Elementary Microarchitecture Algebra

John Matthews and John Launchbury

Oregon Graduate Institute,
P.O. Box 91000, Portland OR 97291-1000, USA
{johnm,jl}@cse.ogi.edu
http://www.cse.ogi.edu/PacSoft/Hawk

Abstract. We describe a set of remarkably simple algebraic laws governing microarchitectural components. We apply these laws to incrementally transform a pipeline containing forwarding, branch speculation and hazard detection so that all pipeline stages and forwarding logic are removed. The resulting unpipelined machine is much closer to the reference architecture, and presumably easier to verify.

1 Introduction

Transformational laws are well known in digital hardware, and form the basis of logic simplification and minimization, and of many retiming algorithms. Traditionally, these laws occur the gate level: de Morgan's law being a classic example. In this paper, we examine whether corresponding transformational laws hold at the microarchitectural level.

A priori, there is no reason to think that large microarchitectural components should satisfy any interesting algebraic laws, as they are constructed from thousands of individual gates. Boundary cases could easily remove any uniformity that has to exist for simple laws to be present. Yet we have found that when microarchitectural units are presented in a particular way, many powerful laws appear. Moreover, as we demonstrate in this paper, these laws *by themselves* are powerful enough to allow us to show equivalence of pipelined and non-pipelined microarchitectures.

We have used this algebraic approach to simplify a pipelined microarchitecture that uses forwarding, branch speculation and pipeline stalling for hazards. The resulting pipeline is very similar to the reference machine specification (i.e. no forwarding logic), while still retaining cycle-accurate behavior with the original implementation pipeline. The top-level transformation proof is simple enough to be carried out on paper, but we have mechanized enough of the theory in the Isabelle theorem prover [20] to have verified it semi-automatically, using Isabelle's powerful rewriting engine.

Interestingly, both circuits and laws can be expressed diagrammatically. A paper proof (transformation using equivalence laws) proceeds as a series of microarchitecture block diagrams, each an incrementally transformed version of the last. The laws often have a geometric flavor to them, such as laws to swap two components with each other, or laws to absorb one component into another. We find this diagrammatic approach an excellent way to communicate proofs.

For us, the most time-consuming part of this technique has been discovering the local behavior-preserving laws. It is our experience that these laws are much easier to discover when one uses the right level of abstraction. In particular, we encapsulate all control and dataflow information concerning a given instruction in the pipeline into an abstract data type called a *transaction* [1, 17]. We have found that not only do transactions reduce the size of microarchitecture specifications, they also provide enough "auxiliary" state information to make law-discovery practical.

The rest of the paper gives a brief introduction to our specification language, and then discusses many of the laws we have discovered. We then show their use by applying the laws in a proof of equivalence between two microarchitectures. While space constraints prohibit us from giving the complete proof, the top-level proof is sketched diagrammatically in [16].

2 Specifying a Pipelined Microarchitecture

We specify microarchitectures using the *Hawk* language [4, 17]. Hawk allows us to express modern microarchitectures clearly and concisely, to simulate the microarchitectures, either directly with concrete values, or symbolically, and provides a formal basis for reasoning about their behavior at source-code level. Currently Hawk is a set of libraries built on top of the pure functional language Haskell, which is strongly typed, supports first-class functions, and infinite data structures, such as streams [8, 21]. It is this legacy that led us to look for transformation laws in the first place: one often-cited benefit of purely functional programs is that they are amenable to verification through equational reasoning. We wanted to see if such algebraic techniques scaled up to microarchitectural verification.

2.1 Hawk Signals

Hawk is a purely declarative synchronous specification language, sharing a semantic base similar to Lustre[7]. The basic data structure underlying Hawk is the *signal*, which can be thought of as an infinite sequence of values, one per clock cycle, and circuits are pure functions from input signals to output signals. The elements of a signal must belong to the same type.

We use a notion of *transactions* to specify the immediate state of an entire instruction as it travels through the microprocessor [1]. A transaction is a record with fields containing the instruction's opcode, source register names and values, and the destination register name and its value, plus any additional information, like the speculative branch target PC for each branching instruction. A microarchitecture is a network of components, each of which processes signals of transactions.

Figure 1 shows the diagram of a simple one-stage microarchitecture, built out of transaction signal processors. Each component incrementally assigns values to various transaction fields, based on the component's internal state (if any) and the values of transaction fields assigned by earlier components. A textual Hawk specification of this circuit consists of set of mutually-recursive stream equations between the components. However, in this paper we will represent Hawk circuits as diagrams.

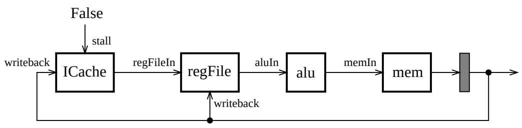

Fig. 1. One-stage pipeline.

For example, the `regFile` component has two transaction signal inputs and one transaction signal output. At a given clock cycle, the first input (called `regFileIn` in Figure 1) contains a transaction whose opcode and register name fields have been initialized, but whose value fields have all been zeroed out. The second input (called `writeback`) contains the completed transaction from the previous clock cycle. The `regFile` component first updates its internal register file state, based on the destination register name and value fields of the `writeback` input. It then fills in the source operand value fields of the `regFileIn` transaction based on the corresponding operand register names and the updated register file, and outputs the filled in transaction, all within the same clock cycle.

The `alu` component examines the opcode and source operand value fields of the transaction output by `regFile`. If the opcode is an ALU operation (which include branch instructions), the `alu` component computes the appropriate result, assigns the result to the destination operand value field of the transaction, and outputs the transaction along the `memIn` wire, again within the same (long) clock cycle. If the opcode is not an ALU operation, the `alu` component outputs the transaction unchanged.

The `mem` component behaves similarly for memory load and store operations. Like the `regFile` component, the `mem` component has internal state, representing the contents of data memory at each clock cycle. This state is updated and referenced based on the transactions sent to the `mem` component. Just as with the `alu` component, all memory and transaction updating occurs within the same clock cycle. The `mem` component sends the completed transaction to a `delay` component (represented in our diagrams as a shaded box), to make it available to the `ICache` and `regFile` components in the next clock cycle. These transactions also become the output of the entire microarchitecture, as is shown by the rightmost arrow. The initial value output by the `delay` component is the default transaction `nopTrans`, which represents an "inert" transaction which behaves like a NOP instruction, but does not affect the `ICache`'s program counter.

The `ICache` component produces new transactions, based on the value of the current program counter and the contents of program memory (the instruction-set architectures we consider have separate address spaces for instructions and data). Both the current PC and the instruction memory contents are internal to `ICache`. The ICache takes on its `writeback` input the completed transaction from the previous clock cycle. The ICache examines the transaction for branches that have been taken. When it finds such an instruction, it modifies its internal PC accordingly and starts fetching transactions from the branch target address. The ICache has as output a signal of transactions representing the newly-fetched instructions. Each transaction's source and destination operand values are initialized to zero, since the ICache doesn't know what values they should have. The other pipeline components will fill in these fields with their correct values. The ICache has a second input, called `stall`, which is a signal of Boolean values. On clock cycles where `stall` is asserted, the ICache will output the same transaction as it did on the previous clock cycle. In this simple microarchitecture, `stall` is always false. In more complex pipelines, the `stall` signal is typically asserted when the pipeline needs to stall due to a branch misprediction.

For more complex pipelines, we also allow the ICache to perform branch prediction, based on an internal branch target buffer. When performing branch prediction, the ICache will also annotate branch instruction transactions with the predicted branch target PC. A `branch_misp` component (not shown in Figure 1) can locally compare the predicted branch target with the actual branch target to determine if a branch misprediction has occurred.

3 Microarchitecture Laws

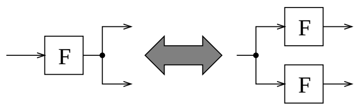

Fig. 2. Universal circuit-duplication law

With any algebraic reasoning there need to be some ground rules. We take as fundamental the notion of *referential transparency* or, in hardware terms, a *circuit duplication law*. Any circuit whose output is used in multiple places is equivalent to duplicating the circuit itself, and using each output once. This law is shown graphically in Figure 2. Because of the declarative nature of our specification language, every circuit satisfies this law. That is, it is impossible within Hawk for a specification of a component to cause hidden side-effects observable to any other component specification. In many specification languages this law does not hold universally. For example, duplicating a circuit that incremented a global variable on every clock cycle would cause the global variable to be incremented multiple times per clock period, breaking behavioral equivalence. Hawk circuits can still be stateful, but all stateful behavior must be local and/or expressed using feedback.

The next few sections introduce many other laws, some of which are specific to particular combinations of components, while others are quite widely applicable. Each instantiation of a law needs to be proved with respect to the specification of the circuit components involved. We have found induction and bisimulation to be the most useful ways of proving the laws in this paper, expressed as proofs in Isabelle.

3.1 Delay Laws

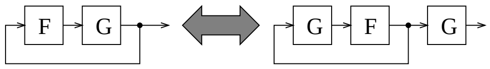

Fig. 3. feedback rotation law

The delay circuit is a fundamental building block of clocked circuits, especially when combined with feedback. A feedback variant of the circuit duplication law shown in Figure 3, called the *feedback rotation* law, allows circuits to be split along feedback wires. This law is not universal, but it is valid for any circuit that does not contain zero-delay cycles (amongst others). Happily, all of the laws we discuss, including the feedback rotation law itself, preserve a well-formedness property: if a circuit contains no zero-delay cycles, then any transformed circuit will also have no zero-delay cycles.

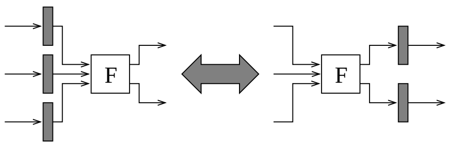

Fig. 4. time-invariance law.

The *time-invariance* law (Figure 4) is also nearly universal. A circuit is *time-invariant* if one can retime the circuit by removing the `delay`s from all the inputs of the circuit and placing new `delay`s on the circuit's outputs. Any combinatorial circuit that preserves default values is automatically time-invariant, but so are stateful circuits like the register file and memory cache. Interestingly, the ICache is not.

We use the above laws extensively to remove pipeline stages. If a pipeline stage is time-invariant, then we can move the pipeline registers (represented as `delay` circuits) from before the pipeline stage to afterwards. If subsequent pipeline stage are also time-invariant, then we can repeat the process, eventually moving all of the delay circuits to the end of the pipeline. However, forwarding logic between pipeline stages must still access the appropriate time-delayed outputs of later pipeline stages. The feedback-rotation law polices this, and ensures that the appropriate time-delay is kept by forcing `delay`s to be inserted on all feedback wires to the forwarding circuits.

3.2 Bypasses and Bypass Laws

The purpose of forwarding logic in a pipeline is to ensure that results computed in later pipeline stages are available to earlier pipeline stages in time to be used. Conceptually, the forwarding logic at each pipeline stage examines its current instruction's source operand register names to see if they match a later stage's destination operand register name. For every matching source operand, the operand value is replaced with the result value computed by the later pipeline stage. Non-matching source operands continue to use operand values given by the preceding pipeline stage.

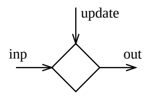

Fig. 5. bypass circuit

This conceptual logic can be implemented concisely using transactions. A *bypass* circuit (Figure 5) has two inputs, each a signal of transactions: The first input (`inp`) contains the transactions from the preceding pipeline stage. The second input (`update`) contains the transactions from a subsequent pipeline stage. The `bypass` circuit at each clock cycle compares the source operand names of the current `inp` transaction with the destination operand names of the current `update` transaction. The output of `bypass` is identical to `inp`, except that source operands matching `update`'s destination operand are updated. Bypasses arise frequently enough in pipeline specifications that we draw them specially, as diamonds with the `update` input connected to either the top or the bottom.

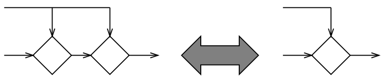

Fig. 6. bypass circuit idempotence law

Bypass circuits have many nice properties. Not only are they time-invariant and obey a kind of idempotence (Figure 6), but they also interact closely with register files and various execution units.

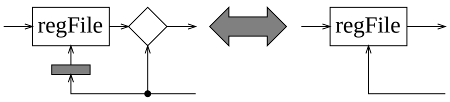

Fig. 7. register-bypass law

The fundamental interaction between a bypass and register file is shown in Figure 7. We call this the *register-bypass law*, and it is used repeatedly in eliminating forwarding logic when simplifying pipelines. The law states that we can delay writing a value into the register file, so long as we also forward the value to be written, in case that register was being read on the same clock cycle.

Initially we considered this law to be a theorem about register files, and accordingly we proved that it held for a number of different implementations. However, it is also tempting to view this law as an *axiom* of register files. In effect, by using the law repeatedly from right to left, we obtain a specification for how the register file must behave for any time prefix.

Hazard - Bypass Law Another bypass law permits the removal of bypasses between execution units. It is often the case that after retiming all delay circuits to the end of a pipeline, two execution units in a pipeline (such as an ALU unit and a Load/Store unit) are connected with one-cycle feedback loops. Each bypass circuit is forwarding the outputs of an execution unit to the inputs of that same execution unit, one clock cycle later.

If the upstream pipeline stages can guarantee that there is no hazard between successive transactions, then the double feedback is equivalent to the single feedback circuit shown at the bottom of Figure 8. This (conditional) identity is called the *hazard-bypass* law.

To be more concrete, suppose exec1 is the ALU and exec2 the memory cache. Then an ALU-mem hazard arises if a transaction which loads a register value from memory is immediately followed by an ALU operation which requires that register's value. Under these circumstances the two feedback loops would give different results. Under all other circumstances the two circuits are equivalent. We express this conditional equivalence using the `no_haz` component. It is an example of a projection component and is discussed in the next section.

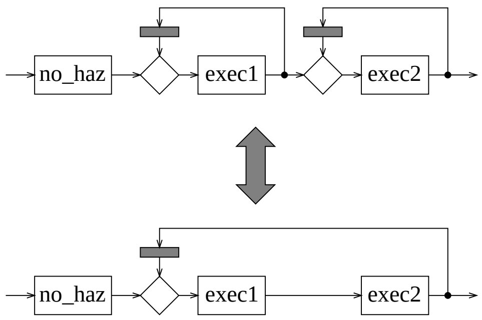

Fig. 8. hazard-bypass law

3.3 Projection Laws

Many laws, like the hazard-bypass law above, require that the input signals satisfy certain properties, and commonly, we may know that the output signal of a given component always satisfies a particular property. We can capture this knowledge of properties using signal *projections*.

A signal projection is a component with one input and one output. As long as the input signal satisfies the property of interest, the component acts like an identity function, returning the input signal unchanged. However, if the input does not satisfy the property we are interested in, the projection component modifies the input signal in some arbitrary way so that the property is satisfied.

Let us consider an example. For the hazard-bypass law we are interested in expressing the absence of ALU-mem hazards in a transaction signal. We reify this property as a `no_haz` projection. On each clock cycle, the `no_haz` component compares the current input transaction with the previous input transaction. If there is no ALU-mem hazard between the two transactions, then the current transaction is output unchanged. If a hazard does exist, then `no_haz` will instead output `nopTrans`, which is guaranteed not to generate a hazard (since `nopTrans` contains no source operands).

Where do projections come from? After all, they are not the sort of component that microarchitectural designers introduce just for fun.

Fig 9 provides an example of a law which "generates" a projection. The hazard-squashing logic guarantees that its output contains no hazards, and this is expressed in that the circuit is unchanged when the `no_haz` component is inserted on its output.

(The `hazard` component outputs a Boolean on each clock cycle stating whether its two input transactions constitute a hazard. The `kill` component takes a transaction signal and a Boolean signal as inputs. On each clock cycle, if the

Boolean input is false, then `kill` outputs its input transaction unchanged. If the Boolean input is true, then `kill` outputs a `nopTrans`, effectively "killing" the input transaction.)

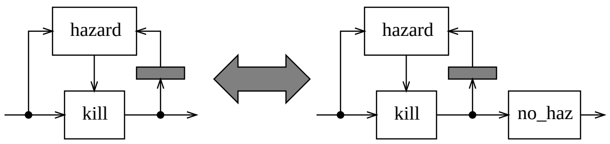

Fig. 9. Hazard-squashing logic guarantees no hazards

To be useful, a projection component needs to be able to migrate from a source circuit that produces it (such as the circuit in Figure 9) to a target circuit that needs the projection to enable an algebraic law (such as the hazard-bypass law). Thus a projection component must be able to commute with the intervening circuits between the source and the target circuit. Well-designed projections commute with many circuits. For instance, the `no_haz` projection commutes with `bypass`, `alu`, `mem`, and `regFile` components. It also commutes with `delay` components (that is, `no_haz` is time-invariant).

Projections are also convenient for expressing the fact that a component only uses some of the fields of an input transaction. For instance, the `hazard` component only looks at the opcode, source, and destination register name fields of its two input transactions. We can create a projection called `proj_ctrl` that sets every other field of a transaction to a default value, and prove a law stating that the `hazard` component is unchanged when `proj_ctrl` is added to any of its inputs. We can then show that `proj_ctrl` commutes with other components, such as bypasses and delays. This allows us to move the input wires to `hazard` across these other components, which is sometimes necessary to enable other laws. Similarly, the `proj_branch_info` projection allows us to move `ICache` and `branch_misp` component inputs.

4 Transforming the Microarchitecture

The laws we have been discussing can be used for aggressively restructuring microarchitectures while retaining equivalence. We have used them to simplify several pipelined microarchitectures with a view to verification. The example we present here contains three levels of forwarding logic, resolves hazards by stalling the pipeline, and performs branch speculation. The block diagram for this microarchitecture is shown in Figure 10.

By using just algebraic laws, we have been able to reduce most of the complexity, leaving essentially an unpipelined microarchitecture. We are currently implementing the algebraic laws as a rewrite system in Isabelle. For this paper we describe our top-level rewrite strategy informally.

Retiming We first remove all delay circuits from the main pipeline path. We accomplish this by repeatedly applying the time-invariance law, and by splitting delays along wires through the circuit duplication and feedback rotation laws.

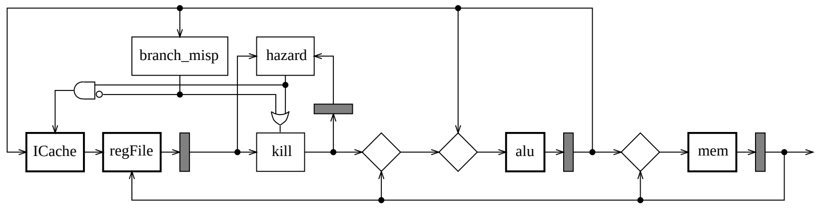

Fig. 10. Microarchitecture before simplification

Move control wires Next, we move all wires not directly involved with forwarding logic to either before or after all of the `bypass` circuits. This is to enable the hazard-bypass laws, which we apply in a later step. We move the wires by inserting projection circuits and using the corresponding projection-commutativity laws.

Propagate hazard information The hazard-bypass laws can only be applied when there are no hazards between the affected stages. So we generate a no-hazard projection at the end of the dispatch stage (which is justified by a projection-absorption law applicable to the kill-circuit complex in that stage), and then move it between the first and second bypass circuits. We also use additional properties of the `proj_ctrl`, `kill`, and `regFile` circuits (discussed in [16]) to swap the hazard/kill complex with the register file, so that the register-bypass law can be used more readily in the next step of the simplification. The circuit in Figure 11 shows the microarchitecture after this step has been completed. Notice that the ALU and memory units are now connected exactly as required for an application of the hazard-bypass law.

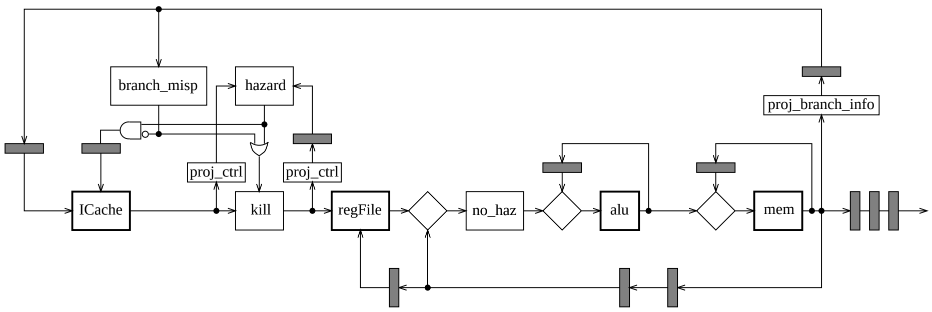

Fig. 11. Microarchitecture after the "propagate hazard information" step

Remove forwarding logic We can now apply the hazard-bypass law to remove the bypass circuit just prior to the memory unit. We eliminate the other two bypass circuits by applying the register-bypass law twice.

Cleanup The pipeline has now been simplified as much as possible, except that there are still some extra delay components as well as several unnecessary projection circuits. We merge delay components, then move the projection circuits back to their places of origin and remove them using the projection laws in the opposite direction.

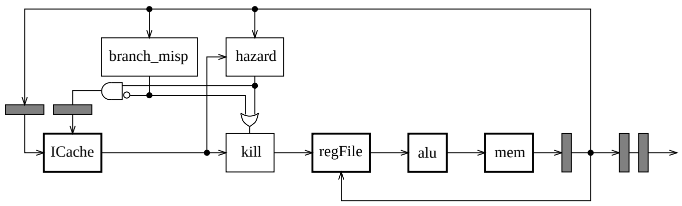

Fig. 12. Microarchitecture after simplification

The final microarchitecture is shown in Figure 12. This circuit still outputs exactly the same transaction values, cycle-for-cycle, as the microarchitecture in Figure 10, but is considerably less complex. We can now apply conventional techniques to verify that this microarchitecture is a valid implementation of the ISA.

5 Discussion

5.1 Related work

Hawk is built on top of the pure functional language Haskell, where algebraic techniques for transforming functional programs are routinely used for equivalence checking and verification [2, 3, 13] and for compilation and optimization [5, 12]. Much of our work can be seen as an extension of these ideas. Hawk itself is very similar in flavor to Lustre [6] except that in Lustre signals are accompanied by additional clock information. The Hawk specification style follows from the work of Johnson[9], O'Donnell[18], and Sheeran[25].

We have also been influenced by the algebraic techniques used in the relational hardware-description language Ruby [24]. Sizeable Ruby circuits have been successfully derived and verified through algebraic manipulation [10, 11]. What distinguishes our work is our focus on microarchitectural units as objects of study in their own right. The Ruby research has emphasized circuits at the gate level.

In terms of verification, our approach is most similar to two known techniques, called retiming [14, 23, 26] and unpipelining [15]. A circuit is *retimed* when the delay components of the circuit are repositioned, while the functional components are left unchanged, effectively through repeated applications of the time-invariance law. Typically, circuits are retimed to reduce the clock cycle time. In contrast, we retime circuits as part of a simplification process. In fact, we often use the time invariance law to increase cycle time!

Unpipelining [15] is a verification technique where a pipelined microarchitecture, specified as a state machine, is incrementally transformed into a functionally-equivalent unpipelined microarchitecture. Unpipelining proceeds by repeatedly merging the last stage of a pipeline into the next to last stage, producing a microarchitecture with one less stage on each iteration. On each iteration, the two microarchitectures are proven equivalent by induction over time. This is similar to our approach, except that we use transactions to encapsulate and reuse many of the verification steps, and we only need to prove the equivalence of the portion of the microarchitecture being transformed, rather than the entire microarchitecture, on each iteration. On the other hand, Levitt and Olukotun's implementation of unpipelining is much more automated than our work up to now.

Transactions were a key concept in allowing us to discover and formulate many of the algebraic laws of microarchitectural components. Unsurprisingly, the usefulness of transactions has been noticed before. Aagaard and Leeser used transactions to specify and verify hierarchical networks of pipelines [1], and Önder and Gupta have used a similar concept of *instruction contexts* as a core datatype in UPFAST, an imperative microarchitecture simulation language [19]. Further, Sawada and Hunt use an extended form of transactions in their verification of a speculative out-of-order microarchitecture [22]. Each transaction records two snapshots of the entire ISA state, before and after the instruction is executed. In their work, however, transactions are not part of the microarchitecture itself, but are constructed separately for verification purposes.

5.2 Next steps in microarchitecture algebra

As we have come to see it, the main principle of applying algebraic techniques to microarchitectures is to use geometric reasoning to move and absorb circuits, and to express that reasoning as local equalities whenever possible. Conditional equalities can be expressed using projections.

Some care is required in the definition of basic components. We have striven to design the component circuits to satisfy as rich a variety of algebraic laws as possible, such as preserving default values, satisfying time-invariance, and so on. Sometimes we hit on the correct definitions immediately, but more commonly adapted the definitions over time admitting more and more laws. One example of this is in pipeline registers. Initially, we used conditional delays to act as pipeline registers, but since then have found it useful to separate clocked behavior from functional behavior, enabling the two dimensions to be manipulated separately.

In some sense the components we now manipulate are not optimal in terms of transistor counts. In particular, many units receive and propagate information they are not interested in. However, much of this overhead can be removed automatically through a similar set of rewrite laws built around more primitive components than those presented in this paper. We plan to write this up in a subsequent paper.

6 Acknowledgements

We wish to thank Borislav Agapiev, Carl Seger, Byron Cook, Sava Krstic, and Thomas Nordin for their valuable contributions to this research. The authors are supported by Intel Strategic CAD Labs and Air Force Material Command (F19628-93-C-0069). John Matthews receives support from a graduate research fellowship with the NSF.

References

1. AAGAARD, M., AND LEESER, M. Reasoning about pipelines with structural hazards. In *Second International Conference on Theorem Provers in Circuit Design* (Bad Herrenalb, Germany, Sept. 1994).
2. BIRD, R., AND WADLER, P. *Introduction to Functional Programming.* Prentice Hall International Series in Computer Science. Prentice Hall, 1988.
3. BIRD, R. S., AND MOOR, O. D. *Algebra of Programming.* Prentice Hall, 1996.
4. COOK, B., LAUNCHBURY, J., AND MATTHEWS, J. Specifying superscalar microprocessors in Hawk. In *FTH'98, Workshop on Formal Techniques for Hardware and Hardware-like Systems* (Marstrand, Sweden, June 1998).
5. GILL, A., LAUNCHBURY, J., AND PEYTON JONES, S. L. A Short Cut to Deforestation. In *FPCA'93, Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark, June 1993), ACM Press, pp. 223–232.
6. HALBWACHS, N. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.
7. HALBWACHS, N., LAGNIER, F., AND RATEL, C. Programming and verifying realtime systems by means of the synchronous data-flow programming language Lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems* (September 1992).
8. HUDAK, P., PETERSON, J., AND FASEL, J. A gentle introduction to Haskell. Available at `www.haskell.org`, Dec. 1997.
9. JOHNSON, S. D. *Synthesis of Digital Systems from Recursive Equations.* ACM Distinguished Dissertation Series. MIT Press, 1984.
10. JONES, G., AND SHEERAN, M. Collecting butterflies. Tech. rep., Oxford University Computing Laboratory, 1991.
11. JONES, G., AND SHEERAN, M. Designing arithmetic circuits by refinement in ruby. In *Mathematics of Program Construction* (1993), vol. 669 of *LNCS*, Springer Verlag.
12. JONES, S. L. P., AND SANTOS, A. L. M. A transformation-based optimiser for Haskell. *Science of Computer Programming 32*, 1–3 (Sept. 1998), 3–47.
13. LAUNCHBURY, J. Graph algorithms with a functional flavour. *Lecture Notes in Computer Science 925* (1995).
14. LEISERSON, C. E., AND SAXE, J. B. Retiming synchronous circuitry. *Algorithmica 6* (1991), 5–35.

15. Levitt, J., and Olukotun, K. A scalable formal verification methodology for pipelined microprocessors. In *33rd Design Automation Conference (DAC'96)* (New York, June 1996), Association for Computing Machinery, pp. 558–563.
16. Matthews, J., and Launchbury, J. Elementary microarchitecture algebra: Top-level proof of pipelined microarchitecture. Tech. Rep. CSE-99-002, Oregon Graduate Institute, Computer Science Department, Portland, Oregon, Mar. 1999.
17. Matthews, J., Launchbury, J., and Cook, B. Specifying microprocessors in Hawk. In *IEEE International Conference on Computer Languages* (Chicago, Illinois, May 1998), pp. 90–101.
18. O'Donnell, J. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Symposium on Functional Programming Languages in Education* (July 1995).
19. Önder, S., and Gupta, R. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages* (Chicago, Illinois, May 1998), pp. 80–89.
20. Paulson, L. *Isabelle: A Generic Theorem Prover.* Springer-Verlag, 1994.
21. Peterson, J., et al. Report on the programming language Haskell: A non-strict, purely functional language, version 1.4. Available at `www.haskell.org`, Apr. 1997.
22. Sawada, J., and Hunt, W. A. Processor verification with precise exceptions and speculative execution. *Lecture Notes in Computer Science 1427* (1998), 135–146.
23. Saxe, J., and Garland, S. Using Transformations and Verifications in Circuit Design. *Formal Methods in System Design 4*, 1 (1994), 181–210.
24. Sharp, R., and Rasmussen, O. An introduction to Ruby. Teaching Notes ID–U: 1995-80, Dept. of Computer Science, Technical University of Denmark, October 1995.
25. Sheeran, M. *μFP, an Algebraic VLSI Design Language.* PhD thesis, Programming Research Group, University of Oxford, 1983.
26. Sheeran, M. Retiming and slowdown in Ruby. In *The Fusion of Hardware Design and Verification* (Glasgow, Scotland, July 1988), G.J. Milne, Ed., IFIP WG 10.2, North-Holland, pp. 289–308.