Arm China Zhouyi Compass Debugger

Version: 1.10

User Guide

Confidential

arm CHINA

Arm China Zhouyi Compass Debugger

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0100-00	28 December 2020	Confidential	Release for 1.0
0101-00	15 March 2021	Confidential	Update for 1.1
0102-00	18 June 2021	Confidential	Update for 1.2
0103-00	30 September 2021	Confidential	Update for 1.3
0104-00	31 December 2021	Confidential	Update for 1.4
0105-00	31 March 2022	Confidential	Update for 1.5
0106-00	30 June 2022	Confidential	Update for 1.6
0107-00	30 September 2022	Confidential	Update for 1.7
0108-00	31 March 2023	Confidential	Update for 1.8
0109-00	30 June 2023	Confidential	Update for 1.9
0110-00	30 September 2023	Confidential	Update for 1.10

Confidential Proprietary Notice

Confidentiality Status

Product Status

The information in this document is Final, that is for a developed product.

Web Address

https://www.armchina.com

Contents
Arm China Zhouyi Compass Debugger User Guide

Preface

This preface introduces the *Arm China Zhouyi Compass Debugger User Guide*.

It contains the following sections:

- *About this book* on page 7.
- *Feedback* on page 9.

About this book

This book is intended to provide a single guide for developers to debug programs for the Zhouyi *Neural Processing Unit* (NPU) processor. The book includes complete information about debugging on both the simulator platform and hardware platform.

This is not an introductory level book. It assumes some knowledge of the C programming language and microprocessors, but not of any Arm China-specific background. We cannot hope to cover every topic in detail. In some chapters, we suggest additional reading (referring either to books or websites) that can give a deeper level of background to the topic in hand, but in this book we focus on the Arm China-specific detail. We do not assume the use of any particular toolchain. We will mention both GNU and Arm China tools in the course of the book. We hope that the book is suitable for programmers who have a desktop PC or x86 background and are taking their first steps into the Arm China processor based world.

The book is meant to complement rather than replace other Arm China documentation available for Zhouyi series processors, such as the Arm China *Technical Reference Manual*s (TRMs) for the processors themselves, and documentation for integration and implementation.

Intended audience

This guide is for developers and programmers debugging the Zhouyi NPU processor.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
This chapter provides an overview of the Zhouyi Compass debugger.

Chapter 2 Getting started
This chapter describes the environment, configuration settings, and how to get started with the debugger.

Chapter 3 NPU program execution
This chapter describes how to control the NPU execution.

Chapter 4 Breakpoints and watchpoints
This chapter describes the breakpoints and watchpoints.

Chapter 5 Inspecting NPU state
This chapter describes operations that are used to check NPU state.

Glossary

The Arm® Glossary is a list of terms used in Arm China documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm China meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic

Introduces special terminology, denotes cross-references, and citations.

bold

Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`

Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace` (underlined: mono)

Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`

Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Additional reading

This book contains information that is specific to this product. See the following documents for other relevant information.

Arm China publications

- *Arm China Zhouyi Z2 AIPU Technical Reference Manual.*
- *Arm China Zhouyi Z3 AIPU Technical Reference Manual.*
- *Arm China Zhouyi NPU X1 Technical Reference Manual.*
- *Arm China Zhouyi Compass Assembly Programming Guide.*
- *Arm China Zhouyi Compass C Programming Guide.*
- *Arm China Zhouyi Compass Getting Started Guide.*
- *Arm China Zhouyi Compass Software Technical Overview.*
- *Arm China Zhouyi Compass NN Compiler User Guide.*
- *Arm China Zhouyi Compass Driver and Runtime User Guide.*

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content, send an e-mail to *errata@armchina.com*. Give:

- The title *Arm China Zhouyi Compass Debugger User Guide.*
- The number 50210006_0110_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm China also welcomes general suggestions for additions and improvements.

Note

Arm China tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Introduction

This chapter provides an overview of the Zhouyi Compass debugger.

It contains the following sections:

1.1 About the debugger

The Zhouyi Compass debugger `aipudbg` is a tool for debugging the NPU core.

The debugger enables you to:

- Debug an NPU C program compiled by `aipucc`.
- Debug the NPU core running on actual hardware or simulator.
- Set breakpoints and single-step NPU applications.
- Inspect and modify memory and variables.

1.2 Debugging on the simulator

Debugging on the simulator platform is quite straightforward. The debugger executes a program on the inner simulator and enables you to debug the program.

1.3 Debugging on hardware

Debugging on the hardware platform is slightly different from debugging on the simulator platform. The real NPU hardware is used to run the target program, and extra peripherals are needed to communicate between the host and NPU.

The following figure shows the connection.

The debugger `aipudbg` is running on the x86 Linux, and it handles the user input and displays the reply from the NPU. The USB-JTAG adaptor is used to connect the PC and CoreSight. TCP/IP connections are used to communicate with the aipudbgserver. The aipudbgserver can also work without TCP/IP connections.

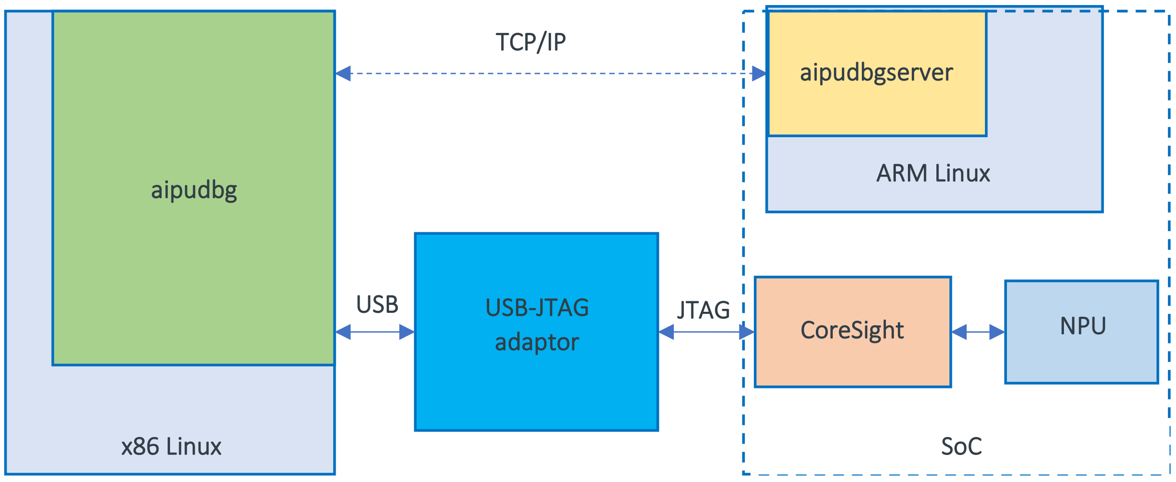


Figure 1-1 Hardware debug architecture

CoreSight

It provides three data paths for the debugger to access the NPU registers. The three data paths are APB, AHB and AXI. The chip developer should provide the configuration file to tell the debugger how to find the device and map the memory. Currently `aipudbg` supports CoreSight-400 and CoreSight-600.

USB-JTAG

The USB-JTAG adaptor is a signal converter for `aipudbg` to access CoreSight. The communication protocol between `aipudbg` and CoreSight is designed based on FTDI FT2232H. Using the 20-pin JTAG as an example, the debugger selects six pins, which are TCK, TDI, TDO, TMS, VCC and GND.

The debugger needs a configuration file to match the USB device and configure the JTAG. The sample configuration file is `aipudbg-TO-PATH/config/jtag-usb.cfg`. The single-core configuration is as follows:

```
#
# ftdi
#

INTERFACE ftdi

# USB Device Feature
# FTDI_VID_PID FTDI_DEVICE_DESC and FTDI_SERIAL_NUMBER are obtained
# from linux command "usb-devices".
```

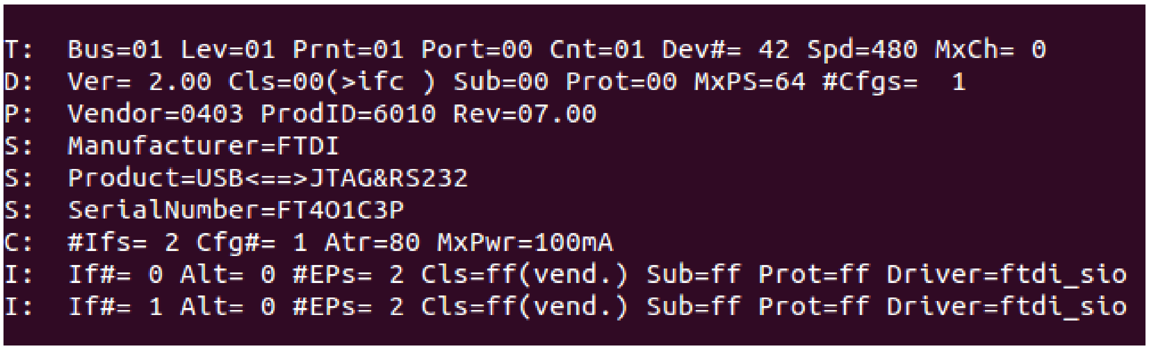
```
FTDI_VID_PID 0x0403 0x6010
FTDI_DEVICE_DESC "Dual RS232-HS"
FTDI_SERIAL_NUMBER "FT58RXED"

# USB Configuration
FTDI_CHANNEL 0
FTDI_LAYOUT_INIT 0x0808 0x0a1b
ADAPTER_KHZ 1000

# APB/AHB/AXI AP_SELECT
CORESIGHT_APB_SELECT  0x00000000
CORESIGHT_AHB_SELECT  0x01000000
CORESIGHT_AXI_SELECT  0x02000000

# Memory map
CORESIGHT_DBG_REG  APBAP 0x00001000
CORESIGHT_AIPU_REG AXIAP 0x64000000
CORESIGHT_DDR      AXIAP 0x80000000

## Enable MMU
ENABLE_MMU         FALSE
```

Note

Lines that start with # are comments.

This configuration file includes two parts of information:

- The first part indicates the feature of the USB device and the configuration for the JTAG interface. The x86 Linux can open the USB device through the configuration.
  - The USB Device Feature can be obtained from Linux command `usb-devices`, as shown in the following sample configuration.
  - Vendor and ProdID are the values of FTDI_VID_PID.
  - Product and SerialNumber are strings for FTDI_DEVICE_DESC and FTDI_SERIAL_NUMBER respectively. If the string is `""`, the feature will be ignored.



```
T:  Bus=01 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 42 Spd=480 MxCh= 0
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=0403 ProdID=6010 Rev=07.00
S:  Manufacturer=FTDI
S:  Product=USB<==>JTAG&RS232
S:  SerialNumber=FT4O1C3P
C:  #Ifs= 2 Cfg#= 1 Atr=80 MxPwr=100mA
I:  If#= 0 Alt= 0 #EPs= 2 Cls=ff(vend.) Sub=ff Prot=ff Driver=ftdi_sio
I:  If#= 1 Alt= 0 #EPs= 2 Cls=ff(vend.) Sub=ff Prot=ff Driver=ftdi_sio
```

- The second part indicates the feature of CoreSight.
- If there is an MMU between the NPU and DDR, change `ENABLE_MMU` to `TRUE`.

The multi-core configuration file is as follows:

```
#
# ftdi
#

INTERFACE ftdi

# USB Device Feature
# FTDI_VID_PID FTDI_DEVICE_DESC and FTDI_SERIAL_NUMBER are obtained
# from linux command "usb-devices".
FTDI_VID_PID 0x0403 0x6010
FTDI_DEVICE_DESC "Dual RS232-HS"
FTDI_SERIAL_NUMBER "FT58RXED"

# USB Configuration
FTDI_CHANNEL 0
FTDI_LAYOUT_INIT 0x0808 0x0a1b
ADAPTER_KHZ 1000

# APB/AHB/AXI AP_SELECT
CORESIGHT_APB_SELECT  0x00000000
CORESIGHT_AHB_SELECT  0x01000000
CORESIGHT_AXI_SELECT  0x02000000

# Memory map of Core 0
CORE_ID 0
CORESIGHT_DBG_REG  APBAP 0x00001000
CORESIGHT_AIPU_REG AXIAP 0x1280010000
CORESIGHT_DDR      AXIAP 0x1000000000

# Memory map of Core 1
CORE_ID 1
CORESIGHT_DBG_REG  APBAP 0x00002000
CORESIGHT_AIPU_REG AXIAP 0x1281010000
CORESIGHT_DDR      AXIAP 0x1000000000

## Enable MMU
```

```
        ENABLE_MMU            FALSE
```

The core specific part includes `CORESIGHT_DBG_REG`, `CORESIGHT_AIPU_REG` and `CORESIGHT_DDR`.

The sample configuration file is for your reference only. You should set up the configuration file based on your actual situations.

1.4 Samples

The `aipudbg` release package includes four samples for different platforms. You can follow the instructions in the `readme.txt` file under the directory of each sample to execute the samples.

Chapter 2
Getting started

This chapter describes the environment, configuration settings, and how to get started with the debugger.

It contains the following sections:

2.1 Setting up the debugger environment

2.1.1 User permissions

To run the simulator platform, you need to find the simulator dynamic libraries under the paths in environment variable LD_LIBRARY_PATH.

To debug on the hardware platform, ensure that you have these permissions:

- Read-write permission to the `/tmp` directory on ARM Linux and to the `/data/local/tmp` directory on the Android platform, because these directories are designed to store temporary files.
- Root permission on ARM Linux to install a module.
- Root permission on aipudbgserver.
- Root permission or USB permission on the host PC to use the USB-JTAG connection.

The path of `libaipudrv.so` needs to be added in LD_LIBRARY_PATH on ARM Linux.

2.1.2 Software dependencies

The recommended operating system is RedHat Enterprise Linux 7.4.

The following lists other packages that are required for the host:

- libusb-1.0.
- libedit and libedit-devel.

The following package is needed on ARM Linux:

- libxml2 version 2.9.1.

2.2 Compiling an application

This section describes how to compile an application.

2.2.1 Compiling for debugging on the simulator platform

The Arm China NPU compiler `aipucc` provides a mechanism for generating the debugging information necessary for `aipudbg` to work properly. The `-g` option must be passed to `aipucc` when an application is compiled to debug with `aipudbg`. In addition, `-O0` is highly recommended because the higher optimization level and the bundle strategy will mess up debugging information, for example:

```
aipucc -O0 -g -o eltwise_max_i8.out eltwise_max_i8.c -mcpu=Z2_1104
```

This command compiles the source file `eltwise_max_i8.c`, and generates an ELF format file named `eltwise_max_i8.out`. The `-mcpu` option specifies the CPU variant. You can find the `eltwise_max_i8.c` file in any of the subdirectories of the `sample` directory in the unzipped debugger release package.

For more information about the available compilation flags, see the *Arm China Zhouyi Compass C Programming Guide*.

To run a program, you need to prepare the program arguments, input data, and a configuration file. It is strongly recommended that you prepare these through `aipubuild`. In the `aipubuild` configuration file, ensure that `mode=run`, and then add `dump=True` to the `[Gbuilder]` part. For more information, see the *Arm China Zhouyi Compass NN Compiler User Guide*.

2.2.2 Compiling for debugging on the hardware platform

Compared to debugging on the simulator platform, further procedures are expected to be taken.

After getting the ELF file from the compiler, you should build the ELF file into a binary file that can run on hardware. For more information about the build, see the *Arm China Zhouyi Compass Driver and Runtime User Guide*.

2.3 Preparing the configuration file

Both the simulator platform and hardware platform need a configuration file to start debugging. However, the contents of these two files are different.

- The configuration file for the simulator mainly describes the simulator settings and program inputs.
- The configuration file for the hardware platform describes the graph file path, program inputs, and JTAG settings.

2.3.1 Configuration file for the simulator platform

As mentioned in *2.2.1 Compiling for debugging on the simulator platform*, the configuration file can be generated by `aipubuild`. The following describes the common options in the configuration file.

- The CONFIG keyword sets the CPU type, and you may find the type on your product information.
- LOG_FILE and LOG_LEVEL define the internal simulator logging.
- The INPUT_INST_CNT keyword must be 1.
- The INPUT_INST_FILE0 contains instruction encoding.
- INPUT_INST_BASE0 is the instruction starting address in DDR.
- INPUT_INST_STARTPC0 is the starting point of the execution. In most cases, it should be the same as INPUT_INST_BASE0.
- INPUT_DATA_CNT indicates the input data count. Each input file is described with two options—INPUT_DATA_FILEn and INPUT_DATA_BASEn which specify the file path and data load address in DDR of the file.

For more information about the configuration options, see Chapter 6 of the *Arm China Zhouyi Compass Software Technical Overview*.

The following is the example configuration file content for the simulator platform.

```
CONFIG=Z2_1104

LOG_LEVEL=0

LOG_FILE=log_default

INPUT_INST_CNT=1

INPUT_INST_FILE0=input.text

INPUT_INST_BASE0=0x0

INPUT_INST_STARTPC0=0x0

INPUT_DATA_CNT=2

INPUT_DATA_FILE0=input.ro

INPUT_DATA_BASE0=0x10000000

INPUT_DATA_FILE1=input.data

INPUT_DATA_BASE1=0x20000000
```

2.3.2 Configuration file for the hardware platform

This configuration file describes the graph file, program inputs and JTAG settings.

The configuration file is divided into three parts. GRAPH INST SETTINGS and SYSTEM SETTINGS are required, while INPUT DATA SETTINGS is optional.

- GRAPH INST SETTINGS: The keyword GRAPH_FILE is the target graph file.
- INPUT DATA SETTINGS: This part sets the input file count and file path.
- SYSTEM SETTINGS: SYSTEM_CONFIG_FILE specifies the USB-JTAG configuration file.

The following is an example configuration file for the hardware platform. Lines that start with # are comments.

```
#-----------------------------------------------------------------

# GRAPH INST SETTINGS (REQUIRED)

#-----------------------------------------------------------------

GRAPH_FILE=aipu.bin


#-----------------------------------------------------------------

# INPUT DATA SETTINGS (OPTIONAL)

#-----------------------------------------------------------------

INPUT_DATA_CNT=2

INPUT_DATA_FILE0=input0.bin

INPUT_DATA_FILE1=input1.bin


#-----------------------------------------------------------------

# SYSTEM SETTINGS (REUIRED)

#-----------------------------------------------------------------

SYSTEM_CONFIG_FILE=../../../config/jtag-usb.cfg
```

2.3.3 Configuration file for the ARM Linux platform

This configuration file describes the graph file and program inputs on ARM Linux.

The configuration file is divided into two parts. GRAPH INST SETTINGS is required, while INPUT DATA SETTINGS is optional.

- GRAPH INST SETTINGS: The keyword GRAPH_FILE is the target graph file on ARM Linux.
- INPUT DATA SETTINGS: This part sets the input file count and file path on ARM Linux.

The following is an example configuration file for the ARM Linux platform. It is similar to the configuration file for the hardware platform. Lines that start with # are comments.

```
#-----------------------------------------------------------------

# GRAPH INST SETTINGS (REQUIRED)

#-----------------------------------------------------------------

GRAPH_FILE=aipu.bin


#-----------------------------------------------------------------

# INPUT DATA SETTINGS (OPTIONAL)

#-----------------------------------------------------------------

INPUT_DATA_CNT=2

INPUT_DATA_FILE0=input0.bin

INPUT_DATA_FILE1=input1.bin
```

2.4 Understanding the command structure

The debugger has a well-structured command syntax. The commands are all in the following form:

```
<noun> <verb> [-options [option-value]] [argument [argument...]]
```

The arguments, options and option values are all white-space separated, and double-quotes are used to protect white-spaces in an argument. If you need to put a backslash or double-quote character in an argument, use a backslash for it in the argument, for example:

```
(aipudbg) file "dir with white-space/filename with \".out"

Current executable set to 'dir with white-space/filename with ".out'
(aipuv2).
```

The `aipudbg` command interpreter performs a shortest unique string match on command names, so the following two commands will both execute the same command:

```
(aipudbg) breakpoint set -n eltwise_max

(aipudbg) br s -n eltwise_max
```

`aipudbg` also supports command completion for source file names, symbol names, and file names. Completion is initiated by pressing the Tab key. Individual options in a command can have different completers, for example, the `–file <path>` option in the `breakpoint` command completes with source files, and the `–name <function-name>` option completes with function names.

2.5 Working with command aliases

The debugger provides a mechanism to construct aliases for commonly used commands.

For example, if you get annoyed typing:

```
(aipudbg) breakpoint set --file  eltwise_max_i8.c --line 72
```

you can use the following to create a command alias:

```
(aipudbg) command alias bfl breakpoint set -f %1 -l %2

(aipudbg) bfl eltwise_max_i8.c 72
```

The aliases for some commonly used commands (for example, `step`, `next` and `continue`) are already added in `aipudbg`.

To remove command alias, use:

```
(aipudbg) command unalias bfl
```

Because `aipudbg` reads the `~/.lldbinit` file at startup, you can store all your aliases in the file so that the aliases are always available to you. The aliases are also documented in the help command, so you can run the help command to check your alias settings, for example:

```
(aipudbg) help bfl

…

'bfl' is an abbreviation for 'breakpoint set -f %1 -l %2'
```

As described in the preceding section, the `~/.lldbinit` file is executed at startup, so you can read the `aipudbg` script using:

```
(aipudbg) command source debugging.cmd
```

2.6 Working with the debugger

Because `aipudbg` is integrated with both the simulator platform and hardware platform, ensure that you do not use the following command by mistake to start debugging:

```
$ aipudbg a.out
```

The following sections describes how to correctly start `aipudbg` and set the target file.

2.6.1 Debugging on the simulator platform

1. Start `aipudbg`.

   ```
   $ aipudbg
   ```

   After the debugger is started, the prompt changes into `(aipudbg)`.

2. Select the simulator backends.

   ```
   (aipudbg) platform select aipu-simulator

     Platform: aipu-simulator

       Triple: x86_64-*-linux-gnu

   OS Version: 3.10.0

     Hostname: 127.0.0.1

   WorkingDir: /home/aipu_debugger/sample/simulator/z2

       Kernel: Linux

      Release: 3.10.0-693.el7.x86_64
   ```

```
         Version: #1 SMP Thu Jul 6 19:56:57 EDT 2017
```

3. Set the target file using the following command:

   ```
   (aipudbg) target create eltwise_max_i8.out
   Current executable set to ' eltwise_max_i8.out' (aipuv2).
   ```

   The arguments in this command must be the same as the arguments in your configuration file, otherwise an error occurs. The `(aipuv2)` in the command output indicates that this is a program compiled to run on Z2 series.

2.6.2 Debugging on the hardware platform

1. Check the hardware connection.
   Ensure that the x86 PC and the hardware board are connected by the USB-JTAG adaptor. The network cable is not required. This solution is used to debug hardware boards that do not support the network.
2. Start `aipudbgserver` on ARM Linux with the root permission.

   ```
   # aipudbgserver jtag --script ./run.cfg
   [AIPU INFO] wait for the debugger to establish a connection.
   ```

   The option `--script` describes the graph file and program inputs. You should put the graph file and program inputs on ARM Linux first. This command will wait for the debugger client connection.
3. Start `aipudbg` on the x86 Linux with the root permission.

   ```
   # aipudbg
   ```

4. Select the hardware platform.

   ```
   (aipudbg) platform select aipu-remote
     Platform: aipu-remote
       Triple: x86_64-*-linux-gnu
   OS Version: 3.10.0
     Hostname: 127.0.0.1
   WorkingDir: /home/aipu_debugger/sample/hardware/z2
       Kernel: Linux
      Release: 3.10.0-693.el7.x86_64
      Version: #1 SMP Thu Jul 6 19:56:57 EDT 2017
   ```

5. Set the target file using the following command:

   ```
   (aipudbg) target create eltwise_max_i8.out
   Current executable set to 'eltwise_max_i8.out' (aipuv2).
   ```

2.6.3 Debugging on the hardware platform with a network connection

1. Ensure that the x86 PC and the hardware board are connected by the USB-JTAG adaptor.
2. Connect the x86 machine and the board with a network cable.

   This solution is used to debug hardware boards with the network support.
3. Start `aipudbgserver` on ARM Linux with the root permission.

   ```
   # aipudbgserver platform –listen *:4321 –server
   ```

This command enables the debugger client to connect to port 4321 from any IP address. You can replace * with the x86 Linux IP address. The graph file and program inputs should be on x86. The debugger will automatically send the files to ARM Linux.

4. Start `aipudbg` on the x86 Linux with the root permission.

   ```
   # aipudbg
   ```

5. Select the hardware platform.

   ```
   (aipudbg) platform select aipu-remote-extend
     Platform: aipu-remote-extend
    Connected: no
   ```

6. Connect `aipudbgserver` on the x86 Linux.

   ```
   (aipudbg) platform connect connect://ARM_LINUX_IP_ADDRESS:4321
     Platform: aipu-remote-extend
       Triple: x86_64-unknown-linux-gnu
   OS Version: 4.14.0 (4.14.0)
       Kernel: #24 SMP Tue Jul 28 15:47:21 CST 2020
     Hostname: (none)
    Connected: yes
   WorkingDir: /home/aipudbgserver
   ```

   Where, `ARM_LINUX_IP_ADDRESS` is the IP address of the ARM Linux system. The port is exactly the same as the port you set in starting `aipudbgserver`.

7. Set the target file using the following command:

   ```
   (aipudbg) target create eltwise_max_i8.out
   Current executable set to 'eltwise_max_i8.out' (aipuv2).
   ```

2.6.4 Attaching to a running hardware

1. Ensure that the x86 PC and the hardware board are connected by the USB-JTAG adaptor.
2. Prepare the ELF file and the binary file on the X86 platform.

   These files are generated by the C compiler and NN compiler.

3. Start `aipudbg` on the x86 Linux.

   ```
   # aipudbg
   ```

4. Select the hardware platform.

   ```
   (aipudbg) platform select aipu-remote
      Platform: aipu-remote
   Connected: no
   ```

   Either the `aipu-remote` platform or the `aipu-remote-extend` platform is available.

5. Set the target file using the following command:

   ```
   (aipudbg) target create eltwise_max_i8.out
   Current executable set to 'eltwise_max_i8.out' (aipuv2).
   ```

6. Start attaching on the hardware platform with `run.cfg`.

   ```
   (aipudbg) process attach --script ./run.cfg
   ```

The configuration file `./run.cfg` describes the graph file, program inputs and JTAG settings. This file is required. The debugger identifies the hardware based on the JTAG settings in this configuration file.

Chapter 3
NPU program execution

This chapter describes the NPU program execution.

It contains the following sections:

3.1 Starting a program

3.1.1 Setting arguments and running on the simulator platform

The configuration files for the `launch` command of the simulator platform and hardware platform are different. For the simulator platform, the arguments of your program are specified by INPUT_DATA_FILE in the configuration file. For more information, see the *Arm China Zhouyi Compass Software Technical Overview*.

Set the configuration file using the `set target.run-args` command:

```
(aipudbg) settings set target.run-args ./run.cfg
```

```
(aipudbg) process launch
```

The debugger will find the configuration file named `run.cfg` in the current directory. If you do not set the file, either a specified file or a default file is needed, otherwise `aipudbg` will fail to run the process.

3.1.2 Setting arguments and running on the hardware platform

The arguments of the hardware platform are as follows:

```
(aipudbg) settings set target.run-args ./run.cfg core-id=0
```

```
(aipudbg) process launch
```

Where, `core-id=`*`N`* is used to specify the core that the program runs on. The core ID must appear in the configuration file as described in *1.3 Debugging on hardware*.

3.2 Continuing a program

After a program is stopped, you can continue the program using:

```
(aipudbg) process continue
```

A useful option is `-i <unsigned-integer>`. This option enables you to ignore <N> crossings of the breakpoint (if it exists).

3.3 Step/Next

Continue running your program until control reaches a different source line, then stop it and return control to `aipudbg`.

```
(aipudbg) thread step-in
```

The `next` operation will make the program continue to the next source line in the current (innermost) stack frame. This is similar to `step`, but function calls that appear within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the original stack level that is:

```
(aipudbg) thread step-over
```

`step-inst` is an operation that is used to execute a bundle of machine instructions, then stop and return to `aipudbg`.

```
(aipudbg) thread step-inst
```

`step-inst-over` is an operation that is used to execute a bundle of machine instructions, but if it is a function call, it proceeds until the function returns.

```
(aipudbg) thread step-inst-over
```

3.4 Interrupt

When a program is running, the debugger may attempt to interrupt it by using Ctrl-C or the `process interrupt` command.

```
(aipuodb) process interrupt
```

3.5 Until

Continue running until a source line past the current line, in the current stack frame, is reached. This operation is used to avoid single stepping through a loop more than once. It is like the `next` option, except that when `until` encounters a jump, it automatically continues execution until the program counter is greater than the address of the jump.

```
(aipudbg) thread until 100
```

3.6 Return

To return from the current location, use:

```
(aipudbg) thread step-out
```

3.7 Stop hook

A stop hook provides a way to run an `aipudbg` command each time the program stops.

```
(aipudbg) target stop-hook add --one-liner "frame variable size"

Stop hook #1 added.
```

This command creates stop hook `1` to show variable `size` each time it stops. The stop hook ID `1` can be used to enable/disable/delete the hook.

Use `-n`/`--name` to stop making the hook only work at the specified function.

```
(aipudbg) target stop-hook add  --name test --one-liner "frame variable
count"

Stop hook #2 added.
```

List all hooks:

```
(aipudbg) target stop-hook list

Hook: 1

  State: enabled

  Commands:

frame variable size1
```

Disable a hook:

```
(aipudbg) target stop-hook disable 1
```

Enable a hook:

```
(aipudbg) target stop-hook enable 1
```

Delete a hook:

```
(aipudbg) target stop-hook delete 1
```

3.8 Attach

The attach command provides a way to debug NPU hardware that has already run a program. The attach command is limited to use in hardware debugging.

```
(aipudbg) process attach --script ./run.cfg
```

```
(aipudbg) process attach -s ./run.cfg
```

The debugger identifies the hardware based on the JTAG settings in the configuration file. Because this is remote debugging, the debugger cannot attach the hardware by the process ID.

3.9 Detach

The detach command provides a way to quit debugging on the hardware by attaching and to allow the hardware to resume the program. After detaching, the hardware is in running state and can be attached again.

```
(aipudbg) process detach
```

3.10 Kill

The kill command provides a way to quit debugging a program. When attaching on the hardware platform, the kill command allows the hardware to terminate the program that is already running. After killing, the hardware is in idle state.

```
(aipudbg) process kill
```

3.11 Quit

Quit provides a way to exit from `aipudbg`. When attaching on the hardware platform, the quit command like the detach command will quit debugging and allow the hardware to resume the program.

```
(aipudbg) quit
```

Chapter 4
Breakpoints and watchpoints

This chapter describes the breakpoints and watchpoints.

It contains the following sections:

4.1 Breakpoints

There are no limits on the breakpoint count.

Multiple ways are supported to set a breakpoint. For example:

```
(aipudbg) breakpoint set --name eltwise_max

Breakpoint 1: where = eltwise_max_i8.out`eltwise_max + 928 at
eltwise_max_i8.c:41:24, address = 0x000113a0
```

Setting a breakpoint creates a logical breakpoint, which can resolve to one or more locations. For example, a file and line breakpoint might result in multiple locations if the file and line are inlined in different places in your code. The logical breakpoint has an integer ID, and its locations have an ID within their parent breakpoint (the two IDs are joined by a '.', such as 1.1 in the following example).

`aipudbg` will always make a breakpoint from your specification, even if it cannot find any locations that match the specification. In this case, the output will report the breakpoint as pending, for example:

```
(aipudbg) breakpoint set --name __max

Breakpoint 5: no locations (pending).

WARNING:  Unable to resolve breakpoint to any actual locations.
```

When you set a pending breakpoint, it means that you have made a typo.

4.1.1 Symbolic breakpoints

To set a breakpoint at the entry of a function, use `-n`/`--name`. The option can be specified multiple times:

```
(aipudbg) breakpoint set --name eltwise_max

(aipudbg) breakpoint set -n eltwise_max
```

4.1.2 Line breakpoints

To set a breakpoint on a specific line number, use the `-f`/`--file` and `-l`/`--line` options.

```
(aipudbg) breakpoint set --file eltwise_max_i8.c --line 72
```

4.1.3 Conditional breakpoints

To add a breakpoint with a condition, use `-c`/`--condition`.

```
(aipudbg) breakpoint set --file eltwise_max_i8.c --line 72 --condition 'i ==
1'
```

4.1.4 Breakpoint hook

A breakpoint hook is a series of `aipudbg` commands to be executed each time the program stops at the breakpoint. To add a breakpoint with a hook, use `-C`/`--command`.

```
(aipudbg) breakpoint set --file eltwise_max_i8.c --line 63 --command 'frame
variable remain_size'
```

When the program stops at this breakpoint, the `frame variable remain_size` command will execute to show the value of local variable `remain_size`.

4.1.5 Checking breakpoints

To check the breakpoints that you set, use:

```
(aipudbg) breakpoint list

Current breakpoints:
```

```
1: name = 'eltwise_max', locations = 1

  1.1: where = eltwise_max_i8.out`eltwise_max + 928 at
eltwise_max_i8.c:41:24, address = eltwise_max_i8.out[0x000113a0], unresolved,
hit count = 0


2: file = 'eltwise_max_i8.c', line = 57, exact_match = 0, locations = 1

  2.1: where = eltwise_max_i8.out`eltwise_max + 1360 at
eltwise_max_i8.c:57:5, address = eltwise_max_i8.out[0x00011550], unresolved,
hit count = 0


3: file = 'eltwise_max_i8.c', line = 72, exact_match = 0, locations = 1

  3.1: where = eltwise_max_i8.out`eltwise_max + 2208 at
eltwise_max_i8.c:72:36, address = eltwise_max_i8.out[0x000118a0], unresolved,
hit count = 0


4: file = 'eltwise_max_i8.c', line = 78, exact_match = 0, locations = 1

  4.1: where = eltwise_max_i8.out`eltwise_max + 2656 at
eltwise_max_i8.c:78:23, address = eltwise_max_i8.out[0x00011a60], unresolved,
hit count = 0
```

The output of `breakpoint list` indicates whether the breakpoint location is resolved. A location remains unresolved until the program starts to run. For example, after you launch an executable in the debugger, the breakpoint address changes the status to resolved.

```
(aipudbg) br list

Current breakpoints:

1: name = 'eltwise_max', locations = 1, resolved = 1, hit count = 1

  1.1: where = eltwise_max_i8.out`eltwise_max + 928 at
eltwise_max_i8.c:41:24, address = 0x00000420, resolved, hit count = 1


2: file = 'eltwise_max_i8.c', line = 57, exact_match = 0, locations = 1,
resolved = 1, hit count = 0

  2.1: where = eltwise_max_i8.out`eltwise_max + 1360 at
eltwise_max_i8.c:57:5, address = 0x000005d0, resolved, hit count = 0


3: file = 'eltwise_max_i8.c', line = 72, exact_match = 0, locations = 1,
resolved = 1, hit count = 0

  3.1: where = eltwise_max_i8.out`eltwise_max + 2208 at
eltwise_max_i8.c:72:36, address = 0x00000920, resolved, hit count = 0


4: file = 'eltwise_max_i8.c', line = 78, exact_match = 0, locations = 1,
resolved = 1, hit count = 0

  4.1: where = eltwise_max_i8.out`eltwise_max + 2656 at
eltwise_max_i8.c:78:23, address = 0x00000ae0, resolved, hit count = 0
```

4.1.6 Disabling/Enabling a breakpoint

To disable a breakpoint, use:

```
(aipudbg) breakpoint disable 2
1 breakpoints disabled.
```

To enable a breakpoint, use:

```
(aipudbg) breakpoint enable 2
1 breakpoints enabled.
```

If no breakpoints are specified in the commands, all breakpoints will be disabled/enabled.

4.1.7 Deleting a breakpoint

To delete a breakpoint, use:

```
(aipudbg) breakpoint delete 2
1 breakpoints deleted; 0 breakpoint locations disabled.
```

If no breakpoints are specified in the command, all breakpoints will be deleted.

4.2 Watchpoints

A watchpoint is used to stop execution whenever the value of an expression changes, without having to predict a particular place where this may happen.

Depending on different platforms, watchpoints may be implemented in software or hardware way.

- A software watchpoint is implemented by single-stepping the program and testing the variable's value each time, which is slower than normal execution. But this may still be worth it, to catch every change in DDR or SRAM, whether due to the ld/st instructions or DMA operations.
- A hardware watchpoint is implemented by hardware itself. Hardware watchpoints execute very quickly, and the debugger reports a change in value at the exact instruction where the change occurs. However, the hardware watchpoints can only watch DDR, if the memory is accessed by ld/st instructions.

The type of a watchpoint can be:

- `read`, which means that it is triggered on a read.
- `write`, which means that it is triggered on a write.
- `read_write`, which means that it is triggered on both read and write.

The simulator platform only supports software watchpoints of the `write` type.

The hardware platform only supports up to two hardware watchpoints.

You can set a watchpoint either on a variable or an expression.

4.2.1 Variable watchpoint

To set a `write` watchpoint on a scalar variable on the hardware platform, run the following command:

```
(aipudbg)  watchpoint set variable each_loop_size -w write

Watchpoint created: Watchpoint 1: addr = 0x600142e0 size = 4 state = enabled
type = w

    declare @ '/home/aipu_debugger/sample/hardware/z2/eltwise_max_i8.c:51'

    watchpoint spec = 'each_loop_size'

    new value at 0x600142e0: 16384
```

To set a `write` watchpoint on a vector variable on the simulator platform, run the following command:

```
(aipudbg) watchpoint set variable -w write arr_v32b_input_l1[1]

Watchpoint created: Watchpoint 2: addr = 0xf0080020 size = 32 state = enabled
type = w

    declare @ '/home/aipu_debugger/sample/simulator/z2/eltwise_max_i8.c:46'

    watchpoint spec = 'arr_v32b_input_l1[1]'

    new value at 0xf0080020: (-98, -15, 125, 75, 106, 122, 36, 116, 94, 116,
95, 83, 62, 109, 41, 83, -30, -97, 70, 66, 36, -67, 88, 53, 3, 58, 85, 60,
63, 65, -12, 47)
```

Only TEC0's memory of the vector variable can be watched.

Where, `write` is the default type, so the `-w write` can be skipped.

To set a `read` watchpoint on a scalar variable on the hardware platform, run the following command:

```
 (aipudbg) watchpoint set variable -w read -s 4 calcu_num

Watchpoint created: Watchpoint 3: addr = 0x600142d8 size = 4 state = enabled
type = r
```

```
    declare @ '/home/aipu_debugger/sample/hardware/z2/eltwise_max_i8.c:55'

    watchpoint spec = 'calcu_num'

   new value at 0x600142d8: 512
```

To set a `read_write` watchpoint on a scalar variable on the hardware platform, run the following command:

```
(aipudbg) watchpoint set variable -w read_write remain_size

Watchpoint created: Watchpoint 4: addr = 0x600142dc size = 4 state = enabled
type = rw

    declare @ '/home/aipu_debugger/sample/hardware/z2/eltwise_max_i8.c:52'

    watchpoint spec = 'remain_size'

    new value at 0x600142dc: 524288
```

4.2.2 Expression watchpoint

To set a watchpoint on the full memory of a vector variable on the simulator platform, run the following commands:

```
(aipudbg) frame variable &arr_v32b_input_l1[1]

(v32int8_t *) &[1] = 0xf0080020

(v32int8_t *) &[1] = 0xf0180020

(v32int8_t *) &[1] = 0xf0280020

(v32int8_t *) &[1] = 0xf0380020

(aipudbg) watchpoint set expression -w write -s 32 -- 0xf0080020

Watchpoint created: Watchpoint 1: addr = 0xf0080020 size = 32 state = enabled
type = w

    new value at 0xf0080020: 1270071688

(aipudbg) watchpoint set expression -w write -s 32 -- 0xf0180020

Watchpoint created: Watchpoint 2: addr = 0xf0180020 size = 32 state = enabled
type = w

    new value at 0xf0180020: 3059752635

(aipudbg) watchpoint set expression -w write -s 32 -- 0xf0280020

Watchpoint created: Watchpoint 3: addr = 0xf0280020 size = 32 state = enabled
type = w

    new value at 0xf0280020: 183581564

(aipudbg) watchpoint set expression -w write -s 32 -- 0xf0380020

Watchpoint created: Watchpoint 4: addr = 0xf0380020 size = 32 state = enabled
type = w

    new value at 0xf0380020: 154070371
```

4.2.3 Checking watchpoints

You can check the watchpoints that you have set using the following command:

```
(aipudbg) watchpoint list
```

```
Current watchpoints:
Watchpoint 1: addr = 0x2000d2d8 size = 4 state = enabled type = r
    declare @ '/home/aipu_debugger/sample/simulator/z2/eltwise_max_i8.c:55'
    watchpoint spec = 'calcu_num'
    new value: 512
Watchpoint 2: addr = 0xf0080020 size = 32 state = enabled type = rw
    declare @ '/home/aipu_debugger/sample/simulator/z2/eltwise_max_i8.c:46'
    watchpoint spec = 'arr_v32b_input_l1[1]'
    new value: (-120, -63, -77, 75, 16, -38, 21, 116, 94, -54, 44, -30, -19,
109, 41, 3, -30, -128, 70, -37, -36, -67, 88, 53, -37, 58, 85, 60, -14, -43,
-12, 47)
```

4.2.4 Disabling/Enabling a watchpoint

To disable a watchpoint, use:

```
(aipudbg) watchpoint disable 2
1 watchpoints disabled.
```

To enable a watchpoint, use:

```
(aipudbg) watchpoint enable 2
1 watchpoints enabled.
```

If no watchpoints are specified in the commands, all watchpoints will be disabled/enabled.

4.2.5 Deleting a watchpoint

To delete a breakpoint, use:

```
(aipudbg) breakpoint delete 2
```

If no watchpoints are specified in the command, all watchpoints will be deleted.

Chapter 5
Inspecting NPU state

This chapter describes how to inspect NPU state.

It contains the following sections:

5.1 Process/Thread status

To show the status of the process, run the following command:

```
(aipudbg) process status

Process 0 stopped

* thread #1, name = 'eltwise_max_i8.out', stop reason = breakpoint 3.1

    frame #0: 0x00000920 eltwise_max_i8.out`eltwise_max(size=524288,
p_in_addr1=543686656, p_in_addr2=544210944, p_out_addr=544735232) at
eltwise_max_i8.c:72:36
```

To show all the threads of the process, run the following command:

```
(aipudbg) thread list

* thread #1: tid = 0x0000, 0x00000920
eltwise_max_i8.out`eltwise_max(size=524288, p_in_addr1=543686656,
p_in_addr2=544210944, p_out_addr=544735232) at eltwise_max_i8.c:72:36, name =
'eltwise_max_i8.out', stop reason = breakpoint 3.1
```

There is always one thread in the output, because the NPU is a single core processor.

To show the status of the current thread, run the following command:

```
(aipudbg) thread info

thread #1: tid = 0x0000, 0x00000920
eltwise_max_i8.out`eltwise_max(size=524288, p_in_addr1=543686656,
p_in_addr2=544210944, p_out_addr=544735232) at eltwise_max_i8.c:72:36, name =
'eltwise_max_i8.out', stop reason = breakpoint 3.1
```

5.2 Backtrace

A backtrace is a summary of how your program gets its current location. After a program is stopped, `aipudbg` will show the current backtrace with thread state, frame state, source location and stop reason. For example:

```
(aipudbg) process launch

…

Process 0 stopped

* thread #1, name = 'eltwise_max_i8.out', stop reason = breakpoint 1.1

    frame #0: 0x00000420 eltwise_max_i8.out`eltwise_max(size=524288,
p_in_addr1=543686656, p_in_addr2=544210944, p_out_addr=544735232) at
eltwise_max_i8.c:41:24

…
```

The full backtrace will show all the caller frames. It starts with the currently executing frame (frame zero), followed by its caller frame (frame one), and on up the stack.

```
(aipudbg) thread backtrace

* thread #1, name = 'eltwise_max_i8.out', stop reason = breakpoint 3.1

  * frame #0: 0x00000920 eltwise_max_i8.out`eltwise_max(size=524288,
p_in_addr1=543686656, p_in_addr2=544210944, p_out_addr=544735232) at
eltwise_max_i8.c:72:36
```

Where, frame 0 is the innermost (currently executing) frame, frame 1 is the frame that called the innermost frame, and so on. The highest level frame is the entry function. The frame marked with a `*` is the current frame.

5.2.1 Selecting a frame

In the preceding section, the `thread backtrace` command can show all the frames of the current thread, and `thread <N>` can change to `frame select <N>`, for example,

```
(aipudbg) frame select 1
```

To select the stack frame that is called by the older stack frame, use:

```
(aipudbg) frame select --relative=1
```

To select the stack frame that is called by the newer stack frame, use:

```
(aipudbg) frame select --relative=-1
```

A specified relative offset is also supported:

```
(aipudbg) frame select --relative=2

(aipudbg) frame select --relative=-3
```

5.3 Scalar variables

The most convenient way to inspect a frame's arguments and local variables is to use the `frame variable` command:

```
(aipudbg) frame variable

(uint32_t) size = 524288

(uint32_t) p_in_addr1 = 543686656

(uint32_t) p_in_addr2 = 544210944

(uint32_t) p_out_addr = 544735232

(uint32_t) tec_num = 1

(uint32_t) lsram_size = 16384

(int32_t) each_loop_size = 16384

(int32_t) remain_size = 524288

(int32_t) calcu_num = 512

(int32_t) i = 0
```

With no arguments specified, the `frame variable` command will show all the arguments and local variables.

To show all the global variables, use `-g` or `--show-globals`.

```
(aipudbg) frame variable -g
```

To show a specified variable, pass the variable name to the command.

```
(aipudbg) frame variable calcu_num

(int32_t) calcu_num = 512
```

The `frame variable` command is not a full expression parser but it supports a few simple operations like `&`, `*`, `->`, `[]`. The array brackets can be used on pointers to treat pointers as arrays.

```
(aipudbg) frame variable &arr_v32b_input_l1[1]

(v32int8_t *) &[1] = 0xf0080020

(aipudbg) frame variable &arr_v32b_input_l1

(v32int8_t (*)[512]) &arr_v32b_input_l1 = 0xf0080000

(aipudbg) frame variable arr_v32b_input_l1[0]

(v32int8_t) arr_v32b_input_l1[0] = (15, 70, 104, 83, 126, -54, 82, 106, -36,
62, 116, -17, -15, 86, 49, 15, 34, 73, 121, 28, 113, -43, 13, 114, 99, -35, -
31, -15, 32, 71, -3, -46)
```

5.4 Vector variables

The display of vector variables also uses the `frame variable` command. The debugger has optimized the format of vector variables.

To check a variable on LSRAM/GSRAM, use:

```
(aipudbg) frame variable arr_v32b_input_l1[0]

(v32int8_t) arr_v32b_input_l1[0] = (-18, 70, 44, 83, 126, -54, -26, 76, -36,
-58, 116, -17, -15, 86, -31, -88, -22, 73, 121, -77, -87, -67, 13, 114, -9, -
35, -31, -38, 32, 71, -3, -46)

(v32int8_t) arr_v32b_input_l1[0] = (107, 88, 23, 92, 74, 8, 115, -70, 110, -
83, 74, 6, 114, 100, -75, 92, 28, 91, 30, 29, -110, 37, 9, -33, -51, -107, -
124, -72, -5, -121, -18, 116)

(v32int8_t) arr_v32b_input_l1[0] = (-41, 50, 93, -93, 76, 126, 11, 101, -105,
-64, 98, -72, 51, -97, -22, 40, 22, 88, 42, -39, -128, -64, -46, 5, 37, -66,
-11, 122, 121, 32, -21, -99)

(v32int8_t) arr_v32b_input_l1[0] = (5, -122, 47, 65, 88, 107, -31, 35, 112,
31, -16, -61, 74, 35, 78, 38, 23, -63, 42, -43, -96, -57, -81, -23, -56, 20,
31, 53, 97, 3, 30, -68)
```

The variables of type `v32bool_t/v16bool_t/v8bool_t` are also well-formatted:

```
(aipudbg) frame variable p1

On Tec 0: (v32bool_t) p1 = (0xffffffff)

On Tec 1: (v32bool_t) p1 = (0xffffffff)

On Tec 2: (v32bool_t) p1 = (0xffffffff)

On Tec 3: (v32bool_t) p1 = (0xffffffff)
```

5.5 Register

If you execute the `register read` command, it will print all general-purpose registers and *Program Counter* (PC).

```
(aipudbg) register read

General Purpose Registers:

        r0 = 0x00000000

        r1 = 0x00000200  eltwise_max_i8.out`eltwise_max + 384 at
eltwise_max_i8.c:40

        r2 = 0xf0180000

        r3 = 0xf0280000

        r4 = 0x00000850  eltwise_max_i8.out`eltwise_max + 2000 at
eltwise_max_i8.c:68:9

        r5 = 0x00000000

        r6 = 0x00000000

        r7 = 0x00000000

        r8 = 0x00000000

        r9 = 0x00000000

       r10 = 0x00000000

       r11 = 0x00000000

       r12 = 0xf0380000

       r13 = 0x00000000

       r14 = 0x00000000

       r15 = 0xf0080000  eltwise_max_i8.out`eltwise_max.arr_v32b_input_l1

       r16 = 0x00004000  eltwise_max_i8.out`__memcpy_d2l1_4tec + 528

       r17 = 0x00004000  eltwise_max_i8.out`__memcpy_d2l1_4tec + 528

       r18 = 0x00004000  eltwise_max_i8.out`__memcpy_d2l1_4tec + 528

       r19 = 0x00004000  eltwise_max_i8.out`__memcpy_d2l1_4tec + 528

       r20 = 0x00004000  eltwise_max_i8.out`__memcpy_d2l1_4tec + 528

       r21 = 0x00004000  eltwise_max_i8.out`__memcpy_d2l1_4tec + 528

       r22 = 0x00000000

       r23 = 0x00000000

       r24 = 0xf0087fff

       r25 = 0x00000001

       r26 = 0xe0000500

       r27 = 0x00003f08  eltwise_max_i8.out`__memcpy_d2l1_4tec + 280

       r28 = 0x10000040

       r29 = 0x00003eb0  eltwise_max_i8.out`__memcpy_d2l1_4tec + 192
       r30 = 0x00000890  eltwise_max_i8.out`eltwise_max + 2064 at
eltwise_max_i8.c:70:22

       r31 = 0x00000000

        pc = 0x00000920  eltwise_max_i8.out`eltwise_max + 2208 at
eltwise_max_i8.c:72:36
```

To show a specified register value, just pass its name to the `register read` command.

```
(aipudbg) register read r1

      r1 = 0x00000200  eltwise_max_i8.out`eltwise_max + 384 at
eltwise_max_i8.c:40
```

In this way, you can read all kinds of registers, including T registers, Acc registers, and P registers, for example:

```
(aipudbg) register read tec0:t0 tec1:t3

tec0:t0 = {0x0f 0x46 0x68 0x53 0x7e 0xca 0x52 0x6a 0xdc 0x3e 0x74 0xef 0xf1
0x56 0x31 0x0f 0x22 0x49 0x79 0x1c 0x71 0xd5 0x0d 0x72 0x63 0xdd 0xe1 0xf1
0x20 0x47 0xfd 0xd2}

tec1:t3 = {0xff 0xff 0xff 0xff 0xff 0x7f 0x00 0x00 0x21 0x90 0x13 0x58 0x7e
0x7f 0x00 0x00 0x07 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xa0 0x21 0xfe 0x59
0x7e 0x7f 0x00 0x00}
```

To write a register, run the following command:

```
(aipudbg) register write r1 32

(aipudbg) register write tec1:t3 "{0x12 0x23 0x45 0x67 0x89 0xab 0xcd 0xef
0x01 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 0x99 0xaa 0xbb 0xcc 0xdd 0xee
0xff 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x77}"
```

5.6 Memory

The memory read command is used to read memory.

Various options can be added to the command, such as:

- `-f/--format` to specify the format.
- `-c/--count` to specify the read count.
- `-s/--size` to specify the element size.

To read 8 32-bit data from the address of a variable in hex format, run the following commands:

```
(aipudbg) frame variable &arr_v32b_input_l0[1]

(v32int8_t *) &[0] = 0xf0000020

(aipudbg) memory read -f x -s 4 -c 8 0xf0000020

0xf0000020: 0x247df19e 0xb6247a6a 0x535f741c 0x5388bf3e

0xf0000030: 0x42e29fd3 0x23eca924 0x2ce09e03 0x8eaa413f
```

The following are some advanced options:

- `-o/--outfile` to specify an output file.
- `-b/--binary` to specify the binary format of the output file.
- `-r/--force` to exceed the maximum size.

To save the memory read command as a file in ASCII format, run the following command:

```
(aipudbg) memory read -f x -s 4 -c 8 0xf0000020 –o output.log
```

To save the result of a layer as a binary file, run the following commands:

```
(aipudbg) frame variable p_out_addr -f x

(uint32_t) p_out_addr = 0x60700000

(aipudbg) frame variable size -f x

(uint32_t) size = 0x00080000

(aipudbg) memory read 0x60700000 -c 0x80000 -s 1 -o p_out_addr.bin -b --force

524288 bytes written to 'p_out_addr1.bin'
```

For detailed information about more options, use `help memory read`.

The `memory write` command is used to write memory.

The following are the options of the `memory write` command:

- `-f/--format` to specify the format.
- `-i/--infile` to specify an input file.
- `-o/--offset` to specify an offset within the input file.
- `-s/--size` to specify the element size.

To change the value of a vector memory, run the following commands:

```
(aipudbg) frame variable &arr_v32b_input_l0[1]

(v32int8_t *) &[0] = 0xf0000020

(aipudbg) memory write -f x -s 4 0xf0000020 0x11111111 0x22222222 0x33333333
0x44444444 0x55555555 0x66666666 0x77777777 0x88888888
```

To use a binary file to change the input at the address p_in_addr1 in the layer, run the following commands:

```
(aipudbg) frame variable p_in_addr1 -f x
(uint32_t) p_in_addr1 = 0x60600000
(aipudbg) frame variable size -f x
(uint32_t) size = 0x00080000
(aipudbg) memory write -i new_input.bin -s 0x80000 0x60600000
524288 bytes were written to 0x60600000
```

For detailed information about more options, use `help memory write`.

5.7 Disassemble

Disassemble specified instructions in the current target. By default, the `disassemble` command works for the current function for the current thread and stack frame.

To disassemble the entire content of the given function name, run the following command:

```
(aipudbg) disassemble --name eltwise_max --count 4

eltwise_max_i8.out`eltwise_max:

    0x80 <+0>:  sub.u sp, sp, #0xf0    &  nop  &  nop  &  nop

    0x90 <+16>: mov   r26, #0xe00      &  nop  &  nop  &  nop

    0xa0 <+32>: lsl   r26, r26, #0x14  &  nop  &  nop  &  mov  r25, #0x500

    0xb0 <+48>: ld    r0, [r28, #0x0]  &  nop  &  nop  &  or   r26, r26, r25
```

To disassemble around the current PC, run the following command:

```
(aipudbg) disassemble --pc

eltwise_max_i8.out`eltwise_max:

->  0x430 <+944>: and  r0, r25, #0x1f  &    nop    &    nop    &    nop

    0x440 <+960>: st   r0, [r27, #0x1e4]     &    nop    &    nop    &    nop

    0x450 <+976>: ld   r0, [r27, #0x1e4]     &    nop    &    nop    &    nop

    0x460 <+992>: lsl  r0, r0, #0xe    &    nop    &    nop    &    nop
```

To start disassembling at an address, run the following command:

```
(aipudbg) disassemble --start-address 0x430 –count 4

eltwise_max_i8.out`eltwise_max:

->  0x430 <+944>: and  r0, r25, #0x1f  &    nop    &    nop    &    nop

    0x440 <+960>: st   r0, [r27, #0x1e4]     &    nop    &    nop    &    nop

    0x450 <+976>: ld   r0, [r27, #0x1e4]     &    nop    &    nop    &    nop

    0x460 <+992>: lsl  r0, r0, #0xe    &    nop    &    nop    &    nop
```

To disassemble every time you perform an instruction level single step in the currently selected thread stop, run the following command:

```
(aipudbg) target stop-hook add

Enter your stop hook command(s).  Type 'DONE' to end.

> disassemble --pc

> DONE

Stop hook #1 added.

(aipudbg) si

eltwise_max_i8.out`eltwise_max:

->  0x430 <+944>: and  r0, r25, #0x1f  &    nop    &    nop    &    nop

    0x440 <+960>: st   r0, [r27, #0x1e4]     &    nop    &    nop    &    nop

    0x450 <+976>: ld   r0, [r27, #0x1e4]     &    nop    &    nop    &    nop

    0x460 <+992>: lsl  r0, r0, #0xe    &    nop    &    nop    &    nop
```

```
Process 0 stopped
* thread #1, name = 'eltwise_max_i8.out', stop reason = instruction step into
    frame #0: 0x00000730 eltwise_max_i8.out`eltwise_max(size=524288,
p_in_addr1=543686656, p_in_addr2=544210944, p_out_addr=544735232) at
eltwise_max_i8.c:41:24
   38   __entry void eltwise_max(uint32_t size, uint32_t p_in_addr1,
   39                            uint32_t p_in_addr2, uint32_t p_out_addr)
   40   {
-> 41       uint32_t tec_num = __builtin_aipu_gettecnum();
   42       uint32_t lsram_size = SIZE * sizeof(v32int8_t) * tec_num;
   43
   44       // request for sram and calculate the parameter we need
```