Dynamically Configured Java Based Register Windows For Efficient Simulation Debug

Dan Helm

ARM, Inc.
Austin, USA

www.arm.com

ABSTRACT

The use of a Verdi Register Window is a powerful means of visually representing data/control as it moves through a system, aiding in debugging efficiency. With the increase in both size and configurability of today's systems, maintaining a library of corresponding Register Window files has a significant overhead. This paper will present a method of using a custom Java GUI with an NPI backend running in conjunction with Verdi as a dynamic replacement for a Register Window.

Table of Contents

Table of Figures

1. Introduction

The Verdi Register Window provides a graphical means of displaying simulation results, allowing the user to display information in a way that is relevant to their design. This paper demonstrates a means to expand upon this concept using Java and NPI (Novas Programming Interface). It will give the reader pros and cons while giving pointers and tips on how to implement such a scheme.

2. Verdi Register Window and it's Limitations

Todays designs are large complex systems that require numerous designers and verification engineers. It is impractical for each engineer to have code-level knowledge of the whole system. When issues arise in logical blocks unfamiliar to an engineer, the engineer must spend time understanding the block to be able to debug it. They then typically open up a wave dump to examine behavior during simulation, see Figure 1.

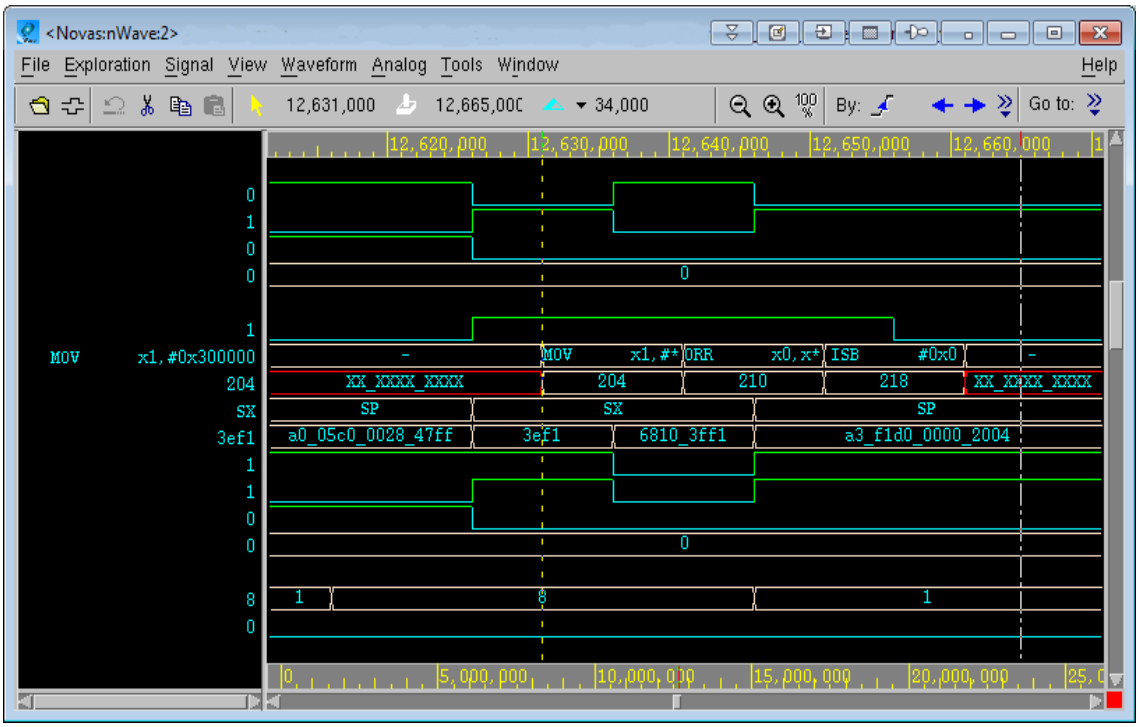

Figure 1. Wave view for debug

A common structure in microprocessors is an Issue Queue (IssueQ). These structures hold instructions and related data to be processed by execution units. The data is held until the unit is ready for the information. Register Windows can be used to simplify wave dump debugging by giving a unique graphical view of blocks in a more (micro)architectural manner. This allows the engineer to have a graphical view of data/control flowing through the system with limited knowledge of how the logic is actually implemented.

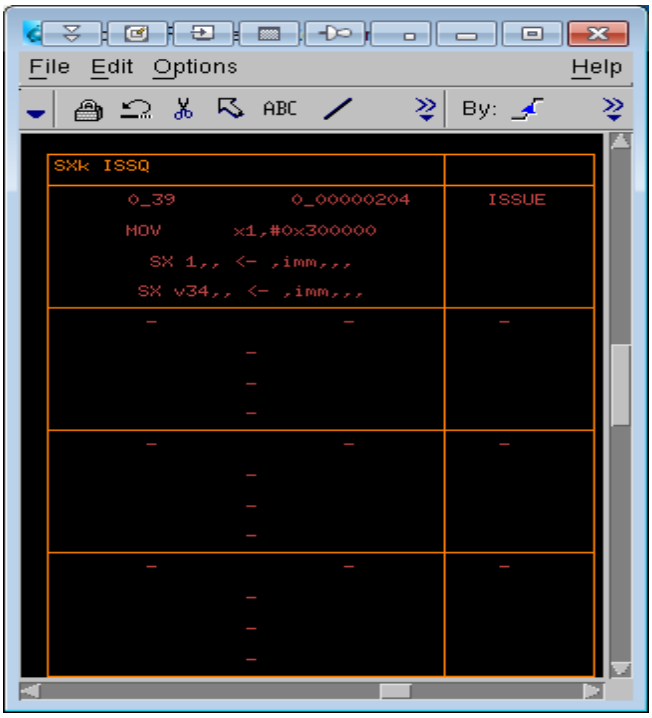

Figure 2. Register Window view of wave dump

Figure 2, shows the same information as Figure 1, which is the issue of an instruction to the SXk IssueQ. Repeated structures in RTL code can be represented using the same graphics, giving the engineer a common means of debugging. For the case in Figure 2, this would easily allow an engineer to see all instructions that are being issued to all execution blocks at once.

To create a Register Window for an IssueQ or any block, an engineer will typically open up the Verdi Register Window GUI. They will then draw the desired shapes by hand using simple widgets, shapes, text boxes etc. provided by the GUI. They will finally place each signal by hand via dragging and dropping, and save the drawing off as a .register file. The instantiation count of a simple RTL structure like an IssueQ can grow as a system gets larger, as microprocessors can have multiple execution units and therefore multiple IssueQ's. Currently the Register Window does not allow for a module approach and a path to be assigned to a variable when a Register Window is instantiated, meaning that if there are multiple instances of the same RTL module, individual drawings/files must be made for each instead of binding a drawing/file to an instance. For this reason, when a structure is repeated or gets complex, scripts tend to be used to create .register files as each instantiation must be unique. The larger the system, the more scripts/files have to be maintained, plus each engineer must know which files are applicable. Figure 3 shows the instantiation growth as a system grows.

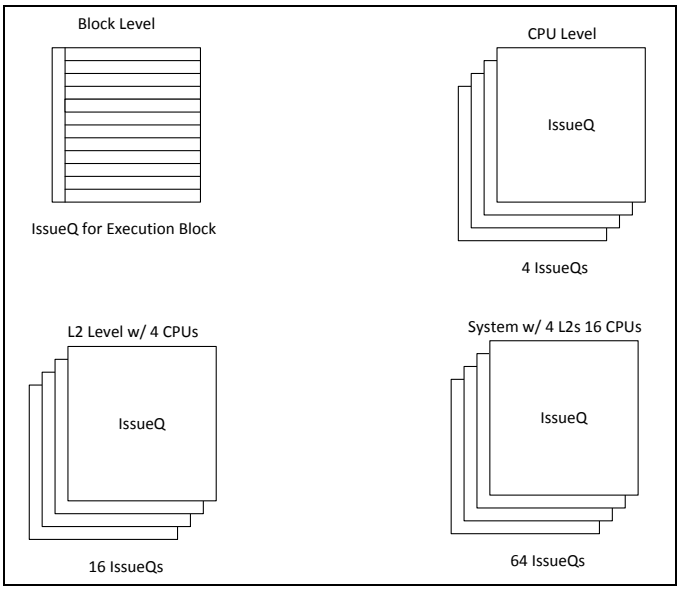

Figure 3. IssueQ instantiation count growth with system growth

For logical blocks that contain multiple sub-blocks, files can be combined. For the example given in Figure 3, each CPU (a microprocessor core shown here with four execution units) could have its own .register file, as could each L2 Cache System (in this case one with four CPUs connected to it), and each full "System" with four L2 Caches. This combining approach reduces the number of files that has to be managed, however it comes at a price. The example above is for only one small logical structure, and as the system gets larger more structures will need to be added. If the individual .register files are combined to make file management easier, the display size of these files can easily go beyond what can be displayed on a single monitor at a time, making debugging frustrating and defeating some of the original purpose.

3. Dynamic Register GUI

Fortunately Verdi and the Verdi Interoperability Apps (VIA) provide a means to replace the Register Window with whatever the user wants. The Novas Programming Interface (NPI) provides an API to manipulate an FSDB model via TCL commands and C APIs. Using this, most any type of GUI can be developed that an engineer wants. For the purpose of this paper Java is used, as it is cross platform capable and relatively easy to make a GUI with. The missing piece of using this GUI to debug, is to be able to update it and synchronize it based on the cursor placement in any Verdi Window (source, wave, schematic). To accomplish this, Verdi just needs to be started in serverPort mode. This allows Verdi to be accessed via the Verdi Interoperability Apps. A CmdClient, which is provided in the VIA Java library, is created and a callback registered. See Figure 4 for a system diagram. Note that the FSDB is actually opened twice and as such requires two licenses.

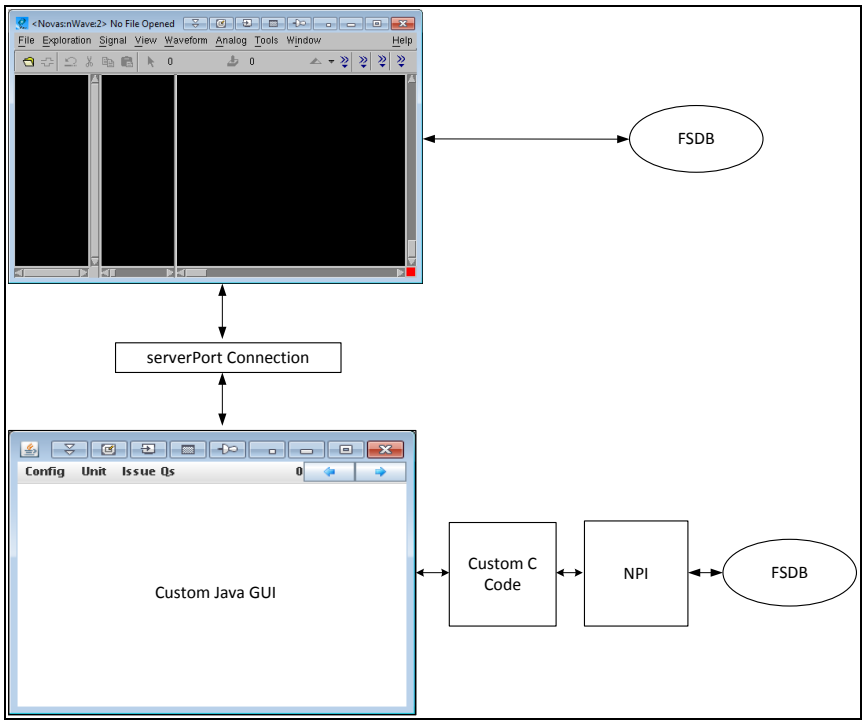

Figure 4. System Diagram

The basic program flow can be seen in Figure 5.

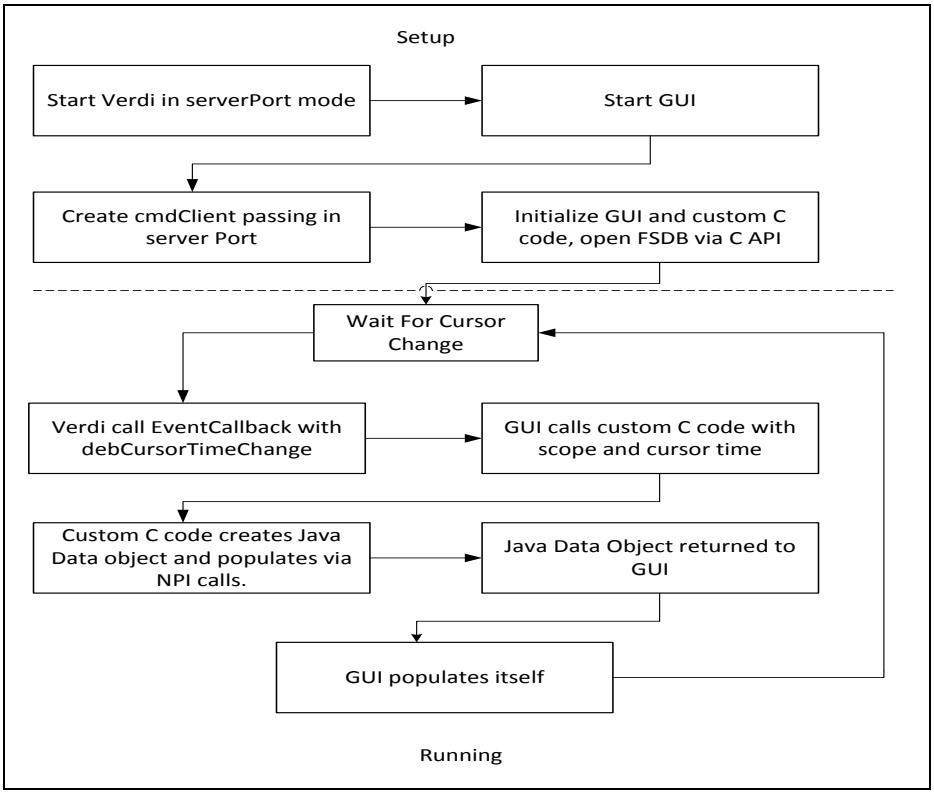

Figure 5. Execution Flow

Since the GUI and the custom C code are object oriented, reuse is vastly simplified over the original Register Window. In the example given in Figure 3 only one IssueQ structure needs to be created for the GUI and a matching population routine in the custom C code. This is then instantiated/executed multiple times, passing in multiple scopes based on the system configuration. Configuration can then be controlled via command line switches, configuration files or even in a self exploratory fashion through the NPI. This last method is possible using calls in the VIA API to obtain information such as the hierarchy, signal width, etc. from the design.

4. Details

While the details of building a GUI are beyond the scope of this paper, it is important to describe some of the details in order for one to determine if this method is worthwhile for their project. Below are code snippets to aid in this.

4.1 Java to Verdi serverPort Connection

To connect the GUI to Verdi, Verdi must be started up in serverPort mode and be running before the connection is made. While not necessary, it is helpful to run both the GUI and Verdi on the same machine, otherwise the machine must also be known, adding one more layer of complexity.

```
   import novas.sockcmd.*
   //Class instantiated by GUI
   public class CmdEventCallBack implements CmdEvent {
    public CmdEventCallBack(…) {
       RegWindowFsdb.initFsdb(FSDB, String.format("%s -
licdebug",VCFile));
   } //constructor

  public void EventCallback(String arg0, String arg1) {
    if(arg0.equals("debCursorTimeChange") {
      RegWindowData data =
RegWindowFsdb.getUpdate(Integer.parseInt(arg1),0,getScope())
;
      populateGUI(data); //Take the data object and populate
gui
      } //if
    } //EventCallback
  } //class
```

Above a CmdEvent, which is provided by the Novas package, is extended so that initFsdb, which is implemented in C, is called to initialize the FSDB. EventCallback is then modified so that on a "debCursorTimeChange" data is populated. This automates the updating of the GUI whenever the cursor in the Wave window changes.

4.2 Java to Custom C Code

A single Java class should be used to interact with the C code as it has to be run through the Java Native Interface (JNI).

```
public class RegWindowFsdb {
  static {
    try
      {
      System.loadLibrary("reg_window");
      } catch (java.lang.UnsatisfiedLinkError e)
      { System.out.println(e);
  } //static

  public static native RegWindowData getFsdbUpdate(int
cursor, int mask,String scope);
  public static native void initFsdb(String fsdb_path, String
arguments);
} //class
```

To produce the C header that enables Java to call C, execute:

```
javah  -jni RegWindowFsdb
```

4.3 C Code to Initialize FSDB

```
#include "npi.h" // header for NPI Models: npi_init,
npi_load_design,npi_end
#include "npi_L1.h" // header for NPI Libraries
#include "npi_fsdb.h"
#include <jni.h> //Java Native Interface header
#include "RegWindowFsdb.h" //Created via javah -jni
JNIEXPORT void JNICALL Java_RegWindowFsdb_initFsdb
(JNIEnv *env, jclass, jstring _fsdbPath, jstring _arguments) {
  //Convert _arguments to const char* and get argument count
then
  output = npi_init(count,pargv);
  //Convert _fsdbPath to const char*  then

  fp = npi_fsdb_open(fsdbPath);
  if ( !fp ) {
    printf("Failed to open FSDB file.\n");
  }
  }
}//Java_RegWindowFsdb_initFsdb
```

To initialize the FSDB in C, multiple headers need to be included. The initFsdb function that was referenced in Java is then implemented calling the correct API.

4.4 Initialize Java Data Object and Update in C

To reduce calls between Java and C and to simplify the interface, a Java data object is created and populated in the C code. This adds complexity at first, but once the basic flow is understood and helper functions are written, the time required to add new items to the GUI is reduced.

```
JNIEXPORT jobject JNICALL Java_RegWindowFsdb_getFsdbUpdate
(JNIEnv *env, jclass, jint cursor, jint mask, jstring _scope) {
  const char *scope = env->GetStringUTFChars(_scope,0);
  //Get a pointer to the Java Data Class
  jclass RegWindowDataClass = (env)->FindClass("RegWindowData");
  //Get a pointer to it's constructor
  jmethodID constructor = (env)->GetMethodID(RegWindowDataClass,
"<init>","()V");
  //Get an instance of the data object
  jobject RegWindowDataInstance = (env)-
>NewObject(RegWindowDataClass,constructor);

  //Update Object
update(env,cursor,RegWindowDataClass,RegWindowDataInstance,scope)
;
 return RegWindowDataInstance;
} //Java_RegWindowFsdb_getFsdbUpdate
```

The GUI then takes this data object and populates all of its menus, text boxes, buttons, etc. Since a single data object is used instead of single calls to C, other Java programs can easily populate the same code, which means that the GUI is no longer limited to only using an FSDB as a data source. Though not implemented for this paper, a monitor could be written for simulation or emulation that produces the needed information to populate the GUI, aiding in debug of other environments.

4.5 C Code to Initialize a Java Variable

The code to update a single variable is complex, using not only the NPI API but JNI as well. Since ideally multiple engineers will be creating Register Windows it is helpful to functionalize this code. Below is an example of how to populate a Java String inside of C, with data coming from the NPI.

```
void update_string(JNIEnv *env, jclass complexClass, jobject
RegWindowData, const int width,const char *signal_name,int
cursor,const char* var_name) {
  char char_sig[256] = " ";
  jfieldID fieldID;
  npiFsdbValue val;
  val.format = npiFsdbHexStrVal;
  npiFsdbSigHandle sigHandle =
npi_fsdb_sig_by_name(fp,signal_name,NULL);
  npiFsdbVctHandle vctHandle = npi_fsdb_create_vct(sigHandle);
  npi_fsdb_goto_time(vctHandle,cursor);
  npi_fsdb_vct_value(vctHandle,&val);
  .
  .
  fieldID = env-
>GetFieldID(complexClass,var_name,"Ljava/lang/String;");
  if(fieldID == NULL)
    printf("update_string fieldID null for
variable:%s\n",var_name);
  if(width == -1)
    env->SetObjectField(RegWindowData,fieldID,(env)-
>NewStringUTF(val.value.str));
  else
    env->SetObjectField(RegWindowData,fieldID,(env)-
>NewStringUTF(char_sig));
}
```

The first part of the code is setting up and getting the correct value from NPI. The second is using JNI to set the value inside the Java object. Similar code can be written for arrays or custom objects. The deeper the object is though, the more steps are required to access it from the C code, as shown below. All subclasses, from the main data object should be initialized via the main data object (RegisterWindowData) on the Java side, otherwise they would have to be done in C which is more complex.

```
//Get the jfieldID of an array of Java IssueQData Objects
jfieldID arrayID = env-
>GetFieldID(complexClass,issueq_name,"[LIssueQData;");
jobject jarray = env->GetObjectField(RegWindowData,arrayID);
jobject object = env->GetObjectArrayElement(
(jobjectArray)jarray,array_element); //where array_element
is an int
.
.
jclass IssueQDataClass = (env)->FindClass("IssueQData");
fieldID = env-
>GetFieldID(IssueQDataClass,var_name,"Ljava/lang/String;");
   env->SetObjectField(object,fieldID,(env)-
>NewStringUTF(char_sig));
```

5. Conclusions

This paper has shown a flexible replacement for the Verdi Register Window that is only limited by the imagination of the programmer and their GUI capabilities. Below are the pros and cons of each system that need to be considered when deciding which to use.

Verdi Register Window

Pros

- Simple to use
- Single system for debug with a wave dump

Cons

- Poor reuse, as each RTL structure must have its own .register file, and multiple instantiation are not supported
- Limited graphical options compared to a full programming environment
- File management issues for large projects

Dynamic Register Window

Pros

- Reuse of logical levels

- GUI population reuse from multiple sources
- Dynamically configurable
- Limited only by engineer's imagination/time

Cons

- Complex
- Most RTL designers/validation engineers are inexperienced at making GUI's
- Need to know multiple programming languages
- Multiple places need to be edited even for simple changes

For a simple one-off design, the overhead of a dynamic system is extensive and most likely prohibitive. However if a configurable system is going to be created, such as an L2 Cache that has a variable number of heterogeneous CPUs or a System with a variable number of these L2 Caches, a dynamically configured Java based solution using the Verdi VIA framework handles the case much more efficiently than a Verdi Register Window.

6. References

[1] Verdi Interoperability Apps (VIA) Novas Programming Interface (NPI) version 2013.04

[2] Java Native Interface Specification
http://docs.oracle.com/javase/7/docs/technotes/guides/jni/index.html.

[3] Java Programming Tutorial Java Native Interface (JNI)
http://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html