Although this work was published in somewhat different form many years ago, I am reissuing it as a Baylor Computer Science technical report in an effort to keep everything in the same place.

The FHDL Rom Tools

Peter M. Maurer
Department of Computer Science
Baylor University
Waco, TX 76798

Abstract

The FHDL (Functional Hardware Design Language) ROM tools provide a method for specifying, simulating, and automatically laying out ROMs. The primary focus of the ROM tools is on providing powerful methods for specifying microcode. Because the ROM tools were designed to support both VLSI design projects and other course work in hardware design, the ROM language contains many features that allow it to emulate other ROM programming languages. This allows students to complete laboratory exercises using a language that is similar to the one used in their textbook. Once the contents of a ROM have been specified, the ROM can be simulated concurrently with the simulation of the other hardware comprising the design. This allows designs to be debugged before they are fabricated. Once a design has been completely verified, the ROM can be laid out automatically and incorporated into a larger VLSI circuit.

The FHDL Rom Tools

Peter M. Maurer
Department of Computer Science
Baylor University
Waco, TX 76798

1 Introduction

The Read-Only-Memory or ROM is a fundamental part of many of today's computer architectures[1,2,3]. In addition to their use in microcoding, they can be used to store tables of constants, and other data that is required for special-purpose applications. The ROM tools described in this paper were developed as part of a much larger scale project called the Functional Hardware Design Language (FHDL)[4]. The FHDL system of tools was designed to be a general-purpose simulation and synthesis system that could be used to construct complex designs, simulate them to verify their correctness, and automatically lay them out. The specification and simulation portions of the tools are particularly important, because they permit students to experiment with complex designs without the expense and frustration of building actual hardware.

Since it was our aim to create a uniform specification system in which one could specify all portions of a design in a common format, it was necessary to include features for specifying the contents of ROMs. One approach to handling ROMs would have been to reduce the ROM code to a collection of gates, and use the gate-level features of FHDL to simulate and synthesize them. The flaw in this approach is that it ignores the fact that ROMs have a well understood structure both at the simulation level and at the layout level. Because the structure of a ROM is well understood, one typically does not need the detail of a gate-level simulation to verify the correctness of a ROM. The correctness of a ROM can be adequately verified by simulating the ROM at a high level. Well understood algorithms can then be used to create the layout of the ROM without the danger of introducing small-scale errors. This is precisely the approach taken by the FHDL ROM tools.

The motivation for developing the FHDL ROM tools was to provide support for for courses in computer architecture and sequential circuits. For IC design projects one could reasonably expect researchers to be flexible enough to adapt their design style to the peculiarities of the tools, particularly when the tools are considerably more efficient than manual design methods. However in supporting course work, one typically does not have this flexibility. To simplify the presentation of the material, textbook authors typically present microcode in terms of some microcoding language. (See, for example [5].) To avoid confusing students, it is necessary to provide a language that is either identical to or a close approximation of the language presented in the textbook. The FHDL ROM compiler provides several features that allow the language to be customized to resemble other ROM programming languages. There are, of course, languages that cannot be approximated in any reasonable way by these tools, but the existing features provide methods for handling most languages that we have encountered in the classroom.

2 The ROM Language.

ROM statements are broken into three main categories, those that define constants and complex commands, those that perform formatting, and those that describe the contents of ROM words. The constant-definition mechanism is similar to that found in many assembly languages. Equate statements are used to assign expression values to symbols. Complex commands are defined using a similar mechanism which will be described more fully below. Most formatting is done using "field" statements which describe the width, position, and characteristics of various portions of the ROM word. The "word" statement is used to define the contents of a single ROM word. For some ROMs the constant and format definitions can become quite lengthy, so the ROM compiler allows these statements to be placed in a separate file and included in each ROM description to which they are applicable. (This feature is particularly useful for emulating other ROM programming languages.) Figure 1 contains an example of a simple ROM description.

```
Out:        field   position=0,width=16,cmdpos=0
Zero:       word    0
One:        word    1
Two:        word    2
Minus_one:  word    0xFFFF
```

Figure 1. A Simple ROM Description.

Figure 1 illustrates the description of a ROM containing the constants 0, 1, 2, and -1. Each word contains a single 16-bit field. The "cmdpos" parameter on the field statement indicates that the value of the field will be supplied by the first parameter on each word statement. In this case the label on the field statement is optional, but in other cases the name of the field can be used to assign an explicit value to the field, using an expression of the form "12->Out" as an operand on a "word" statement. The labels on the "word" statements are also optional, but if they are used in an expression, their value is equal to the address of the word generated by the "word" statement. By default, addresses are assigned sequentially starting from zero, but the user can alter this assignment by using an "org" statement similar to that provided by most assembly languages.

Although the "field" and "word" statements provide the basic mechanism necessary to define the contents of a ROM, there are several types of ROM coding that are difficult or cumbersome without additional compiler support. For example, many ROM programming languages that are presented in textbooks are modeled after more conventional assembly languages. The commands used to describe the contents of a word are in the form of instructions with meaningful operation codes such as "jump" or "skip." In some cases the format of the ROM word depends on the operation code. For example, the ROM address field used by a "jump" instruction might be used for an entirely different purpose in some other instruction. In many cases the syntax of an instruction will depend on the operation code. For example, a jump command might require a single operand to specify the address of the target ROM word while a conditional jump might require two operands, one to specify the condition and one to specify the target.

To solve these problems in a uniform way, an integrated mechanism was designed that allows the user to define new operation codes, define multiple word-formats, associate operation codes with a particular format, and associate fields with the values of the operation codes. To

illustrate how this mechanism is used, assume that it is necessary to construct a ROM using the two word-formats illustrated in Figure 2.

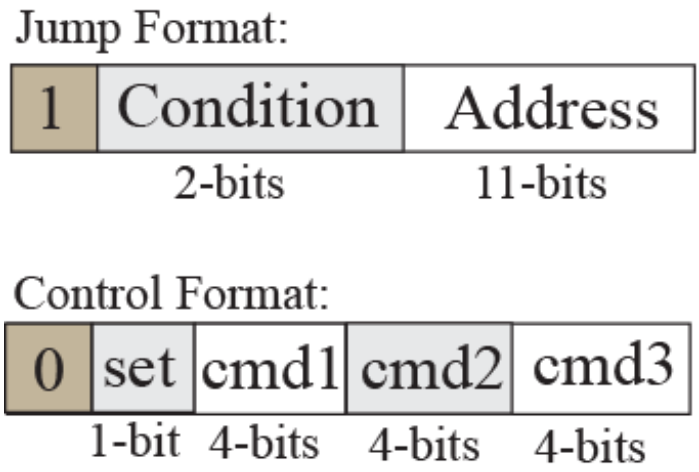

Figure 2. Sample Word-Formats.

Figure 2 illustrates two different word-formats that will be used in the same ROM. The first bit of each word determines the format. In the "jump" format there are two fields, a condition and an address. The condition is coded as follows: 00-jump always, 01-don't jump (noop), 10-jump on overflow, 11-jump on carry-out. The second format contains three horizontally-coded control commands. The commands are interpreted differently depending on the value of the second bit. The bit is set to zero to select a set of commands for performing arithmetic functions, while it is set to one to select a set of commands for performing control functions. To program this ROM we will define six operation codes: jump, noop, jovfl, jcarry, arithmetic, and control. The definitions for these operation codes appear in Figure 3.

```
jump:       equ     0
noop:       equ     1
jovfl:      equ     2
jcarry:     equ     3
arithmetic: equ     0
control:    equ     1
```

Figure 3. The definition of operation codes.

Next, each of the operation codes is assigned to a format as illustrated in Figure 4.

```
jfmt:   format  jump,noop,jovfl,jcarry
cfmt:   format  arithmetic,control
```

Figure 4. Assigning operation codes to formats.

Finally, the fields of each format are defined as illustrated in Figure 5.

```
typej:      field   position=0,type=constant,default=1,format=jfmt
condition:  field   position=1,width=2,type=opcode,format=jfmt
address:    field   position=3,width=11,cmdpos=0,format=jfmt

typec:      field   position=0,type=constant,default=0,format=cfmt
set:        field   position=1,type=opcode,format=cfmt
cmd1:       field   position=2,width=4,format=cfmt
cmd2:       field   position=6,width=4,format=cfmt
cmd3:       field   position=10,width=4,format=cfmt
```

Figure 5. Field definitions.

Note that in Figure 5, each field definition contains a reference to a format. The first field in each format is constant and cannot be changed by the user. This field defines the format of the word to the hardware. The field "condition" of format "jfmt" and the field "set" of format "cfmt" are both of type "opcode." This implies that the value of the operation code as defined in Figure 3, will be placed in the field automatically. The "address" field of format "jfmt" contains the parameter "cmdpos=0" which implies that the value of the field will be supplied by the first operand of the microinstruction. The fields "cmd1" through "cmd3" must be explicitly assigned values by the micro-instruction. (A default value of zero is used for any unspecified fields. The field definition may supply a different default value, as illustrated by the definition of the field "typej" in Figure 5.)

Once these definitions are complete, one can proceed to create microcode for the ROM. A considerable amount of external logic is required for this microcode to function correctly, but this is beyond the scope of this paper. Figure 6 illustrates some microcode that uses the definitions of Figures 3, 4 and 5.

```
Add:    control     12->cmd1,15->cmd2,0->cmd3
        arithmetic  10->cmd1,0->cmd2,6->cmd3
        jovfl       OverFlow
        jcarry      Negative
        jump        NewOperation
```

Figure 6. Sample Microcode.

The main problem with the microcode in Figure 6, is that the function of the first two lines is not apparent from the specification of the operands. The readability of this microcode can be greatly improved by adding definitions for constants and complex commands. Suppose that part of the hardware being controlled by this microcode is that pictured in Figure 7.

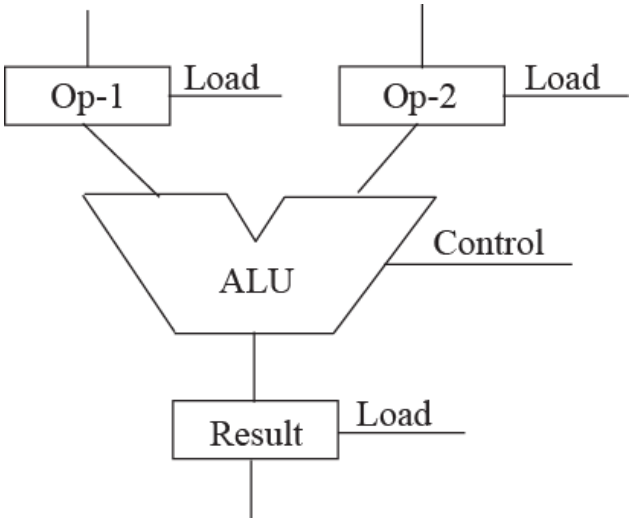

Figure 7. Sample Hardware.

Suppose that the load signals of the operand registers Op-1 and Op-2 pictured in Figure 7 are activated by the expressions 12->cmd1 and 15->cmd2 pictured in Figure 6. Furthermore, suppose that the load signal of the result register is activated by the expression 10->cmd1 and that the ALU addition operation is selected by the expression 6->cmd3. The first step is to provide definitions for the constants 12, 15, 10, and 6. However, the two sets of expressions that appear in Figure 6 perform two specific operations. To make the microcode as readable as possible, it is desirable to collect these expressions together into a single command that can be used to perform the desired function. Figure 8 illustrates how this is done.

```
Load1:          equ         12
Load2:          equ         15
LoadR:          equ         10
SelAdd:         equ         6
None:           equ         0
LoadOperands:   command     Load1->cmd1,Load2->cmd2,None->cmd3
DoAddition:     command     LoadR->cmd1,SelAdd->cmd3,None->cmd1
Add:            control     LoadOperands
                arithmetic  DoAddition
                jovfl       OverFlow
                jcarry      Negative
                jump        NewOperation
```

Figure 8. Sample Microcode with Improved Readability.

3 More Complex Addressing Schemes.

One method of performing conditional branching within a microprogram is illustrated by the microcode of Figures 6 and 8. Another popular method for performing conditional branches is to allow one or two low-order bits of the microinstruction address to be supplied by control signals from the hardware. This presents a particular problem for microcoding languages, because a

"jump" microinstruction contains only a portion of the target address. It is necessary to be able to specify *which* portion of the address is to be assigned to the address field.

To illustrate how this problem is solved in the FHDL ROM language, consider the word format pictured in Figure 9, which is similar to the word format used in some models of the IBM 360[6].

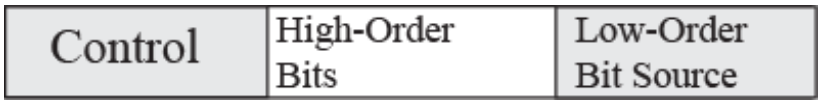

Figure 9. A Complex Word Format.

The format pictured in Figure 9 is used for all words in the ROM. Every word specifies the address of the next word. The field labeled "High-Order Bits" contains all bits of the address except the low-order bit. The field labeled "Low-Order Bit Source" is a multi-bit field that determines the source of the low-order address bit. For the purpose of this example, we will assume that the Low-Order Bit Source is a two bit field that has the following meanings. A value of 00 implies that the low-order bit is zero, while a value of 01 implies that the value of the low-order bit is 1. A value of 10 implies that the low order bit is equal to the "carry-out" of the ALU, while a value of 11 implies that the low order bit is equal to the overflow output of the ALU.

The first problem is determining how to specify the value of the "High-Order Bits" field. If we assume that addresses are eleven bits long, then the High-Order Bits field must be ten bits long. However, if this field is simply declared as a ten-bit field, it is the ten *low-order* bits of the address that will be placed in the field. The "bitpos" parameter of the "field" statement can be used to override this behavior. The source data for any field is a sequence of bits that are numbered sequentially from zero starting at the right. By default, the rightmost bits of the source are assigned to the field, however when the "bitpos" parameter is specified, the transfer of data into the field begins with the source-bit specified by the "bitpos" parameter. Therefore, the High-Order Bits field must be specified with a "bitpos" parameter of 10, as illustrated in Figure 10.

The next problem is relieving the user of the responsibility of specifying the next address for microinstructions that are executed sequentially. This can be done by specifying a default value for the address field that specifies that the next address is to be used if none is specified by the microinstruction. Figure 10 illustrates how this is done.

```
High:   field   position=x,width=10,bitpos=10,default=*+1
Low1:   field   position=x+10,width=1,default=0
Low2:   field   position=x+11,width=1,default=*+1
```

Figure 10. Field Definitions for Sequential Addressing.

In Figure 10, the default value for the fields labeled "High" and "Low2" contain the symbol "*," which represents the address of the current microinstruction. When such an expression appears as the default value for a field, it is evaluated for every microinstruction containing the field. The field "High" will receive the ten high-order bits of the next microinstruction, while the field "Low2" will receive the low-order bit of the next microinstruction.

Even though in this example there is a single format for all microinstructions, it is still convenient to define several formats for the purpose of distinguishing between sequentially executed microinstructions and jump microinstructions. In fact, it is best to define three formats, one for sequential microinstructions, one for unconditional jumps, and one for conditional jumps. (It is possible to declare a field to be common to several formats, so it will not be necessary to redefine the "Control" fields for each format.) Figure 11 illustrates how the address fields would be defined for unconditional jumps. The format for conditional jumps can be defined in a manner similar to that presented in Section 2.

```
JHigh:    field   position=x,width=10,bitpos=10,cmdpos=0
JLow1:    field   position=x+10,width=1,default=0
JLow2:    field   position=x+11,width=1,cmdpos=0
```

Figure 11. Field Definitions for Unconditional Jumps.

4 Simulation.

Once the content of a ROM has been specified, simulation is quite simple. The simulation of the ROM is scheduled as if it were an ordinary gate. The content of the ROM is stored in one or more arrays, depending on the width of each ROM word. The maximum array width supported by the simulator is 32-bits, but several arrays can be combined automatically to simulate ROMs of greater widths. Simulation of the ROM is performed in conjunction with the simulation of the ROMs sequencer and the circuitry under its control. It is the responsibility of the control circuitry to supply the next address and to execute the microinstructions. Simulation of the ROM involves extracting the appropriate words from the arrays, and reformatting the data into the variables that represent the outputs of the ROM.

The ability to simulate a ROM at the high level in conjunction with a more detailed simulation of the external logic gives an efficient method for debugging microcode prior to creating the layout of a circuit.

5 Conclusion

The FHDL ROM language provides a powerful method for defining, simulating, and automatically laying out ROMs. The ROM tools can be used in conjunction with other FHDL tools to create a unified simulator for all portions of a circuit. Furthermore, the ROM tools can be used to significantly speed up the process of creating the layout of a ROM. One of the most attractive features of the FHDL ROM language is the ability to emulate other ROM programming languages. This allows courses in ROM-based sequential circuits to use a unified approach in both classroom work and laboratory exercises. The ability to test designs through simulation rather than fabrication of hardware allows more laboratory exercises to be assigned, since the time and expense of hardware fabrication is eliminated. Even when a design is fabricated, pre-verification of the design through simulation can greatly reduce the amount of time required to complete the design, because the detection of design flaws can be done before fabrication of the design is attempted.

Future work on the ROM tools will extend the scope of the language to allow more types of ROM programming languages to be emulated. Although the ROM language currently supports most existing microcoding schemes, it is possible that some new scheme will require enhancements to the language.

6 REFERENCES

1. M. V. Wilkes, "The Best Way to Design an Automatic Calculating Machine," *Manchester University Computer Inaugural Conference*, Ferranti Ltd., London, 1951, pp. 16-21.

2. A. K. Agrawala, T. G. Rauscher, *Foundations of Microprogramming*, Academic Press, Inc., New York, 1976.

3. M. Shoji, *CMOS Digital Circuit Technology*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

4. P. M. Maurer, "The Functional Hardware Design Language," Baylor Computer Science Technical Report, http://hdl.handle.net/2104/5444, Oct 28, 2009.

5. S. Habib, *Microprogramming and Firmware Engineering Methods*, Van Nostrand Reinhold, New York, 1988.

6. S. G. Tucker, "Microprogram Control for System/360," *IBM Systems Journal*, Vol. 6, No. 4, 1967, pp. 222-241.

7. C. Mead, L. Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing Company, Reading, Mass, 1980.