Connecting the Dots: Computer Systems Education using a Functional Hardware Description Language (Extended Abstract)

John T. O'Donnell
School of Computing Science
University of Glasgow
john.odonnell@glasgow.ac.uk

A functional hardware description language enables students to gain a working understanding of computer systems, and to see how the levels of abstraction fit together. By simulating circuits, digital design becomes a living topic, like programming, and not just a set of inert facts to memorise. Experiences gained from more than 20 years of teaching computer systems via functional programming are discussed.

1 Introduction

Since 1991, a variety of courses on computer systems in the Department (now the School) of Computing Science at the University of Glasgow have been presented by the author using a functional hardware description language. Approximately 30 one-semester courses over 20 years have used these methods. The language is called Hydra, and is currently implemented be embedding in Haskell. Development of the approach to teaching computer systems and research on the Hydra language have interacted strongly, and each has influenced the other. A paper discussing how an earlier version of Hydra was used in teaching computer architecture appeared in 1995 [2]. This paper gives an overview of the current approach, which has evolved significantly since then, and discusses the experience that has been gained.

A sign of maturity of functional programming is its quiet application to practical problem domains, including education, without too much emphasis on the functional language itself. The focus in this paper is on the challenges in understanding computer systems, and how these challenges can be met successfully. The emphasis is on the subject area, not on the language. A thesis of this paper, however, is that the aims of teaching computer hardware are supported very well by a functional hardware description language, and rather poorly by traditional imperative ones. The final result is not a claim that “functional is better”, but simply that students gain a deeper insight into computer systems.

Section 2 discusses the rationale and motivations behind this whole approach to teaching computer systems. Section 3 briefly describes how a functional language can be used to specify hardware, and describes some of the useful software tools that are available, but a complete and detailed description of Hydra is not given here. Section 4 summarises the main topics in the computer hardware courses, and gives some code snippets that illustrate how Hydra supports them. Section 5 draws some conclusions.

2 Rationale

Computer science education has traditionally focussed on programming and software development. It is common to provide some courses on computer systems, including an introduction to digital circuit
design and computer architecture, but these courses often lack the depth found in software engineering. Standard topics include an introduction to digital circuits, instruction set architecture, and basic processor organisation.

A major problem with typical courses on computer hardware is a failure to “connect the dots”: various details of each level of abstraction are covered, but the crucial connections between the levels are largely ignored. To understand computer systems properly, it is necessary to have a working understanding of each level of abstraction, and also to understand the connections between the levels.

What is meant here by a “working understanding” goes beyond just knowing a collection of facts (the names of the registers, the names of the techniques, the names of the events that occur). A working understanding means being able to hand-execute the system, as well as to design parts or all of it. A working understanding provides the foundation for quantitative assessment of performance.

In contrast, many courses and textbooks on computer systems just give descriptions of the hardware, without providing a working understanding. This is reflected in many textbook problems that ask for descriptive answers that can be culled from the text.

Here is an example of what a working understanding means. There is a technique in processor design called bypassing, which can speed up a processor by eliminating pipeline stalls and thus increasing the amount of parallelism that is achievable. Many textbooks describe bypassing, sometimes in considerable detail, but it is easy to puncture the veneer of understanding a student gets by asking some probing questions: *Here is a fragment of machine code; how many clock cycles faster will it run with bypassing?* *Modify a processor to implement bypassing (i.e. design absolutely every logic gate and signal needed to do it). What performance penalty, if any, is introduced by bypassing?* Unfortunately, there is not room in this paper to provide a working understanding of bypassing, but it can be done in a 20-lecture course using a powerful hardware description language—and Hydra is ideal for this purpose.

Consider a thought experiment. What if computer programming and software engineering were taught descriptively? There would be explanations of what it’s like to write a program, discussions of how to organise your thinking, and of course a mass of UML diagrams. But suppose the students never write a program, and never hand-execute any code. The obvious objection is that they would not be able to do real-world programming after such an education. But a further objection is that the students would not have a working understanding of programming: they would not understand conceptually what a computer is doing. (There are indeed some courses like this, with titles like “Computing for Poets”, but they are intended for non-computing students.)

For many computer science students, hardware topics are taught descriptively, and the students fail to gain a working understanding. Two reasons for this are:

- Computer systems are complex, with many levels of abstraction, and each level is a large subject on its own. There isn’t enough time to cover all this material.

- Hardware is seen as an ancillary topic; it’s just there to run software, which is the important part of computing. According to a cliché, computer science is not about computers, so presumably there is no need to find out how they work.

A response to these problems is:

- Although systems are complex, it isn’t necessary to cover all the details of all the levels. With suitable notation (this is where the functional language enters), the essential details can be covered clearly and concisely, providing a working understanding. In addition, we can achieve a working understanding of how each level of abstraction connects to the levels above and below it.
• Hardware is just as much about algorithms as software is. The qualitative difference between hardware and software is partly the result of ineffective languages for describing the hardware, which unnecessarily obscure the algorithmic content of the hardware.

We would not have computer science without computers, and surely it is reasonable for any educated computing professional to have a good working understanding of how a processor executes programs. Fortunately, this does not require excessive learning time.

3 Hydra: a functional hardware description language

The computer systems courses described here use Hydra [3], a functional hardware description language that is embedded in Haskell. Hydra supports many, though not all, aspects of computer hardware, including digital circuits, design methodology, processor organisation, and machine language programming. Several levels of abstraction are covered, including logic gates, combinational and sequential circuits, register transfer level, datapath and control, synthesis of control circuits from control algorithms, and machine language.

Hydra treats circuits as functions. A circuit specification is a function from input signals to output signals. This can obviously be done for combinational circuits (logic gates), but the simplicity and power of Hydra stem from the fact that all circuits—including sequential ones—can be modeled as pure functions. The insight that makes this possible was discovered by Steven D. Johnson [1], and is based on using streams to record the entire history of a signal value as a single denotational value.

A Hydra specification combines the structure and behaviour of a circuit in a single function definition. The system provides a number of alternative semantics, which are selected using type classes. By choosing a suitable signal representation, a specification can be executed, analysed, or a netlist generated, all by simply executing the specification.

The Hydra software consists of a library, an executable application with a GUI, and a set of examples. The library provides facilities for specifying and generating circuits, as well as extensive tools for performing simulations. The examples include a variety of useful circuits, as well as a complete computer architecture called Sigma16. The GUI provides an easy interface to using the tools.

The Sigma16 system is an extended example for Hydra, and is used as an example of basic techniques in computer systems for the courses. There have been extensive discussions in the literature about the tradeoffs between using real architectures or synthetic ones (like Sigma16), and also debates about using emulation vs. execution on real hardware. The full paper will explain our position, but in a nutshell an emulation of a synthetic architecture is advocated here.

There is an assembler for Sigma16, a loader, and emulator, and a GUI that makes it easy to use these tools. Figure 4.1 shows the GUI while the emulated processor is running an example program. There is a library of assembly language example programs for Sigma16.

There is also a complete digital circuit that implements the Sigma16 instruction set architecture. The circuit comprises logic gates and flip flops, and a separate memory component. The circuit is complete: absolutely every component and wire needed for the computer is present; thus a full understanding of the processor can be obtained by studying the complete circuit specification. And this is not too hard: the complete circuit specification is about 500 lines of code, many of which are comments.

The student can write a program in assembly language, and translate it to machine language using the assembler (or by hand). The machine language program can be executed in two ways: using the emulator, and by simulating the digital circuit with the program as input. These tools are very effective at connecting the dots between the levels of digital circuits and computer architectures.
All of Hydra is implemented in Haskell, including the library, the assembler and emulator, and the GUI.

4 Course topics

Hydra is used for teaching instruction set architecture in a required course for second year computing science students, and for teaching circuit design and computer architecture in an elective course for fourth year students.

Some snippets of materials used in these courses are given below, although they are minimised for this extended abstract.

4.1 Instruction set architecture

Both the elementary and advanced course use Sigma16 as an example architecture. This is a RISC style architecture modeled on the MIPS. It has 16-bit words, and 16 general registers. There are two instruction formats: RRR (where the three operands are all in registers), and RX (where one operand is in a register and the other is specified as a memory address with an index register).

Some of the RRR instructions are:

<table>
<thead>
<tr>
<th>op</th>
<th>mnemonic</th>
<th>operands</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>add</td>
<td>R1,R2,R3</td>
<td>R1 := R2+R3</td>
</tr>
<tr>
<td>1</td>
<td>sub</td>
<td>R1,R2,R3</td>
<td>R1 := R2-R3</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>cmplt</td>
<td>R1,R2,R3</td>
<td>R1 := R2&lt;R3</td>
</tr>
<tr>
<td>5</td>
<td>cmpeq</td>
<td>R1,R2,R3</td>
<td>R1 := R2=R3</td>
</tr>
<tr>
<td>6</td>
<td>cmpgt</td>
<td>R1,R2,R3</td>
<td>R1 := R2&gt;R3</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>trap</td>
<td>R1,R2,R3</td>
<td>trap interrupt</td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td>(expand to XX format)</td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td>(expand to RX format)</td>
</tr>
</tbody>
</table>

The RX instructions use an expanding opcode (and this useful architectural technique is explained fully in the courses—a working understanding is interesting and useful for anyone who needs to learn about compilers). Some of the RX instructions are:

<table>
<thead>
<tr>
<th>op</th>
<th>mnemonic</th>
<th>operands</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>lea</td>
<td>Rd,x[Ra]</td>
<td>Rd := x+Ra</td>
</tr>
<tr>
<td>f</td>
<td>load</td>
<td>Rd,x[Ra]</td>
<td>Rd := mem[x+Ra]</td>
</tr>
<tr>
<td>f</td>
<td>store</td>
<td>Rd,x[Ra]</td>
<td>mem[x+Ra] := Rd</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>jumpt</td>
<td>Rd,x[Ra]</td>
<td>if Rd/0 then pc := x+Ra</td>
</tr>
<tr>
<td>f</td>
<td>jal</td>
<td>Rd,x[Ra]</td>
<td>Rd := pc, pc := x+Ra</td>
</tr>
</tbody>
</table>

Many of the instructions are illustrated in the program ArrayMax, which demonstrates basic arithmetic, arrays, comparisons and conditional jumps, and loops. Figure 4.1 shows the GUI for the emulator as it is executing ArrayMax.

4.2 Datapath and control

An important concept in designing complex circuits is to partition the design into a datapath and control. The datapath contains the registers and the circuits that perform calculations. The datapath for Sigma16 includes the following equations, which define a set of registers and the output of the ALU:
Figure 1: Hydra GUI. The Processor tab is selected, and the machine is running the ArrayMax program.

One of the most interesting aspects of a datapath is that it provides a set of alternative potential operations that can be performed, and these must be supported by multiplexors with corresponding control signals. For example, the first data input to the ALU is called \( x \), and sometimes this should be the value of \( a \) (a readout from the register file) and sometimes it should be the \( pc \) (in order to increment the \( pc \)). To support this, we define \( x \) to be the output of a multiplexor that selects between \( a \) and \( pc \), and introduce a control signal to determine which value to use. The datapath contains a number of similar equations.

\[
\begin{align*}
(a,b) &= \text{regfile} \cdot n \cdot k \cdot \text{ctl_rf ld} \cdot \text{ir_d} \cdot \text{rf_sa} \cdot \text{rf_sb} \cdot p \\
\text{ir} &= \text{reg} \cdot n \cdot \text{ctl_ir ld memdat} \\
\text{pc} &= \text{reg} \cdot n \cdot \text{ctl_pc ld} \cdot r \\
\text{adr} &= \text{reg} \cdot n \cdot \text{ctl_ad ld} \cdot q \\
(\text{ovfl},r) &= \text{alu} \cdot n \cdot \text{ctl_alu op} \cdot x \cdot y
\end{align*}
\]

While the datapath provides potential operations, the control uses a collection of control signals to determine which operations should actually take place during the current clock cycle. There are about
20 control signals in the basic Sigma16 M1 circuit. A few of them are:

- `ctl_rf_ld` Load register file (if 0, remain unchanged)
- `ctl_x_pc` Transmit pc on x (if 0, transmit reg[sa])
- `ctl_y_ad` Transmit ad on y (if 0, transmit reg[sb])
- `ctl_rf_alu` Input to register file is ALU output (if 0, use m)

### 4.3 Control algorithm

Textbooks on circuit design often treat control as just another circuit, albeit a large and complex one. There are more sophisticated design methodologies that use Algorithmic State Machines (i.e. flowcharts) to describe the control. One of our conclusions, however, is that it is best to treat control as an algorithm that is expressed in a language, and then to develop systematic methods for synthesising the control circuit.

The control algorithm for Sigma16 is a state machine written in an imperative style. The top of the main loop repeatedly fetches the next instruction and decodes it:

```plaintext
repeat forever
   ir := mem[pc], pc++;
   case ir_op of
      ...
```

There is a separate case for each instruction; the case for the load instruction (1) fetches the second word of the instruction (the displacement), (2) calculates the effective address, and (3) fetches the data from memory and loads it into the destination register:

```plaintext
load:
   ad := mem[pc], pc++;
   ad := reg[ir_sa] + ad
   reg[ir_d] := mem[ad]
```

The control algorithm can be thought of as a program running on the datapath, which can be thought of as a programming language. In order to make the datapath perform the operations specified by the control algorithm, we need to figure out which control signals must be asserted. In general, this can be assisted by software tools, but for a student just learning how a processor works, it’s best to work this out by hand, at least for several instructions.

```plaintext
st_load0:
   ad := mem[pc], pc++;
   Assert [ctl_ma_pc, ctl_adr_ld, ctl_x_pc,
      ctl_alu_abcd=1100, ctl_pc_ld]

st_load1:
   ad := reg[ir_sa] + ad
   Assert [set ctl_y_ad, ctl_alu_abcd=0000,
      set ctl_adr_ld]

st_load2:
   reg[ir_d] := mem[ad]
   Assert [ctl_rf_ld]
```

### 4.4 Running programs on the circuit

The basic circuit (version M1) for the Sigma16 architecture can execute machine language programs, simply by simulating the circuit. However, the circuit contains many input and output signals, and
Clock cycle 67
Computer system inputs
reset=0 dma=0 dma_a=0000 dma_d=0000
cntl_start = 1

Control state
st_instr_fet = 0 st_dispatch = 0 st_add = 0 st_sub = 0
st_mul0 = 0 st_cmplt = 0 st_cmpq = 0 st_cmpgt = 0
st_trap0 = 0 st_lea0 = 0 st_leal = 0 st_load0 = 0
st_load1 = 0 st_load2 = 0 st_store0 = 0 st_store1 = 0
st_store2 = 0 st_jmp0 = 0 st_jmp1 = 0 st_jmp0f = 0
st_jmp1f = 1 st_jumpt0 = 0 st_jumpt1 = 0 st_jal0 = 0
st_jal1 = 0

Control signals
ctl_alu_a = 0 ctl_alu_b = 0 ctl_alu_c = 0 ctl_alu_d = 0
ctl_rf_ld = 0 ctl_rf_pc = 0 ctl_rf_alu = 0 ctl_rf_sd = 0
ctl_ir_ld = 0 ctl_pc_ld = 1 ctl_ad_ld = 0 ctl_ad_alu = 0
ctl_ma_pc = 0 ctl_x_pc = 0 ctl_y_ad = 1 ctl_sto = 0

Datapath
ir = f604 pc = 0010 ad = 0011 a = 0000 b = 0012 r = 0011
x = 0000 y = 0011 p = 0331 ma = 0011 md = 0000 cnd = 0

Memory
ctl_sto = 0 m_sto = 0
m_addr = 0011 m_real_addr = 11 m_data = 0000 m_out = 0331

Fetched displacement = 0011
jumpf instruction jumped
************************************************************************
Executed instruction: jumpf R6,0011[R0] effective address = 0011
jumped to 0011 in cycle 67
Processor state: pc = 0010 ir = f604 ad = 0011
************************************************************************

Figure 2: Simulation output.

thousands of internal signals. It would be impossible to figure out what is going on by looking at a bit-level simulator.

Hydra contains a sublanguage for expressing simulation drivers (also called testbenches). This is a piece of software that accepts input from the user in a readable format, converts it to the bit signal representations, connects the input signals to the circuit, executes the circuit (thereby simulating it), monitors the circuit’s output signals, converts their values to a readable form, and prints that out. Figure 4.4 shows the output from the simulation driver, as the circuit is on clock cycle 67 while executing the ArrayMax program.

Sometimes it is hard to see what is going on, even looking at the values of registers and signals. Therefore, the simulation driver language maintains a state and provides tools that allow the driver to observe signal values and record partial information as it goes. The simulation driver for Sigma16-M1 watches the output signals from the circuit, collects information, and uses that to print an informative message when a major event (such as the execution of an instruction) occurs. The last few lines of the figure show a message indicating that a jumpf instruction has just executed.
4.5 Advanced techniques in processor design

The M1 circuit for Sigma16 takes the simplest approach to solve all the problems needed to execute computer programs. It takes several hours to understand, but average students in the computer architecture course really do understand it. Furthermore, they develop a working understanding, and demonstrate this in exercises that involve modifications (adding new instructions, implementing interrupts, etc.).

There are also circuits that introduce more advanced techniques, including pipelining and superscalar execution. And, with these powerful design techniques, it is quite straightforward to implement different instruction set architectures.

5 Conclusion

The full paper will discuss the conclusions in more detail, but here are some brief observations.

- Any hardware description language will look like software to weaker students, who will not understand the distinction between the circuit and the software notations used to specify the circuit. This happens because of inadequate thinking about abstraction, not as a result of the choice of a functional or imperative hardware description language.
- It is important to focus on circuit models, not just a language. The work described above is all based on a single-clock pure synchronous model (this will be discussed in the full paper). Some other circuit models can be handled by Hydra, but not all.
- Simulation helps to get a working understanding of what a processor is doing.
- The real benefits from learning about circuits and computer architecture come from connecting the concepts, not just learning them in isolation.

References

