Verification of Pipeline Circuits

Matt Kaufmann
David M. Russinoff

September 1, 2000

Abstract

The use of pipelines is an important technique in contemporary hardware design, particularly at the level of register-transfer logic (RTL). Earlier formal analysis (e.g., [4]) using the ACL2 theorem prover showed correctness of pipelined floating-point RTL. This paper extends that work by considering a notion of a *conditional pipeline*, essentially the result of sharing hardware among several distinct pipelines. We have employed a *pipeline tool*, written in ACL2 but completely unverified, to find a pipeline-related bug in an industrial hardware design, which has since been corrected. We then enhanced this tool to generate lemmas that we have used to prove properties of the corrected design. This paper presents a theoretical basis for this tool and describes its design and operation at an abstract level, showing how it fits into the correctness proof. This work may be viewed, from a high-level perspective, as encouragement for formal verifiers to prove properties of the actual RTL-level models, rather than stopping with their various abstractions.

1 Introduction

Practical algorithms for even the most elementary floating-point operations are sufficiently complex that their hardware implementations may require at least several cycles to execute. In order to maximize throughput, such operations may be pipelined by partitioning them into simpler computations that can be performed independently, each within a single cycle. Verifying that an algorithm is implemented correctly by a pipeline circuit requires demonstrating the absence of any resource conflicts that might result in interference between pipeline stages. Once this is done, the circuit may be modeled at a more abstract level without reference to time, i.e., it may be treated as combinational, which greatly simplifies its analysis.

In an earlier paper [4], we formalized the notion of a pipelined circuit and described a verification methodology that involves establishing the behavioral equivalence between a pipeline and a derived combinational circuit, which allows theorems about the latter to be transferred to the former. The theorems are mechanically checked with ACL2, supported by a library of lemmas pertaining

to bit vectors and floating-point arithmetic [5]. This methodology has been applied successfully in proving the correctness of a number of floating-point instructions of the AMD Athlon™ processor[1] with ACL2.

In general, however, pipelined circuits do not strictly conform to the simple definition presented in [4], which requires that a unique pipeline depth be associated with each signal. On the contrary, it is not uncommon for a single floating-point module to perform several operations of different latencies that involve similar computations, and may therefore be optimized through the sharing of circuitry. With respect to each operation, the module behaves as a simple pipeline subject to certain constraints. A given signal may be of different depths with respect to different pipelines.

An example of such a complex pipeline is a version of the AMD Athlon processor floating-point adder that performs several addition and subtraction operations, each of latency 4, along with a variety of floating-point comparisons of latencies 2 and 3. The module is known as the *merged adder* because it is the result of modifying an earlier version by implementing additional instructions, including the *3DNow!*™ instruction set, designed by AMD to support 3-dimensional graphics. The purpose of this paper is to extend our simple pipeline methodology to a larger class of circuits that includes the merged adder.

The circuitry with which we are concerned (indeed the entire AMD Athlon processor) is designed in a simple register-transfer logic (RTL) language, which is defined in [4]. Our approach is based on an automatic translator from this language to the logic of ACL2, also described in [4]. We begin in Sections 2 and 3 by reviewing both the RTL language and the translator, including extensions to the latter that have been implemented for our present purpose. In Section 4 we develop a theory based on a notion of *conditional pipeline* and identify a context in which several distinct conditional pipelines within a circuit may be modeled by means of the same combinational circuit. In Section 5, we present a scheme, based on this theory, using the merged adder as an illustration, by which ACL2 theorems pertaining to the pipelines can be derived from corresponding theorems about the combinational circuit. Section 6 describes a tool, written in ACL2, that analyzes pipelines and facilitates the ACL2 proofs of these theorems. This tool discovered a flaw in the pipeline structure of the merged adder: a number of signals were being accessed one cycle before their values were valid. The problem was easily corrected, before the design was committed to silicon.

2 Circuit Descriptions

Each signal s of a circuit description $\mathcal{D}$ in our RTL language is defined by a statement of one of three forms,

$$\texttt{input}\ s[n:0]; \tag{1}$$

[1] AMD, the AMD logo and combinations thereof, 3DNow!, and AMD Athlon are trademarks of Advanced Micro Devices, Inc.

$$s[n:0] \texttt{ = } E; \tag{2}$$

or

$$s[n:0] \texttt{ <= } E; \tag{3}$$

and s is thereby identified as an *input*, a *wire*, or a *register*, respectively. The *size* of s, which will be denoted as $\lambda(s)$, is defined to be $n+1$, where $n \in \mathbb{N}$. In the case of a wire or a register, E is an expression constructed from signals of $\mathcal{D}$ and standard logical connectives, called the *expression for* s. Any signal (input, wire, or register) may also appear in a declaration of the form

$$\texttt{output } s[n:0]; \tag{4}$$

and is thus identified as an *output* of $\mathcal{D}$.

Let $S(\mathcal{D})$ denote the set of signals of $\mathcal{D}$, and let $I(\mathcal{D})$, $O(\mathcal{D})$, $W(\mathcal{D})$, and $R(\mathcal{D})$ denote the subsets consisting of its inputs, outputs, wires, and registers, respectively. Note that $S(\mathcal{D})$ is the disjoint union of $I(\mathcal{D})$, $W(\mathcal{D})$, and $R(\mathcal{D})$.

Let $s, s' \in S(\mathcal{D})$ and let E be the expression for s. Then s' is a *combinational supporter* of s iff $s \in W(\mathcal{D})$ and either s' occurs in E or s' is a combinational supporter of some wire occurring in E. It is a syntactic requirement of the language that no wire is a combinational supporter of itself. We derive a weaker notion of support by dropping the requirement that s be a wire: s' is a *supporter* of s iff either s' occurs in E or s' is a supporter of some signal occurring in E. A circuit $\mathcal{D}$ is *acyclic* if no signal of $\mathcal{D}$ supports itself. In this paper, we shall consider only acyclic circuits.[2]

A valuation for $\mathcal{D}$ is a partial function v from $S(\mathcal{D})$ to $\mathbb{N}$ such that for each $s \in dom(v)$, $v(s)$ is a bit vector of length $\lambda(s)$. If a valuation v is defined on all signals that occur in an expression E, then $v(E)$ is defined in the natural way, e.g., $v(\texttt{x \& y}) = v(\texttt{x}) \texttt{ \& } v(\texttt{y})$, and in this case, if E is the expression for a signal s, then $v(E)$ is always a bit vector of length $\lambda(s)$. If the domain of v is $I(\mathcal{D})$ or $R(\mathcal{D})$, then v is called an *input valuation* or a *register state*, respectively. Given an input valuation $\mathcal{I}$ and a register state $\mathcal{R}$, there exists a unique valuation with domain $S(\mathcal{D})$, which we shall denote as $\mathcal{V}_{\mathcal{D},\mathcal{I},\mathcal{R}}$, that extends both $\mathcal{I}$ and $\mathcal{R}$ and such that $\mathcal{V}_{\mathcal{D},\mathcal{I},\mathcal{R}}(s) = \mathcal{V}_{\mathcal{D},\mathcal{I},\mathcal{R}}(E)$ whenever s is a wire and E is the expression for s.

The semantics of circuit descriptions are based on an underlying notion of *clock cycle*. Let $\mathcal{I}_0, \mathcal{I}_1, \ldots$ be a sequence of input valuations and let $\mathcal{R}_0$ be a register state for $\mathcal{D}$. We think of each $\mathcal{I}_k$ as representing the values of the input signals of $\mathcal{D}$ on the k^{th} cycle of an execution, and $\mathcal{R}_0$ as an initial set of register values. Register states for successive cycles are computed as

$$\mathcal{R}_{k+1} = next_{\mathcal{D}}(\mathcal{I}_k, \mathcal{R}_k),$$

[2] In the actual RTL, a register is sometimes assigned conditionally, i.e., it can hold its previous value when this value is irrelevant. Our translation replaces this irrelevant value with 0 in order to produce a strictly acyclic circuit.

where the function $next_{\mathcal{D}}$ is defined as follows: Given an input valuation $\mathcal{I}$ and a register state $\mathcal{R}$, $next_{\mathcal{D}}(\mathcal{I}, \mathcal{R})$ is the register state $\mathcal{R}'$, where for each $s \in R(\mathcal{D})$, if E is the expression for s, then

$$\mathcal{R}'(s) = \mathcal{V}_{\mathcal{D},\mathcal{I},\mathcal{R}}(E).$$

If $R(\mathcal{D})$ is empty, then $\mathcal{D}$ is *combinational*; otherwise, $\mathcal{D}$ is *sequential*. In the combinational case, the semantics of $\mathcal{D}$ are considerably simpler: the value of each signal on cycle k is determined by the input valuation $\mathcal{I}_k$, and thus we may write $\mathcal{V}_{\mathcal{D},\mathcal{I}}$ unambiguously. Naturally, it is in general easier to characterize the behavior of combinational circuits than sequential circuits.

For any acyclic circuit $\mathcal{D}$, the result of replacing each register assignment (3) of $\mathcal{D}$ by the corresponding wire assignment (2) is a combinational circuit, which we shall denote as $\tilde{\mathcal{D}}$. In Section 4, we shall examine a class of acyclic circuits $\mathcal{D}$ for which we can establish a relation between $\mathcal{D}$ and $\tilde{\mathcal{D}}$ that allows us to derive properties of $\mathcal{D}$ from corresponding properties of $\tilde{\mathcal{D}}$. Since we hope to support our proofs with the ACL2 prover, we shall first describe our scheme for modeling circuits in the ACL2 language.

3 Translation to ACL2

The RTL-ACL2 translator analyzes the input declarations and assignments of a circuit description $\mathcal{D}$, classifies each signal as an input, a wire, or a register, and determines its size $\lambda(s)$. If a signal s is a wire or a register, then the expression for s is translated into an ACL2 term whose free variables are the signals occurring in the expression. This requires a translation of each RTL construct into some ACL2 function. For example, the term derived from the wire definition

```
M4_mantissa_ols_high[22:0] =
  {M4_sum71_co[68:47],
   M4_sum71_co[46] & ~(~M4_sum71_co[45] &
                       ~M4_sticky_ols_high & M4_RN_sel)};
```

is

```
(cat (bits m4_sum71_co 68 47)
     (logand (bitn m4_sum71_co 46)
             (comp1 (logand (logand (comp1 (bitn m4_sum71_co 45)
                                           1)
                                    (comp1 m4_sticky_ols_high
                                           1))
                            m4_rn_sel)
                    1))
     1)).
```

Here, `logand` is the LISP primitive for logical (bitwise) "and", and `cat`, `comp1`, `bitn`, and `bits` are defined functions that perform concatenation, bitwise complementation, bit extraction, and bit slicing, respectively (see [4]).

Next, the translator generates a unary function corresponding to each signal, the argument of which represents a cycle number. For $s \in I(\mathcal{D})$, the function that represents s is undefined, constrained only to have the required length $\lambda(s)$. Let $I(\mathcal{D}) = \{q_1, \ldots, q_N\}$. Then the following single `encapsulate` form is generated:

```
(encapsulate ((q1 (n) t) ... (qN (n) t))
  (local (defun q1 (n) (declare (ignore n)) 0))
  (defthm bvecp-q1
     (bvecp (q1 n) λ(q1))
     :rule-classes :rewrite ...)
  ...
  (local (defun qN (n) (declare (ignore n)) 0))
  (defthm bvecp-qN
     (bvecp (qN n) λ(qN))
     :rule-classes :rewrite ...)).
```

The functions corresponding to wires and registers are then defined in terms of the input functions. For each wire s, the following definition is generated, in which we write `(f` $a_1 \ldots a_j$`)` to denote the form derived from the expression for s:

```
(defun s (n)
  (f (a1 n) ... (aj n))).
```

If s is a register, then the value of s on cycle n is determined by the values of $a_1, \ldots, a_j$ on cycle $n-1$. In order to avoid any assumptions about the value of s on cycle 0, we make use of an undefined function `unknown`, which is constrained to return a bit vector of a specified length. The following definition is generated:

```
(defun s (n)
  (if (zp n)
      (unknown 's λ(s))
    (f (a1 (1- n)) ... (aj (1- n))))).
```

In the general case, these functions form a mutually recursive clique. In order for the definitions to be admitted by ACL2, a well-founded measure must be supplied explicitly. To this end, the translator computes the number $\chi(s)$ of combinational supporters of each signal s ($\chi(s) = 0$ unless s is a wire), and inserts the following declaration in the definition for s:

```
(declare (xargs :measure (cons (1+ n) χ(s)))).
```

The prover is then able to establish their admissibility automatically.

The library lemmas that we plan to apply to these functions generally depend on the length $\lambda(s)$ of the bit vector `(s n)`. Therefore, the translator has been

programmed to generate, in addition to the bvecp-q_i lemmas, the following for each wire or register s:

```
(defthm bvecp-s
  (bvecp (s n) λ(s))
  :rule-classes :rewrite ...)
```

(An auxiliary file contains lemmas that guarantee that these lemmas are proved automatically.)

The translator behaves differently if the user declares the circuit $\mathcal{D}$ to be acyclic. In this case, once the claim of acyclicity is verified, the signal definitions are ordered so that no signal precedes any of its supporters. Since there is no mutual recursion, the measure declarations may be omitted. More significantly, in addition to the primary model described above, two alternative models of the combinational circuit $\tilde{\mathcal{D}}$ are constructed.

The first of these two is intended for execution. We define a second function corresponding to each wire s of $\tilde{\mathcal{D}}$ (i.e., each wire or register of $\mathcal{D}$). This function belongs to a separate package, named "+"; all functions previously mentioned belong to the "ACL2" package. Its arguments correspond to the signals that occur in the expression for s:

```
(defun +::s (a_1 ... a_j)
  (f a_1 ... a_j)).
```

One other function is included in this model of $\tilde{\mathcal{D}}$. Let $W(\tilde{\mathcal{D}}) = \{s_1, \ldots, s_L\}$, ordered so that no signal precedes any of its supporters, and for $i = 1, \ldots, L$, let $a_{i1}, \ldots, a_{ij_i}$ be the arguments of $+::s_i$. Suppose that we are interested in observing the behavior of some set of signals $\{r_1, \ldots, r_\ell\}$. Then we may program the translator to generate this definition:

```
(defun execute (var q_1 ... q_N)
  (let* ((s_1 (+::s_1 a_11 ... a_1j_1))
         (s_2 (+::s_2 a_21 ... a_2j_2))
          ...
         (s_N (+::s_L a_N1 ... a_Nj_N)))
    (case var
      (r_1 r_1)
      (r_2 r_2)
      ...
      (r_ℓ r_ℓ).
```

The arguments of execute are the input signals $q_1, \ldots, q_N$ and one additional argument, var, which ranges over the signals of interest. It is clear from the definition that the function simply computes the value of each signal in succession and returns the value of the selected signal.

For the other model of $\tilde{\mathcal{D}}$, yet another function is generated for each signal, in a package named "*". These functions have no arguments; their values represent the values of the corresponding signals that are determined by an unspecified

set of input values. The inputs are again introduced by an `encapsulate` form that constrains their values only to be bit vectors of the appropriate lengths:

```
(encapsulate ((*::q1 () t) ... (*::qN () t))
  (local (defun *::q1 () 0))
  (defthm bvecp*q1
     (bvecp (*::q1) λ(q1))
     :rule-classes ...)
  ...
  (local (defun *::qN () 0))
  (defthm bvecp*qN
     (bvecp (*::qN) λ(qN))
     :rule-classes ...)).
```

The other signal values are then defined in terms of these:

```
(defun * :: s () (f (*::a1) ... (*::aj))).
```

As in the `"ACL2"` model, we also have the following for each s:

```
(defthm bvecp-*-s
  (bvecp (*::s) λ(s))
  :rule-classes :rewrite ...)
```

Whenever the `"*"` or `"ACL2"` model is included in a proof session, these lemmas are loaded along with the signal definitions. The definitions are then all disabled, to be enabled individually as required.

Of the two alternative models of $\tilde{\mathcal{D}}$, the `"+"` model has the advantage of being executable while the `"*"` model (for reasons discussed in [4]) is more amenable to formal analysis. It is not difficult to show, for any acyclic circuit $\mathcal{D}$, that the two are equivalent. First, we first prove the following rewrite rule for each signal s_i, $1 \leq i \leq L$:

```
(equal (+::si (*::ai1) ... (*::aiji))
       (*::si)).
```

(These lemmas may be generated and proved automatically.) Then, with all of the functions `*::`s_i and `+::`s_i disabled (and the function `execute` enabled), the following equivalence is readily proved, for each of the signals $r_1, \ldots, r_\ell$:

```
(equal (*::ri) (execute 'ri (*::q1) ... (*::qN))).
```

Ultimately, however, we are interested in the behavior of the primary `"ACL2"` model of $\mathcal{D}$. The other two are useful only insofar as we are able to relate them to the `"ACL2"` model. Of course, if $\mathcal{D}$ happens to be a combinational circuit, i.e., $\mathcal{D} = \tilde{\mathcal{D}}$, then equivalence is trivial—the following theorem may be proved, for each signal s_i of $\mathcal{D}$, simply by enabling and disabling definitions as appropriate:

```
(implies (and (equal (q1 n) (*::q1))
              (equal (q2 n) (*::q2))
              ...
```

```
        (equal (q_N n) (*::q_N)))
(equal (s_i n) (*::s_i))).
```

Of course, the circuits with which we are concerned are generally not combinational. However, in the next section, we shall describe a class of pipeline circuits for which we can prove lemmas similar to the above, providing a means for "lifting" results pertaining to the `"*"` model to the `"ACL2"` model. For these circuits, we are thus free to focus on the combinational circuit $\tilde{\mathcal{D}}$, using the `"+"` model for testing and concentrating our proof effort on the simpler `"*"` model.

4 Pipelines

Let $\delta : S(\mathcal{D}) \to \mathbb{N}$. We shall say that $\mathcal{D}$ is a *pipeline with depth function* δ if

(1) for all $s \in W(\mathcal{D})$, $\delta(s') = \delta(s)$ for each signal s' occurring in the expression for s, and

(2) for all $s \in R(\mathcal{D})$, $\delta(s) > 0$ and $\delta(s') = \delta(s) - 1$ for each signal s' occurring in the expression for s.

Obviously, any combinational circuit is a pipeline with constant depth function $\delta(s) = 0$. For a nontrivial example of a sequential pipeline, the reader is referred to the floating-point multiplier presented in [4].

It is clear that any pipeline is acyclic. It is also easy to show that a pipeline $\mathcal{D}$ is related to the derived combinational circuit $\tilde{\mathcal{D}}$ as stated in the following theorem.

Theorem 1 *Let $\mathcal{D}$ be a pipeline with depth function δ. Let $\{\mathcal{I}_0, \mathcal{I}_1, \dots\}$ be a sequence of input valuations and let $\mathcal{R}_0$ be a register state for $\mathcal{D}$. For all $k \in \mathbb{N}$, let $\mathcal{R}_{k+1} = next_{\mathcal{D}}(\mathcal{I}_k, \mathcal{R}_k)$. Let $\mathcal{I}$ be the input valuation defined by $\mathcal{I}(s) = \mathcal{I}_{\delta(s)}(s)$ for each input s. Then for any signal s of $\mathcal{D}$,*

$$\mathcal{V}_{\mathcal{D},\mathcal{I}_{\delta(s)},\mathcal{R}_{\delta(s)}}(s) = \mathcal{V}_{\tilde{\mathcal{D}},\mathcal{I}}(s).$$

In fact, we shall prove a generalization of Theorem 1, concerning a larger class of circuits that behave as pipelines under certain constraints.

A *constraint set* for $\mathcal{D}$ is defined to be a set of triples $\mathcal{A} \subset S \times \mathbb{N} \times \mathbb{N}$ such that if $(s, c, d), (s', c', d') \in \mathcal{A}$ and $s = s'$, then $c = c'$. The *domain* of a constraint set $\mathcal{A}$ is

$$dom(\mathcal{A}) = \{s \in S(\mathcal{D}) : (s, c, d) \in \mathcal{A} \text{ for some } c \text{ and } d\}.$$

We shall say that an expression E is *forced* to the value u at depth d under $\mathcal{A}$ if $v(E) = u$ for every valuation v that satisfies $v(s) = c$ for all $(s, c, d) \in \mathcal{A}$. Note that a given expression cannot be forced to different values at different depths. Given a set P of signals, E is *determined* by P at depth d under $\mathcal{A}$ if $v(E) = v'(E)$ for any two valuations v and v' such that

(a) $v(s) = v'(s)$ for all $s \in P$, and

(b) $v(s) = v'(s) = c$ for all $(s, c, d) \in \mathcal{A}$.

A sequence of input valuations $\{\mathcal{I}_0, \mathcal{I}_1, \mathcal{I}_2, \dots\}$ *satisfies* a constraint set $\mathcal{A}$ if for any register state $\mathcal{R}_0$, and for all $(s, c, d) \in \mathcal{A}$, $\mathcal{V}_{\mathcal{D},\mathcal{I}_d,\mathcal{R}_d}(s) = c$, where $R_{k+1} = next_{\mathcal{D}}(\mathcal{I}_k, \mathcal{R}_k)$ for $k \in \mathbb{N}$.

The *closure* $\bar{\mathcal{A}}$ of $\mathcal{A}$ is recursively defined as follows: $(s, c, d) \in \bar{\mathcal{A}}$ iff

(a) $(s, c, d) \in \mathcal{A}$, or

(b) $s \in W$ and the expression for s is forced to c at depth d under $\bar{\mathcal{A}}$, or

(c) $s \in R$, $d > 0$, and the expression for s is forced to c at depth $d-1$ under $\bar{\mathcal{A}}$.

It is clear that any sequence of input valuations that satisfies $\mathcal{A}$ must also satisfy $\bar{\mathcal{A}}$.

We are ready to consider a generalized notion of pipeline, which involves selecting a relevant subset P of the signals of $\mathcal{D}$ according to a set $\mathcal{A}$ of constraints. Let $\mathcal{A}$ be a constraint set for $\mathcal{D}$, let $P \subset S(\mathcal{D})$ such that $P \cap dom(\mathcal{A}) = \emptyset$, and let $\delta : P \to \mathbb{N}$. Then $\mathcal{D}$ is a *conditional pipeline under $\mathcal{A}$ with pipeline signals P and depth function δ* if for all $d \in \mathbb{N}$,

(a) for every $s \in W(\mathcal{D}) \cap P$, if $\delta(s) = d$, then the expression for s is determined by $P \cap \delta^{-1}(d)$ at depth d under $\bar{\mathcal{A}}$; and

(b) for every $s \in R(\mathcal{D}) \cap P$, if $\delta(s) = d$, then $d > 0$ and the expression for s is determined by $P \cap \delta^{-1}(d-1)$ at depth $d-1$ under $\bar{\mathcal{A}}$.

The circuit that originally motivated this definition is the merged adder mentioned in Section 1. The operations performed by this circuit include those of the original AMD Athlon processor floating-point adder, to which we shall refer as *FPA*, as well as the 3DNow! instruction extensions, which we shall call *F3A*. Each of these sets contains operations of latencies 2, 3, and 4. Thus, the full set of operations is partitioned into six subsets. For each of these subsets, the circuit may be viewed as a conditional pipeline with corresponding pipeline signals and constraints.

Correct behavior of the merged adder requires that two operations are never initiated on the same cycle, and that two operations of different latencies are never scheduled to terminate on the same cycle. An FPA or F3A operation is initiated by setting the control input `EPC_FPA_EN0_11` or `EPC_F3A_EN0_11`, respectively, and encoding the operation in the opcode input, `EPC_EX1_FP0pcode0_11`. Operands are represented by the signals `PIPE0_FSRC1_11` and `PIPE0_FSRC2_11`. There is an additional control input, `EPC_FPA_WB0_14`, which simply echoes `EPC_FPA_EN0_11` after a 3-cycle delay.

Thus, for example, in order for a long (4-cycle) F3A operation to be initiated at cycle n and to execute correctly, the following predicate must hold:

```
(defun f3a-long-op-assumptions (n)
 (and
      ;n is a positive integer:

      (not (zp n))

      ;;f3a 4-cycle op is initiated at n:

      (equal (epc_f3a_en0_11 n) 1)
      (f3a-long-op (epc_ex1_fpopcode0_11 n))
      (admissible-operand-p (pipe0_fsrc1_11 n))
      (admissible-operand-p (pipe0_fsrc2_11 n))

      ;;no fpa op is initiated at n:

      (equal (epc_fpa_en0_11 n) 0)
      (equal (epc_fpa_wb0_14 (+ 3 n)) 0)

      ;;no 3-cycle op is initiated at n+1:

      (or (equal (epc_f3a_en0_11 (+ 1 n)) 0)
          (not (f3a-mid-op (epc_ex1_fpopcode0_11 (+ 1 n)))))
      (or (equal (epc_fpa_en0_11 (+ 1 n)) 0)
          (not (fpa-mid-op (epc_ex1_fpopcode0_11 (+ 1 n)))))

      ;;no 2-cycle op is initiated at n+2:

      (or (equal (epc_f3a_en0_11 (+ 2 n)) 0)
          (not (f3a-short-op (epc_ex1_fpopcode0_11 (+ 2 n)))))
      (or (equal (epc_fpa_en0_11 (+ 2 n)) 0)
          (not (fpa-short-op (epc_ex1_fpopcode0_11 (+ 2 n)))))).
```

This definition states not only that the operation is initiated correctly, but also that no other operation interferes with its execution. This latter point is made rigorous by showing that the circuit may be described as a conditional pipeline; we do so by expressing the appropriate constraints.

The condition that no FPA operation is initiated at cycle n is obviously ensured by two constraints:

```
(EPC_FPA_EN0_11,0,0)
(EPC_FPA_WB0_14,0,3).
```

The other constraints are less obvious, but for each relevant class of operations, there happens to be a signal that is set whenever an operation from that class is initiated. For example, there are three 2-cycle F3A operations that return the minimum of two operands. Their opcodes are represented by

the constants F3MIN, F3MINPS, and F3MINSS, and thus they correspond to the signal F3A_min_11:

```
F3A_min_11 = EPC_F3A_EN0_11 &
             (EPC_EX1_FPOpcode0_11[11:0] == 'F3MIN |
              EPC_EX1_FPOpcode0_11[11:0] == 'FPKMINPS |
              EPC_EX1_FPOpcode0_11[11:0] == 'FPKMINSS).
```

Clearly, the condition that none of these operations is initiated at either cycle n or cycle $n+2$ is represented by these two constraints:

```
(F3A_min_11,0,0)
(F3A_min_11,0,2).
```

Altogether, 37 constraints are required in the formulation of this conditional pipeline.

The pipeline signals naturally include the operands and opcode, which are assigned depth 0. There are five other pipeline inputs—four exception masks and one that controls rounding—which are received one cycle later and therefore have depth 1. There are seven outputs of interest—the result of the operation and six exception flags—all of which have depth 4. Thus, correctness of the pipeline is described by the following implication, in which the conclusion expresses some relation among the pipeline inputs and outputs:

```
(defthm correctness-of-f3a-long-ops
  (implies (f3a-long-op-assumptions n)
           (f3a-long-op-result (pipe0_fsrc1_11 n)
                               (pipe0_fsrc2_11 n)
                               (epc_ex1_fpopcode0_11 n)
                               (rq_speccwrc_13 (+ 1 n))
                               (rq_speccwom_13 (+ 1 n))
                               (rq_speccwum_13 (+ 1 n))
                               (rq_speccwdm_13 (+ 1 n))
                               (rq_speccwim_13 (+ 1 n))
                               (pipe0_fres_fpa_15 (+ 4 n))
                               (ex_faultnormalize0_fpa (+ 4 n))
                               (ex_invalidxcp0 (+ 4 n))
                               (ex_denormxcp0 (+ 4 n))
                               (ex_inexactresult0 (+ 4 n))
                               (ex_numoverflow0 (+ 4 n))
                               (ex_tinyresult0 (+ 4 n))))).
```

Of course, the other pipelines of the circuit have different constraint sets and depth functions, although pipeline signals are shared. For the 3-cycle (resp., 2-cycle) operations, the operands and opcode are pipeline signals of depth 1 (resp., depth 2) while the outputs still have depth 4.

A natural approach to proving theorems about conditional pipelines, such as the one above, would be to derive an actual pipeline $\mathcal{D}'$ from a given conditional

pipeline $\mathcal{D}$ such that the signals of $\mathcal{D}'$ coincide with the pipeline signals of $\mathcal{D}$, and such that the behaviors of these signals can be shown to be equivalent for input sequences that satisfy the constraints of $\mathcal{D}$. This would allow us to derive properties of $\mathcal{D}$ by examining the combinational circuit $\tilde{\mathcal{D}}'$ and applying Theorem 1. This approach, however, is not quite suitable for our needs.

The real purpose of a conditional pipeline, as illustrated by the merged adder, is to perform several operations that have different latencies but involve similar computations that may be executed on the same hardware. Thus, a single circuit is designed as an embodiment of several distinct conditional pipelines that perform different functions, each with its own sets of constraints and pipeline signals. Just as circuitry is shared by these pipelines, we would like to be able to prove theorems about this circuitry that could be shared in our analysis of the various pipeline behaviors. Thus, rather than construct and prove separate theorems about a different combinational circuit for each of several conditional pipelines that are comprised by a circuit $\mathcal{D}$, we would prefer to focus our proof efforts on the single combinational circuit $\tilde{\mathcal{D}}$. This requires that we find a way to lift results pertaining to $\tilde{\mathcal{D}}$ to results about $\mathcal{D}$, which motivates the following theorem.

Theorem 2 *Let $\mathcal{D}$ be a conditional pipeline under $\mathcal{A}$ with pipeline signals P and depth function δ. Let $\{\mathcal{I}_0, \mathcal{I}_1, \dots\}$ be a sequence of input valuations that satisfies $\mathcal{A}$, and let $\mathcal{I}$ be an input valuation such that*

(a) for all $s \in I(\mathcal{D}) \cap P$, $\mathcal{I}(s) = \mathcal{I}_{\delta(s)}(s)$, and

(b) for all $(s, c, d) \in \mathcal{A}$, $\mathcal{V}_{\bar{\mathcal{D}},\mathcal{I}}(s) = c$.

Let $\mathcal{R}_0$ be a register state, and let $\mathcal{R}_{k+1} = next_{\mathcal{D}}(\mathcal{I}_k, \mathcal{R}_k)$ for $k \in \mathbb{N}$. Then for all $s \in P$, $\mathcal{V}_{\mathcal{D},\mathcal{I}_{\delta(s)},\mathcal{R}_{\delta(s)}}(s) = \mathcal{V}_{\bar{\mathcal{D}},\mathcal{I}}(s)$.

Proof: First, we claim that for all $(s, c, d) \in \bar{\mathcal{A}}$, $\mathcal{V}_{\bar{\mathcal{D}},\mathcal{I}}(s) = c$. More generally: if expression E is forced to the value c at depth d under $\bar{\mathcal{A}}$, then $\mathcal{V}_{\bar{\mathcal{D}},\mathcal{I}}(s) = c$. But this is easy to show by induction on the appropriate lexicographic order, considering first d and then the sum of the numbers of supporters of signals occurring in E, by using hypothesis (b) and the definition of $\bar{\mathcal{A}}$.

The conclusion of the theorem is derived using a similar induction scheme, based on d and then the number of supporters of s. In the base case, $s \in I(\mathcal{D})$ and using hypothesis (a), we have

$$\mathcal{V}_{\mathcal{D},\mathcal{I}_{\delta(s)},\mathcal{R}_{\delta(s)}}(s) = \mathcal{I}_{\delta(s)}(s) = \mathcal{I}(s) = \mathcal{V}_{\bar{\mathcal{D}},\mathcal{I}}(s).$$

For the inductive case, let E be the expression for s and let

$$d = \begin{cases} \delta(s) & \text{if } s \in W(\mathcal{D}) \\ \delta(s) - 1 & \text{if } s \in R(\mathcal{D}). \end{cases}$$

By the inductive hypothesis, for all $s' \in P$ occurring in E with $\delta(s') = d$,

$$\mathcal{V}_{\mathcal{D},\mathcal{I}_d,\mathcal{R}_d}(s') = \mathcal{V}_{\bar{\mathcal{D}},\mathcal{I}}(s').$$

Now suppose that $(s', c, d) \in \bar{\mathcal{A}}$. Since $\{\mathcal{I}_0, \mathcal{I}_1, \dots\}$ satisfies $\bar{\mathcal{A}}$, $\mathcal{V}_{\mathcal{D},\mathcal{I}_d,\mathcal{R}_d}(s') = c$. As we have already noted, $\mathcal{V}_{\bar{\mathcal{D}},\mathcal{I}}(s') = c$ as well. Thus, since E is determined by $P \cap \delta^{-1}(d)$ under $\bar{\mathcal{A}}$ at depth d, and since we have just shown that valuations $\mathcal{V}_{\mathcal{D},\mathcal{I}_d,\mathcal{R}_d}$ and $\mathcal{V}_{\bar{\mathcal{D}},\mathcal{I}}$ satisfy the criteria for v and v' in the definition of *determined*,

$$\mathcal{V}_{\mathcal{D},\mathcal{I}_{\delta(s)},\mathcal{R}_{\delta(s)}}(s) = \mathcal{V}_{\mathcal{D},\mathcal{I}_d,\mathcal{R}_d}(E) = \mathcal{V}_{\bar{\mathcal{D}},\mathcal{I}}(E) = \mathcal{V}_{\bar{\mathcal{D}},\mathcal{I}}(s).\Box$$

5 Correctness of Pipelines

Theorem 2 above is the theoretical basis for our scheme for deriving ACL2 theorems pertaining to a conditional pipeline $\mathcal{D}$ from corresponding theorems about $\tilde{\mathcal{D}}$, which we describe in this section.

Suppose that $\mathcal{D}$ is a conditional pipeline under a constraint set $\mathcal{A}$ with pipeline signals P and depth function δ, where

$$\mathcal{A} = \{(s_1, c_1, d_1), \dots, (s_A, c_A, d_A)\}.$$

Again let

$$I(\mathcal{D}) = \{q_1, \dots, q_N\},$$

and assume that

$$I(\mathcal{D}) \cap P = \{q_1, \dots, q_k\},$$

and that the set of all signals in $I(\mathcal{D})$ that belong to $dom(\mathcal{A})$ is

$$\{q_{k+1}, \dots, q_m\},$$

where $1 \leq k \leq m \leq N$. Also, let

$$O(\mathcal{D}) \cap P = \{r_1, \dots, r_\ell\}.$$

As a matter of convenience, assume that δ has been extended to the inputs $q_{k+1}, \dots, q_m$ by defining $\delta(q_i)$ in each case to be a depth at which q_i is constrained by $\mathcal{A}$, i.e., $\delta(q_i) = d_j$, where $(s_j, c_j, d_j) \in \mathcal{A}$ and $s_j = q_i$. Note that there may be several possible choices of j, but the selection is of no consequence.

We are interested in deriving properties of $\mathcal{D}$ of the form

```
(implies (assumptions n)
         (result (q1 (+ δ(q1) n)) ... (qk (+ δ(qk) n))
                 (r1 (+ δ(r1) n)) ... (rℓ (+ δ(rℓ) n))))
```

such as the theorem `correctness-of-f3a-long-ops` of the preceding section. In general, `(assumptions n)` is a predicate constructed from applications of the input functions of the form `(`q_i `(+ `j `n))`, where $1 \leq i \leq m$ and $j \geq 0$.

Note that the input functions $q_1, \dots, q_N$ represent a sequence of input valuations $\{\mathcal{I}_0, \mathcal{I}_1, \dots\}$ for $\mathcal{D}$, given by

$$\mathcal{I}_i(q_j) = \texttt{(}q_j\texttt{(+ }i\texttt{ n))}.$$

Similarly, the corresponding constant functions in the "*" package represent an input valuation $\mathcal{I}$ for $\tilde{\mathcal{D}}$, defined by

$$\mathcal{I}(q_j) = \texttt{(*::}q_j\texttt{)}.$$

In order to establish the relevance of Theorem 2 in this context, we must be able to ensure that the sequence $\{\mathcal{I}_0, \mathcal{I}_1, \dots\}$ satisfies $\mathcal{A}$. Thus, we assume that we have proved the following lemma with the ACL2 prover:

```
(defthm constraint-lemma-1
  (implies (assumptions n)
           (and (equal (s1 (+ d1 n)) c1)
                (equal (s2 (+ d2 n)) c2)
                ...
                (equal (sA (+ dA n)) cA)))).
```

We further must be able to ensure that the valuation $\mathcal{I}$ satisfies the conditions (a) and (b) of the theorem. To this end, we introduce the following macro.

```
(defmacro bindings (n)
  `(and (equal (*::q1) (q1 (+ δ(q1) ,n)))
        (equal (*::q2) (q2 (+ δ(q2) ,n)))
        ...
        (equal (*::qm) (qm (+ δ(qm) ,n))))).
```

Now assume that we are also able to prove the following.

```
(defthm constraint-lemma-2
  (implies (and (assumptions n) (bindings n))
           (and (equal (*::s1) c1)
                (equal (*::s2) c2)
                ...
                (equal (*::sA) cA)))).
```

Then all of the hypotheses of Theorem 2 must hold, given `(assumptions n)` and `(bindings n)`. It follows that we should be able to prove this lemma:

```
(defthm pipeline-lemma
  (implies (and (assumptions n) (bindings n))
           (and (equal (r1 (+ δ(r1) n)) (*::r1))
                (equal (r2 (+ δ(r2) n)) (*::r2))
                ...
                (equal (rℓ (+ δ(rℓ) n)) (*::rℓ))))).
```

Before considering the proof of this lemma or its implications, let us examine `constraint-lemma-1` and `constraint-lemma-2` in the context of the merged adder. As illustrated by this example, if the constraint set $\mathcal{A}$ has been judiciously constructed, then the proofs of both of these lemmas should be trivial. For example, the conjunct of `constraint-lemma-1` corresponding to the constraint `(F3A_min_11,0,2)` is

```
(equal (f3a_min_11 (+ 2 n)) 0),
```

where

```
(defun f3a_min_11 (n)
 (logand (epc_f3a_en0_11 n)
         (logior (log= (epc_ex1_fpopcode0_11 n) 1492)
                 (logior (log= (epc_ex1_fpopcode0_11 n) 2050)
                         (log= (epc_ex1_fpopcode0_11 n) 2114)))))
```

is the definition generated for this signal. On the other hand, the definition of `assumptions` includes the conjunct

```
(or (equal (epc_f3a_en0_11 (+ 2 n)) 0)
    (not (f3a-short-op (epc_ex1_fpopcode0_11 (+ 2 n))))),
```

where `f3a-short-op` is defined by

```
(defun f3a-short-op (op)
  (member op '(... 1492 2050 2114 ...))).
```

Thus, when the definitions are expanded, the term `(equal (f3a_min_11 (+ 2 n)) 0)` is immediately rewritten to `T` under the hypothesis `(assumptions n)`.

Similarly, the corresponding conjunct of `constraint-lemma-2` is

```
(equal (*::f3a_min_11) 0).
```

The macro `bindings` in this case is given by

```
(defmacro bindings (n)
  `(and (equal (*::pipe0_fsrc1_11)
               (pipe0_fsrc1_11 ,n))
        (equal (*::pipe0_fsrc2_11)
               (pipe0_fsrc2_11 ,n))
        (equal (*::epc_ex1_fpopcode0_11)
               (epc_ex1_fpopcode0_11 ,n))
        (equal (*::rq_speccwrc_13)
               (rq_speccwrc_13 (+ 1 ,n)))
        ...
        (equal (*::epc_fpa_en0_11)
               (epc_fpa_en0_11 ,n))
        (equal (*::epc_fpa_wb0_14)
               (epc_fpa_wb0_14 (+ 3 ,n))))).
```

Note that this macro binds the pipeline inputs (at depths 0 and 1) as well as the constrained inputs (at depths 0 and 3). In this case, `assumptions` is taken to be the predicate `f3a-long-op-assumptions` defined earlier. Thus, the hypothesis `(and (assumptions n) (bindings n))` implies

```
(f3a-long-op (*::epc_ex1_fpopcode0_11)),
```

which in turn can be shown to imply

```
(equal (*::f3a_min_11) 0).
```

As noted above, once we have established both `constraint-lemma-1` and `constraint-lemma-2`, Theorem 2 guarantees the truth of `pipeline-lemma`. However, we would like to derive `pipeline-lemma` formally as a theorem of ACL2. Suppose for the moment that this has been accomplished. It remains for us to show how `pipeline-lemma` can be used to derive our stated goal from some analogous theorem pertaining to the `"*"` functions.

Suppose that for some predicate `spec` we are able to prove the following two lemmas:

```
(defthm spec-lemma-1
  (implies (assumptions n)
           (spec (q1 (+ δ(q1) n)) ... (qm (+ δ(qm) n))


(defthm spec-lemma-2
  (implies (spec (*::q1) ... (*::qm))
           (result (*::q1) ... (*::qk) (*::r1) ... (*::rℓ)))).
```

In our example, `spec-lemma-1` and `spec-lemma-2` were proved for the following predicate:

```
(defun spec (pipe0_fsrc1_11 ... epc_fpa_wb0_14)
  (and (equal epc_f3a_en0_11 1)
       (f3a-long-op epc_ex1_fpopcode0_11)
       (admissible-operand-p pipe0_fsrc1_11)
       (admissible-operand-p pipe0_fsrc2_11)
       (equal epc_fpa_en0_11 0)
       (equal epc_fpa_wb0_14 0))).
```

The next lemma can then be proved automatically by applying the rewrite rules `pipeline-lemma` and `spec-lemma-2`:

```
(defthm lemma-to-be-instantiated
  (implies (and (assumptions n)
                (bindings n)
                (spec (*::q1) ... (*::qm)))
           (result (*::q1) ... (*::qk)
                   (r1 (+ δ(r1) n)) ... (rℓ (+ δ(rℓ) n))))).
```

The desired theorem may now be derived by functional instantiation:

```
(defthm correctness-of-pipeline
  (implies (assumptions n)
           (result (q1 (+ δ(q1) n)) ... (qk (+ δ(qk) n))
                   (r1 (+ δ(r1) n)) ... (rℓ (+ δ(qℓ) n))))
  :hints
```

```
(("Goal" :use ((:functional-instance lemma-to-be-instantiated
                (*::q1 (lambda () (q1 (+ δ(q1) n))))
                (*::q2 (lambda () (q2 (+ δ(q2) n))))
                ...
                (*::qm (lambda () (qm (+ δ(qm) n)) ))))))).
```

In order to use the indicated functional instance of `lemma-to-be-instantiated`, the prover is obligated to establish the corresponding instances of the constraints on the $*::q_i$, namely the lemmas `bvecp*`q_i, for $i = 1, \ldots, m$. This amounts to proving the subgoals

```
(bvecp (qi(+ δ(qi) n))).
```

But these are simply instances of the corresponding rewrite rules `bvecp-`q_i. Thus, we have the functional instance

```
(implies (and (assumptions n)
              (and (equal (q1 (+ δ(q1) ,n)) (q1 (+ δ(q1) ,n)))
                   (equal (q2 (+ δ(q2) ,n)) (q2 (+ δ(q2) ,n)))
                   ...
                   (equal (qm (+ δ(qm) ,n)) (qm (+ δ(qm) ,n)))))
              (spec (q1 (+ δ(q1) n)) ... (qm (+ δ(qm) n))))
         (result (q1 (+ δ(q1) n)) ... (qk (+ δ(qk) n))
                 (r1 (+ δ(r1) n)) ... (rℓ (+ δ(qℓ) n))))),
```

which is automatically rewritten to the desired term by applying `spec-lemma-1`. Note that it is critical that `bindings` was defined as a macro rather than a function, in order for the functional instance of `(bindings n)` to be a conjunction of trivial equalities as included above.

We are left with the task of proving `pipeline-lemma`, which is precisely the function of our pipeline tool, as described in the next section.

6 The Pipeline Tool

Retaining the notation of Section 5, suppose we have a predicate `assumptions` pertaining to the signals of $\mathcal{D}$ for which we can prove both `constraint-lemma-1` and `constraint-lemma-2`. Assume that we are also able to prove the following:

```
(defthm n-positive-lemma
  (implies (assumptions n)
           (not (zp n))))
```

The goal of our pipeline tool is to generate a file of events culminating in `pipeline-lemma`, which may then be used to establish the correctness of $\mathcal{D}$ by the procedure described in Section 5.

The tool is based on a simple recursive rewriting procedure. This procedure requires the computation of the *size* $\lambda(t)$ of a term t that is generated by the translator, as a generalization of the *size* of a signal. For example,

$$\lambda(\texttt{bits}\ x\ i\ j) = i - j + 1,$$

$$\lambda(\texttt{comp1}\ x\ n) = n,$$

and if $\lambda(x) = \lambda(y) = k$, then

$$\lambda(\texttt{logior}\ x\ y) = \lambda(\texttt{logand}\ x\ y) = k.$$

Now, a term is rewritten in the context of a constraint set $\mathcal{C}$, relative to a given depth $d \in \mathbb{N}$, as follows. (If no case below applies, then the term is returned unchanged.)

(1) If the term is a signal s and there exists $(s, c, d) \in \mathcal{C}$, then the constant c is returned.

(2) Otherwise, assume that the term is a function call. Each of its arguments is first rewritten recursively. Then the following rules are applied.

(3) If all of its arguments are constants, then the term is evaluated and the result is returned.

(4) Suppose the term is `(logior x y)`. If either x or y is 0, then the other argument is returned; if x is the constant $2^{\lambda(y)} - 1$, then that constant is returned; if y is the constant $2^{\lambda(x)} - 1$, then that constant is returned.

(5) Suppose the term is `(logand x y)`. If either x or y is 0, then 0 is returned; if x is the constant $2^{\lambda(y)} - 1$, then y is returned; if y is the constant $2^{\lambda(x)} - 1$, then x is returned.

(6) Suppose the term is `(if x y z)`. If $x =$ `T`, then y is returned; if $x =$ `NIL`, then z is returned; if $y = z$, then y is returned.

The input required from the user consists of four components:

(1) the ordered set of signal definitions produced by the translator, with an indication of whether each signal is an input, a wire, or a register,

(2) the size $\lambda(s)$ of each signal s,

(3) the basic constraint set $\mathcal{A}$, and

(4) the depth $\delta(r_i)$ of each of the pipeline signals of interest, $r_1, \ldots, r_\ell$. (These are generally some of the outputs of $\mathcal{D}$.)

Three passes through the signal list are required. On the first pass, two sets are constructed. The first is a set $\mathcal{C}$ of constraints, generated by $\mathcal{A}$. We do not guarantee that $\mathcal{C} = \bar{\mathcal{A}}$, but in general, $\mathcal{A} \subseteq \mathcal{C} \subseteq \bar{\mathcal{A}}$, and $\mathcal{C}$ is a sufficiently close approximation to $\bar{\mathcal{A}}$ for our purpose. The other is a set $\mathcal{T}$ of triples (s, Q, d), such that

(a) s is a signal that does not occur in $\mathcal{C}$,

(b) $d \in \mathbb{N}$, and

(c) Q is a set of supporters of s such that the expression for s is determined by Q at depth d under $\mathcal{C}$ (and hence under $\bar{\mathcal{A}}$).

These two sets are constructed concurrently as all signals are examined in order beginning with inputs. $\mathcal{C}$ is initialized to $\mathcal{A}$ and $\mathcal{T}$ is initially empty. For each signal s, the following procedure is executed:

For each $d' \in \mathbb{N}$ for which there exists at least one constraint $(s', c', d') \in \mathcal{C}$ such that s' occurs in the term that the translator produced for s, that term is rewritten in the context of the current value of $\mathcal{C}$ relative to the depth d'. Thus, we may have several rewritten terms for s corresponding to different values of d'. For each of these, a single element will be inserted into either $\mathcal{C}$ or $\mathcal{T}$.

Let t be the rewritten term corresponding to d' and let

$$d = \begin{cases} d' & \text{if } s \in W(\mathcal{D}) \\ d'+1 & \text{if } s \in R(\mathcal{D}). \end{cases}$$

First suppose that t is a constant. In this case, if there already exists some $(s, c, d'') \in \mathcal{C}$, with $c \neq t$, then an error is signaled. Otherwise, (s, t, d) is added to $\mathcal{C}$. On the other hand, if t is not a constant and Q is the set of signals that occur in t, then (s, Q, d) is added to $\mathcal{T}$.

On the second pass, a set of pairs $\mathcal{P} \subset S(\mathcal{D}) \times \mathbb{N}$ is constructed, representing the pipeline signals P and depth function δ. Initially, we set

$$\mathcal{P} = \{(r_1, \delta(r_1)), \ldots, (r_\ell, \delta(r_\ell))\}.$$

The signals are then examined in reverse order, beginning with outputs and proceeding toward inputs, and processed as follows:

(1) If s does not occur in $\mathcal{P}$, then no action is taken.

(2) Assume $(s, d) \in \mathcal{P}$. If s occurs in $\mathcal{C}$, then an error is signaled. (Recall that our definition does not allow a pipeline signal to be constrained.) If there exists $(s, Q, d) \in \mathcal{T}$, then consider the set of signals Q; if not, then consider the set of all signals that occur in the definition of s. If any signal in this set occurs in $\mathcal{C}$, then an error is signaled. Otherwise, for each signal s' in the set, the pair (s', d') is added to $\mathcal{P}$, where

$$d' = \begin{cases} d & \text{if } s \in W(\mathcal{D}) \\ d-1 & \text{if } s \in R(\mathcal{D}). \end{cases}$$

In the event of successful termination of this procedure, the claim that our circuit $\mathcal{D}$ is a conditional pipeline under $\mathcal{A}$ has been established and P and δ have been derived.

Finally, on the third pass, an ACL2 event file is generated, loosely based on the proof of Theorem 2. The first event in the file is the definition of the macro

`bindings`. This involves nothing more than printing a line corresponding to each input signal q_i that occurs either in $\mathcal{A}$ or in P, as follows:

`(equal (*::`q_i`) (`q_i `(+ `$\delta(q_i)$ `,n))).`

(Recall that for $q_i \in dom(\mathcal{A})$, $\delta(q_i)$ is selected arbitrarily from the depths at which q_i is constrained.)

This is followed by a second definition, which is immediately disabled:

```
(defun assumptions-and-bindings (n)
  (and (assumptions n)
       (bindings n)))

(in-theory (disable assumptions-and-bindings))
```

The purpose of this definition is to allow us to prevent the needless expansion of `(bindings n)` when it appears in the hypothesis of a lemma.

The rest of the file consists of lemmas that are constructed from the sets $\mathcal{C}$ and P as the signal list is traversed in order, starting with inputs. Whenever a signal $s \in dom(\mathcal{C})$ is encountered, two or more lemmas are generated, the precise forms of which depend on whether s is an input or not. Suppose s is an input. Then for each $(s, c, d) \in \mathcal{C}$, we have the event

```
(defthm s-d-simp
  (implies (assumptions n)
           (equal (s (+ d n)) c))
  :hints (("Goal" :in-theory (enable assumptions)))).
```

These are followed by

```
(defthm s-*-simp
  (implies (assumptions-and-bindings n)
           (equal (*::s) c))
  :hints
  (("Goal" :in-theory (enable assumptions-and-bindings)))).
```

If s is any signal other than an input, then a similar set of events is generated, differing only in the `enable` hints:

```
(defthm s-d-simp
  (implies (assumptions n)
           (equal (s (+ d n)) c))
  :hints (("Goal" :in-theory (enable s))))
```

followed by

```
(defthm s-*-simp
  (implies (assumptions-and-bindings n)
           (equal (*::s) c))
  :hints (("Goal" :in-theory (enable *::s)))).
```

Whenever a signal $s \in P$ is encountered, a single lemma is generated:

```
(defthm s-pipe
  (implies (assumptions-and-bindings n)
           (equal (*::s) (s (+ δ(s) n))))
  :hints (("Goal" :in-theory (enable s *::s)))).
```

In general, this event file may be certified with minimal guidance from the user. It is necessary, however, that the ACL2 environment be initialized by loading both the `"ACL2"` and `"*"` models of $\mathcal{D}$ along with the floating-point library. The function `(assumptions n)` must then be appropriately defined and disabled.

Once this is done, the only events in the file that may require modification are the lemmas *s*-*d*-`simp` and *s*-`*`-`simp` that correspond to constraints belonging to the original constraint set $\mathcal{A}$. For example, in order to prove the lemma

```
(defthm f3a_min_11-simp
  (implies (f3a-long-assumptions n)
           (equal (f3a_min_11 (+ 2 n)) 0))
  :hints (("Goal" :in-theory (enable f3a_min_11))))
```

the prover must be able to show, according to the definition of `f3a_min_11`, that

```
(f3a-long-assumptions n)
```

and

```
(not (equal (epc_f3a_en0_11 (+ 2 n))) 0)
```

together imply that `(epc_ex1_fpopcode0_11 (+ 2 n))` is not 1492, 2050, or 2114. But this may be accomplished simply by extending the hint to enable the definitions of `f3a-long-assumptions` and `f3a-short-op`.

All other lemmas in the file, including the *s*-*d*-`simp` and *s*-`*`-`simp` lemmas that correspond to the constraints in the set-theoretic difference $\mathcal{C} - \mathcal{A}$, as well as the *s*-`pipe` lemmas corresponding to the pipeline signals, can be proved automatically with no user guidance. The reason for this is that in processing the statements of these lemmas, once the enabled definitions of s and `*::`s are expanded, the resulting terms are rewritten by ACL2 by following essentially the same procedure as that used by the pipeline tool.

For example, suppose that the file contains a lemma

```
(defthm s-d-simp
  (implies (assumptions n)
           (equal (s (+ d n)) c))
 :hints (("Goal" :in-theory (enable s)))).
```

based on a constraint $(s, d, c) \in \mathcal{C} - \mathcal{A}$. Let `(f` $a_1 \ldots a_k$`)` denote the term generated by the translator for s. The pipeline tool must have rewritten this

term to c at depth d', where

$$d' = \begin{cases} d & \text{if } s \in W(\mathcal{D}) \\ d-1 & \text{if } s \in R(\mathcal{D}). \end{cases}$$

But then the ACL2 rewriter, after expanding `(`s `(+` d `n))` to

`(f (`a_1 `(+` d' `n)) ... (`a_k `(+` d' `n)))`

(using `n-positive-lemma` in the case of a register), will likewise rewrite this term to c under the hypothesis `(assumptions n)`. In order to see this, we refer to the steps of the pipeline tool's rewriting procedure (page 18):

(1) If the tool rewrites some a_i to a constant c' at depth d', then $(a_i, c', d') \in \mathcal{C}$, and there must already be a lemma a_i-d'`-simp`, which ACL2 will invoke to rewrite `(`a_i `(+` d' `n))` to c'.

(2) In rewriting a function call, ACL2 first rewrites the arguments recursively.

(3) ACL2 rewrites a function call with constant arguments simply by evaluating it.

(4) The floating-point library includes appropriate rewrite rules for reducing `(logior` x y`)`, (a) when either argument is 0, and (b) when one argument is $2^k - 1$ and the other is a bit vector of length k.

(5) The library includes similar rules pertaining to `(logand` x y`)`.

(6) The ACL2 rewriter has built-in procedures for reducing `(if` x y z`)` when $x =$ `T`, $x =$ `NIL`, or $y = z$.

Similarly, it is clear that the accompanying lemma s`-*-simp` is proved by expanding `(*::`s`)` to `(f (*::`a_1`) ... (*::`a_k`))` and rewriting that term to c.

Finally, suppose the file contains an event

```
(defthm s-pipe
  (implies (assumptions-and-bindings n)
           (equal (*::s) (s (+ δ(s) n))))
  :hints (("Goal" :in-theory (enable s *::s))))
```

corresponding to some $(s, \delta) \in \mathcal{P}$. Let `(f` $a_1 \ldots a_k$`)` denote the term generated by the translator for s. We may assume that the pipeline tool rewrote this term, at depth

$$d' = \begin{cases} \delta(s) & \text{if } s \in W(\mathcal{D}) \\ \delta(s)-1 & \text{if } s \in R(\mathcal{D}), \end{cases}$$

to `(g` $a_1 \ldots a_j$`)`, where $j \leq k$ and $a_1, \ldots, a_j$ are pipeline signals. Then the two sides of the equation in s`-pipe` are expanded to

`(f (*::`a_1`) ... (*::`a_k`))`

and

$$\texttt{(f (}a_1\ \texttt{(+ }d'\ \texttt{n)) ... (}a_k\ \texttt{(+ }d'\ \texttt{n)))},$$

respectively, which are in turn rewritten to

$$\texttt{(g (*::}a_1\texttt{) ... (*::}a_j\texttt{))}$$

and

$$\texttt{(g (}a_1\ \texttt{(+ }d'\ \texttt{n)) ... (}a_j\ \texttt{(+ }d'\ \texttt{n))).}$$

The proof is completed by applying the lemmas a_i`-pipe`, rewriting `(*::`a_i`)` to `(`a_i `(+` d' `n))`, for $i = 1, \ldots, j$.

7 Conclusion

This work grew out of an effort to verify the behavior of a floating-point RTL module, namely the AMD Athlon processor merged adder, using ACL2. Our initial approach was to verify a combinational version of this module, derived by replacing register assignments with wire assignments. While it might have been reasonable to stop there, we were concerned about the possibility of interference between coexisting pipelines that may have been abstracted away by the reduction to a purely combinational design.

This concern led to the development of a theory of pipeline circuits, which we applied in our analysis of the adder in order to justify the abstraction. We wrote a prototype tool to perform checks for correct pipeline behavior under various sets of assumptions on the inputs corresponding to the distinct operations performed by a module. The tool proved its utility by exposing a bug in the adder, which was subsequently fixed. This experience reinforced the importance of preserving our goal of verifying the actual RTL, rather than a combinational abstraction of it. Thus, we extended the pipeline tool to generate a sequence of ACL2 events that provide support for a complete ACL2 proof of correctness of the original RTL.

This effort illustrates an important verification technique: the use of unverified code to generate provable lemmas. (Macros, of course, have been used through ACL2's history for such purposes, but here we are referring to the automatic generation of substantial certifiable books.) The automation of such lemmas proved valuable here, in support of reasoning about bit vectors (the lemmas `bvecp-`s, etc., of Section 3) and in a critical step in deriving the final correctness theorem (the lemma `pipeline-lemma` of Section 6). This illustration of the use of lemma generation, along with functional instantiation and other techniques, will, we hope, encourage other formal verification workers to reason about actual RTL hardware models, rather than limiting their efforts to higher-level abstractions.

References

[1] Moore, J, Lynch, T., and Kaufmann, M., "A Mechanically Checked Proof of the Correctness of the Kernel of the $AMD5_K86$ Floating Point Division Algorithm", *IEEE Transactions on Computers*, 47:9, September, 1998.

[2] Russinoff, D., "A Mechanically Checked Proof of IEEE Compliance of the AMD-K5 Floating Point Square Root Microcode", *Formal Methods in System Design* 14:1, January 1999. See URL `http://www.onr.com/user/russ/david/fsqrt.html`.

[3] Russinoff, D., "A Mechanically Checked Proof of IEEE Compliance of the AMD-K7 Floating Point Multiplication, Division, and Square Root Algorithms", *Journal of Computation and Mathematics* 1, London Math. Society, December 1998. See URL `http://www.onr.com/user/russ/david/k7-div-sqrt.html`.

[4] Russinoff, D. and Flatau, A., "RTL Verification: A Floating-Point Multiplier", in Kaufmann, M., Manolios, P., and Moore, J, eds., *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Press, 2000. See URL `http://www.onr.com/user/russ/david/acl2.html`.

[5] Russinoff, D., "An ACL2 Library of Floating-Point Arithmetic", 1999. See URL `http://www.cs.utexas.edu/users/moore/publications/others/-fp-README.html`.