}), \mathit{earliest}^{\uparrow}(\mathtt{c.T}))) + \delta_{min}
\end{aligned}$$

Since we assume the same minimum and maximum delays for all gates, the min and max operators can be pushed inwards and then evaluated on the constant factors. Thus, we can obtain the bounds on the settling times in the form

$$\begin{aligned}
\mathit{earliest}^{\uparrow}(\mathtt{y}) &= \mathtt{pb}(i).\tau + k_{e,\mathtt{y}}\,\delta_{min}, \\
\mathit{latest}^{\uparrow}(\mathtt{y}) &= \mathtt{pb}(i).\tau + k_{l,\mathtt{y}}\,\delta_{max},
\end{aligned} \tag{1}$$

with integers $k_{e,\mathtt{y}}$ and $k_{l,\mathtt{y}}$ that roughly correspond to shortest and longest paths in the graph.

We relate the above timing bounds for a signal to the signal's values as follows: Evaluation starts out with the signal being low, and the signal may not change before $\mathit{earliest}^{\uparrow}(\mathtt{y})$. Therefore, while *eval* holds,

$$\tau < \mathit{earliest}^{\uparrow}(\mathtt{y}) \Rightarrow \neg\mathtt{y}.$$

Correspondingly, $\mathtt{y}$ may not change after $\mathit{latest}^{\uparrow}(\mathtt{y})$, i.e. the signal must have settled. Thus, while *eval*,

$$\tau \geq \mathit{latest}^{\uparrow}(\mathtt{y}) \Rightarrow \mathit{settled}(\mathtt{y}).$$

The predicate *settled* has the property that for all non-input signals $\mathtt{y}$, $\mathit{settled}(\mathtt{y}) \Rightarrow (\mathtt{y} = g_{\mathtt{y}})$ and $\mathit{settled}(\mathtt{y}) \Rightarrow \mathit{settled}(\mathtt{x})$ for all input signals $\mathtt{x}$ of the gate driving $\mathtt{y}$.

4 The Proof Checker

The proofs presented in this paper were mechanically checked using a proof checker which is currently under development at UBC. This section will briefly introduce the main characteristics of the proof checking environment and explain how the domain specific proof rules introduced in the previous section were embedded in the verification framework.

4.1 Light-Weight Proof Checking

The design of our proof checker is motivated by the observation that there are many, highly automated techniques such as model checking, symbolic simulation, and various timing analysis algorithms for verifying specific properties of a design. However, many properties that a designer would wish to verify can not be addressed by a single, specialized tool. Theorem provers provide a general framework, but they also force the designer to translate the verification problem and proof strategy into mathematical logic, often a very tedious task. Our approach is to implement a simple proof checker where the results from specialized verification tools can be combined to produce a complete proof.

Similar to theorem proving environments such as ACL2, HOL, ISABELLE or PVS [13, 10, 19, 18], a (backwards-style) proof in our proof checker is represented as a sequence of proof states. A proof state in turn is implemented by an abstract data type, and consists essentially of a formula (the **claim**), a list of formulas (the **pending obligations**), as well as some book-keeping information. In the initial proof state of a proof, the list of pending obligations consists only of the claim. Rules of inference are implemented as functions from proof state to proof state, and are used to transform one or more pending obligations into zero or more (usually simpler) obligations. A proof state with no pending obligations corresponds to a proven claim, i.e. a **theorem**.

There are two important differences to existing theorem proving systems for 'general-purpose' logics such has first- or higher-order predicate logic:

- The core functionality of the checker, such as the handling of proof-states and theorems, is parameterized with respect to the syntax of formulas in a logic and its inference rules. In contrast to ISABELLE [19], where different object logics are embedded in the system's meta-logic, we use a simpler and more general, but less formal approach: The syntax of a logic is represented by an abstract data type and a function that recognizes well-formed formulas; inference rules are implemented as functions on lists of formulas.

- Our system is designed to support a style of proof that emphasizes the use of decision procedures. In addition to procedures that decide the validity of a single formula and can thus be used to discharge obligations, we also support proof rules where the justification of a proof step involves a non-trivial amount of computation (rather than a fairly simple rewrite of an obligation).

  In contrast to the more traditional embedding of decision procedures such as in ACL2 or PVS [13, 18], our system encourages the development of domain-specific decision procedures, allowing the controlled introduction of sound, but not formally verified reasoning into proofs.

The system has been implemented in SML/NJ [3]. The parameterization of the core checker is realized by its implementation as an SML functor. Similar to HOL, the user interface of the checker is the ML interactive top level.

4.2 General Purpose Decision Procedures

The proof checker includes a library that provides decision procedures for propositional logic and linear inequalities. The propositional logic tautology checker is based on Binary Decision Diagrams [5, 6]; to allow the treatment of non-trivial problems, we have integrated a state-of-the-art BDD package [22] into the SML/NJ runtime system. The availability of a variety of variable reordering techniques has proven vital to the completion of the divider verification.

We have also implemented a simple decision procedure for boolean expressions whose leaf-nodes are linear inequalities with rational coefficients. The procedure attempts to find a counter-example for a formula by rewriting its negation into sum-of-products form and solving the resulting linear programs [21]. The latter step uses an implementation of the Simplex algorithm for linear programs with arbitrary precision rational coefficients. Both the propositional and the linear inequality decision procedure can produce counter-examples for non-valid formulas. Such counter-examples are very helpful when developing a proof.

4.3 Reasoning about ST Programs

In the following, we will briefly describe the syntax and proof rules of a logic for reasoning about ST programs with which the generic core of the checker is instantiated. The logic itself is an extension of first order predicate logic and is explicitly typed (i.e. all variables have to be declared to be of a uniquely defined type).

The syntax of the logic includes all valid ST programs which allows ST programs to directly appear in proof obligations. Programs are imported into the proof checker with the help of a translator that compiles ST source modules into SML structures. The translator uses the front-end of an existing ST to C compiler; its back-end is very simple and merely transliterates abstract syntax trees into SML type constructors. This approach was deemed simpler and less error-prone than including an ST parser into the proof checker. It also ensures a high degree of consistency between the proof checker and other tools for ST programs.

4.4 Domain-Specific Proof Rules

Properties of ST programs are stated in terms of predicates such as $Inv(T_P, proto_P, I)$ or $SafetyP(T_P, proto_P, Q_{0,P}, Q)$, stating that an ST program P with transitions, protocol and initial state predicate T_P, $proto_P$, $Q_{0,P}$ has an invariant I and a safety property Q, respectively.

We provide proof rules based on the ST semantics described in section 3.2 that allow obligations involving these predicates to be rewritten into simpler obligations that can be discharged using the proof rules and decision procedures for first-order logic.

The (sequent-style) proof rules for invariants and safety properties take the form

$$\frac{\Gamma \vdash (I \Rightarrow wp(T_P, I)) \wedge ((I.pre \wedge proto_P) \Rightarrow I.post)}{\Gamma \vdash Inv(T_P, proto_P, I)} \tag{2}$$

and

$$\frac{\Gamma \vdash Inv(T_P, proto_P, I) \wedge (Q_{0,P} \Rightarrow Q) \wedge (I \Rightarrow Q)}{\Gamma \vdash SafetyP(T_P, proto_P, Q_{0,P}, Q)}$$

Note that proofs in our checker are written in backwards style, i.e. an obligation matching the conclusion of a rule may be replaced by its premises. Applying rule (2) requires the symbolic computation of the weakest precondition $wp(T_P, I)$. Recall that the weakest precondition of a single transition t_i and a predicate Q is given as

$$wp(\ll G_i \rightarrow M_i \gg, Q) = G_i \Rightarrow (Q \circ M_i). \tag{3}$$

Given an expression for Q, an expression for the composition $(Q \circ M_i)$ can be computed symbolically by substituting right-hand sides for left-hand sides of the multi-assignment in Q. If Q is given as a BDD and the left-hand sides of M_i are all Boolean variables, the substitution can be performed directly on the BDD for efficiency.

In order to apply the proof rule for invariants (2) the user is required to provide an invariant I that supports the safety property. For programs with finite state spaces, a supporting invariant can be computed automatically using the win-computation introduced in section 3.2. In cases where Q is in fact not a safety property, counter-example traces can be derived from the intermediate results of the underlying iteration.

The proof rule for refinement reduces an obligation claiming refinement between two programs to the obligation that the match predicate $Q_{P' \preceq_A P}$ is a safety property of the implementation and that the condition on the initial states holds. This requires the symbolic computation of an expression for $Q_{P' \preceq_A P}$. Since we restrict abstraction mappings to be functions, A can be represented as a vector of functions with a component for each variable in the abstract state program's state space. Then the function compositions with A that appear in the definition of $match_{T,A}$ and $match_{proto,A}$ can be computed symbolically by substitution in the same fashion as for multi-assignments in the wp computation.

The above proof rules assume that the ST program under consideration has the standard form $t_1 \| \ldots \| t_n$, where $t_1, \ldots, t_n$ are simple transitions. However, it is not practical to write non-trivial ST programs in this form. The specifications of the divider hierarchy make use of several advanced features of the ST language, such as cells, set-expressions, modules [24] etc. These language constructs provide syntactic sugar and can be defined in terms of transitions and the asynchronous combinator. We provide proof rules that allow a program that uses these features to be rewritten into a program in standard form.

4.5 Timing Verification

To allow the use of timing-based properties in the divider-proof, we have implemented a proof rule based on the timing analysis presented section 3.4. The rule introduces a theorem into the proof that essentially states that if a program has certain safety properties (the premises of the timing analysis) then it also has certain other safety properties (earliest and latest transition times):

$$\frac{\Gamma, \mathit{GraphAssmp} \Rightarrow \mathit{GraphProps} \vdash \Delta}{\Gamma \vdash \Delta}$$

The proof rule also enforces the provisos stated as syntactic properties of the program involved.

The assumptions and conclusions of the timing analysis have been described in section 3.4. In the following, we will describe in more detail how one of the assumptions as well as the conclusion are formalized in the proof checker.

Consider an ST program P with transitions, protocol and initial state predicate $T_P, proto_P, Q_{0,P}$. Let $\mathcal{T}$ be a subset of the transitions in T_P and $\overline{\mathcal{T}}$ the transitions of T_P not in $\mathcal{T}$.

Recall from section 3.4 that one of the assumptions of the timing analysis is that the "rest of the program" does not modify any of the inputs or internal signals of the circuit while it is in evaluation mode. This requirement is formalized using a predicate that none of a set of transitions is enabled to modify a particular signal:

$$\mathit{stable}_T(\{u_1, \ldots, u_k\}, y) \quad \triangleq \quad \bigwedge_{i=1}^{k} \mathit{stable}_t(u_i, y)$$

where

$$\mathit{stable}_t(\ll G \rightarrow M \gg, y) = \begin{cases} \neg G, & \text{if } (y := v) \in M, \\ \mathit{true}, & \text{otherwise.} \end{cases}$$

Likewise, non-interference with respect to a protocol is expressed as

$$\mathit{stable}_{proto}(\mathit{proto}, y) \quad \triangleq \quad \mathit{proto} \Rightarrow (y.\mathit{pre} = y.\mathit{post}).$$

Using these predicates, the requirement of non-interference of the "other" transitions $\overline{\mathcal{T}}$ and the protocol $proto_P$ with the internal signals $N_{\mathcal{T}}$ of a circuit is formalized as

$$\begin{aligned} &\mathit{NonInterference} \;= \\ &\quad \mathit{SafetyP}(T_P, \mathit{proto}_P, Q_{0,P}, \\ &\qquad \mathit{eval} \Rightarrow \\ &\qquad\quad \forall_{y \in N_{\mathcal{T}}}\ \mathit{stable}_T(\overline{\mathcal{T}}, y) \wedge \\ &\qquad\qquad \mathit{stable}_{proto}(\mathit{proto}_P, y)). \end{aligned}$$

The other safety properties appearing in the assumptions of the timing analysis are formulated in a similar manner; the syntactic condition that the timing graph is acyclic is checked by graph traversal.

The derived timing properties are stated as safety properties in a similar fashion:

$$\begin{aligned} &\mathit{GraphProps} \;\triangleq \\ &\quad \mathit{SafetyP}(T_P, \mathit{proto}_P, Q_{0,P}, \\ &\qquad \forall_{y \in N_{\mathcal{T}}}\ \mathit{eval} \Rightarrow \\ &\qquad\quad \tau < \mathit{earliest}^{\uparrow}(y) \Rightarrow \neg y \\ &\qquad\quad \wedge\, \tau \geq \mathit{latest}^{\uparrow}(y) \Rightarrow \mathit{settled}(y). \end{aligned}$$

5 Proofs

This section describes the proofs for the refinement hierarchy introduced in section 3.3. Space does not permit us to give a detailed account of all the proofs; instead we give a high-level overview of the proofs and provide further detail only for the proofs of selected key obligations and properties.

5.1 The Word Level Programs P_{wl} and P_{si}

The ST implementation of the speed-independent program is given in table 1. The remainder value determined by stage i is represented using two bit-vectors,

$$
\begin{array}{l}
\{ \ \|_{i=0}^{2} \ | \\
\quad\quad \ll \ \neg \mathrm{pb}(i \oplus 1) \rightarrow \mathrm{pb}(i) := \mathrm{TRUE} \ \gg \\
\quad \| \ll \mathrm{pb}(i \oplus 1) \ \wedge \ \mathrm{q}(i \oplus 1) \rightarrow \\
\quad\quad\quad \mathrm{pb}(i), \mathtt{sum(i)}, \mathtt{carry(i)}, \mathtt{qbit(i)} \ := \\
\quad\quad\quad \mathrm{FALSE}, \vec{\mathrm{FALSE}}, \vec{\mathrm{FALSE}}, 0 \ \gg \\
\quad \| \ll \neg \mathrm{pb}(i) \rightarrow \mathrm{q}(i) \ := \ \mathrm{FALSE} \ \gg \\
\quad \| \ll \mathrm{pb}(i) \ \wedge \ \neg \mathrm{q}(i \oplus 1) \rightarrow \\
\quad\quad\quad \mathrm{q}(i), \mathtt{sum(i)}, \mathtt{carry(i)}, \mathtt{qbit(i)} \ := \\
\quad\quad\quad \mathrm{TRUE}, \\
\quad\quad\quad \mathit{SRTsum}(\mathtt{sum}(i \ominus 1), \ \mathtt{carry}(i \ominus 1), \\
\quad\quad\quad\quad\quad \mathtt{qbit}(i \ominus 1), \ \mathtt{divisor}), \\
\quad\quad\quad \mathit{SRTcarry}(\mathtt{sum}(i \ominus 1), \ \mathtt{carry}(i \ominus 1), \\
\quad\quad\quad\quad\quad \mathtt{qbit}(i \ominus 1), \ \mathtt{divisor}), \\
\quad\quad\quad \mathit{SRTquot}(\mathtt{sum}(i \ominus 1), \ \mathtt{carry}(i \ominus 1), \\
\quad\quad\quad\quad\quad \mathtt{qbit}(i \ominus 1), \ \mathtt{divisor}) \ \gg \\
\}
\end{array}
$$

Table 1. Speed-Independent Program P_{si}

`carry(i)` and `sum(i)`, which correspond to the carry and sum part of the remainder. The variable `qbit(i)` is the quotient bit determined by stage i. The second guard of the fourth transition, $\neg\mathtt{q}(i \oplus 1)$, is necessary to ensure the speed-independence of the program. It guarantees that a stage will set its `q` signal to high *after* the successor stage has finished precharging (i.e. set its `q` signal to low). The speed-independence of this program was shown using model checking.

In the timed program P_{wl} the variables $\mathtt{pb}(i)$ and $\mathtt{q}(i)$ are structures with two fields, τ and v. The discrete value of the variable is stored in v and the time at which this value was assigned is stored in τ. P_{wl} is almost identical to P_{si}, except that the guard $\neg\mathtt{q}(i \oplus 1)$ of the last transition is replaced with the lower timing bound $(\tau \geq \mathtt{pb}(i).\tau + T_{\mathrm{evaluate}})$, which ensures that evaluation will take at least T_{evaluate} time units.

Besides stating that time is monotonic and that the environment does not modify any variable except τ, the environment protocol contains an upper bound for the time taken by precharging:

$$\mathtt{q}(i) \wedge \neg\mathtt{pb}(i).v \ \Rightarrow \ \tau.\mathit{post} < (\mathtt{pb}(i).\tau + T_{\mathrm{precharge}}) \, .$$

It is assumed that $0 < T_{\mathrm{precharge}} < T_{\mathrm{evaluate}}$.

There is an obvious abstraction mapping $A_{\mathrm{wl,si}}$ between P_{wl} and P_{si}, that identifies the corresponding data variables and maps the variables $\mathtt{pb}(i).v$ and $\mathtt{q}(i).v$ of the timed program to the variables $\mathtt{pb}(i)$ and $\mathtt{q}(i)$ of the speed-independent program. The $.\tau$ components are invisible under the abstraction mapping.

The initial state for both programs (as well as all other programs in the remainder of this section) is such that stage one is in hold-mode, stage two is precharged and about to begin evaluation and stage three is precharging. In all cases, the refinement condition for the initial states is discharged automatically by the tautology checker.

Theorem 3 *P_{wl} is a refinement of P_{si} under the abstraction mapping $A_{\mathrm{wl,si}}$.*

Proof *(sketch).* The first three transitions of P_{wl} are matched trivially by the corresponding transitions of P_{si}. The protocol actions of P_{wl} translate into stuttering steps. It remains to be shown that the "evaluate" transition of P_{si} is enabled whenever the "evaluate" transition of P_{wl} is enabled (note that their assignments trivially correspond under the abstraction mapping). This is reduces to showing that

$$(\tau \geq \mathtt{pb}(i).\tau + T_{\mathrm{evaluate}}) \ \Rightarrow \ \neg\mathtt{q}(i \oplus 1)$$

is a safety property of P_{wl}.

We discharge this obligation by stating an invariant Inv_{wl} that enumerates the possible combinations of values and time-stamps for the `pb` and `q` variables as indicated in figure 4. When showing that Inv_{wl} is indeed an invariant, the obligations resulting from the application of the proof rule for invariants (2) were automatically discharged using the linear inequality decision procedure. □

5.2 The Transistor Level Program P_{tl}

In the transistor level program P_{tl} (see table 2) each arithmetic stage is modeled by four cells, a multiplexer (*MUX*), a carry-save adder (*CSA*), a carry-propagate adder (*CPA*), and a quotient selection logic (*QSL*). Each of these cells corresponds to a functional block of the arithmetic stage and expands into transitions modeling the block's implementation in precharge logic.

The cell *QSL_Done* contains an or-gate that detects when there is a quotient available on the quotient signals `qbit`, and asserts the `qsl_done` signal accordingly. Likewise, cell *CSA_Done* detects the presence of valid dual-rail codes on the `carry` and `sum` outputs. Notice that `csa_done` is not read by any transition; the *CSA_Done* cell has been added to make the proofs less cumbersome and could in principle be removed.

Proving the refinement $P_{\mathrm{tl}} \preceq P_{\mathrm{wl}}$ under an appropriate abstraction mapping requires establishing a safety property of P_{tl}. However, due to the complexity and timed nature of this program, we can neither model-check, nor do we have the patience to manually devise a supporting invariant. We circumvent this problem using timing analysis of the transistor-level model, and

$\{ \|_{i=0}^{2} \;|$
$\quad \ll \neg \mathtt{pb}(i \oplus 1).v$
$\qquad \to \mathtt{pb}(i).v, \mathtt{pb}(i).\tau := \text{TRUE}, \tau \gg$
$\quad \| \ll \mathtt{pb}(i \oplus 1).v \;\wedge\; \mathtt{q}(i \oplus 1).v$
$\qquad \to \mathtt{pb}(i).v, \mathtt{pb}(i).\tau := \text{FALSE}, \tau \gg$
$\quad \| \ll \neg \mathtt{pb}(i) \to \mathtt{q}(i) \;:=\; \text{FALSE} \gg$
$\quad \| \ll \mathtt{pb}(i) \;\wedge\; \mathtt{qsl\_done} \to \mathtt{q}(i) \;:=\; \text{TRUE} \gg$
$\quad \|$ MUX ($\mathtt{pb}(i).v$, $\mathtt{qbit}(\mathtt{i} \ominus 1)$, $\mathtt{divisor}$, $\mathtt{muxout}$)
$\quad \|$ *CSA* ($\mathtt{pb}(i).v$, $\mathtt{carry}(\mathtt{i} \ominus 1)$, $\mathtt{sum}(\mathtt{i} \ominus 1)$, $\mathtt{muxout}$, $\mathtt{carry(i)}$, $\mathtt{sum(i)}$)
$\quad \|$ *CPA* ($\mathtt{pb}(i).v$, $\mathtt{carry(i)}$, $\mathtt{sum(i)}$, $\mathtt{cpsum}$)
$\quad \|$ *QSL* ($\mathtt{pb}(i).v$, $\mathtt{cpsum}$, $\mathtt{qbit(i)}$)
$\quad \|$ *QSL_Done* ($\mathtt{pb}(i).v$, $\mathtt{qbit(i)}$, $\mathtt{qsl\_done}$)
$\quad \|$ *CSA_Done* ($\mathtt{pb}(i).v$, $\mathtt{carry(i)}$, $\mathtt{sum(i)}$ $\mathtt{csa\_done}$)
$\}$

Table 2. Transistor Level Program P_{tl}

by introducing a side-hierarchy of models to establish key-safety properties of P_{tl} (see figure 8):

- The conclusions of the timing analysis as described in section 3.4, together with the safety property *validInputs*(i) that a stage's inputs are valid while it is evaluating, are strong enough to show refinement. However, the property *validInputs* as well as the safety properties in *GraphAssmp* (see section 4.5) cannot be "inherited" from P_{wl} since the latter does not contain sufficient detail (there are no dual-rail signals at the word level).

- To establish the required safety properties of P_{tl}, we construct an abstract model P_{abs} that has all the signals of P_{tl} but no functionality except for the handshake between stages. P_{abs} is constructed to be the most general program that still has the required safety properties. We show $P_{\text{tl}} \preceq P_{\text{abs}}$ and thus establish these properties for P_{tl} as well.

- P_{abs} is still a timed program; it has the same timing bounds for stages as the word level program P_{wl}. This is necessary to keep the proof for $P_{\text{tl}} \preceq P_{\text{abs}}$ manageable. However, we still cannot model check P_{abs}; instead, we construct a speed-independent version $P_{\text{abs,si}}$ and show $P_{\text{abs}} \preceq P_{\text{abs,si}}$.

- The program $P_{\text{abs,si}}$ finally admits model-checking, and we can show the required safety properties which are then inherited down the chain of refinements.

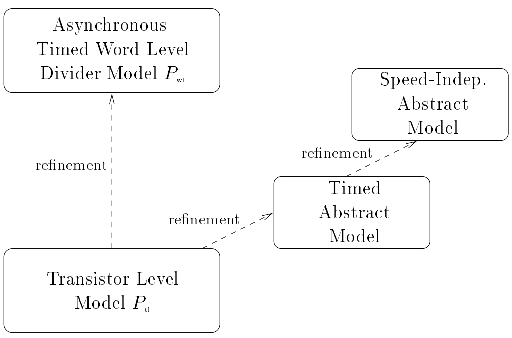

Figure 8. Refinement Proof

In the following, we will describe selected aspects of this proof in more detail.

5.2.1 Refinement between P_{tl} and P_{wl}

First, we need to define an appropriate abstraction mapping $A_{\text{tl,wl}}$ between the transistor-level and word-level specifications. In the word-level program all the bits of $\mathtt{carry}(i)$, $\mathtt{sum}(i)$ and $\mathtt{qbit}(i)$ are assigned simultaneously and always have consistent values. In the transistor-level program each transition only writes one variable. This means that there are reachable states of P_{tl} where some of the dual-rail encoded output bits of a stage are empty and others are TRUE or FALSE. The abstraction mapping between P_{tl} and P_{wl} needs to "hide" those states and only make the output of a stage "visible" when all signals have valid values.

Let A_{DR} be a function that maps dual-rail variables of the transistor level stage to the corresponding variable in the word level program, with the following conversion of dual-rail values to Boolean:

$$A_{\text{DR}}(x.T, x.F) \;=\; \begin{cases} \text{FALSE}, & \text{if } \neg x.T, \\ \text{TRUE}, & \text{if } x.T. \end{cases}$$

Furthermore, let $A_{\text{1-hot}}$ be a mapping from the three, one-hot encoded, transistor level $\mathtt{qbit}(\mathrm{i})$ variables to the integer valued $\mathtt{qbit}$ variable in P_{wl}:

$$A_{\text{1-hot}}(\mathtt{qbit}(i)) \;=\; \begin{cases} -1, & \text{if } \mathtt{qbit}_{-1}(i), \\ 1, & \text{if } \mathtt{qbit}_{1}(i), \\ 0, & \text{otherwise.} \end{cases}$$

Let $A_{\text{tl,wl}}$ be the mapping from states in the transistor level program P_{tl} to states in the word level program P_{wl}, defined component-wise as follows: If $\mathtt{pb}(i) \wedge \mathtt{q}(i)$ holds for stage i, then the transistor level $\mathtt{sum}$ and $\mathtt{carry}$ variables of stage i are mapped to the word level $\mathtt{sum}$ and $\mathtt{carry}$ variables using the function A_{DR}, and $\mathtt{qbit}(i)$ is mapped using $A_{\text{1-hot}}$. Otherwise, $A_{\text{tl,wl}}$ maps the $\mathtt{qbit}$ variables to the value 0 and all the transistor level `sum` and `carry` dual-rail variables to FALSE. The variables $\texttt{pb}(i)$ and $\texttt{q}(i)$ are mapped to the corresponding variables in the word level program. The internal signals of the arithmetic stages are not visible under the abstraction mapping.

Theorem 4 *Given that P_{tl} has the safety properties of GraphAssmp and the safety property validInputs, and two assumptions on gate delay bounds (stated below), then the program P_{tl} is a refinement of P_{wl} under the abstraction mapping $A_{\text{tl,wl}}$.*

Proof *(sketch).*
From the definition of the abstraction mapping $A_{\text{tl,wl}}$ follows that transitions in one of the *CSA*, *MUX*, *CPA*, *QSL* or *CSA_Done* cells of P_{tl} translate into stuttering steps of P_{wl}. The transitions that write `pb` are identical in both programs (subject to the abstraction mapping); note that the P_{wl} transition that sets $\texttt{pb}(i).v$ to low also clears the data signals to exactly the values that are visible under the abstraction mapping if the stage is not in hold-mode. Transition matching is therefore trivial for all state transitions except those where the value of $\texttt{q}(i)$ changes from FALSE to TRUE.

It follows that showing $P_{\text{tl}} \preceq_{A_{\text{tl,wl}}} P_{\text{wl}}$ requires proving the following obligations:

(*i*) When a transition in P_{tl} is enabled to set $\texttt{q}(i)$ to high, the transition in P_{wl} that sets $\texttt{q}(i)$ to high is also enabled.

(*ii*) Furthermore, it has to be shown that after $\texttt{q}(i)$ is set to high and the values of $\texttt{carry}(i)$, $\texttt{sum}(i)$ and $\texttt{qbit}(i)$ become "visible" under the abstraction mapping, these values agree with the values computed by the word level program.

(*iii*) The protocol of P_{tl} implies the protocol of P_{wl} composed with $A_{\text{tl,wl}}$.

Obligation (i):

It needs to be shown that

$$eval(i) \wedge \texttt{qsl\_done} \Rightarrow (\tau \geq \texttt{pb}(i).\tau + T_{\text{evaluate}})$$

is a safety property of P_{wl}. Since our assumptions permit timing analysis, we can establish

$$eval(i) \Rightarrow (\tau < earliest^{\uparrow}(\texttt{qsl\_done}) \Rightarrow \neg\texttt{qsl\_done}) \tag{4}$$

as a safety property. Recall from equation (1) that $earliest^{\uparrow}(\texttt{qsl\_done}) = \texttt{pb}(i).\tau + k_{e,qld}\,\delta_{min}$ for some integer $k_{e,qld}$. Thus, the obligation can be discharged under the assumption $T_{\text{evaluate}} \leq k_{e,qld}\,\delta_{min}$.

Obligation (ii):

To discharge this obligation, it needs to be shown that the circuitry of the transistor-level program indeed computes the Boolean functions prescribed by the word-level program.

Since from the guard of the P_{tl} transition it is only known that `qsl_done` is high, it needs to be established that the outputs of the carry-save adder are settled at this point. Again, this is done using timing analysis, from which we obtain

$$eval(i) \Rightarrow (\tau \geq latest^{\uparrow}(\texttt{csa\_done}) \Rightarrow settled(\texttt{csa\_done})),$$

where $latest^{\uparrow}(\texttt{csa\_done}) = \texttt{pb}(i).\tau + k_{l,csd}\,\delta_{max}$. Together with (4), this yields

$$eval(i) \wedge \texttt{qsl\_done} \Rightarrow settled(\texttt{csa\_done}),$$

under the assumption $k_{l,csd}\,\delta_{max} \leq k_{e,qld}\,\delta_{min}$.

The predicate *settled* entails the input-output relation of the combinational circuit implemented by the arithmetic stage. Based on this predicate, tautology checking proves that the values of the transistor-level stage outputs correspond (under the abstraction mapping) to the values of the word-level stages.

Obligation (iii):

The protocol of P_{tl} includes an upper bound for the transition that sets $\texttt{q}(i)$ to FALSE:

$$(\neg\texttt{pb}(i) \wedge \texttt{q}(i)) \quad \Rightarrow \quad (\tau < \texttt{pb}(i).\tau + \gamma_T^{\text{max}}).$$

Therefore, if $\gamma_T^{\text{max}} \leq T_{\text{precharge}}$, the protocol of P_{tl} implies the protocol of P_{wl} under the abstraction mapping $A_{\text{tl,wl}}$, and obligation (*iii*) is discharged.

□

5.2.2 The Abstract Programs $P_{\text{abs, si}}$ and P_{abs}

The abstract programs consist of three stages that have the same inputs and outputs as the transistor level stages, but they do not divide.

The higher-level abstract program, $P_{\text{abs, si}}$ is speed-independent and is used to establish safety properties for P_{tl} using model checking. The second abstract program, P_{abs}, has the same timing bounds as P_{wl} and is only used as an intermediate model to prove that the transistor level program is a refinement of $P_{\text{abs, si}}$. Both abstract programs have the same variables as the transistor level program and the same initial state predicate.

Each of the 55 carry and sum bits is assigned TRUE or FALSE nondeterministically under the restriction that dual-rail pairs cannot be set to *undefined*. All internal variables are also assigned values in this way. The quotient is nondeterministically set to -1, 0 or 1. During the precharge phase, all signals are set to FALSE.

The abstract models, like the transistor level divider, also have $\mathtt{q}(i)$ signals that control the evaluation flow. In the abstract models this signal is only set to high after explicitly testing that all the outputs of a stage (i.e. `qbit`, `carry` and `sum`) have valid values. In addition to testing that all outputs of a stage have valid values, $P_{\text{abs, si}}$ has $\neg\mathtt{q}(i \oplus 1)$ in the guard of the transition that sets $\mathtt{q}(i)$ to high.

Thus, the abstract models have the same state space as P_{tl} and are constructed to be as general as possible, but at the same time maintain the handshake protocol and several key safety properties. Most importantly, these include the property that a stage's outputs are valid and stable while it is in hold-mode (and its successor state in eval-mode). These and several other safety properties are proven for the speed-independent abstract model $P_{\text{abs,si}}$ using model checking.

The timed version of the abstract program P_{abs} is identical to $P_{\text{abs,si}}$ except that the $\neg\mathtt{q}(i \oplus 1)$-guard in the "evaluation completed" transition is replaced by timing bounds on evaluation and precharging in the same fashion as for the speed-independent and the timed word-level programs. Accordingly, the proof that $P_{\text{abs}} \preceq P_{\text{abs,si}}$ (with respect to the obvious abstraction mapping) is completely analogous to the proof of theorem 3.

The proof for $P_{\text{tl}} \preceq P_{\text{abs}}$ (again with respect to the obvious abstraction mapping) is very similar to the proof for $P_{\text{tl}} \preceq P_{\text{wl}}$ (theorem 4), but there are some notable differences:

1. Because the specification program has the safety properties needed for the timing analysis, we apply theorem 2 to discharge the assumptions of the timing analysis in this refinement proof.

2. The internal transitions of the stage do not translate into stuttering steps. Instead, it has to be shown that when a transition is enabled to raise a signal, then its dual-rail counterpart is low.

3. Instead of showing that the circuit computes a particular Boolean we show that its input-output relation transforms valid dual-rail codes into valid dual rail-codes.

6 Conclusions

We have described the verification of William's self-timed implementation of an SRT-based floating-point divider. We have established refinement between a speed-independent, word-level specification of the divider and a timed, transistor-level implementation. We are currently completing a proof that shows refinement between the speed-independent specification and a synchronous functional specification. The latter is suitable for bit-vector-level reasoning about the correct operation of the division algorithm.

Due to the large state space of the transistor level model, automatic verification based on timed state-space exploration is not feasible. Likewise, constructing a proof from first principles in a traditional theorem prover would be extremely time-consuming and tedious. Our approach employs automatic methods where possible (timing analysis and model checking), but also integrates key insights about the correct operation of the design (precharging faster than evaluation, carry-save-adder faster than quotient selection) into the proofs.

We devised a proof strategy for showing refinement between the transistor-level implementation and the timed word-level specification without actually proving any invariants or safety properties directly at the transistor-level. Instead, we used an approach based on refinement and timing analysis: Key safety properties of the implementation were established with the help of an abstraction that was constructed solely to support these properties. Then, these safety properties were used to justify the applicability of a graph-based timing analysis procedure, which in turn yielded safety properties of the transistor level that were sufficient to prove refinement. This step used a theorem that permits the use of safety properties of the abstract model as an assumption for the refinement proof.

Our first goal concerning future work is to complete the pending proofs (see figure 6). Secondly, we used a very simple timing model for the precharge gates, with common minimum and maximum gate delays, and data-independent bounds on settling-times. This was sufficient for verifying the divider and showed how a timing analysis procedure can be integrated into a more general verification environment. We believe that more sophisticated timing analysis algorithms could be incorporated using a similar approach.

Acknowledgements

Our thanks to Ted Williams who explained many details of his design to on our graph-based approach when we were just starting this research. And thanks to Andrew Appel and Lorenz Huelsbergen for answering our questions on integrating the CUDD package into SML.

References

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In J. de Bakker et al., editors, *Proceedings of the REX Workshop, "Real-Time: Theory in Practice"*. Springer, 1992. LNCS 600.

[2] R. Alur, T. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.

[3] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13. Springer-Verlag, Aug. 1991.

[4] W. Belluomini and C. J. Myers. Efficient timing analysis algorithms for timed state space exploration. In *Proceedings of the 1997 International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 88–100. IEEE, Apr. 1997.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[6] R. E. Bryant. Symbolic boolean manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.

[7] J. Burch, E. Clarke, et al. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, Apr. 1994.

[8] J. J. F. Cavanagh. *Digital Computer Arithmetic : Design and Implementation*. McGraw-Hill, New York, 1984.

[9] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[10] M. J. Gordon. HOL: a proof generating system for higher-order logic. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 74–128. Kluwer Academic Publishers, 1988.

[11] T. Henzinger, P. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Proceedings of the 27th Annual Symposium on Theory of Computing*, pages 373–382. ACM Press, 1995.

[12] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello. An algorithm for exact bounds on the time separation of events in concurrent systems. *IEEE Transactions on Computers*, 44(11):1306–1317, Nov. 1995.

[13] M. Kaufmann and J. S. Moore. ACL2: an industrial strength version of Nqthm. In *Proc. 11th Annual Conference on Computer Assurance (COMPASS '96)*, pages 23–34. IEEE Computer Society Press, June 1996.

[14] L. Lamport. *win* and *sin*: Predicate transformers for concurrency. Technical Report 17, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, May 1987.

[15] K. Larsen and F. Laroussinie. Compositional model checking of real time systems. In *Proceedings of CONCUR'95*, 1995. LNCS 962.

[16] G. Matsubara and N. Ide. A low power zero-overhead self-timed division and square root unit. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Apr. 1997.

[17] R. Miller. *Switching Theory*. Wiley, New York, 1965.

[18] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *8th International Conference on Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science. Springer-Verlag, July/Aug. 1996.

[19] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1994.

[20] C. L. Seitz. System timing. In *Introduction to VLSI Systems* (Carver Mead and Lynn Conway), chapter 7, pages 218–262. Addison Wesley, 1979.

[21] R. E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, Apr. 1979.

[22] F. Somenzi. CUDD: CU Decision Diagram Package. URL: `http://bessie.colorado.edu/~fabio/CUDD/cuddIntro.html`.

[23] J. Sparsø and J. Staunstrup. Delay-insensitive multi-ring structures. *INTEGRATION*, 15(3):313–340, Oct. 1993.

[24] J. Staunstrup. *A Formal Approach to Hardware Design*. Kluwer, 1993.

[25] J. Staunstrup and N. Mellergaard. Localized verification of modular designs. *Formal Methods in System Design*, 6(3):295–320, June 1995.

[26] S. Taşiran and R. K. Brayton. STARI: A case study in compositional and hierarchical timing verification. In *Proceedings of the Ninth Conference on Computer Aided Verification*, pages 191–201, Haifa, Israel, June 1997. Springer. LNCS 1254.

[27] D. Weih and M. Greenstreet. Verifying asynchronous data path circuits. *IEE Proceedings, Part E, Computers and Digital Techniques*, 143(5):295–300, Sept. 1996.

[28] T. Williams, M. Horowitz, et al. A self-timed chip for division. In *Proceedings of the Conference on Advanced Research in VLSI*. Computer Science Press, 1987.

[29] T. E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, May 1991.

[30] T. E. Williams. Analyzing and improving latency and throughput in self-timed pipelines and rings. In *TAU 1992 ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, Princeton, NJ, Mar. 1992.

[31] T. E. Williams, M. A. Horowitz, R. L. Alverson, and T. S. Yang. A self-timed chip for division. In *Stanford Conference on Advanced Research in VLSI*, pages 75–96, Mar. 1987.