Verification of Pipeline Circuits

Matt Kaufmann and David Russinoff
Advanced Micro Devices, Inc.

5900 E. Ben White Blvd., Austin, TX 78741
{matt.kaufmann,david.russinoff}@amd.com

October 31, 2000

INTRODUCTION

- **A GENERAL GOAL:** formal verification of RTL
- **A MORE SPECIFIC GOAL:** restrict to *conditional pipelines*
- **TODAY:** We'll focus on a toy example.
- For more rigor and details, see the proceedings paper and the supporting materials.

OUTLINE

- BACKGROUND
- EXAMPLE
- PIPELINE TOOL
- SUMMARY

BACKGROUND

SOME CLASSES OF CIRCUITS

PURELY COMBINATIONAL (see Figure 1)

- Even these can require considerable effort.

SIMPLE PIPELINE (see Figure 2)

- Each signal has a well-defined depth;
- See Russinoff/Flatau, last year's ACL2 Workshop, [1].

CONDITIONAL PIPELINE (see Figure 3)

- Really several overlapping circuits merged together;
- Each reduces to a simple pipeline under appropriate input assumptions.

A MERGED ADDER

The AMD Athlon™ processor has a single floating-point unit for doing not only addition but also other operations, for example, comparisons. This allows hardware to be shared.

But operations on this *merged adder* can have different latencies, so this unit is not a simple pipeline. Nevertheless, for any given operation there are corresponding input assumptions under which the circuit behaves like a simple pipeline.

CONDITIONAL PIPELINES

- **MOTIVATION:** Two operations should not terminate simultaneously.

- **NOTE:**
  IF `longop` is 1 at pipe stages 0 and 1,
  THEN `out` (depth 2) does not depend on
  `s01_muxed_flopped` (depth 1).

- **CONDITIONAL PIPELINE,** P: Value of signals in P depend only on signals of appropriate depth under a given constraint.

In our toy, P contains all signals except `longop` and `s01_muxed_flopped`.

EXAMPLE

- RTL
- COMPONENTS
- EXCERPTS OF MODELS
- THEOREM AND PROOF

TOY MODEL RTL

```
s01 = in[0] & in[1];
s23 = in[2] & in[3];

s01_muxed = (longop) ? s01 : ~s01;

s01_muxed_flopped <= s01_muxed;
s23_flopped       <= s23;

s0123 <= s01_muxed_flopped & s23_flopped;

longop_flopped <= longop;

out = (longop_flopped) ? s0123 : s01_muxed_flopped;
```

COMPONENTS OF EXAMPLE

- `packages.lisp`: Definitions of `"+"` and `"*"` packages.
- `books/rtl/lib3/top.lisp`: RTL functions (e.g., `bvecp`) from ACL2 distribution.
- Function `unknown` for initial flop values:

```
(encapsulate
 ((unknown (key size n) t))
 (local (defun unknown (key size n) ...))
 (defthm bvecp-unknown
   (bvecp (unknown key size n) size)
   ...))
```

and then a number of books:

- `model`: formal sequential model
- `exec`: formal `"+"` model (combinational)
- `declarations`:
  basic input assumptions for `"*"` model
- `constants`: formal `"*"` model
  (combinational; uses `"+"` model)
- `inputs`:
  timed input constraints for sequential model
- `pipe`: pipeline lemma
- `main`: top-level theorem

SEQUENTIAL MODEL (EXCERPTS)

```
(IN-PACKAGE "ACL2")

(ENCAPSULATE
  ((LONGOP (N) T) (IN (N) T))
  (LOCAL (DEFUN LONGOP (N)
              (DECLARE (IGNORE N))
              0))
  (LOCAL (DEFUN IN (N) (DECLARE (IGNORE N)) 0))
  (DEFTHM BVECP$LONGOP (BVECP (LONGOP N) 1) ... )
  (DEFTHM BVECP$IN (BVECP (IN N) 4) ...))
```

```
(DEFUN S23 (N)
       (LOGAND (BITN (IN N) 2)
               (BITN (IN N) 3)))

(DEFUN S0123 (N)
       (IF (ZP N)
           (UNKNOWN 'S0123 1 N)
           (LOGAND (S01_MUXED_FLOPPED (1- N))
                   (S23_FLOPPED (1- N)))))

(DEFUN OUT (N)
       (IF (EQL (LONGOP_FLOPPED N) 0)
           (S01_MUXED_FLOPPED N)
           (S0123 N)))
```

"+" MODEL (EXCERPTS)

```
(IN-PACKAGE "+")

(DEFUN S23 (IN)
       (LOGAND (BITN IN 2) (BITN IN 3)))

(DEFUN S0123 (S01_MUXED_FLOPPED S23_FLOPPED)
       (LOGAND S01_MUXED_FLOPPED S23_FLOPPED))

(DEFUN OUT
       (LONGOP_FLOPPED S01_MUXED_FLOPPED S0123)
       (IF (EQL LONGOP_FLOPPED 0)
           S01_MUXED_FLOPPED S0123))
```

"*" MODEL (EXCERPTS)

The "declarations" book, included below, is as follows.

```
(in-package "*")

(encapsulate ; Input declarations.
 ((in () t)
  (longop () t))
 (local (defun in () 0))
 (local (defun longop () 0))
 (defthm acl2::input-spec-thm
   (and (bvecp (in) 4)
        (bvecp (longop) 1))
   :rule-classes ()))
```

Now, excerpts from file "constants.lisp":

```
(IN-PACKAGE "*")

(INCLUDE-BOOK "exec")

(INCLUDE-BOOK "declarations")

(DEFUN S23 NIL (+::S23 (IN)))

(DEFUN S0123 NIL
       (+::S0123 (S01_MUXED_FLOPPED)
                 (S23_FLOPPED)))

(DEFUN OUT NIL
       (+::OUT (LONGOP_FLOPPED)
               (S01_MUXED_FLOPPED)
               (S0123)))
```

THEOREM AND PROOF

```
; The first definition is from inputs.lisp.
(defun input-assumptions (n)
  (and (not (zp n))
       (equal (longop n) 1)
       (equal (longop (+ 1 n)) 1)))

; Other events are from main.lisp.
(defun result (in out)
  (equal out
         (logand (logand (bitn in 0)
                         (bitn in 1))
                 (logand (bitn in 2)
                         (bitn in 3)))))
```

```
(defthm correctness-of-pipeline
  (implies (input-assumptions n)
           (result (in n)
                   (out (+ 2 n))))
  ...)

; We try to prove this using:

(defthm spec-lemma-1 ; should be trivial
  (implies (input-assumptions n)
           (spec (in n) (longop n))))

(defthm spec-lemma-2 ; heart of the matter
  (implies (spec (*::in) (*::longop))
           (result (*::in) (*:: out)))
  :hints ...)
```

The choice of function `spec` is made when the combinational proof is performed (`spec-lemma-2`). Here it is in the toy example.

```
(defun spec (in longop)
  (declare (ignore in))
  (equal longop 1))
```

This approach allows us to reuse proofs about various signals when doing other cases, e.g., when `longop` is 0.

Below is the lemma to be functionally instantiated, using the following functional substitution.

```
((*::in     (lambda () (in n)))
 (*::longop (lambda () (longop n))))

(defthm lemma-to-be-instantiated
  (implies (and (input-assumptions n)
                (equal (*::longop) (longop n))
                (equal (*::in) (in n))
                (spec (*::in) (*::longop)))
           (result (*::in)
                   (out (+ 2 n)))))
```

THE CATCH

The following *pipeline lemma* allows us to derive the preceding lemma from `spec-lemma-2`.

```
(implies (and (input-assumptions n)
              (equal (*::longop) (longop n))
              (equal (*::in) (in n)))
         (equal (out (+ 2 n)) (*::out)))
```

Such a theorem is easy to prove for simple pipelines.

Our example, however, is a conditional pipeline.

PIPELINE TOOL

TOOL EVOLUTION

- The tool was written to propagate constraints and compute depths, thus verifying that we have a conditional pipeline.

- It found a bug in the merged adder, which has since been corrected.

- We then modified the tool to write out appropriate lemmas.

RUNNING THE TOOL

```
(clear-pipeline "./" "toy.acl2" "pipe.lisp"
 ((out 2)) ; output depth
 ()        ; input depths (optional)
 (((longop) (1 0 1))) ; assumptions
 (0 1 2))
```

If we instead use `((longop) (1 0))`, i.e., we forget to specify that `longop` must be 1 at time 1, then the tool reports:

```
Signal S01_MUXED_FLOPPED:
  Existing depth of 2 conflicts with
  expected depth of 1.
```

SUMMARY

For a more rigorous presentation, see our paper.

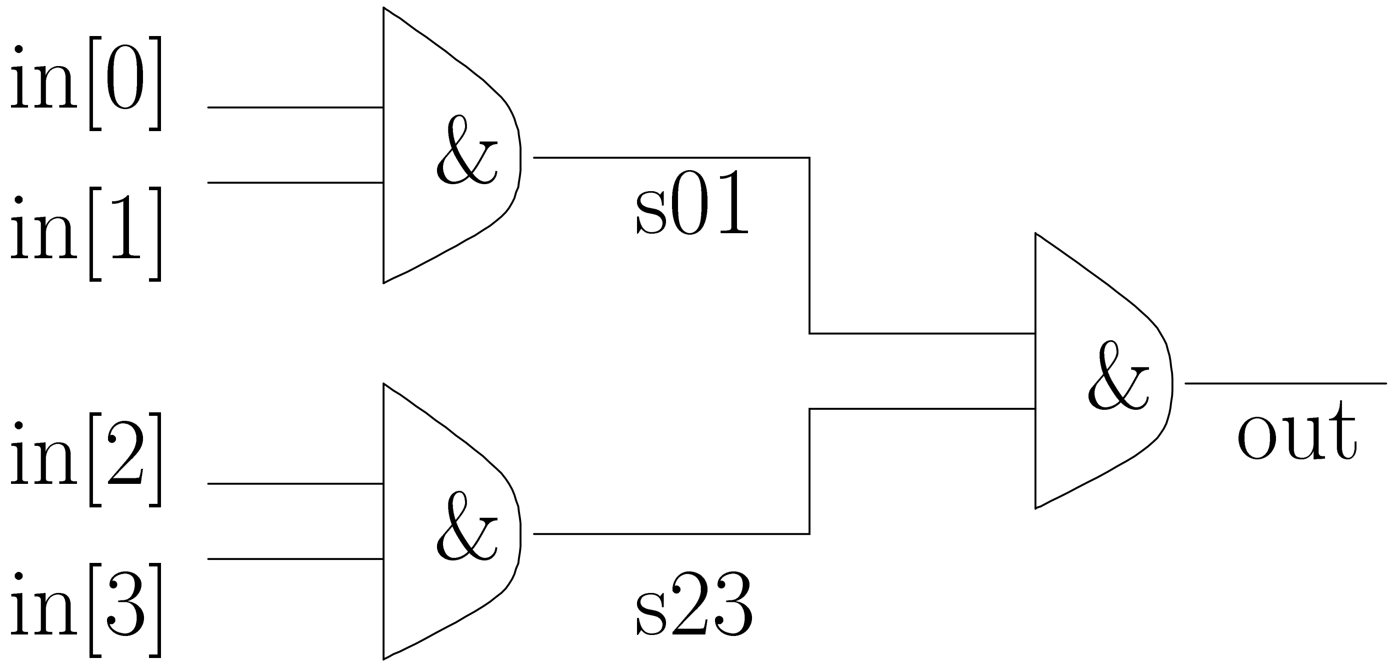
Originally, the pipeline tool was just used to find bugs. When it did so, we were motivated to modify it to produce lemmas of the sort we needed:

```
(implies (and (input-assumptions n)
              (equal (*::longop) (longop n))
              (equal (*::in) (in n)))
         (equal (out (+ 2 n)) (*::out)))
```

Formal verification of the model actually of interest need not be abandoned!

References

[1] 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`.



$$\mathrm{s01} = \mathrm{in}[0]\ \&\ \mathrm{in}[1];$$
$$\mathrm{s23} = \mathrm{in}[2]\ \&\ \mathrm{in}[3];$$

$$\mathrm{out} = \mathrm{s01}\ \&\ \mathrm{s23};$$

Figure 1: A Simple Combinational Circuit

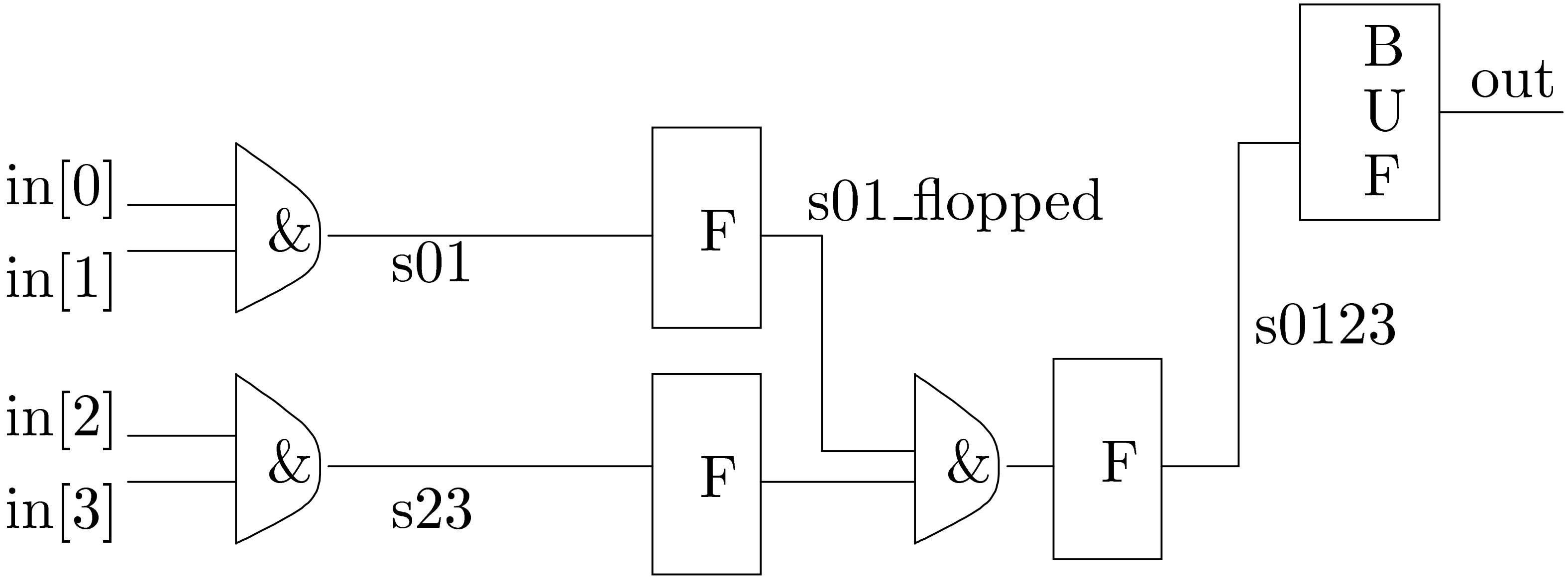


Figure 2: A Simple Pipeline

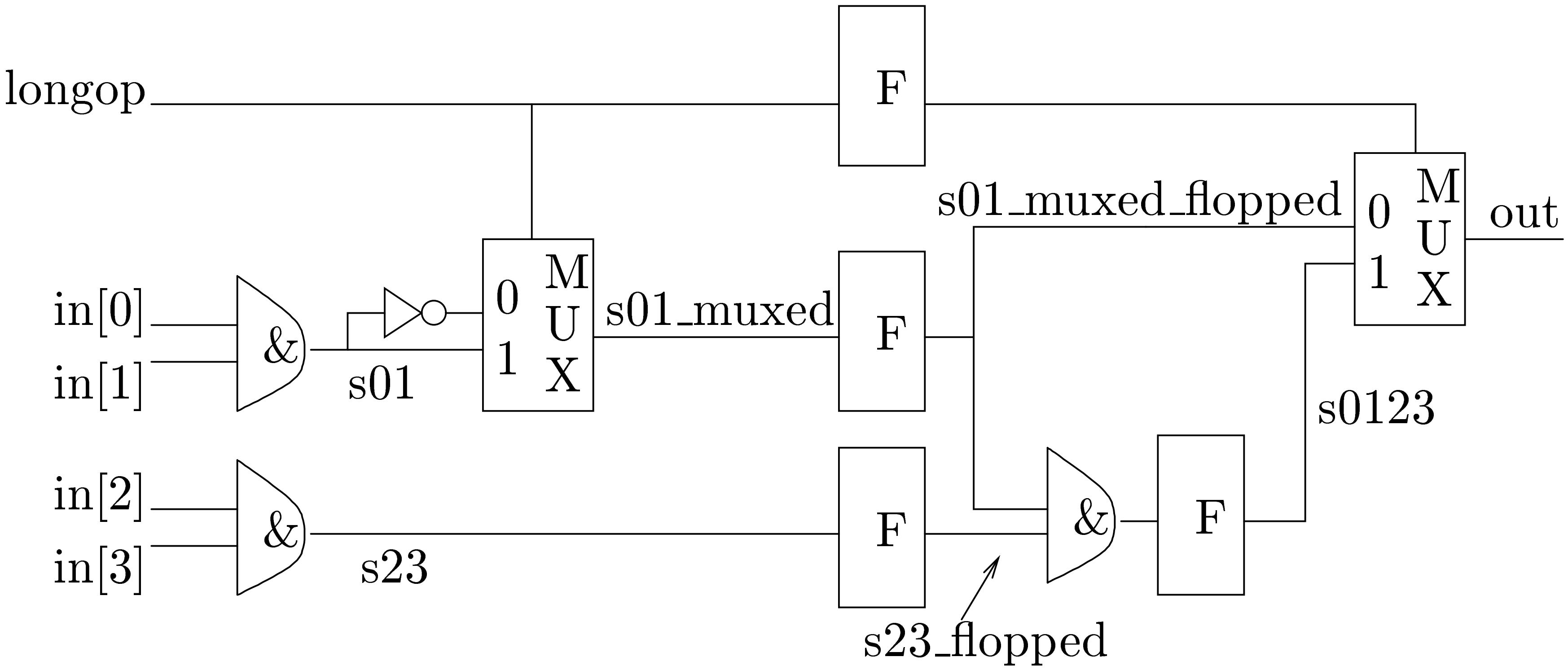


Figure 3: A Conditional Pipeline