Microprocessor Specification in Hawk*

John Matthews johnm@cse.ogi.edu
Byron Cook byron@cse.ogi.edu
John Launchbury jl@cse.ogi.edu

Abstract

Modern microprocessors require an immense investment of time and effort to create and verify, from the high-level architectural design downwards. We are exploring ways to increase the productivity of design engineers by creating a domain-specific language for specifying and simulating processor architectures. We believe that the structuring principles used in modern functional programming languages, such as static typing, parametric polymorphism, first-class functions, and lazy evaluation provide a good formalism for such a domain-specific language, and have made initial progress by creating a library on top of the functional language Haskell. We have specified the integer subset of an out-of-order, superscalar DLX microprocessor, with register-renaming, a reorder buffer, a global reservation station, multiple execution units, and speculative branch execution. Two key abstractions of this library are the signal abstract data type (ADT), which models the simulation history of a wire, and the transaction ADT, which models the state of an entire instruction as it travels through the microprocessor.

1 Introduction

Modern microprocessor technologies have substantially increased processor performance. For example, *pipelining* allows a processor to overlap the execution of several instructions at once. With *superscalar* execution, multiple instructions are read per clock cycle. *Out-of-order execution*, where some instructions that logically come after a given instruction may be executed before the given instruction, can also greatly increase processor speed [6]. All of these technologies dramatically increase design complexity. In fact, creating and verifying these designs is a significant proportion of the total microprocessor development lifecycle. As the number of possible gates in future microprocessors increases exponentially, so too does design complexity.

At OGI, we have developed the *Hawk* language for building executable specifications of microprocessors, concentrating on the level of micro-architecture. In the long term we plan for Hawk to be a standalone language. In the meantime we have embedded our language into Haskell, a strongly-typed functional language with lazy (demand-driven) evaluation, first-class functions, and parametric polymorphism [5] [12].

The library makes essential use of these features. As an example, we have used Hawk to specify and simulate the integer portion of a pipelined DLX microprocessor[4]. The DLX is a complete microprocessor and is a widely used model among researchers. Several DLX simulators exist, as well as a version of the Gnu C compiler that generates DLX assembly instructions. The processor includes the most common instructions found in commercial RISC processors. Our specification, including data and control hazard resolution, is only two pages of Hawk code. A non-pipelined version of the processor was specified in half of a page.

In this report, we introduce the concepts behind Hawk. Rather than attempting a detailed explanation of the whole of the DLX with all of its inherent complexity, we have chosen to exhibit the techniques on a considerably simplified model. A corresponding annotated specification of the DLX itself can be found in [13].

2 The Hawk Library

We start with a simple example that introduces several functions used in later examples. Consider the resettable counter circuit of Figure 1.

The *reset* wire is Boolean valued, while the other

*Copyright 1998 IEEE. Published in the Proceedings of ICCL'98, May 1998 Chicago, Illinois. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

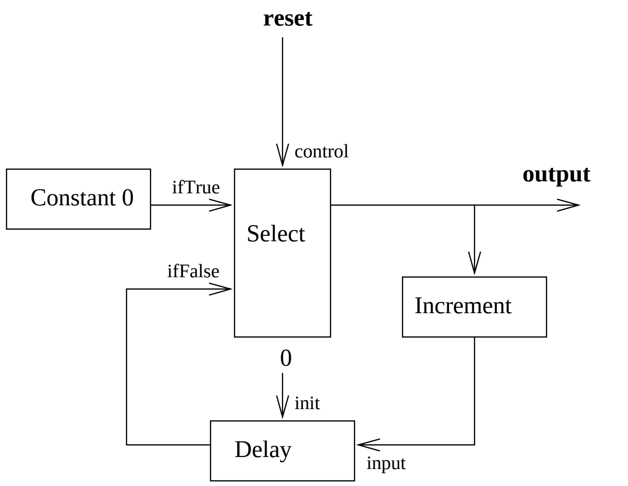

Figure 1: **Resettable Counter. A simple circuit that counts the number of clock cycles between reset signals.**

wires are integer valued. Of course, in silicon, integer-valued wires are represented by a vector of Boolean wires, but as a design abstraction, a Hawk user may choose to use a single wire. The circuit counts (and outputs) the number of clock cycles since *reset* was last asserted.

2.1 Signals

Notice that there is no explicit clock in the diagram. Rather, each wire in the diagram carries a *signal* (integer or boolean valued) which is an implicitly clocked value. The output of a circuit only changes between clock cycles. We build signals using an abstract type constructor called `Signal`. As a mental model we could think of a value of type `Signal a` as a function from integers to values of type `a`.

```
type Signal a = (Int -> a)
```

The integers denote the current time, measured as the number of clock cycles since the start of the simulation. Circuits and components of circuits are represented as functions from signals to signals. This view of signals is used extensively in the hardware verification community [9] [14]. Equivalently, we can think of signals as infinite sequences of values.

In the resettable counter example above, the *constant 0* circuit outputs zero on every clock cycle. The *select* component chooses between its inputs on each clock cycle depending on the value of *reset*. If *reset* is asserted on a given cycle (has value *true*), then the output is equal to *select*'s top input, in this case zero. If *reset* is not asserted, then its output is the value of its bottom input. In either case, *select*'s output is the output of the entire circuit, as well as the input to the *increment* component, which simply adds 1 to its input. The output of *increment* is fed into the *delay* component. A delay component outputs whatever was on its input in the previous clock cycle: it "delays" its input by one cycle. However, on the first clock cycle of the simulation there is no previous input, so on the first cycle *delay* outputs whatever is on its *init* input, which is zero in this circuit.

2.2 Components

The components used in the resettable counter are trivial examples of the sorts of things provided by the Hawk library, but let's look at a specification of each component in turn.

The simplest component is `constant`

```
constant :: a -> Signal a
```

The constant function takes an input of any type `a`, and returns an output of type `Signal a`, that is, a sequence of values of type `a`. For every clock cycle, `(constant x)` always has the same value `x`.

The next component is `select`:

```
select :: Signal Bool ->
          Signal a ->
          Signal a ->
          Signal a
```

This declares `select` to be a function. In a Hawk declaration, anything to the left of an arrow is a function argument. Thus, the expression `(select bs xs ys)`, where `bs` is a Boolean signal, and `xs` and `ys` are signals of type `a`, will return an output signal of type `a`. The values of the output signal are drawn from `xs` and `ys`, decided each clock tick by the corresponding value of `bs`. For example, if

```
bs = <True,False,True,False,...>,
xs = <x1,x2,x3,x4,...>,
ys = <y1,y2,y3,y4,...>
```

then `(select bs xs ys)` is equal to the signal `<x1,y2,x3,y4,...>`.

Hawk treats functions as first-class values, allowing them to be passed as arguments to other functions or returned as results. First-class functions allow us to specify a generic `lift` primitive, which "lifts" a normal function from type `a` to type `b` into a function over the corresponding signal types:

```
lift :: (a -> b) -> Signal a -> Signal b
```

The expression `(lift f xs)`, where `xs = <x1,x2,x3,...>`, is equal to the signal `<f x1, f x2, f x3, ...>`.

The `increment` component is defined in terms of `lift`:

```
increment :: Signal Int -> Signal Int
increment xs = lift (+ 1) xs
```

Given the `xs` input signal, `increment` adds one to each component of `xs` and returns the result.

The `delay` component is more interesting:

```
delay :: a -> Signal a -> Signal a
```

This function takes an initial value of type `a`, and an input signal of type `Signal a`, and returns a value of type `Signal a` (the input arguments are in reverse order from the diagram). At clock cycle zero, the expression `(delay initVal xs)` returns `initVal`. Otherwise the expression returns whatever value `xs` had at the previous clock cycle. This function can thus propagate values from one clock cycle to the next. Note that `delay` is polymorphic, and can be used to delay signals of any type.

2.3 Using the components

Once we have defined primitive signal components like the ones above, we can define the resettable counter:

```
resetCounter :: Signal Bool -> Signal Int
resetCounter reset = output
  where
    output =
      select reset
             (constant 0)
             (delay 0 (increment output))
```

The `resetCounter` definition takes `reset` as a Boolean signal, and returns an integer signal. The `reset` signal is passed into `select`. On every clock cycle where `reset` returns `True`, `select` outputs `0`, otherwise it outputs the result of the `delay` function. On the first clock cycle `delay` outputs `0`, and thereafter outputs the result of whatever `(increment output)` was on the previous clock cycle. The output of the whole circuit is the output of the `select` function, here called `output`. Notice that `output` is used twice in this function: once as the input to `increment`, and once as the result of the entire function. This corresponds to the fact that the `output` wire in Figure 1 is split and used in two places. Whenever a wire is duplicated in this fashion, we must use a `where` statement in Hawk to name the wire.

2.4 Recursive Definitions

There is something else curious about the `output` variable. It is being used recursively in the same place it is being defined! Most languages only allow such recursion for functions with explicit arguments. In Hawk, one can also define recursive data-structures and functions with implicit arguments, such as the one above.

If we didn't have this ability, we would have had to define `resetCounter` as follows:

```
resetCounter reset = output
  where
    output time =
      (select reset
              (constant 0)
              (delay 0 (increment output))) time
```

Every time we have a cycle in a circuit, we have to create a local recursive function, passing an explicit time parameter. This breaks the abstraction of the `Signal` ADT. In fact, in the real implementation of signals, we don't use functions at all. We use infinite lists instead. Each element of the list corresponds to a value at a particular clock cycle; the first list element corresponds to the first clock cycle, the second element to the second clock cycle, and so on. By storing signals as lazy lists, we compute a signal value at a given clock cycle only once, no matter how many times it is subsequently accessed.

Haskell allows recursive definitions of abstract data structures because it is a lazy language, that is, it only computes a part of a data structure when some client code demands its value. It is lazy evaluation that allows Haskell to simulate infinite data structures, such as infinite lists.

3 A Simple Microprocessor

As we noted in the introduction, the DLX architecture is too complex to explain in fine detail in an introductory report. Thus for pedagogical purposes we show how to use similar techniques to specify a simple microprocessor called SHAM (Simple HAwk Microprocessor). We begin with the simplest possible SHAM architecture (unpipelined), and then add features: pipelining, and a memory-cache.

The unpipelined SHAM diagram is shown in Figure 2. The microprocessor consists of an ALU and a register file. The ALU recognizes three operations: `ADD`, `SUB`, and `INC`. The ADD and SUB operations add and subtract, respectively, the contents of the two ALU inputs. The `INC` operation causes the ALU to increment its first input by one and output the result.

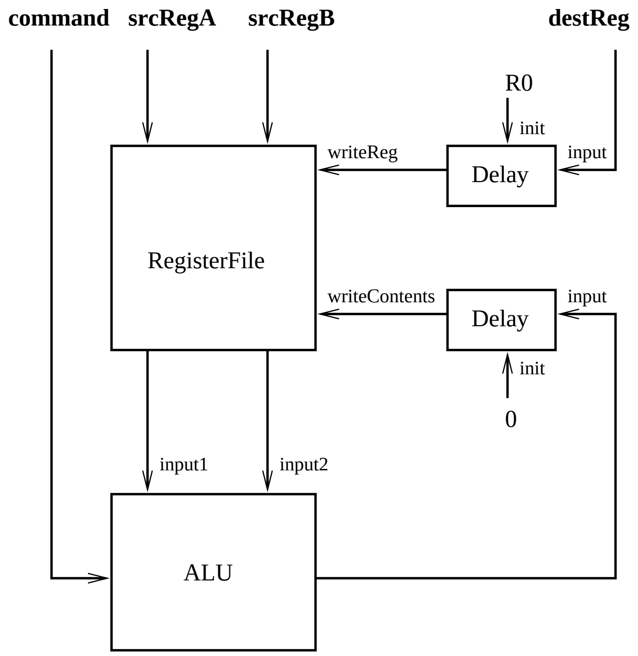

Figure 2: **Unpipelined version of SHAM.**

The register file contains eight integer registers, numbered `R0` through `R7`. Register `R0` is hardwired to the value zero, so writes to `R0` have no effect. The register file has one write-port and two read-ports. The write-port is a pair of wires; the register to update, called *writeReg*, and the value being written, called *writeContents*. The input to each read-port is a wire carrying a register name. The contents of the named read-port registers are output every cycle along the wires *contentsA* and *contentsB*. If a register is written to and read from during the same clock cycle, the newly written value is reflected in the read-port's output. This is consistent with the behavior of most modern microprocessor register files.

SHAM instructions are provided externally; in our drive for simplicity there is no notion of a program counter. Each instruction consists of an ALU operation, the destination register name, and the two source register names. For each instruction the contents of the two source registers are loaded into the ALU's inputs, and the ALU's result is written back into the destination register.

3.1 Unpipelined SHAM Specification

Let us assume we have already specified the register file and ALU, with the signatures below:

```
data Reg = R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7

regFile :: (Signal Reg, Signal Int) ->
           Signal Reg ->
           Signal Reg ->
           (Signal Int, Signal Int)

data Cmd = ADD | SUB | INC

alu :: Signal Cmd -> Signal Int -> Signal Int ->
       Signal Int
```

The `regFile` specification takes a write-port input, two read-port inputs, and returns the corresponding read-port outputs. The `alu` specification takes a command signal and two input signals, and returns a result signal. Given these signatures and the previous definition of `delay`, it is easy in Hawk to specify an unpipelined version of SHAM:

```
sham1 :: (Signal Cmd,Signal Reg,
          Signal Reg,Signal Reg) ->
         (Signal Reg,Signal Int)

sham1 (cmd,destReg,srcRegA,srcRegB) =
  (destReg',aluOutput')
  where
    (aluInputA,aluInputB) =
      regFile (destReg',aluOutput')
              srcRegA srcRegB
    aluOutput = alu cmd aluInputA aluInputB
    aluOutput' = delay 0 aluOutput
    destReg' = delay R0 destReg
```

The definition of `sham1` takes a tuple of signals representing the stream of instructions, and returns a pair of signals representing the sequence of register assignments generated by the instructions. The first three lines in the body of `sham1` read the source register values from the register file and perform the ALU operation. The next two lines delay the destination register name and ALU output, in effect returning the values of the previous clock cycle. The delayed signals become the write-port for the register file. It is necessary to delay the write-port since modifications to the register file logically take effect for the next instruction, not the current one.

3.2 Pipelining

Suppose we wanted to increase SHAM's performance by doubling the clock frequency. We will assume that, while `sham1` could perform both the register file and ALU operations within one clock cycle, with the increased frequency it will take two clock cycles to perform both functions serially. We use pipelining to increase the overall performance. While the ALU is working on instruction n, the register file will be writing the result of instruction $n-1$ back into the appropriate register, and simultaneously reading the source registers of instruction $n+1$.

But now consider the following sequence of instructions, such as:

```
R2 <- R1 ADD R3
R4 <- R2 SUB R5
```

When the `ADD` instruction is in the ALU stage, the `SUB` instruction is in the register-fetch stage. But one of the registers that is being fetched (`R2`), has not been written back into the register file yet, because the ALU is still calculating the result. The `SUB` instruction will read an out-of-date value for `R2`. This is an example of a *data hazard*, where naive pipelining can produce a result different from the unpipelined version of a microprocessor. To resolve this hazard, we will first add *bypass logic* to the pipeline, then later abstract away from this added inconvenience.

Figure 3 contains the diagram of a pipelined version of SHAM with bypass logic. By the time the source operands to the `SUB` instruction (`R2` and `R5`) are ready to be input into the ALU, the up-to-date value for `R2` is stored in the delay circuit between the ALU and the register file's write-port. The bypass logic uses this stored value of `R2` as the input to the ALU, rather than the out-of-date value read from the register file. The bypass logic examines the incoming instructions to determine when this is necessary. The following code contains the Hawk specification:

```
sham2 :: (Signal Cmd,Signal Reg,
          Signal Reg,Signal Reg)
         ->
         (Signal Reg,Signal Int)

sham2 (cmd,destReg,srcRegA,srcRegB) =
  (destReg'',aluOut')
  where
    (valueA,valueB) = regFile (destReg'',aluOut')
                                srcRegA srcRegB

    valueA'    = delay 0 valueA
    valueB'    = delay 0 valueB
    destReg'   = delay R0 destReg
    cmd'       = delay ADD cmd

    aluInputA  = select validA valueA' aluOut'
    aluInputB  = select validB valueB' aluOut'

    aluOut     = alu cmd' aluInputA aluInputB

    aluOut'    = delay 0 aluOut
    destReg''  = delay R0 destReg'

    --- Control logic ---

    validA     = delay True (noHazard srcRegA)
    validB     = delay True (noHazard srcRegB)

    noHazard :: Signal Reg -> Signal Bool
    noHazard srcReg =
      sigOr (sigEqual destReg' (constant R0))
            (sigNotEqual destReg' srcReg)
```

The first two lines after the `where` keyword read the contents of the source registers from the register file. The next four lines delay the source register contents, the ALU command, and the destination register name by one cycle. The two `select` commands decide whether the delayed values should be bypassed. The decision is made by the Boolean signals `validA` and `validB`, which are defined in the control logic section. The next line performs the ALU operation. The last two lines in the data-flow section delay the ALU result and the destination register. The delayed result, called `aluOut'`, is written back into the register file in the register named by `destReg''`, as indicated in the first two lines of the section.

The control logic section determines when to bypass the ALU inputs. The signals `validA` and `validB` are set to `True` whenever the corresponding ALU input is up-to-date. The definition of these signals uses the function `noHazard`, which tests whether the previous instruction's destination register name matches a source register name of the current instruction. If they do, then the function returns `False`. The exception to this is when the destination register is `R0`. In this case the ALU input is always up-to-date, so `noHazard` returns `True`.

3.3 Transactions

The definition of `sham2` highlights a difficulty of many such specifications. Although the data flow section is relatively easy to understand, the control logic section is far from satisfactory. In fact, it often takes nearly as many lines of Hawk code to specify the control logic as it does to specify the data flow, and mistakes in the control logic may not be easy to spot. We need a more intuitive way of defining control logic sections in microprocessors.

We use a notion of *transactions* within Hawk to specify the state of an entire instruction as it travels through the microprocessor (similar in spirit to

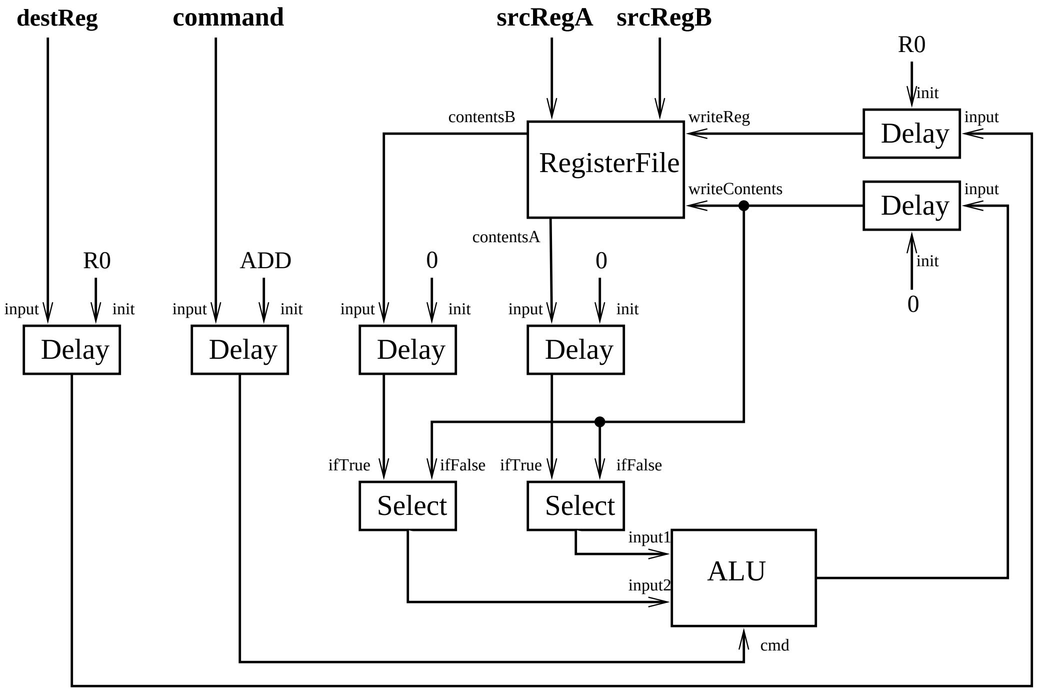

Figure 3: **Pipelined SHAM. Since the register file and the ALU each now take one clock cycle to complete, we now need *Delay* circuits. The *Delay* circuits in turn require us to add *Select* circuits to act as bypasses. The logic controlling the *Select* circuits is not shown.**

Aagaard and Leeser [1]). A transaction holds an instruction's source operand values, the ALU command, and the destination operand value. Transactions also record the register names associated with the source and destination operands:

```
data Transaction = Trans DestOp Cmd [SrcOp]

type DestOp  = Operand
type SrcOp   = Operand
type Operand = (Reg,Value)

data Value = Unknown | Val Int
```

An operand is a pair containing a register and its value. Values can either be "unknown" or they can be known, e.g. `Val 7`.

For example, the instruction `(R3 <- R2 ADD R1)`, when it has completed, would be encoded as shown below (assume that register `R2` holds the value `3`, and `R1` holds `4`):

```
Trans (R3,Val 7) ADD [(R2,Val 3),(R1,Val 4)]
```

This expression states that register `R3` should be assigned the value `7` as a result of adding the contents of register `R2` and `R1`.

Not all of the register values in a transaction are known in the early stages of the pipeline. When a register name does not have an associated value yet, it is assigned the value `Unknown`. For example, if the above instruction had not reached the ALU stage yet, then the corresponding transaction would be:

```
Trans (R3,Unknown) ADD [(R2,(Val 3)),(R1,Val 4))]
```

Figure 4 shows how a transaction's values are filled in as it flows through the pipeline.

3.4 Transaction structure

In general, the `Transaction` datatype contains three subfields. The first field holds the destination register name and its current state. The *state* of a register indicates the current value for the register at a given stage of the pipeline. Possible state values are `Unknown`, or `(Val k)`. The second field is the instruction's ALU operation, in this case the `ADD` command. The third field holds a list of source operand register names and their corresponding states. In this example, it holds the names and states for the source operands `R2` and `R1`.

The instruction `(R3 <- R2 ADD R1)`, before it enters the SHAM pipeline, is encoded as the transaction:

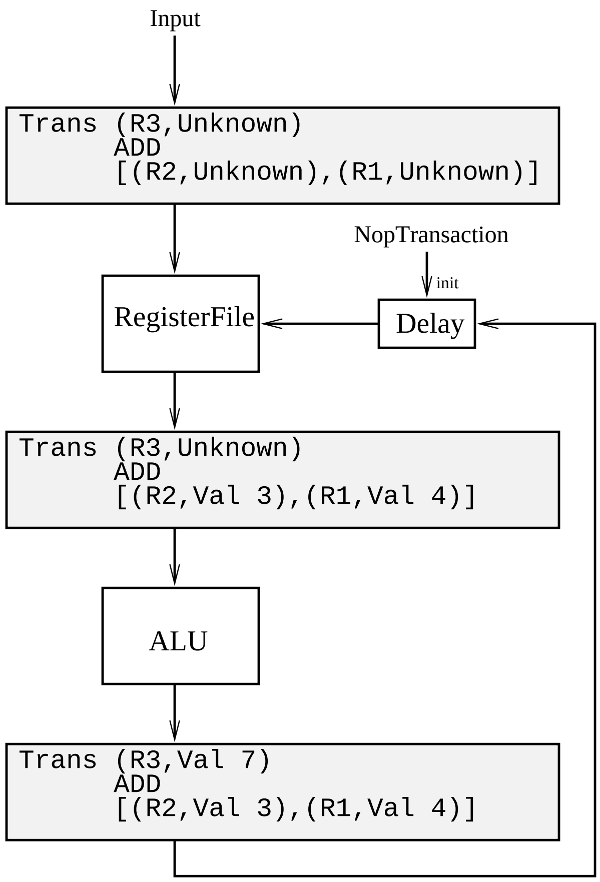

Figure 4: **A transaction as it flows through the pipeline. As the transaction progresses, its operands become more refined.**

```
Trans (R3,Unknown) ADD [(R2,Unknown),(R1,Unknown)]
```

At this point, none of the register values are known.

3.5 Changes to handle transactions

We change the `regFile` and `alu` functions so that they take and return transactions:

```
regFile :: Signal Transaction ->
           Signal Transaction ->
           Signal Transaction

alu :: Signal Transaction ->
       Signal Transaction
```

Because the register file needs to both write new values to the CPU registers and read values from them, the `regFile` function takes a *write-transaction* and a *read-transaction* as inputs. The function examines the destination register field of the write-transaction and updates the corresponding register in the register file. It outputs the read-transaction, modified so that all of the source register fields contain current values from the register file. For example, suppose `regFile` is applied to the completed write-transaction:

```
Trans (R1,Val 4) INC [(R1,Val 3)]
```

and uses as the read transaction:

```
Trans (R3,Unknown) ADD [(R2,Unknown),(R1,Unknown)]
```

Further, assume that register `R1` is assigned `20` and `R2` is assigned `3` before `regFile`'s application. Then `regFile` will update `R1` to contain `4` from the write-transaction, and will output a new transaction that is identical to the read-transaction, except that all of the source registers have been assigned current values from the register file:

```
Trans (R3,Unknown) ADD [(R2,Val 3),(R1,Val 4)]
```

The revised `alu` function takes a transaction whose source operands have values, performs the appropriate operation, and outputs a modified transaction whose destination field has been filled in. Thus if the `ADD` transaction above were given to `alu`, it would return:

```
Trans (R3,Val 7) ADD [(R2,Val 3),(R1,Val 4)]
```

3.6 Unpipelined SHAM

Using transactions, the unpipelined version of SHAM is even easier to specify than it was before.

```
sham1Trans :: Signal Transaction ->
              Signal Transaction
sham1Trans instr =  aluOutput'
  where
    aluInput = regFile aluOutput' instr
    aluOutput = alu aluInput
    aluOutput' = delay nop aluOutput

nop = Trans (R0,Val 0) ADD [(R0,Val 0),(R0,Val 0)]
```

But the real benefit of transactions comes from specifying more complex micro-architectures, as we shall see next.

3.7 `SHAM2` with Transactions

Transactions are designed to contain the necessary information for concisely specifying control logic. The control logic needs to determine when an instruction's source operand is dependent on another instruction's destination operand. To calculate the dependency, the source and destination register names must be available. The transaction carries these names for each instruction. Because of this additional information, bypass logic is easily modeled with following combinator:

```
bypass :: Signal Transaction ->
          Signal Transaction ->
          Signal Transaction
```

The `bypass` function usually just outputs its first argument. Sometimes, however, the second argument's destination operand name matches one or more of the first argument's source operand names. In this case, the source operand's state values are updated to match the destination operand state value. The updated version of the first argument is then returned.

So if at clock cycle n the first argument to bypass is:

```
Trans (R4,Unknown) ADD [(R3,Val 12),(R2,Val 4)]
```

and the second argument at cycle n is:

```
Trans (R3,Val 20) SUB [(R8,Val 2),(R11,Val 10)]
```

then because `R3` in the second transaction's destination field matches `R3` in the first transaction's source field, the output of `bypass` will be an updated version of the first transaction:

```
Trans (R4,Unknown) ADD [(R3,Val 20),(R2,Val 4)]
```

One special case to `bypass`'s functionality is when a source register is `R0`. Since `R0` is a constant register, it does not get updated. The pipelined version of SHAM with bypass logic is now straightforward. Notice that no explicit control logic is needed, as all the decisions are taken locally in the bypass operations.

```
SHAM2Trans :: Signal Transaction ->
              Signal Transaction

SHAM2Trans instr = aluOutput'
  where
    readyInstr = regFile aluOutput' instr
    readyInstr' = delay nopTrans readyInstr
    aluInput = bypass readyInstr' aluOutput'
    aluOutput = alu aluInput
    aluOutput' = delay nopTrans aluOutput
```

The first line takes `instr` and fills in its source operand fields from the register file. The filled-in transaction is delayed by one cycle in the second line. In the third line `bypass` is invoked to ensure that all of the source operands are up-to-date. Finally the transaction result is computed by `alu` and delayed one cycle so that the destination operand can be written back to the register file.

3.8 Hazards

There are some microprocessor hazards that cannot be handled through bypassing. For example, suppose we extended the SHAM architecture to process load and store instructions:

```
R3 <- MEM[R2]
MEM[R5] <- R2
```

The first instruction above is a load instruction; it loads the contents of the address pointed to by `R2` into `R3`. The second instruction is a store; it stores the contents of `R2` into the address pointed to by `R5`. A block diagram of the extended SHAM architecture is shown in Figure 5. There is now a load/store pipeline stage after the ALU stage. However, this introduces a new problem. Suppose SHAM executes the following two instructions in sequence:

```
R2 <- MEM[R1]
R4 <- R2 ADD R3
```

These two instructions have a data hazard, just as before, but we can not use bypassing to resolve it. Bypassing depends on having a value to bypass at the *beginning* of a clock cycle, but `R2`'s value won't be known until the end of the cycle, after the memory contents have been retrieved from the memory cache. To resolve this hazard, we have to *stall* the pipeline at the register-fetch stage. When the first instruction has reached the end of the ALU stage, the second instruction will have reached the end of the register-fetch stage. At this point the delay circuits between the register-fetch stage and the ALU stage are overridden; on the next clock cycle they instead output the equivalent of a no-op instruction. The register-fetch stage itself re-reads the second instruction on the next clock cycle. In effect, the pipeline stall inserts a no-op instruction between the two instructions involved in the hazard:

```
R2 <- MEM[R1]
NOP
R4 <- R2 ADD R3
```

Now when the `ADD` instruction is about to be processed by the ALU, the load instruction has already completed the memory stage. `R2`'s value is held in the pipeline registers after the memory stage, so bypass logic can be used to bring the ALU's input up-to-date. In order to stall correctly, we have to re-read the second instruction. Thus stalling reduces the performance of the pipeline.

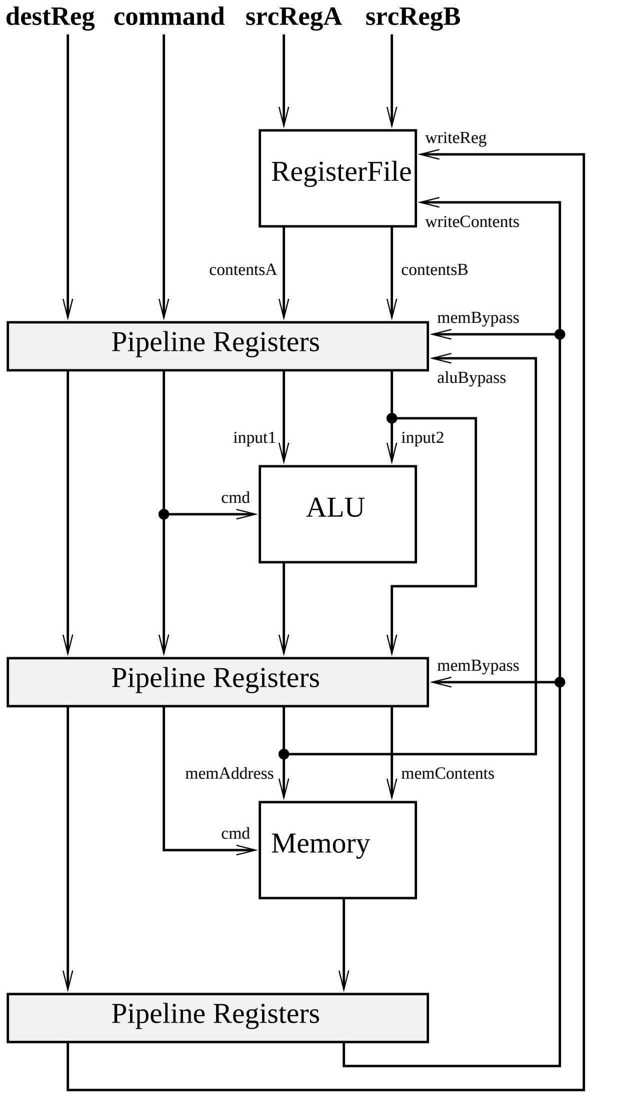

Figure 5: **Block diagram of extended SHAM pipeline. Each *Pipeline Register* circuit is made up of multiple *Delay* and *Select* circuits. The *Select* circuits are used for bypassing, ensuring that the source operands are up-to-date.**

3.9 Hawk Specification of Extended SHAM

In this section we will give more evidence of the simplifying power of transactions by specifying the extended SHAM architecture. The load/store extension significantly complicates the control logic for the SHAM architecture. We shall see that transactions hold up well when we must add stalling logic to the pipeline.

To start, we need to add the commands `LOAD` and `STORE` to the `Cmd` type:

```
data Cmd = ADD | SUB | INC | LOAD | STORE
```

We also need to define some additional Hawk circuits. The first circuit, `defaultDelay`, augments the normal `delay` circuit so that when a stall hazard is detected, the augmented circuit will output a default value on the next clock cycle, rather than its current input value:

```
defaultDelay :: Signal Bool -> a -> Signal a ->
                Signal a

defaultDelay emitDefault default input =
  delay default (select emitDefault
                        (constant default)
                        input)
```

The `defaultDelay` circuit uses `delay` to store values between clock cycles. The value it stores for the next clock cycle is `default` if `emitDefault` is equal to `True` on the current cycle, otherwise it stores `input`. On the first cycle of the simulation `defaultDelay` always returns `default`.

The `isLoadTrans` circuit returns `True` whenever its argument signal is a load transaction:

```
isLoadTrans :: Signal Transaction -> Signal Bool
isLoadTrans ts = lift isLoad ts
  where
    isLoad (Trans _ cmd _) = (cmd == LOAD)
```

Although we previously passed SHAM instructions as parameters, we now need to call a function, `instrCache`, to explicitly retrieve them:

```
instrCache :: Signal Bool -> Signal Transaction
```

Since the pipeline can stall, we need a way to ask for the same instruction two cycles in a row. The `instrCache` function takes a Boolean signal and returns the current transaction. Whenever the argument signal is `True`, then on the next cycle `instrCache` returns the same transaction as it did for the current clock cycle. Otherwise, it returns the next transaction as normal.

We also need a circuit that actually performs the loads and stores:

```
mem :: Signal Transaction -> Signal Transaction
```

On those clock cycles where the input transaction is anything but a load or store transaction, the `mem` function simply returns the transaction unchanged. On loads, `mem` updates the destination operand of the input transaction, based on the input load address. On stores, `mem` updates its internal memory array according to the address and contents given in the input transaction. The destination operand value is set to zero.

We also define a new Hawk function, `transHazard`, that returns `True` whenever its two transaction arguments would cause a hazard, if the first transaction preceded the second transaction in a pipeline:

```
transHazard :: Signal Transaction ->
               Signal Transaction ->
               Signal Bool
```

The extended Hawk specification using transactions is given below:

```
SHAM3Trans :: Signal Transaction
SHAM3Trans = memOut'
  where

     -- register-fetch stage --
     instr = instrCache loadHzd
     readyInstr = regFile memOut' instr
     readyInstr' =
       defaultDelay loadHzd nopTrans readyInstr

     -- ALU stage --
     aluIn = bypass (bypass readyInstr' memOut')
                    aluOut'
     aluOut = alu aluIn
     aluOut' = delay nopTrans aluOut

     -- memory stage --
     memIn = bypass aluOut' memOut'
     memOut = mem memIn
     memOut' = delay nopTrans memOut

  ----- Control logic -----

     loadHzd =
       sigAnd (isLoadTrans readyInstr')
              (transHazard readyInstr'
                           readyInstr)
```

The register-fetch stage retrieves the instruction and fills in its source operands from the register file. The register-fetch pipeline register delays the transaction by one clock cycle, although if there is a load hazard, the register instead outputs a nop-instruction on the next cycle. The ALU stage first updates the source operands of the stored transaction with the results of the two preceding transactions (`memOut'` and `aluOut'`) by invoking bypass twice. It then performs the corresponding ALU operation, if any, on the transaction and stores it in the ALU-stage pipeline register. The memory stage again updates the stored transaction with the immediately preceding transaction, performs any required memory operation, and stores the transaction. The stored transaction is written back to the register file on the next clock cycle. The control logic section determines whether a load hazard exists for the current transaction, that is, whether the immediately preceding transaction was a load instruction that is in hazard with the current transaction.

As we can see, the body of the specification remains manageable. The small control logic section to detect load hazards is straightforward and is a minority of the overall specification. In contrast, an equivalent specification of this pipeline where the components of each transaction were explicitly represented contained over three times as many source lines. The lower-level specification's control section was almost as large as the dataflow section, and not nearly as intuitive.

We feel the transaction ADT is close to the level of abstraction design engineers use informally when reasoning about microprocessor architectures.

4 Modelling the DLX

Using techniques comparable to those described in this report we have modeled several DLX architectures:

- An unpipelined version, where each instruction executes in one cycle.
- A pipelined version where branches cause a one-cycle pipeline stall.
- A more complex pipelined version with branch prediction and speculative execution. Branches are predicted using a one-level branch target buffer. Whenever the guess is correct, the branch instruction incurs no pipeline stalls. If the guess is incorrect, the pipeline stalls for two cycles.
- An out-of-order, superscalar microprocessor with speculative execution. The microarchitecture contains a reorder buffer, register alias table, reservation station, and multiple execution units. Mispredicted branches cause speculated instructions to be aborted, with execution resuming at the correct branch successor.

The microarchitectural specification for the unpipelined DLX is written in a quarter page of uncommented source code; the most complicated pipelined version takes up just over half a page.

4.1 Executing the model

We used the Gnu C compiler that generates DLX assembly to test our specifications on several programs. These test cases include a program that calculates the greatest common divisor of two integers, and a recursive procedure that solves the towers of Hanoi puzzle.

We have not made detailed simulation performance measurements yet. Although we plan to test Hawk on several benchmark programs, we do not expect to break simulation-speed records. Hawk is built on top of a lazy functional language, which imposes some performance costs. Transactions also perform some run-time tests that are "compiled-away" in a lower-level pipeline specification. While it would be nice to get high performance, Hawk is primarily a specification language, and only secondarily a simulation tool. Our main interest is in using Hawk to formally verify microarchitectures, while at the same time retaining the ability to directly execute Hawk programs on concrete test cases.

5 Related Work

There are several research areas that bear a relation on this work, in particular, modeling specific application domains with Haskell, and modeling hardware in various programming languages. We will pick an example or two from these two categories.

Haskell has been used to directly model hardware circuits at the gate level. O'Donnell [10] has developed a Haskell library called Hydra that models gates at several levels of abstraction, ranging from implementations of gates using CMOS and NMOS pass-transistors, up to abstract gate representations using lazy lists to denote time-varying values. Hydra has been used to teach advanced undergraduate courses on computer design, where students use Hydra to eventually design and test a simple microprocessor. Hydra is similar to Hawk in many ways, including the use of higher-order functions and lazy lists to model signals. However, Hydra does not allow users to define *composite* signal types, such as signals of integers or signals of transactions. In Hydra, these composite types have to be built up as tuples or lists of Boolean signals. While this limitation does not cause problems in an introductory computer architecture course, composite signal types significantly reduce specification complexity for more realistic microprocessor specifications.

There are many other languages for specifying hardware circuits at varying levels of abstraction. The most widely used such languages are Verilog and VHDL. Both of these languages are well suited for their roles as general-purpose, large-scale hardware design languages with fine-grained control over many circuit properties. Both of these languages are more general than Hawk in that they can model asynchronous as well as synchronous circuits. However, Verilog and VHDL are large languages with complex semantics, which makes circuit verification more difficult. Also, neither of these languages support polymorphic circuits, nor higher-order circuit combinators, as well as Hawk.

The Ruby language, created by Jones and Sheeran [7], is a specification and simulation language based on relations, rather than functions. Ruby is more general than Hawk in that relations can describe more circuits than functions can. On the other hand, existing Ruby simulators require Ruby relations to be *causal*, i.e. to be implementable as functions. Thus Hawk is equal in expressive power to currently executable Ruby programs. In addition, much of Ruby's emphasis is on circuit layout. There are combinators to specify where circuits are located in relation to each other and to external wires. Hawk's emphasis is on behavioral correctness, so we do not need to address layout issues.

Two other languages that are strongly related are HML [8] and MHDL[2]. HML is a hardware modeling language based on the functional language ML. It also has higher-order functions and polymorphic types, allowing many of the same abstraction techniques that are used in Hawk, with similar safety guarantees. On the other hand, HML is not lazy, so does not easily allow the recursive circuit specifications that turned out to be key in specifying micro-architectures. The goal of HML is also rather different from Hawk, concentrating on circuits that can be immediately realized by translation to VHDL.

MHDL is a hardware description language for describing analog microwave circuits, and includes an interface to VHDL. Though it tackles a very different part of the hardware design spectrum, like Hawk, MHDL is essentially an extended version of Haskell. The MHDL extensions have to do with physical units on numbers, and universal variables to track frequency and time etc.

6 Future Directions

We have just completed the specification of a superscalar version of DLX, with speculative and out-of-order instruction execution. The use of transactions has scaled well to this architecture; it turns out that superscalar components like *reservation stations* and *reorder buffers* are naturally expressed as queues of transactions.

Beyond this, we intend to push in a number of directions.

- We hope to use Hawk to formally verify the correctness of microprocessors through the mechanical theorem prover Isabelle [11]. Isabelle is well-suited for Hawk; it has built-in support for manipulating higher-order functions and polymorphic types. It also has well-developed rewriting tactics. Thus simplification strategies for functional languages like partial evaluation and deforestation [3] can be directly implemented.

  We also expect that transactions will aid the verification process. Transactions make explicit much of the pipeline state needed to prove correctness. In lower-level specifications this data has to be inferred from the pipeline context.

- We are also working on a visualization tool which will enable the microprocessor engineer to inspect values passing along internal wires.

- We have made initial progress on formally extracting stand-alone control logic from the transaction-based models of pipelines. Stand-alone control logic may be more amenable to conventional synthesis techniques.

7 Acknowledgements

We wish to thank Simon Peyton Jones, Carl Seger, Borislav Agapiev, Dick Kieburtz, and Elias Sinderson 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 recieves 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] Barton, D. Advanced modeling features of MHDL. In *International Conference on Electronic Hardware Description Languages* (Jan. 1995).

[3] Gill, A., Launchbury, J., and Jones, S. P. A short-cut to deforestation. In *ACM Conference on Functional Programming and Computer Architecture* (Copenhagen, Denmark, June 1993).

[4] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1995.

[5] Hudak, P., Peterson, J., and Fasel, J. A gentle introduction to Haskell. Available at www.haskell.org, Dec. 1997.

[6] Johnson, M. *Superscalar Microprocessor Design.* Prentice Hall, 1991.

[7] Jones, G., and Sheeran, M. Circuit design in Ruby. In *Formal Methods for VLSI Design*, J. Staunstrup, Ed. North-Holland, 1990.

[8] Li, Y., and Leeser, M. HML: An innovative hardware design language and its translation to VHDL. In *Conference on Hardware Design Languages* (June 1995).

[9] Melham, T. Abstraction mechanisms for hardware verification. In *VLSI Specification, Verification and Synthesis*, G. Birtwhistle and P. A. Subrahmanyam, Eds. Kluwer Academic Publishers, 1988.

[10] O'Donnell, J. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Symposium on Functional Programming Languages in Education* (July 1995).

[11] Paulson, L. *Isabelle: A Generic Theorem Prover.* Springer-Verlag, 1994.

[12] Peterson, J., and et al. Report on the programming language Haskell: A non-strict, purely functional language, version 1.4. Available at www.haskell.org, Apr. 1997.

[13] Sinderson, E., and et al. Hawk: A hardware specification language, version 1. Available at www.cse.ogi.edu/PacSoft/projects/Hawk/, Oct. 1997.

[14] Windley, P., and Coe, M. A correctness model for pipelined microprocessors. In *Second International Conference on Theorem Provers in Circuit Design* (Sept. 1994).