Arm Instruction Emulator v1.2.1 User Guide

Development Solutions Group

Document number: 101212 Version: 1.1

Date of Issue: 30/03/2018

Abstract

This document contains usage instructions for users of Arm Instruction Emulator version 1.2.1. If you're using Arm Instruction Emulator 18.0 or later, please refer to the latest content for Arm Instruction Emulator on the Arm Developer website.

Contents

1 About this document

This document is an archive of Arm Instruction Emulator usage instructions for versions prior to v18.0.

There are no anticipated changes to this document.

If you are using version 18.0 of Arm Instruction Emulator, refer to the documentation on the Arm Developer website: https://developer.arm.com/products/software-development-tools/hpc/arm-instruction-emulator

2 Download and install Arm Instruction Emulator

Arm Instruction Emulator runs on AArch64 platforms and emulates SVE instructions. The emulator lets you develop and compile SVE code with Arm Compiler for HPC, and run the SVE binary without needing access to SVE-enabled hardware. Use the following steps to download and install Arm Instruction Emulator.

1. Login at https://silver.arm.com/browse using your Arm developer account. You can create one, if you do not have one.

   The **Downloads** home page appears. If not, click on the **Downloads** option in the **Arm Self Service** navigation pane on the left.

2. Expand the **Development Tools** option in the list of products available to you. Under **Arm HPC Tools**, select **Arm Instruction Emulator**.

3. In the **Public Downloads** section, find the appropriate package for your Linux host platform and select **Download Now**.

4. Extract the downloaded package using the following command:
   `tar -xvf <package_name>`

   Replace `<package_name>` with the full name of the downloaded package.

5. Change to the package directory to see the extracted files:
   `cd <package_name>`

6. Run the installation script as a privileged user:
   `% su root Password: ******** % ./<package_name>.sh`

   Packages are unpacked to `<install_dir>/opt/arm/<package_name>`. Optionally, use the `--install-to` option to specify the install location:
   `% ./<package_name>.sh --install-to <install_dir>`

   Note: This results in a user-only installation along with the relevant module files. The included *uninstall.sh* helper script does not uninstall this type of installation.

7. The installer displays the EULA and prompts you to agree to the terms. To agree, type `yes` at the prompt.

For more information about the release contents, see the release notes, located in the `/opt/arm/<package_name>/share` directory.

8. Configure your Linux environment. If Environment Modules are not already installed on your machine, refer to Environment configuration topic on the Arm Developer website for instructions on how to install them.
    a. To see which Environment Modules are available, run:
    ```
    % module avail
    ```

    Note: You might need to configure the `MODULEPATH` environment variable to include the installation directory:
    ```
    % export MODULEPATH=$MODULEPATH:/opt/arm/modulefiles/
    ```

    Tip: Add the `module load` command to your `.profile` to run it automatically every time you log in.

    b. Load the appropriate Arm Instruction Emulator module for the processors in your system, and for the compiler you are using:
    ```
    % module load
    <architecture>/<linux_variant>/<linux_version>/suites/arm-
    instruction-emulator/<version>
    ```

    For example:
    ```
    % module load Generic-AArch64/SUSE/12/suites/arm-instruction-
    emulator/1.2.1
    ```

    c. Check your environment by examining the `PATH` variable. It should contain the appropriate Arm Instruction Emulator bin directory from `/opt/arm`:
    ```
    % echo $PATH /opt/arm/arm-instruction-emulator-1.2.1_Generic-
    AArch64_SUSE-12_aarch64-linux/bin:...
    ```

Note: For information about environment variables used by the suite of HPC tools provided by Arm, see our Environment variables reference topic.

3 Getting started with Arm Instruction Emulator

This section uses a series of examples to demonstrate how to compile Scalable Vector Extension (SVE) code, run the resulting executable and gathering profiling data with Arm Instruction Emulator.

This section also uses the Arm C/C++ Compiler from the Arm HPC tools suite. Refer to Download and install Arm Compiler for HPC and Arm Performance Libraries and Environment configuration for instructions on installing and configuring your Linux environment for Arm Compiler for HPC, respectively. **Note:** Ensure that you load the necessary modules for Arm C/C++ Compiler before beginning these exercises.

3.1 Basic example: Compiling and running a Hello World program

This example shows how write a *Hello World* program. Compile it using Arm C/C++ Compiler, and run it using Arm Instruction Emulator.

1. Create a simple *Hello World* C program and save it as a file. In this example, it is saved in a file named *hello.c*.
```
/* Hello World */
#include <stdio.h>
int main()
{
    printf("Hello World\n");
   return 0;
}
```

2. To generate an executable binary, compile your program with Arm C/C++ compiler:
```
armclang -O3 -march=armv8-a+sve -o hello hello.c
```

   The `-O3` flag ensures that the highest optimization level with auto-vectorization is enabled.
   The `-march=armv8-a+sve` flag targets hardware with Armv8-A architecture.

   Note: In this example, no SVE code is used. However, it is good practice to enable the highest level of auto-vectorization and target an SVE-enabled architecture when compiling any code to be run using Arm Instruction Emulator.

3. Run the generated binary hello using Arm Instruction Emulator:
```
armie -msve-vector-bits=256 ./hello
Hello World
```

   For this basic Hello World example, Arm Instruction Emulator runs the code on an emulated SVE-enabled architecture without utilizing SVE instructions.

   The next example shows a more complex program that uses Arm Instruction Emulator to its full potential and emulates SVE instructions.

3.2 Advanced example: Compiling and running a program with SVE code

This example demonstrates how to compile and vectorize SVE code using Arm C/C++ to target the SVE-enabled Armv8-A architecture, and how to emulate running the SVE code using Arm Instruction Emulator.

1. Create a new file called *example.c*.
2. Open the file, insert the following C code, then save and close the file.

   Note: This example C program subtracts corresponding elements in two arrays, and writes the result to a third array. The three arrays are declared using the restrict keyword, indicating to the compiler that they do not overlap in memory.

```
// example.c
#include <stdio.h>
#include <stdlib.h>
#define ARRAYSIZE 1024
int a[ARRAYSIZE];
int b[ARRAYSIZE];
int c[ARRAYSIZE];
void subtract_arrays(int *restrict a, int *restrict b, int *restrict c)
{
    for (int i = 0; i < ARRAYSIZE; i++)
    {
        a[i] = b[i] - c[i];
    }
}
int main() {
    for (int i = 0; i < ARRAYSIZE; i++)
    {
      // Generate a random number between 200 and 300
      b[i] = (rand() % 100) + 200;
      // Generate a random number between 0 and 100
      c[i] = rand() % 100;
    }
    subtract_arrays(a, b, c);
    printf("i \ta[i] \tb[i] \tc[i] \n");
    printf("============================\n");
    for (int i = 0; i < ARRAYSIZE; i++)
    {
        printf("%d \t%d \t%d \t%d\n", i, a[i], b[i], c[i]);
    }
}
```

3. Compile the program as follows:

```
armclang -O3 -march=armv8-a+sve -o example example.c
```

4. Run the binary using Arm Instruction Emulator:

```
armie -msve-vector-bits=256 ./example
```

   This returns the following:

```
i       a[i]    b[i]    c[i]
============================
0       197     283     86
1       262     277     15
```

```
2       258     293     35
...
1021    165     234     69
1022    232     295     63
1023    204     235     31
```

The SVE architecture extension specifies an implementation-defined vector length. The -msve-vector-bits option allows you to specify the vector length used by Arm Instruction Emulator. The vector length is a multiple of 128 bits, with a maximum of 2048 bits.

Use the `-mlist-vector-lengths` option to list all valid vector lengths:

```
armie -mlist-vector-lengths
```

This returns the following:

```
128 256 384 512 640 768 896 1024 1152 1280 1408 1536 1664 1792 1920
2048
```

3.3 Advanced example: Gathering profiling data with Arm Instruction Emulator

Arm Instruction Emulator helps to understand which parts of your code affect program performance. It samples which instruction is being executed at a user-specified frequency while the program is running.
This feature is available in Arm Instruction Emulator version 1.1 and later.

1. The following example uses the LULESH 2.0 simulation. Download LULESH 2.0 as follows:
   a. Download the latest release version of LULESH 2.0 CPU Models from https://codesign.llnl.gov/lulesh.php:
      ```
      wget https://codesign.llnl.gov/lulesh/lulesh2.0.3.tgz
      ```

      Note: At the time of writing, the latest version is 2.0.3
   b. Uncompress and extract the downloaded package:
      ```
      tar -xvf lulesh2.0.3.tgz
      ```

2. By default, the LULESH build configuration compiles using g++. Change this to use Arm C/C++ Compiler for HPC and generate insights by making the following changes in the Makefile:
   a. Change `SERCXX = g++ -DUSE_MPI=0`
      To `SERCXX = armclang++ -DUSE_MPI=0`
   b. Change `CXX = $(MPICXX)`
      To `CXX = $(SERCXX)`
   c. Change `CXXFLAGS = -g -O3 -fopenmp -I. -Wall`
      To `CXXFLAGS = -g -O3 -fopenmp -I. -Wall -march=armv8-a+sve -insight`

3. Build the LULESH application:
   ```
   Make
   ```

   The build produces an executable binary, lulesh2.0 in the current directory.

4. Run the LULESH 2.0 binary with armie using the --profile-period or –p option to specify the sample period in microseconds:

```
armie -msve-vector-bits=512 -p 100 -- ./lulesh2.0 -s 9
```

This runs LULESH 2.0, sampling the program counter every 100 microseconds. When the program terminates, a samples file is created in the current directory with the name format *<binary name>_<PID>.samples*, for example: *lulesh2.0_3076.samples*. This file contains a list of the samples taken. The samples are the instruction address followed by the number of executions, for example:

```
head lulesh2.0_3076.samples
```

This returns:

```
0x402578 62
0x402580 51
0x406e60 22
0x406e5c 20
0x406e58 14
0x402570 14
0x406e64 12
0x406004 10
0x406214 10
0x406630 9
```

5. This format enables you to use GNU Linux tools like *addr2line* which map instruction addresses to source, to understand program behavior.

   Using the addr2func helper script, which comes as part of the release, you can use the samples file to identify which functions were the hottest in the LULESH 2.0 run:

   ```
   addr2func lulesh2.0 lulesh2.0_3076.samples
   ```

   This returns the following:

   ```
   CalcElemNodeNormals: 143
   SumElemFaceNormal: 2
   ApplyMaterialPropertiesForElems: 12
   CalcEnergyForElems: 115
   CalcPressureForElems: 311
   CalcMonotonicQGradientsForElems: 3
   CalcElemCharacteristicLength: 1
   SQRT: 1
   CalcHourglassControlForElems: 1
   CalcForceForNodes: 1
   IntegrateStressForElems: 1
   CollectDomainNodesToElemNodes: 3
   Domain::xd: 1
   CalcElemFBHourglassForce: 1
   UpdateVolumesForElems: 11
   InitStressTermsForElems: 8
   CalcFBHourglassForceForElems: 3
   CalcMonotonicQRegionForElems: 2
   ApplyAccelerationBoundaryConditionsForNodes: 2
   FABS: 14
   EvalEOSForElems: 103
   ```

   The hottest function in this list is `CalcPressureForElems`; which was executed 311 times.

Note: The accuracy of the sampling profiler and accurate performance measurement of programs, depends on their run time. The longer the run, the more accurate the numbers describing hot code.

3.4 Troubleshooting

In the event of a program crash, the operating system kernel creates a core dump file. The location and name of this core dump file depends on your system's core dump configuration.

If your configuration specifies that core dump filenames include the name of the crashed binary, this is the name of the executable being emulated rather than the Arm Instruction Emulator binary name armie.
Send core dump files to Arm support along with the output of armie `--version`.

Note:

- If you have confidentiality concerns regarding sensitive data in the core dump file, do not send the core dump to Arm. However, Arm might not be able to investigate your issue.
- If you encounter problems running a binary with Arm Instruction Emulator, use the `--debug` option to run internal checks (assert calls) during execution. If Arm Instruction Emulator finds an internal inconsistency, it stops executing, and outputs a message to `stderr`. Send this file to Arm support. For example, use the following command:

  ```
  armie -msve-vector-bits=256 --debug ./example
  ```

  This produces the following output:

  ```
  example: ./src/sve_decode.h:93:
  aarch64_i_rsp_reg::aarch64_i_rsp_reg(unsigned int,
           aarch64_i_rsp_reg::element_type): Assertion `reg_id < 32'
  failed.
  ```

  Alternatively, print output messages to an output file and include `-o` or `--output` in the command line input.

  The `--debug` option also helps you identify the instructions that were executed by the emulator. The first column is the address of the instruction, the second is the instruction encoding, and the third is the number of times the instruction was executed. For example:

  ```
  0x400684: 0x043f57df 1
  0x4006a0: 0x04bf5028 1
  0x4006c8: 0x2538c000 1
  0x4006cc: 0x25291fe0 1
  0x4006d4: 0xe4084140 13
  0x4006d8: 0x04285028 13
  0x4006dc: 0x25291d00 13
  0x4006ec: 0x25a91fe0 1
  0x4006f4: 0xe58103a0 1
  0x4006f8: 0x04a14500 13
  0x4006fc: 0xe5484140 13
  0x400700: 0x04b0e3e8 13
  0x400704: 0x25a91d00 13
  0x400740: 0x858103a1 1
  0x40074c: 0x25b8c020 1
  0x400758: 0x2598e3e0 1
  ```

```
0x40075c: 0xa5484521 13
0x400760: 0x04938001 13
0x400764: 0xe5484541 13
0x400768: 0x04b0e3e8 13
0x40076c: 0x25ab1d01 13
0x4007bc: 0x043f505f 1
```

If you need further assistance, contact Arm Support.

4 Analyzing SVE code with Arm Instruction Emulator and Arm Code Advisor

The Arm Instruction Emulator is an emulator that runs on AArch64 platforms and emulates Scalable Vector Extension (SVE) instructions. Arm Code Advisor can integrate with Arm Instruction Emulator, allowing you to develop, analyze, and optimize code even if you don't have access to hardware that implements SVE for the Armv8-A architecture.
This section uses LULESH 2.0 to demonstrate how to compile SVE code, run it using Arm Instruction Emulator, and collect analysis data from the emulator using Arm Code Advisor.

1. Ensure that the Arm tools are available by loading the environment modules for Arm Compiler for HPC and Arm Code Advisor:
   ```
   module load Generic-AArch64/SUSE/12/suites/arm-compiler-for-hpc/1.3
   module load Generic-AArch64/SUSE/12/suites/arm-code-advisor-beta/1.0
   ```

   Note: See Installing Arm Compiler for HPC and Installing Arm Code Advisor for more information about installing the Arm tools and configuring environment modules.

2. Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) is a hydrodynamics modeling application. This section uses LULESH 2.0 as an example application to demonstrate Arm Code Advisor. Download the latest release version of LULESH 2.0 CPU Models from https://codesign.llnl.gov/lulesh.php:

   ```
   wget https://codesign.llnl.gov/lulesh/lulesh2.0.3.tgz
   ```

   Note: At the time of writing, the latest version is 2.0.3

   Uncompress and extract the downloaded package:
   ```
   tar -xvf lulesh2.0.3.tgz
   ```

3. By default, the LULESH build configuration compiles using g++. We'll change this to use Arm C/C++ Compiler for HPC and generate insights by making the following changes in the `Makefile`.
   a. Change `SERCXX = g++ -DUSE_MPI=0`
      To `SERCXX = armclang++ -DUSE_MPI=0`
   b. Change `CXX = $(MPICXX)`
      To `CXX = $(SERCXX)`

c. Change `CXXFLAGS = -g -O3 -fopenmp -I. -Wall`
To `CXXFLAGS = -g -O3 -fopenmp -I. -Wall` **`-march=armv8-a+sve -insight`**

4. To build the LULESH application:
`make`

The build produces an executable binary, *lulesh2.0* in the current directory.

5. The application is built for the Armv8-A architecture with Scalable Vector Extension. As it makes use of SVE instructions, running this binary on a platform that does not implement the SVE extension causes an illegal instruction error. Because Arm Instruction Emulator can execute AArch64 and SVE instructions, we can instruct Arm Code Advisor to use the emulator to collect profiling data rather than running the executable directly:
`armcadvisor collect` **`--armie`** `-- ./lulesh2.0 -s 15`

Arm Code Advisor runs the specified executable using Arm Instruction Emulator, and collects performance profiling data about the code.

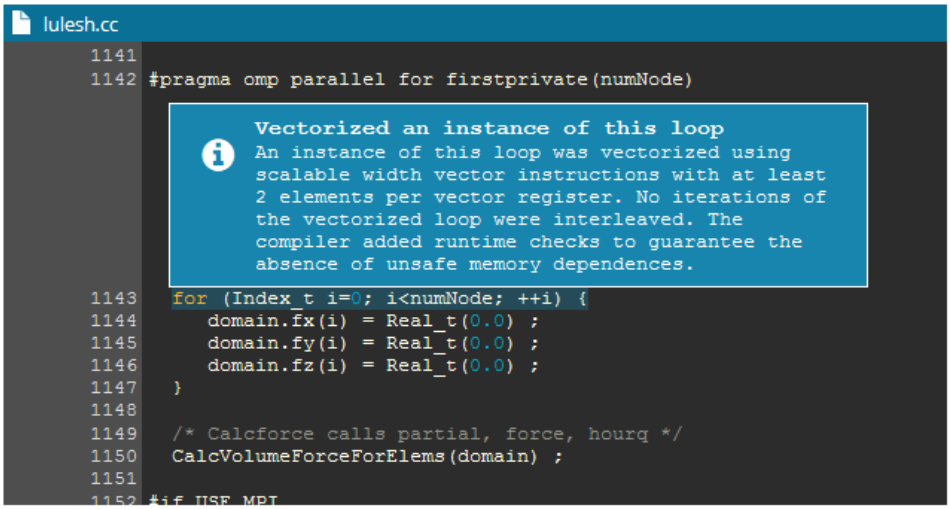
The SVE architecture extension specifies an implementation-defined vector length. The `-msve-vector-bits` option lets you specify the vector length used by Arm Instruction Emulator:
`armcadvisor collect --armie` **`-msve-vector-bits`**`=256 -- ./lulesh2.0 -s 15`

6. Analysis brings together the insights generated by the compiler and the performance data collected by Arm Code Advisor to produce prioritized advice, ranking insights in order of importance.To perform analysis:
`armcadvisor analyze`

This generates the analysis file, *armcadvisor.advice*.

7. Use the `armcadvisor web` command to display the Arm Code Advisor web interface:
`armcadvisor web -ne`

Open your browser to one of:
`http://server.arm.com:8080`
`http://127.0.0.1:8080`



Advice in the source code shows where loops have been vectorized with SVE instructions to optimize performance.

For more information about Arm Code Advisor, see Getting started with Arm Code Advisor on the Arm Developer website.

5 Memory tracing for SVE code with Arm Instruction Emulator

To run a compiled binary with a memory trace of SVE instructions, complete the following steps.

Note: This is an experimental feature that is under development, and therefore has undergone limited testing. Support is limited for experimental features. Such features can also be removed, or change significantly in future releases.

1. To start and stop the trace around regions of interest, define and use the following intrinsic functions in source code:
```
#define __START_TRACE() {asm volatile (".inst 0x2520e020");}
#define __STOP_TRACE() {asm volatile (".inst 0x2520e040");}
```

   Note: Ensure that intrinsic functions are not used inside a loop to avoid vectorization being hindered.

2. To enable memory tracing, run `armie` using the `-msve-memtrace=<text | binary>` command. This prints the trace in either textual or binary format to a file called *armie-mem-trace.log*.

   The layout of the binary memory trace uses the format:
```
// Memory tracing interface
typedef enum {
      bundle_none    = 0,  // no bundle (contiguous access)
      bundle_first   = 1,  // first element in gather/scatter bundle
      bundle_elt     = 2,  // just another element in bundle
      bundle_last    = 4,  // last element in gather/scatter bundle
    } mem_ref_bundle_t;

    typedef struct {
      int tid;           // thread id
      unsigned bundle;   // support bundling of multiple mem_refs
                         // for gather/scatter/strided accesses
      unsigned isWrite;  // true if write, false if load
      size_t size;       // num bytes loaded
      void* addr;        // load/store  addr
      void* pc;          // instruction addr
  } mem_ref_t;
```

The textual memory trace uses comma-separated value format with the following fields, as described above:
`tid`, `bundle`, `isWrite`, `size`, `addr`, `pc`

Example program:

```
#define N 42
int a[N], b[N], c[N];

#define __START_TRACE() {asm volatile (".inst 0x2520e020");}
#define __STOP_TRACE() {asm volatile (".inst 0x2520e040");}

int main(void) {
  for(int i=0; i<N; ++i)
    c[i] = i;

  __START_TRACE();
  for(int i=0; i<N; ++i)
    a[i] = b[i] + b[c[i]];
  __STOP_TRACE();
}
```

3. To build the example program above, save it as memtrace.c, and use:
   `armclang -O3 -march=armv8-a+sve -o memtrace memtrace.c`

4. To run the `memtrace` binary with Arm Instruction Emulator and using textual memory trace option, use:
   `armie -msve-vector-bits=512 -msve-memtrace=text – ./memtrace`

   For 512-bit vectors, this gives a trace like the following, in a file called *armie-mem-trace.log*:

```
479789056, 0, 0, 64, 0x4206c4, 0x400638
479789056, 0, 0, 64, 0x420034, 0x40063c
479789056, 3, 0, 4, 0x420034, 0x400640
479789056, 2, 0, 4, 0x420038, 0x400640
479789056, 2, 0, 4, 0x42003c, 0x400640
479789056, 2, 0, 4, 0x420040, 0x400640
479789056, 2, 0, 4, 0x420044, 0x400640
479789056, 2, 0, 4, 0x420048, 0x400640
479789056, 2, 0, 4, 0x42004c, 0x400640
479789056, 2, 0, 4, 0x420050, 0x400640
479789056, 2, 0, 4, 0x420054, 0x400640
479789056, 2, 0, 4, 0x420058, 0x400640
479789056, 2, 0, 4, 0x42005c, 0x400640
479789056, 2, 0, 4, 0x420060, 0x400640
479789056, 2, 0, 4, 0x420064, 0x400640
479789056, 2, 0, 4, 0x420068, 0x400640
479789056, 2, 0, 4, 0x42006c, 0x400640
479789056, 6, 0, 4, 0x420070, 0x400640
479789056, 0, 1, 64, 0x420d54, 0x400648
479789056, 0, 0, 64, 0x420704, 0x400638
```

5. To print the instruction addresses in a readable format, use the addr2func script on text based *armie-mem-trace.log* trace output:
```
addr2func memtrace armie-mem-trace.log
```

   This returns the following:
```
479789056, 0, 0, 64, 0x4206c4, main @ main.c:13
479789056, 0, 0, 64, 0x420034, main @ main.c:13
479789056, 3, 0, 4, 0x420034, main @ main.c:13
479789056, 2, 0, 4, 0x420038, main @ main.c:13
...
```

Note: If the program is compiled with debug information, line numbers are included.

6 armie command reference

The `armie` command runs a compiled binary using Arm Instruction Emulator. Arm Instruction Emulator is an emulator that can execute AArch64 and Scalable Vector Extension (SVE) instructions on Armv8-A hardware.

To execute and provide operational instructions to the Arm Instruction Emulator, use:
```
armie  [options] -- <command to execute>
```

Option	*Description*
`-msve-vector-bits=<uint>`	Vector length to use. Must be a multiple of 128 bits up to 2048 bits.
`-mlist-vector-lengths`	List all valid vector lengths.
`-msve-memtrace=<string>`	Set the output format for memory access tracing for SVE instructions. Must be one of "none" (default), "text" or "binary". **Note:** This is an experimental feature under development, with limited testing and support. Experimental features may be removed or changed significantly in future versions.
`d, --debug`	Enable assertion checks in the emulator to help isolate and diagnose bugs. Cannot be used with -p, --profile-period.
`-s, --stats`	Enable statistics about the emulated SVE instructions.
`-o, --output <file name>`	Redirect all messages generated by armie to a file.
`-p, --profile-period <uint>`	Enable the performance profiler and set the sampling period (in microseconds). Cannot be used with -d, --debug.
`-h, --help`	Print this help message.
`-V, --version`	Print the version.

Examples using the armie command:

- To list all valid vector lengths:
  ```
  armie -mlist-vector-lengths
  ```

  returns
  ```
  128 256 384 512 640 768 896 1024 1152 1280 1408 1536 1664 1792 1920
  2048
  ```
- To run the compiled binary 'sve_program' with 256-bit vectors:
  ```
  armie -msve-vector-bits=256 -- ./sve_program
  ```

- To run the compiled binary 'sve_program' with 256-bit vectors, specifying a sample period of 100 microseconds:
  ```
  armie -msve-vector-bits=256 --profile-period 100 -- ./sve_program
  ```

  When the program terminates, a samples file is created in the current directory containing a list of the samples taken. The samples are shown as the instruction address followed by the number of executions.

  To map sample addresses to functions, the helper script *addr2func* can be used. The helper script *addr2func* sums the samples and groups them by function name.

- To run the compiled binary "sve_program" with 2048-bit vectors, assertion checks and passing `optionA` as an argument, use:
  ```
  armie -msve-vector-bits=2048 --debug -- ./sve_program -optionA
  ```