Methodology for automating coverage-driven interrupt testing of instruction sets

David McConnell
EM Microelectronic-US
David.McConnell@emmicro-us.com

Greg Tumbush
EM Microelectronic-US
Greg.Tumbush@emmicro-us.com

Abstract: This paper will cover a novel method for coverage-driven interrupt verification of CPU cores to obtain 100% functional coverage in minimal time.

I. Introduction

Closing functional coverage is a typically laborious task. The default response of low coverage is more simulation cycles which does not scale as the Device Under Test (DUT) becomes larger and/or more functional coverage is added. This paper presents a methodology for truly coverage-driven interrupt verification. The stimulus timing of a test is automatically modified using scripting and uses only pre-existing firmware. This method is best used as a detailed sanity check after large RTL changes, or as a cron job, to be completed nightly to allow for identifying individual RTL changes that break interrupt functionality. The methodology is designed to execute an interrupt for every instruction once and only once.

II. Background

EM Microelectronic is a subsidiary of the Swatch Group, and has design centers in Switzerland, the Czech Republic, and the United States. Recent projects have involved contributing to the OpenHW group, a not-for-profit organization with a shared goal of developing open source RISC-V CPU IP. EM Microelectronic's contributions to that end have encompassed design, compliance testing of the XPULP extension, CSR verification, and exception verification. Reference 1 has more detail as to the functionality of the first OpenHW core, CV32E40P, and includes a detailed section about the XPULP extension.

Due to customer requirements, EM Microelectronic, while utilizing the majority of the CV32E40P core, implements a different interrupt system. This requires the generation of large amounts of random interrupt stimulus with finite verification resources. As such, a method for generating coverage in an efficient and timely manner was needed while minimizing script development and debugging effort.

An important metric for declaring verification complete is functional coverage. For a CPU a simple functional coverage metric of executing every instruction is easy to obtain. However, covering that every instruction has been interrupted is a significant task, requiring many random simulations. At some point, the default method of simply running more tests breaks down. This paper will cover a novel methodology for automating interrupt coverage closure.

III. Methodology

While useful in all settings, automation of routine tasks is especially important for companies with limited resources. Automation gives the ability to reduce the time spent on routine tasks, such as quickly regathering and reporting coverage after significant RTL changes. It also allows testing patterns to dynamically reconfigure to match given firmware, simplifying handling of addition and removal of firmware from testing, as well as modification of firmware.

In essence, the proposed methodology takes existing directed interrupt tests and firmware, and turns them into a coverage-driven test that leverages existing firmware in order to obtain complete instruction coverage. This is accomplished by parsing the tracer files from uninterrupted runs of each firmware as well as the associated objdump files, which can be produced by most compilers. Tracer files can be produced by the testbench as the by-product of a test and often include pertinent information from the running of a given test. This information often includes the time of each instruction's execution, the instruction executed, the instruction's PC value, the instruction operands, and in some cases the operands values. A brief example of a tracer file can be seen in Figure 1 below. These parsed files are

used to build a list of all the instructions executed, the first instance of each instruction in terms of execution time, a valid timeframe in which the instruction could be interrupted, and which firmware the instruction occurred within. From this gathered data, the test can dynamically reconfigure interrupt timing within the testbench to target interrupt coverage of specific instructions on a case by case basis. From this evolution, this methodology takes individual tests and a moderate amount of reconfigurability and makes a test of tests to the extent that complete interrupt coverage can be gathered in a relatively short timeframe.

	Time	Cycles	PC	Instr	Mnemonic	Operands			
2	121.000ns	4	00008000	00004517	auipc	x10, 0x4000	x10=0000c000		
3	141.000ns	5	00008004	00050513	addi	x10, x10, 0	x10=0000c000	x10:0000c000	
4	161.000ns	6	00008008	30551073	csrrw	x0, x10, 0x305	x10:0000c000		
5	291.000ns	7	0000800c	00208137	lui	x2, 0x208000	x2=00208000		
6	431.000ns	8	00008010	f8010113	addi	x2, x2, -128	x2=00207f80	x2:00208000	
7	571.000ns	9	00008014	2400406f	jal	x0, 16960			
8	851.000ns	11	0000c254	000fd197	auipc	x3, 0xfd000	x3=00109254		
9	991.000ns	12	0000c258	f5018193	addi	x3, x3, -176	x3=001091a4	x3:00109254	
10	1011.000ns	13	0000c25c	000fc517	auipc	x10, 0xfc000	x10=0010825c		
11	1151.000ns	14	0000c260	75c50513	addi	x10, x10, 1884	x10=001089b8	x10:0010825c	
12	1291.000ns	15	0000c264	000ff617	auipc	x12, 0xff000	x12=0010b264		
13	1431.000ns	16	0000c268	fd860613	addi	x12, x12, -40	x12=0010b23c	x12:0010b264	
14	1571.000ns	17	0000c26c	40a60633	sub	x12, x12, x10	x12=00002884	x12:0010b23c	x10:001089b8
15	1711.000ns	18	0000c270	00000593	addi	x11, x0, 0	x11=00000000		
16	1851.000ns	19	0000c274	540040ef	jal	x1, 17728	x1=0000c278		
17	2131.000ns	21	000107b4	00f00313	addi	x6, x0, 15	x6=0000000f		
18	2271.000ns	22	000107b8	00050713	addi	x14, x10, 0	x14=001089b8	x10:001089b8	
19	2291.000ns	23	000107bc	02c37e63	bgeu	x6, x12, 60	x6:0000000f	x12:00002884	
20	2431.000ns	24	000107c0	00f77793	andi	x15, x14, 15	x15=00000008	x14:001089b8	
21	2571.000ns	25	000107c4	0a079063	bne	x15, x0, 160	x15:00000008		
22	2991.000ns	28	00010864	00279693	slli	x13, x15, 0x2	x13=00000020	x15:00000008	
23	3011.000ns	29	00010868	00000297	auipc	x5, 0x0	x5=00010868		
24	3031.000ns	30	0001086c	005686b3	add	x13, x13, x5	x13=00010888	x13:00000020	x5:00010868
25	3171.000ns	31	00010870	00008293	addi	x5, x1, 0	x5=0000c278	x1:0000c278	
26	3311.000ns	32	00010874	fa0680e7	jalr	x1, x13, -96	x1=00010878	x13:00010888	
27	3591.000ns	34	00010828	00b703a3	sb	x11, 7(x14)	x11:00000000	x14:001089b8	PA:001089bf
28	3731.000ns	35	0001082c	00b70323	sb	x11, 6(x14)	x11:00000000	x14:001089b8	PA:001089be
29	3751.000ns	36	00010830	00b702a3	sb	x11, 5(x14)	x11:00000000	x14:001089b8	PA:001089bd
30	3911.000ns	37	00010834	00b70223	sb	x11, 4(x14)	x11:00000000	x14:001089b8	PA:001089bc
31	4071.000ns	38	00010838	00b701a3	sb	x11, 3(x14)	x11:00000000	x14:001089b8	PA:001089bb
32	4231.000ns	39	0001083c	00b70123	sb	x11, 2(x14)	x11:00000000	x14:001089b8	PA:001089ba
33	4391.000ns	40	00010840	00b700a3	sb	x11, 1(x14)	x11:00000000	x14:001089b8	PA:001089b9
34	4551.000ns	41	00010844	00b70023	sb	x11, 0(x14)	x11:00000000	x14:001089b8	PA:001089b8
35	4711.000ns	42	00010848	00008067	jalr	x0, x1, 0	x1:00010878		
36	5031.000ns	44	00010878	00028093	addi	x1, x5, 0	x1=0000c278	x5:0000c278	
37	5171.000ns	45	0001087c	ff078793	addi	x15, x15, -16	x15=fffffff8	x15:00000008	
38	5191.000ns	46	00010880	40f70733	sub	x14, x14, x15	x14=001089c0	x14:001089b8	x15:fffffff8
39	5331.000ns	47	00010884	00f60633	add	x12, x12, x15	x12=0000287c	x12:00002884	x15:fffffff8
40	5471.000ns	48	00010888	f6c378e3	bgeu	x6, x12, -144	x6:0000000f	x12:0000287c	

Figure 1: Example Tracer File

A. Tracer Parsing

In order for the script to generate accurate timing parameters to interrupt each instruction, an uninterrupted copy of each firmware must be run, as any interrupted test would include ISR execution, and as such would have inaccurate timing measurements. The script accomplishes this by starting its test procedure by running an uninterrupted copy of every firmware on its own prior to generating timing data. The list of firmware used in this portion of the script can easily be hardcoded or created dynamically using a function that checks for file and directory names (such as glob in Perl). It is also important for the script to create a distinctive way of keeping track of the tracers from this portion of the script. Recommended methods include putting said files in a different directory than where tracers are normally generated, or appending some unique regular expression to the tracer file names such that the script can easily parse them from other tracer files. This portion of the script can take a somewhat significant amount of time depending on your testing environment and the number of firmware files utilized. In the case of EM's test environment, this step takes around 35-40 minutes.

From the information in the tracer files, a list of instructions can be generated. This is best accomplished by parsing each tracer file, line by line, and executing the following steps in order:

First, check if interrupts are enabled. In many architectures interrupts are disabled until enabled by firmware, and as such, attempts to interrupt instructions prior to interrupts being enabled would be a rather fruitless endeavour.

Checking if interrupts have been enabled can be done using hardcoded values, but this is inadvisable. It is recommended to have a specific expression that shows that interrupts have been enabled. In the case of the CV32E40P, checking that an access of the CSR mstatus resulted in bit 3 being set high sufficed, since that enables interrupts (see reference 1, section “Control and Status Registers”).

Once interrupts are enabled, parsing can begin in earnest, starting with checking if the given instruction mnemonic has already been scheduled to be interrupted. If this is the first interruptible instance of an instruction, it should be scheduled for interruption at a later time. This can be done by having a set of three arrays that contain (at matching element numbers) the instruction mnemonic, the interrupt time, and the firmware the instruction is in. Alternatively a pair of hashes can be used where the mnemonic is the key and the firmware name and interrupt time are the related values for those keys. An example of this code can be seen in Code Example 1.

```
1  #!/usr/bin/perl
2
3  foreach $file (@tests){
4      $filename = ("trace_core_" . $file . ".log");
5      open $fh ,'<', "$filename" or die "can't open trace file for test $file";
6      INT_CHECK: while (<$fh>){
7          chomp;
8          @fields = split(/\s+/);
9          if(($fields[4] eq "csrrs") || ($fields[4] eq "csrrw")) && ($fields[7] eq "0x300")) {
10             last INT_CHECK;
11         }
12     }
13     my $dummy = <$fh>;
14     my $dummy = <$fh>;
15     chomp $dummy;
16     @fields = split(/\s+/, $dummy);
17     $interrupttime = $fields[0];
18     $interrupttime =~ s/.000ns//;
19     while (<$fh>){
20         chomp;
21         @fields = split(/\s+/);
22         $instruction = $instr_tracker{$file}{$fields[2]};
23         unless ((exists($exectime{$instruction})) || (exists($execfw{$instruction}))) {
24             $execfw{$instruction} = $file;
25             $exectime{$instruction} = ($interrupttime - 6);
26         }
27         $interrupttime = $fields[0];
28         $interrupttime =~ s/.000ns//;
29     }
30     close $fh;
31 }
```

Code Example 1: Tracer Parsing Process

In Code Example 1, each tracer file is opened, and then checked to see if CSR mstatus is modified (lines 3-12). Until mstatus is modified, interrupts are by default disabled and as such, instructions until then cannot be interrupted. Once interrupts are enabled, the following two instructions are ignored (lines 13, 14). At this point the tracer file is parsed to determine interrupt timing and the instruction mnemonic of each instruction (lines 19-30). As long as a copy of the instruction is not already scheduled to be interrupted the script will note a valid time to interrupt the instruction, as well as its mnemonic (lines 23-26). This portion of the methodology can be observed as a flowchart in Figure 2.

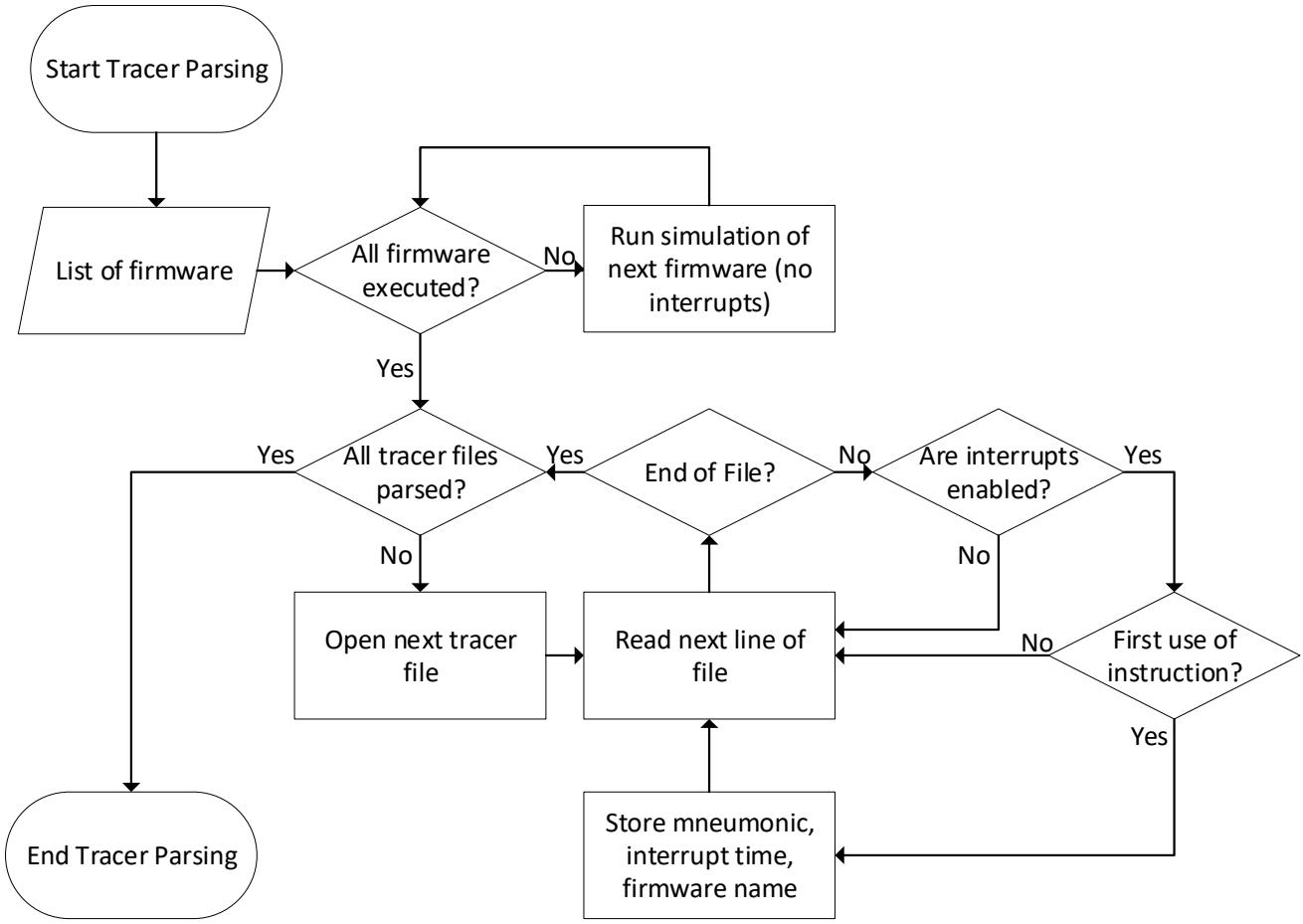

Figure 2: Tracer Parsing Flowchart

In Code Example 1, it should be noted that when calculating interrupt timing the previous instruction's execution time is used, since the interrupt is triggered during the preceding instruction's execution phase, and the CV32E40P allows for use of a clock divider and multicycle instructions. If a fixed offset were utilized from the execution time of the instruction being interrupted, the value could be incorrect for the given clock rate and cause the incorrect instruction to be executed. It is also worth noting that constrained randomization can be utilized by replacing ($interrupttime - 6) on line 25 with the value of $interrupttime subtracted by a randomized value within a valid range. This will still allow for the test to interrupt the correct instruction, and this randomization could be configured to account for instructions that take multiple cycles to execute (such as division instructions).

B. Interrupt Testing

Now that interrupt timing has been determined, that information can be used to execute each interrupt test while displaying the pertinent information to the user, as can be seen in Code Example 2.

```
3  foreach $instruction (sort(keys %exectime)){
4      open $fh, '<', "verilog/testbench/test_pkg.sv" or die "cannot modify test_pkg.sv";
5      open OUT_FILE, '>', "test_pkg.sv" or die "can't open temporary test_pkg file";
6      while(<$fh>){
7          if($modify){
8              $modify = 0;
9              print OUT_FILE "      get_delay = $exectime{$instruction};\n";
10         }
11         elsif($_ eq "      //PERL SCRIPT HOOK\n"){
12             print OUT_FILE $_;
13             $modify = 1;
14         }
15         else {
16             print OUT_FILE $_;
17         }
18     }
19     close $fh;
20     close OUT_FILE;
21     $fw = $execfw{$instruction};
22     `rm verilog/testbench/test_pkg.sv`;
23     `mv test_pkg.sv verilog/testbench/test_pkg.sv`;
24     #run test and check for no errors here
25     $next = "false";
26     open my $fh, '<', "application/log/" . $execfw{$instruction} . ".log";
27     PARSE: while(<$fh>){
28         chomp;
29         @fields = split(/\s+/);
30         foreach $field (@fields){
31             if($next eq "true"){
32                 $seed = $field;
33                 $next = "false";
34             last PARSE;
35             }
36             if($field eq "-svseed"){
37                 $next = "true";
38             }
39         }
40     }
41     close $fh;
42     #move passing and failing test logs to the relevant directories
43 }
```

Code Example 2: Interrupt Testing Execution and Log Parsing

For each instruction's interrupt scheduled by Code Example 1, package test_pkg.sv is modified to hardcode the interrupt timing value into the test package (lines 4-23). Once the interrupt timing is set the test is run, which results in a log file of the simulation that can be parsed to determine if the test passed or failed. In the case of the logfiles being stored by EM, we then parse for and append the seed value to the log file name to ensure no log files are overwritten during testing (lines 26-41).

Running all the interrupt tests takes a significant amount of time, and in the case of the CV32E40P core with the XPULP extension enabled, testing takes approximately 5.5-6 hours on a single core, plenty of time to complete in an overnight regression to obtain 100% interrupt coverage.

It is worth pointing out that this script could be simplified by utilizing a single firmware file containing one copy of every instruction (after interrupts have been enabled) and randomly interrupting these instructions. This would actually be a more ideal solution than the script described in this paper, as this script is a rather brute-force solution to a convoluted problem. Unfortunately, the Instruction Set Simulator (ISS) being utilized by EM Microelectronic and OpenHW for the purpose of verifying the CV32E40P currently does not model the instructions in the XPULP extension. As such, multiple custom, self-checking directed tests are utilized to test XPULP instructions. Writing a directed test that exercises all nearly 400 instructions in the CV32E40P in a single file would likely require more work

than the development of the script described, and randomized testing would take longer to gather complete coverage than the method described in this paper.

C. *Processing and Parsing Results*

Unfortunately the testbench is presently incapable of adding the mnemonics of interrupted instructions to the log files generated during testing. Fortunately, the testbench can put the 32-bit value of the instruction into the log file, and this can be used to confirm the mnemonics of the interrupted instructions. This is done by parsing out the interrupted 32-bit value from the log, and making it a .word(0xXXXXXXXX) line in an assembly file, and then compiling the file containing all the interrupted instructions, then simply enumerating and outputting the results to a CSV file for analysis. This can be seen in part in Code Example 3.

```
1  #!/usr/bin/perl
2
3  #detect all valid log files using glob
4  open OUT_FILE, '>', "interrupted_instructions.S" or die "Can't open file interrupted_instructions.S\n";
5  foreach $file (@files) {
6      chomp $file;
7      open my $fh, '<', "$file" or die "Can't open log file $file\n";
8      $file_name = $file;
9      $file_name=~ (s/_(\d+).log//);
10     @fields= split(/\//, $file_name);
11     $file = $fields[(scalar(@fields)-1)];
12     while(<$fh>) {
13         #parse file to find hex value of interrupted instruction
14     }
15     close $fh;
16 }
17 close OUT_FILE;
18
19 #compile interrupted_instructions.S into a .objdump file
20
21 open my $fh, '<', "interrupted_instructions.objdump" or die "Can't open file interrupted_instructions.objdump\n";
22 while (<$fh>) {
23     chomp;
24     @fields = split(/\s+/);
25     if(length($fields[2])==8)
26         $interrupt_count{$fields[3]}++;
27     }
28 }
29 close $fh;
```

Code Example 3: Result Parsing and Reporting

In Code Example 3, the set of log files is determined, and then each log file is parsed to find the hex value of the interrupted instruction from that test (lines 3-17). Then, the list of instructions is compiled into a .objdump file (line 19) which can then be parsed to create a hash of the results (lines 21-29), which can then be output to a CSV file for ease of reading. Sections B *Interrupt Testing* and *C Processing and Parsing Results* of the methodology can also be observed as a flowchart in Figure 3.

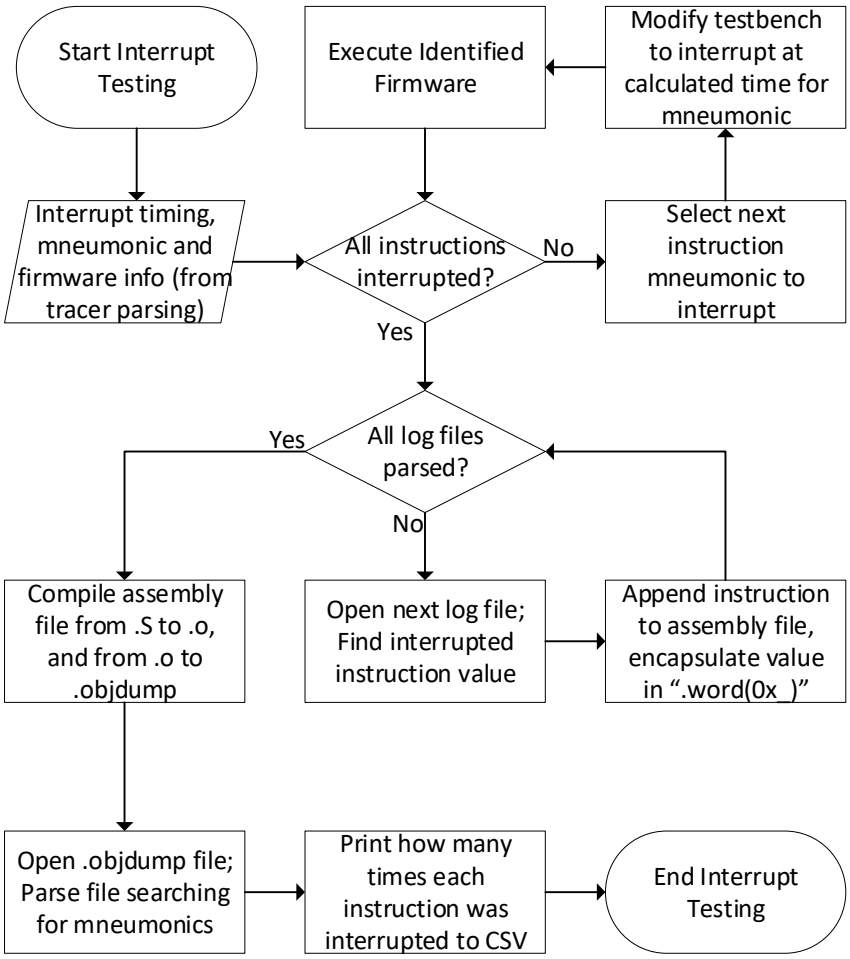

Figure 3: Interrupt Testing and Result Parsing Flowchart

IV. Conclusion

To summarize this methodology, a script reads (or generates) a list of valid firmware, and runs each firmware once without any interrupts being triggered. Then the resultant tracer files are parsed to find the first interruptible instance of every instruction, noting the firmware said instruction is in, as well as what time to interrupt said instruction. For each instruction a file is modified in the testbench to customize interrupt timing, and an interrupt test is run. Then the resultant logs are parsed and the hex values of each interrupted instruction are dumped into an assembly file. This assembly file is then compiled into an objdump file, which is finally parsed to determine which instructions were successfully interrupted and the results are output to a CSV file for ease of readability.

This methodology can be quite powerful, but has its pitfalls as well. For example, the implementation of this methodology implemented at EM Microelectronic was done in Perl, which while powerful, also can be quite difficult to use when it comes to regexp parsing, and this leads to more bugs needing to be fixed. In addition, our implementation encountered issues with differences between the compiler and tracer. This results in some extra tests being run (and as a result, some instructions being covered multiple times), and some mnemonics need to be fixed by the script during execution. Another issue with this methodology is that some firmware might disable interrupts mid-execution, and the provided methodology and code examples do not handle this, as they only parse for interrupts being enabled in the firmware. This methodology and script could be improved in the future to target the OpenHW testbench by changing how tests are run and the file structure utilized by the script. The amount of time it would take to retarget the existing script to the OpenHW testbench would be less than the total time to develop and debug this script, about 5 days.

The original method EM Microelectronic used for interrupt coverage took a little under 1 month in total, with 3 weeks of that time being spent running randomized testing in the background to generate most of the coverage, as well as another week or so of time spent writing and running directed tests to fill coverage holes. This method was inefficient in terms of time utilization and resulted in ~17000 tests being run, as many commonly utilized instructions were tested dozens or in select cases, hundreds of times. In contrast, utilizing the methodology described in this paper resulted in interrupt coverage closure in around 6 hours, interrupting each instruction only a single time. This enables interrupt coverage to be closed in every night's regression.

References

[1] OpenHW CV32E40P User Manual:
https://core-v-docs-verif-strat.readthedocs.io/projects/cv32e40p_um/en/latest/