

# Dual Core SoC Debug with Synopsys Hardware Software Debug Tool

A. Mark Jesensky  
Andy Sha  
Analog Devices, Inc

September 11, 2014  
SNUG Boston

# Agenda

Brief review of SoC debug methods – Traditional method

Brief review of SoC debug methods – HWSW Debug

Multicore implementation

Debug cases

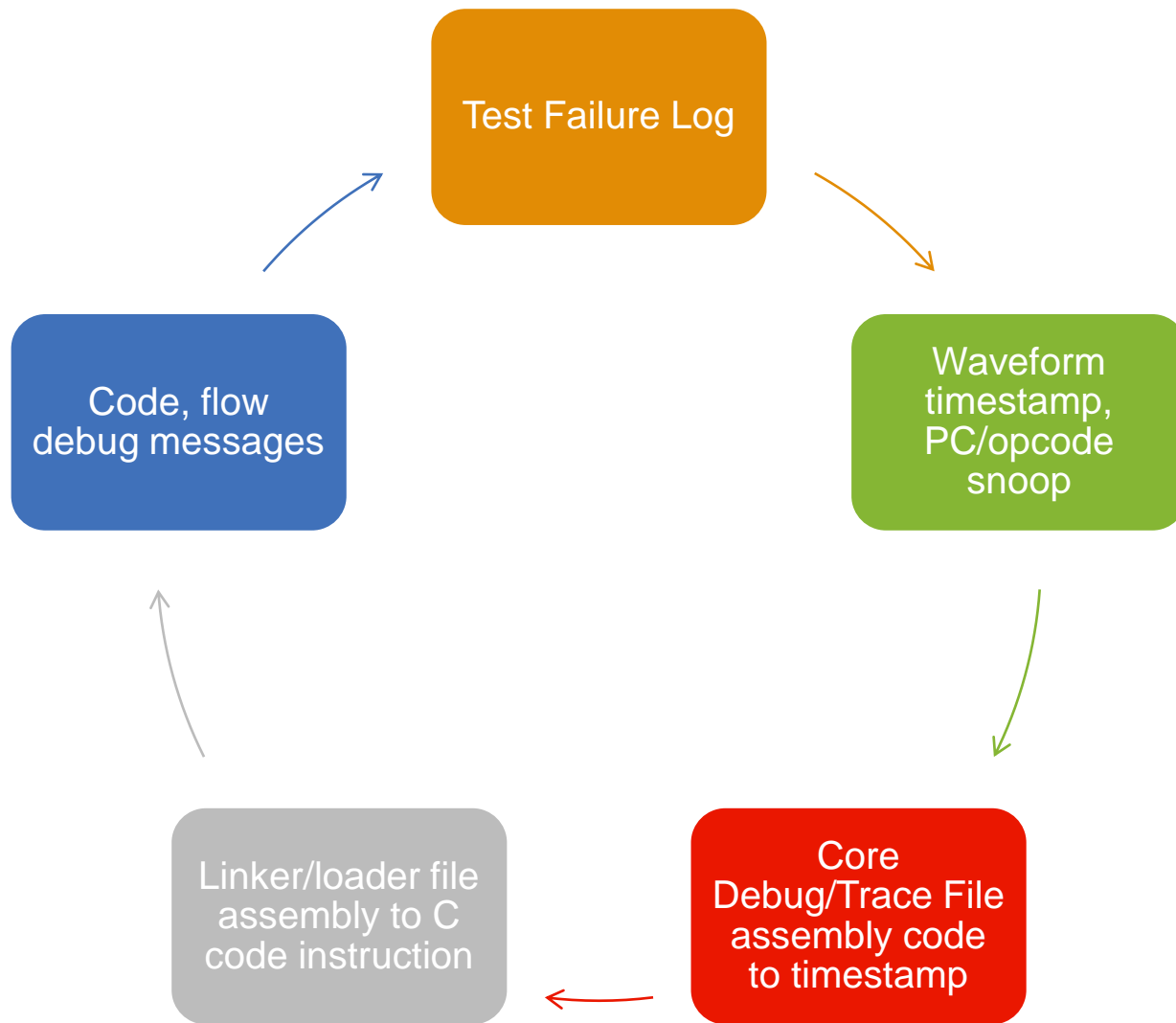
Feature requests

Conclusions

# Brief Review of Debug Methods

## *Traditional Debug*

# SoC Traditional Code Debug



# SoC Debug Files

## *Single Core*

Run log:

Timestamps, debug  
messages

.elf/.out File:

C code, assembled  
core code, memory  
segments

Waveform file:

Internal signal  
waves, timestamps

Debug/Trace File :

Core register and  
pipeline information,  
with timestamps

# SoC Debug Files

*Multi Core*

Run log:

Timestamps, debug  
messages

.elf/.out File n:

.elf/.out File2:

.elf/.out File1:

C code, assembled  
core code, memory  
segments

Waveform file:

Internal signal  
waves, timestamps

Debug/Trace File

Core Debug/Trace File

Debug/Trace File n

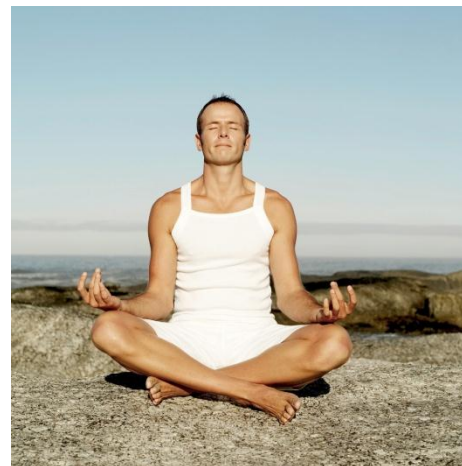
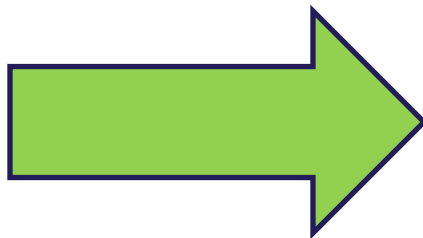
Core register and  
pipeline information,  
with timestamps

# More Cores, More challenges

- SoC have become a blend of reuse blocks, IP black boxes, and custom fabrics often coming from unrelated groups
- Complexity drives debug times longer
  - Have you ever heard of a more complex design with a lower debug time?
- The right person for debug
  - Get all verification engineers experts on all aspects
  - Have an expert who goes from problem to problem

# More cores, better solution

- Have the knowledgeable verification engineer do one setup of the project in a tool that multiple others can use
- Combine all of the debug files into one interface
- Automate as much of the cross referencing as possible
- More importantly, find a way to compress hours of test debug into minutes



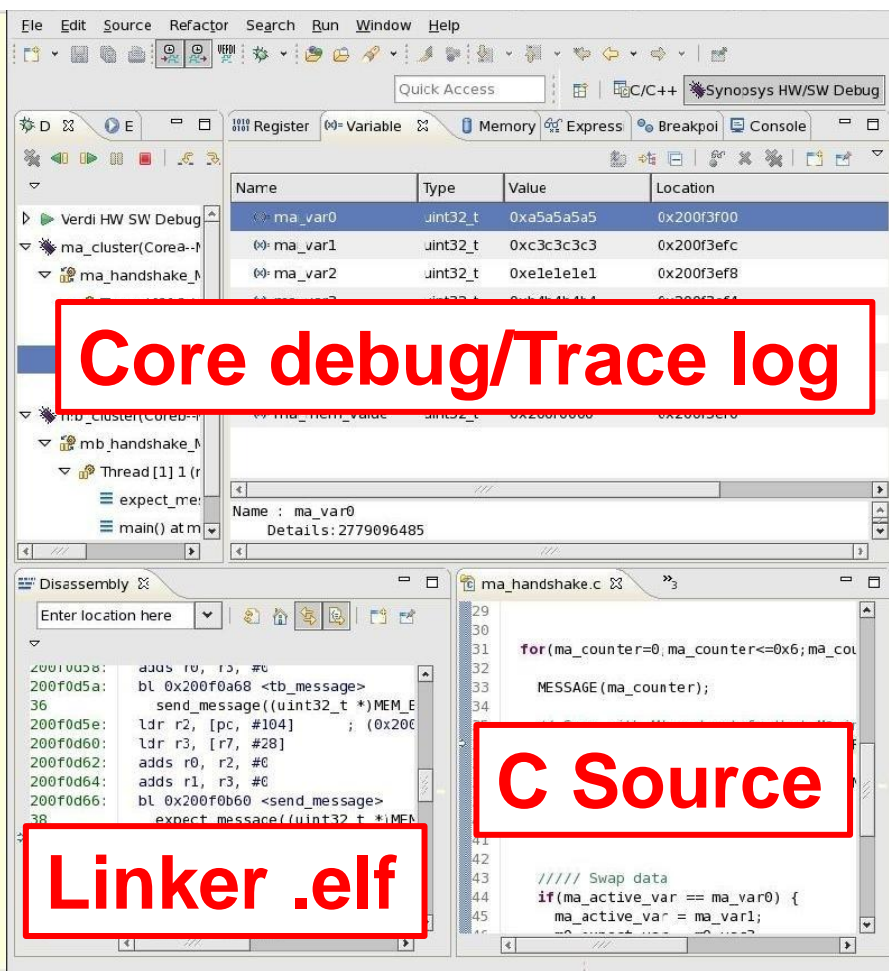
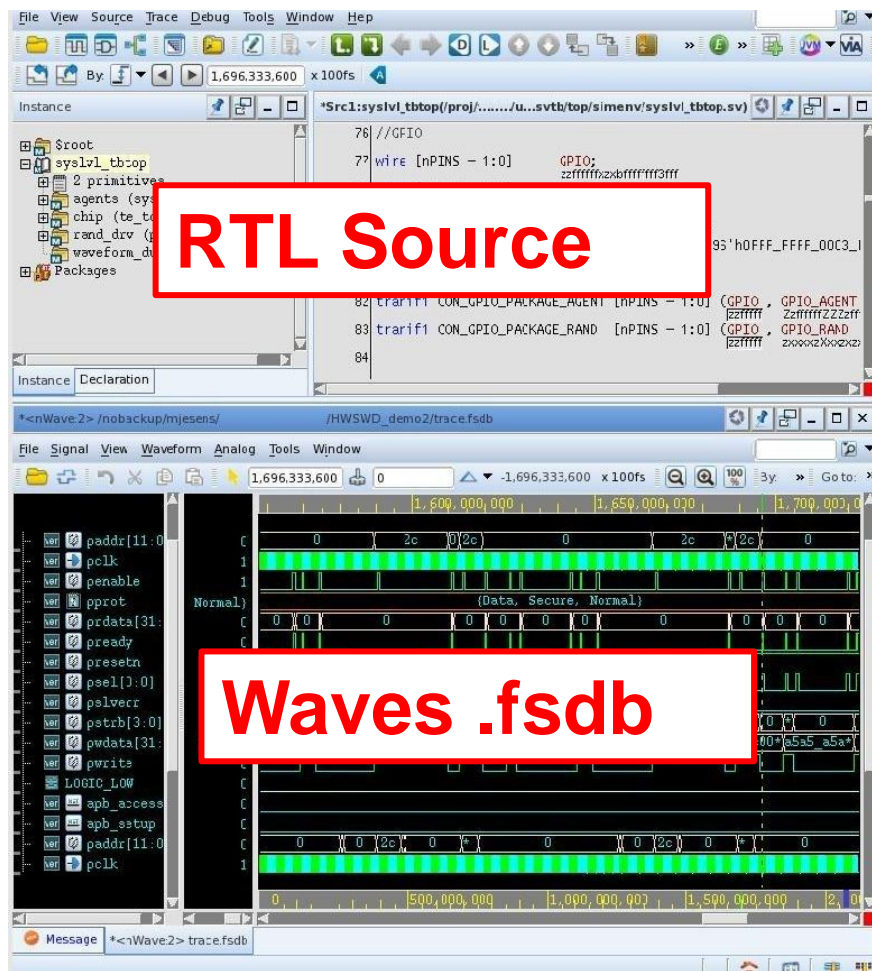


# Brief Review of Debug Methods

*HWSW Debug Tool*

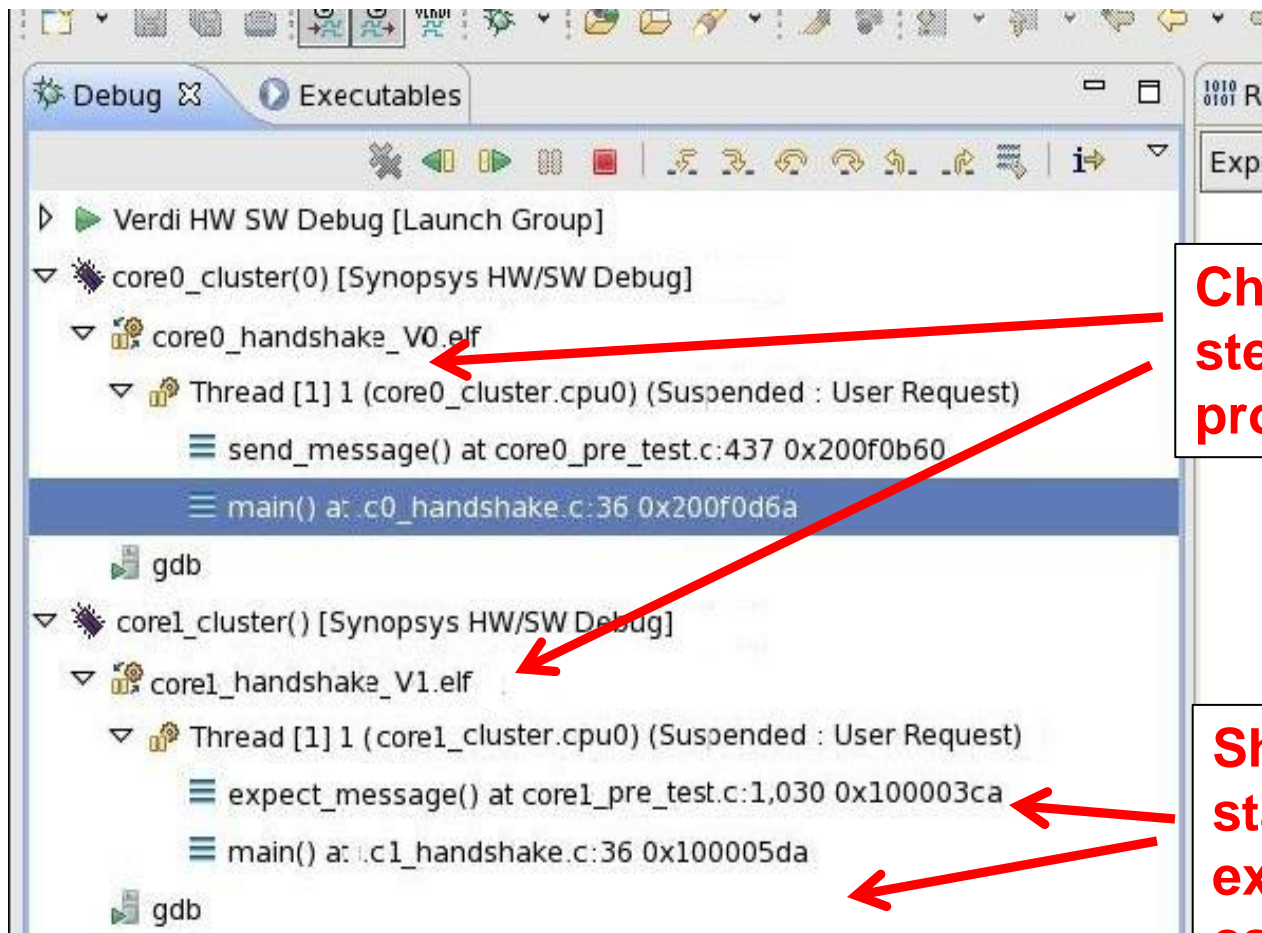
# Verdi HW SW Debug Tool

*Combines and links all of these files into GUIs*



# Hardware Software Tool GUI

## *Scope and Stepping – choosing a processor*



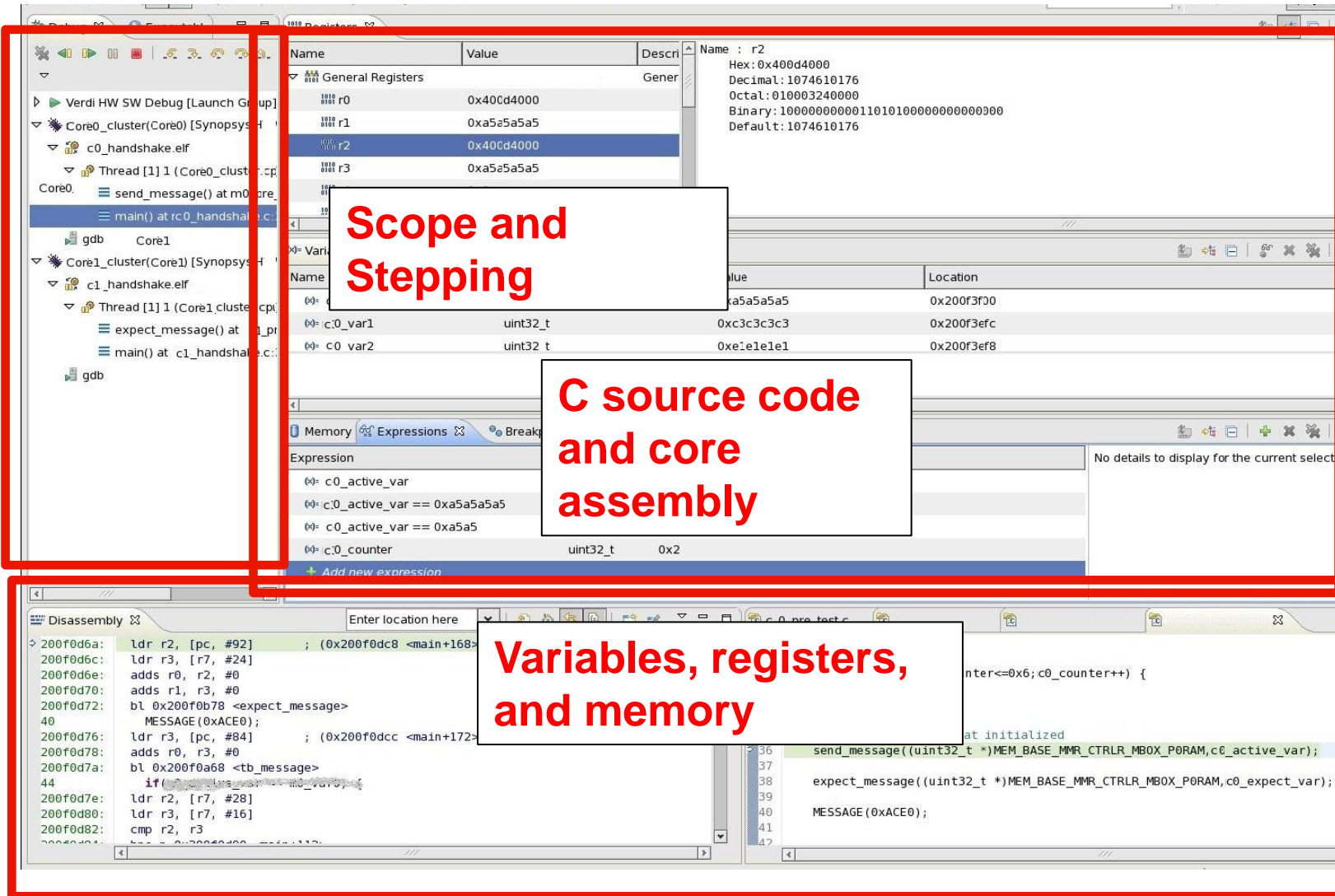
**Choose scope of stepping (specific processor)**

**Show current call stack, such as: expect\_message(), called from main()**



# Hardware Software Tool GUI

## *Multi Core Debug in a single GUI*



The screenshot displays the Hardware Software Tool GUI with several key components highlighted by red boxes and text overlays:

- Scope and Stepping:** A red box highlights the left-hand project tree, showing the hierarchy of cores and threads. A text overlay "Scope and Stepping" is placed over this area.
- C source code and core assembly:** A red box highlights the bottom section of the GUI, which displays disassembled code and C source code. A text overlay "C source code and core assembly" is placed over this area.
- Variables, registers, and memory:** A red box highlights the central area of the GUI, which displays a table of registers (r0, r1, r2, r3) and their values. A text overlay "Variables, registers, and memory" is placed over this area.

The registers table shows the following data:

Name	Value	Description
General Registers		Gener
r0	0x40cd4000	
r1	0xa5a5a5a5	
r2	0x40cd4000	
r3	0xa5a5a5a5	

The disassembly view shows the following instructions:

```

200f0d6a: ldr r2, [pc, #92] ; (0x200f0dc8 <main+168>
200f0d6c: ldr r3, [r7, #24]
200f0d6e: adds r0, r2, #0
200f0d70: adds r1, r3, #0
200f0d72: bl 0x200f0b78 <expect_message>
40 MESSAGE(0xA000);
200f0d76: ldr r3, [pc, #84] ; (0x200f0dcc <main+172>
200f0d78: adds r0, r3, #0
200f0d7a: bl 0x200f0a68 <tb_message>
44 if (c0_active_var == 0xa5a5a5a5) {
200f0d7e: ldr r2, [r7, #28]
200f0d80: ldr r3, [r7, #16]
200f0d82: cmp r2, r3
  
```

# Main drawback of the tool:

- Name is too clunky:

Verdi Hardware Software Debug Tool

- No cool name: DaVinci, Certitude, Hector
- No Three Letter Acronym: DVE, VCS
- VHWSWDT doesn't roll off the tongue
- Tried opening a support ticket for this, was unsuccessful

# Multicore Implementation

# Waveform, Core Debug, Linker files

- No difference for these from single core Verdi HW/SW Debug Tool Implementation
- .fsdb waveforms
  - Dumpvars with all switches and options
- Linker files
  - With debug info and source file paths
- Trace logs:
  - ARM instantiated, with Tarmac switches
  - Custom trace logs



# Recorder Module

```
`include "verdiRecorderCore0.svp"  
`include "verdiRecorderCore1.svp"
```

**Include one recorder per core type**

```
module verdiHwsWDebugTop;  
  
Core0_cluster Core0_cluster();  
Core1_cluster Core1_cluster();  
  
    // FSDB dumper: change FSDB file name if needed.  
    initial #0 $fsdbDumpvars(0,"verdiHwsWDebugTop","+all","+parameter");  
`VERDI_HWSW_INIT_TOP  
endmodule // verdiHwsWDebugTop  
  
module Core0_cluster();  
    verdiRecorderCore0 #(.clusterId(0), .cpuId(0), .tarmacFileName("dfile_Core0.log"))  
    cpu0(.cpuClock(1'b0));  
endmodule  
  
module Core1_cluster();  
    verdiRecorderCore1 #(.clusterId(1), .cpuId(0), .tarmacFileName("dfile_Core1.log"))  
    cpu0(.cpuClock(1'b0));  
endmodule
```



# Recorder Module

```
`include "verdiRecorderCore0.svp"  
`include "verdiRecorderCore1.svp"
```

```
module verdiHwsWDebugTop;
```

```
Core0_cluster Core0_cluster();  
Core1_cluster Core1_cluster();
```

**Instantiate one per core in design**

```
// FSDB dumper: change FSDB file name if needed.
```

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

```
`VERDI_HWSW_INIT_TOP
```

```
endmodule // verdiHwsWDebugTop
```

```
module Core0_cluster();
```

```
verdiRecorderCore0 #(.clusterId(0), .cpuId(0), .tarmacFileName("dfile_Core0.log"))  
cpu0(.cpuClock(1'b0));
```

```
endmodule
```

```
module Core1_cluster();
```

```
verdiRecorderCore1 #(.clusterId(1), .cpuId(0), .tarmacFileName("dfile_Core1.log"))  
cpu0(.cpuClock(1'b0));
```

```
endmodule
```

# Recorder Module

```
`include "verdiRecorderCore0.svp"  
`include "verdiRecorderCore1.svp"
```

```
module verdiHwsWDebugTop;
```

```
Core0_cluster Core0_cluster();
```

**Waves the same as single core**

```
Core1_cluster Core1_cluster();
```

```
// FSDB dumper: change FSDB file name if needed.
```

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

```
`VERDI_HWSW_INIT_TOP
```

```
endmodule // verdiHwsWDebugTop
```

```
module Core0_cluster();
```

```
verdiRecorderCore0 #(.clusterId(0), .cpuId(0), .tarmacFileName("dfile_Core0.log"))
```

```
cpu0(.cpuClock(1'b0));
```

```
endmodule
```

```
module Core1_cluster();
```

```
verdiRecorderCore1 #(.clusterId(1), .cpuId(0), .tarmacFileName("dfile_Core1.log"))
```

```
cpu0(.cpuClock(1'b0));
```

```
endmodule
```

# Recorder Module

```
`include "verdiRecorderCore0.svp"
`include "verdiRecorderCore1.svp"

module verdiHwsWDebugTop;

Core0_cluster Core0_cluster();
Core1_cluster Core1_cluster();

    // FSDB dumper: change FSDB file name if needed.
    initial #0 $fsdbDumpvars(0,"verdiHwsWDebugTop","+all","+parameter");
`VERDI_HWSW_INIT_TOP
endmodule // verdiHwsWDebugTop
```

Define modules for each core

```
module Core0_cluster();
    verdiRecorderCore0 #(.clusterId(0), .cpuId(0), .tarmacFileName("dfile_Core0.log"))
    cpu0(.cpuClock(1'b0));
endmodule
```

```
module Core1_cluster();
    verdiRecorderCore1 #(.clusterId(1), .cpuId(0), .tarmacFileName("dfile_Core1.log"))
    cpu0(.cpuClock(1'b0));
endmodule
```

# Further Multicore Implementation

```
/// Add includes for all core types
`include "verdiRecorderCore0.svp"
`include "verdiRecorderCore1.svp"
....
`include "verdiRecorderCoreN.svp"
```

Include recorders for  
each core TYPE

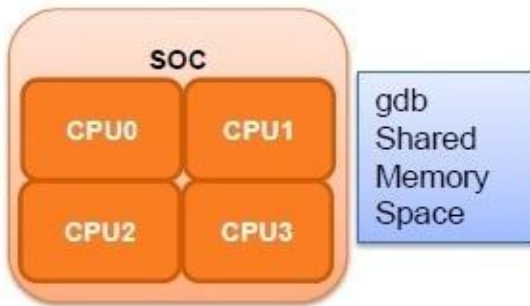
```
module verdiHwsWDebugTop;
```

```
/// Instantiate each core
```

```
Core0_cluster Core0_cluster();
Core1_cluster Core1_cluster();
....
Core1_cluster CoreN_cluster();
```

Instantiate each core

# Further Multicore Implementation

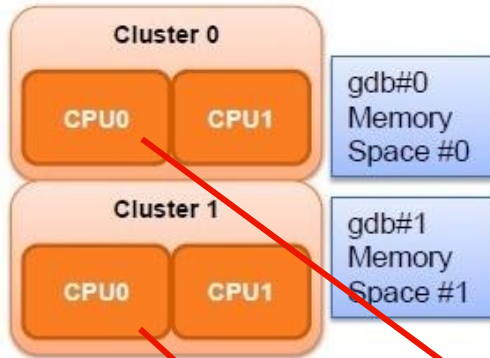


- If cores have shared memory that they are executing from, and a single linker file, then they will be the same clusterID and different cpuIDs.

```
module Core0_cluster();  
    verdiRecorderCore0 #(.clusterId(0), .cpuId(0),  
        .tarmacFileName("dfile_dbg_drv_Core0.log")) cpu0(.cpuClock(1'b0));  
endmodule
```

```
module Core1_cluster();  
    verdiRecorderCore1 #(.clusterId(0), .cpuId(1),  
        .tarmacFileName("dfile_dbg_drv_Core1.log")) cpu0(.cpuClock(1'b0));  
Endmodule
```

# Further Multicore Implementation



- If cores have separate memory, they have separate linker files, and they will be separate clusterIDs.

```
module Core0_cluster();  
    verdiRecorderCore0 #(.clusterId(0), .cpuId(0),  
        .tarmacFileName("dfile_dbg_drv_Core0.log")) cpu0(.cpuClock(1'b0));  
endmodule
```

```
module Core1_cluster();  
    verdiRecorderCore1 #(.clusterId(1), .cpuId(0),  
        .tarmacFileName("dfile_dbg_drv_Core1.log")) cpu0(.cpuClock(1'b0));  
endmodule
```

# Debug Cases

# Memory Aliasing

- The code for Core0 executes an output to some flags on the processor to track execution flow. Core1 had some randomized local memory write/reads
- In the flow, the Core0 execution became corrupted
- A day was spent on this with traditional debug methods without any positive result
- Depending on how much additional code was added for debug, the test would pass correctly, pass prematurely, or fail as before



# Memory Aliasing (con't)

- Using the HW/SW Debug Tool, Core0 was stepped through and the PC was monitored, but the test just ended on a fault
- The HW/SW Tool's message window was flagged with an unexpected memory location change, which detailed the time at which an address in the instruction memory was changing

```
GDB connection established on port 1024.
```

```
Warning-[HwSwDbgMemPred] @7077400fs: The value read at address 0x200f4d56 by core0_cluster.cpu0 (=0xa5a5)  
is different from the last value written by the CPU to this address (=0x0041).
```

```
This could indicate a change of the memory not issued by a CPU, for example:
```

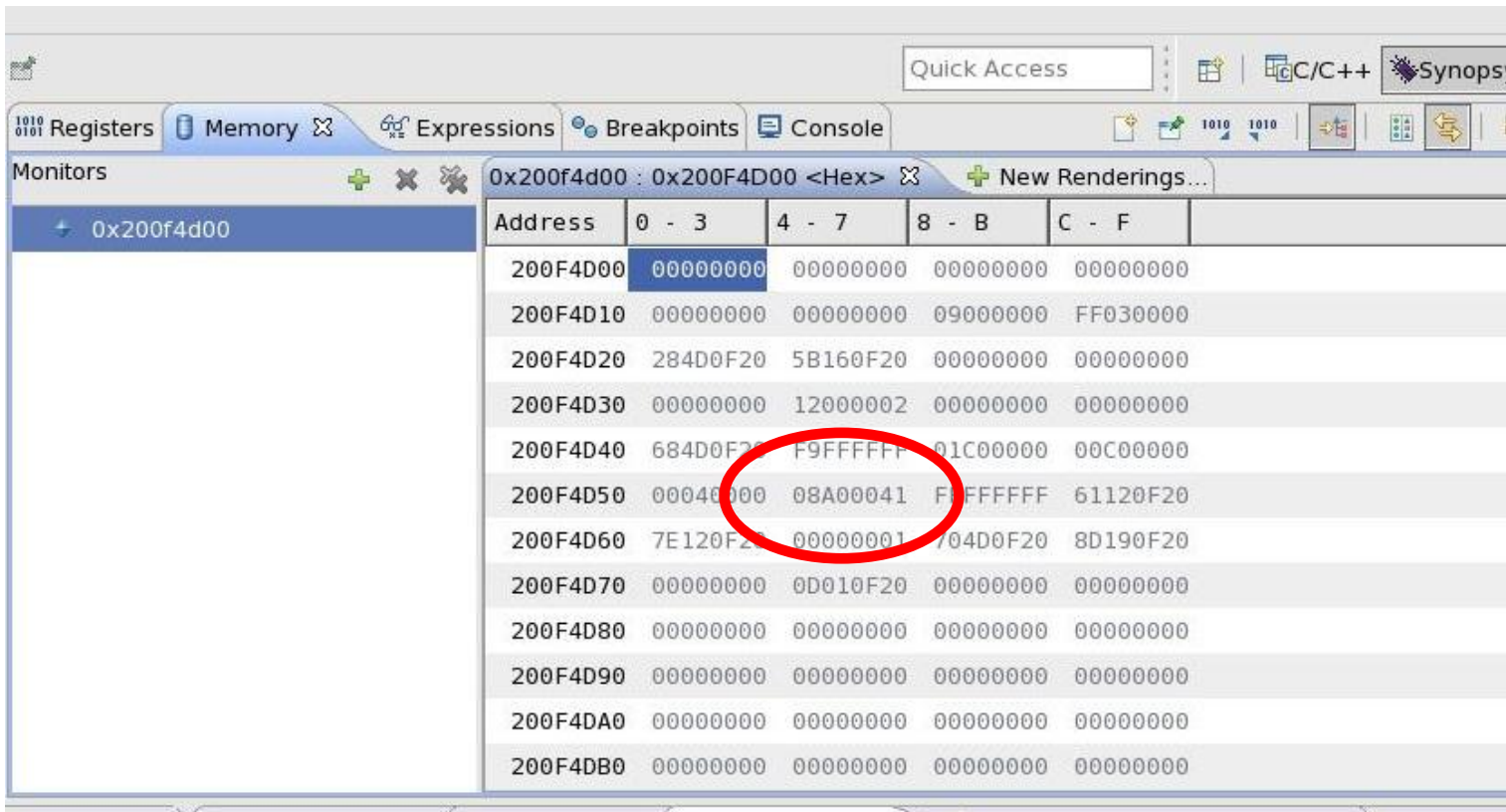
- a DMA writing to the memory.
- a change in a memory mapped HW register.
- a cache inconsistency.

```
Further warnings for this address will be suppressed.
```

- Using the Verdi waveform, an expression was added to the memory interface address pins to flag when this location was accessed. It happened more often than expected

# Memory Aliasing (con't)

One step before the offending memory access occurs.



Quick Access | C/C++ | Synopsys

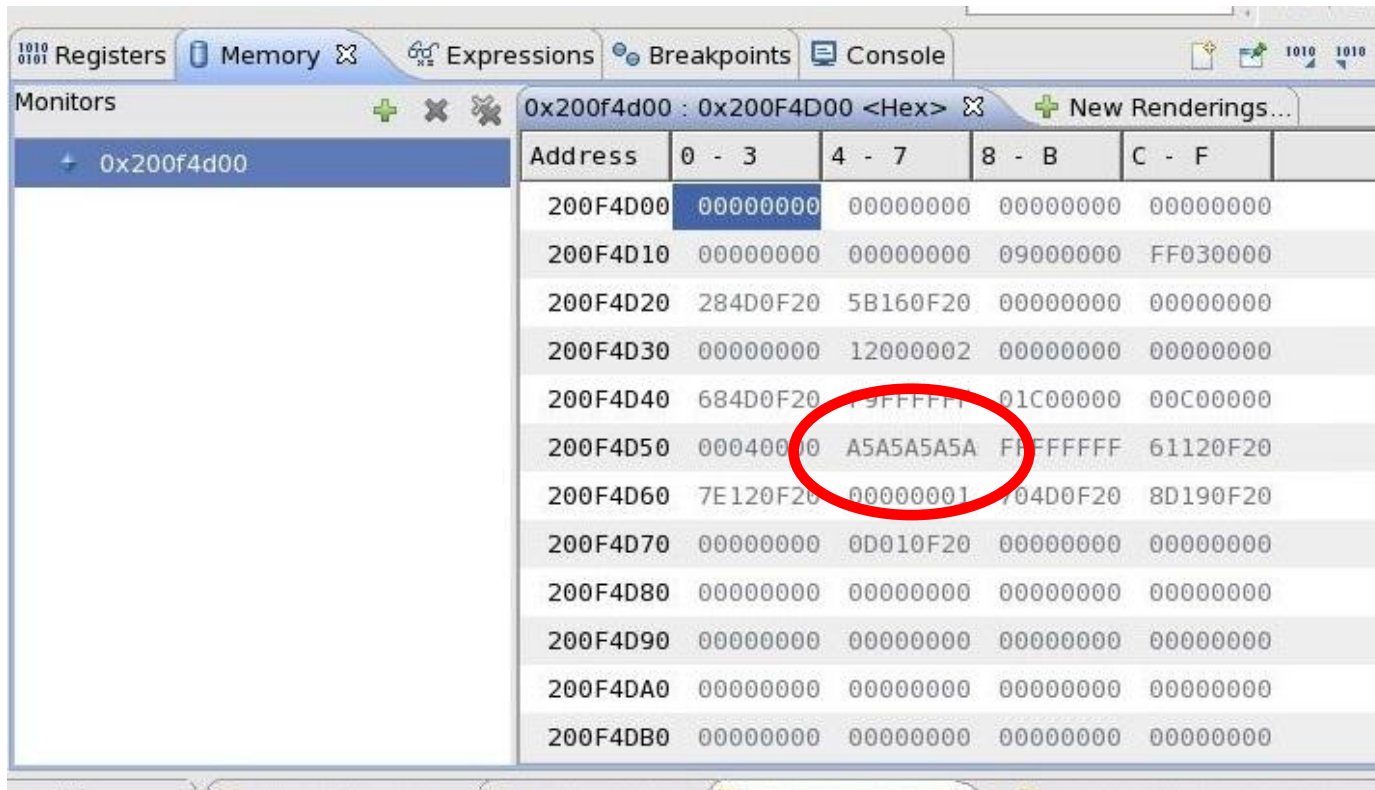
Registers | **Memory** | Expressions | Breakpoints | Console

Monitors | 0x200f4d00 : 0x200F4D00 <Hex> | New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
200F4D00	00000000	00000000	00000000	00000000
200F4D10	00000000	00000000	09000000	FF030000
200F4D20	284D0F20	5B160F20	00000000	00000000
200F4D30	00000000	12000002	00000000	00000000
200F4D40	684D0F20	F9FFFFFF	01C00000	00C00000
200F4D50	00040000	08A00041	FFFFFFFF	61120F20
200F4D60	7E120F20	00000001	704D0F20	8D190F20
200F4D70	00000000	0D010F20	00000000	00000000
200F4D80	00000000	00000000	00000000	00000000
200F4D90	00000000	00000000	00000000	00000000
200F4DA0	00000000	00000000	00000000	00000000
200F4DB0	00000000	00000000	00000000	00000000

# Memory Aliasing (con't)

One step after the offending memory access occurs.



Monitors

0x200f4d00 : 0x200F4D00 <Hex>

Address	0 - 3	4 - 7	8 - B	C - F
200F4D00	00000000	00000000	00000000	00000000
200F4D10	00000000	00000000	09000000	FF030000
200F4D20	284D0F20	5B160F20	00000000	00000000
200F4D30	00000000	12000002	00000000	00000000
200F4D40	684D0F20	79FFFFFF	01C00000	00C00000
200F4D50	00040000	A5A5A5A5	FFFFFFFF	61120F20
200F4D60	7E120F20	00000001	704D0F20	8D190F20
200F4D70	00000000	0D010F20	00000000	00000000
200F4D80	00000000	00000000	00000000	00000000
200F4D90	00000000	00000000	00000000	00000000
200F4DA0	00000000	00000000	00000000	00000000
200F4DB0	00000000	00000000	00000000	00000000

# Memory Aliasing Conclusion

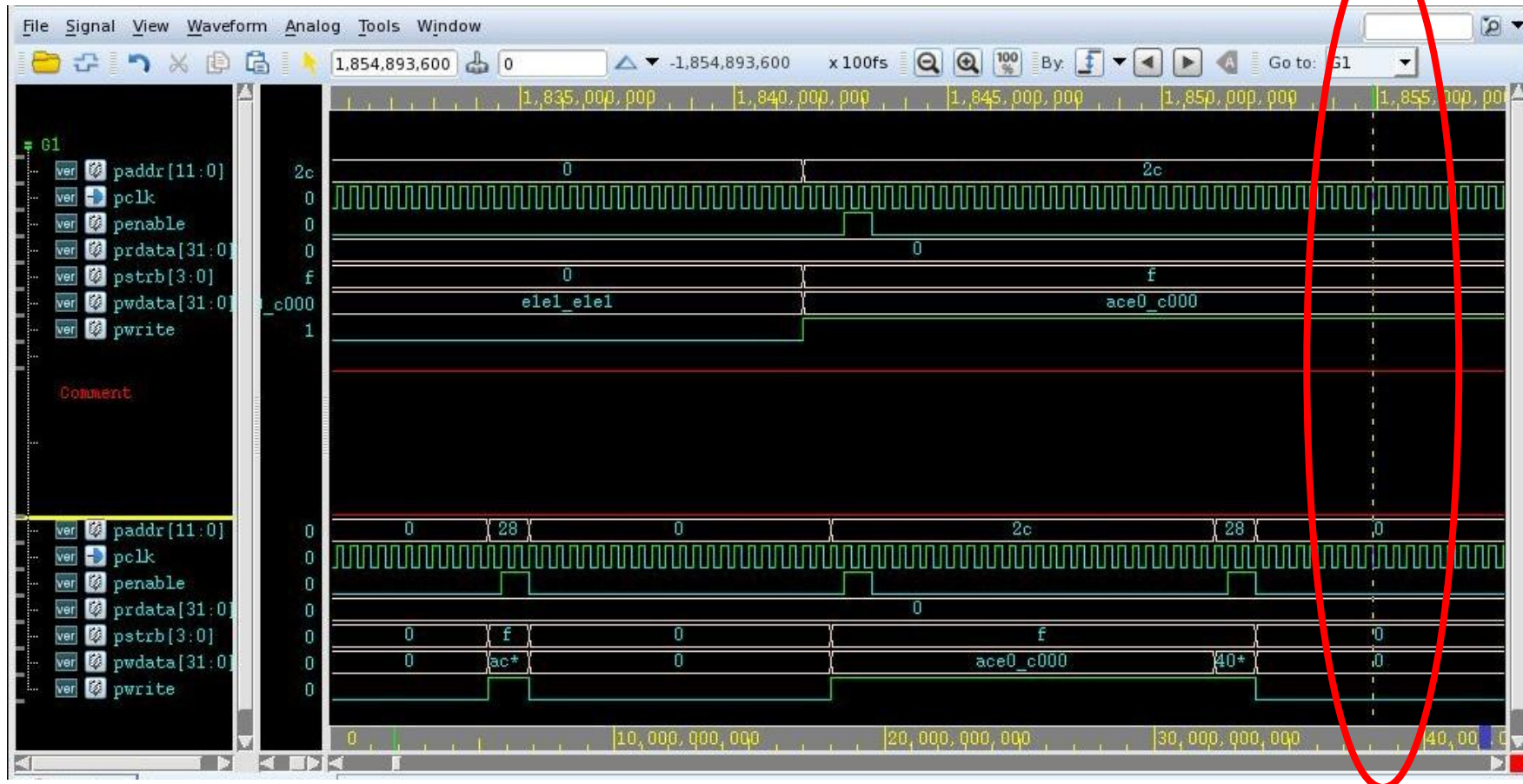
- After stepping through Core1 code, it did not show any access to that memory location, though a local memory access was occurring
- The waveform view showed the corrupted address was seen on the Core1 memory interface
- The address was traced back through the design to an aliasing offset register in Core1 that had been randomized without proper constraints, corrupting the Core0 program stack
- Debug time using the tool was a little over an hour

# Mailbox lockup

- A handshaking protocol between Core0 and Core1 was locking up
- Simple test: send\_message, expect message
- Messages originally printed in the log file showed that traffic in both directions worked properly for a number of transactions and then the execution stalled

# Mailbox lockup (con't)

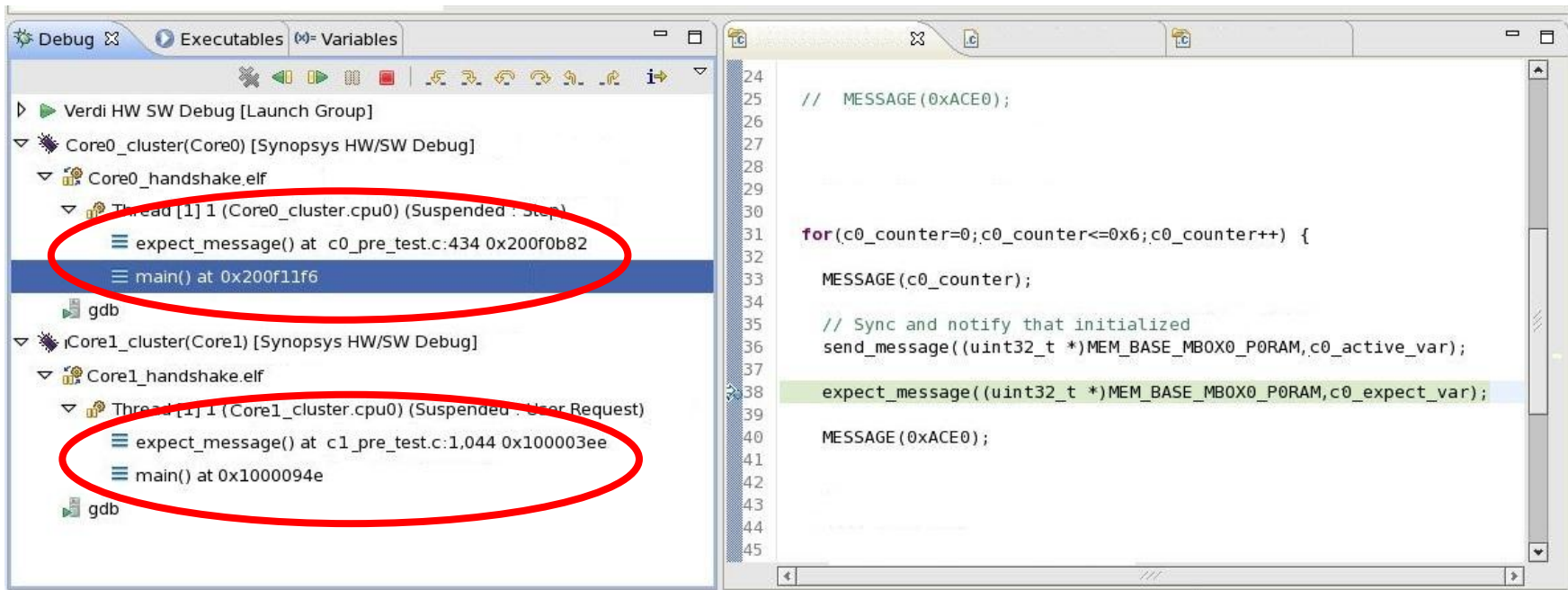
First step was to open the waveform in Verdi and click cursor in waveform at last interesting thing to happen





# Mailbox lockup (con't)

The stack tracing and C code of Core0 and Core1 are below, where both are locked up at the same register access for expect\_message()



The screenshot displays a debug interface with two main panels. The left panel shows the stack trace for two cores, Core0 and Core1, both of which are suspended. The right panel shows the C code for Core0, with the line `expect_message((uint32_t *)MEM_BASE_MBOX0_P0RAM, c0_expect_var);` highlighted, indicating the point of lockup.

**Core0 Stack Trace:**

- Verdi HW SW Debug [Launch Group]
  - Core0\_cluster(Core0) [Synopsys HW/SW Debug]
    - Core0\_handshake.elf
      - Thread [1] 1 (Core0\_cluster.cpu0) (Suspended : Step)
        - expect\_message() at c0\_pre\_test.c:434 0x200f0b82
        - main() at 0x200f11f6
      - gdb

**Core1 Stack Trace:**

- Core1\_cluster(Core1) [Synopsys HW/SW Debug]
  - Core1\_handshake.elf
    - Thread [1] 1 (Core1\_cluster.cpu0) (Suspended : User Request)
      - expect\_message() at c1\_pre\_test.c:1,044 0x100003ee
      - main() at 0x1000094e
    - gdb

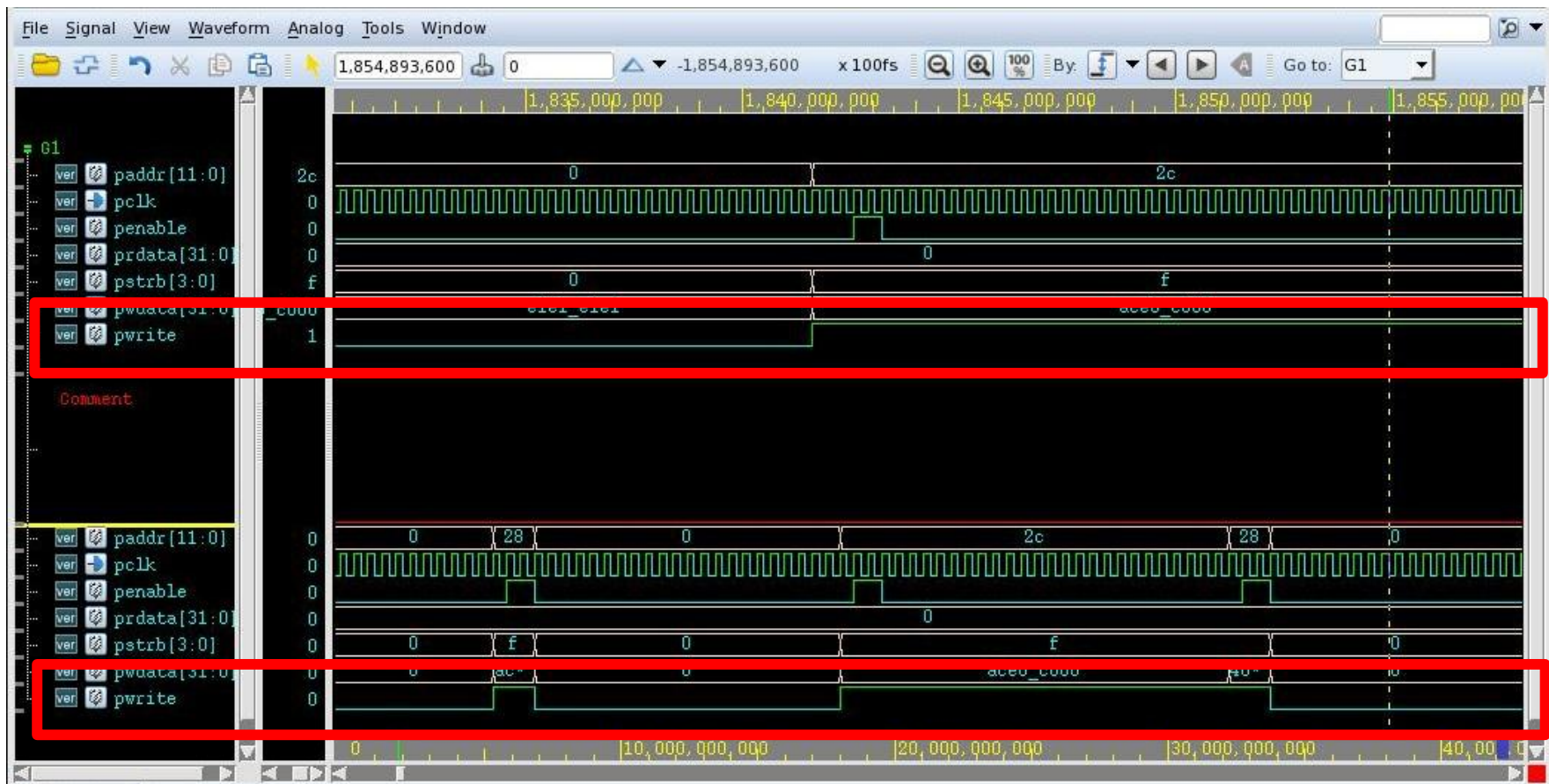
**C Code Snippet (Core0):**

```

24
25 // MESSAGE(0xACE0);
26
27
28
29
30
31 for(c0_counter=0;c0_counter<=0x6;c0_counter++) {
32
33     MESSAGE(c0_counter);
34
35     // Sync and notify that initialized
36     send_message((uint32_t *)MEM_BASE_MBOX0_P0RAM,c0_active_var);
37
38     expect_message((uint32_t *)MEM_BASE_MBOX0_P0RAM,c0_expect_var);
39
40     MESSAGE(0xACE0);
41
42
43
44
45
  
```

# Mailbox lockup (con't)

We can see that the write signal never deasserts for Core0, while Core1 functioned as expected. We single stepped through the assembly to see where the Core0 write happens.





# Mailbox lockup Conclusion

- Core1 wrote it's data and then went to the expect wait state while the Core0 RTL fabric was waiting for arbitration from it's own write register access
- But after the write out to the fabric Core 0 execution continued on.
- Stepping through the code, we could see where Core0 sent out the data and the bus was stalled without feedback to this RTL logic
- The subsequent "expect\_data" read caused the bus to lockup.

# Compiler Issue

- A test was running to timeout on Core0, and messages showed it was stuck in a status check loop.
- The interesting note is that executing on Core1, the test **passed** (same code, same peripheral, etc)
- Based on this, first suspect was Core0 fabric logic
- Stepping into the loop with the HW/SW Debug Tool the status bit was observed being read out but it was not triggering the check.

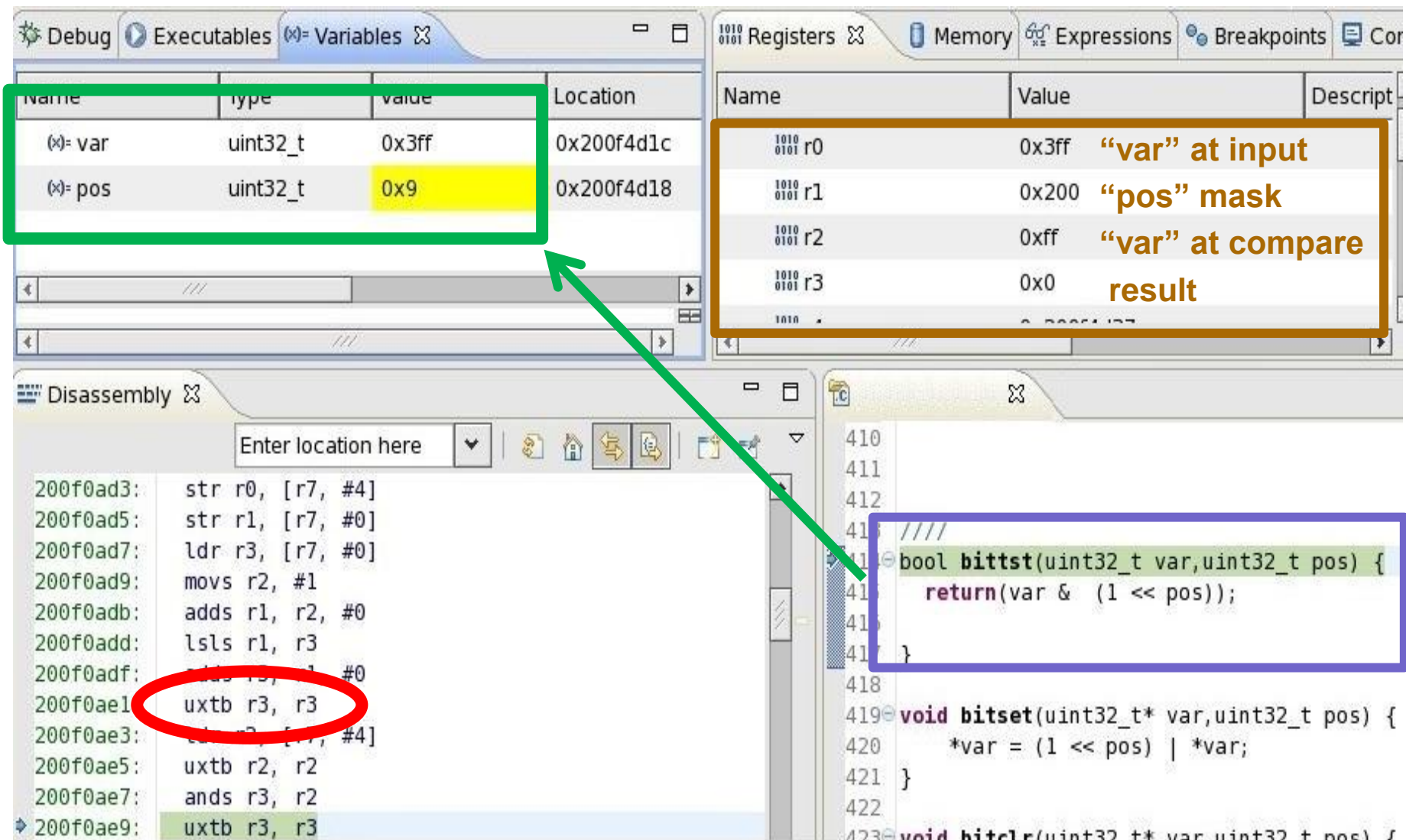
# Compiler Issue (con't)

```
21
22 void UART0_STAT_IVSR() {
23     MESSAGE(0xc003);
24     uint32_t var0 = 0x0000 ;
25     uint32_t var1 = 0x0000 ;
26     bool cc0;
27
28
29
30
31     cc0 = bittst(*pREG_UART0_STAT, BITP_UART_STAT_OE);
32     if(!cc0) goto UART0_NEXT_CHECK_1;
33     *pPTR_UART0_RXCORE_DIS_CHECK = 0x0001;
34     cc0 = bittst(*pREG_UART0_IMSK, BITP_UART_STAT_OE);
35     if(!cc0) goto UART0_NEXT_CHECK_1;
36
37
38
39
...
```

Boolean check bit  
defined

STAT register read  
and bittst function  
call

# Compiler Issue (con't)



**Variables Window:**

Name	Type	Value	Location
(*)= var	uint32_t	0x3ff	0x200f4d1c
(*)= pos	uint32_t	0x9	0x200f4d18

**Registers Window:**

Name	Value	Description
r0	0x3ff	"var" at input
r1	0x200	"pos" mask
r2	0xff	"var" at compare
r3	0x0	result

**Disassembly Window:**

```

200f0ad3: str r0, [r7, #4]
200f0ad5: str r1, [r7, #0]
200f0ad7: ldr r3, [r7, #0]
200f0ad9: movs r2, #1
200f0adb: adds r1, r2, #0
200f0add: lsls r1, r3
200f0adf: adds r3, r1, #0
200f0ae1: uxtb r3, r3
200f0ae3: ldr r2, [r7, #4]
200f0ae5: uxtb r2, r2
200f0ae7: ands r3, r2
200f0ae9: uxtb r3, r3
  
```

**Source Code Window:**

```

410
411
412
413
414 bool bittst(uint32_t var, uint32_t pos) {
415     return(var & (1 << pos));
416 }
417
418
419 void bitset(uint32_t* var, uint32_t pos) {
420     *var = (1 << pos) | *var;
421 }
422
423 void bitclr(uint32_t* var, uint32_t pos) {
  
```

# Compiler Issue Conclusion

- Comparing to Core1 there were assembly differences for such a simple function
- Core1 could support 32bit boolean checks, while Core0 could only support 8bit boolean checks

## Core0

```
100002da: 683b      ldr r3, [r7, #0]
100002dc: 2201      movs r2, #1
100002de: fa02 f303 lsl.w r3, r2, r3
100002e2: b2da      uxtb r2, r3
100002e4: 687b      ldr r3, [r7, #4]
```

## Core1

```
100004b2: 683b      ldr r3, [r7, #0]
100004b4: f04f 0201 mov.w r2, #1
100004b8: fa02 f303 lsl.w r3, r2, r3
100004bc: 461a      mov r2, r3
100004be: 687b      ldr r3, [r7, #4]
```

- The compiler “helpfully” truncated the data to 8 bits for the compare

~~**Things I whine about like a 2 year old**~~

**Feature Requests**

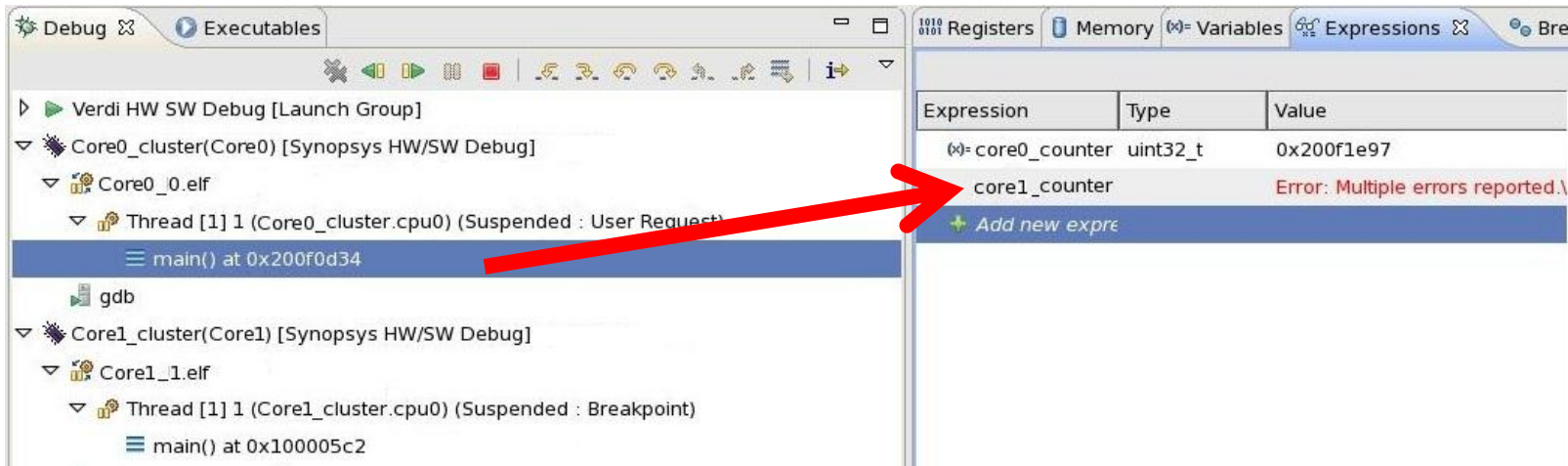
# Great tool, warts and all:

- No tracking of delta time between instruction steps
- Cannot map RTL register information into the Eclipse GUI for same-window debug of registers
- Cannot send C variables to the waveform from the GUI
- No restart trace button
  - Need to set a breakpoint at the start and then resume backwards, or go to line number of startup
- No scope independence for any Eclipse window, even for user defined expressions

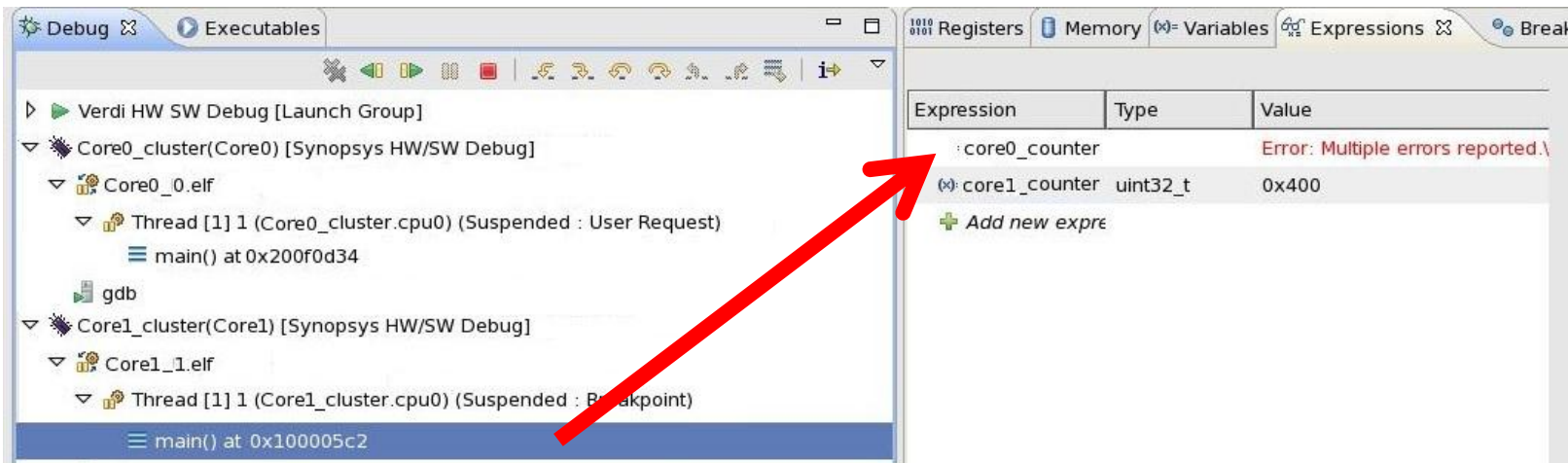


# Scope Error Messages

*Add snapshot of red expression not found in windows*



Expression	Type	Value
core0_counter	uint32_t	0x200f1e97
core1_counter		Error: Multiple errors reported.\



Expression	Type	Value
:core0_counter		Error: Multiple errors reported.\
core1_counter	uint32_t	0x400



# “Worst” Issue With the Tool

- Name is too clunky:

Verdi Hardware Software Debug Tool

- No cool name: DaVinci, Certitude, Hector
- No Three Letter Acronym: DVE, VCS
- VHWSWDT doesn't roll off the tongue

# Conclusion

# Conclusion

## *Who Shouldn't Use the HWSW Debug Tool ?*

- People who really like print messages for debug
- People who need to do neck exercises by switching views between multiple windows
- People who get paid by the hour
- The tooth fairy



## *Who Should Use the HWSW Debug Tool ?*

- People who actually exist



# Conclusion

*Who Should Use the HWSW Debug Tool ?*

- Use Case test writers and verification engineers porting legacy tests
- Algorithm engineers doing debug, optimization, and benchmarking
- RTL designers debugging logic blocks, system fabrics, and SoCs
- Verification engineers who want it all (when things fail)

# Conclusion

## *Results*

- We've found the fault isolation time using this tool versus traditional methods was reduced by about 80%
  - *Save your time for Coverage Closure!*
- The options of debug allow a broader range of user to be in a comfort zone while using it:
  - Waveforms and opcodes for RTL designers
  - C code for test writers and algorithm developers
  - Benchmarkers tracing performance
  - All of these for DV engineers

# Acknowledgements

- Andy Sha (co author), ADI : Implementing the first single core HWSW Debug Tool at ADI, and standardized a lot of the flows
- Alex Wakefield, Synopsys : Providing the templates for the implementation and instruction for even the simplest things I didn't grasp at first
- Dave Brownell, ADI : Assisting with integration into our testbench and sim environment



# Thank You