Agile Hardware Development with Rapid Hardware Definition Language

Jacob N. Allen  
Student Member, IEEE  
Microelectronics System Design Lab  
Dept. of Elect. and Comp. Engineering  
Oakland University  
Rochester Hills, Michigan 48309

Dr. Hoda S. Abdel-Aty-Zohdy  
Member, IEEE  
Microelectronics System Design Lab  
Dept. of Elect. and Comp. Engineering  
Oakland University  
Rochester Hills, Michigan 48309

Dr. Robert L. Ewing  
Member, IEEE  
Sensors Directorate  
Air Force Research Labs  
Wright-Patterson Air Force Base  
Ohio 45431

Abstract—Rapid HDL is a hardware definition language developed in C# for the Microsoft .Net platform. Rapid HDL unifies digital FPGA hardware development and co-software integration. Hardware is scripted using reusable software objects, communication between hardware and software is automatic, and synthesis is automated using a free tool chain. Rapid HDL represents 18,000 lines of code and is freely available under the MIT license.

I. INTRODUCTION

Rapid HDL is a software library for creating hardware using agile software programming techniques. The library was written out of pure necessity to speed up development of hardware necessary to support our various research projects. To aid other researchers, Rapid HDL is now freely published under the MIT license. Rapid HDL was first introduced three years ago, and has evolved. This paper provides an abbreviated summary of its structure, and serves as an introduction to anybody interested in working with Rapid HDL.

Rapid HDL is certainly not the first software library created to generate hardware, and perhaps [1] holds that title. Table I provides a summary of related projects and shows how Rapid HDL compares, according to the author’s admittedly unscientific opinion. When considering HDL systems, we must distinguish between (a) high level software languages that are synthesized from their native logic directly into hardware, and (b) software libraries of hardware components that are manipulated and combined by a high level language to create hardware. In the first category, we have systems like HandleC, ImpulseC, and Matlab’s Simulink toolbox, where functional code is compiled directly into hardware implementations. In the second category, we have libraries like JHDL, Viva, and Rapid HDL, where the construction of hardware is scripted from a high level language.

Rapid HDL is designed to make hardware development like agile software development, with emphasis on hardware/software co-development. Hardware created in Rapid HDL is built using object oriented libraries, using the same computer language as the co-developed software, featuring:

- Communication between software and hardware is seamless and versatile, after the code for a generic hardware interface is developed.
- Test benching and test scripts are innate in the language, allowing for hardware test suites to be developed.
- Rapid HDL is a library that runs on the Common Language Runtime, so hardware can be constructed using any language supported by Microsoft .Net, including C#, VB.Net, Ruby, F#, Cobal, and others.
- Rapid HDL is released under the non-restrictive MIT license, so all types of users are free to make changes without being troubled by a requirement to release their code under the GPL.
- Rapid HDL is tightly integrated with the hardware synthesis chain. Synthesis can take place remotely via web services in the internet cloud.
- Rapid HDL makes agile programming techniques a reality for FPGA based hardware design.

II. RAPID HDL

Rapid HDL is an object oriented library for generating Verilog HDL. It also creates a seamless interface between hardware and software, making it easy for the software to interact with the hardware. Ease of building, testing, debugging, and deployment are also goals of Rapid HDL. Finally, Rapid HDL is integrated with the free Xilinx Webpack synthesis tools, making the hardware development cycle fluid, like software development.

A very early version of Rapid HDL was discussed in a previous paper in depth [2]. As the paper points out, a major strength of Rapid HDL is the ability to programmatically make connections between object oriented hardware components without worrying about hardware component hierarchy. Wires are automatically routed through the hierarchy when the final Verilog is generated. Freedom from manually tracking hardware hierarchy is advantageous when arbitrarily passing objects as parameters, which is typical in object oriented programming. Further, debugging is greatly simplified, because a signal can be called out and viewed from any component.

A. Rapid HDL Organization

Basic objects in Rapid HDL include the node, the node vector, and the component. Nodes represent wire connections;
<table>
<thead>
<tr>
<th>HDL Language</th>
<th>Stage of Maturity</th>
<th>Intellectual Property</th>
<th>Software Integration</th>
<th>Modern Software Language</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rapid HDL</td>
<td>adolescent</td>
<td>MIT license</td>
<td>unified</td>
<td>yes</td>
<td>Verilog</td>
<td>Microsoft .Net library for hardware generation and software integration [2]</td>
</tr>
<tr>
<td>VHDL &amp; Verilog</td>
<td>mature</td>
<td>public domain or</td>
<td>no</td>
<td>no</td>
<td>VHDL or</td>
<td>most used hardware definition languages</td>
</tr>
<tr>
<td>SystemC</td>
<td>mature</td>
<td>open source or</td>
<td>Integrated</td>
<td>no</td>
<td>FPGA level</td>
<td>IEEE standard C++ hardware description language [6]</td>
</tr>
<tr>
<td>JHDL</td>
<td>mature</td>
<td>GPL</td>
<td>Integrated</td>
<td>yes</td>
<td>FPGA level</td>
<td>Java based library, targets specific FPGA’s [10]</td>
</tr>
<tr>
<td>Viva</td>
<td>mature</td>
<td>commercial</td>
<td>not tightly</td>
<td>N/A</td>
<td>FPGA level</td>
<td>Graphical and Component Oriented Design</td>
</tr>
<tr>
<td>Ruby HDL</td>
<td>adolescent</td>
<td>GPL</td>
<td>Integrated</td>
<td>yes</td>
<td>simulation only</td>
<td>Simulation Library in Ruby [4]</td>
</tr>
<tr>
<td>HandleC</td>
<td>adolescent</td>
<td>commercial</td>
<td>Integrated</td>
<td>no</td>
<td>FPGA level</td>
<td>Generate Hardware from C Code [7]</td>
</tr>
<tr>
<td>ImpulseC</td>
<td>mature</td>
<td>commercial</td>
<td>integrated</td>
<td>no</td>
<td>VHDL</td>
<td>Auto-Generate Hardware from C Code [8]</td>
</tr>
</tbody>
</table>

and node vectors represent a collection of one or more nodes. Components are the base hardware object, and can be either (a) defined as a truth table that maps sink nodes to source nodes, or (b) as a container that combines and connects several other components.

The code for an arbitrarily sized multiplexor is shown below, in Figure 1. The code demonstrates some basic ideas. In the component constructor, the component connections are defined as NodeVectors. Note that the input width, and the number of inputs to the mux are arbitrary.

Figure 2 shows the mux component being used in the pipeline serializer component. The pipeline serializer decomposes a large packet into several small packets, adding a sequence address to each smaller packet. In the example, a mux is used to decompose the large packet and route the smaller packets into a single, smaller output. Note that the mux is dynamically defined and connected in reaction to the specific pipeline serializer initialization parameters. This type of flexibility is not possible in Verilog, and demonstrates Rapid HDL’s ability to generate fluid hardware designs.

### B. Rapid HDL High Level Components

After a solid library of base components was created in Rapid HDL, hardware generation of high-level function components became possible. Two of three components important to hardware software system implementation will be highlighted here, namely the pipeline and the state machine. Rapid HDL also includes a micro-controller fabric which will not be discussed here.

1) Pipeline: The basic idea of the pipeline component is shown in Figure 3. The pipeline stage combines a data bus with two synchronization data bits and is used to route and transfer data packets between cores. A stall signal propagates backwards, and signals to a packet producer whenever a pipeline is full. A data ready signal propagates forward, and signals to a packet consumer that the data bus contains valid data which needs to be serviced.

Pipelines can then be combined in several configurations to create data flow routes.

Important pipeline components are listed in Table II.

2) State Machine: Using a high level software language, state machines are easily defined. Figure 4 shows the state machine definition for a pipeline assembler. In the example shown, we see that the machine transitions from “state_dr” to “state_jump” only when the “nvJump” NodeVector is active. We can also see that the “nvJump” NodeVector goes high when nvInstBitHigh is high, and nvInstBitLow is low. Finally, we note that the nvEnable nodevector requires that the state machine’s state be “state_dr”, and at least one of the nvControl or nvDone nodes be high.

### III. FPGA System

The Pico E-12 FPGA is used to implement this project [5]. The advantages of the Pico E-12 include: (a) open-source drivers for integration with host-PC, (b) a relatively fast PCMCIA bus to communicate with the host-PC, (c) a small,
Fig. 1. Constructor for a Multiplexor

```java
public class Mux : Component
{
    public NodeVector SelectNodes;
    public NodeVector[] InputNodes;
    public NodeVector OutputNodes;

    public Mux(Component pParentComponent, string pName, int pWidth, int pInputCount)
    : base(pParentComponent, pName)
    {
        int iSelWidth = Conversion.ToIntBinaryString(pInputCount - 1).Length;
        OutputNodes = this.CreateNodeVector("DATA_O", pSelWidth, NodeFlowType.Source);
        InputNodes = new NodeVector[pInputCount];
        for (int iIdx = 0; iIdx < pInputCount; iIdx++)
            InputNodes[iIdx] = new NodeVector(this, "DATA_" + iIdx.ToString() + ",", pWidth);
        SelectNodes = this.CreateNodeVector("SEL_I", iSelWidth, NodeFlowType.Source);
    }
}
```

Fig. 2. Partial code for PipelineSerializer component (that consumes the mux component)

3) Mobile SDRAM: Memory access proved to be a time consuming problem. A basic SDRAM controller for the Samsung Mobile SDRAM was developed in Verilog and then integrated into the firmware. The SDRAM controller was tested using the Micron MT48H16M16LF SDRAM test chip.

Since our ultimate goal is to create an ASIC, the Power PC was not used. Also, only one PPC would be available, although we require several processor cores for our research application. Finally, we do not want memory access to be monopolized by the Power PC, so our final pipeline architecture allows memory access that can be shared by several processor cores implemented in Rapid HDL.

Relevant specifications for the Pico E12:
- PCMCIA interface.
- Uses a Xilinx Virtex 4 FX-12 FPGA
- Includes 128MB of PC-133 SDRAM memory
- Verilog firmware and Windows XP drivers
- Operates at 200 MHz and 100 MHz
- About 1 Mbps data transfer rate between host PC and FPGA
- $900 price

The Virtex 4 FPGA includes a Power PC micro-processor.
models, which are a virtual clone of the Samsung memory in the Pico card. Presently, a memory read or write consumes a minimum of 16 clock cycles using the current memory controller and data routing system. To speed up memory access, a better memory controller should be implemented.

**A. Rapid HDL Firmware**

Rapid HDL uses firmware written in Verilog. This firmware is unique for each FPGA package. We include the FPGA firmware and RHDL interface for the Pico E-12 FPGA package. With the appropriate firmware created, Rapid HDL generated hardware is very tightly integrated with software. Rapid HDL can single step the hardware, one clock cycle at a time. Software inputs and outputs connected to the FPGA hardware are automatically created when the hardware is scripted using the syntax shown in Figure 6.

When the hardware is running, signal views and signal sources can be changed in software just as if they are a software variable. Additionally, two FIFO’s are available in the software to interact with the hardware. One FIFO sends 64 bits of data to the FPGA, and another FIFO receives 42 bits of data from the FPGA. The FIFO is used, for example, to communicate with the Pico’s SDRAM memory, and the FIFO is also used to interact with the developed hardware.

Synthesizing and debugging hardware is typically a time-consuming task. Rapid HDL includes a mechanism to synthesize and build hardware on a faster, remote computer. When building hardware, generated Verilog and the base firmware are packaged in a compressed file and optionally transferred over the internet to a build server (in Amazon’s Compute...
SignalView oSignalView = ContainerBlock.AddSignalView("A", "OUT", SignalFormat.Unsigned);

oSignalView.SignalFormat = SignalFormat.Unsigned;

Fig. 6. Example of how signals can be routed from Hardware to Software

Cloud), as shown in Figure 7. The build server utilizes a WebService to invoke the Xilinx synthesis tool-chain. When the Xilinx tool-chain completes, the logs and the FPGA bit-file are returned to the Rapid HDL application. The bit file is then automatically loaded onto the FPGA.

IV. RAPID HDL USER INTERFACE

A Rapid HDL workbench is shown in Figure 8.

A – Buttons control the hardware execution through software. It is possible to step by a number of clock cycles, start the clock running without interruption, synthesize hardware, or execute a test script. These functions are also easy to call from within any software/hardware interface.

B – Any signal source or signal view defined in the hardware is accessible and viewable from the hardware tree. Changes to the signal sources are reflected immediately in the signal views. Signal sources update as the clock is stepped.

C – The tree allows drill down into hardware hierarchy. The tree view includes the ability to view the Verilog that was generated for each component. Also, the properties of signals and child components in the tree view are manifest.

D – The transcript window shows the results of FPGA logs and hardware interaction. Here, for example, it shows the results of test scripts. This window may also be used to show Verilog output. When debugging a microprocessor core, the assembly code is displayed, and allows a debug trace through assembler code, line by line.

E – The trace option shows the assembler code in a selected microprocessor code. The MicroStep button advances the microprocessor one instruction at a time.

F – Various buttons can be added to the test bench and used for testing the chip. These buttons are not required, and generally customized for each hardware type.

G – The ‘Properties’ window shows the details of RHDL components, signal sources, and signal views. Signal source values can be manually entered here and the change is immediately posted to the hardware.

V. CONCLUSION

In summary, Rapid HDL is an object oriented library created in a high level, modern, software language to create the complex hardware quickly, through software techniques. Rapid HDL makes complicated hardware for our research purposes possible, and provides a foundation to modify or upgrade the resulting hardware. The source program is about 18,000 lines of code (not counting comments, computer generated code, or blank spaces) and available under an open source license. Rapid HDL makes hardware development more intuitive, like software development, and thereby makes it easier to develop complex yet flexible hardware required for a hardware/software application.


ACKNOWLEDGMENT

This work is partially supported by a research grant #FA8650-04-2-4201, No. RSC04025, Task 1 from the AFRL (2007-2009). Views and conclusions herein are those of the authors, and not necessarily the views of the US Air Force or the US government.

REFERENCES


Fig. 8. Rapid HDL Workbench Interface