Overview of Hydra: A Concurrent Language for Synchronous Digital Circuit Design

John T. O'Donnell
Computing Science Department, University of Glasgow,
Glasgow G12 8QQ, United Kingdom
jtod@dcs.gla.ac.uk
www.dcs.gla.ac.uk/~jtod/

Abstract

Hydra is a computer hardware description language that integrates several kinds of software tool (simulation, netlist generation and timing analysis) within a single circuit specification. The design language is inherently concurrent, and it offers black box abstraction and general design patterns that simplify the design of circuits with regular structure. Hydra specifications are concise, allowing the complete design of a computer system as a digital circuit within a few pages. This paper discusses the motivations behind Hydra, and illustrates the system with a significant portion of the design of a basic RISC processor.
Keywords: hardware description language, synchronous circuit

1 Introduction

Software systems that support the design of digital circuits have become crucial for the entire application area of hardware design. Such systems are called *computer hardware description languages* (CHDLs), and they generally comprise a specification language which the designer uses to describe the circuit precisely, along with a set of software tools that provide a range of necessary services such as simulation, netlist generation, and timing analysis.

Hardware description languages are much like programming languages, as both are used to express algorithms. The main difference lies in the execution platform: programming languages express algorithms that run on computers, whereas hardware description languages describe the structure and behaviour of digital circuits. At a practical level, several issues—particularly multiple semantics and a highly concurrent model of behaviour—make hardware design languages quite different from programming languages.

The purpose of this paper is to give an overview of Hydra, a CHDL designed specifically to address the issues of concurrency, multiple semantics and behavioural models. Hydra was motivated by Johnson's work on modeling hardware with recursion equations [13], and previous papers have discussed its use of recursion equations [19], geometric design patterns [20], overloaded semantics [22], preliminary approaches to the difficult problem of netlist generation [21], and applications to formal reasoning [23]. This paper gives a brief introduction

to the current version of Hydra, discusses how the key issues identified above are addressed, and illustrates how Hydra is used in practice by presenting a significant portion of a digital circuit implementing a simple CPU.

Section 2 motivates Hydra, while Section 3 defines the underlying synchronous model. Multiple circuit semantics is discussed in Section 4, and Section 5 describes design patterns. Section 6 illustrates the flavour of the language through an extended example, Section 7 describes related work, and Section 8 concludes.

2 Motivation

Digital circuits are highly concurrent, yet many circuit specification languages are based on sequential programming languages with special support for concurrency. This masks the concurrency inherent in the hardware, making it unnecessarily difficult for the designer to grasp the timing of events. Ideally, a hardware description language should not require the designer to introduce spurious serialization that has nothing to do with the hardware, but is simply inherited from an imperative programming paradigm.

Practical hardware design requires a variety of software tools to help with expressing the design, synthesizing parts of the circuit that can be generated automatically, ensuring correctness, analysing the performance, and building the hardware. These software tools are so different that many systems provide a family of related specification languages, leading to potential inconsistency. For example, four of the most important software tools are regular pattern generators, circuit simulators, netlist generators, and timing analysers. However, these tools require quite different information about the circuit, and consequently many circuit design languages separate the specification of the structure of a circuit from the specification of its behaviour. Unfortunately, this approach can sometimes allow inconsistencies to arise among the multiple specifications of a circuit. Thus the designer might simulate a circuit and find that it works, and then generate the netlist in order to fabricate the circuit, only to discover that the real hardware fails to work because it is different from the version of the circuit that was simulated. The multiple specification approach is dangerous.

Complex digital circuits contain many components, ranging from the order of 10^4 components for a simplified processor to 10^8 for a current high performance chip. Designers are more productive when the specifications of such large systems are kept as simple as possible. Several effective methods for achieving this are utilized in Hydra, including black box abstraction (Section 4), design patterns (Section 5), and the separation between datapath and control (Section 6).

Digital circuits have behaviour at many levels of abstraction [12], ranging from effects that depend on quantum physics (such as tunnel diodes) all the way up to large scale systems (such as multiprocessor systems). Some CHDLs attempt to cover both physical and logical aspects of a circuit in the same specification. This is flexible, but it can be difficult to control so many levels of complexity at the same time.

One example of the confusion that can result from mixing the physical and logical levels arises from the treatment of clocks. Some standard circuits perform an action (e.g. loading a register) at every clock tick, but the designer may prefer

to load the register only at clock ticks when a special control signal *ctl_ld* is 1. Exactly this situation is common in circuit design, and a popular solution is to generate a special clock signal for the register by anding the real clock with *ctl_ld*. The problem with this approach is that clock skew results from the and-gate on the register's clock input, and the circuit may no longer have a simple behaviour describable as a state machine. This is arguably a misuse of the components: a true conditional load register (as presented in Section 4.1) should be used, and it is poor design style to force the wrong component to do a job by misusing its inputs. An aim of Hydra is to support simple reasoning about the correctness of circuits, so tricks like performing logic on clock signals are banned.

All of the issues discussed above were considered in the development of Hydra. The language uses a simple functional model of circuits, enabling it to be implemented by embedding within an ordinary functional language, Haskell [25]. Hydra is inherently concurrent, and it provides alternative semantics to allow different software tools to be applied to the same circuit specification. Finally, Hydra embodies a precise implementation of the synchronous model for digital circuits.

3 The Synchronous Model

Most modern circuit design is carried out within the synchronous model, which simplifies reasoning about behaviour at the cost of strict requirements on clocking. An active current research subject is aimed at finding ways to relax the constraints of the synchronous model in order to achieve higher speeds without having too damaging an impact on the ability to understand and reason about designs. This section briefly summarizes the synchronous model [3].

All digital components are classified as either combinational or sequential. A combinational component produces one or more outputs that depend solely on the current values of the inputs; i.e. there is no internal state. Every physical component takes some time to respond to a change in its inputs, so a change in the input values to component f at time t will be reflected in the outputs at time $t + d_f$, where d_f is the gate delay of f.

Suppose that all the inputs I to a combinational circuit are known at the beginning of a clock cycle, and they remain stable for the duration of the cycle. Each signal y has a path depth defined as follows. If $y \in I$ then $pd(y) = 0$. Otherwise, y must be an output of a component c whose inputs are $x_0, x_1, \ldots, x_{k-1}$, and y becomes valid one gate delay after all those inputs are valid. Therefore $pd(y) = 1 + \max(pd(x_0), \ldots, pd(x_{k-1}))$. Feedback within combinational circuits is disallowed, so each signal has a well founded path depth, and it is guaranteed that a signal y is valid after $pd(y)$ gate delays have elapsed. The critical path depth of the circuit is the maximum path depth over all signals.

A sequential component produces outputs that depend on an internal state and a global clock, as well as on the current input values. The clock can be viewed abstractly as defining a sequence of points in time called *clock ticks*. The delay flip flop `dff` has a one-bit state which it outputs continuously, and it receives a data input which it stores in the state at each clock tick. As long as the clock period is larger than the product of the gate delay and the critical path depth, the inputs to all flip flops are guaranteed to be valid whenever a tick occurs.

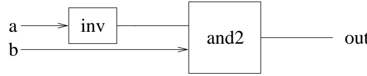

Figure 1: Named and anonymous signals

The synchronous model has costs that may impair performance: it prohibits logic operations on clock signals, disallows purely combinational feedback loops, and places a speed limit on the clock. The benefit is a simple semantics that supports the design of very large circuits: the combinational logic can be analysed using Boolean algebra, and the behaviour of the entire system can be analysed as a single state machine, where the state of the entire circuit is a vector consisting of all the individual flip flop states.

4 Executable Specification

Hydra uses mathematical functions to model circuits: the inputs to a circuit are the arguments to the function, and the result defines the circuit outputs.

By defining the circuit functions in a suitable programming language, we can give a semantics to the circuit. As explained in Section 2, there are several different semantics required for practical design. Hydra addresses this problem by providing libraries containing alternative definitions for all the primitive circuits. For example, the inverter function `inv` has several definitions; one will simulate it, another will generate its netlist, and another will analyse its timing. Hydra uses a distinct signal type for each semantics, so we can execute a circuit simply by applying its specification to inputs of a suitable type *without modifying the circuit specification.* For example, by applying the `inv` function to a Boolean we can simulate it; by applying it to a wire name we get a netlist; by applying it to a path depth table we get a timing analysis.

Hydra is implemented by embedding it in the standard functional language Haskell [25], which supports the choice of semantics through function overloading based on type. Haskell also supports formal reasoning, it provides nonstrict semantics, which simplifies the implementation of circuit simulation, and it has a rich syntax and semantics for programming with functions.

4.1 Hydra Specifications

A component is introduced by providing it with inputs, and the result is one or more output signals. For example, the expression `and2 a b` denotes the output produced by a two-input logical and gate connected to inputs `a` and `b`. Equations are used to give names to signals; the name appears on the left hand side, and the signal value on the right hand side. The input to a circuit can be either a named signal or an output from a circuit. For example, the circuit shown in Figure 1 is specified by `out = and2 (inv a) b`. This defines `out` to be the name of the output produced by the and2 gate, while the inverter output is simply connected to an input of the and2 gate.

Abstraction—the definition and reuse of black box circuits—is the most important technique for keeping specifications concise while allowing large circuits.

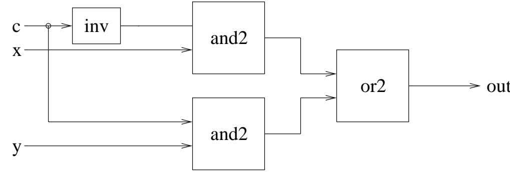

Figure 2: Multiplexer

A Hydra circuit definition is used to create a new black box circuit using existing components and circuits. With the definition in place, further circuits can incorporate the newly defined black box just like any other component or circuit. A circuit definition consists of two parts: an optional type declaration (the line containing `::`) which specifies the number and organization of the inputs and outputs to the circuit, and a defining equation. The multiplexer (Figure 2) is a standard black box that implements a conditional: its output is x if $c = 0$, and otherwise y.

```
mux1 :: Signal a => a -> a -> a -> a
mux1 c x y = or2 (and2 (inv c) x)
                 (and2 c y)
```

A word is a group of signals $[x_0, x_1, \ldots, x_{k-1}]$, where the elements are indexed. A tuple is a group of related signals, where the elements are referred to by name rather than index.

Feedback is introduced using self-referential equations with a unique annotation of the form `label` i (used to assist with netlist generation). The 1-bit register circuit `reg1` contains a delay flip flop within a feedback loop. The register outputs the state of the flip flop, which is given the name `s` by the local equation. At each clock tick, the flip flop will store a new value, and this should be either the old state value `s` (if the load control `ld` is 0) or the data input value `x` (if `ld` is 1). The correct value to be placed into the flip flop is selected by a multiplexer.

```
reg1 :: Clocked a => a -> a -> a
reg1 ld x =
  let s = label 0 (dff (mux1 ld s x))
  in s
```

4.2 Circuit Behaviour: Simulation

Hydra defines the behaviour of a circuit to be a simulation function that takes the circuit inputs as arguments, and returns the circuit outputs as results. In addition to serving as a formal definition of the circuit behaviour, the simulation functions are useful for practical testing [11], since it is costly to fabricate a circuit using real hardware.

Circuit simulation in Hydra is based on recursively defined systems of streams. A stream is an infinite list of values $[x_0, x_1, x_2, \ldots]$, where x_i is the value of the signal during clock cycle i. Combinational components are simulated

by mapping a logic function over a stream of values. For example, the inverter implements the $(\neg)$ logic function, so an inverter whose input signal is $[x_0, x_1, x_2, \ldots]$ will output $[\neg x_0, \neg x_1, \neg x_2, \ldots]$. Thus the inverter semantics is defined by $\mathit{inv}\ \mathit{xs}\ =\ \mathit{map}\ (\neg)\ \mathit{xs}$.

The delay flip flop introduces a time shift: its input during cycle i becomes its output during cycle $i+1$, and it has a characteristic "power up" value dff_0 during cycle 0. This means that a flip flop with input $[x_0, x_1, x_2, \ldots]$ will produce the output stream $[\mathit{dff}_0, x_0, x_1, x_2, \ldots]$, which has the value x_i during cycle $i+1$. Thus the semantics of the flip flop is defined as $\mathit{dff}\ \mathit{xs}\ =\ \mathit{dff}_0 : \mathit{xs}$.

A great advantage of stream simulation is that feedback is handled correctly [13, 19]. For example, the register circuit `reg1`, defined above, introduces feedback through the signal s, which appears on both sides of the equation. This corresponds exactly to the circularity in the circuit diagram. Despite the circularity, the value of $s = [s_0, s_1, \ldots]$ is well founded: s_0 is the initial power-up value dff_0, and s_i depends on s_{i-1} for $i > 0$.

4.3 Parallel Simulation

The main benefit of concurrency in Hydra is that it simplifies reasoning about the timing in digital circuits, which are inherently concurrent. A second benefit is that it supports the development of parallel circuit simulators. All the function applications corresponding to components that operate in parallel can be evaluated simultaneously by a parallel implementation of Haskell, such as GPH [27]. Another approach, which is the subject of current research, is to define an analysis-based transformation that produces an efficient SPMD style parallel simulator from a Hydra specification.

4.4 Circuit Structure: Netlist

A *netlist* is a precise specification of the structure of a circuit which can be used to fabricate a physical hardware system. A netlist consists a list of the components in a circuit, along with a list of all the connections to be made between component ports. Netlists are unreadable to humans, but they can be used to construct hardware automatically. For example, a wire wrap system, used for prototyping circuits of moderate complexity, can take a netlist as input and will produce a physical circuit board as output. VLSI CAD systems can take a netlist (possibly augmented with information about geometric layout) and will generate the masks used by integrated circuit fabrication lines. In short, the ultimate aim of a circuit designer is to produce a netlist that will generate a correct circuit.

Hydra uses a two-step process to generate netlists [20]. First, a new signal type is introduced, representing signals as nodes in a graph structure, and primitive components are redefined to produce a node in the graph. For example, to generate a netlist for the circuit `x = and2 (inv a) b`, we must apply it to suitable netlist-typed inputs `InPort "a"` and `InPort "b"`. Executing this circuit specification now produces a definition of `x` as a graph node:

```
x = OutPort "x" (Prim2 And2
      (Prim1 Inv (InPort "a"))
      (InPort "b"))
```

In the second step, Hydra traverses the graph to build the netlist, which consists of lists of input ports, output ports, components, and wires. A signal is produced by an InPort or a numbered output of a numbered component, and a wire specifies which sinks are connected to a signal. A netlist is straightforward though unreadable; it is intended for fabrication machines, not for being read by people:

```
([(0, InPort "a"), (1, InPort "b")],
 [(2, OutPort "x")],
 [(3, Inv), (4, And2)],
 [((0,0), [(3,0)]), ((1,0), [(4,1)]),
  ((3,1), [(4,0)]), ((4,2), [(2,0)])])
```

Circuits with feedback yield circular graphs. In fact, the circuit graph produced by Hydra is always isomorphic to the corresponding circuit schematic diagram. There is a technical problem with traversing circular graphs in pure functional programming languages, and this gives rise to a difficulty with generating netlists in functional CHDLs. There are several ways the problem can be solved, including pointer equality [20, 5] and explicit signal labeling [21]. The current version of Hydra uses a program transformation to introduce the necessary labels automatically; this allows the `label 0` to be omitted from the definition of `reg1`.

4.5 Circuit Analysis: Path Depth

To make circuits fast, the designer must minimize the critical path depth (the number of gate delays that must be allowed between clock ticks). Hydra provides a special circuit semantics which causes the execution of a specification to output a timing analysis, including the critical path depth.

4.6 Formal Reasoning

Haskell supports formal derivations and correctness proofs through equational reasoning, a form of algebra based on "substituting equals for equals". Hydra inherits this property; indeed, the ability to reason formally about hardware was one of its original aims. Experience has shown that formal reasoning is practical for realistic and useful circuits [23]. An advantage of this technique is that the formalism can help in producing the design, as well as in establishing its correctness. Theorem proving technology [18] can also be applied to Hydra circuits during the design process.

5 Design Patterns

Typical circuits contain large numbers of components arranged in regular patterns. For example, the four-bit ripple carry adder consists of a full adder for each bit position, where each full adder is connected to its neighbors in a uniform manner. A straightforward specification mentions each component and signal explicitly:

```
rippleAdd4 :: Signal a => a -> [(a,a)] -> (a,[a])
rippleAdd4 cin [(x0,y0),(x1,y1),(x2,y2),(x3,y3)] =
  let (c0,s0) = fullAdd (x0,y0) c1
      (c1,s1) = fullAdd (x1,y1) c2
      (c2,s2) = fullAdd (x2,y2) c3
      (c3,s3) = fullAdd (x3,y3) cin
  in (c0, [s0,s1,s2,s3])
```

This style of specification does not scale up well: a realistic 64-bit adder would require 64 equations. A more subtle problem is that it works for only one word size, so we may need to design a large family of circuits at different sizes that are conceptually the same.

A solution to these problems is a *design pattern*, which describes how to build a larger system by replicating a building block circuit and making connections according to a regular pattern. Hydra offers a large library of design patterns. One of them, called `mscanr`, describes a row of building blocks where each takes an input from its right neighbor and sends an output to the left neighbor, allowing a succinct definition of a general n-bit ripple carry adder:

```
rippleAdd :: Signal a => a -> [(a,a)] -> (a,[a])
rippleAdd = mscanr fullAdd
```

Hydra has a library of design patterns for circuits with linear organisation (such as `mscanr`), as well as grid structures, trees, banyans and butterflies, and other general interconnection patterns. Design patterns in Hydra are not built-in language constructs, they can be defined by the designer and added to the libraries for reuse. For example, the `mscanr` design pattern is defined recursively using a general building block circuit `f`. In the base case, when the word input is the empty word `[]`, the pattern generates an empty circuit. In the recursive case, when the input data word has the form `(x:xs)`, with an initial bit `x` and suffix `xs`, the pattern generates one `f` circuit, and it uses the `mscanr` pattern recursively to handle `xs`, and finally it connects the `a'` output from the `mscanr f a xs` to the appropriate input of the `f` circuit.

```
mscanr :: (a->b->(b,c)) -> b -> [a] -> (b,[c])
mscanr f a [] = (a,[])
mscanr f a (x:xs) =
  let (a',ys) = mscanr f a xs
      (a'',y) = f x a'
  in (a'',y:ys)
```

The designer can also use recursion directly to specify circuits, as in the design of a register file. This family of circuits takes an integer size parameter k; by applying `regfile1` to a specific number k we get a register file circuit that contains 2^k registers. The circuit takes a load control `ld` and data input `x`, and three register addresses `d`, `sa` and `sb`. The register file continuously outputs the contents of the registers addressed by `sa` and `sb`, and at a clock tick it assigns `reg[d] := x` if the load control is 1.

The base case of the recursion says that a register file of size 0 is simply a register. The inductive case defines a circuit containing 2^{k+1} registers by constructing two smaller ones with 2^k registers, and introducing multiplexers and demultiplexers to decode the address bits.

```
regfile1 :: Clocked a => Int
   -> a -> [a] -> [a] -> [a] -> a -> (a,a)
regfile1 0 ld d sa sb x =
  let r = reg1 ld x
  in (r,r)
regfile1 (k+1) ld (d:ds) (sa:sas) (sb:sbs) x =
  let (ld0,ld1) = demux1 d ld
      (a0,b0) = regfile1 k ld0 ds sas sbs x
      (a1,b1) = regfile1 k ld1 ds sas sbs x
      a = mux1 sa a0 a1
      b = mux1 sb b0 b1
  in (a,b)
```

6 Design of a CPU

Hydra specifications are more concise than corresponding definitions in VHDL, and much more concise (as well as precise) than schematic diagrams. An entire CPU circuit, including all definitions going right down to individual transistors and wires, can be written in a few hundred lines of Hydra. Several complex circuits, including complete computer systems, have been designed successfully using Hydra. This section gives an impression of what is involved by presenting the key parts of a basic RISC processor circuit. The processor is simplified, and does not support cache or pipelining, but these features can be added.

In addition to black box abstraction and design pattern abstraction, described above, another helpful technique in designing large systems is a separation of datapath and control. A datapath is the part of the circuit that contains the basic registers and combinational functions. The datapath defines a large set of possible operations, and its behaviour is controlled by a set of control signals generated by a control circuit. A useful analogy is to think of the datapath as being like a programming language (it provides a set of possible operations), while the control is like a program (it says which operations will actually be performed).

6.1 Datapath Circuit

The datapath circuit has two inputs: `control`, a tuple consisting of all the signals produced by the control circuit, and an input data word `indat`, which can come either from the memory or from an input device. The Hydra specification defines the word size n and register file size k, and then defines names for the individual control signals: thus `ctl_rf_ld` is the name of the first control signal. Each control signal is analogous to a command, which the datapath should perform when that signal is 1; for example, `ctl_rf_ld` is a command to the register file to perform a load.

```
datapath control indat =
 let
   n = 16  -- word size
   k =  4  -- 2^k registers
   (ctl_rf_ld,  ctl_rf_alu, ctl_rf_sd,
```

```
    ctl_alu_op, ctl_ir_ld,  ctl_pc_ld,
    ctl_ad_ld,  ctl_ad_alu, ctl_ma_pc,
    ctl_x_pc,   ctl_y_ad,   ctl_sto)
         = control
```

The heart of the datapath is the set of internal registers. The register file contains the registers visible to the machine language programmer. As explained above, it reads out two registers in parallel, and these readouts are called `a` and `b`. The datapath also contains an instruction register `ir`, a program counter `pc`, and an address register `ad`. Each register is specified with a size parameter n giving the word size, and has its own load control. For example, `ctl_pc_ld` is a control signal that determines whether the program counter register will load its input `r`. Note that the address register `ad` can load one of two values—the `indat` input or the internal signal `r`—and a multiplexer is used to decide which, according to the control signal `ctl_ad_alu`.

```
    (a,b) = regfile n k ctl_rf_ld ir_d rf_sa rf_sb p
    ir = reg n ctl_ir_ld indat
    pc = reg n ctl_pc_ld r
    ad = reg n ctl_ad_ld (wmux1 ctl_ad_alu indat r)
```

The ALU (arithmetic logic unit) performs operations on two operands `x` and `y` and produces a result `r`. The complete ALU can perform addition, subtraction, and comparisons on two's complement numbers. Its complete specification (about 10 lines of Hydra) is omitted but is similar to the ripple carry adder defined above.

```
    (ovfl,r) = alu n ctl_alu_op x y
```

The internal signals and buses are defined next. Many of them use multiplexers to select the value that is appropriate. For example, the first input to the ALU, `x`, may be either the first register file output `a` or the program counter `pc`. The reason for this is that the ALU is used both to add register values (for example, while executing an instruction like ADD R1,R2,R3) and for incrementing the `pc`.

```
    x  = wmux1 ctl_x_pc a pc
    y  = wmux1 ctl_y_ad b ad
    rf_sa = wmux1 ctl_rf_sd ir_sa ir_d
    rf_sb = ir_sb
    p  = wmux1 ctl_rf_alu indat r
    ma = wmux1 ctl_ma_pc ad pc
    cond = any1 a
```

The next set of equations simply gives names to the fields of an instruction. These equations could be omitted, but they help make the design more readable.

```
    ir_op = field ir  0 4
    ir_d  = field ir  4 4
    ir_sa = field ir  8 4
    ir_sb = field ir 12 4
```

Finally, the set of signals which are to be output by the datapath is specified:

```
in (ma,cond,a,b,ir,pc,ad,ovfl,r,x,y,p)
```

This is the complete datapath design; nothing has been omitted from the full specification except for some comments.

6.2 Control Algorithm

It is best to define the control system in two stages: first as an abstract control algorithm and then as a detailed control circuit. This helps the designer to focus on the intended function of the control without worrying about the implementation details. Furthermore, Hydra provides tools for synthesizing a control circuit automatically from a control algorithm. A small change in the algorithm may result in a large change to the corresponding circuit, so it is much better for the designer to write the algorithm and compile it into hardware

The control algorithm is written in the form of an imperative program, where each statement generates a set of control signals that will cause the datapath to perform the appropriate set of operations. The entire algorithm is embedded within an infinite loop; a control algorithm must never terminate. Inside the loop we have a sequence of statements that generate control signals which take the datapath through the right sequence of operations needed to execute a program. In the first state, called `st_instr_fet`, the machine is about to fetch an instruction whose address is in the program counter and to increment the program counter. Thus it will perform two operations *concurrently*:

```
ir := mem[pc], pc++;
```

Each of these two operations requires several control signals to be set. To fetch a word of memory at the address specified by the `pc` and put it into the instruction register, we need to set `ctl_ma_pc` and `ctl_ir_ld`, and three more control signals will cause the datapath to increment the `pc`. Thus the complete control statement for this state is:

```
st_instr_fet:
  ir := mem[pc], pc++;
     {ctl_ma_pc, ctl_ir_ld, ctl_x_pc,
      ctl_alu=alu_inc, ctl_pc_ld}
```

The control algorithm then performs a case dispatch using the operation code field of the instruction register. Within the case expression is a group of control statements for each instruction in the processor's instruction set. The full algorithm is more than a page long; to give the flavour, only the Load instruction's control is shown in detail. It executes in three clock cycles. In the first cycle (state `st_load0`) the machine fetches the second word of the instruction and places it in the address register `ad`; the `pc` is incremented in parallel. Next, state `st_load1` computes the effective address by using the ALU to add the index register to `ad`, and the last state, `st_load2`, fetches a word from memory at the effective address and places it in the destination register within the register file.

```
st_load0:
  ad := mem[pc], pc++;
     {ctl_ma_pc, ctl_ad_ld,
      ctl_alu_abcd=1100,
      ctl_x_pc, ctl_pc_ld}
st_load1:
  ad := reg[ir_sa] + ad
     {set ctl_y_ad, set ctl_ad_ld,
      ctl_alu_abcd=0000}
st_load2:
  reg[ir_d] := mem[ad]
     {ctl_rf_ld}
```

6.3 Control Circuit

There are many ways to derive a control circuit from a control algorithm. Hydra supports several automatic control circuit synthesis algorithms [8]; a particularly simple one, called the delay element method, will be used here.

The control circuit has three inputs: a system reset signal, the contents of the instruction register and the condition bit (produced by the datapath). It outputs a word representing the control state, and a number of individual control signals. The circuit contains a flip flop corresponding to each state in the control algorithm. For example, if the algorithm is in state `st_instr_fet` then the flip flop with that name contains 1, and all the other state flip flops contain 0. The input to each state flip flop is connected to the previous state flip flop, so that a unique 1 value, meaning "I am in this state", moves through the state flip flops exactly as the locus of execution moves through the control algorithm. A demultiplexer is used in the `st_dispatch` state to determine which state to enter next. If the opcode in the instruction register is i, then bit i of `p` will be 1, and the control circuit enters the first state for that instruction.

```
      st_instr_fet = dff start
      st_dispatch  = dff st_instr_fet
      p = demux4w ir_op st_dispatch
      ...
-- Load has opcode 1
      st_load0  = dff (p!!1)
      st_load1  = dff st_load0
      st_load2  = dff st_load1
```

The control signals are generated by the logical or of the states in which that signal should be set. For example, the signal `ctl_rf_ld`, which causes the register file to load a value into one of the indexable registers, needs to be set when the processor is in any one of eight states:

```
ctl_rf_ld = orw
  [st_load2, st_ldval1,
   ...,
   st_cmpgt]
```

6.4 Simulation of the System

Simple circuits can be simulated in Hydra simply by applying the circuit directly to input signal values. For complex circuits, however, the large number of individual signals would make the input difficult to provide and the output difficult to decipher.

To address these problems, Hydra provides a set of tools for defining *simulation drivers*. These are functions that take inputs in a convenient form (e.g. decimal or hexadecimal numbers) and generate the corresponding circuit input signals, and similarly format the circuit outputs in order to make them readable. One advantage of basing a hardware description language on a full-scale programming language is that all the usual programming techniques are available for writing simulation drivers. The simulation drivers are merely an interface between the circuit behaviour functions and the user, but the actual simulation is performed by the stream recursion equations.

For the CPU design described above, the simulation driver performs some additional services: it takes the machine language program to be executed, generates the control signals needed to load it into memory via direct memory access I/O (DMA), it starts the machine, and it formats the various control and datapath outputs.

7 Related Work

Most CHDLs are based on existing programming languages. This enables large portions of standard language and compiler techniques to be reused, while still allowing for adaptations to the domain of hardware design. However, it is generally necessary to extend the underlying language, since some features needed to describe hardware cannot be expressed in typical imperative languages. VHDL [24, 28], currently the most widely used hardware description language in industry, is based on the Ada programming language. VHDL is highly expressive, although it is relatively difficult to reason about VHDL circuit specifications.

There has been widespread interest in hardware description languages based on mathematical models of circuits. Most of the resulting languages are based on relations, functions, or logic.

Ruby [15] is a language for specifying and reasoning about hardware, using a relational calculus to model circuit behaviour. The use of relations rather than functions to model circuits simplifies certain kinds of formal reasoning, and Ruby has been used extensively for circuit derivation and correctness proof [14]. Ruby requires all circuits to be specified using geometric combinators, and does not allow circuit topologies to be specified abstractly using named signals. Hydra offers both forms of specification [20, 22].

Several hardware description languages have been based on functional programming languages. Lustre [2] is a general stream processing language intended for specifying concurrent systems with synchronous communications, including hardware, and it also offers support for formal reasoning about circuits [26]. Lava [6, 4] is similar to Hydra, and also provides alternative circuit semantics via overloading. The main difference is that Lava introduces "observable sharing" for netlist generation [5]. Observable sharing was used in Hydra'87 [19], but was replaced in Hydra'92 [21] by a different method that does not interfere

with formal reasoning about circuits. Hawk [9] is another hardware description language based on Haskell. Hawk is similar to Hydra, also using stream simulation and type classes for multiple circuit semantics, but the emphasis has been on verifying hardware design at the register transfer level or higher [17, 7].

Mathematical logic, in particular higher order logic [10], has been used extensively to prove theorems about circuit behaviour [18, 1, 16]. The emphasis in this work is on the theorem proving, rather than on circuit design methodology, simulation, and other aspects of design.

8 Conclusion

Hydra is a hardware description language for designing synchronous digital circuits. It uses mathematical functions to model components and circuits, and provides alternative semantics to describe both the behaviour and the structure of a circuit using just one specification. This avoids potential errors caused by inconsistencies between the behaviour and structure of a circuit design.

Circuit specifications in Hydra are inherently concurrent. This helps in reasoning about timing of events in a circuit, and it is also useful for developing parallel circuit simulation algorithms.

Hydra supports a systematic methodology for practical circuit design, from the level of transistors up to complete systems. Its functional style of specification and powerful abstraction mechanisms allow the precise and complete specifications of large scale circuits using relatively small definitions. This paper briefly describes a RISC processor design; the complete specification is about 200 lines of Hydra code.

Acknowledgement. This is a revised version of a paper that appeared in *Proc. 16th Int. Parallel & Distributed Processing Symposium, PDSECA Workshop.* I would like to thank the anonymous referees for several useful suggestions.

References

[1] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs: Proceedings of the IFIP WG 10.2 Working Conference, Grenoble, September 1986*, pages 43–67. North-Holland, 1987.

[2] P. Caspi, N Halbwachs, D. Pilaud, and J. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages (POPL'87)*. ACM, 1987.

[3] Zhou Chaochen and C. A. R. Hoare. A model for synchronous switching circuits and its theory of correctness. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits*, pages 196–211. Springer-Verlag, 1991. DCC Workshop, Oxford 1990.

[4] Koen Claessen. *Embedded Languages for Describing and Verifying Hardware.* PhD thesis, Chalmers University of Technology, April 2001.

[5] Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Asian Computing Science Conference*. ACM SIGPLAN, 1999.

[6] Koen Claessen and Mary Sheeran. *A Tutorial on Lava: A Hardware Description and Verification System*. Chalmers University of Technology, April 2000. `www.cs.chalmers.se/~koen/Lava`.

[7] Byron Cook, John Launchbury, and John Matthews. Specifying superscalar microprocessors in Hawk. In *Formal Techniques for Hardware and Hardware-like Systems*, 1998.

[8] M. D. Edwards. *Automatic Logic Synthesis Techniques for Digital Systems*. MacMillan, 1992.

[9] John Launchbury et. al. Hawk, 1998. `www.cse.ogi.edu/PacSoft/Projects/Hawk/`.

[10] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[11] John B. Gosling. *Simulation in the Design of Digital Electronic Systems*. Cambridge University Press, 1993.

[12] L. J. Herbert. *Integrated Circuit Engineering: Establishing a Foundation*. Oxford University Press, 1996.

[13] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. MIT Press, 1984. The ACM Distinguished Dissertation Series.

[14] G. Jones and M. Sheeran. Designing arithmetic circuits by refinement in Ruby. In *Proc. Second Int. Conf. on Mathematics of Program Construction*, LNCS. Springer, 1992.

[15] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods in VLSI Design*, chapter 1, pages 13–70. North-Holland, 1990. IFIP WG 10.5 Lecture Notes.

[16] R. B. Jones, J. W. O'Leary, C.-J. H. Seger, M. D. Aagaard, and T. F. Melham. Practical formal verification in microprocessor design. *IEEE Design and Test of Computers*, 18(4):16–25, July/August 2001.

[17] John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in Hawk. In *Proceedings ICCL'98*. IEEE Press, 1998.

[18] T. Melham. *Higher Order Logic and Hardware Verification*, volume 31 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

[19] John O'Donnell. Hardware description with recursion equations. In *Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications*, pages 363–382, Amsterdam, April 1987. North-Holland.

[20] John O'Donnell. Hydra: hardware description in a functional language using recursion equations and high order combining forms. In G. J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 309–328, Amsterdam, 1988. North-Holland.

[21] John O'Donnell. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming, Glasgow 1992*, Workshops in Computing, pages 178–194. Springer-Verlag, 1992.

[22] John O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *FPLE'95: Symposium on Functional Programming Languages in Education*, volume 1022 of *LNCS*, pages 195–214. Springer-Verlag, 1995.

[23] John O'Donnell and Gudula Rünger. Derivation of a logarithmic time carry lookahead addition circuit. *Journal of Functional Programming*, 14(6):697–713, 2004.

[24] Douglas Perry. *VHDL*. McGraw-Hill Inc., 1991.

[25] Simon Peyton Jones. Haskell 87 language and libraries: the revised report. *Journal of Functional Programming*, 13(1), Jan. 2003.

[26] Ghislaine Thuau and Daniel Pilaud. Using the declarative language LUSTRE for circuit verification. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits*, pages 313–331. Springer-Verlag, 1991. DCC Workshop, Oxford 1990.

[27] P. W. Trinder, K. Hammond, H. W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[28] Sudhakar Yalamanchili. *Introductory VHDL: From Simulation to Synthesis*. Prentice Hall, 2001.