
Verdi - Embedded Software Debug

SNUG Boston 2014

Dan Grabowski (dan.grabowski@nvidia.com),

nVidia

www.nvidia.com

Alex Wakefield (AlexW@synopsys.com),

Synopsys Inc

www.synopsys.com

ABSTRACT

This paper provides an overview of how to install, configure, and use the Verdi HW/SW Debug solution with a multi-core based design. Many modern SoC designs have an embedded core and some amount of C or assembly code is executed during the validation of the RTL. The Verdi HW/SW debug tool provides an efficient mechanism to debug the embedded software standalone or in conjunction with the RTL design.

1. Introduction	3
2. MinSoC Design	3
3. Embedded Software Compile.....	4
3.1 C-Compiler Switches	4
3.2 Boot Loader	5
4. Verdi HW/SW Recorders.....	8
4.1 Verdi Release	8
4.2 Verdi HW/SW Debug Recorders	8
4.3 Instantiating Recorders.....	9
4.4 Memory Configuration	10
4.5 Compile Switches	13
4.6 Runtime Switches	13
5. Software Debug.....	14
5.1 Verdi then Eclipse Invocation	14
5.2 Eclipse then Verdi Invocation	15
5.3 Time Synchronization	16
6. Eclipse Software Debug Features.....	17
7. Sample Software Debug	20
7.1 Interactive Post-Process Debug	20
7.2 UART Transmit	21
7.3 Waveform View.....	22
7.4 Debug of characters	23
8. Conclusion	25

1. Introduction

The amount of embedded software present in designs is increasing and validation of this software along with hardware is a difficult and time consuming problem. Further complicating this is the impact of a software/hardware bug late in the design cycle or after first silicon is available. These problems cause additional silicon re-spins and delay time to volume silicon and profits.

The Verdi HW/SW debug tool allows verification or software engineers to efficiently debug software and hardware issues during RTL simulation. This paper will describe the features provided by the tool and the necessary steps to install and configure the tool on a single-core design.

2. MinSoC Design

To show the flow of the design, we will use an open-source design containing an OpenRisc1200 core along with UART and Ethernet peripherals. While this is not the design used at nVidia, the design is representative of a single-core design and we can easily share the demonstration design. The demonstration tar-file is available from Synopsys; please contact vcs_support@synopsys.com to obtain a copy of the design used in this paper.

The design consists of a single OpenRisc1200 core, memory and Uart + Ethernet peripherals as shown in Figure 1.

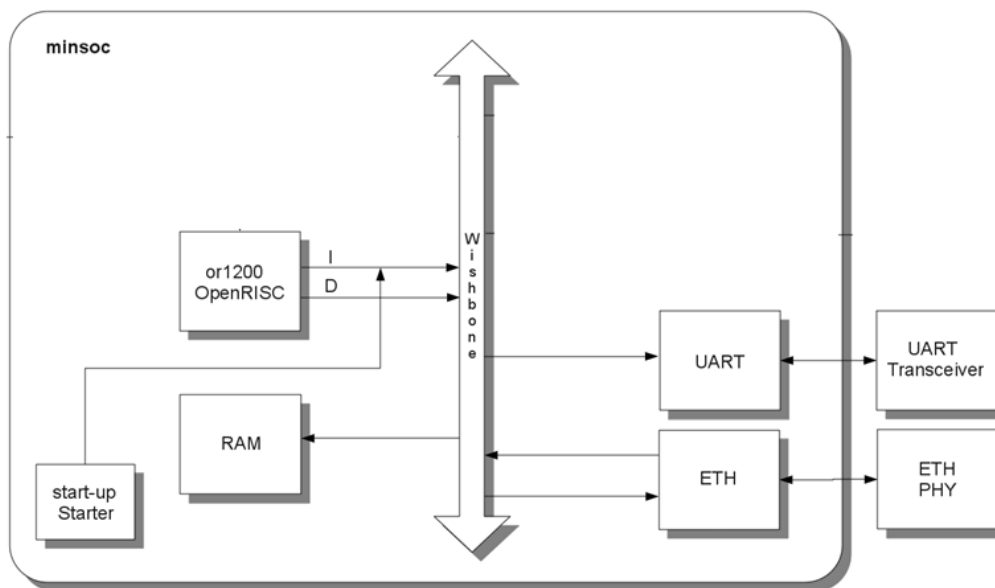


Figure 1 MinSoC Block Diagram

3. Embedded Software Compile

The software running on the embedded core is compiled using several steps, and produces several different format files that are needed for debug. Verdi HW/SW-Debug utilizes the GDB debugger to perform the debugging of the software and any compiler that is compatible with GDB can be used. ARM or GNU C compilers (armcc, gcc) are often used although other C, C++ or an assembler can also be used.

When the compiler reads the code, it generates an ELF or AXF file containing the opcodes that the processor will execute, along with initial memory values, and if requested debug information. This ELF or AXF file is needed by GDB to correlate the program counter back to the source file/line-number and decode the variable values.

The ELF or AXF file is then converted into a memory image (typically HEX file) for loading into the memory of the simulation, or into the memory of the hardware when it is available. The HEX file contains only the binary opcodes for the CPU to execute and the initial memory values – it does not contain the debug information.

Verdi HW/SW (and also the debug tools used when debugging silicon) require the ELF/AXF file to decode the executable opcodes and the debug information to link to the source, variables, and stack.

3.1 C-Compiler Switches

The C-compiler can generate code with varying levels of optimization for speed or memory size. When debugging the hardware you will likely want to reduce the optimization so the C-source code matches the disassembly view more closely. This will result in a slower-running executable, as the compiler will not optimize out some sections of code, however it will result in a more efficient debug of the software/hardware interaction.

When debugging production embedded code, you will need to use the same optimization flags used by the software team, likely with full software optimization enabled. Verdi HW/SW debug supports all levels of optimization; however the disassembly may not correlate to the C-source code as easily. Some examples are unused variables will be optimized out, small loops may be unrolled and the code simply duplicated, common statements inside if/else branches will be moved outside the if/else, and statements may be re-ordered.

Separate from the C-code optimization is the generation of the debug information including the filename/line-number that each program counter address refers to, the address of each function, the address of each variable and information to decode the C call-stack. This debug

information can be generated for any level of C-code optimization and is enabled by different flags.

GCC uses the following flags, check your compiler documentation for similar flags:

-g : Include debug information in the executable

-O0 : Minimal optimization (not always available for all cores)

-O1 : Low optimization, use for debugging with software-driven testing of RTL

-O2 : Default optimization, perform most optimizations

-O3 : Highest optimization, used for production code

In the MinSoC design, the Makefile contains the following steps to compile the design.

```
gcc -g -c except.S -o except.o
gcc -g -c reset.S -o reset.o
gcc -g -c eth.c -o eth.o
gcc -g reset.o except.o eth.o -o eth.elf

or32-elf-objcopy -O binary eth.elf eth.bin
bin2hex eth.bin 1 -size_word > eth.hex
```

3.2 Boot Loader

When C-code is compiled for Linux, the compiler generates a boot-loader to initialize the CPU registers, setup the stack, and initialize memory regions. This code is often customized by the software team for the specific hardware onto which the embedded software is loaded.

Verdi HW/SW Debug is particularly useful for debugging this boot-loader code, as the code interacts heavily with the hardware, and errors in this code often cause the RTL to hang and become unresponsive.

In the demonstration design, the boot-loader code initializes all registers, performs some setup, and then calls the C main routine. Note, the reset section and the registers are all getting initialized with 0x0. Most boot-loaders will follow a similar, but often more complex set of steps. You can view this in the following file:

```
Csh% more support/reset.S

#include "or1200.h"
#include <board.h>
```

```

_reset_vector:
    l.nop
    l.nop
    l.addi r2,r0,0x0 // Reset R2
    l.addi r3,r0,0x0 // Reset R3
    l.addi r4,r0,0x0 // Reset R4
    l.addi r5,r0,0x0 // Reset R5
    l.addi r6,r0,0x0 // Reset R6
    l.addi r7,r0,0x0 // Reset R7

----8<----8<----8<----

    l.movhi r3,hi(_start)
    l.ori   r3,r3,lo(_start)
    l.jr    r3
    l.nop

.section .text
_start:
    l.addi r10,r0,0 // Flush IC and/or DC
    l.addi r11,r0,0

```

a) ELF Sections and Debug Data

The ELF file has several sections for storage of different data items and these sections can be listed using the *objdump* utility. The command to view the ELF sections, along with a sample of the output is shown below. This utility is useful to see where the memory sections of an ELF file are located.

```

Csh% objdump -h eth/eth.elf

Sections:

```

Idx	Name	Size	VMA	LMA	File off	Align
0	.reset	00000190	00000000	00000000	00002000	2**0
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE					
1	.vectors	00000e1c	00000200	00000200	00002200	2**0
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE					
2	.text	00004a74	00001200	00001200	00003200	2**2
	CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE					
3	.rodata	00000042	00005c74	00005c74	00007c74	2**0
	CONTENTS, ALLOC, LOAD, READONLY, DATA					

```

4 .bss          00000e84 00005cc0 00005cc0 00007cb6 2**4
                ALLOC
5 .stack        00001000 00006b44 00006b44 00007cb6 2**0
                CONTENTS, READONLY
6 .stab         00003090 00000000 00000000 00008cb8 2**2
                CONTENTS, RELOC, READONLY, DEBUGGING
7 .stabstr      000032d9 00000000 00000000 0000bd48 2**0
                CONTENTS, READONLY, DEBUGGING
8 .comment      000000a2 00000000 00000000 0000f021 2**0
                CONTENTS, READONLY

```

Debug information must also be generated and saved in the ELF file so GDB can display and correlate the C-source, C-stack, and C-variables. If the debug sections are missing, (as shown by the command below) you will only see the disassembly view and will need to re-compile the C-code with the “-g” (or similar flag) to enable the other views.

```

Csh% objdump -g eth/eth.elf

eth/eth.elf:      file format elf32-big
objdump: eth/eth.elf: no recognized debugging information

```

b) HEX File

This file is just hexadecimal data. It is the raw memory image that will be loaded into the RAM model, and the RTL will fetch and execute these binary opcodes. It does not contain any debug or symbol information, so this can't be used by GDB. You can view the hex file using the following command.

```

Csh% more eth/eth.hex

00
00
5c
b8
00

```

4. Verdi HW/SW Recorders

Verdi HW/SW Debug requires the CPU instruction trace to be saved into the FSDB waveform file when the simulation runs. This will allow the tools to replay the opcodes that the RTL executed. It will allow you to debug the software that ran on the embedded CPU along with all of the regular RTL debug features provided by Verdi. This section details how to record this information into the FSDB file using a synthesizable or log-file recorder.

4.1 Verdi Release

HW/SW debugger is available in the Verdi³ 2013.09 and later releases; however we recommend you use the latest Verdi³ release available. A 64-bit installation of Java and Eclipse are provided with the Verdi installation which simplifies the install process as these tools do not need to be downloaded and configured separately.

Download the latest release of Verdi from the Synopsys Product FTP site, including “HwSwDebug”, “NPI”, and “demo” packages.

4.2 Verdi HW/SW Debug Recorders

The Verdi HW/SW Debug tool needs the following information saved into the FSDB file for post-process or interactive debug:

- Time in nsec or clock cycle counts
- Program Counter Values
- Register Changes (writes to the register file)
- CPU → L1 Memory read/write address/data/size

ARM® and several other CPU vendors create an instruction trace-log as the simulation executes. It contains the information above in text format. Recorder modules that read this text file and save the information into the FSDB file in the correct location are inside the Verdi release.

An alternative approach is to create a recorder module that extracts the above information directly from the RTL design and saves this data directly into the FSDB file in the correct location. This approach is used for the OpenRisc 1200 core and several other smaller simpler cores.

Both approaches result in the same format FSDB file with the same set of information recorded. The only difference is how the information is obtained. The log-file approach is often simpler and isolated from small RTL changes, as the instruction-trace file typically does not change from one release of the Core RTL to another.

As the simulation executes, examine the instruction trace-log from your design, and if needed consult the CPU vendor documentation for steps on how to include this feature into the testbench and enable it for each core in the design.

4.3 Instantiating Recorders

You must instantiate one recorder module per physical CPU present in your design. It is possible to set parameters on these instances to specify the log filename and if needed the CPU clock pin instance path. If you have multiple cores of different types, then you will need to instantiate one recorder instance per core of the corresponding type.

The clock pin is needed if the instruction traces have cycle-counts or clock-counts. It is not required if the instruction trace has nanosecond time stamps and in this case the port can be connected to zero (1'b0) to prevent compiler unconnected pin warnings.

A sample wrapper file containing all recorder names is provided in the Verdi³ installation. `$VERDI_HOME/share/hwsw_debug/recorders/customHwswDebugTop.sv`. This file can simply be copied into the design area and edited to leave only the instances for your SoC design. A sample of this file for a quad CortexM3 design is shown in Figure 2.

This top-level wrapper file can be an extra top-level instance or instantiated into the design hierarchy. The Verdi HW/SW-Debug tools look for a top-level instance called `verdiHwswDebugTop` by default; however, this can be overridden in the invocation dialog.

```
// (C) SYNOPSYS 2013
`include "verdiRecorderM3.svp"
module verdiHswDebugTop;

    // M3 Cores
    verdiRecorderM3 #(.cpuId(0), .tarmacFileName("cpu0.tarmac")) cpu0();
    verdiRecorderM3 #(.cpuId(1), .tarmacFileName("cpu1.tarmac")) cpu1();
    verdiRecorderM3 #(.cpuId(2), .tarmacFileName("cpu2.tarmac")) cpu2();
    verdiRecorderM3 #(.cpuId(3), .tarmacFileName("cpu3.tarmac")) cpu3();

    // FSDB dumper
    initial #0 $fsdbDumpvars(0, "verdiHswDebugTop", "+all", "+parameter");

    // Verdi HW/SW Initialization
    `VERDI_HWSW_INIT_TOP
endmodule
```

Figure 2 Single Cluster HWSWDebugTop.sv file

4.4 Memory Configuration

Verdi HW/SW-Debug can invoke and connect to one or more GDB debuggers for debug of a multi-CPU system and each GDB can execute multiple software threads inside. This combination of multiple-GDB and multiple-threads allows us to model complex multi-CPU systems efficiently.

Each GDB contains a single memory model, representing a single memory space or memory map. If several physical CPU's in your system share memory, then each of these CPU's should be modeled using a separate thread within a single GDB.

If you have several memory spaces where each space is accessed by a different core, then these spaces should be modeled by separate GDBs.

GDB threads are used to model a physical CPU, and not used to model software threads that are provided by an operating system running on the CPU. Any software thread activity will be displayed within a single GDB thread as the CPU core saves software-thread data and jumps from one thread to another inside the user-code.

The two common configurations are:

a) Multiple CPUs, all share a single memory space

Here all CPUs inside the same cluster share the same memory space. We want to see several CPUs all executing code in parallel, but all CPUs share the same single memory space.

We model this using multiple threads inside one GDB server that has a single memory model.

b) Multiple CPUs, all have separate memory space

Here all CPUs have their own memory space, and the memory is not shared. We want to see several CPUs all executing code in parallel, and each CPU has its own memory space.

We model this using one GDB server per CPU, and each GDB server has a single thread and a single memory model.

c) Two clusters, each share memory within the cluster but not between

This is modeled using two GDB servers, each with multiple threads representing the cores inside each cluster. Each GDB server is configured using (a) above, but we just have two of these GDB servers running in parallel.

d) Several CPUs share memory, Several CPUs have separate memory

This is a combination of (a) and (b). The cores that share memory will be modeled by a single GDB server with one thread per core, and all will share the same GDB memory model. The cores that don't share memory will run in separate GDB servers, each having one thread.

Specifying the configuration

To specify the configuration of GDB servers and threads inside, we use the hierarchy inside the `verdiHwswDebugTop()` module. If no hierarchy is present, then all cores will be executed as a thread within a single GDB server - configuration (a) above. This is shown earlier in Figure 2.

To define a cluster or GDB server, simply add a level of hierarchy into the `verdiHwswDebugModule` and set the corresponding `cpuld` and `clusterId` parameters. A sample a 4xM3 clusters sharing memory and a 2xA15 cluster sharing memory is shown below in Figure 3.

Check the latest Verdi HW/SW Debug user-guide for additional details on how to specify memory configurations, as additional features for sharing memory regions and specifying the memory configuration are planned in 2014.12 and later releases.

```
`include "verdiRecorderM3.svp"
`include "verdiRecorderA15.svp"

module verdiHswDebugTop;

    // Instantiate Clusters

    m3 m3();

    a15 a15();

    // FSDB dumper

    initial #0 $fsdbDumpvars(0, "verdiHswDebugTop", "+all", "+parameter");

    // Verdi HW/SW Initialization

    `VERDI_HWSW_INIT_TOP
endmodule

// M3 Cores
module m3();

    verdiRecorderM3 #(.clusterId(0),.cpuId(0), .tarmacFileName("cpu0.tarmac")) cpu0();
    verdiRecorderM3 #(.clusterId(0),.cpuId(1), .tarmacFileName("cpu1.tarmac")) cpu1();
    verdiRecorderM3 #(.clusterId(0),.cpuId(2), .tarmacFileName("cpu2.tarmac")) cpu2();
    verdiRecorderM3 #(.clusterId(0),.cpuId(3), .tarmacFileName("cpu3.tarmac")) cpu3();

endmodule

// A15 Cores
module a15();

    verdiRecorderA15 #(.clusterId(1),.cpuId(0), .tarmacFileName("cl1_cpu0.tarmac")) cpu0();
    verdiRecorderA15 #(.clusterId(1),.cpuId(1), .tarmacFileName("cl1_cpu1.tarmac")) cpu1();

endmodule
```

Figure 3 Multiple Cluster HWSWDebugTop.sv file

4.5 Compile Switches

Several switches must be added to the simulator compile command to enable FSDB dumping and to specify the location of the recorder include files, and the top-level recorder wrapper file detailed in the previous step. The additional switches needed for VCS are shown below in Figure 4.

```
csh% vcs -sverilog -debug -lca HswDebugTop.sv \
      +incdir+${VERDI_HOME}/share/hsw_debug/recorders \
      -P ${VERDI_HOME}/share/PLI/VCS/LINUX/pli.tab
      ${VERDI_HOME}/share/PLI/VCS/LINUX/pli.a
```

Figure 4 Compile Change File

To compile and run the MinSoC demo design, use the makefile commands below. Modify these Makefile commands as needed if your environment has wrapper scripts around the VCS or Verdi tools.

```
Csh% gmake compile run
```

4.6 Runtime Switches

At runtime the recorders save the name of the ELF file into the FSDB waveform file along with all of the instruction trace information. Each core can have a boot-ELF and an application-ELF specified, and GDB will load both of these when debug starts. The ELF pathnames are specified using a runtime plus-arg switch.

Depending on the number of cores or clusters in the design you can specify a single ELF for all cores, a different ELF per cluster or a different ELF per core. The documentation has a full list of switches available for ELF loading.

```
Csh% ./simv +verdi_hsw_exe=test.elf
Csh% ./simv +verdi_hsw_boot_exe=boot.elf +verdi_hsw_exe=test.elf
```

The recorders will save the instruction trace during simulation. One useful check, if you can perform it, is to verify that data is in fact saved into the FSDB file. Within nWave's > Get Signal browser, simply locate and view some signals in the VerdiHswDebugTop.cpu0.* scope to ensure that data is present. This data is shifted in time, so you should not refer to these signals directly.

If no data is in the verdiHwswDebugTop hierarchy then this indicates the recorder was unable to read the instruction-trace information written by the RTL or DSM CPU model. Check that the instruction trace-log file exists, and there are no errors in the simulation log.

5. Software Debug

The Eclipse window can be launched from Verdi, which is the logical manner for a verification engineer. Also, the eclipse window can be launched first, and then if needed, Verdi can be invoked, which is logical for software engineers. This section will detail the two approaches for invoking the tools.

5.1 Verdi then Eclipse Invocation

The Eclipse window can be invoked from the Verdi tool by choosing the “Tools > Invoke HW SW Debug...” menu item as shown in Figure 5.

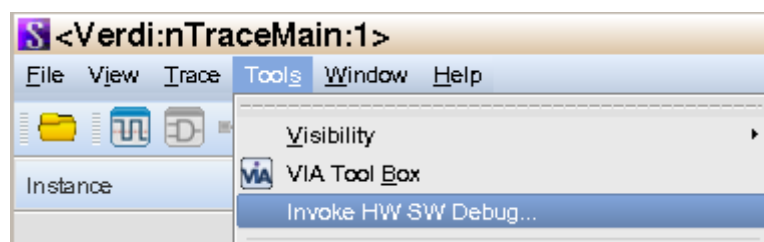


Figure 5 Invoke HW/SW Feature in Verdi

The invoke dialog shown in Figure 6 allows you to select a set of cores to debug, and this will cause the corresponding number of GDB and threads to be invoked. It is possible (and recommended) that you record instruction streams for all cores during simulation, then during debug, select only the active cores you want to debug. This will minimize the number of simulations needed and retain the flexibility and performance by selecting the cores at debug time.

The location of the simulation working directory and the starting time can also be specified. Some additional options are available on the advanced tab, however these are rarely needed.

In the MinSoc design, one core is present, so OK this dialog in Verdi, and then OK the following two dialogs that appear once eclipse invokes for workspace location and eclipse perspective (window layout).

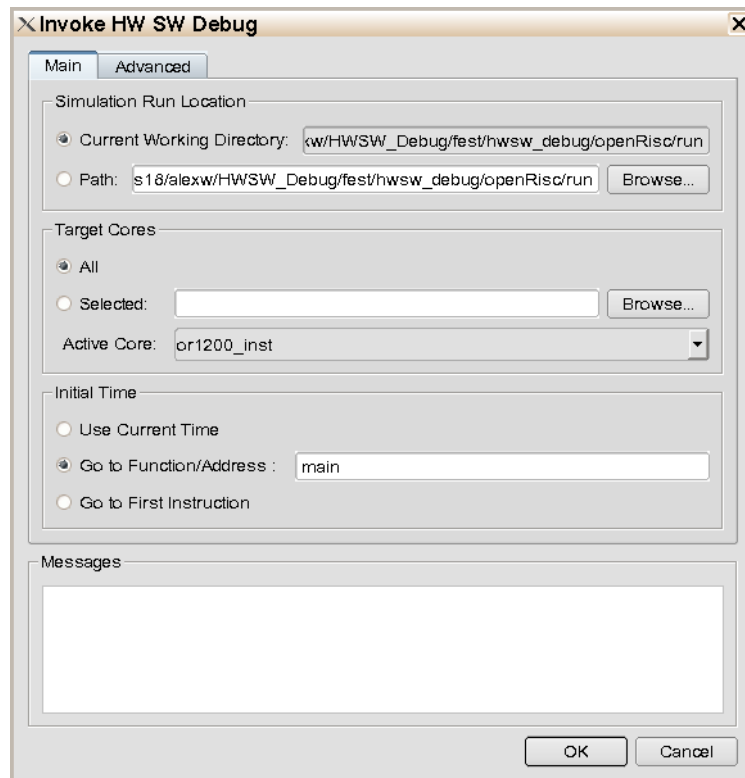


Figure 6 Invoke Dialog for HW/SW Debug

5.2 Eclipse then Verdi Invocation

The eclipse tool can be invoked first using the following command and then specify the location of the FSDB file as shown below. Many other options are available, use the `-help` switch to view these options or check the Verdi HW/SW-Debug users guide for a full description.

```
Csh% hwsd_debug -ssf filename.fsdb
```

During software debug, if the Verdi tool is needed to view waveforms or perform other RTL debug, then the debug, then the Verdi tool can be launched using the “Verdi” button in Eclipse as shown in

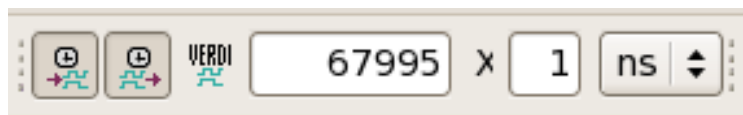


Figure 7.

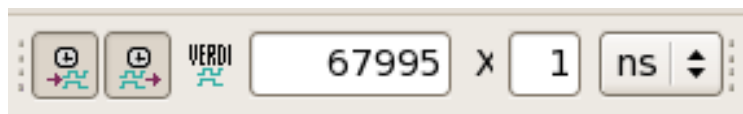


Figure 7 Invoke Verdi Button in Eclipse

5.3 Time Synchronization

The Verdi and Eclipse tools are synchronized in time. Any change in either window will cause the time to be matched in the other view. This can also be controlled using the two HW → SW and SW → HW buttons in the Eclipse toolbar. Activating both of these buttons is the default and causes time changes to be passed in both directions. If only the SW → HW button is active, then any changes in Eclipse will cause the Verdi time to move, but not the other way around. Similarly if the HW → SW button is active, then any time changes in Verdi will cause Eclipse to be updated. Finally, if both buttons are de-activated, the Eclipse and Verdi times will be independent and not affect each other.

Also note that the current simulation time is displayed in the Eclipse toolbar, and this field can be edited. So, typing a time in this field will cause the time viewed in Eclipse to be updated. If the SW → HW button is active, it will also cause Verdi to advance forward or backward to this time.

With the 2014.03 release of the Verdi tools, Eclipse cannot advance time during an interactive simulation. All time advances of the simulator must be performed using the Verdi interactive-features in the Verdi GUI. Check the latest documentation and user-guide in future releases, as additional interactive features are planned.

6. Eclipse Software Debug Features

An Eclipse perspective is included that provides a convenient set of initial Eclipse windows for embedded software debug. You can then modify the layout of windows if needed and save this layout in the Eclipse workspace directory. The initial window layout is shown below in Figure 8.

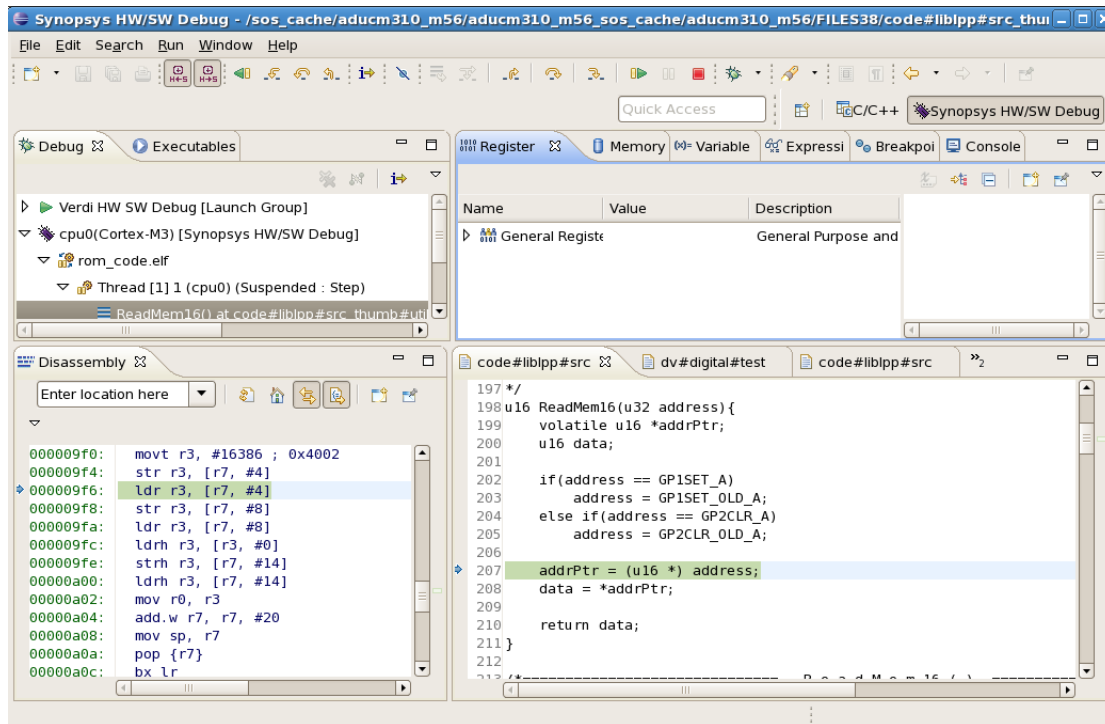


Figure 8 Main Eclipse/Software Debug Window

A complete software debug environment is provided by the Verdi HW/SW Debug tool including the following views and features.

a) C source code view

The C source code is displayed with coloring of keywords. The current line is highlighted and step forward and backward features are available even in post-process mode. Breakpoints can be added into the code by double clicking on the line number, and then, you can run forwards or backwards to any breakpoint. The problem of stepping one instruction too far and then having to re-run the simulation is avoided as you can simply step or run backwards in this environment.

Running and stepping forward and backward feels like interactive debug, except that it is very fast. In this mode, you cannot change values as the simulation has already completed.

b) Disassembly View

The disassembly of the C-source is provided and cross highlights to the C-source view. You can step forward or backward one C line at a time or one opcode at a time. If opcode stepping is used then the C line will advance when required to stay synchronized with the disassembly view.

c) Stack View

The C call stack is shown which allows you to easily identify which line of code called the current routine. This information is particularly difficult to obtain using traditional debug methods as you would have to decode the C call stack in memory manually.

d) Register View

Each of the register values is displayed, including registers that change due to user or supervisor modes. The registers are easy to find in simple RTL designs; however with the more complex cores these registers may be virtual and not correspond directly to a signal in the RTL design. The correct value is displayed for all registers by the Verdi³ tool.

e) Variables

Local variables are displayed in this window, easily showing the value even if the variable is global memory, a register or on the C stack. Manually finding these values is possible, but again is time consuming as you would need to examine and correlate the disassembly back to the C-code to identify where the variables are stored and then find the values.

f) Expressions

Any variable or other expression can be added to this window and acts as a scratchpad of values that you are interested in. This view (along with the variable view) supports complex data types such as structures, unions, and pointers. A convenient tree format is shown here allowing you to expand the fields as needed.

g) Memory

The memory view provides a programmers view of the memory space. This is calculated by taking the initial ELF/AXF image, and then applying all read/write transactions up to the current time. This method is efficient for disk space as only the read/write stream is saved, and then when needed, the full memory contents are calculated for any simulation time. It is a programmer's view and shows only values

read or written by the processor. If you want to see memory-mapped peripheral registers you will need to use the Verdi³ views.

A simple example is a UART Tx/Rx register pair. When the CPU writes to the Tx register, you will see the value in the memory view. Shortly after, the UART will send the bits out over the serial interface. When a byte is received, the UART will typically generate an interrupt. When the interrupt routine is executed, the Rx register is read. Only at this time will the memory location show the received value.

If you require the values to be updated immediately, you can add the UART Tx/Rx registers into the register window by specifying the register name and path in the FSDB file. The register window will then update the values immediately.

The memory view is intended to be the programmer's view of memory/peripherals, while the Verdi³ views are the hardware view of memory/peripherals. If errors occur, these two views provide the software and hardware perspectives allowing you to debug the mismatch.

h) GDB Console

The Verdi³ software debug solution uses GDB-debugger as part of the debug engine. So GDB is actually running and thinks that it is connected to a live CPU during the debug process. This means that GDB is providing all of the data in the views. Any feature present in GDB can be performed with the Verdi³ solution. The console window is the GDB command line and output, and can be used by software engineers familiar with the GDB tool.

7. Sample Software Debug

The software debugger can be used to debug the embedded code, while Verdi can be used to debug the hardware. A combination of these two tools is used in the following sections to show how to debug an issue with the SoC Design.

Verdi HW/SW Debug can execute in post-process mode, where the simulation has already completed. However, the features inside Eclipse behave like an interactive debugger, as we can set breakpoints, single-step through the code, view variables, examine the stack, and explore memory. You can also step or run backwards to any previous time which is very powerful. Of course, time is synchronized with the Verdi hardware views.

7.1 Interactive Post-Process Debug

We can easily add a breakpoint on the “uart_init” line of the eth.c file by double clicking on the line number, so we can go back to this location.



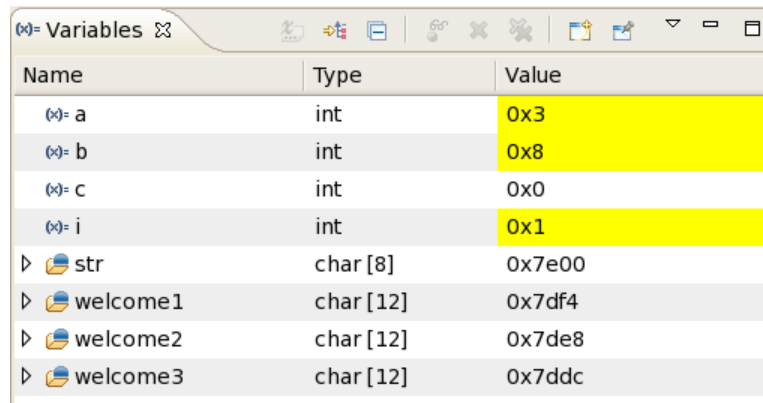
Add breakpoints on some other lines, run forward and back to these to become familiar with the interactive post-processing debug idea. Try running to the start of the boot code, and to the end of the simulation time using the green left and right arrows.



Use the shortcut, “I”, to turn on instruction stepping mode. Note that you can step forward/backward one opcode at a time, and also, add breakpoints in the disassembly view. Note that several disassembled instructions make up a single C-statement when many steps forward in the disassembly view are needed to advance the C-source view one line.

View the local variables for “a”, “b” and “c” by selecting the Variables tab.

Run forward through the loop where the a, b, c values are calculated.



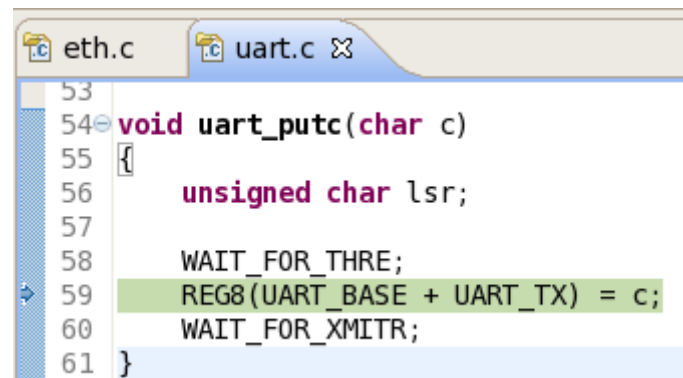
Name	Type	Value
(x) a	int	0x3
(x) b	int	0x8
(x) c	int	0x0
(x) i	int	0x1
str	char [8]	0x7e00
welcome1	char [12]	0x7df4
welcome2	char [12]	0x7de8
welcome3	char [12]	0x7ddc

Run backward or forward to return to line 43.

7.2 UART Transmit

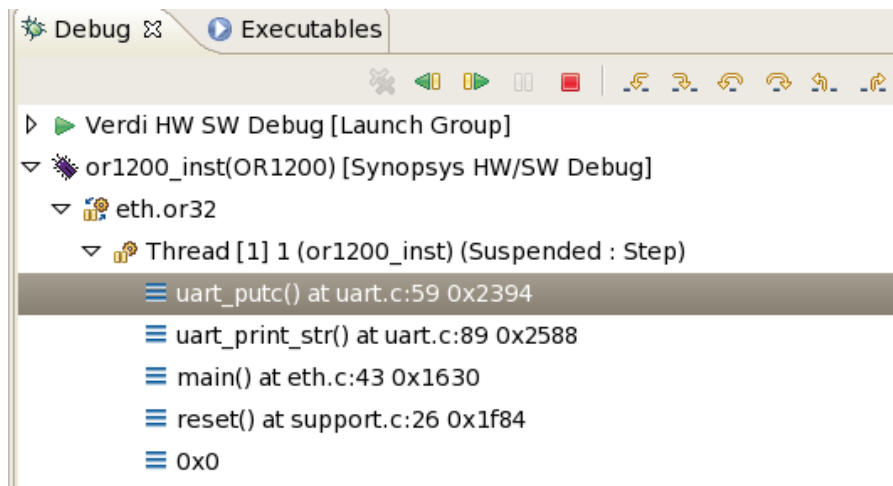
Now we are familiar with the basic interactive post-processing features, let's return to debug the missing characters from the UART output stream.

Use the Run Forward, Run Backward, Step Over, Step Back, Step Into buttons to step through the C-code to the location of the code that is writing the characters to the UART peripheral (inside the `uart_print_str()` routine).



```

53
54 void uart_putc(char c)
55 {
56     unsigned char lsr;
57
58     WAIT_FOR_THRE;
59     REG8(UART_BASE + UART_TX) = c;
60     WAIT_FOR_XMITR;
61 }
  
```



Select various routines in the stack window to see how the source is displayed.

This stack data is typically difficult to obtain, as you would manually decode the stack data from memory and associate the memory values with the symbol names. It is possible, but quite tedious and few people in a SoC team can do this, limiting deployment of software-driven testing.

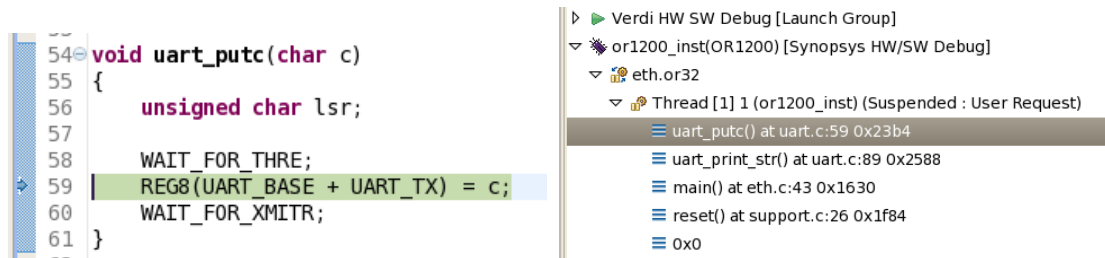
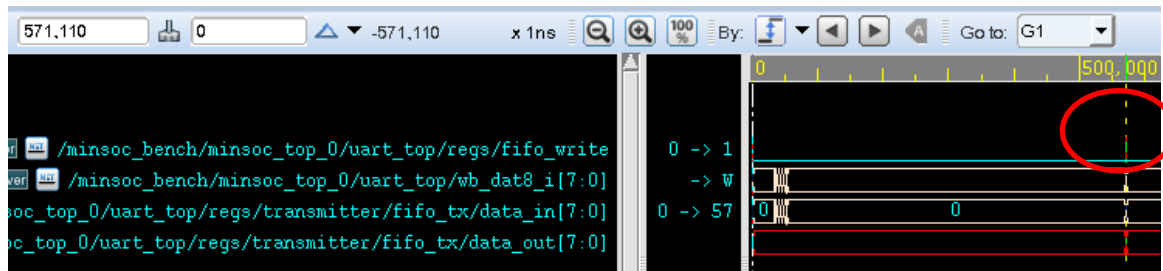
7.3 Waveform View

In this case, we were able to find the C-code that wrote to the UART easily, however sometimes it is not this easy, and we need to find the time when a character is written into the UART register.

Run backwards to `uart_init` in `main.c`, or run backwards multiple times until you reach time 0 (look in the Verdi windows to see the current time).

Re-arrange the windows so you can see the Verdi waveform view and the Eclipse view at the same time. This will be challenging on a laptop display, we don't need to see much in view, just some waves and the source-code window in Eclipse.

Now select one of the `fifo_write` edges in the Verdi waveform view. Note how the Eclipse view advances to the same point in time, and shows the code that is performing the write to the UART.



Select some other times in the Verdi views, including the next fifo_write edge. Note how the Eclipse view is always synchronized.

7.4 Debug of characters

So why are the characters wrong in the log-file?

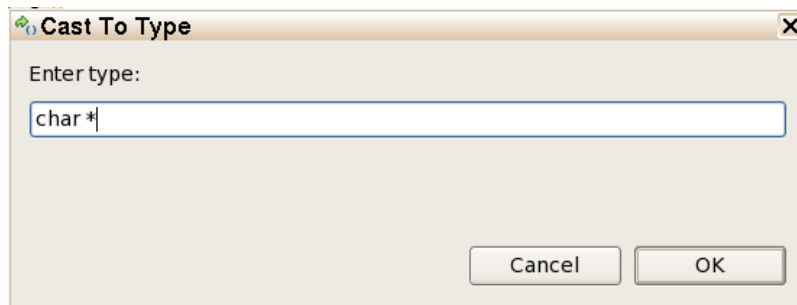
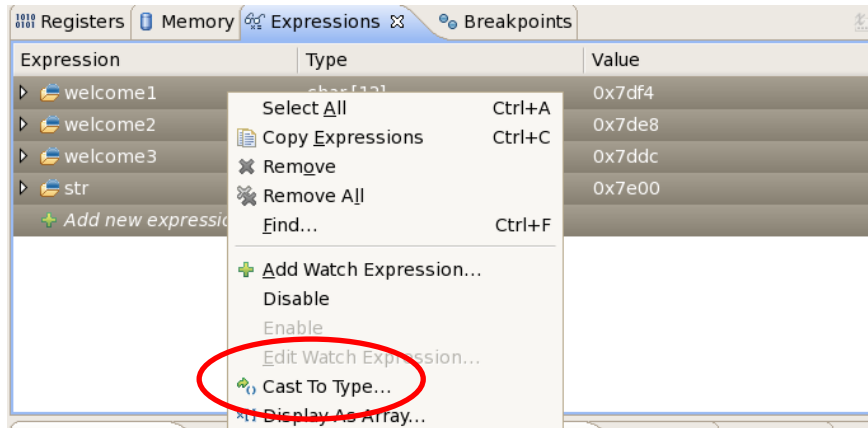
Let's view the characters in the C-code and try to see why it's wrong.

It is useful to see the strings for welcome1, welcome2, welcome3 and str.

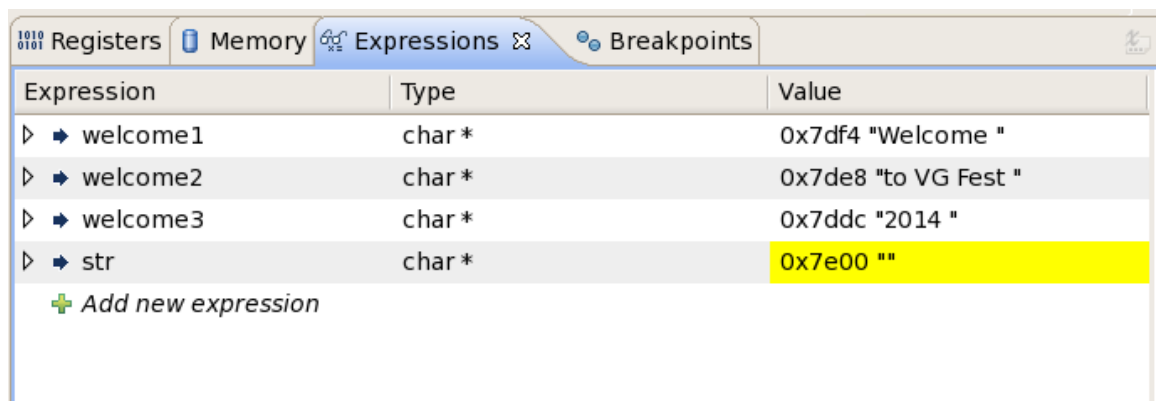
Add each of these to the expression window, by selecting the "+" icon and entering the name of each variable.

Expression	Type	Value
welcome1	char [12]	0x7df4
welcome2	char [12]	0x7de8
welcome3	char [12]	0x7ddc
str	char [8]	0x7e00
+ Add new expression		

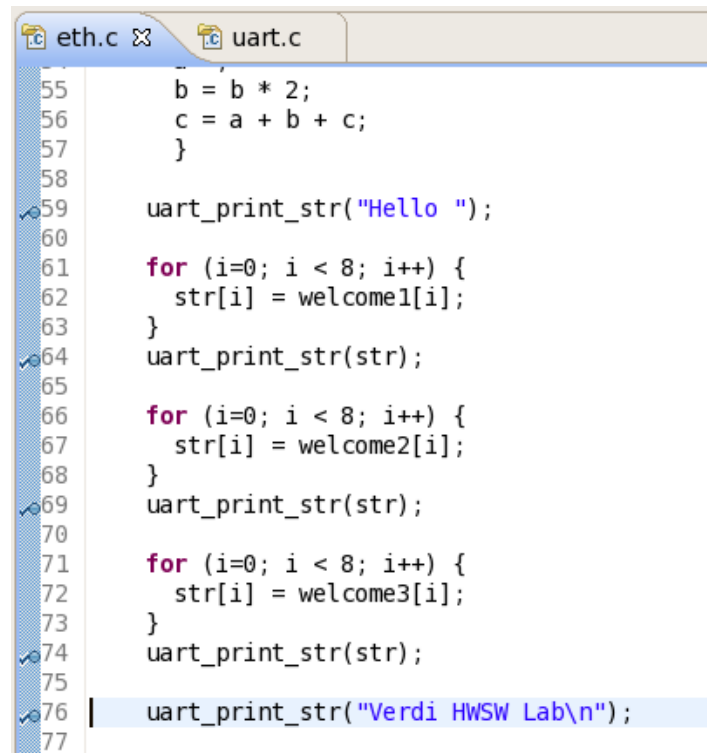
We really want to see the contents of the strings and not the address of the string, so select all 4 lines in the expression window (ctrl-A), then popup-menu/right-mouse-button and choose "Cast to type" and enter "char *" in the dialog.



Now you can see the strings, and they all look OK.



Add breakpoints into the C-source code at each of the `uart_print_str()` lines, and run forward and backward until you notice something strange with the strings.



```
55     b = b * 2;
56     c = a + b + c;
57 }
58
59 uart_print_str("Hello ");
60
61 for (i=0; i < 8; i++) {
62     str[i] = welcome1[i];
63 }
64 uart_print_str(str);
65
66 for (i=0; i < 8; i++) {
67     str[i] = welcome2[i];
68 }
69 uart_print_str(str);
70
71 for (i=0; i < 8; i++) {
72     str[i] = welcome3[i];
73 }
74 uart_print_str(str);
75
76 | uart_print_str("Verdi HWSW Lab\n");
77
```

Now you should be able to see where the bug is in the C-code. Imagine trying to find this typo looking at the opcode running in the CPU...

8. Conclusion

The Verdi HW/SW Debug solution was easy to install and configure for our SoC design as the recorders and instructions are provided in the release tree. The productivity gains using an industry standard GDB/Eclipse debugger to debug embedded software from our simulation environment was enormous. It enabled us to quickly migrate from a low-level trace-file, object dump, disassembly and waveform views to a modern efficient debug environment. The productivity gain makes this a “cannot live without” tool for anyone debugging embedded software.