

PulseRain
TECHNOLOGY

Doc# PGM-0966-00001, Rev 1.0.0

Copyright © 2021

PulseRain Technology, LLC.



<https://www.pulserain.com>



858-877-3485

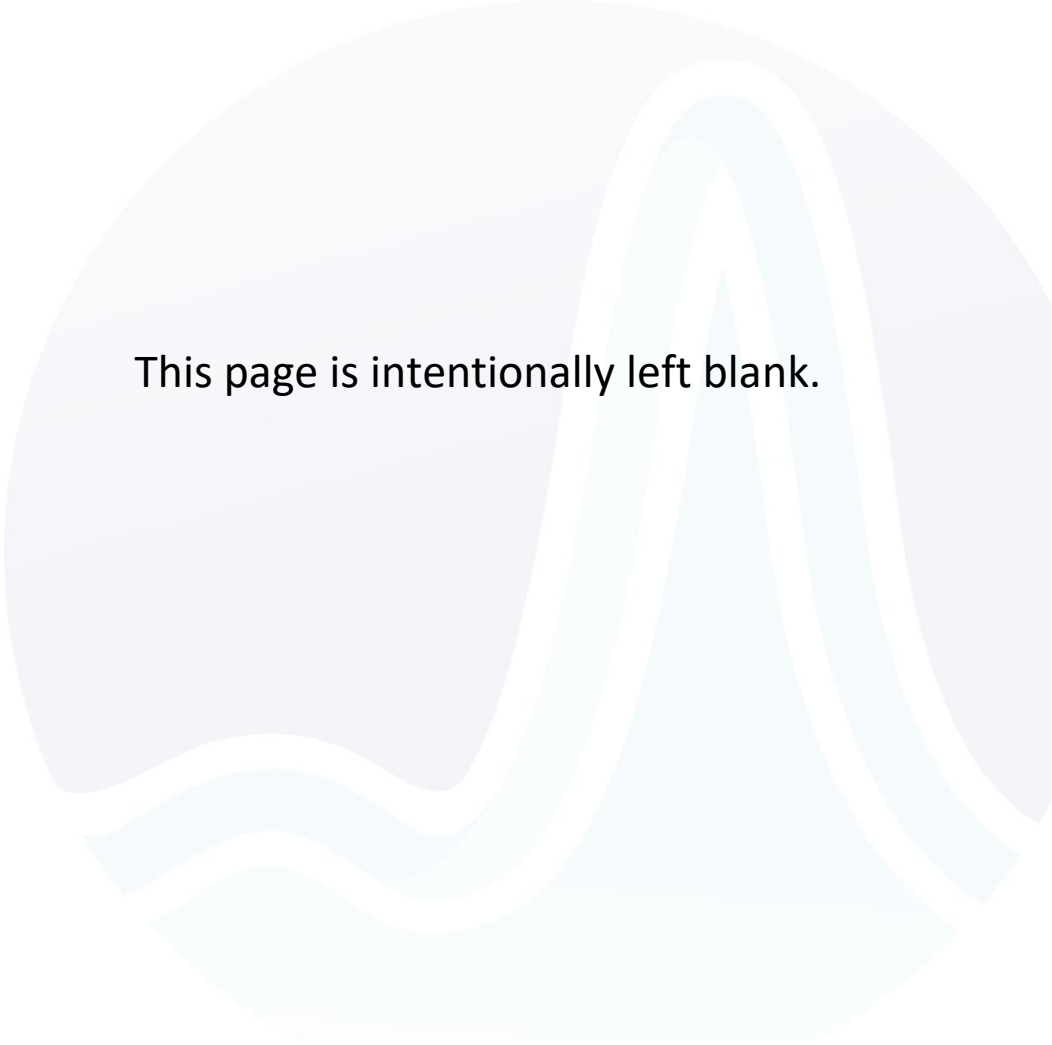


858-408-9550

PulseRain GRV3000D Digital Signal Controller

Programmer's Guide

May, 2021



This page is intentionally left blank.

Table of Contents

REFERENCES.....	0
ACRONYMS AND ABBREVIATIONS	1
1 INTRODUCTION	2
1.1 HARDWARE	2
1.2 INSTRUCTION SET.....	2
1.3 DSP COPROCESSOR.....	2
1.4 SOFTWARE LIBRARY.....	3
1.5 DEV BOARD / PROBE	3
2 HARDWARE ARCHITECTURE	4
2.1 OVERVIEW.....	4
2.2 I-TCM (TIGHTLY COUPLED MEMORY FOR INSTRUCTIONS)	4
2.3 D-TCM (TIGHTLY COUPLED MEMORY FOR DATA)	5
2.4 REGISTER FILE	6
2.5 PIPELINE STAGES.....	6
2.5.1 <i>Instruction Fetcher (IF)</i>	7
2.5.2 <i>Pre-decoder</i>	7
2.5.3 <i>Reservation Station</i>	7
2.5.4 <i>Instruction Decoder (ID)</i>	7
2.5.5 <i>Functional Units (FU)</i>	7
2.5.5.1 ALU (Arithmetic Logic Unit).....	7
2.5.5.2 Branch Unit	7
2.5.5.3 Store Unit	8
2.5.5.4 Load Unit.....	8
2.5.5.5 Division Unit.....	8
2.5.5.6 MUL Unit	8
2.5.5.7 System Unit	8
2.5.5.8 DSP Coprocessor	8
2.5.6 <i>Write Back (WB)</i>	9
2.6 AXI4-LITE SWITCH.....	9
3 ADDRESS MAPPING	12
3.1 ADDRESS SPACE.....	12
3.2 GPR (GENERAL PURPOSE REGISTER) AND CALLING CONVENTION.....	13
3.3 CSR (CONTROL AND STATUS REGISTER) AND MACHINE MODE.....	14
3.3.1 <i>Privilege Level and Machine Mode</i>	14
3.3.2 <i>Control and Status Register (CSR)</i>	14
3.3.2.1 mvendorid (Vendor ID)	16
3.3.2.2 marchid (Architecture ID).....	16
3.3.2.3 mimpid (Implementation ID).....	16
3.3.2.4 mstatus	16
3.3.2.5 mscratch.....	16
3.3.2.6 CSRs related to Interrupt / Exception	16

3.3.2.7	Counters.....	17
3.4	MEMORY SPACE	17
3.4.1	<i>NOP Region in I-TCM</i>	18
3.4.2	<i>Top Half and Bottom Half in D-TCM</i>	19
3.4.3	<i>AXI4-Lite Switch and Memory Mapped Registers</i>	19
3.4.3.1	System Timer	20
3.4.3.2	UART	20
3.4.3.3	GPIO	21
3.4.3.4	External Interrupt.....	22
3.4.4	<i>Trap (Exception and Interrupt)</i>	22
3.4.4.1	Trap Trigger and Handle.....	23
3.4.4.2	Trap Return	25
3.4.4.3	WFI.....	25
3.4.4.4	ECALL and EBREAK	26
3.4.4.5	External Interrupt.....	26
3.5	JTAG DEBUG MODULE AND HW BASED BOOTLOADER.....	27
4	SOFTWARE – GENERAL FLOW	28
4.1	INSTRUCTION SUPPORTED.....	28
4.1.1	<i>Base Instruction</i>	28
4.1.2	<i>CSR Instruction</i>	30
4.1.3	<i>ECALL / EBREAK Instruction</i>	30
4.1.4	<i>M Extension</i>	30
4.1.5	<i>C Extension (Optional)</i>	31
4.2	COMPILER	32
4.3	ASSEMBLER.....	34
4.4	IMAGE LOADING	35
5	SOFTWARE - ARDUINO FLOW	37
5.1	INTRODUCTION TO ARDUINO.....	37
5.2	ARDUINO IDE	37
5.2.1	<i>Introduction to Bare Metal Systems</i>	38
5.2.2	<i>Installation of Arduino IDE and Board Support Package for the GRV3000D</i>	39
5.2.3	<i>The Workflow for Arduino IDE</i>	41
5.2.4	<i>What is Under the Hood for Arduino IDE</i>	41
5.3	ARDUINO LANGUAGE.....	42
5.3.1	<i>Data Type</i>	42
5.3.2	<i>APIs</i>	43
5.3.2.1	Digital IO.....	43
5.3.2.2	Time and Delay.....	43
5.3.2.3	Serial Port.....	44
5.3.2.4	Interrupt.....	44
5.4	INTERRUPT HANDLER.....	44
5.4.1	<i>Shared ISR (Interrupt Service Routine)</i>	44
5.4.2	<i>Device ISR (Interrupt Service Routine)</i>	47
5.4.3	<i>Device ISR Example</i>	48
5.5	ARDUINO LIBRARY.....	49

5.5.1	Use 3rd-party library in Arduino IDE	49
5.5.1.1	Install 3rd-party Library	49
5.5.1.2	Use 3rd-party Library in Sketches	50
5.5.2	Use 3rd-party library in Arduino IDE	50
5.5.2.1	Create GitHub Repository	50
5.5.2.2	Submit the Library for Review	52
6	SOFTWARE – JTAG DEBUG	53
6.1	EXTERNAL DEBUG PROBE	53
6.2	SEGGER J-LINK AND EMBEDDED STUDIO	53
6.2.1	Connect the Probe	54
6.2.2	Build Project under the Embedded Studio for RISC-V	54
6.2.3	Debug under the Embedded Studio for RISC-V	59
7	SOFTWARE – DSP (DIGITAL SIGNAL PROCESSING)	61
7.1	INTRODUCTION TO HINT INSTRUCTIONS	61
7.2	FUNCTION CALL WITH DSP HINT	61
7.2.1	Function Calls on RISC-V	62
7.2.2	Format of the DSP HINT	62
7.3	DSP COMMAND ON PULSERAIN GRV3000D	64
7.3.1	DSP Command with Single Source	64
7.3.1.1	DSP Command: Exponent / Normalization	64
7.3.1.2	DSP Command: Absolute Value	65
7.3.1.3	DSP Command: Negate	66
7.3.1.4	DSP Command: Clipping	66
7.3.1.5	DSP Command: Arithmetic Shift Right	66
7.3.1.6	DSP Command: Logic Shift Right	67
7.3.1.7	DSP Command: Shift Left	67
7.3.1.8	DSP Command: Arithmetic Shift Right and Round	68
7.3.2	DSP Command with Dual Sources	68
7.3.2.1	FIR (Finite Impulse Response) Filter	68
7.3.2.2	BIQUAD Filter	70
7.3.2.3	Dot Product (Real and Complex)	72
7.3.2.4	Fast Fourier Transform (FFT)	75
8	SOFTWARE – PMSIS LIBRARY	79
8.1	EXPONENT / NORMALIZATION	79
8.2	ABSOLUTE VALUE	79
8.3	NEGATE	80
8.4	CLIPPING	80
8.5	ARITHMETIC SHIFT RIGHT	80
8.6	LOGIC SHIFT RIGHT	81
8.7	ARITHMETIC SHIFT RIGHT AND ROUND	81
8.8	FIR FILTER	82
8.9	BIQUAD FILTER	82
8.10	DOT PRODUCT	83
8.11	FFT	83
8.12	COMPILE AND SELF-TEST	84

References

1. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213, Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, December 2019.
2. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified, Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, June 2019
3. RISC-V External Debug Support, Version 0.13.2, Editors: Tim Newsome, Megan Wachs, SiFive, Inc., Fri Mar 22, 2019
4. David Patterson and Andrew Waterman, The RISC-V Reader: An Open Architecture Atlas
5. Andrew Waterman: Design of the RISC-V Instruction Set Architecture
6. Building Embedded Systems – Programmable Hardware, Changyi Gu, APress Media, July 2016
<http://www.apress.com/us/book/9781484219188>
7. Computer Organization and Design - The Hardware / Software Interface (3rd edition), David A. Patterson and John L. Hennessy, Morgan Kaufmann Publication, 2005
8. AMBA AXI and ACE Protocol Specification, AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite, ARM 2011
9. Wikipedia – Arduino, <https://en.wikipedia.org/wiki/Arduino>

Acronyms and Abbreviations

Acronyms / Abbreviations	Definition
ABI	Application Binary Interface
ADC	Analog to Digital Converter
API	Application Program Interface
BCD	Binary-Coded Decimal
CISC	Complex Instruction Set Computer
CODEC	Coder-Decoder
CP	Co-processor
DPTR	Data Pointer
DSP	Digital Signal Processor
FARM	FPGA + Arduino + RISC-V + Make
FPGA	Field Programmable Gate Array
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IO	Input and Output
IRQ	Interrupt Request Line
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LSB	Least Significant Bit
MCU	Microcontroller Unit
MSB	Most Significant Bit
NOP	No Operation
OCD	On-chip Debugger
OS	Operating System
PC	Personal Computer or Program Counter
PSW	Program Status Word
PWM	Pulse Width Modulation
RISC	Reduced Instruction Set Computer
RISC-V	Berkeley Fifth Generation RISC Instruction Set
SFR	Special Function Register
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
TCM	Tightly Coupled Memory
UART	Universal Asynchronous Receiver-Transmitter

1 Introduction

1.1 Hardware

The PulseRain GRV3000D is a fixed-point digital signal controller. It features a RISC-V dual-issue in-order processor core, plus a versatile digital signal co-processor. The highlights of the GRV3000D are as following:

- 6-stage pipeline that can issue and execute up to two instructions in one clock cycle
- 24-bit address bus and 64-bit data bus
- A true Harvard architecture with completely separate data memory and instruction memory
- AXI4-Lite switch for peripherals
- Support two 32-bit MAC operation in one clock cycle
- Built-in DSP coprocessor (CP)
- JTAG port for Debugging

1.2 Instruction Set

The GRV3000D supports the following RISC-V instruction sets (ISA):

- RISC-V RV32I Base Integer Instruction Set, Version 2.1 (Ref [1])
- RISC-V "Zifencei", Instruction-Fetch Fence, Version 2.0 (Ref [1])
- RISC-V "M" Standard Extension for Integer Multiplication and Division, Version 2.0 (Ref [1])
- (Optional) RISC-V "C" Standard Extension for Compressed Instructions, Version 2.0 (Ref [1])
- RISC-V "Zicsr", Control and Status Register (CSR) Instructions, Version 2.0 (Ref [1])
- RISC-V Base Counters and Timers (Ref [1])

1.3 DSP Coprocessor

Unlike other RISC-V DSP solutions, the DSP Coprocessor (CP) in GRV3000D triggers the DSP acceleration through RISC-V HINT, without using any additional private instructions. In this way, the DSP acceleration is non-intrusive and it does not rely on any specific compilers to function. And the DSP CP currently supports the following operations:

- Normalization for samples in block
- Absolute value for samples in block
- Negative value for samples in block
- Value clipping for samples in block
- Left logic shift for samples in block
- Right logic shift, right arithmetic shift and right arithmetic shift with rounding for samples in block
- FIR filter for samples in block, with a throughput of two samples per clock cycle
- BIQUAD cascade filter for samples in block, with a throughput of three clock cycles per BIQUAD
- Dot Product (Real and Complex)
- FFT (Fast Fourier Transform)

1.4 Software Library

To further facilitate the adoption of the GRV3000D in DSP applications, an open-source software library called PMSIS (PulseRain Microcontroller Software Interface Standard) is also provided to the general public for free.

1.5 Dev Board / Probe

And to help developers evaluate the GRV3000D, a FPGA bitstream is also provided for the Digilent Arty7-100T Dev Board. Software can either be downloaded through JTAG with Segger J-Link Probe, or through UART with Arduino IDE.

2 Hardware Architecture

2.1 Overview

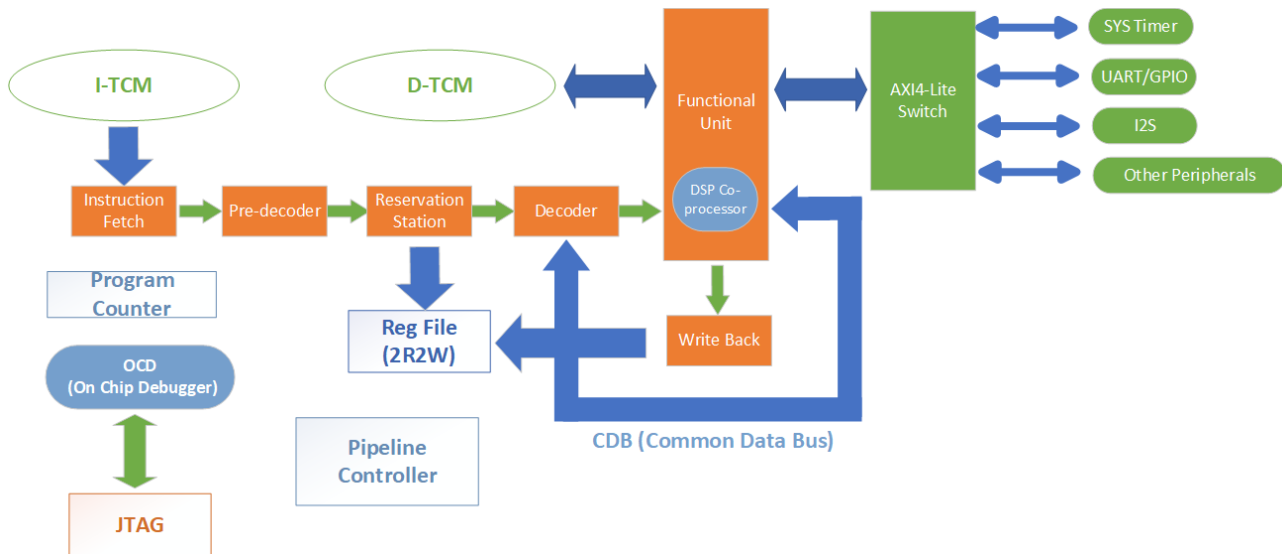


Figure 2-1 Hardware Overview

As illustrated in Figure 2-1, the GRV3000D's major hardware blocks are as following:

- I-TCM: Tightly Coupled Memory for Instructions
- D-TCM: Tightly Coupled Memory for Data
- Register File
- 6-stage pipeline and pipeline controller
- CDB (Common Data Bus)
- OCD (On Chip Debugger)
- Peripherals through AXI-Lite Switch.

2.2 I-TCM (Tightly Coupled Memory for Instructions)

As mentioned early, the GRV3000D has a true Harvard architecture with completely separated instruction memory and data memory. All instructions for the GRV3000D are stored in the I-TCM. The I-TCM has a read latency of one clock cycle.

To software developers, the I-TCM is a read-only memory. It can only be accessed by OCD or by Instruction Fetcher. The write port (by OCD) for I-TCM is 32-bit wide, while the read port for I-TCM (by OCD or Instruction Fetcher) is 64-bit. Thus, the I-TCM can feed the rest of the processor with a throughput of two instructions per clock cycle.

The size of the I-TCM depends on the hardware variants. It is typically configured as 64KB.

2.3 D-TCM (Tightly Coupled Memory for Data)

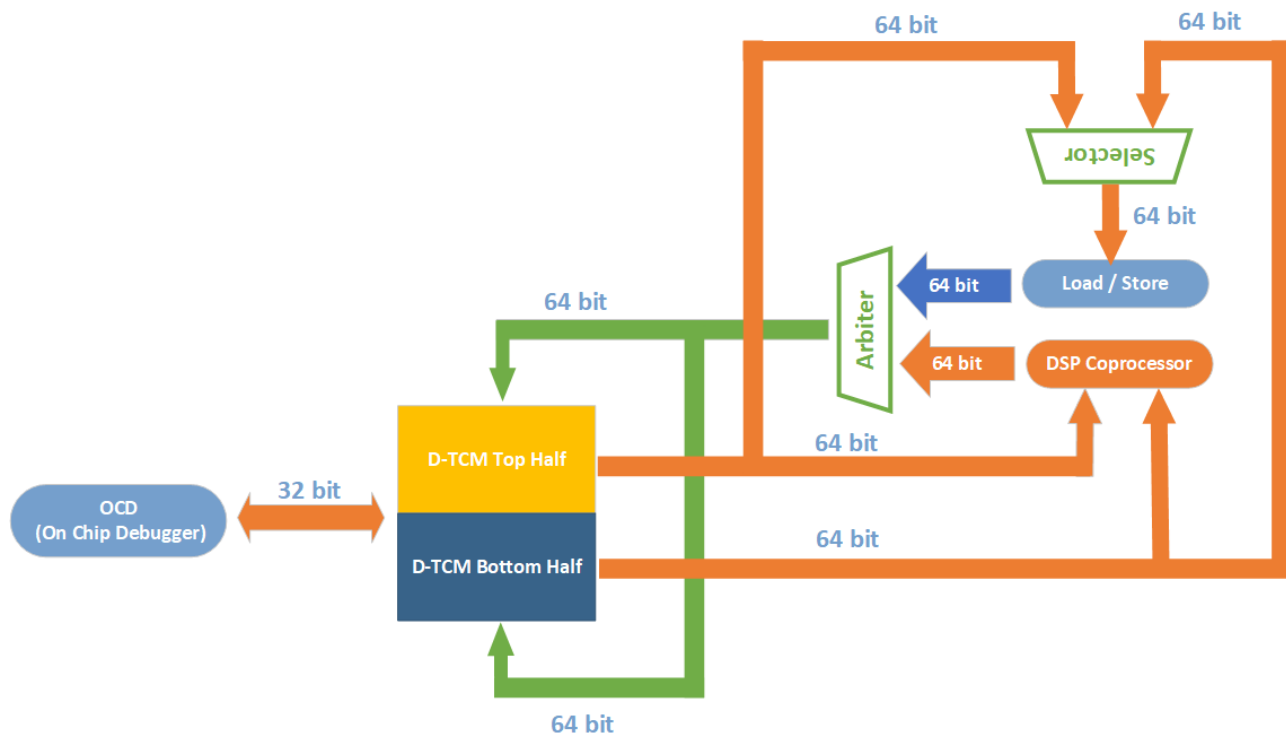


Figure 2-2 D-TCM

Like the I-TCM, the D-TCM also has a read latency of one clock cycle. As shown in Figure 2-2, the D-TCM can be read from or written to by OCD, by the Load/Store Unit or by the DSP CP.

Unlike the I-TCM, the D-TCM is physically divided into two parts: The top half and bottom half, each occupies 50% of the D-TCM size. And each half of the D-TCM has a 64-bit wide read bus and 64-bit write bus, which means two 32-bit LOAD or two 32-bit STORE can be carried out in a single clock cycle, assuming the two 32-bit words for R/W are next to each other in address.

Dividing the D-TCM into two sections will help the DSP Coprocessor. For FIR and BIQUAD operations, it is intended that samples are stored in one of the sections while coefficients are store in the other. In this way, total of 128 bits of data can be pulled out of the D-TCM in each clock cycle. And two 32-bit MAC operation can thus be executed simultaneously. The details of DSP operation will be discussed in later chapters of this document.

And the size of the D-TCM depends on the hardware variants. It is typically configured as 64KB.

2.4 Register File

As defined by the RISC-V standard (Ref [1]), the GRV3000D has 32 GPR (General Purpose Registers). Each GPR is 32-bit wide. Among the 32 GPRs, register 0 is a special one, for which it always holds the value as zero, regardless of what is being written into it.

And for the GRV3000D, its register file can carry two independent 32-bit read operations and two 32-bit independent write operations in a single clock cycle.

2.5 Pipeline Stages

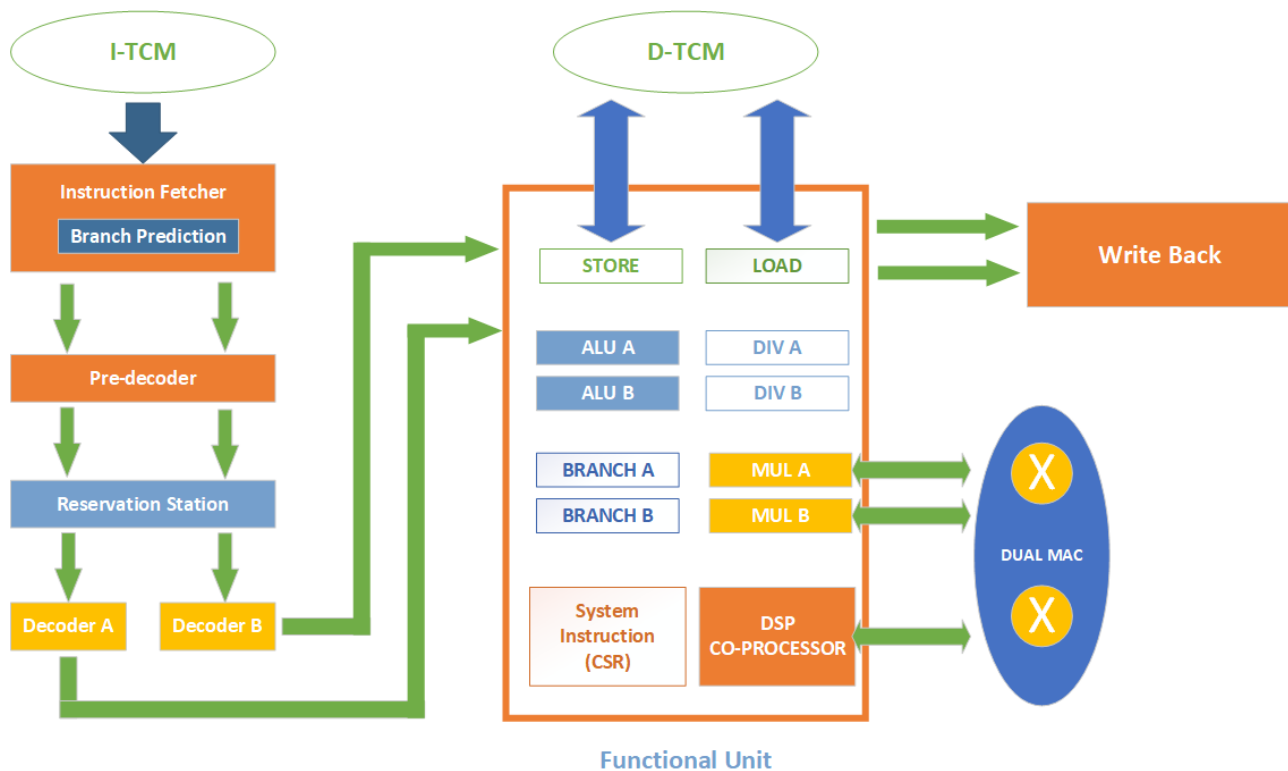


Figure 2-3 Pipeline of 6 Stages

The GRV3000D features a 6-stage pipeline, as illustrated in Figure 2-3. The 6 stages are:

1. Instruction Fetcher
2. Pre-decoder
3. Reservation Station
4. Decoder
5. Functional Units
6. Write Back

As the GRV3000D is a dual-issue processor, the instructions flow through those stages in a 64-bit bus.

2.5.1 Instruction Fetcher (IF)

As shown in Figure 2-3, the IF stage reads instructions out of the I-TCM through a 64-bit wide bus. And to improve the efficiency of the pipeline, the IF stage also contains a branch prediction unit. And the software developers can make the branch prediction close to be 100% hit ratio, if they follow certain rules. The details will be discussed in later chapters of this document.

And the 64-bit bus forms a dual issue packet of two instructions. Based on their conceptual execution order, the two instructions are named as instruction A and B by this document.

2.5.2 Pre-decoder

The Pre-decoder will sort the instructions into the correspondent function unit (FU), and extract the FU specific parameters.

2.5.3 Reservation Station

The Reservation Station acts like a reservoir, and conceptually it can be viewed as a FIFO. On one end, it takes two instructions per clock cycle from the Pre-decoder, while on the other end the instruction decoder will pull instructions out of the FIFO for further processing.

2.5.4 Instruction Decoder (ID)

There are two ID units in this stage. And there is also a data forwarding unit in the ID to monitor the latest register update on the CDB (See Figure 2-1).

2.5.5 Functional Units (FU)

The decoded instructions will be executed by various functional units, as shown in Figure 2-3. It has another data forwarding unit to monitor the latest update on the CDB (See Figure 2-1). The GRV3000D has the following FUs:

2.5.5.1 ALU (Arithmetic Logic Unit)

The GRV3000D has two ALUs. Each ALU can complete a shift/and/or/exclusive-or/addition/subtraction instruction or a LUI/AUIPC instruction in one clock cycle.

2.5.5.2 Branch Unit

The GRV3000D has two Branch Units. Each Branch Unit can execute one unconditional branch (JAL/JALR) or one conditional branch (BEQ/BNE/BLT/BGE/BLTU/BGEU) in one clock cycle.

For JAL instruction, the IF stage will automatically fetch the instruction from its target address, thus the branch unit will simply skip the JAL and the pipeline will not stall or flush.

For JALR, its target address will be determined by the Branch Unit. The pipeline will be flushed and reloaded. The branch prediction unit in the IF stage will not generate dynamic prediction for JALR.

For conditional branch, the IF stage will fetch the next instruction based on static branch prediction. If the software developers follow the guidelines provided in Ref [0], the conditional branch will not cause pipeline stall for the majority of the time. The details will be discussed in later chapters of this document.

PulseRain GRV3000D – Programmer's Guide

2.5.5.3 Store Unit

The GRV3000D has one Store Unit that can execute one store instruction (32-bit or less), or merge two 32-bit store instructions with consecutive addresses (and with the same base register) into one 64-bit store operation. The latency of the store operation depends on the Arbiter (Figure 2-2). And it takes one clock cycle to complete a store instruction, if the Arbiter is at the active position for the Store Unit.

2.5.5.4 Load Unit

The GRV3000D has one Load Unit that can execute one load instruction (32-bit or less), or merge two 32-bit load instructions with consecutive addresses (and with the same base register) into one 64-bit load operation. The latency of the load operation depends on the Arbiter (Figure 2-2). And it takes two clock cycles to complete a load instruction, if the Arbiter is at the active position for the Load Unit.

2.5.5.5 Division Unit

The GRV3000D has two Division Units that can execute 32-bit divisions with Radix-4 SRT algorithm. Each 32-bit division will take 20 clock cycles to complete.

2.5.5.6 MUL Unit

The GRV3000D has two MUL Units. Each MUL unit can complete a 32-bit multiplication in one clock cycle.

2.5.5.7 System Unit

The System Unit will handle the following instructions defined by RISC-V specification (Ref [1][2]):

- The RISC-V "Zicsr", namely Control and Status Register Instructions
- ECALL
- WFI (Wait for the Interrupt)
- MRET

As mentioned in Section 2.5.1, the IF will put out the initial dual-issue packet as Instruction A and B. However, unlike other instructions, the instructions handled by the System Unit will not be sorted to the B position. In other words, if it is fetched by the IF at the B position initially, the IF will put it into the next issue packet.

And the System Unit will complete the "Zicsr" and ecall in one clock cycle.

As for the WFI instruction, the pipeline will stall until interrupt happens.

The latency of the MRET depends on the accuracy of the RAS (Return Address Stack, part of branch prediction) inside the IF stage. And its overhead is only one clock cycle if the RAS provides the right address.

2.5.5.8 DSP Coprocessor

The DSP Coprocessor will use RISC-V HINT instructions to achieve DSP acceleration. The details will be discussed in later chapters of this document.

And the DSP Coprocessor and the MUL units share a dual-MAC DSP block, as indicated in Figure 2-3.

2.5.6 Write Back (WB)

The WB stage will put data on the CDB and write them to the destination registers (specified as "rd" in RISC-V instructions). As mentioned in Section 2.4, the Register File has two write ports. If both instruction A and instruction B write to the same register, the write from instruction B will prevail.

2.6 AXI4-Lite Switch

To accommodate peripherals, the GRV3000D has an AXI4-Lite Switch. It is configured as a 2 x N matrix, with two bus masters (The optional DMA and the Load/Store Unit) and N bus slaves (Peripherals like UART, GPIO, I2S etc.). And N is configured as 4 by default.

The details of the AXI4-Lite specification can be found in Ref [8]. For the AXI4-Lite switch in GRV3000D, it can execute one write transaction and one read transaction simultaneously. The pathways of the AW (Write Address) channel and AR (Read Address) channel are shown in Figure 2-4. The pathways of W (Write Data) channel are shown in Figure 2-5. The pathways of B (Write Response) channel are shown in Figure 2-6. And the pathways of R (Read Data) channel are shown in Figure 2-7. Basically, the AW and AR channels will have two clock cycles of latency. The write data will reach the slave port at the same time as the address.

And the address allocation of the AXI4-Lite switch will be discussed in later chapters of this document.

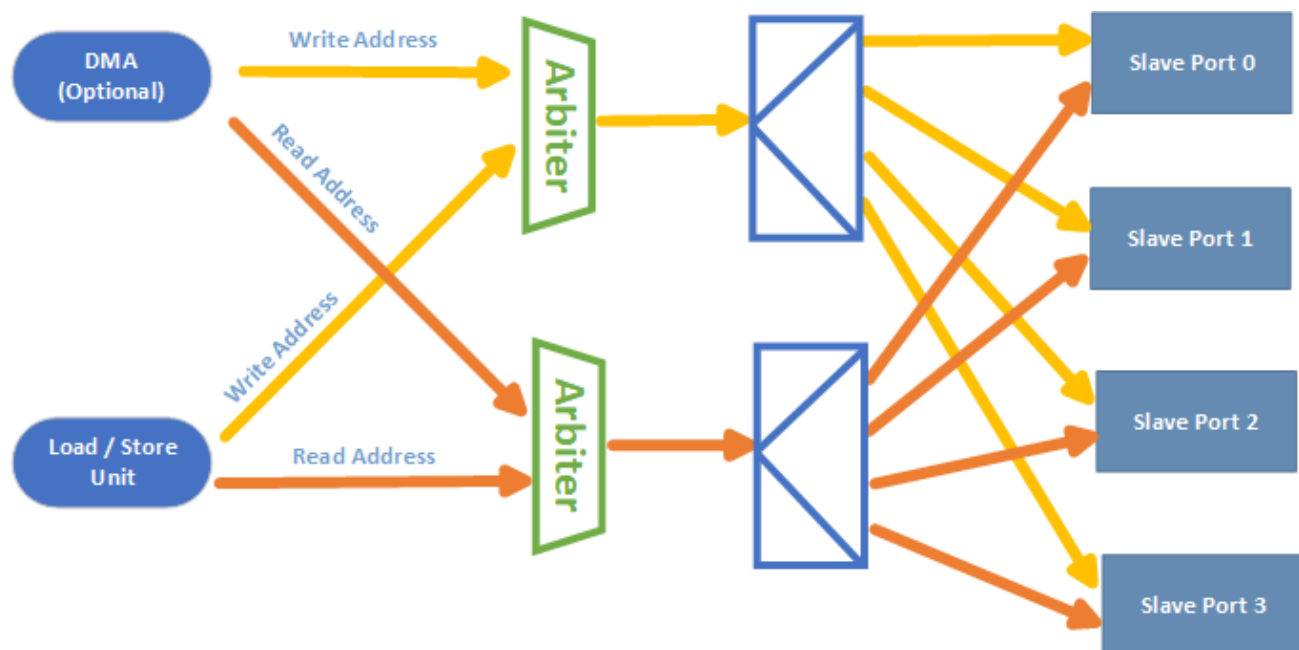


Figure 2-4 AXI4-Lite Switch, AW and AR Channel

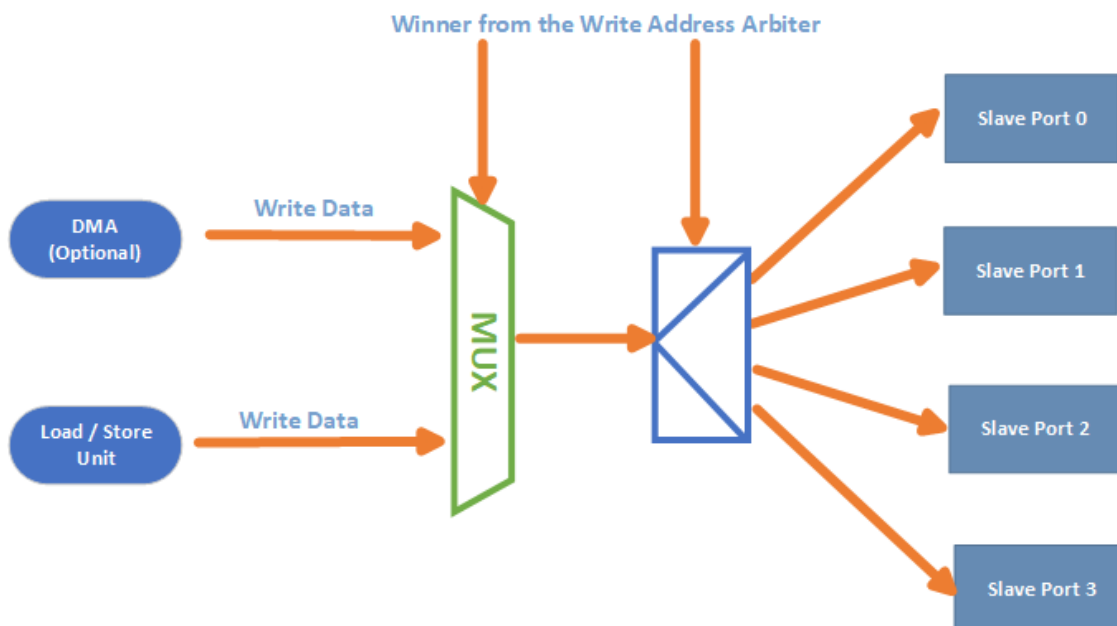


Figure 2-5 AXI4-Lite Switch, W Channel

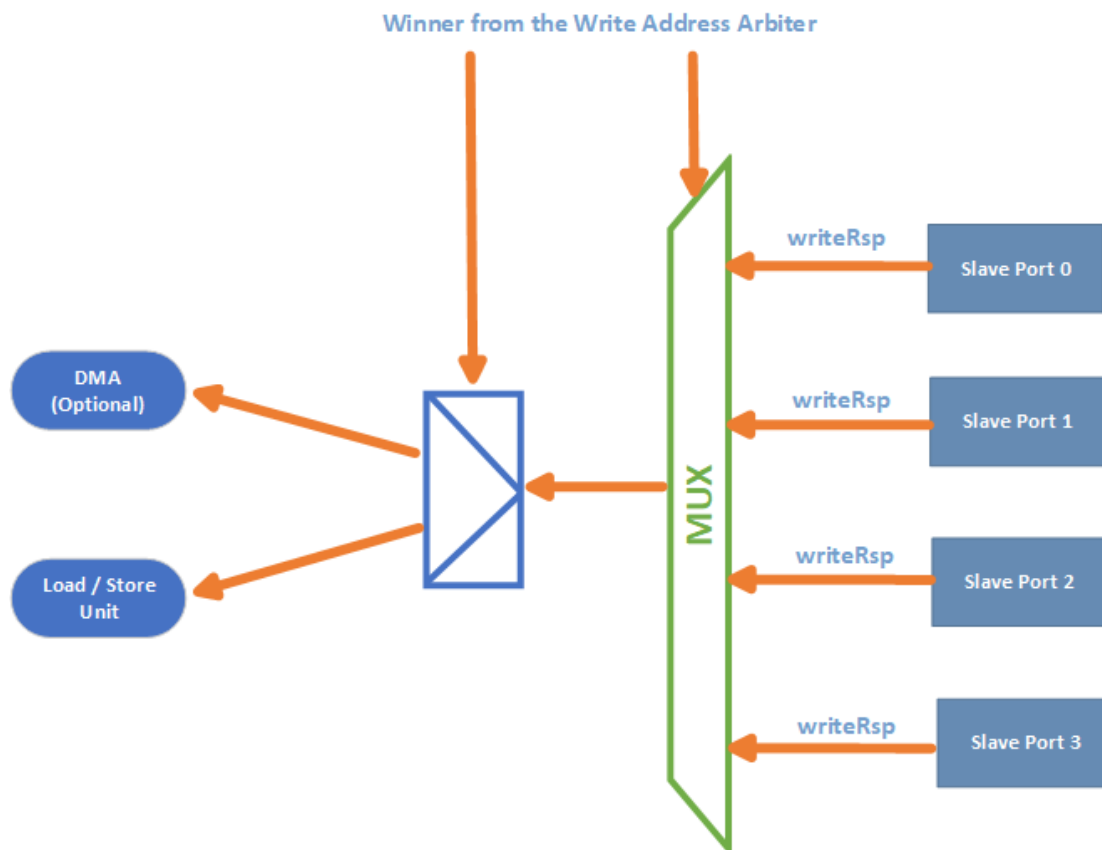


Figure 2-6 AXI4-Lite Switch, B Channel

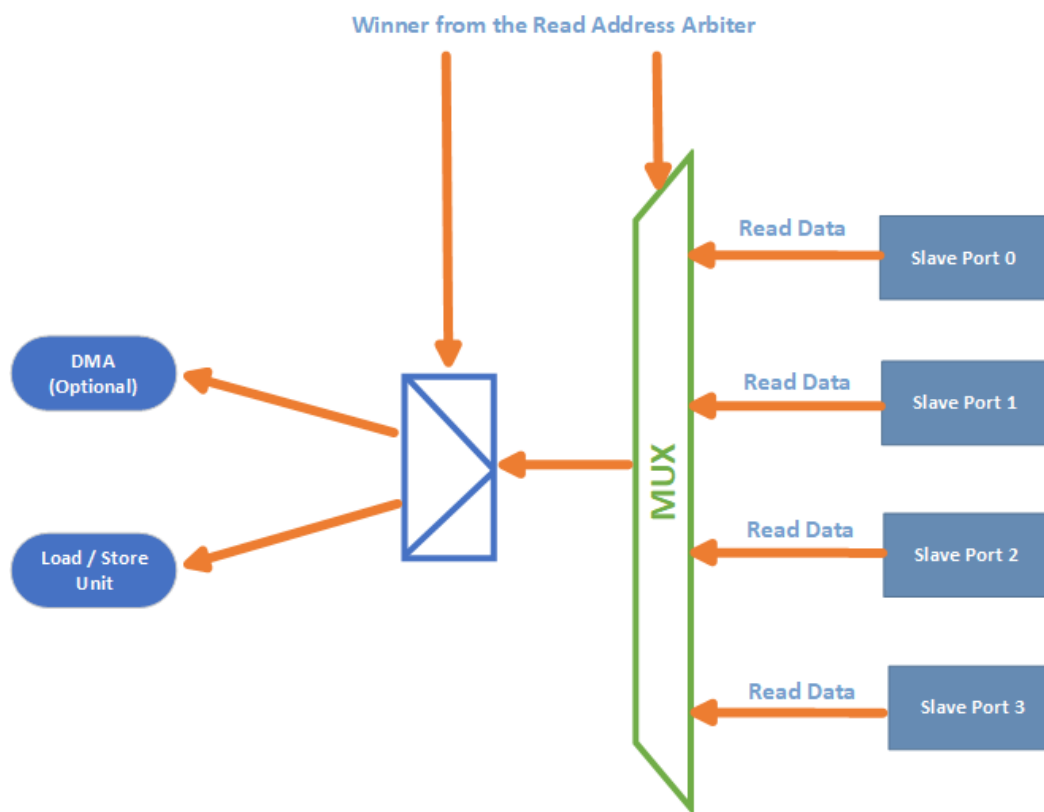


Figure 2-7 AXI4-Lite Switch, R Channel

3 Address Mapping

3.1 Address Space

Per RISC-V ISA Manual (Ref [1][2]), the GRV3000D has 3 separate address spaces, as shown in Figure 3-1. Those address spaces are:

1. GPR (General Purpose Registers)
2. CSR (Control and Status Registers)
3. Memory Space (Include Memory Mapped Registers)

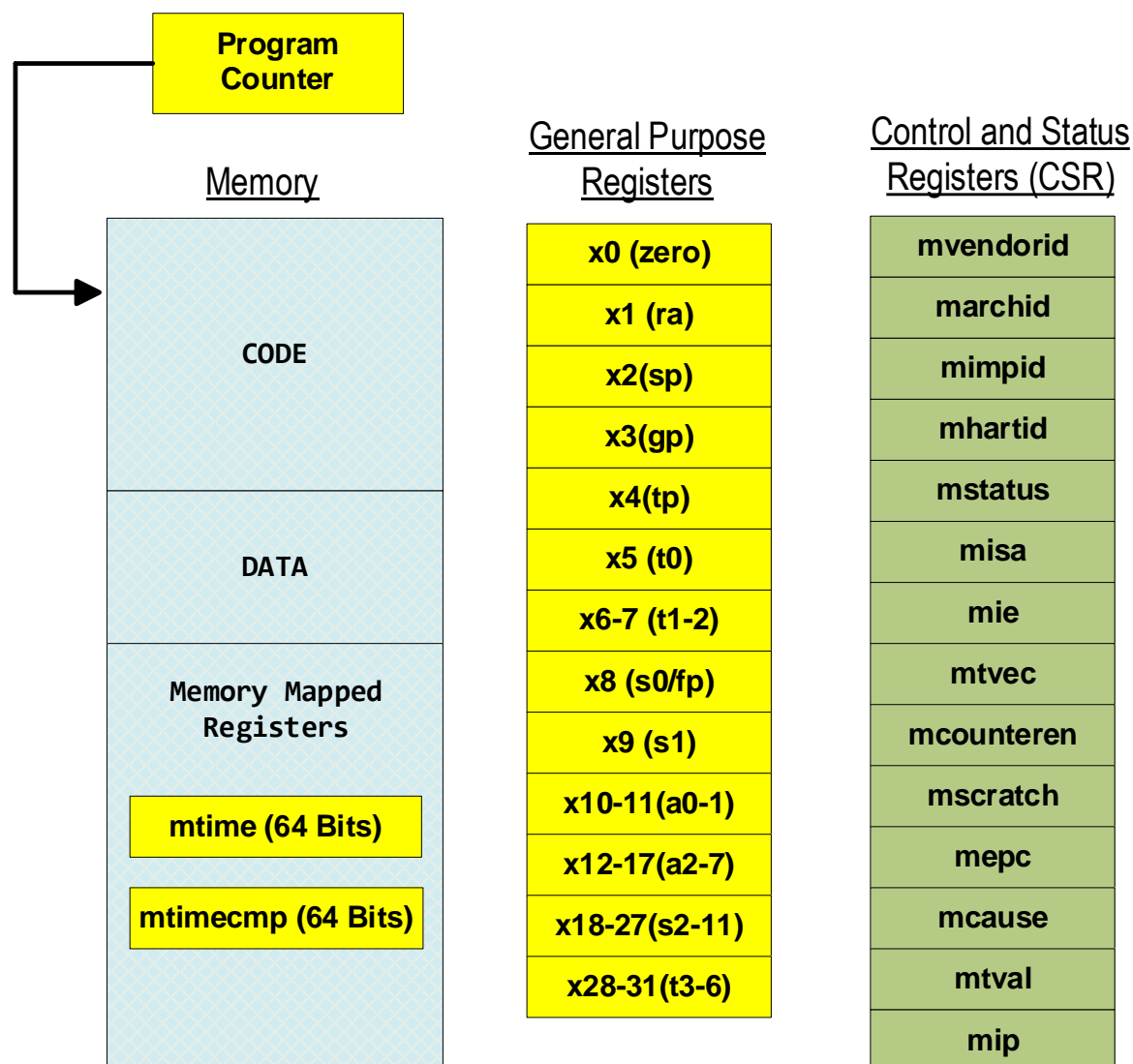


Figure 3-1 Address Spaces

3.2 GPR (General Purpose Register) and Calling Convention

As mentioned in Section 2.4, the GRV3000D has 32 General Purpose Registers. Each is 32-bit wide. They are marked as x0-x31. Among them, x0 (register zero) is a read-only register, and it always returns zero.

One of the design goals for RISC-V ISA is to provide support for high level programming languages, such as C or C++, and to keep compatibility at ABI level. Thus, RISC-V has also standardized the calling convention. For assembly language, the names of the 32 registers are given new names to reflect their functions in the calling convention, as shown in Table 3-1.

Register Name	Assembly Name	Description	Value Unchanged after return from the function call
x0	<i>zero</i>	<i>Read Only, always zero</i>	<i>Undefined</i>
x1	<i>ra</i>	<i>Return Address</i>	<i>No</i>
x2	<i>sp</i>	<i>Stack Pointer</i>	<i>Yes</i>
x3	<i>gp</i>	<i>Global Pointer</i>	<i>Undefined</i>
x4	<i>tp</i>	<i>Thread Pointer</i>	<i>Undefined</i>
x5	<i>t0</i>	<i>temp register, or be used as alternate link register (discussed in a later part of this document)</i>	<i>No</i>
x6~x7	<i>t1~t2</i>	<i>Temp Registers</i>	<i>No</i>
x8	<i>s0/fp</i>	<i>Saved by Callee Function, or be used as stack frame pointer</i>	<i>Yes</i>
x9	<i>s1</i>	<i>Saved by Callee Function</i>	<i>Yes</i>
x10~x11	<i>a0~a1</i>	<i>Function Parameters or Function Return Value</i>	<i>No</i>
x12~x17	<i>a2~a7</i>	<i>Function Parameters</i>	<i>No</i>
x18~x27	<i>s2~s11</i>	<i>Saved by Called Function</i>	<i>Yes</i>

Register Name	Assembly Name	Description	Value Unchanged after return from the function call
x28~x31	t3~t6	Temp Register	No

Table 3-1 RISC-V Calling Convention

Please note that registers a0 – a7 (x10 – x11) are used for function parameters. And those registers will find their places in DSP acceleration, as demonstrated in later chapters of this document.

3.3 CSR (Control and Status Register) and Machine Mode

3.3.1 Privilege Level and Machine Mode

The RISC-V Instruction Set Manual Volume II (Ref [2]) has defined 3 working mode, as Machine Mode Supervisor Mode and User Mode. And each mode is mapped to a particular privilege level, as illustrated in Figure 3-2.

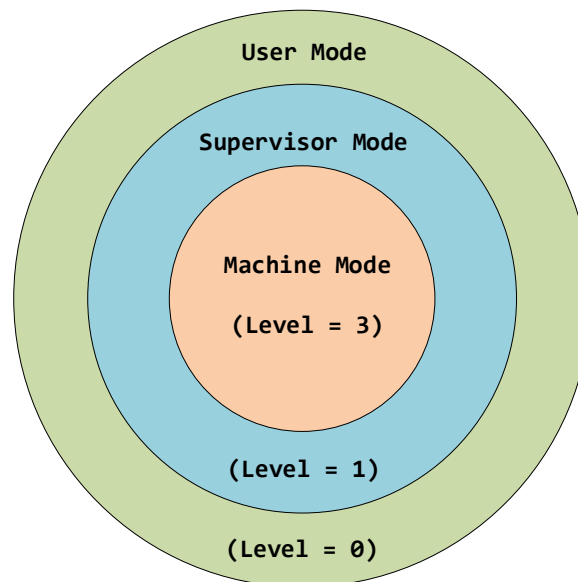


Figure 3-2 RISC-V Privilege Levels

And the Machine Mode is mandatory in Ref [2], while the Supervisor Mode and User Mode are optional. For the GRV3000D Processor, only the Machine Mode is supported.

3.3.2 Control and Status Register (CSR)

As mentioned early, the RISC-V Instruction Set Manual Volume II (Ref [2]) has defined a separate address space for Control and Status Registers (CSR).

PulseRain GRV3000D – Programmer's Guide

And 12 bits of address are used for CSR indexing. Among the 12 bits, [11 : 10] are used for read/write permission. If they are both high (2'b11), it means the correspondent CSR is a read-only register. Otherwise, the CSR can be written to.

Address [9 : 8] correspond to the privilege level associated with this CSR. As the GRV3000D processor only supports Level 3 (Machine Mode), the rest of this document will only discuss those CSRs listed in Table 3-2.

Address	Read / Write Permission	Privilege Level	Register Name	Description
0xF11	Read Only	3	<i>mvendorid</i>	Vendor ID
0xF12	Read Only	3	<i>marchid</i>	Architecture ID
0xF13	Read Only	3	<i>mimpid</i>	Implementation ID
0xF14	Read Only	3	<i>mhartid</i>	HART ID
0x300	R/W	3	<i>mstatus</i>	Machine Status
0x301	R/W	3	<i>misa</i>	ISA and Extensions Supported
0x304	R/W	3	<i>mie</i>	Interrupt Enable
0x305	R/W	3	<i>mtvec</i>	Base address for Exception Vectors
0x340	R/W	3	<i>mscratch</i>	Scratch Register
0x341	R/W	3	<i>mepc</i>	The PC value when exception happens
0x342	R/W	3	<i>mcause</i>	The cause of the Exception
0x343	R/W	3	<i>mtval</i>	Machine Trap Value Register
0x344	R/W	3	<i>mip</i>	Interrupt Pending
0xB00	R/W	3	<i>mcycle</i>	Number of Clock Cycles (lower 32 bits)
0xB02	R/W	3	<i>minstret</i>	Number of Instruction Retired (lower 32 bits)
0xB80	R/W	3	<i>mcycleh</i>	Number of Clock Cycles (higher 32 bits)
0xB82	R/W	3	<i>minstreth</i>	Number of Instruction Retired (higher 32 bits)

Table 3-2 CSR Registers Supported by PulseRain GRV3000D Processor

PulseRain GRV3000D – Programmer's Guide

3.3.2.1 [mvendorid \(Vendor ID\)](#)

In order to distinguish between vendors, this CSR is defined by RISC-V ISA to save the vendor ID. In fact, it is a 32-bit read-only value derived from JEDEC manufacturer ID. For all the RISC-V processors designed by PulseRain Technology, this register will always be set as 32'h0000055E.

3.3.2.2 [marchid \(Architecture ID\)](#)

For PulseRain GRV3000D Processor, this CSR is read-only with the value as 32'h80C00002.

3.3.2.3 [mimpid \(Implementation ID\)](#)

This is a 32-bit read-only CSR, defined in Table 3-3.

Bit Index	Comments
[31 : 24]	<i>always 8'h0A</i>
[23 : 8]	<i>Determined at CPU level, Platform Specific</i>
[7 : 0]	<i>For GRV3000D, it is 8'hD8.</i>

Table 3-3 Bit Definition for mimpid

3.3.2.4 [mstatus](#)

This CSR is responsible for recording the processor status. Its full definition can be found in Ref [2]. However, for the GRV3000D processor, only those bits shown in Table 3-4 are implemented.

Bit Index	Comments
[7]	<i>MPIE (Machine Previous Interrupt Enable). When Interrupt or Exception happens. this bit will record the current MIE value. When return through the mret instruction, this value will be copied back to MIE.</i>
[3]	<i>MIE (Machine Interrupt Enable). When this bit is set to zero, all the external interrupt and timer interrupt will be disabled.</i>

Table 3-4 Bit Definition for mstatus

3.3.2.5 [mscratch](#)

This CSR can be used as 32-bit general purpose storage.

3.3.2.6 [CSRs related to Interrupt / Exception](#)

The following CSRs are for Interrupt and Exception handling, and they will be discussed with more details in a later part of this document.

- mtvec (Machine Trap Vector Base-Address Register)
- mip (Machine Interrupt Register, Pending Interrupt)
- mie (Machine Interrupt Register, Interrupt Enable)
- mcause (Machine Cause Register)
- mepc (Machine Exception Program Counter)
- mtval (Machine Trap Value Register)

3.3.2.7 Counters

As a way to monitor the hardware performance, RISC-V ISA has defined a few counters to keep track of the time span starting from reset. Those counters are:

- mcycle and mcycleh
RISC-V ISA has defined a 64-bit cycle register to keep track of the number of clock cycles passed since reset. The lower 32 bits and the higher 32 bits of this register are stored in CSR mcycle and CSR mcycleh respectively. As the GRV3000D is a 32-bit processor, the 64-bit value cannot be read out atomically. Thus, the software has to follow certain rules in order to get this 64-bit value correctly. For the GRV3000D processor, the software can achieve this through one of the following two ways:
 1. read the mcycle first, immediately followed by a read on mcycleh. Make sure there are no interrupts between those two reads. The processor will automatically save the higher 32 bits of cycle register into mcycleh when mcycle is being read.
 2. Another way to this can be found in Ref [2]. As shown below, the software can use a loop to read between mcycleh and mcycle, and exit the loop if the two mcycleh read return the same value.

```
again:
    rdcycleh x3
    rdcycle  x2
    rdcycleh x4
    bne x3, x4, again
```

- minstret and minstreth
RISC-V ISA has also defined another 64-bit register to keep track of the number of instructions retired since reset. Its lower 32-bit and higher 32-bit are stored in CSR minstret and CSR minstreth respectively.

Like CSR mcycle and CSR mcycleh, this 64-bit register can not be read out atomically in a single read. And the same software approach mentioned previously for mcycle/mcycleh can also be used here to read minstret/minstreth.

3.4 Memory Space

The GRV3000D has a unified 32-bit memory address space for I-TCM, D-TCM and Memory Mapped Registers. They are allocated according to Table 3-5.

Address	Description
0x00000000 – 0x1FFFFFFF	Reserved
0x20000000 – 0x3FFFFFFF	Memory Mapped Register
0x40000000 – 0x7FFFFFFF	Reserved
0x80000000 – 0xBFFFFFFF	I-TCM
0xC0000000 – 0xFFFFFFFF	D-TCM

Table 3-5 Address Allocation for Memory Space

3.4.1 NOP Region in I-TCM

If the GRV3000D is configured with N 32-bit words of physical I-TCM, the GRV3000D will (conceptually) append another N words to the physical I-TCM (Namely the virtual I-TCM in Figure 3-3). Those N words appended will all be 0x00000013, which is the NOP instruction in RV32I ISA.

The purpose of appending NOP region to the physical I-TCM is to provide a default location for the branch prediction without introducing any side effect.

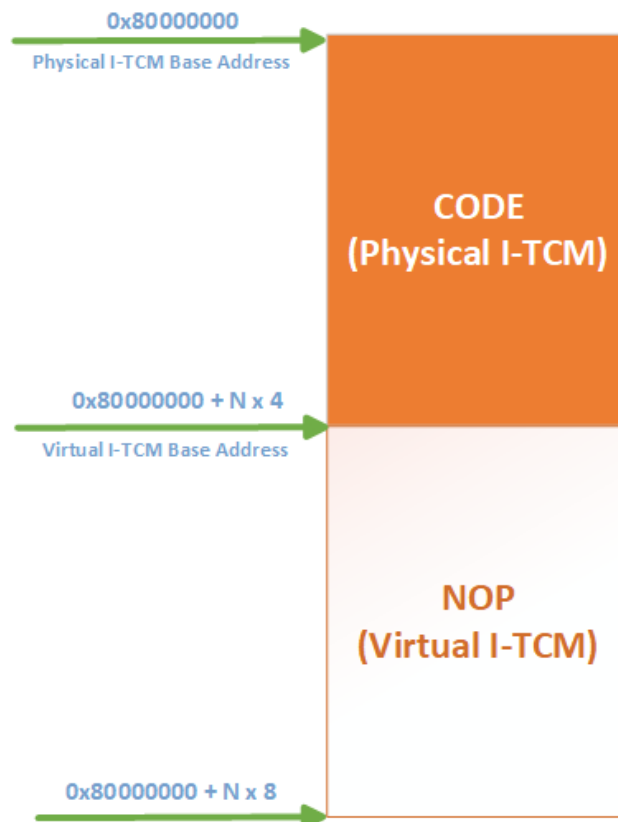


Figure 3-3 NOP Region after Physical I-TCM

PulseRain GRV3000D – Programmer's Guide

3.4.2 Top Half and Bottom Half in D-TCM

If the GRV3000D is configured with N 32-bit words of D-TCM, it will be physically divided into two equal parts (the top half and the bottom half), as illustrated in Figure 3-4.

The purpose of dividing the D-TCM into two equal parts is to provide more parallelism for D-TCM, without introducing true dual-port memory. As will be demonstrated later, such extra parallelism will be useful to the DSP acceleration.

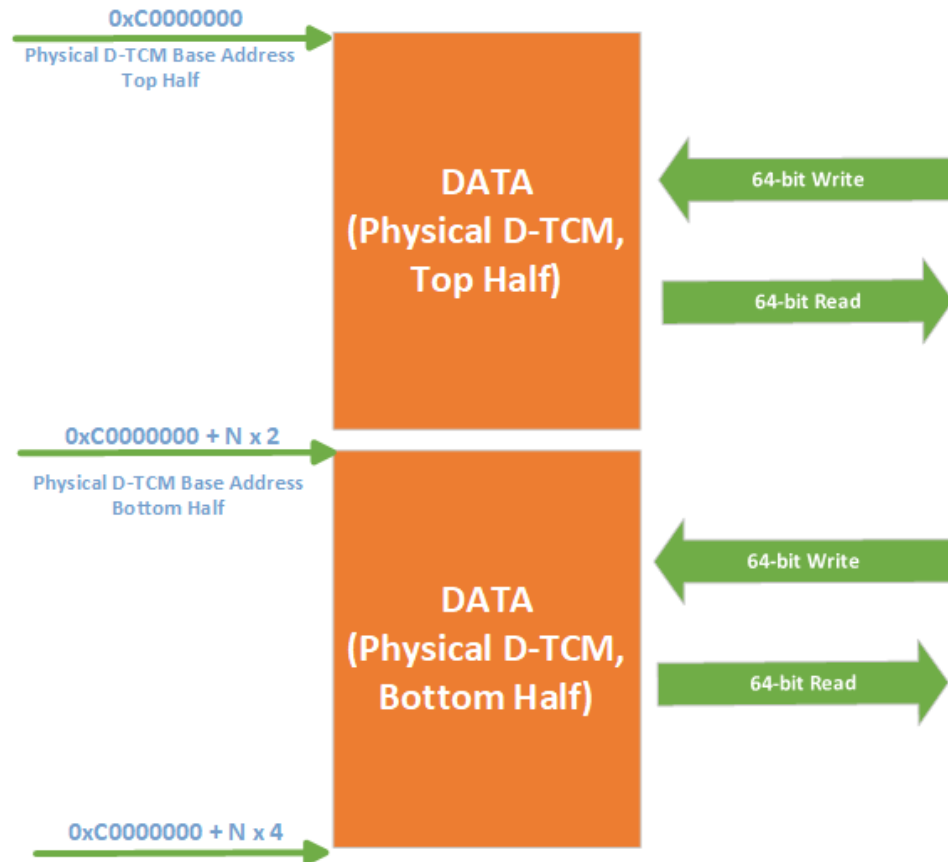


Figure 3-4 D-TCM Top Half and Bottom Half

3.4.3 AXI4-Lite Switch and Memory Mapped Registers

As shown in Figure 2-4, the AXI4-Lite Switch will have slave ports for peripherals. Each slave port will be allocated a range of addresses for the peripheral registers. By default, slave port 0 will be allocated to the System Timer, and slave port 1 will be allocated to UART / GPIO. Their address ranges are as following:

Address	Description
0x20000000 – 0x2000003F	System Timer (Machine Timer)
0x20000040 – 0x2000007F	UART and GPIO

Table 3-6 Address Allocation for Peripherals

PulseRain GRV3000D – Programmer's Guide

3.4.3.1 System Timer

To prepare for RTC (Real Time Clock) implementation, the RISC-V ISA has defined two 64-bit registers for this purpose. They are called `mtime` and `mtimecmp`. And different from CSR, those two registers are memory-mapped into the memory space, as illustrated in Figure 3-1.

- `mtime`
On the GRV3000D processor, `mtime` is a 64-bit timer counter runs at 1MHz resolution. And its lower 32-bit part and higher 32-bit part are mapped to the addresses shown in Table 3-7. Like other 64-bit read, the software has to make sure it can get the value correctly through multiple reads. And the solution is already mentioned in Section 3.3.2.7.
- `mtimecmp`
As its name suggests, the purpose of `mtimecmp` is to compare its value against that of `mtime`'s. When the `mtime` is no less than `mtimecmp`, a timer interrupt will be generated. And its lower 32-bit part and higher 32-bit part are mapped to the addresses shown in Table 3-7.

As `mtimecmp` is a 64-bit register, it takes at least two write instructions to update its entire value. And partial update of `mtimecmp` might cause spurious timer interrupt. And in this regard, there are mainly two ways to deal with it:

1. Disable timer interrupt before writing to `mtimecmp`. And the timer interrupt can be re-enabled after the writing is completed.
2. Use the code sequence suggested in Ref [2], as shown below:

```
li t0, -1          # Write 0xFFFFFFFF into t0
sw t0, mtimecmp    # Set mtimecmp's lower 32 bits as 0xFFFFFFFF
sw a1, mtimecmp + 4 # Set mtimecmp's higher 32 bits
sw a0, mtimecmp    # Set mtimecmp's lower 32 bits
```

Please note that the assembly code above has to be executed exactly in order. Compiler optimization, the intervention of ISR etc. may affect the correctness of the above code.

Address	Name	Description
0x20000000	MTIME_LOW	lower 32 bits of <code>mtime</code>
0x20000004	MTIME_HIGH	higher 32 bits of <code>mtime</code>
0x20000008	MTIMECMP_LOW	lower 32 bits of <code>mtimecmp</code>
0x2000000C	MTIMECMP_HIGH	higher 32 bits of <code>mtimecmp</code>

Table 3-7 Memory Mapped Address for `mtime` and `mtimecmp`

3.4.3.2 UART

By default, the GRV3000D will carry a UART with a fixed baud rate of 115200 bps. Its transmitter can be controlled and monitored through the register `UART_TX`. And its receiver can be controlled and monitored

PulseRain GRV3000D – Programmer's Guide

through the register UART_RX. The Memory Mapped addresses for those two registers are shown below in Table 3-8:

Address	Register Name	Description
0x20000010	UART_TX	TX control / status register for UART
0x20000014	UART_RX	RX control / status register for UART

Table 3-8 Peripheral Address for UART

And the bit definition of UART_TX can be found in Table 3-9:

Bits Index	Name	R/W	Default	Description
7 : 0	TX Data	RW	0	The byte to be transmitted. Always zero when being read.
29 : 8	Reserved	N/A	0	Always zero
30	tx_sync_reset	W1P	0	Write '1' to Reset UART TX
31	tx_active	RO	0	When this bit is high, the transmitter is busy

Table 3-9 UART_TX Bit Definition

The bit definition of UART_RX can be found in Table 3-10:

Bits Index	Name	R/W	Default	Description
7 : 0	RX Data	RO	0	The byte received.
27 : 8	Reserved	N/A	0	Always zero
28	rx_sync_reset	W1P	0	Write '1' to Reset UART RX
29	uart_rx_fifo_not_empty	RO	0	This bit will be set to high if there is data in the receiving FIFO.
30	uart_rx_fifo_full	RO	0	This bit will be set to high if the receiving FIFO is full.
31	Uart_rx_fifo_read	RW	0	RX FIFO Read Request

Table 3-10 UART_RX Bit Definition

And an Arduino Package has been offered to assist software developers programming the UART. The detail will be mentioned in later chapters of this document.

3.4.3.3 GPIO

The GRV3000D processor can support 16 Input Pins and 16 Output Pins as GPIO. And they are controlled and monitored through the peripheral register GPIO, whose address is defined in Table 3-11. And its bit definition can be found in

Address	Register Name	Description
0x20000018	GPIO	control / status register for general purpose In / Out

Table 3-11 GPIO Address

Bits Index	Name	R/W	Default	Description
15 : 0	GPIO Out	RW	0	Output pin
31 : 16	GPIO In	RO	0	Input Pin

Table 3-12 GPIO Bit Definition

When the GPIO register is being written to, the correspondent output pin will change accordingly. When the GPIO register is being read, the logic level for the correspondent input pin will be extracted.

3.4.3.4 External Interrupt

The GRV3000D also has peripheral registers to control and monitor the external interrupt. Their addresses are defined in Table 3-13.

Address	Register Name	Description
0x2000001C	INT_SOURCE	status register for external interrupt
0x20000020	INT_ENABLE	control register for external interrupt

Table 3-13 Peripheral Address for External Interrupt

And the details of those registers will be discussed in the next Section.

3.4.4 Trap (Exception and Interrupt)

Exception and interrupt are the mechanism for processors to handle asynchronous events. The difference between them is that:

1. The exception is caused by the software execution itself. The usual cases are:
 - Accessing a non-exist CSR
 - Unaligned access to memory
 - Break Point or System Call from Operating Systems

However, please note that divided-by-zero will not cause exception on RISC-V processors.

2. The interrupt is caused by events independent of software execution. On PulseRain FRV2000 processor, the interrupt can only come from two sources, as shown in :
 - a) Timer Interrupt
 - b) External Interrupt, mainly caused by peripherals

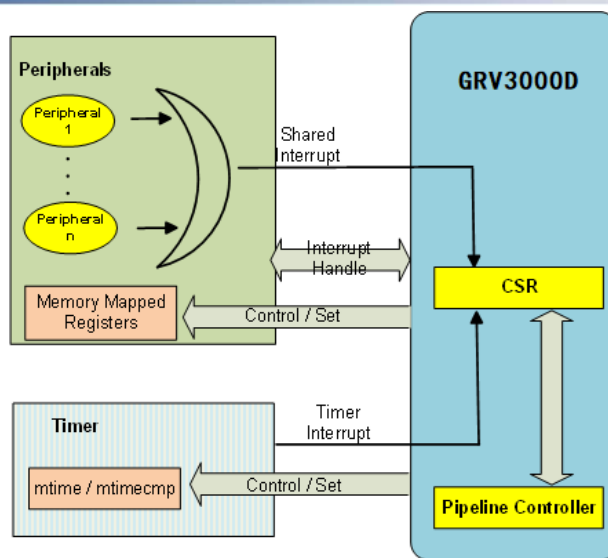


Figure 3-5 Interrupt for PulseRain GRV3000D Processor

3.4.4.1 Trap Trigger and Handle

As the exception and interrupt are very similar, they are handled in the same way on the GRV3000D Processor. And they can both be called trap. When trap happens, the processor will go through the following procedures:

1. If the trap is an interrupt, the following two CSRs will be checked to see if the interrupt is masked
 - a) The MIE bit in mstatus (Table 3-4).
 - b) The correspondent bits in mie (Machine Interrupt Register). And the bit definition for mie can be found in Table 3-14.

Bit Index	Comments
7	<i>Machine Mode Timer Interrupt Enable(MTIE)</i>
11	<i>Machine Mode External Interrupt Enable (MEIE)</i>
[31 : 12]	<i>For future expansion</i>

Table 3-14 Bit Definition for MIE

2. Identify the cause of the trap.

When trap happens, the processor needs to put the cause of the trap into CSR mcause. The MSB (bit 31) of mcause will identify whether this trap is interrupt (high) or exception (low). The resting bits of mcause are used as Exception Code. At this point, only the lower 4 bits of them are being used. If the trap is an interrupt, its correspondent Exception Code (The interrupt source) is shown in Table 3-15.

Exception Code	Interrupt Source
7	<i>Machine Mode Timer Interrupt</i>
11	<i>Machine Mode External Interrupt</i>

Table 3-15 Interrupt Source

And if the trap is an exception, the correspondent exception category is shown Table 3-16.

Exception Code	Exception Category
0	<i>Unaligned Instruction Memory Access</i>
1	<i>Instruction Fetch Failure</i>
2	<i>Illegal Instruction</i>
3	<i>Break Point</i>
4	<i>Unaligned Data Memory Load Access</i>
5	<i>Memory Load Failure</i>
6	<i>Unaligned Data Memory Store Access</i>
7	<i>Memory Store Failure</i>
11	<i>Environment Call for Machine Mode</i>

Table 3-16 Exception Category

3. Identify the instruction address of the trap

For the machine mode, RISC-V ISA has defined a CSR called mepc (Machine Exception Program Counter). For exception, the mepc will have the address of the instruction that causes the exception. For interrupt, the value in the mepc will be used by mret instruction to return from ISR.

4. Provide additional parameters for the trap

To help handle the trap, RISC-V ISA has defined a CSR called mtval (Machine Trap Value Register). to provide additional parameters for the trap. When the exception is caused by memory access, the load/store address will be saved in this CSR.

PulseRain GRV3000D – Programmer's Guide

5. Load the PC with trap handler

For machine mode, RISC-V ISA has defined a CSR called mtevec (Machine Trap Vector Base-Address Register). This CSR will be used to determine the address of the trap handler. The lower two bits of this CSR are used for trap mode, while the higher 30 bits are used as base address. And the trap mode is defined in Table 3-17.

Trap Mode	Description
0	<i>Direct Mode. In this mode, PC = mtevc</i>
1	<i>Vector Mode. In this Mode</i> <i>PC = BASE + 4 * Exception_Code.</i> <i>Exception Code comes from mcause, which can be found in Table 3-15 and Table 3-16</i>
2, 3	<i>Reserved</i>

Table 3-17 Trap Mode

6. Setup CSR mip for interrupts

For interrupts, the correspondent bits in CSR mip should be set to indicate the pending status. And these bits are defined in Table 3-18:

Bit Index	Comments
7	<i>Machine Mode Timer Interrupt Pending (MTIP)</i>
11	<i>Machine Mode External Interrupt Pending (MEIP)</i>

Table 3-18 Bit Definition for MIP

3.4.4.2 Trap Return

As mentioned early, in machine mode, when the trap handler is completed, it needs to use the mret instruction to return. And the Program Counter will be loaded with the value in CSR mepc.

3.4.4.3 WFI

As a way to provide support for Operating System scheduling, RISC-V Instruction Manual Volume II (Ref [2]) has defined an instruction for interrupt waiting, called WFI (Wait for Interrupt). When the processor encounters WFI instruction, the pipeline will enter stall state, until an interrupt happens.

PulseRain GRV3000D – Programmer's Guide

3.4.4.4 ECALL and EBREAK

To provide support for Operating System and Debugger, RISC-V ISA has also defined ECALL (Environment Call) and EBREAK (Breakpoint) instructions. Both of those instructions will cause exception, and their correspondent exception code can be found in Table 3-16.

One thing to be noted is that when the processor encounters ECALL/EBREAK, it will fill mepc with the address of current instruction, instead of the one for the next instruction.

3.4.4.5 External Interrupt

As shown in Figure 3-5, all external interrupts will be wired-or together as one shared interrupt to the GRV3000D processor core. The external interrupts can come from the peripherals, or come from the INTx pins on the platform port list. All interrupts are level triggered.

The control and status of the external interrupt are through the following peripheral registers:

➤ **Interrupt Source**

The external interrupt status can be found out from the register INT_SOURCE (Table 3-19), whose address is defined in Table 3-13.

Bits Index	Name	R/W	Default	Description
0	Reserved	N/A	0	Always zero
1	uart_rx_fifo_not_empty	Read Only	0	Interrupt when UART RX FIFO is not empty
29 : 2	Reserved	N/A	0	Always zero
31 : 30	INT1 and INT0	Read Only	0	Status of INTx pin on the port list

Table 3-19 INT_SOURCE Bit Definition

➤ **Interrupt Enable / Disable**

The interrupt sources can be enabled/disabled individually by a register called INT_ENABLE, whose address is defined in Table 3-13. Its bit definition will match what is defined in INT_SOURCE (Table 3-19). A logic high in the correspondent bit will enable the interrupt, and a logic low will disable it.

3.5 JTAG Debug Module and HW Based Bootloader

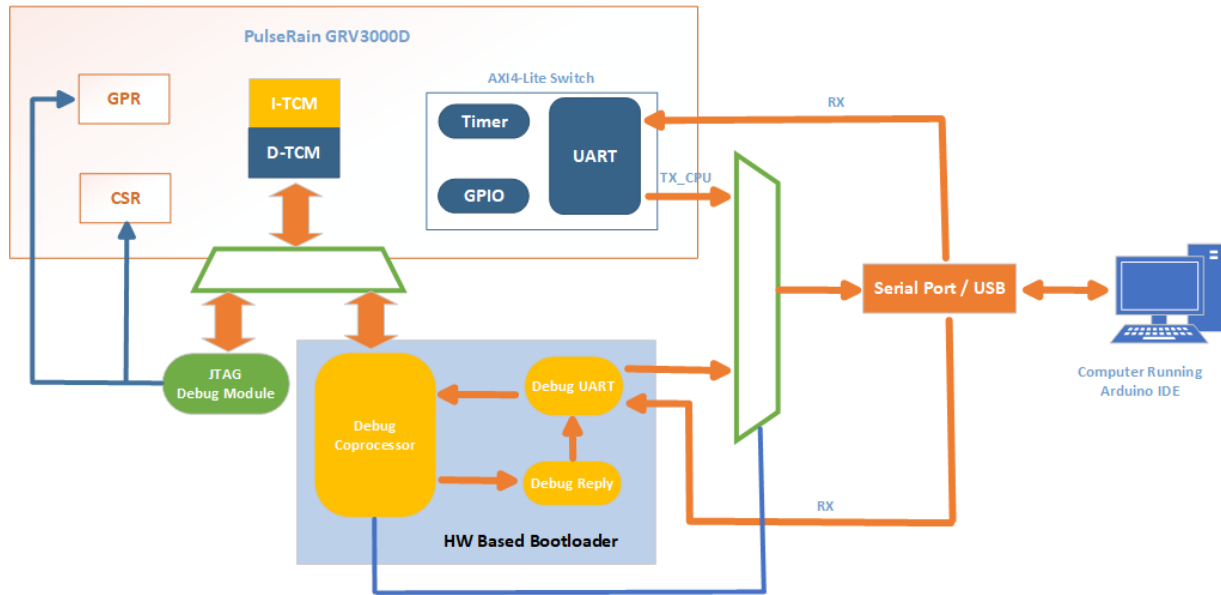


Figure 3-6 JTAG Debug Module and HW Base Bootloader

PulseRain Technology also provides additional IPs, like the JTAG Debug Module and the HW Based Bootloader to make software developers' job easier.

As illustrated in Figure 3-6, software developers can access the full information of the GRV3000D processor through JTAG Debug Module. The JTAG Debug Module works with external JTAG probes (such as Segger J-Link) to support software load, break point, single step etc.

And software developers can do their job through the Arduino IDE, for which the HW Based Bootloader will come to aid. The HW Based Bootloader will share the UART TX pin with the main UART inside the GRV3000D. For bare-metal code and quick bring up, the Arduino IDE is a good alternative to the full-blown JTAG probe.

The details of the debugging process will be discussed in later chapters of this document.

4 Software – General Flow

The GRV3000D is compatible with the RISC-V instruction set mentioned early in Section 1.2, so the software developers can use any compiler or assembler as they would like to choose. However, as the GCC has become the de-facto compiler for embedded systems, it will be the focus of this document.

And this chapter will quick go through the machine instructions supported by the GRV3000D, followed by a discussion on Makefile and Assembly for GNU. Arduino and JTAG Debugger will also be discussed in this chapter.

4.1 Instruction Supported

4.1.1 Base Instruction

The GRV3000D Processor supports the RISC-V RV32I base instruction set, as listed in Table 4-1, Table 4-2, Table 4-3, Table 4-4, Table 4-5 and Table 4-6.

Mnemonics	Description	Notes
ADDI rd, rs1, imm	$rd = rs1 + imm$	add immediate number
SLTI rd, rs1, imm	$rd = (rs1 < imm) ? 1 : 0$	signed compare with immediate number
SLTIU rd, rs1, imm	$rd = (rs1 < imm) ? 1 : 0$	unsigned compare with immediate number
SEQZ rd, rs	pseudo instruction for SLTI rd, rs1, 1	
ANDI rd, rs1, imm	$rd = rs1 \& imm$	logic and with immediate number
ORI rd, rs1, imm	$rd = rs1 imm$	logic or with immediate number
XORI rd, rs1, imm	$rd = rs1 \wedge imm$	exclusive-or with immediate number
SLLI rd, rs1, imm	$rd = rs1 \ll imm[4:0]$	shift left with immediate number
SRLI rd, rs1, imm	$rd = rs1 \gg imm[4:0]$	logic shift right with immediate number
SRAI rd, rs1, imm	$rd = rs1 \ggg imm[4:0]$	arithmetic shift right with immediate number
LUI rd, imm[31 : 12]	$rd = \{imm[31 : 12], 12'h000\}$	load upper immediate
AUIPC rd, imm[31 : 12]	$rd = PC + \{imm[31 : 12], 12'h000\}$	add upper immediate to PC
NOP	pseudo instruction for ADDI 0, 0, 0	no operation

Table 4-1 Integer Register – Immediate Instructions

Mnemonics	Description	Notes
ADD rd, rs1, rs2	$rd = rs1 + rs2$	add register to register
SUB rd, rs1, rs2	$rd = rs1 - rs2$	subtract register from register
SLT rd, rs1, rs2	$rd = (rs1 < rs2) ? 1 : 0$	signed compare with registers
SLTU rd, rs1, rs2	$rd = (rs1 < rs2) ? 1 : 0$	unsigned compare with registers
AND rd, rs1, rs2	$rd = rs1 \& rs2$	logic and with registers
OR rd, rs1, rs2	$rd = rs1 rs2$	logic or with registers
XOR rd, rs1, rs2	$rd = rs1 \wedge rs2$	exclusive-or with registers
SLL rd, rs1, rs2	$rd = rs1 \ll rs2[4:0]$	shift left with registers
SRL rd, rs1, rs2	$rd = rs1 \gg rs2[4:0]$	logic shift right with registers
SRA rd, rs1, rs2	$rd = rs1 \ggg rs2[4:0]$	arithmetic shift right with registers

Table 4-2 Integer Register – Register Instructions

Mnemonics	Description	Notes
JAL rd, imm[20 : 1]	rd = PC + 4; PC = PC + {11{imm[20]}, imm[20 : 1], 1'b0}	jump and link
JALR rd, rs1, imm[11 : 0]	rd = PC + 4; PC = (rs1 + {20{imm[11]}, imm[11 : 0]}) & 0xFFFFFFF	jump and link register

Table 4-3 Unconditional Jump Instructions

Mnemonics	Description	Notes
BEQ rs1, rs2, imm[12 : 1]	PC = (rs1 == rs2) ? PC + {19{imm[12]}, imm[12:1], 1'b0} : PC + 4	conditional branch, jump if equal
BNQ rs1, rs2, imm[12 : 1]	PC = (rs1 != rs2) ? PC + {19{imm[12]}, imm[12:1], 1'b0} : PC + 4	conditional branch, jump if unequal
BLT rs1, rs2, imm[12 : 1]	PC = (rs1 < rs2) ? PC + {19{imm[12]}, imm[12:1], 1'b0} : PC + 4	conditional branch, jump if less than (signed compare)
BGE rs1, rs2, imm[12 : 1]	PC = (rs1 >= rs2) ? PC + {19{imm[12]}, imm[12:1], 1'b0} : PC + 4	conditional branch, jump if greater or equal (signed compare)
BLTU rs1, rs2, imm[12 : 1]	PC = (rs1 < rs2) ? PC + {19{imm[12]}, imm[12:1], 1'b0} : PC + 4	conditional branch, jump if less than (unsigned compare)
BGEU rs1, rs2, imm[12 : 1]	PC = (rs1 >= rs2) ? PC + {19{imm[12]}, imm[12:1], 1'b0} : PC + 4	conditional branch, jump if greater or equal (unsigned compare)

Table 4-4 Conditional Branch Instructions

Mnemonics	Description	Notes
LB rd, rs1, imm[11 : 0]	rd [7 : 0] = (int8_t)*(rs1 + {20{imm[11]}, imm[11 : 0]}) rd [31 : 8] = sign extension	load signed byte
LBU rd, rs1, imm[11 : 0]	rd [7 : 0] = (int8_t)*(rs1 + {20{imm[11]}, imm[11 : 0]}) rd [31 : 8] = 0	load unsigned byte
LH rd, rs1, imm[11 : 0]	rd [15 : 0] = (int16_t)*(rs1 + {20{imm[11]}, imm[11 : 0]}) rd [31 : 16] = sign extension	load signed half-word
LHU rd, rs1, imm[11 : 0]	rd [15 : 0] = (int16_t)*(rs1 + {20{imm[11]}, imm[11 : 0]}) rd [31 : 16] = 0	load unsigned half-word
LW rd, rs1, imm[11 : 0]	rd [31 : 0] = (int32_t)*(rs1 + {20{imm[11]}, imm[11 : 0]})	load word

Table 4-5 Load Instructions

Mnemonics	Description	Notes
SB rs1, rs2, imm[11 : 0]	(int8_t)*(rs1 + {20{imm[11]}, imm[11 : 0]}) = rs2 & 0xFF	store byte
SH rs1, rs2, imm[11 : 0]	(int16_t)*(rs1 + {20{imm[11]}, imm[11 : 0]}) = rs2 & 0xFFFF	store half word
SW rs1, rs2, imm[11 : 0]	(int32_t)*(rs1 + {20{imm[11]}, imm[11 : 0]}) = rs2	store word

Table 4-6 Store Instructions

PulseRain GRV3000D – Programmer's Guide

Mnemonics	Description	Notes
FENCE / FENCE.I	Flush the pipeline	

Table 4-7 FENCE/FENCE.I Instructions

4.1.2 CSR Instruction

For CSR instructions, the CSR register address can be found in bit [31 : 20]. And RISC-V has defined 6 CSR access instructions, which can be found in Table 4-8.

Mnemonics	Description	Notes
CSRRW rd, rs1, CSR	(CSR)→(rd): (rs1)→(CSR)	read / write CSR
CSRRS rd, rs1, CSR	(CSR)→(rd): (rs1) (CSR) →(CSR)	read and set bits in CSR
CSRRC rd, rs1, CSR	(CSR)→(rd): (rs1) ~(CSR) →(CSR)	read and clear bits in CSR
CSRRWI rd, CSR, imm	(CSR)→(rd): immediate →(CSR)	read / write CSR with immediate number
CSRRSI rd, CSR, imm	(CSR)→(rd): (CSR) immediate →(CSR)	read and set bits in CSR with immediate number
CSRRCI rd, CSR, imm	(CSR)→(rd): ~(CSR) & immediate →(CSR)	read and clear bits in CSR with immediate number

Table 4-8 CSR Instructions

4.1.3 ECALL / EBREAK Instruction

Mnemonics	Description	Notes
ECALL	Please see Section 3.4.4.4	
EBREAK	Please see Section 3.4.4.4	

Table 4-9 ECALL/EBREAK Instructions

4.1.4 M Extension

Mnemonics	Description	Notes
MUL rd, rs1, rs2	Signed Multiplication, keep lower 32 bits to rd	
MULH rd, rs1, rs2	Signed Multiplication, keep higher 32 bits to rd	
MULHSU rd, rs1, rs2	Signed multiplies unsigned number, keep higher 32 bits to rd	
MULHU rd, rs1, rs2	Unsigned multiplication, keep higher 32 bits to rd	
DIV rd, rs1, rs2	Signed division, save quotient to rd	
DIVU rd, rs1, rs2	Unsigned division, save quotient to rd	
REM rd, rs1, rs2	Signed division, save remainder to rd	
REMU rd, rs1, rs2	Unsigned division, save remainder to rd	

Table 4-10 MUL / DIV

Please note that those instructions in Table 4-10 will never trigger exceptions. And the following are two special cases that could happen during the division:

1. Divided by Zero
In this case, the quotient is 32'hFFFFFFF, and the remainder will be the same as rs2.
2. Overflow during signed division
This could happen when rs1 = 32'80000000 and rs2 = 32'hFFFFFFF. In this case, the quotient will be 32'h80000000, and the remainder will be zero.

PulseRain GRV3000D – Programmer's Guide

4.1.5 C Extension (Optional)

The C extension (16-bit Compressed Instructions) is not an independent instruction set. Each instruction in the C extension can be mapped to a 32-bit instruction from Section 4.1.1. For PulseRain GRV3000D, the C Extension is optional. If C Extension is supported, the GRV3000D's instruction fetch stage will translate the 16-bit instructions into their 32-bit counterparts, and keep the other pipeline stages consistent and unchanged for 32-bit processing only.

And to compress the instructions into 16-bit format, the C extension has introduced 3-bit register index rd' , $rs1'$ and $rs2'$ that can only be mapped to register x8-x15.

The instructions from C extension are listed in Table 4-11, Table 4-12, Table 4-13, Table 4-14, Table 4-15, Table 4-16, Table 4-17 and Table 4-18.

Mnemonics	32-Bit Counterpart	Notes
C.LWSP	lw rd, offset[7 : 2](x2)	Stack Pointer Load
C.LW	lw rd', offset[6 : 2](rs1')	

Table 4-11 16-bit Load

Mnemonics	32-Bit Counterpart	Notes
C.SWSP	sw rs2, offset[7 : 2](x2)	Stack Pointer Store
C.SW	sw rs2', offset[6 : 2](rs1')	

Table 4-12 16-bit Store

Mnemonics	32-Bit Counterpart	Notes
C.J	jal x0, offset[11:1]	
C.JAL	jal x1, offset[11:1]	
C.JR	jalr x0, rs1, 0	
C.JALR	jalr x1, rs1, 0	

Table 4-13 16-bit Unconditional Jump

Mnemonics	32-Bit Counterpart	Notes
C.BEQZ	beq rs1', x0, offset[8:1]	
C.BNEZ	bne rs1', x0, offset[8:1]	

Table 4-14 16-bit Conditional Branch

Mnemonics	32-Bit Counterpart	Notes
C.LI	addi rd, x0, imm[5:0] (rd \neq 0)	
C.LUI	lui rd, nzuimm[17:12] (rd \neq 0, rd \neq 2)	

Table 4-15 16-bit Constant-Generation

Mnemonics	32-Bit Counterpart	Notes
C.ADDI	addi rd, rd, nzimm[5:0] (rd \neq 0)	
C.ADDI4SPN	addi rd', x2, nzuimm[9:2]	
C.SLLI	slli rd, rd, shamt[4:0] (shamt[4:0] \neq 0)	
C.SRLI	srlr rd', rd', shamt[4:0] (shamt[4:0] \neq 0)	
C.SRAI	srai rd', rd', shamt[4:0] (shamt[4:0] \neq 0)	
C.ANDI	andi rd', rd', imm[5:0]	

Table 4-16 16-bit Integer Register-Immediate Instruction

Mnemonics	32-Bit Counterpart	Notes
C.MV	add rd, x0, rs2	
C.ADD	add rd, rd, rs2	
C.AND	and rd', rd', rs2'	
C.OR	or rd', rd', rs2'	
C.XOR	xor rd', rd', rs2'	
C.SUB	sub rd', rd', rs2'	

Table 4-17 16-bit Integer Register-Register Instruction

Mnemonics	32-Bit Counterpart	Notes
C.NOP	addi x0, x0, 0	
C.EBREAK	ebreak	

Table 4-18 16-bit Miscellaneous

4.2 Compiler

PulseRain Technology will provide a board package to be used by the Arduino IDE. This board package will include the GNU RISC-V Embedded GCC (<https://xpack.github.io/riscv-none-embed-gcc/>), which will produce bare-metal images to be uploaded by the Arduino IDE.

The Arduino board package also includes a Makefile and a link script the software developers can use for their own project. They can be found in the following link on GitHub:

https://github.com/PulseRain/Arduino_RISCV_IDE/tree/master/PulseRain_RISCV/GRV3000D/variants/generic

And the main part of the Makefile is also listed below in List 4-1:

```
RISCV_PREFIX = riscv-none-embed-
CC = $(RISCV_PREFIX)gcc
CXX = $(RISCV_PREFIX)g++
AS = $(RISCV_PREFIX)as
LD = $(RISCV_PREFIX)gcc
OBJDUMP = $(RISCV_PREFIX)objdump

CFLAGS = -I.
CFLAGS += --specs=nosys.specs -march=rv32i -mabi=ilp32 -O0 -fno-builtin-printf -
fdata-sections -ffunction-sections -fno-exceptions -fno-unwind-tables
CFLAGS += -W -DVMAJOR=1 -DVMINOR=1 -D__GXX_EXPERIMENTAL_CXX0X__
```

PulseRain GRV3000D – Programmer's Guide

```
LDFLAGS = -T ./GRV3000D.ld --specs=nosys.specs -march=rv32i -mabi=ilp32 -Os -fdata-
sections -ffunction-sections -Wl,--gc-sections -static -fno-exceptions -fno-unwind-
tables

#pattern rule

#== Put all the object files here
obj = main.o

target = GRV3000D.elf
dump = $(patsubst %.elf,%.dump,$(target))

all: $(obj)
    @echo "====> Linking $(target)"
    @$(LD) $(LDFLAGS) $(obj) -o $(target)
    @chmod 755 $(target)
    @echo "====> Dumping sections for $@"
    @$(OBJDUMP) --disassemble-all --disassemble-zeroes --section=.text --
section=.text.startup --section=.data --section=.rodata --section=.sdata --
section=.sdata2 --section=.init_array --section=.fini_array --section=.dsp_top --
section=.dsp_bottom $(target) > $(dump)

%.o : %.c
    @echo "====> Building $@"
    @echo "=====> Building Dependency"
    @$(CC) $(CFLAGS) -M $< | sed "s,$(@F)\s*:,,$@ :," > $.d
    @echo "=====> Generating OBJ"
    @$(CC) $(CFLAGS) -c -o $@ $<; \
    if [ $$? -ge 1 ] ; then \
        exit 1; \
    fi
    @echo "-----"

%.o : %.cpp
    @echo "====> Building $@"
    @echo "=====> Building Dependency"
    @$(CXX) $(CFLAGS) -M $< | sed "s,$(@F)\s*:,,$@ :," > $.d
    @echo "=====> Generating OBJ"
    @$(CXX) $(CFLAGS) -c -o $@ $<; \
    if [ $$? -ge 1 ] ; then \
        exit 1; \
    fi
    @echo "-----"
_"

dependency = $(patsubst %.o,%.d,$(obj))

ifneq "$(MAKECMDGOALS)" "clean"
    -include $(dependency)
endif

clean :
    -@rm -vf $(target)
    -@find . -type f \( -name "*.riscv" -o -name "*.d" -o -name "*.o" -o -name "*.lst"
-o -name "*.dump" \
        -o -name "*.bin" -o -name "*.out" -o -name "*.elf" \) -exec rm -vf {} \;

.PHONY: clean all
```

List 4-1 Makefile for Bare Metal

Notes for this Makefile:

1. For a project with multiple source files, they can be covered in this Makefile by adding them to the variable "obj" with .o file extension (separated by space).

PulseRain GRV3000D – Programmer's Guide

- Dependency information will be extracted from the .c/.cpp files and dumped into .d file before object is being generated. For each .c/.cpp file, its correspondent .d file will be included by Makefile, as shown in List 4-1. The default dependency information produced by compiler does not include directory path. That's what `sed "s,$(@F)\s*:$,@ :,"` is there for to correct this behavior, since directory path is required when .d files are being included.

The Makefile above relies heavily on a link script called GRV3000D.ld. This link script will allocate sections as the following in Table 4-19:

Base Address	Segment Name	Sections Included
0x80000000	text-segment	.text / .init / .text.startup / .text.exit
0xC0000000	data-segment	.data / .rodata / .ctors / .dtors / .init_array

Table 4-19 text-segment and data-segment

And it will also create two extra segments for DSP acceleration, assuming the D-TCM is configured as 64KB:

Base Address	Segment Name
0xC0007800	dsp-top
0xC0008000	dsp-bottom

Table 4-20 Segments for DSP Acceleration

The application of those segments from Table 4-20 will be discussed in later chapters of this document.

4.3 Assembler

For those who like to squeeze the performance to the last drop, the assembly language is a good tool. And this is especially true for DSP acceleration. The Makefile mentioned in Section 4.2 also supports GNU Assembler. And it assumes the assembly file ends with .rvs extension.

To help the software developers to program in Assembly, the following head files are provided along with the Makefile:

Head File	Description
mm_reg.rvh	Memory Mapped Registers' address for Peripherals mentioned in Section 3.4.3
macros.rvh	Handy macros, useful to assembly programmer

Table 4-21 Assembly Head Files

And the following marco defintions are provided by the macros.rvh: (Those macros are optimized to take advantage of the dual-issue pipeline in the GRV3000D.)

Macros	Description
wr_mtimecmp	Update the 64-bit MTIMECMP register mentioned in Section 3.4.3.1
wr_mm_reg	Write to Memory Mapped Register
rd_mm_reg	Read from Memory Mapped Register

Table 4-22 Assembly Macros

In addition, inline assembly is another way to achieve DSP acceleration with C language, which will be discussed in later chapters of this document.

4.4 Image Loading

As illustrated in Figure 3-6, binary image can be loaded into the TCM through either JTAG or HW Based Bootloader. For the HW Based Bootloader, a Python script called GRV3000D_Config.py can be used on the host side. This script will invoke the GNU Binary Utilities to extract sections from the .elf file, and transfer them to the TCM through UART. The source code and its binary (Windows executable) can be found in

https://github.com/PulseRain/Arduino_RISCV_IDE/tree/master/GRV3000D_upload

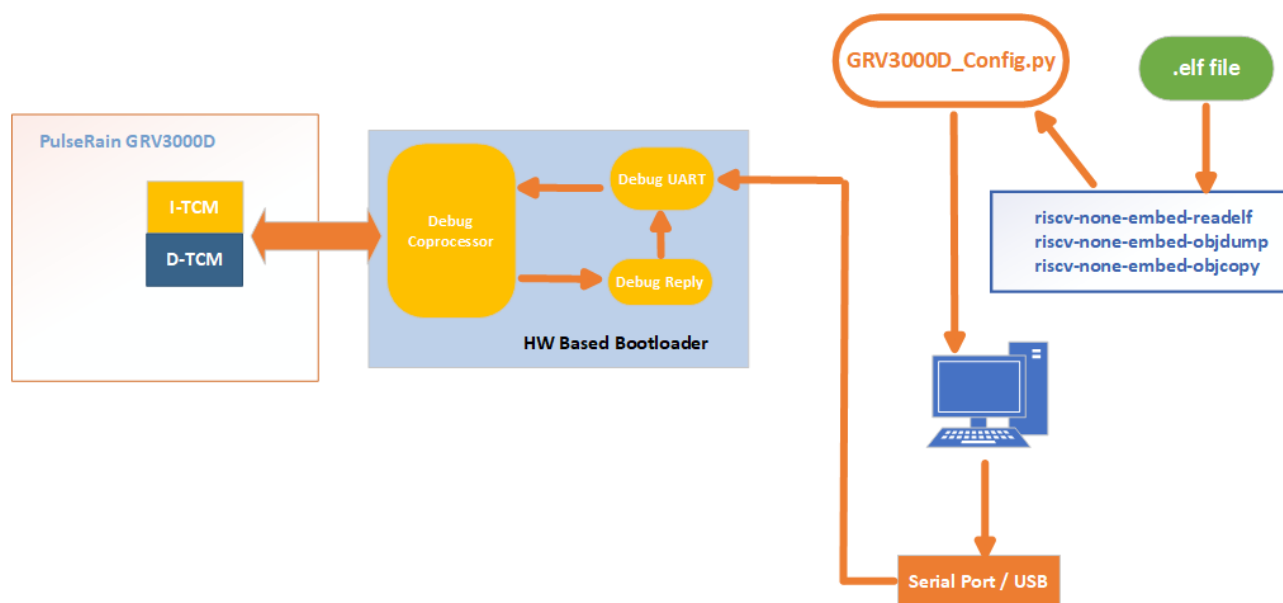


Figure 4-1 Python Script to Load Image through HW Base Bootloader

The command line options of this script are listed in Table 4-23.

Options	Description
-r, --reset	Reset the processor
-a, --start_addr=	Start address for execution. The default is 0x80000000
-R, --run	Execute from the start address (Set PC to the start address)

Options	Description
-d, --baud=	The baud rate of the COM port. The default is 115200
-t, --toolchain=	Set up the toolchain. By default, the prefix "riscv-non-embed-" is used
-i, --image=	Path and name to the image file
-c, --console_enable	Set the active UART to the one in AXI4-Lite Switch (See Figure 3-6), observe its output after image is loaded.

Table 4-23 Command Options for GRV3000D_Config.py

On Windows platform, a typical example for using this script is like the following:

Python GRV3000D_Config.py --port=COMx --reset --run --console_enable --image=path_to_the_image

where COMx is the correspondent COM port for the UART connection on host PC.

(On Windows, instead of using the Python script, its binary version GRV3000D_Config.exe can be used alternatively. The Windows executable does not require Python to be installed, but the GNU Binary Utilities should be accessible through the environment variable "PATH".)

5 Software - Arduino Flow

For bare metal code, there are mainly three approaches for software developers to proceed:

1. Using Makefile (such as the one mentioned in Section 4.2) to build the image. And use the Python Script mentioned in Section 4.4. The code can be fine-tuned, but for most part, this flow is done through command line.
2. Using Arduino IDE, which comes handy and quick.
3. Using JTAG Debug Probe, which is a full-blown debug process. But it requires external hardware (JTAG Probe) and the correspondent commercial software tools (such as Segger Embedded Studio).

For those options above, option 1 has already been discussed early, and this chapter will focus on option 2. Option 3 will be discussed in the next chapter.

5.1 Introduction to Arduino

According to Ref [9], Arduino originated from the master thesis of Hernando Barragán, who was a student of Interaction Design Institute Ivrea, Italy back then. In his thesis, he proposed a development platform called Wiring. And later his advisor Massimo Banzi and others gave rise to the Arduino platform, based on a fork of Wiring.

After Arduino came into being, it was well received by the open-source community and Makers around the world. The highlights of Arduino are mainly from the following three:

1. Arduino Dev Board
In particular, Arduino has put the Dev Board into two categories: the base board and the shield (extension board). Arduino Dev Board will not be the focus of this document. And inquisitive readers can find more information for Dev Board on Arduino Website (<https://www.arduino.cc>)
2. Arduino IDE and Arduino Language, which will be the focus of this chapter
3. Arduino community
Arduino community is another value asset for developers around the world. Other than being a public forum, the Arduino community has standardized the software publishing flow. With the flow being streamlined, third party developers can release their new BSP (Board Support Package) and Library to the whole community in a timely fashion.

5.2 Arduino IDE

Arduino Integrated Development Environment (IDE) is a handy tool for bare metal programming. With the Board Support Package provided by PulseRain Technology, the Arduino IDE can be used on the GRV3000D dev-kit.

5.2.1 Introduction to Bare Metal Systems

For many embedded systems, they can finish the job without resorting to embedded operating systems (OS). And the software is interacting directly with the bare hardware, without using any system calls from OS. Such systems are thus called bare-metal systems.

And for some bare metal systems, in order to improve the portability of the application code, there might be a middle layer called HAL (Hardware Abstraction Layer) between the application code and the hardware, as shown in Figure 5-1.

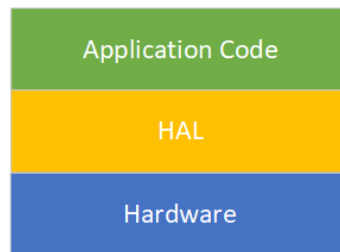


Figure 5-1 Application Code, HAL (Hardware Abstraction Layer) and Hardware

For bare-metal systems, they often run as a super loop, where the tasks are executed sequentially. And before the program enters the loop, there might be an initialization process in the very beginning, as illustrated in Figure 5-2.

To handle asynchronous events, such as key stroke from the users, ISR (Interrupt Service Routine) can be used. Of course, the asynchronous events can also be dealt with in the loop, through polling. However, the polling may have a long latency as the polling task may have to wait until all the tasks before it are completed. If tasks are added or removed to/from the loop, the latency may also change.

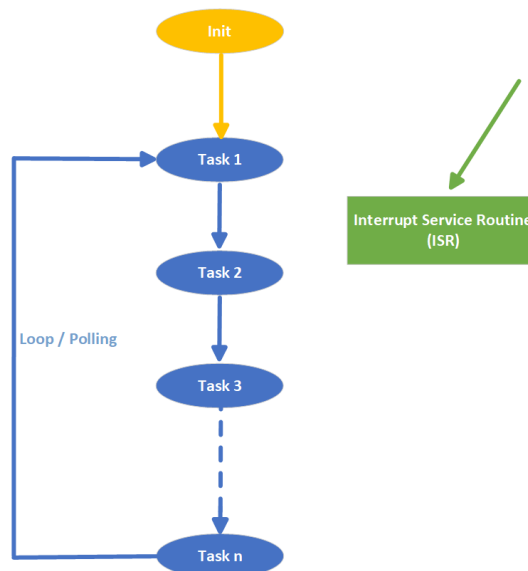


Figure 5-2 Bare Metal System

PulseRain GRV3000D – Programmer's Guide

5.2.2 Installation of Arduino IDE and Board Support Package for the GRV3000D

On Windows 10, the Arduino IDE can be installed directly from the Microsoft Store. On other platforms, it can be downloaded from Arduino Website <https://www.arduino.cc/en/software>.

After the Arduino IDE is installed, it should be configured for the GRV3000D Board Support Package. And the procedures are as following:

1. Launch the Arduino IDE and open Menu File / Preferences, as shown in Figure 5-3



Figure 5-3 Arduino IDE, Menu File / Preferences

2. Add https://github.com/PulseRain/Arduino_RISCV_IDE/raw/master/package_pulserain.com_index.json to Additional Boards Manager URLs, as illustrated in Figure 5-4.

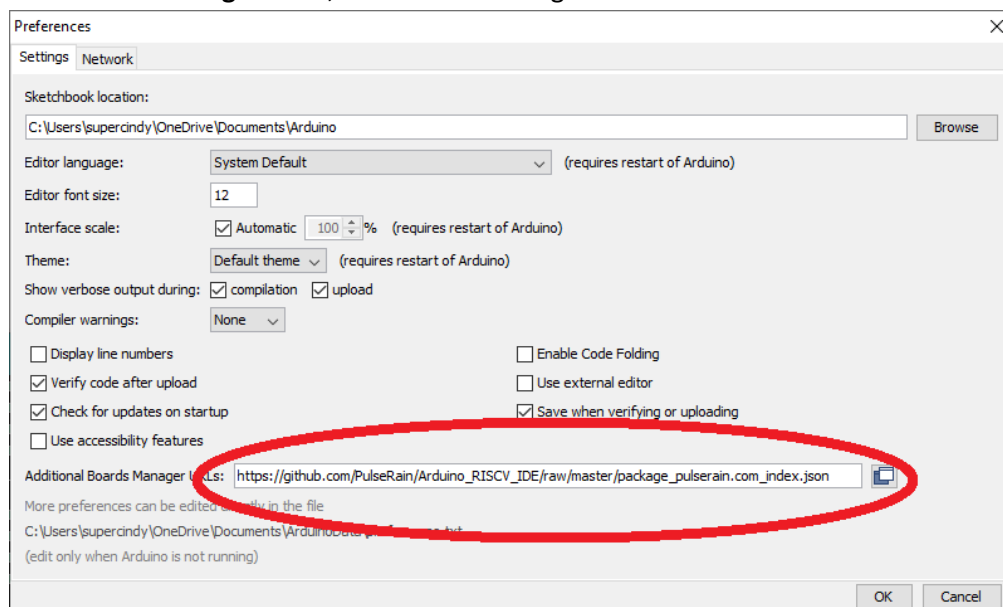


Figure 5-4 Arduino IDE, Setup Additional Boards Manager URLs

PulseRain GRV3000D – Programmer's Guide

3. Open **Board Manager** under Menu Tools / Board / Board Manager...

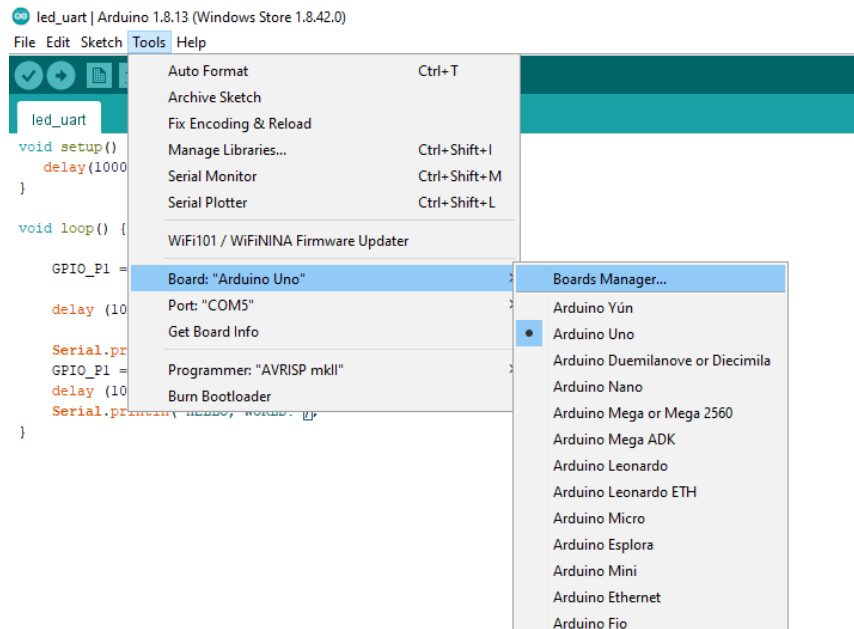


Figure 5-5 Arduino IDE, Open Board Manager

4. Type in "PulseRain" in the Search Box to find the board for GRV3000D and install

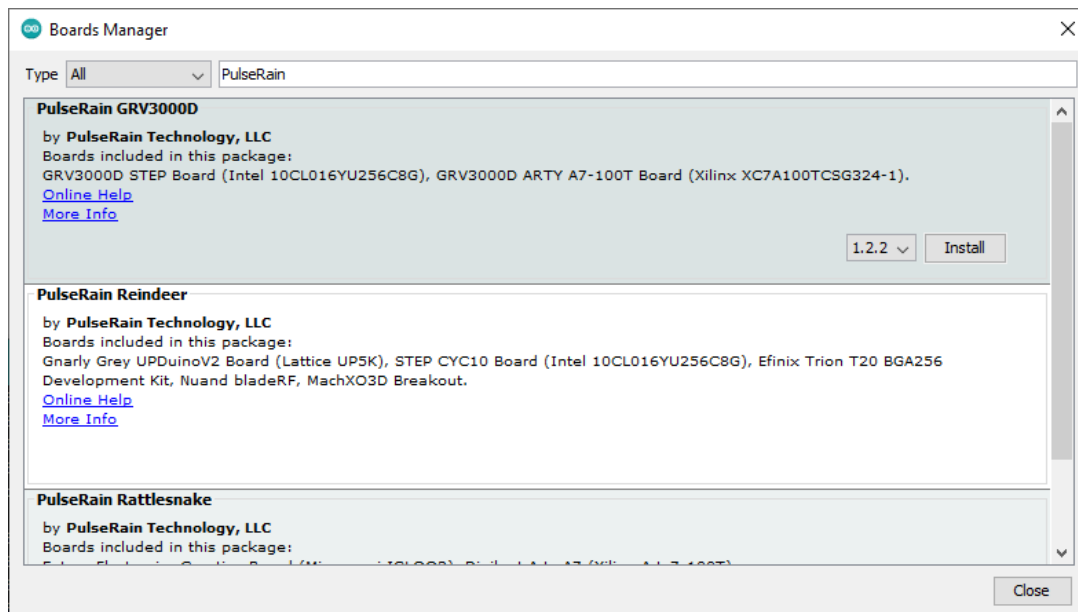


Figure 5-6 Arduino IDE, Install Board Support Package for PulseRain GRV3000D

PulseRain GRV3000D – Programmer's Guide

5.2.3 The Workflow for Arduino IDE

Under Arduino IDE, the program written by the developers is called Sketch (with extension as .ino). The Sketches are actually C/C++ code snippets. It needs to provide two functions: `setup()` and `loop()`.

As illustrated in Figure 5-7, the `setup()` function will be invoked once, while the `loop()` function will be called up continuously. Thus, the `setup()` function is mainly responsible for the system initialization, and the main tasks are executed through the `loop()` function. If interrupt handler (ISR) is needed, the ISRs can be hooked through the `attachInterrupt()` function.

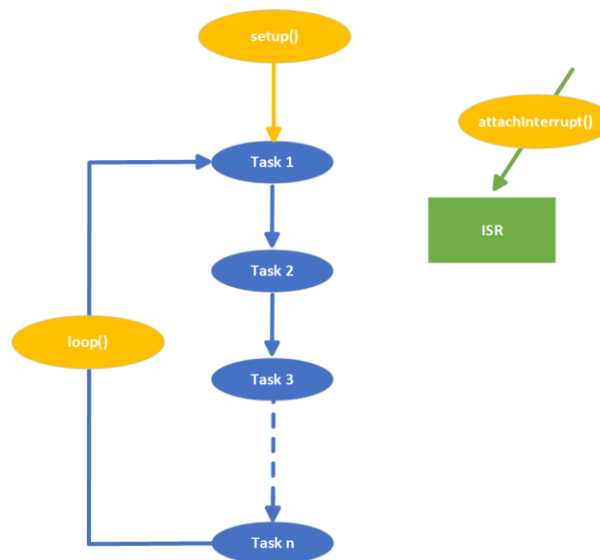


Figure 5-7 Arduino Workflow

Readers may have noticed that Figure 5-7 bears a striking resemblance to Figure 5-2, which speaks generally for the way of bare metal systems.

5.2.4 What is Under the Hood for Arduino IDE

For those who are familiar with C/C++, it is quite odd that head files, `main()` function and C library are nowhere to be found in Figure 5-7. In this regard, Arduino IDE has done a good job hiding those devil details.

After the developer finishes the Sketch and starts compiling, Arduino IDE will do the following behind the scene:

1. Assume the Sketch is called `hello.ino`. Arduino IDE will thus generate another file called `hello.ino.cpp`. Other than including everything from `hello.ino`, `hello.ino.cpp` will also prepend the following:

```
#include <Arduino.h>
#line 1 "C\".....\hello.ino"
```

So what it does is to include the head file, and adjust the line number so the line number reported will still align with `hello.ino`.

PulseRain GRV3000D – Programmer's Guide

2. Arduino will also attach main() and C library to the Sketch. And those source code can be found in the Board Support Package for the GRV3000D. On Windows Platforms, they can be found in

Documents\ArduinoData\packages\PulseRain_RISCV\hardware\GRV3000D\version\cores\GRV3000D

And the main() function can be found in main.cpp under the above directory, which may look like the code snippets shown in List 5-1. Basically, the main() function is to turn the flow chart in Figure 5-7 into reality and hide the details from the developers.

```
int main()
{
    noInterrupts();
    Serial.begin(115200);
    write_csr (mtvec, (uint32_t)shared_isr);

    //+++++
    // ARDUINO sketch
    //+++++
    setup();

    while (1) {
        loop();
    } // End of while loop

    //+++++

    return 0;
} // End of main()
```

List 5-1 Arduino main()

5.3 Arduino Language

Another contributing factor for Arduino's popularity is the Arduino Language, which "turns the Sketch into a cinch". The Arduino Language is derived from the Wiring Language mentioned in Section 5.1. And its official document can be found from Arduino's website <https://www.arduino.cc/reference/en/>

With the BSP provided by PulseRain Technology, the GRV3000D processor has managed to present a software programming interface that is compatible with the Arduino Language, and supports a large subset of it. Fundamentally the Arduino Language is not a new programming language, but a collection of API (Application Interface) functions and data types. From which, portability can be achieved among different hardware platforms. And the implementation of those APIs can be also be found in the path mentioned in Section 5.2.4.

5.3.1 Data Type

Compatible with the Arduino Language, the following data types are defined in common_type.h:

```
typedef unsigned long    uint32_t;
typedef long             int32_t;
typedef unsigned short   uint16_t;
typedef short            int16_t;
```



```
typedef unsigned char    uint8_t;
typedef signed char      int8_t;
typedef uint8_t          byte;
typedef uint16_t          word;
typedef uint8_t          boolean;
```

List 5-2 Common Data Type

And the String class is defined in WString.h, which has the following Constructors:

```
String(char c);
String(unsigned char, unsigned char base=10);
String(int, unsigned char base=10);
String(unsigned int, unsigned char base=10);
String(long, unsigned char base=10);
String(unsigned long, unsigned char base=10);
String(float, unsigned char decimalPlaces=2);
String(double, unsigned char decimalPlaces=2);
```

List 5-3 String Type

5.3.2 APIs

The complete list of the API supported by the BSP can be found in the Appendix. And the following are some of the most frequently used ones:

5.3.2.1 Digital IO

Digital IO implementation can be found in GRV3000D.cpp, with the functions like:

- *void digitalWrite (uint8_t pin, uint8_t value)*
// set the logic level of the IO pin
- *uint8_t digitalRead (uint8_t pin)*
// read the logic level of the IO pin

5.3.2.2 Time and Delay

Time and Delay are also implemented in GRV3000D.cpp, with functions like:

- *void delay (uint32_t delay_in_ms)*
// delay in the granularity of millisecond
- *void delayMicroseconds (uint32_t delay_in_us)*
// delay in the granularity of microsecond
- *uint32_t millis ()*
// get the number of milliseconds passed since reset
- *uint32_t micros ()*
// get the number of microseconds passed since reset

PulseRain GRV3000D – Programmer's Guide

5.3.2.3 Serial Port

The implementation of the Serial Port Class (HardwareSerial) can be found in HardwareSerial.cpp and Print.cpp, with functions like:

- `int Serial.available ()`
// get the number of available bytes in the RX FIFO
- `int Serial.read ()`
// read data from the RX FIFO
- `size_t Serial.write (unsigned long n)`
`size_t Serial.write (long n)`
`size_t Serial.write (unsigned int n)`
`size_t Serial.write (int n)`
`size_t Serial.println (const String &s)`
`size_t Serial.println (const char c[])`
// write number or print string to the Serial Port

5.3.2.4 Interrupt

Please see Section 5.4 for Interrupt Handler.

5.4 Interrupt Handler

The GRV3000D's AXI4-Lite Switch will have a cluster of peripherals (Section 3.4.3). And some of those peripherals, such as the System Timer and UART, will generate interrupts. As illustrated in Figure 3-5, the System timer will have its dedicated pin to the processor core, while the rest of the peripherals will share another interrupt pin (shared external interrupt). The status of those two pins can be read out from CSR register **mip**. The former is mapped to the MTIP (Machine Timer Interrupt Pending) bit in Table 3-18 while the latter is mapped to the MEIP (Machine External Interrupt Pending) bit.

And each individual external interrupt will be assigned an Interrupt Index according to Table 3-19.

5.4.1 Shared ISR (Interrupt Service Routine)

As defined by the RISC-V specification (Ref [1][2]), when interrupt or exception happens, the PC (Program Counter) will be set to the address stored in CSR **mtvec**. Because of that, one shared ISR can be used to handle all the interrupts (both system timer interrupt and external interrupt) or exceptions. And inside this shared ISR, the interrupt source can be determined by reading out the **INT_SOURCE** (memory mapped register) in Table 3-19.

In the Arduino BSP for the GRV3000D, a shared ISR like that has been provided in main.cpp. Its main content is shown in List 5-4. And its correspondent flowchart is in Figure 5-8.

```
static void shared_isr(void) __attribute__((interrupt));  
static void shared_isr(void) {
```

```
uint32_t t = read_csr (mcause);
uint8_t i, code = t & 0xFF;
uint32_t old_mstatus_value = read_csr (mstatus);
write_csr(mstatus, 0);

if ((t & MCAUSE_INTERRUPT) == 0) { // Exception
    Serial.print ("Exception !!!!! Exception Code = 0x");
    Serial.println (code, HEX);
    Serial.print ("MEPC = 0x");
    Serial.println (read_csr(mepc), HEX);
} else { // Interrupt

    if (code == MCAUSE_TIMER) {

        t = read_csr (mip);
        t &= ~MCAUSE_TIMER_MASK;
        write_csr (mip, t);

        if (timer_isr) {
            timer_isr();
        }

    } else {

        t = read_csr (mip);
        t &= ~MCAUSE_PERIPHERAL_MASK;
        write_csr (mip, t);

        t = *REG_INT_SOURCE;

        if (t & (1 << INT_UART_RX_INDEX)) {
            if (uart_rx_isr) {
                uart_rx_isr ();
            }
        }

        for (i = INT_EXTERNAL_1ST; i <= INT_EXTERNAL_LAST; ++i) {
            if (t & (1 << i)) {

                if (INTx_isr[i - INT_EXTERNAL_1ST]) {
                    INTx_isr[i - INT_EXTERNAL_1ST]();
                }
            }
        } // End of for loop
    }
}

write_csr(mstatus, old_mstatus_value);
} // End of shared_isr()
```

List 5-4 Shared ISR

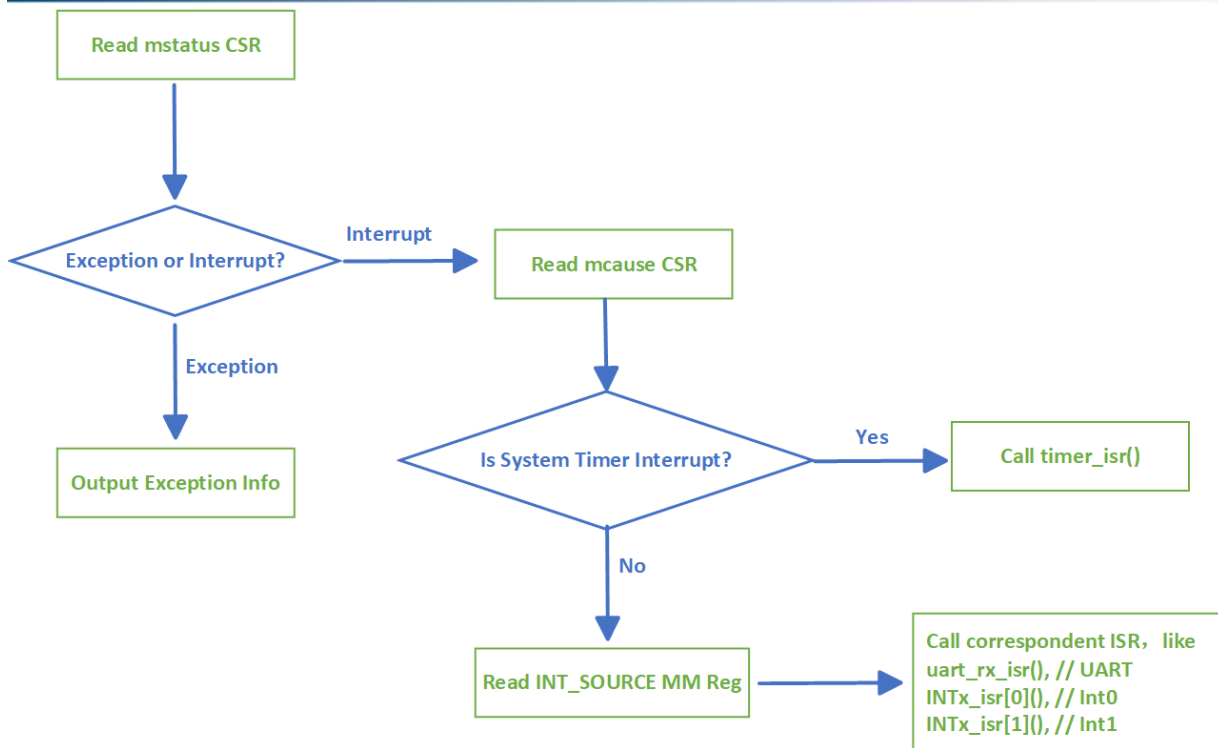


Figure 5-8 Flowchart for Shared ISR

Please note that there is a GCC attribute "`__attribute__((interrupt))`" attached to the `shared_isr()`. Its main purpose is to tell GCC to add prologue and epilogue to the function entrance and function exit. Inquisitive readers can use "`riscv-none-embed-objdump -d ...`" command to disassemble the code and observe the details of the prologue and epilogue, as shown in List 5-5.

```

80000e58 <_ZL10shared_isr_v>:
80000e58: fb010113      addi    sp,sp,-80
80000e5c: 04112623      sw      ra,76(sp)
80000e60: 04512423      sw      t0,72(sp)
80000e64: 04612223      sw      t1,68(sp)
80000e68: 04712023      sw      t2,64(sp)
80000e6c: 02812e23      sw      s0,60(sp)
80000e70: 02912c23      sw      s1,56(sp)
80000e74: 02a12a23      sw      a0,52(sp)
80000e78: 02b12823      sw      a1,48(sp)
80000e7c: 02c12623      sw      a2,44(sp)
80000e80: 02d12423      sw      a3,40(sp)
80000e84: 02e12223      sw      a4,36(sp)
80000e88: 02f12023      sw      a5,32(sp)
80000e8c: 01012e23      sw      a6,28(sp)
80000e90: 01112c23      sw      a7,24(sp)
80000e94: 01212a23      sw      s2,20(sp)
80000e98: 01c12823      sw      t3,16(sp)
80000e9c: 01d12623      sw      t4,12(sp)
80000ea0: 01e12423      sw      t5,8(sp)
80000ea4: 01f12223      sw      t6,4(sp)
  
```

...

PulseRain GRV3000D – Programmer's Guide

```

80000f00: 03c12403      lw    s0,60(sp)
80000f04: 04c12083      lw    ra,76(sp)
80000f08: 04812283      lw    t0,72(sp)
80000f0c: 04412303      lw    t1,68(sp)
80000f10: 04012383      lw    t2,64(sp)
80000f14: 03812483      lw    s1,56(sp)
80000f18: 03412503      lw    a0,52(sp)
80000f1c: 03012583      lw    a1,48(sp)
80000f20: 02c12603      lw    a2,44(sp)
80000f24: 02812683      lw    a3,40(sp)
80000f28: 02412703      lw    a4,36(sp)
80000f2c: 02012783      lw    a5,32(sp)
80000f30: 01c12803      lw    a6,28(sp)
80000f34: 01812883      lw    a7,24(sp)
80000f38: 01412903      lw    s2,20(sp)
80000f3c: 01012e03      lw    t3,16(sp)
80000f40: 00c12e83      lw    t4,12(sp)
80000f44: 00812f03      lw    t5,8(sp)
80000f48: 00412f83      lw    t6,4(sp)
80000f4c: 05010113      addi  sp,sp,80
80000f50: 30200073      mret

```

List 5-5 Prologue and Epilogue for Shared ISR

5.4.2 Device ISR (Interrupt Service Routine)

As demonstrated in List 5-4 and Figure 5-8, the shared ISR will determine the source of the interrupt, and invoke the correspondent device specific ISR, such as `timer_sr()`, `uart_rx_isr()`, `INTx_isr[0]` etc.

Those device ISRs are installed through **`attachInterrupt()`**, or uninstalled through **`detachInterrupt()`**. They all have the prototype as **`"typedef void (*ISR)();"`**.

```

void attachInterrupt (uint8_t int_index, ISR isr, uint8_t mode)
{
    if (mode == RISING) {
        if (int_index == INT_TIMER_INDEX) {
            timer_isr = isr;
        } else if (int_index == INT_UART_RX_INDEX) {
            uart_rx_isr = isr;
            (*REG_INT_ENABLE) |= 1 << INT_UART_RX_INDEX;
        } else if ((int_index >= INT_EXTERNAL_1ST) && (int_index <=
INT_EXTERNAL_LAST)) {
            INTx_isr [int_index - INT_EXTERNAL_1ST] = isr;
            (*REG_INT_ENABLE) |= 1 << int_index;
        } else {
            Serial.print ("unknown interrupt index ");
            Serial.println(int_index);
        }
    } else {
        Serial.print ("unsupported mode ");
        Serial.print (mode);
        Serial.println (" for attachInterrupt");
        return;
    }
} // End of attachInterrupt()

```

List 5-6 `attachInterrupt()`

```
void detachInterrupt (uint8_t int_index)
{
    if (int_index == INT_TIMER_INDEX) {
        timer_isr = 0;
    } else if (int_index == INT_UART_RX_INDEX) {
        uart_rx_isr = 0;
        (*REG_INT_ENABLE) &= ~(1 << INT_UART_RX_INDEX);
    } else if ((int_index >= INT_EXTERNAL_1ST) && (int_index <=
INT_EXTERNAL_LAST)) {
        INTX_isr [int_index - INT_EXTERNAL_1ST] = 0;
        (*REG_INT_ENABLE) &= ~(1 << int_index);
    } else {
        Serial.print ("unknown interrupt index ");
        Serial.println(int_index);
    }
} // End of detachInterrupt()
```

List 5-7 detachInterrupt()

The main content of **attachInterrupt()** is shown in List 5-6, and the one for the **detachInterrupt()** is shown in List 5-7. The "int_index" in List 5-6 and List 5-7 is the index for external interrupt, which is defined in Table 3-19 as the bit position for the correspondent device. Accordingly, the INT_ENABLE register (Section 3.4.4.5) is used to enable or mask the individual interrupt from each device.

5.4.3 Device ISR Example

With the shared ISR structure set out in Figure 5-8, most of the time software developers only need to provide device specific ISR. And the following is an example for the UART RX ISR, as shown in List 5-8. In this ISR, the data read from the UART RX FIFO will be put into a buffer (uart_rx_buf) that is managed by the read pointer and write pointer.

```
uint8_t uart_rx_buf [256] = {0};
uint8_t uart_rx_index_write_point = 0;
uint8_t uart_rx_index_read_point = 0;

void uart_rx_isr()
{
    uint8_t t;

    t = Serial.read();
    uart_rx_buf [uart_rx_index_write_point++] = t;
} // End of uart_rx_isr()
```

List 5-8 ISR for UART RX

After the ISR is prepared, use "attachInterrupt (INT_UART_RX_INDEX, uart_rx_isr, RISING);" to install the ISR in setup(). At this point, the last parameter should always be RISING for the GRV3000D BSP.

Call interrupts() in setup() to enable interrupt.

PulseRain GRV3000D – Programmer's Guide

And finally in the loop() function, check the content of the uart_rx_buf, as shown in List 5-9.

```
if (uart_rx_index_read_point != uart_rx_index_write_point) {
    Serial.print("\n Got Message: ");
    do {
        Serial.write(&uart_rx_buf[uart_rx_index_read_point++], 1);
    } while (uart_rx_index_read_point != uart_rx_index_write_point);
    Serial.print("\n");
}
```

List 5-9 Read uart_rx_buf in loop()

5.5 Arduino Library

With the contribution from developers around the world, Arduino has a huge collection of 3rd-party libraries. And those libraries are probably the most valuable asset for Sketch writers. In addition, Sketch writers can also offer their own library to the Arduino Library Manager to benefit the Arduino community worldwide.

5.5.1 Use 3rd-party library in Arduino IDE

5.5.1.1 Install 3rd-party Library

Under the Arduino IDE, 3rd-party library can be installed through the following steps:

1. Open Menu Sketch / Include Library / Manage Libraries..., as shown in Figure 5-9.

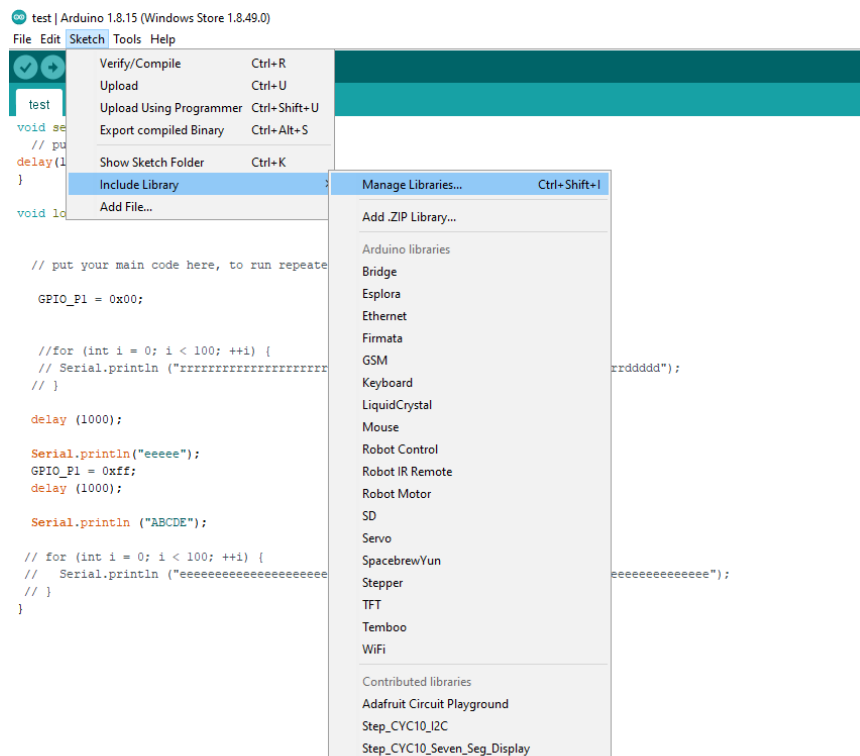


Figure 5-9 Install Arduino Library

2. This menu will bring out a dialogue called Library Manager, as shown in Figure 5-10. And users can type in the library name in the search box to find the relevant ones.

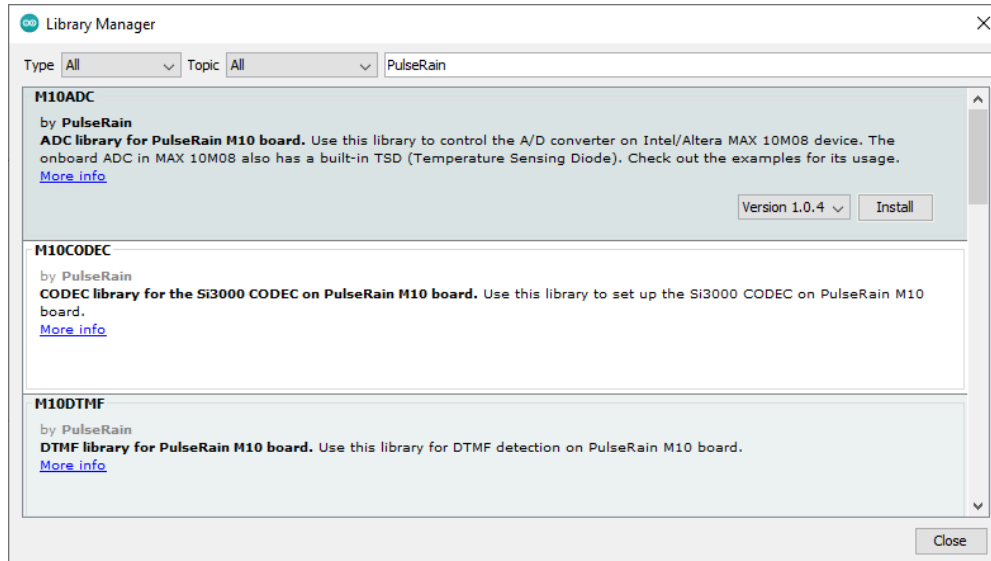


Figure 5-10 Arduino Library Manager

3. On Windows 10 platform, if the library is installed successfully, it will usually be found in the path: Documents\Arduino\libraries

5.5.1.2 Use 3rd-party Library in Sketches

An Arduino Library usually includes three sub-folders:

1. src: the source code of the library
2. extras: document is often put in this folder
3. examples: example sketches for the library

To use the Arduino Library in Sketch, the library's head file needs to be included in the beginning of the Sketch. The Arduino Library is written in C++, which includes Class definition and one correspondent object. This object will expose methods as API for the Sketches to use.

5.5.2 Use 3rd-party library in Arduino IDE

5.5.2.1 Create GitHub Repository

If the software developer likes to create a new library, he or she needs to create a new repository on GitHub. And the folder structure of this repository should be like the one shown in Figure 5-11.

The "src" folder in Figure 5-11 is mandatory. As mentioned early, the Arduino Library should be written in C++, which includes the definition of Class and one correspondent object.

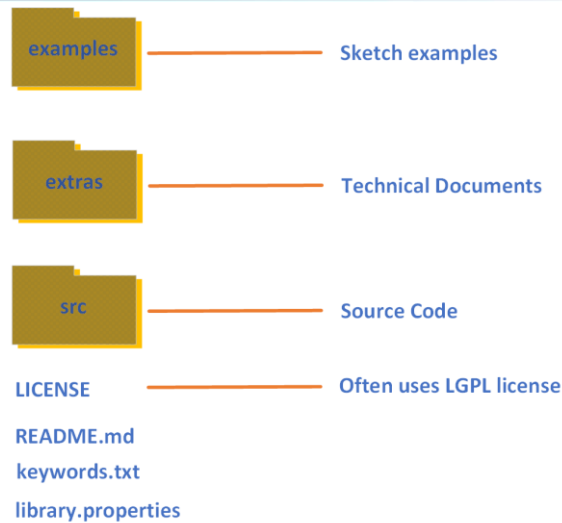


Figure 5-11 Arduino Library Folder Structure

Other than the folders, this repository should include the following two txt files:

1. keywords.txt

This file will be used by Arduino IDE for syntax color. And List 5-10 is one example.

```
#####
# Syntax Coloring Map For Step_CYC10_Seven_Seg_Display
#####

#####
# Datatypes (KEYWORD1)
#####

Step_CYC10_Seven_Seg_Display KEYWORD1

#####
# Methods and Functions (KEYWORD2)
#####

reset KEYWORD2
set_display_value KEYWORD2
start_refresh KEYWORD2
stop_refresh KEYWORD2

#####
# Instances (KEYWORD3)
#####

SEVEN_SEG_DISPLAY KEYWORD3
#####
# Constants (LITERAL1)
#####
TIMER_RESOLUTION LITERAL1
```

List 5-10 Example of keywords.txt

PulseRain GRV3000D – Programmer's Guide

2. library.properties

library.properties should contain the library name, version, GitHub URL etc. List 5-11 is one such example. Please note that the version here should match the correspondent tag of the repository. In other words, after library.properties being created, a new release needs to be created as well, whose tag version should be identical to one provided by library.properties.

```
name=Step_CYC10_Seven_Seg_Display
version=1.0.3
author=PulseRain
maintainer=PulseRain <info@pulserain.com>
sentence=Library for the 7-segment display on Step CYC10 FPGA board
paragraph=Use this library to control the 7-segment display on Step CYC10 FPGA board
category=Signal Input/Output
url=https://github.com/PulseRain/Step_CYC10_Seven_Seg_Display
architectures=RISC-V
includes=Step_CYC10_Seven_Seg_Display.h
```

List 5-11 Example of library.properties

5.5.2.2 Submit the Library for Review

After the GitHub Repository being created and tagged, it needs to be submitted to the Arduino official representatives for review. The developers need to open a new issue under Arduino's GitHub Repository: <https://github.com/arduino/Arduino>

The title of the issue can be something like "[Library Manager] Please add ... library to the Library Manager". And the content of the issue should contain the GitHub URL for the new library.

After the issue is opened, Arduino representatives will verify the integrity of the GitHub Repository. If problems are found, they will notify the author and offer advices for correction. If the review is passed, the issue will be marked as Component: Board/Lib Manager, and another group of Arduino representatives will take over and add the new library to the Library Manger, so that this library can be found through the search box in Figure 5-10. In the meantime, the newly added library will be broadcasted on Twitter through @adduinoilibs.

6 Software – JTAG Debug

As illustrated in Figure 3-6, the software for PulseRain GRV3000D can be developed with the assistance of HW Based Bootloader or JTAG Debugger. The Arduino Flow mentioned in Chapter 5 is mainly using the HW Based Bootloader. And this chapter will focus on the JTAG Debugger.

6.1 External Debug Probe

Generally speaking, the JTAG debugger needs an external debug probe to function, as illustrated in Figure 6-1.

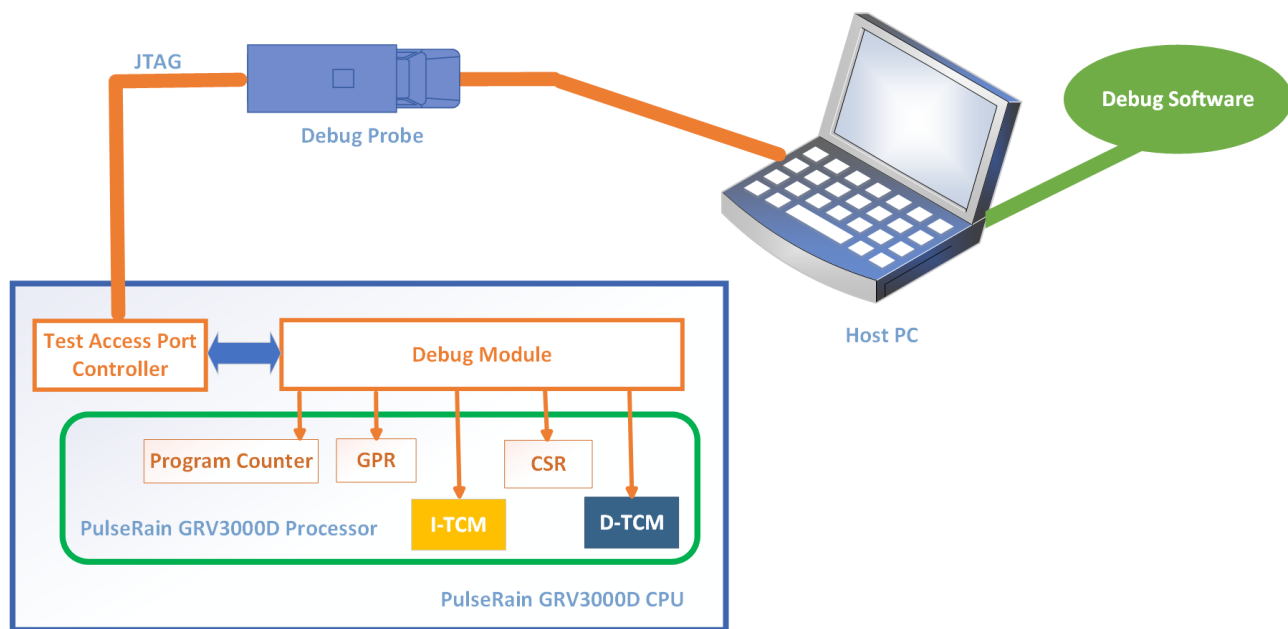


Figure 6-1 External Debug Probe

In Figure 6-1 the host PC is connected to the GRV3000D CPU through a debug probe external to the CPU, and the debug probe interfaces with the CPU through JTAG link. Between the host PC and the probe, the link can usually be USB, RS232 or Ethernet. The host PC runs the debug software, such as GNU debugger, debugger server etc.

6.2 Segger J-Link and Embedded Studio

The GRV3000D complies with RISC-V External Debug Support Version 0.13.2 (Ref [3]). Any debug probe that is compatible with Ref [3] can be used as the external probe shown in Figure 6-1. And this section will demonstrate how to use Segger J-Link and Embedded Studio to debug the code on PulseRain GRV3000D.

J-Link is a debug probe manufactured by Segger Microcontroller GmbH (<https://www.segger.com>), while Embedded Studio is the correspondent IDE (Integrated Development Environment) that can be used to both compile and debug the code. And the rest of this section is based on the following hardware and software version:

PulseRain GRV3000D – Programmer's Guide

- Segger J-Link Base V11, Emulator Firmware 04/27/2021, 16:36
- Segger Embedded Studio for RISC-V, Release 5.44, Build 2021050402.46116, Windows x64

6.2.1 Connect the Probe

The standard JTAG has 6 signal pins:

1. nRESET
2. TCK
3. nTRST
4. TMS
5. TDI
6. TDO

On the PCB, the pin-head for JTAG can be in a wide variety of layouts or form factors (10 pin / 12 pin / 20 pin etc.). The ribbon cable that comes with the J-Link BASE has a 20-pin connector. If the JTAG pin-head on the PCB is incompatible with the J-Link connector, an adapter is needed. For the Digilent Arty7-100T Dev Board (See the Appendix), Segger offers a 20-pin to 12-pin adapter to be purchased.

And the J-Link Probe is connected to host PC through USB cable. On the host PC, Embedded Studio for RISC-V should be installed, as illustrated in Figure 6-2.

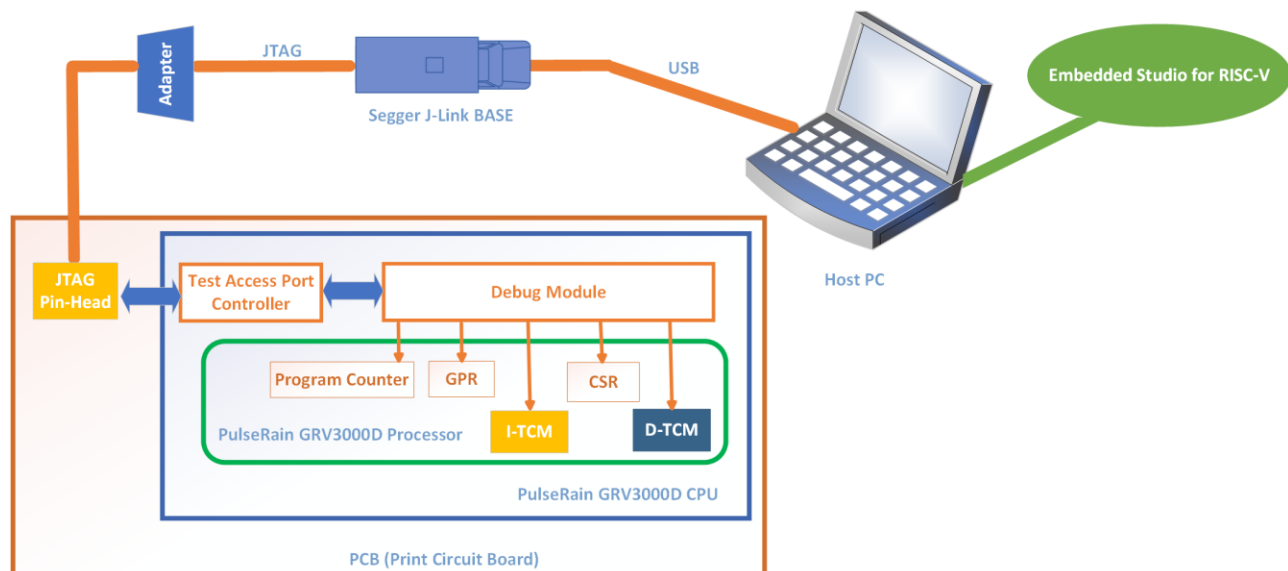


Figure 6-2 Connect the J-Link Probe and Install Embedded Studio

6.2.2 Build Project under the Embedded Studio for RISC-V

After the J-Link probe is connected, the software can be compiled and debugged entirely under the Embedded Studio for RISC-V. And this section will demonstrate the necessary steps for a "hello world" project:

PulseRain GRV3000D – Programmer's Guide

1. Install the Embedded Studio for RISC-V from Segger, and update the firmware on J-Link to the latest.
2. Connect J-Link to the Dev Kit for the GRV3000D. The details of the Dev Kit can be found in Appendix.
3. Power on the Dev Kit
4. Start the Embedded Studio for RISC-V, close the current solution, and Choose Menu File / New Project ... to start a new solution, as illustrated below in Figure 6-3.

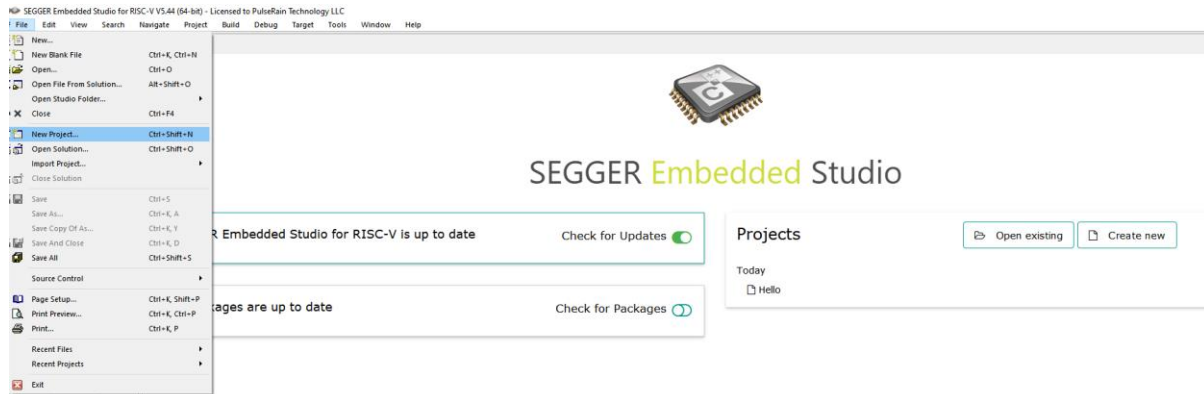


Figure 6-3 Embedded Studio, Start Page

5. In the New Project Dialogue, select "A C/C++ executable for a RISC-V processor.". And type in "hello_word" for the project name, as illustrated in Figure 6-4.

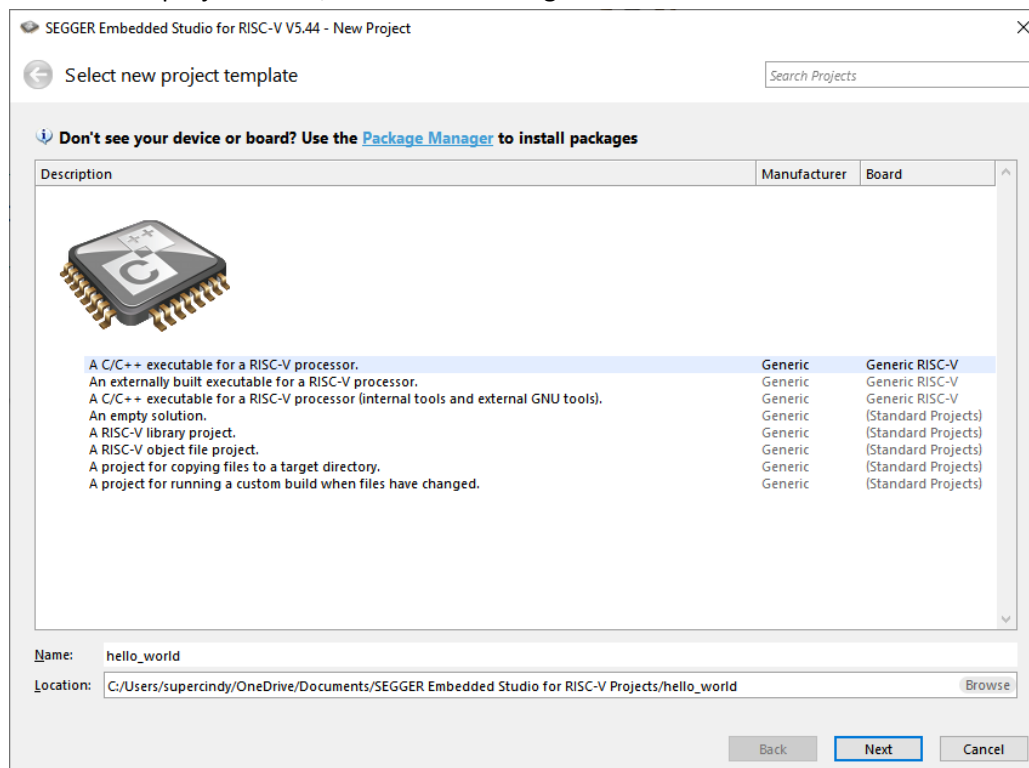


Figure 6-4 Embedded Studio, New Project Dialogue

6. And on the next dialogue, choose RV32, as shown in Figure 6-5.

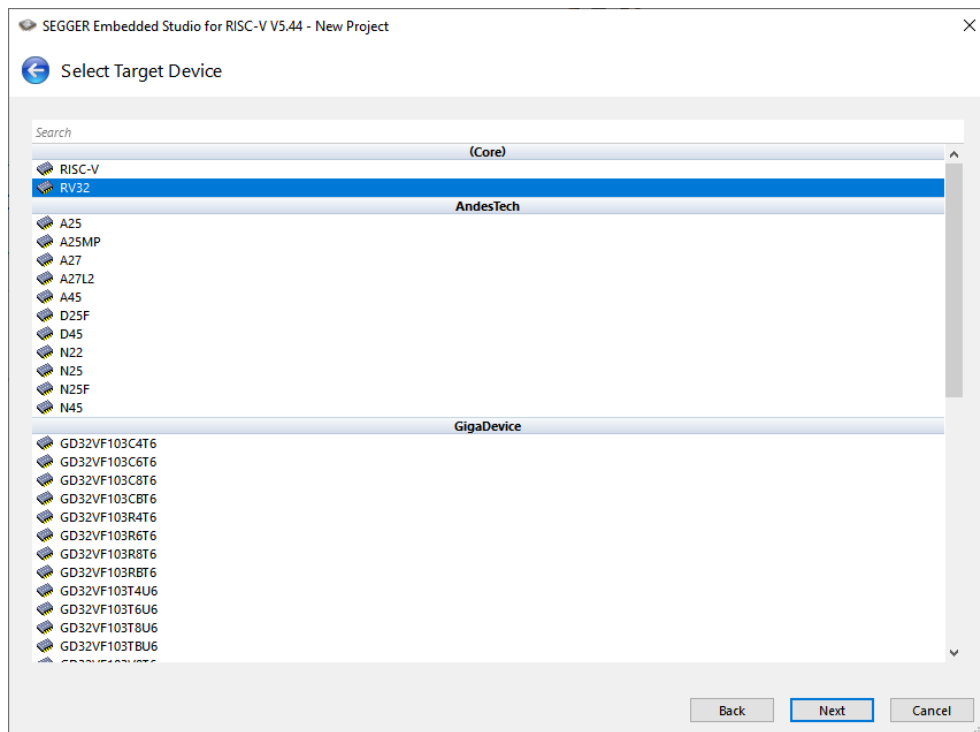


Figure 6-5 Embedded Studio, RV32

7. And for the project setting, please change the Link and Section Placement to "GNU-LD Flash Placement", as shown in Figure 6-6.

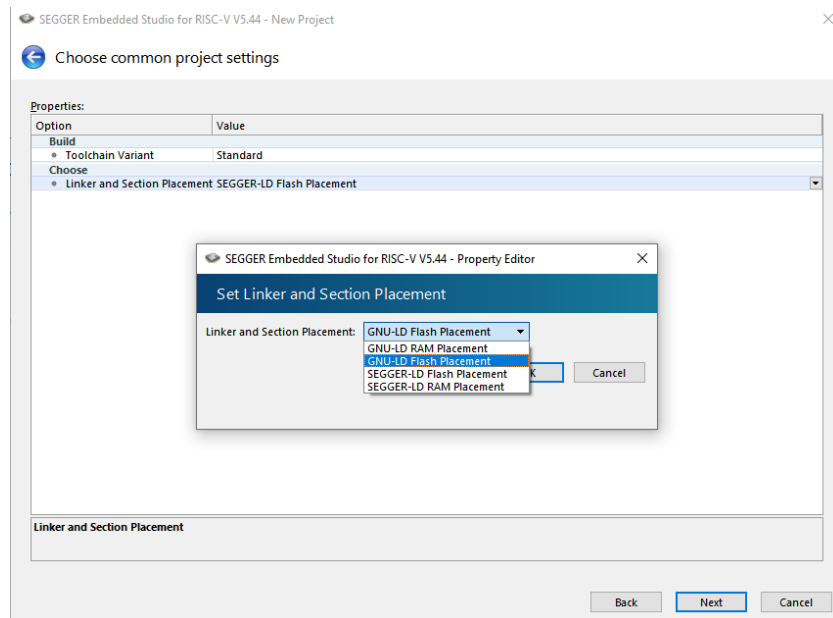


Figure 6-6 Embedded Studio, Linker and Section Placement

PulseRain GRV3000D – Programmer's Guide

8. For the rest of the dialogues, click "Next" and then "Finish" to complete the project setup.

At this point, a solution called "hello_world" should have been created under the Embedded Studio, with a project also called "hello_world" under the solution.

Before we proceed to compile and debug, some of the project options need to be changed to match the GRV3000D's memory layout (The GRV3000D has a Harvard architecture). The details are as following:

1. Use mouse to right click the "hello_world" project, and choose "Options...", as shown in Figure 6-7.

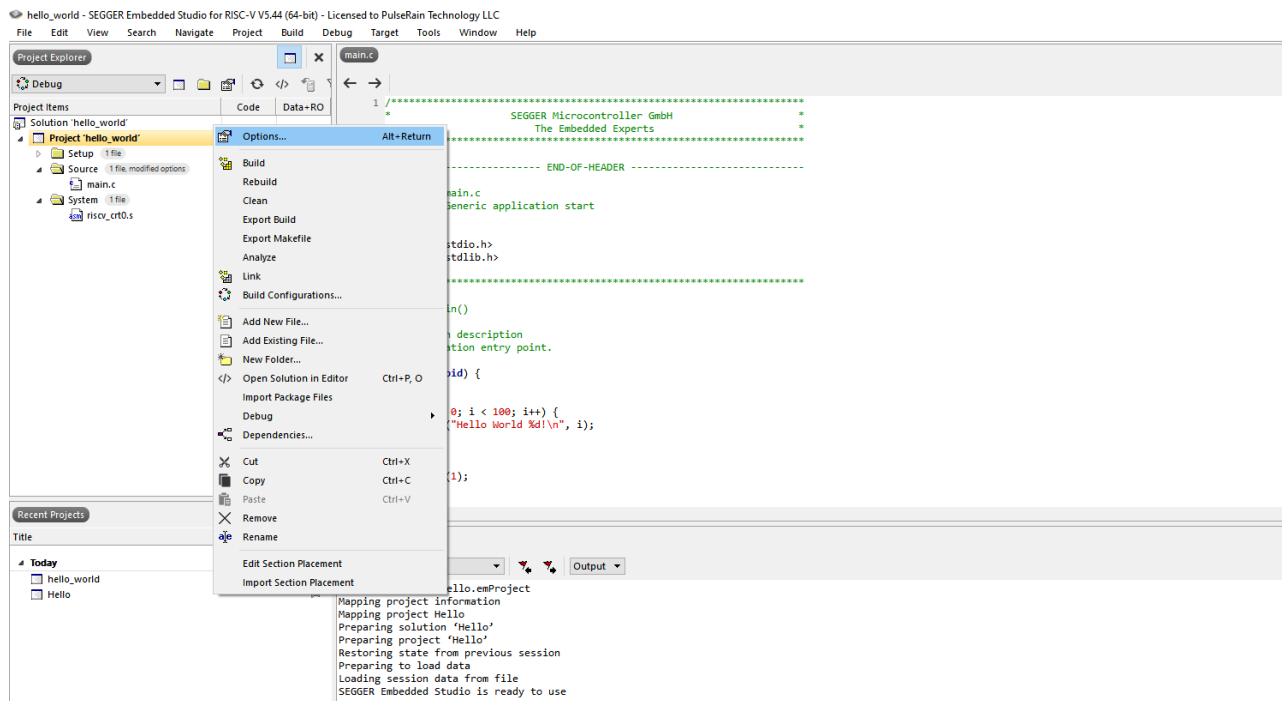


Figure 6-7 Embedded Studio, Project Options

2. And the following options need to be modified:

- Under Code Generation, set RISC-V ABI to "ilp32", set RISC-V ISA to "rv32i", as shown in Figure 6-8.
- Under Debugger, set Target Connection to "J-Link", as shown in Figure 6-8.
- Under Linker, set the Memory Segments to "FLASH1 RWX 0x80000000 0x00010000;RAM1 RWX 0xC0000000 0x00010000", as shown in Figure 6-9. This memory segment matches the memory allocation defined in Table 3-6.
- As illustrated in Figure 6-9, under Linker, set the Section Placement File to the one shown in List 6-1. Basically, this xml file will put .text and .init sections into the I-TCM, and put the rest sections into D-TCM.

And the same xml file can also be found on GitHub path:

https://github.com/PulseRain/GRV3000D_Software/raw/main/Segger/Embedded_Studio/GRV3000D_flash_placement_riscv.xml

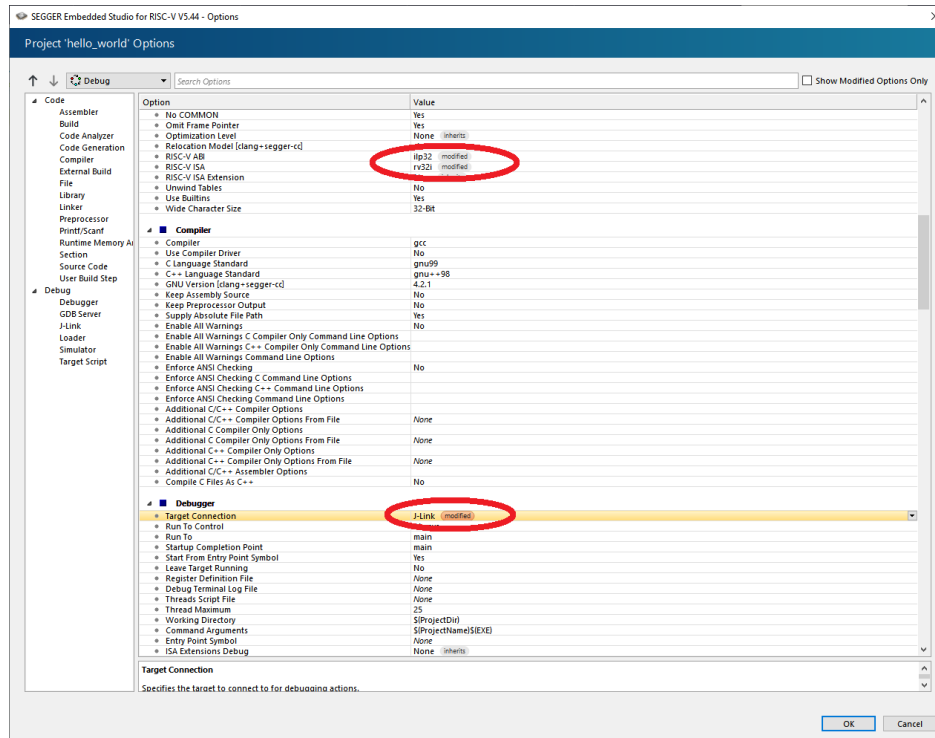


Figure 6-8 Embedded Studio, RISC-V ABI, RISC-V ISA and Target Connection

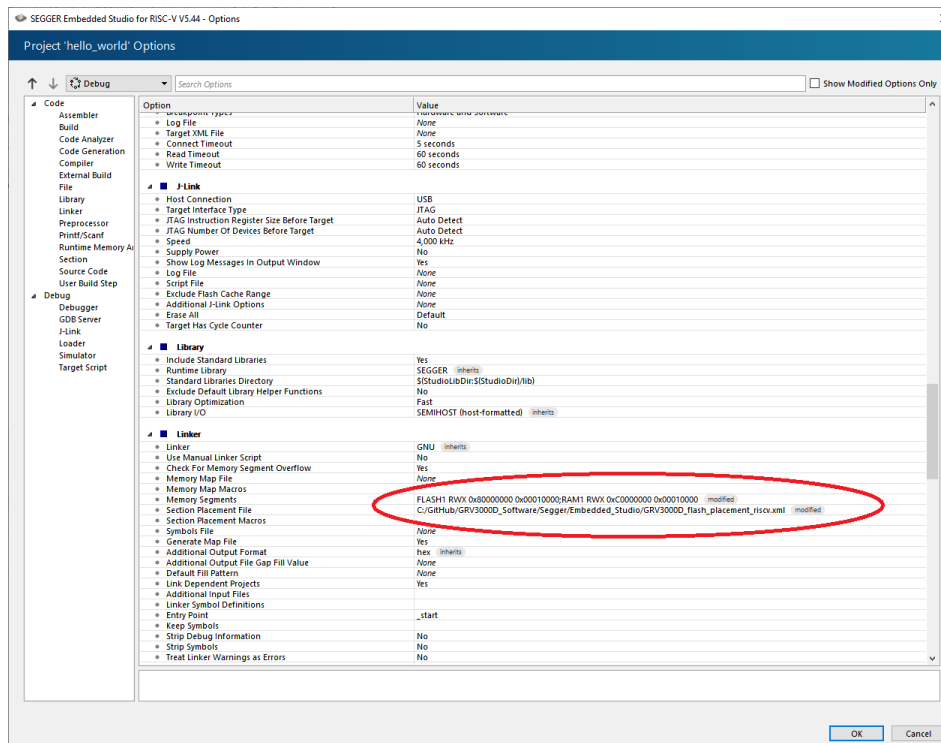


Figure 6-9 Embedded Studio, Linker Options

PulseRain GRV3000D – Programmer's Guide

```
<!DOCTYPE Linker_Placement_File>
<Root name="Flash Section Placement">
  <MemorySegment name="FLASH;FLASH1">
    <ProgramSection load="Yes" name=".vectors" start="" />
    <ProgramSection alignment="4" load="Yes" name=".init" />
    <ProgramSection alignment="4" load="Yes" name=".text" />
  </MemorySegment>
  <MemorySegment name="SRAM;RAM1">
    <ProgramSection alignment="4" load="Yes" name=".init_rodata" />
    <ProgramSection alignment="4" load="Yes" name=".dtors" />
    <ProgramSection alignment="4" load="Yes" name=".ctors" />
    <ProgramSection alignment="4" load="Yes" name=".rodata" />
    <ProgramSection alignment="4" load="Yes" name=".srodata" />
    <ProgramSection alignment="4" load="Yes" name=".eh_frame" keep="Yes" />
    <ProgramSection alignment="4" load="Yes" name=".gcc_except_table" />
    <ProgramSection alignment="4" load="Yes" name=".fast" />
    <ProgramSection alignment="4" load="Yes" name=".data" />
    <ProgramSection alignment="4" load="Yes" name=".tdata" />
    <ProgramSection alignment="4" load="Yes" name=".sdata" inputsections="*(.sdata.*
.sdata2.*)" />
    <ProgramSection alignment="4" load="No" name=".bss" />
    <ProgramSection alignment="4" load="No" name=".tbss" />
    <ProgramSection alignment="4" load="No" name=".sbss" />
    <ProgramSection alignment="4" load="No" name=".non_init" />
    <ProgramSection alignment="16" size="__HEAPSIZE__" load="No" name=".heap" />
    <ProgramSection alignment="4" size="__STACKSIZE__" load="No"
place_from_segment_end="Yes" name=".stack" />
  </MemorySegment>
</Root>
```

List 6-1 GRV3000D_flash_placement_riscv.xml

After everything is set, use menu Build / Rebuild hello_world (Alt+F7) to build the project and generate the correspondent .elf file.

6.2.3 Debug under the Embedded Studio for RISC-V

Once the project is built successfully, the software developers can load the elf file onto the Dev Kit through JTAG probe, simply by pressing F5 (or Menu Debug / Go), as illustrated in Figure 6-10.

And then the code can be traced through either "Step Into (F11)" or "Step Over (F10)". If everything falls into place, the hello_world project will print out the "Hello World" message to the Debug Terminal through Semi Host (a way to pass message through JTAG), as illustrated in Figure 6-11.

PulseRain GRV3000D – Programmer's Guide

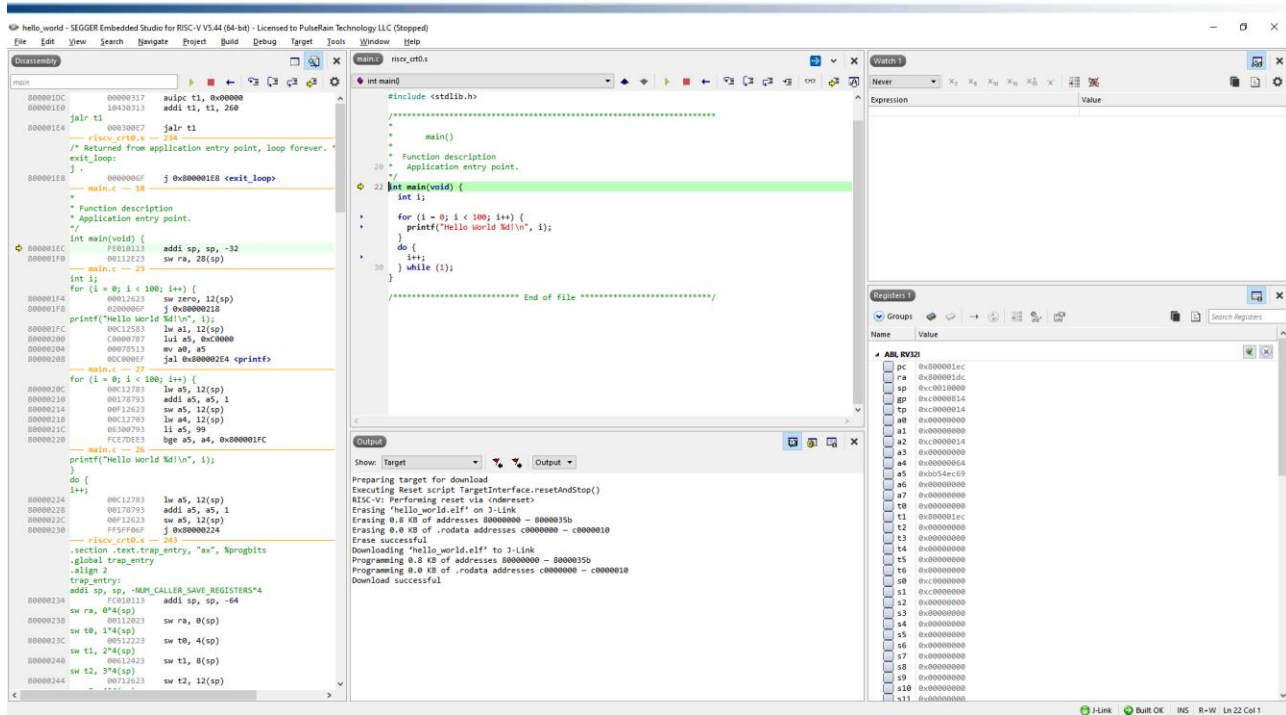


Figure 6-10 Embedded Studio, Debugger

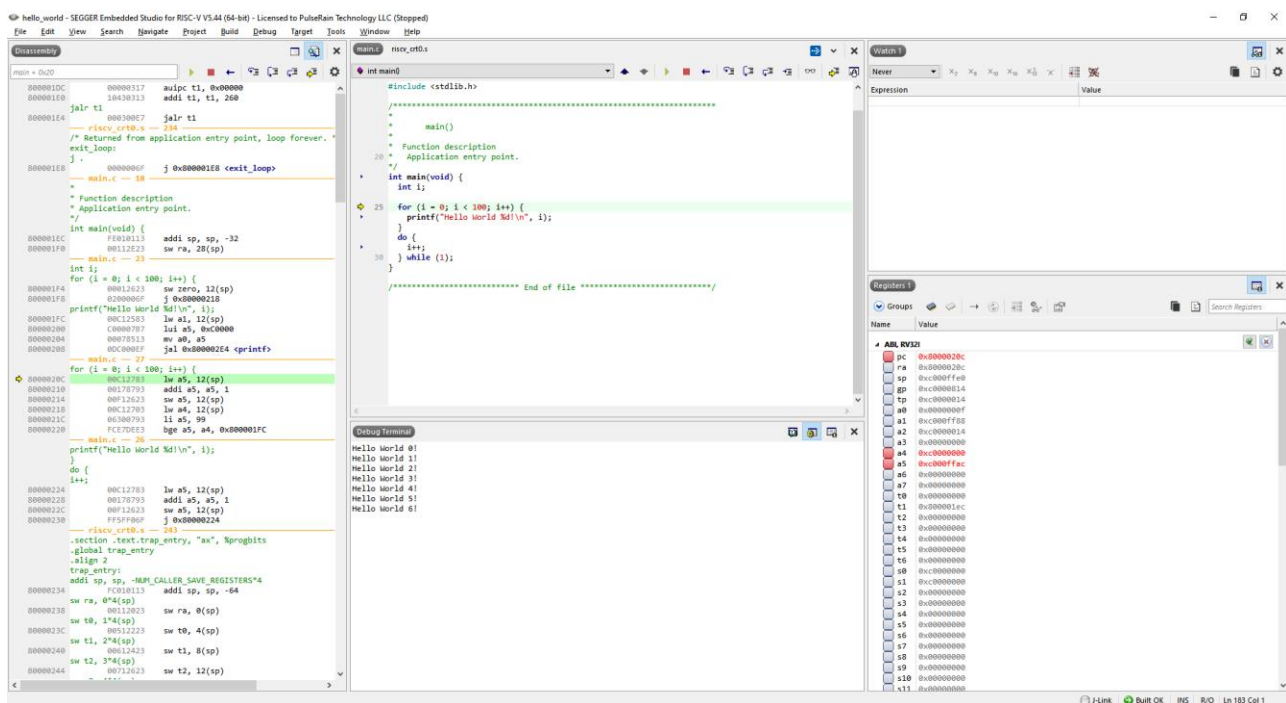


Figure 6-11 Embedded Studio, "Hello World"

7 Software – DSP (Digital Signal Processing)

PulseRain GRV3000D is in fact a DSC (Digital Signal Controller). As its name suggests, the GRV3000D can process digital signal in an accelerated fashion.

Traditionally, DSP acceleration is achieved by expanding the instruction set. The drawback of expanding instruction set is that it is an intrusive approach for the software developers, as the compiler has to be customized to produce those new instructions.

So instead of creating new instructions for DSP extension, the GRV3000D adopts a nimble approach by taking advantage of the HINT instructions already defined by RISC-V spec (Ref [1]). In this way, the DSP acceleration is non-intrusive and it does not rely on any specific compilers to function.

7.1 Introduction to HINT Instructions

As mentioned in Section 3.2, among the 32 GPRs, x0 is a read-only register and it always return zero. So, if an instruction has x0 as the destination register (rd), this instruction will not change any visible state architecturally, other than advancing the program counter.

In that light, any instruction with rd being zero can be used as NOP. However, in RISC-V Instruction Set (Ref [1]), only one of those instructions is defined as canonical NOP (namely ADDI x0, x0, #0). And the rest of them are called HINT instructions.

And according to Ref [1], 91% of the HINT space is reserved for standard HINTs, while the remainder of the HINT space is reserved for custom HINTs.

For the GRV3000D, it will recognize the custom HINT "SLTIU (rd=x0)" as the HINT for DSP acceleration. And for the rest of this document, "SLTIU (rd=x0)" will be called as **DSP HINT**.

7.2 Function Call with DSP HINT

Like most modular programming approach, Digital Signal Processing is usually put together as library function calls. For example, ARM's CMSIS (Cortex Microcontroller Software Interface Standard) library can offer a huge collection of DSP functions. And by the same token, PulseRain Technology also offers the PMSIS library (PulseRain Microcontroller Software Interface Standard) to realize a long list of DSP functions.

And for the GRV3000D processor, the PMSIS library has been optimized with the assistance of DSP HINT. Thus, on the GRV3000D, those PMSIS function calls will be executed in an accelerated fashion through DSP coprocessor. On the other hand, these PMSIS functions calls will be executed in a non-accelerated fashion when the coprocessor is disabled, or when they are running on a regular RISC-V processor.

In fact, as a dual-issue processor, the GRV3000D will merge JAL/JALR and DSP HINT into one mega operation, and throw it to its DSP coprocessor. As long as the software programmer correctly sets the DSP HINT next to

a JAL/JALR instruction, the DSP coprocessor will be triggered. For PMSIS functions, the function body can be viewed as a soft implementation of the same DSP function as the DSP coprocessor. The details are explained below:

7.2.1 Function Calls on RISC-V

For RISC-V, the function call always starts with JAL or JALR instruction. And per the RISC-V calling convention mentioned in Section 3.2, register x10 – x17 will be used to pass parameters to the function.

As illustrated in Figure 7-1, assuming the DSP coprocessor is available, if the software developer puts DSP HINT immediately after the JAL/JALR with DSP HINT carrying the same parameters as the function call, the DSP coprocessor will take over, skip the function call and carry out the same DSP function in an accelerated fashion. Otherwise, the function call will be executed in a regular fashion without going through coprocessor.

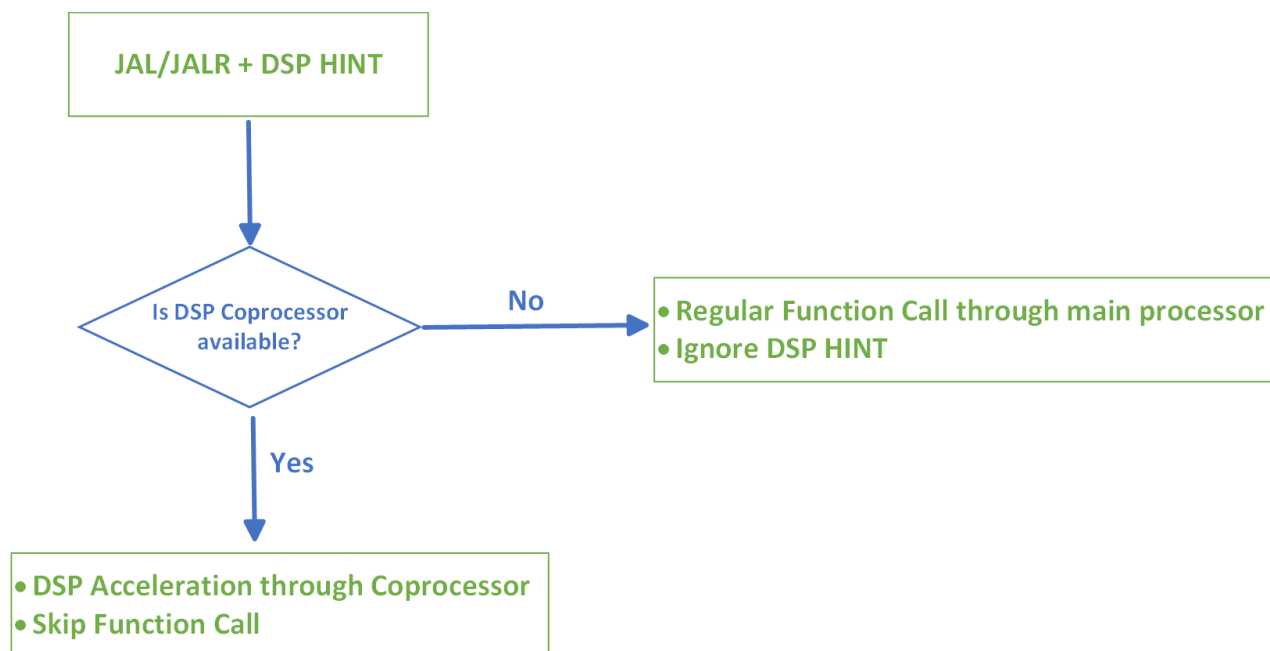


Figure 7-1 Function Call with DSP HINT

7.2.2 Format of the DSP HINT

To achieve DSP acceleration, the software developer should put 3 DSP HINT immediately after JAL/JALR. And as a dual issue processor, the GRV3000D will always put JAL/JALR instruction in issue slot A. If issue slot B happens to be a DSP HINT (DSP HINT 0), the DSP coprocessor will be chosen as the active function unit (Figure 2-1). Inside the DSP coprocessor, further parameters will be extracted from DSP HINT 1 and DSP HINT 2, as illustrated in Figure 7-2. And in Figure 7-2, it is assumed that RV32I is being used. That's why the DSP HINT 1 has an address of PC + 8.



Figure 7-2 JAL/JALR + 3 DSP HINT

As mentioned in Section 7.1, the GRV3000D uses "SLTIU (rd=x0)" as DSP HINT. In fact, the complete instruction format of DSP HINT is

SLTIU x0, rs1, imm12

For which rs1 is the source GPR index (0 ~ 31), and imm12 is a 12-bit immediate number. So, from DSP HINT 0 to DSP HINT 2, there are total of 6 configurable parameters. And they are defined below by DSP Coprocessor in Table 7-1:

DSP HINT #	rs1	Imm12 [11 : 0]
DSP HINT 0	Address of the GPR that stores the source buffer pointer	[9 : 0] DSP Command Index (See Table 7-2)
		[11 : 10] value size index (32 bit, 16 bit, 8 bit, See Table 7-3)
DSP HINT 1	DSP Function Dependent	[4 : 0] GPR address, DSP function dependent
		[11 : 5] parameter 1
DSP HINT 2	DSP Function Dependent	[4 : 0] GPR address, DSP function dependent
		[11 : 5] parameter 2

Table 7-1 Format of the DSP HINT

DSP Command Index	Description
10'd0	get exponent for normalization
10'd1	absolute value
10'd2	negative value
10'd3	clip the value to the range [-limit, limit]
10'd4	arithmetic shift right
10'd5	logic shift right
10'd6	logic shift left
10'd7	arithmetic shift right and round
10'd16	FIR filter
10'd17	BIQUAD filter

Table 7-2 DSP Command Index

Value Size Index	Description
2'd0	32 bit value. Each 32-bit memory word contains one value.
2'd1	16 bit value. Each 32-bit memory word contains two values
2'd2	8 bit value. Each 32-bit memory word contains four values.
2'd3	RESERVED

Table 7-3 DSP Value Size

7.3 DSP Command on PulseRain GRV3000D

The DSP commands supported by the Coprocessor are shown early in Table 7-2. Those commands can be divided into two categories:

1. Single source command, such as "absolute value".
2. Dual source command, such as FIR and BIQUAD

All the DSP commands work on buffers. And there are two constraints for the buffer starting address and length:

1. The buffer length should be in the power of 2, such as 1, 2, 4, 8, 16 And the length is in terms of 32-bit word.
2. The starting address should be aligned to the same power of 2 derived from the length.

7.3.1 DSP Command with Single Source

For those commands who only need one source buffer, the buffer pointer (source or destination) can point to any address as long as the address is aligned to power-of-2 boundary derived from the length. And the source and destination can be the same as well.

7.3.1.1 DSP Command: Exponent / Normalization

For a signed two's complement number x on the GRV3000D, its exponent is defined as:

$$Exponent = \begin{cases} \text{number of leading zeros minus one,} & \text{if } x \geq 0 \\ \text{number of leading ones minus one,} & \text{if } x < 0 \end{cases}$$

The normalization command will go through the whole source buffer, find the smallest exponent, and save it to the first 32-bit word designed by the destination pointer, as illustrated in Figure 7-3. The result can be used by the shift left command later to normalize the whole source buffer.

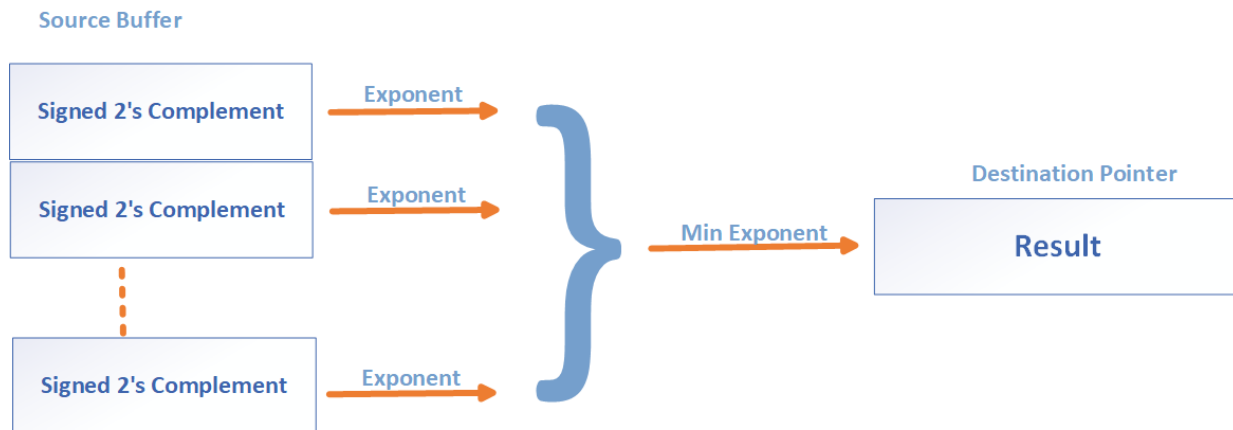


Figure 7-3 Exponent / Normalization

And the correspondent format for DSP HINT is shown below in Table 7-4.

DSP HINT #	rs1	Imm12 [11 : 0]
DSP HINT 0	Address of the GPR that stores the source buffer pointer	[9 : 0] 10'd0
		[11 : 10] value size index (32 bit, 16 bit, 8 bit, See Table 7-3)
DSP HINT 1	Address of the GPR that stores the destination buffer pointer	[4 : 0] Address to the GPR that stores the buffer length minus one.
		[11 : 5] Not Used
DSP HINT 2	Not Used	[4 : 0] Not Used
		[11 : 5] Not Used

Table 7-4 DSP HINT Format for Normalization

7.3.1.2 DSP Command: Absolute Value

This DSP command will turn every value in the source buffer into its absolute value, and write to the correspondent location in the destination buffer. However, for N bits of 2's complement, the most negative number is -2^{N-1} while the biggest positive number is $2^{N-1} - 1$. Thus, the most negative number will not get its correspondent absolute value correctly under this command. If this becomes a concern, the software developers can call the clip command prior to applying the absolute value.

The DSP HINT format for absolute value can be found in Table 7-5.

DSP HINT #	rs1	Imm12 [11 : 0]
DSP HINT 0	Address of the GPR that stores the source buffer pointer	[9 : 0] 10'd1
		[11 : 10] value size index (32 bit, 16 bit, 8 bit, See Table 7-3)
DSP HINT 1	Address of the GPR that stores the destination buffer pointer	[4 : 0] Address to the GPR that stores the buffer length minus one.
		[11 : 5] Not Used
DSP HINT 2	Not Used	[4 : 0] Not Used
		[11 : 5] Not Used

Table 7-5 DSP HINT Format for Absolute Value

PulseRain GRV3000D – Programmer's Guide

7.3.1.3 DSP Command: Negate

This DSP command will negate every value in the source buffer and write each result to the destination buffer. However, for N bits of 2's complement, the most negative number is -2^{N-1} while the biggest positive number is $2^{N-1} - 1$. Thus, the most negative number will not get its correspondent positive value correctly under this command. If this becomes a concern, the software developers can call the clip command prior to negating the value.

The DSP HINT format for negating value can be found in Table 7-6.

DSP HINT #	rs1	Imm12 [11 : 0]
DSP HINT 0	Address of the GPR that stores the source buffer pointer	[9 : 0] 10'd2
		[11 : 10] value size index (32 bit, 16 bit, 8 bit, See Table 7-3)
DSP HINT 1	Address of the GPR that stores the destination buffer pointer	[4 : 0] Address to the GPR that stores the buffer length minus one.
		[11 : 5] Not Used
DSP HINT 2	Not Used	[4 : 0] Not Used
		[11 : 5] Not Used

Table 7-6 DSP HINT Format for Negating

7.3.1.4 DSP Command: Clipping

This DSP command will clip every value x in the source buffer between $-limit$ and $limit$, as shown below:

$$Clipping(x, limit) \begin{cases} limit, & \text{if } x \geq limit \\ x, & \text{if } -limit < x < limit \\ -limit, & \text{if } x \leq -limit \end{cases}$$

And the DSP HINT format for clipping can be found in Table 7-7.

DSP HINT #	rs1	Imm12 [11 : 0]
DSP HINT 0	Address of the GPR that stores the source buffer pointer	[9 : 0] 10'd3
		[11 : 10] value size index (32 bit, 16 bit, 8 bit, See Table 7-3)
DSP HINT 1	Address of the GPR that stores the destination buffer pointer	[4 : 0] Address to the GPR that stores the buffer length minus one.
		[11 : 5] Not Used
DSP HINT 2	Address of the GPR that stores the value of limit	[4 : 0] Not Used
		[11 : 5] Not Used

Table 7-7 DSP HINT Format for Clipping

7.3.1.5 DSP Command: Arithmetic Shift Right

This DSP command will shift every value in the source buffer to the right by "shamt" bits, sign-extend the result and then write to the destination buffer.

The DSP HINT format of arithmetic shift right can be found in Table 7-8.

PulseRain GRV3000D – Programmer's Guide

DSP HINT #	rs1	Imm12 [11 : 0]
DSP HINT 0	Address of the GPR that stores the source buffer pointer	[9 : 0] 10'd4
		[11 : 10] value size index (32 bit, 16 bit, 8 bit, See Table 7-3)
DSP HINT 1	Address of the GPR that stores the destination buffer pointer	[4 : 0] Address to the GPR that stores the buffer length minus one.
		[11 : 5] Not Used
DSP HINT 2	Address of the GPR that stores the shamt	[4 : 0] Not Used
		[11 : 5] Not Used

Table 7-8 DSP HINT Format for Arithmetic Shift Right

7.3.1.6 DSP Command: Logic Shift Right

This DSP command will shift every value in the source buffer to the right by "shamt" bits without sign-extending, and then write the result to the destination buffer.

The DSP HINT format for logic shift right can be found in Table 7-9.

DSP HINT #	rs1	Imm12 [11 : 0]
DSP HINT 0	Address of the GPR that stores the source buffer pointer	[9 : 0] 10'd5
		[11 : 10] value size index (32 bit, 16 bit, 8 bit, See Table 7-3)
DSP HINT 1	Address of the GPR that stores the destination buffer pointer	[4 : 0] Address to the GPR that stores the buffer length minus one.
		[11 : 5] Not Used
DSP HINT 2	Address of the GPR that stores the shamt	[4 : 0] Not Used
		[11 : 5] Not Used

Table 7-9 DSP HINT Format for Logic Shift Right

7.3.1.7 DSP Command: Shift Left

This DSP command will shift every value in the source buffer to the left by "shamt" bits, and then write the result to the destination buffer.

The DSP HINT format for shift left can be found in Table 7-10.

DSP HINT #	rs1	Imm12 [11 : 0]
DSP HINT 0	Address of the GPR that stores the source buffer pointer	[9 : 0] 10'd6
		[11 : 10] value size index (32 bit, 16 bit, 8 bit, See Table 7-3)
DSP HINT 1	Address of the GPR that stores the destination buffer pointer	[4 : 0] Address to the GPR that stores the buffer length minus one.
		[11 : 5] Not Used
DSP HINT 2	Address of the GPR that stores the shamt	[4 : 0] Not Used
		[11 : 5] Not Used

Table 7-10 DSP HINT Format for Shift Left

PulseRain GRV3000D – Programmer's Guide

7.3.1.8 DSP Command: Arithmetic Shift Right and Round

This DSP command will shift every value in the source buffer to the right by "shamt" bits arithmetically, round the results to bit zero, and then write to the destination buffer.

The DSP HINT format for arithmetic shift right and rounding can be found in Table 7-11.

DSP HINT #	rs1	Imm12 [11 : 0]
DSP HINT 0	Address of the GPR that stores the source buffer pointer	[9 : 0] 10'd6
		[11 : 10] value size index (32 bit, 16 bit, 8 bit, See Table 7-3)
DSP HINT 1	Address of the GPR that stores the destination buffer pointer	[4 : 0] Address to the GPR that stores the buffer length minus one.
		[11 : 5] Not Used
DSP HINT 2	Address of the GPR that stores the shamt	[4 : 0] Not Used
		[11 : 5] Not Used

Table 7-11 DSP HINT Format for Arithmetic Shift Right and Rounding

7.3.2 DSP Command with Dual Sources

As mentioned in Section 3.4.2, the D-TCM of the GRV3000D is divided into two halves: the top half and the bottom half. Each half has a 64-bit bus for read/write. For those DSP commands who need two sources (FIR/BIQUAD), one of the sources should be in the top half while the other should be in the bottom half. In this way, the coprocessor's dual MAC structure can be fully utilized to achieve maximum acceleration for digital signal processing. And both source pointers should be aligned to the power of 2 boundary derived from buffer length.

7.3.2.1 FIR (Finite Impulse Response) Filter

The DSP command for FIR filter needs two source buffers:

1. The buffer that holds all the samples (sample buffer)
2. The buffer that holds the coefficients

The sample buffer can be any size as long as it is a power of 2. So is the number of the coefficients (the order of FIR filter). And the sizes of the two buffers are not necessarily the same.

As illustrated in Figure 7-4, the sample buffer and the coefficient buffer should reside separately by the top half / bottom half structure. And the dual MACs shown in Figure 7-4 have extra 8 guard bits. In other words, the accumulator of the dual MAC has a total bit width of 72.

And for this DSP command, the source buffer of sample and the destination buffer are the same. In other words, the sample buffer will be overwritten by the filter results after this DSP command is completed.

Because of that, each 72-bit accumulated result will be arithmetically shifted to the right by "shamt" bits, round, and then save the lower 32 bits to the destination (namely the source sample buffer).

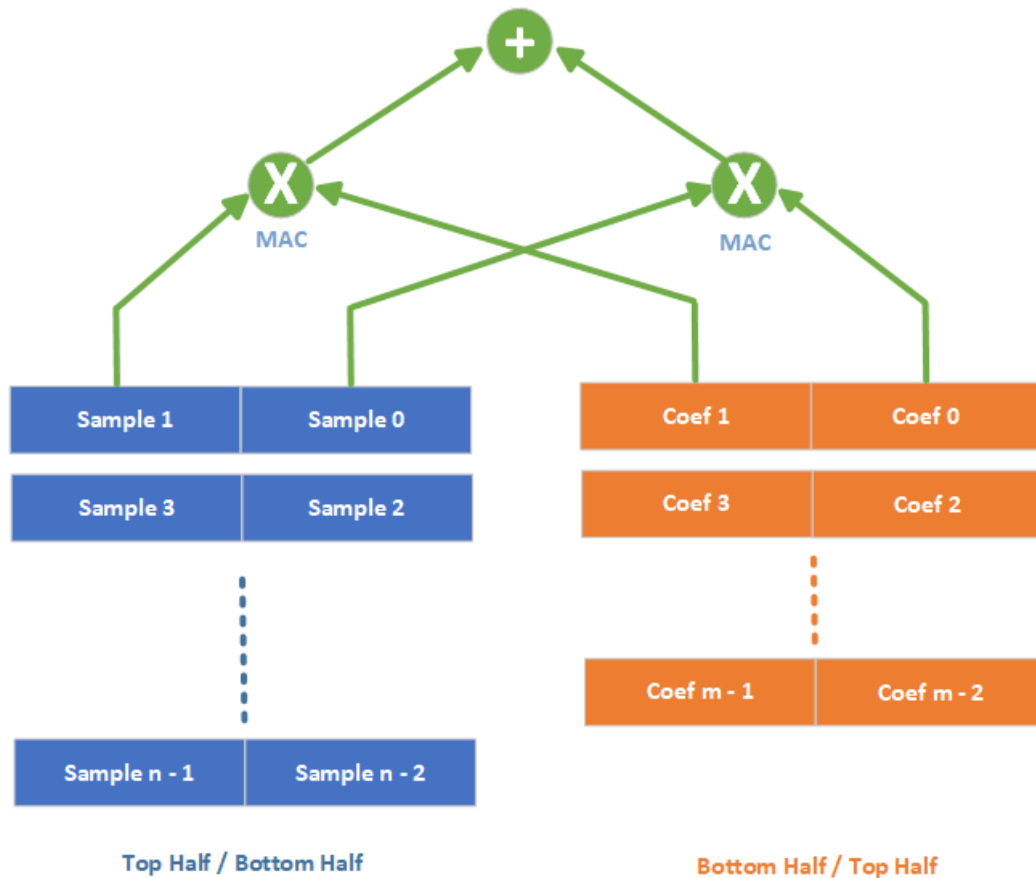


Figure 7-4 FIR Filter

So mathematically, the filter result will be calculated as the following:

$$Result(i) = round((\sum_{j=0}^{m-1} coef(j) * sample(i + j)) \gg shamt), i \in [0, n - 1]$$

And the DSP HINT format of FIR filter can be found in Table 7-12.

DSP HINT #	rs1	Imm12 [11 : 0]
DSP HINT 0	Address of the GPR that stores the source sample buffer pointer (also the destination)	[9 : 0] 10'd16
		[11 : 10] always zero
DSP HINT 1	Address of the GPR that stores the coef buffer size minus one.	[4 : 0] Address to the GPR that stores the sample buffer length minus one.
		[11 : 5] Not Used
DSP HINT 2	Address of the GPR that stores the source coefficient buffer pointer.	[4 : 0] Address of the GPR that stores the "shamt"
		[11 : 5] Not Used

Table 7-12 DSP HINT Format for FIR Filter

PulseRain GRV3000D – Programmer's Guide

7.3.2.2 BIQUAD Filter

The mathematic background for BIQUAD filter can be found in Appendix. And high order IIR (Infinite Impulse Response) filters are often implemented as cascading of BIQUAD filters. For a BIQUAD filter with normalized coefficients, it will be calculated as

$$y(i) = y(i-1) * a1^{(0)} + y(i-2) * a2^{(0)} + x(i) * b0^{(0)} + x(i-1) * b1^{(0)} + x(i-2) * b2^{(0)}$$

Equation 7-1

And if another BIQUAD is cascaded to it, it will be like

$$z(i) = z(i-1) * a1^{(1)} + z(i-2) * a2^{(1)} + y(i) * b0^{(1)} + y(i-1) * b1^{(1)} + y(i-2) * b2^{(1)}$$

Equation 7-2

Here $a1^{(n)}, a2^{(n)}, b0^{(n)}, b1^{(n)}, b2^{(n)}$ are the 5 coefficients for n th BIQUAD.

For BIQUAD filter, the software developer needs to allocate a sample buffer with a size as power of 2. The reason is that for BIQUAD filters, the sample buffer will store not only the input sample $x(i)$, but also the internal state like $y(i)$ and $z(i)$. So the sample buffer will be arranged like a circular buffer, for which the input samples will be placed in the ascending order, while the internal states will be stored in the descending order. When more input samples come in, the old input samples will be overwritten by the internal states. That's why the sample buffer size has to be power of 2, so the index address can wrap naturally.

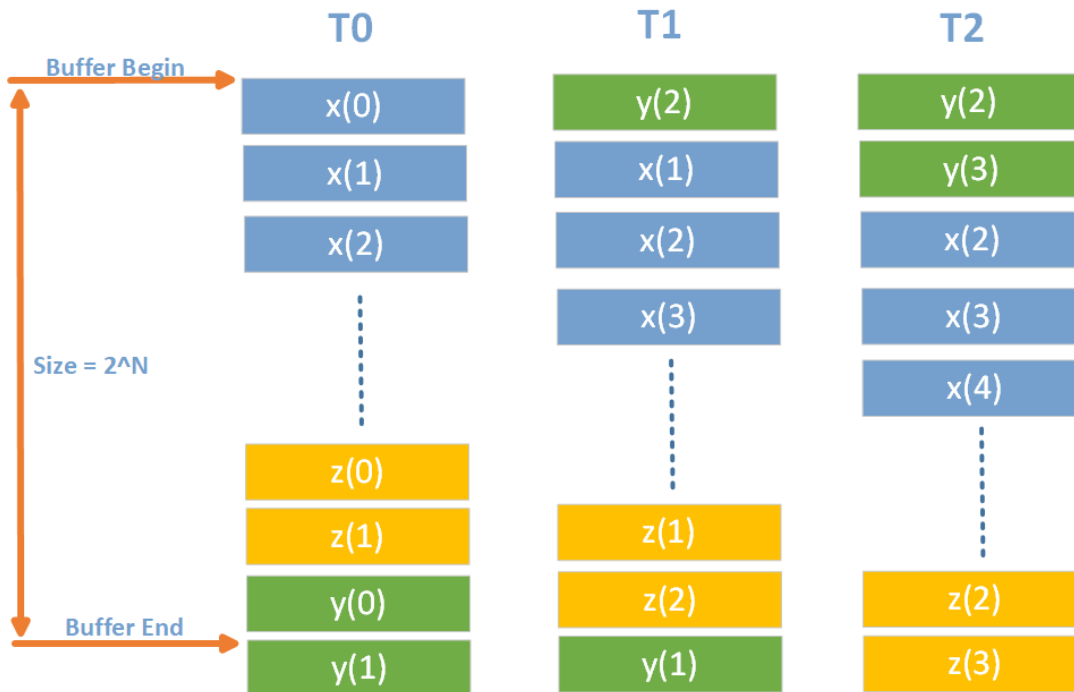


Figure 7-5 BIQUAD Filter, Sample Buffer Memory Layout

PulseRain GRV3000D – Programmer's Guide

As illustrated by the two BIQUAD cascading example in Figure 7-5, the newer samples and states are always stored in address ascending order until the address wraps.

At T0 the input samples are placed at the beginning of the sample buffer in address ascending order ($x(0)$, $x(1)$ and $x(2)$), and the internal states are stored in the opposite direction ($y(0)$, $y(1)$ and $z(0)$, $z(1)$). Their addresses are wrapped naturally to the end of the sample buffer.

At T1, the oldest input sample $x(0)$ is overwritten by the newest y state $y(2)$, while the oldest y state is overwritten by the newest z state $z(2)$.

And at T2, the oldest input sample $x(1)$ is overwritten by the newest y state $y(3)$, while the oldest y state is overwritten by the newest z state $z(3)$.

Overall, each BIQUAD will have two variables as internal state. As long as the sample buffer size is big enough, the GRV3000D's DSP coprocessor can handle any number of BIQUADs in an accelerated fashion. With dual MAC, each BIQUAD takes 3 clock cycles to finish. So for n input samples and m BIQUADs, it takes a total of $3 \cdot n \cdot m$ clock cycles.

Like FIR, the coefficients of the BIQUAD filters should be stored in a coefficient buffer. And the coefficient buffer and the sample buffer should reside separately by the top half / bottom half structure. For each BIQUAD filter, it needs 5 coefficients mathematically, as explained in Equation 7-1 and Equation 7-2. And to match the sample buffer and dual MAC, a zero should be padded to those 5 coefficients. Thus the coefficient buffer should look like the one shown in Figure 7-6.

The coefficients in Figure 7-6 are always 32-bit. As mentioned early, the accumulator of the dual MAC has a total bit width of 72. For each BIQUAD, the 72-bit accumulated result will be arithmetically shifted to the right by "shamt" bits, round, and then save the lower 32 bits to the sample buffer.

And the DSP HINT format of BIQUAD filter can be found in Table 7-13.

DSP HINT #	rs1	Imm12 [11 : 0]
DSP HINT 0	Address of the GPR that stores the sample buffer pointer	[9 : 0] 10'd17
		[11 : 10] always zero
DSP HINT 1	Address of the GPR that stores the number of available input samples.	[4 : 0] Address to the GPR that stores the sample buffer length minus one.
		[11 : 5] Not Used
DSP HINT 2	Address of the GPR that stores the coefficient buffer pointer.	[4 : 0] Address of the GPR that stores the "shamt"
		[11 : 5] the total number of BIQUAD

Table 7-13 DSP HINT Format for BIQUAD Filter

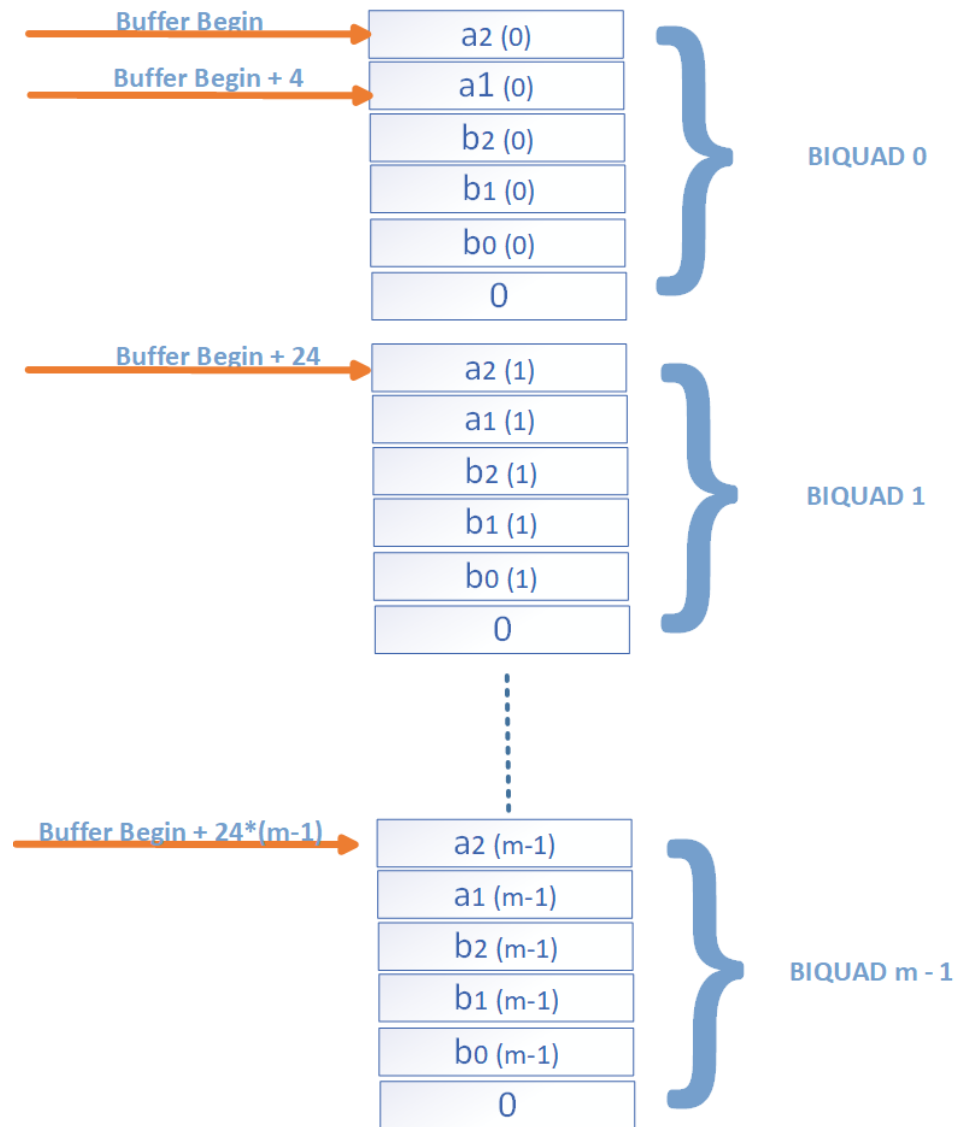


Figure 7-6 Coefficients for BIQUAD Filter

7.3.2.3 Dot Product (Real and Complex)

The DSP command for Dot Complex FIR filter needs two source buffers:

1. Buffer X
2. Buffer Y

PulseRain GRV3000D – Programmer's Guide

The length of those two buffers should be the same. And the length can be any size as long as it is a power of 2. For dot product of real numbers, it will be carried out two products per clock cycle, as illustrated in Figure 7-7. The result from the MAC will be shifted and saved back to buffer X as the final result.

For complex numbers, the real part should be stored in the word of even address, while the imaginary part should be stored in the word of odd address, as illustrated in Figure 7-8. And it will take 2 clock cycles to complete a complex number. The result from the two MAC will be summed together, shifted and saved back to buffer X as the final result.

And the DSP HINT format for Real Number Dot Product can be found in Table 7-14. The DSP HINT format for Complex Number Dot Product can be found in Table 7-15.

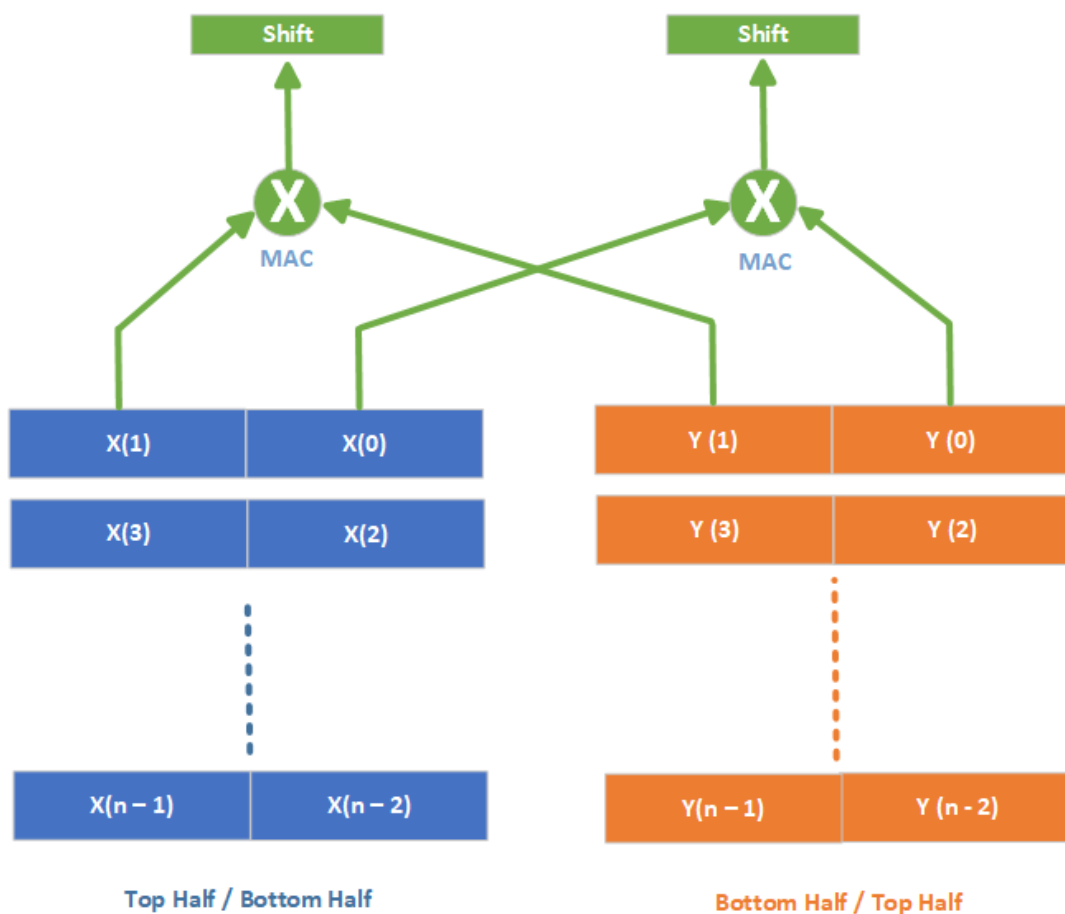


Figure 7-7 Dot Product for Real Numbers

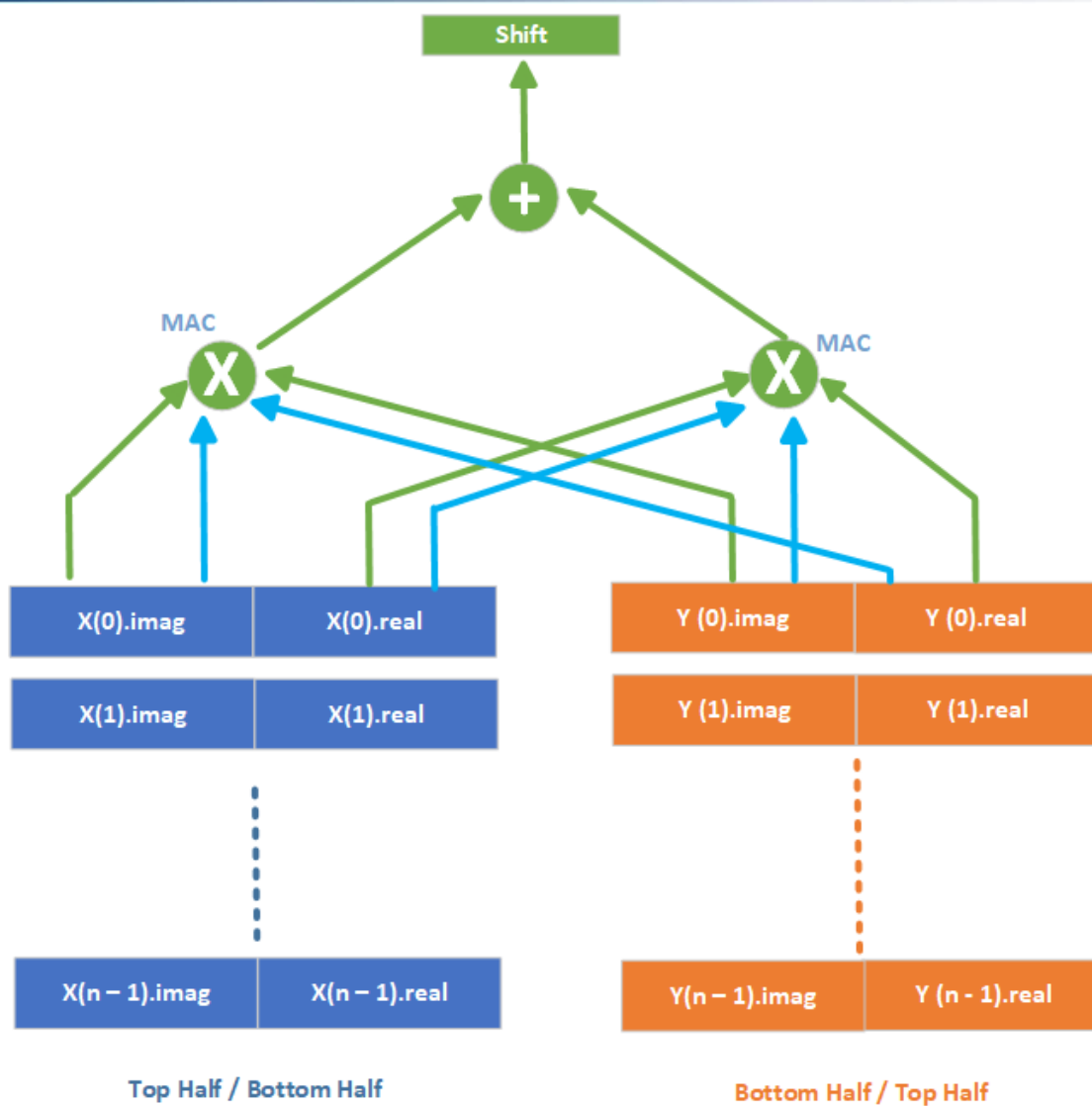


Figure 7-8 Dot Product for Complex Numbers

DSP HINT #	rs1	Imm12 [11 : 0]
DSP HINT 0	Address of the GPR that stores the buffer pointer for X (also the destination)	[9 : 0] 10'd20
		[11 : 10] always zero
DSP HINT 1	Address of the GPR that stores the buffer size minus one. (In number of 32-bit words)	Always zero
DSP HINT 2	Address of the GPR that stores the buffer pointer for Y.	[4 : 0] Address of the GPR that stores the "shamt"
		[11 : 5] Not Used

Table 7-14 DSP HINT Format for Real Number Dot Product

PulseRain GRV3000D – Programmer's Guide

DSP HINT #	rs1	Imm12 [11 : 0]
DSP HINT 0	Address of the GPR that stores the buffer pointer for X (also the destination)	[9 : 0] 10'd21
		[11 : 10] always zero
DSP HINT 1	Address of the GPR that stores the buffer size minus one. (In number of 32-bit words)	Always zero
DSP HINT 2	Address of the GPR that stores the buffer pointer for Y.	[4 : 0] Address of the GPR that stores the "shamt"
		[11 : 5] Not Used

Table 7-15 DSP HINT Format for Complex Number Dot Product

7.3.2.4 Fast Fourier Transform (FFT)

The GRV3000D has built-in support for the Fast Fourier Transform (FFT). FFT is a very computation intensive operation, and it usually breaks down to the operations of butterfly calculation (BF) and twiddle factor multiplication (TFM). There are multiple flavors for FFT, such as DIT (Decimation in Time), DIF (Decimation in Frequency) etc. In this regard, the GRV3000D recommends the following:

Radix – 2² Decimation in Frequency (DIF) FFT

The math theory behind this can be found in the following paper

Efficient FPGA implementation of FFT/IFFT Processor,

by Ahmed Saeed, M. Elbably, G. Abdelfadeel, and M. I. Eladawy,

International Journal of Circuits, Systems and Signal Processing, Issue 3, Volume 3, 2009

Basically, the idea is to break the FFT operation into the following two types:

- BT-I (Butterfly Type I) – Basic Butterfly and last quarter rotation
- BT-II (Butterfly Type II) – Basic Butterfly and TFM

In favor of flexibility, the GRV3000D will break the above into three hardware operations for acceleration:

1. Basic Butterfly Operation
2. BFI (Butterfly Type I, Basic Butterfly with last quarter rotation)
3. TFM (Twiddle Factor Multiplication)

Using the above three operations, an example of 16-point DIF FFT is illustrated in Figure 7-9. As $\log_4^{16} = 2$, there are only two stages in Figure 7-9. For the last stage in Figure 7-9, the TFM is not shown as all the Twiddle Factors are ones at that time.

7.3.2.4.1 Basic Butterfly Operation

If the FFT section size is N, the basic FFT operation will look like the one in Figure 7-10. The index is from 0 to N/2 – 1. Depending on the stage of the FFT, there could be multiple sections in that stage.

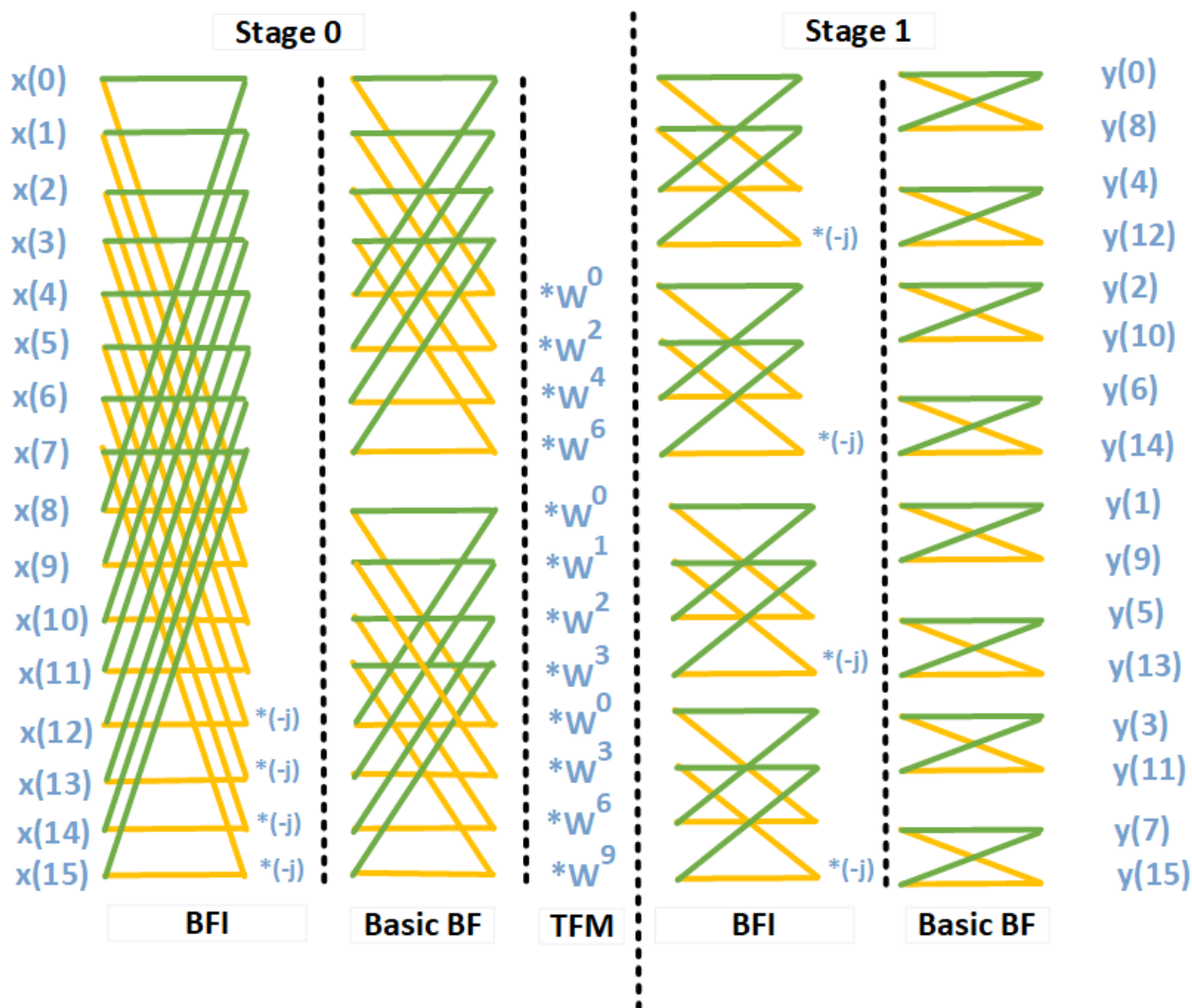


Figure 7-9 Radix – 2² DIF FFT for 16 Points

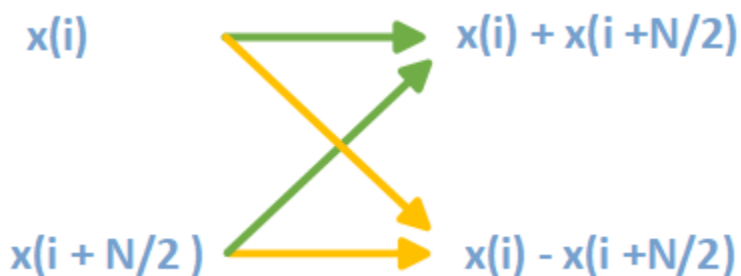


Figure 7-10 Basic Butterfly Operation for FFT

PulseRain GRV3000D – Programmer's Guide

7.3.2.4.2 Butterfly Type I

As mentioned early, Butterfly Type I is the combination of Basic Butterfly and last quarter rotation. Conceptually, after applying the basic butterfly operation, the last 1/4 in each section will be rotated clockwise by 90 degrees (multiply by -j), as illustrated by the example in Figure 7-9.

7.3.2.4.3 TFM (Twiddle Factor Multiplication)

For Radix – 2^2 DIF FFT, the twiddle factors for each section are defined as following:

(For FFT, $W = e^{-\frac{\pi}{N}}$, For IFFT, $W = e^{\frac{\pi}{N}}$)

1. For the first quarter, all twiddle factors are defined as one.
2. For the second quarter, all twiddle factors are defined as W^{i*2} ($i = 0, 1, 2, \dots$)
3. For the third quarter, all twiddle factors are defined as W^i ($i = 0, 1, 2, \dots$)
4. For the last quarter, all twiddle factors are defined as W^{i*3} ($i = 0, 1, 2, \dots$)

The first stage in Figure 7-9 shows the TFM for a section of size 16. For the GRV3000D, the Twiddle Factors need to be generated beforehand and saved in memory. The PMSIS library (See later chapters of this document) will provide a function for this purpose.

7.3.2.4.4 Complete FFT/IFFT

To put the above three operations together, the complete FFT/IFFT flow is illustrated in Figure 7-11. The difference between FFT and IFFT is mainly reflected in the Twiddle Factor Table. And the PMSIS library also provides functions for the complete implementation of Figure 7-11.

7.3.2.4.5 DSP HINT Format for FFT/IFFT

The DSP HINT format for BF-I and Basic Butterfly can be found in Table 7-16. For TFM operation, there is no correspondent DSP HINT instructions, as the TFM can be carried out by Dot Product. The PMSIS library also has functions to implement the TFM efficiently.

DSP HINT #	rs1	Imm12 [11 : 0]
DSP HINT 0	Address of the GPR that stores the buffer pointer (also the destination)	[9 : 0] 10'd21
		[11 : 10] always zero
DSP HINT 1	Address of the GPR that stores the number of sections	[4 : 0] Address to the GPR that stores the fft/fft flag. A value of one indicates IFFT operation.
		Always zero
DSP HINT 2	Not Used	[4 : 0] Not Used
		[11 : 5] 1 for BF Type I, 2 for Basic Butterfly

Table 7-16 DSP HINT Format for BF-I and Basic Butterfly

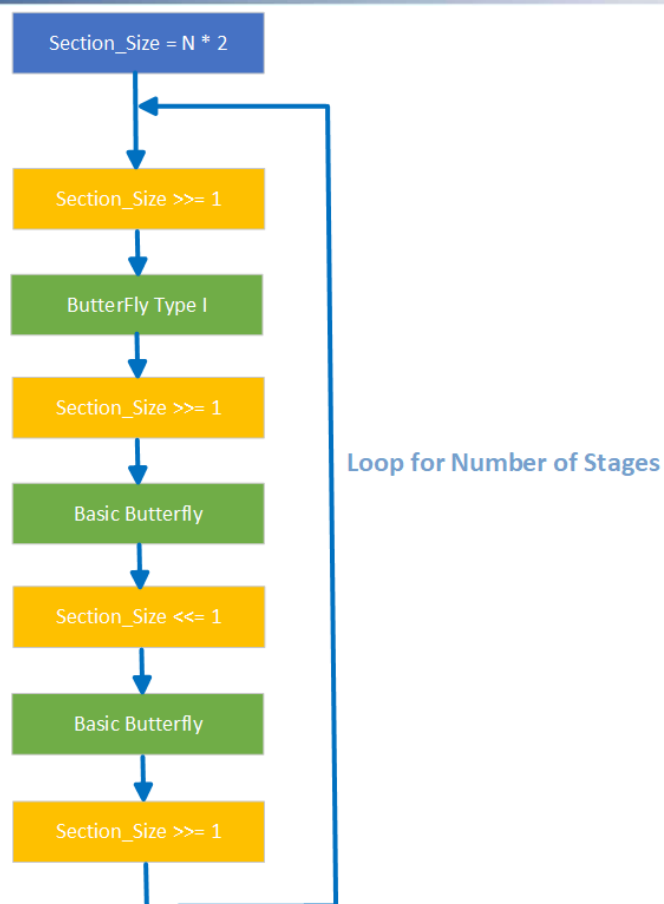


Figure 7-11 Complete FFT/IFFT Flow

7.3.2.4.6 The Performance of FFT on GRV3000D

Compared to other DSC (Digital Signal Controller) or DSP (Digital Signal Processor) on the market, the GRV3000D offers unrivaled performance for signal processing. In particular, Table 7-17 is a comparison of PulseRain GRV3000D's FFT performance against various ARM cores. (The ARM cortex performance metrics are from the white paper published by ARM Limited.). As we can see, the GRV3000D beats them hands down.

CFFT Q31	Block Size					
	32	64	128	256	512	1024
Cortex-M3	3374	6695	18549	36779	94267	187204
Cortex-M4	2577	5282	13823	28000	69253	139898
Cortex-M7	1497	3235	8050	17235	41076	87128
GRV3000D	395	874	1641	3912	7879	18662

Table 7-17 FFT Performance GRV3000D vs ARM Cortex

8 Software – PMSIS Library

To help users fully take advantage of the high-performance features in the GRV3000D, PulseRain Technology has provided an opensource software library called PMSIS (PulseRain Microcontroller Software Interface Standard). The PMSIS is released under MIT License, and can be found on GitHub at

<https://github.com/PulseRain/PMSIS>

It has macros for all the DSP HINT definitions mentioned in Chapter 7, and it also provides APIs for the following:

8.1 Exponent / Normalization

- `void pulserain_exp_q31 (q31_t *pSrc, q31_t *pExp, uint32_t blockSize)`

This function is the C wrapper for the Exponent / Normalization operation mentioned in Section 7.3.1.1. It will go through the whole source buffer, find the smallest exponent, and save it to the first 32-bit word designed by the destination pointer, as illustrated in Figure 7-3. Its parameters are defined in Table 8-1.

Parameters	Definition
pSrc	Pointer to the source buffer
pExp	Pointer to the 32 bit word that will hold the result
blockSize	The number of 32-bits words in the source buffer

Table 8-1 Parameters for Exponent / Normalization Operation

8.2 Absolute Value

- `void pulserain_abs_q31(q31_t *pSrc, q31_t *pDst, uint32_t blockSize)`

This function is the C wrapper for the absolute value operation mentioned in Section 7.3.1.2. It will turn every value in the source buffer into its absolute value, and write to the correspondent location in the destination buffer. Its parameters are defined in Table 8-2.

Parameters	Definition
pSrc	Pointer to the source buffer
pDst	Pointer to the destination buffer
blockSize	The number of 32-bits words in the source/destination buffer

Table 8-2 Parameters for Absolute Value Operation

8.3 Negate

- void pulserain_negate_q31(q31_t *pSrc, q31_t *pDst, uint32_t blockSize)

This function is the C wrapper for the negate operation mentioned in Section 7.3.1.3. It will negate every value in the source buffer and write each result to the destination buffer. Its parameters are defined in Table 8-3.

Parameters	Definition
pSrc	Pointer to the source buffer
pDst	Pointer to the destination buffer
blockSize	The number of 32-bits words in the source/destination buffer

Table 8-3 Parameters for Negate Operation

8.4 Clipping

- void pulserain_clip_q31(q31_t *pSrc, q31_t *pDst, uint32_t blockSize, q31_t limit)

This function is the C wrapper for the clipping operation mentioned in Section 7.3.1.4. It will clip every value x in the source buffer between $-limit$ and $limit$. Its parameters are defined in Table 8-4.

Parameters	Definition
pSrc	Pointer to the source buffer
pDst	Pointer to the destination buffer
blockSize	The number of 32-bits words in the source/destination buffer
limit	Clipping limit

Table 8-4 Parameters for Clipping Operation

8.5 Arithmetic Shift Right

- void pulserain_sra_q31(q31_t *pSrc, q31_t *pDst, uint32_t blockSize, uint32_t shamt)

This function is the C wrapper for the arithmetic shift right operation mentioned in Section 7.3.1.5. It will shift every value in the source buffer to the right by "shamt" bits, sign-extend the result and then write to the destination buffer. Its parameters are defined in Table 8-5.

Parameters	Definition
pSrc	Pointer to the source buffer

Parameters	Definition
pDst	Pointer to the destination buffer
blockSize	The number of 32-bits words in the source/destination buffer
shamt	Shift amount

Table 8-5 Parameters for Arithmetic Shift Right Operation

8.6 Logic Shift Right

- void pulserain_sll_q31(q31_t *pSrc, q31_t *pDst, uint32_t blockSize, uint32_t shamt)

This function is the C wrapper for the logic shift right operation mentioned in Section 7.3.1.6. It will shift every value in the source buffer to the right by "shamt" bits without sign-extending, and then write the result to the destination buffer. Its parameters are defined in Table 8-6.

Parameters	Definition
pSrc	Pointer to the source buffer
pDst	Pointer to the destination buffer
blockSize	The number of 32-bits words in the source/destination buffer
shamt	Shift amount

Table 8-6 Parameters for Logic Shift Right Operation

8.7 Arithmetic Shift Right and Round

- void pulserain_sra_rnd_q31(q31_t *pSrc, q31_t *pDst, uint32_t blockSize, uint32_t shamt)

This function is the C wrapper for the arithmetic shift right and round operation mentioned in Section 7.3.1.8. It will shift every value in the source buffer to the right by "shamt" bits arithmetically, round the results to bit zero, and then write to the destination buffer. Its parameters are defined in Table 8-7.

Parameters	Definition
pSrc	Pointer to the source buffer
pDst	Pointer to the destination buffer
blockSize	The number of 32-bits words in the source/destination buffer
Shamt	Shift amount

Table 8-7 Parameters for Arithmetic Shift Right and Round Operation

8.8 FIR Filter

- void pulserain_fir_q31(q31_t *pSampBuf, uint32_t sampBufLength, q31_t *pCoef, uint32_t numOfCoef, uint32_t shamt)

This function is the C wrapper for the FIR filter mentioned in Section 7.3.2.1. Its parameters are defined in Table 8-8.

Parameters	Definition
pSampBuf	Pointer to the sample buffer
sampBufLength	The number of samples in the sample buffer
pCoef	Pointer to the coefficients
numOfCoef	The number of coefficients
Shamt	Shift amount

Table 8-8 Parameters for FIR Filter

8.9 BIQUAD Filter

- void pulserain_X_biquad_q31(q31_t *pSampBuf, uint32_t numOfInputSample, uint32_t sampBufLen, q31_t *pCoef, uint32_t shamt)

(Here the X is the number of BIQUADs in the filter chain. X can be a number from 1 to 16.)

These functions are the C wrapper for the BIQUAD filter mentioned in 7.3.2.2. Its parameters are defined in Table 8-9.

Parameters	Definition
pSampBuf	Pointer to the sample buffer
numOfInputSample	The number of input samples
sampBufLen	The total size of the sample buffer, which should be a power of two. Please see Section 7.3.2.2 for more details.
pCoef	Pointer to the Coefficients. The layout of coefficients can be found in Section 7.3.2.2.
Shamt	Shift amount

Table 8-9 Parameters for BIQUAD Filter

8.10 Dot Product

- void pulserain_dot_product_q31_real(q31_t *pX, q31_t *pY, uint32_t bufLength, uint32_t shamt)
- void pulserain_dot_product_q31_complex(q31_t *pX, q31_t *pY, uint32_t bufLength, uint32_t shamt)

The above two functions are the C wrapper for the dot product operation mentioned in Section 7.3.2.3. As the function name suggests, one of the above two is for real number operation, while the other is for complex numbers.

The parameters for the above functions are defined in Table 8-10.

Parameters	Definition
pX	Pointer to buffer X
pY	Pointer to buffer Y
bufLength	The total size of the buffer, in number of 32-bit words.
Shamt	Shift amount

Table 8-10 Parameters for Dot Product Operation

8.11 FFT

- void pulserain_twiddle_gen (q31_t *pTwiddleTable, uint32_t log2N, q31_t scaleFactor, uint8_t ifft)
- void pulserain_FFT(q31_t *pBuffer, uint32_t log2N, q31_t *pTwiddleTable, uint32_t scaleShift, uint8_t ifft)

There are mainly two functions at the top level, which can be used to carry out the FFT operation.

1. The pulserain_twiddle_gen function can be used to generate twiddle table. Of course, users can also choose to generate their own twiddle table.
2. The pulserain_FFT function is the main function for FFT. And it needs a twiddle table as the input.

The parameters for pulserain_twiddle_gen can be found in Table 8-11.

Parameters	Definition
pTwiddleTable	Pointer to buffer that will hold the twiddle table
log2N	Logarithmic 2 of the FFT size N
scaleFactor	Scale Factor for the Twiddle Table elements
Ifft	One for IFFT operation, zero for FFT operation

Table 8-11 Parameters for Twiddle Table Generation

And the parameters for `pulserain_FFT` can be found in Table 8-12.

Parameters	Definition
<code>pBuffer</code>	Pointer to buffer that is for both source and destination
<code>log2N</code>	Logarithmic 2 of the FFT size N
<code>pTwiddleTable</code>	Pointer to the Twiddle Table
<code>scaleShift</code>	Shift amount to scale the result
<code>ifft</code>	One for IFFT operation, zero for FFT operation

Table 8-12 Parameters for FFT Operation

8.12 Compile and Self-Test

The details for PMSIS compiling and self-test can be found in <https://github.com/PulseRain/PMSIS#readme>

Appendix: Dev Kit for the GRV3000D

PulseRain GRV3000D can be evaluated on the Digilent Arty A7 – 100T FPGA board, as shown below:

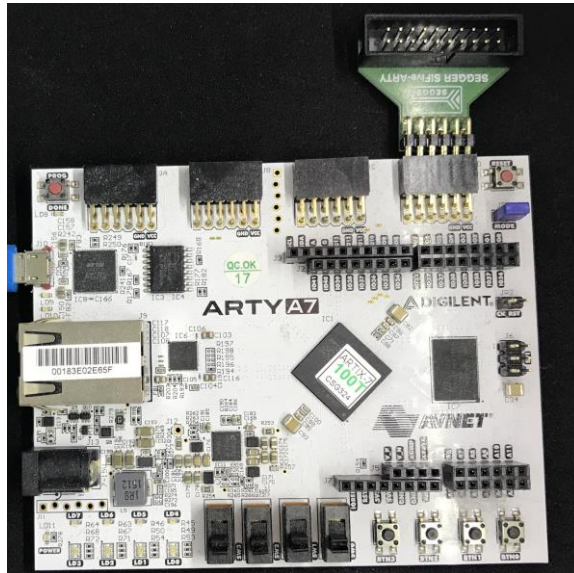


Figure Digilent Arty A7 – 100T FPGA Board

In the figure above, the pin head connector in the up-right corner is an adapter for Segger J-Link JTAG debug probe. This adapter (SEGGER SiFive-ARTY) can be purchased from the Segger website:

<https://www.segger.com/products/debug-probes/j-link/accessories/adapters/j-link-sifive-arty-adapter/>

And with this adapter, the Segger JLink can be attached to the Arty A7 board for debug, as illustrated below:

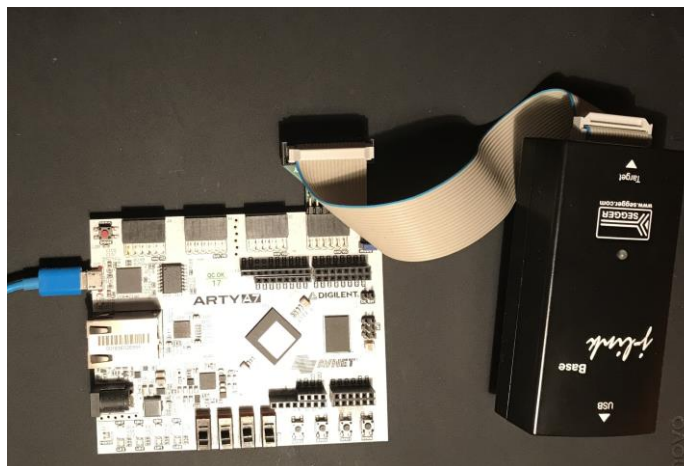
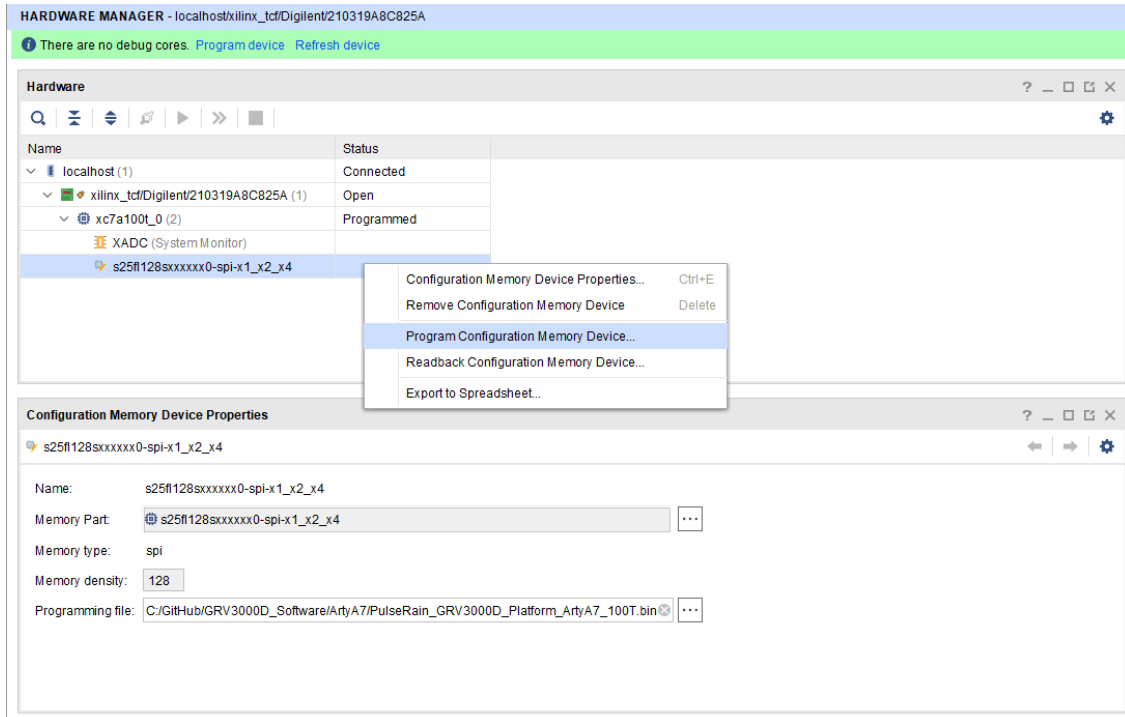


Figure Segger JLink Attached to the Digilent Arty A7 Board

PulseRain GRV3000D – Programmer's Guide

To use the board to evaluate PulseRain GRV3000D, please do the following:

1. Connect the board to a Windows PC through a micro-USB cable.
2. Using Xilinx Vivado Hardware Manager to program the board. The image can be found in https://github.com/PulseRain/GRV3000D_Software/raw/main/ArtyA7/PulseRain_GRV3000D_Platform_ArtyA7_100T.bin, as illustrated in the figure below:



3. Follow the README instructions in <https://github.com/PulseRain/PMSIS> to run the test for PMSIS
4. To run the Arduino Flow, use the sketch below to test the LED and UART on Arty7 board: https://github.com/PulseRain/GRV3000D_Software/tree/main/Sketches/Arty7_test

The LEDs on the board are assigned to the GPIO port as following:

LED	Connections
LD0_RED	processor_paused
LD0_GREEN	~processor_paused
LD0_BLUE	Low
LD1_RED	Low
LD1_GREEN	Low
LD1_BLUE	Low
LD2_RED	Low
LD2_GREEN	Low
LD2_BLUE	Low
LD3_RED	gpio[15]
LD3_GREEN	Low
LD3_BLUE	~gpio[15]
LD4	gpio[12]
LD5	gpio[13]
LD6	gpio[14]
LD7	gpio[15]

Appendix: Arduino Language

The complete reference of Arduino Language can be found in <https://www.arduino.cc/reference/en/>

And the Arduino Flow mentioned in Section 5.3 supports a subset of the Arduino Language as following:

- `uint32_t micros ()`
- `uint32_t millis ()`
- `void interrupts()`
- `void noInterrupts()`
- `void delay (uint32_t ms)`
- `uint8_t isDigit(uint8_t c)`
- `uint8_t isAscii(uint8_t c)`
- `uint8_t isAlpha(uint8_t c)`
- `uint8_t isAlphaNumeric(uint8_t c)`
- `uint8_t isControl(uint8_t c)`
- `uint8_t isGraph (uint8_t c)`
- `uint8_t isHexadecimalDigit(uint8_t c)`
- `uint8_t toLowerCase(uint8_t c)`
- `uint8_t toUpperCase(uint8_t c)`
- `uint8_t isLowerCase(uint8_t c)`
- `uint8_t isUpperCase(uint8_t c)`
- `uint8_t isPrintable(uint8_t c)`
- `uint8_t isPunct(uint8_t c)`
- `uint8_t isSpace(uint8_t c)`
- `uint8_t isWhitespace(uint8_t c)`
- `uint8_t toAscii(uint8_t c)`
- `uint8_t digitalRead(uint8_t pin)`
- `void digitalWrite(uint8_t pin, uint8_t value)`

Appendix: Parameters for BIQUAD Filter

How to Determine the Parameters for Biquad Filter

It seems that analog guys have different takes on the parameters when it comes to the Biquad filter design. For digital filter, the lingo would usually be as "cutoff frequency, passband frequency and transition band roll off. etc". In Matlab, the "filterDesigner" will come handy to start the design. However, analog guys, when they design Biquad filter, sometimes like to talk about corner frequency and Q value. For $Q < 0.707$, the corner frequency is defined as the 3dB frequency. For $Q > 0.707$, the corner frequency is defined at the peak of the ripple.

➤ Parameters for the Biquad low pass filter

In the S domain, the general form of a Biquad low pass filter can be set as:

$$\frac{1}{S^2 + \left(\frac{\omega_0}{Q}\right) \cdot S + \omega_0^2}$$

At DC, its gain is $|H_s(0)| = \frac{1}{\omega_0^2}$, which is constant.

At frequency ω , plug ' $j\omega$ ' into ' S ', the gain becomes $|H_s(\omega)| = \frac{1}{\sqrt{\omega^4 + \omega^2 \cdot \omega_0^2 \cdot \left(\frac{1}{Q^2} - 2\right) + \omega_0^4}}$

if $\frac{1}{Q^2} - 2 < 0$, i.e., $Q > \sqrt{0.5} = 0.707$, $|H_s(\omega)|$ would have a peak value at $\omega = \omega_0 \cdot \sqrt{1 - \frac{1}{2 \cdot Q^2}}$

$$\text{Gain at the peak would be } \left| \frac{H_s(\omega_0 \cdot \sqrt{1 - \frac{1}{2 \cdot Q^2}})}{H_s(0)} \right| = \frac{2 \cdot Q^2}{\sqrt{4 \cdot Q^2 - 1}}$$

If $\frac{1}{Q^2} - 2 > 0$, i.e. $Q < \sqrt{0.5} = 0.707$, $|H_s(\omega)|$ would be monotonically descending,

PulseRain GRV3000D – Programmer's Guide

The 3dB point would be at $\omega_{3dB} = \frac{\omega_0}{\sqrt{1 - \frac{1}{2Q^2} + \frac{1}{2}\sqrt{\frac{1}{Q^4} - \frac{4}{Q^2} + 8}}}$

Now move on to the Z domain, sample rate f_s has to be much greater than the frequency of interest so that frequency warping can be ignored. (i.e., what's in the S domain is almost the same as what's in the Z domain) Doing Bi-linear transformation, replace S with $S = 2 \cdot f_s \cdot \frac{z-1}{z+1}$, we have

$$H_z(Z) = \frac{1 + 2 \cdot Z^{-1} + Z^{-2}}{4 \cdot f_s^2 + \left(\frac{2 \cdot \omega_0 \cdot f_s}{Q}\right) \cdot Z^{-1} + \omega_0^2 \cdot Z^{-2}}$$

➤ Parameters for the Biquad high pass filter

In the S domain, the general form of a Biquad high pass filter can be set as: $\frac{S^2}{S^2 + \left(\frac{\omega_0}{Q}\right) \cdot S + \omega_0^2}$

At infinite, $|H_s(\infty)| = 1$, which is constant unit.

At frequency ω , plug ' $j\omega$ ' into ' S ', the gain becomes

$$|H_s(\omega)| = \frac{\omega^2}{\sqrt{\omega^4 + \omega^2 \cdot \omega_0^2 \cdot \left(\frac{1}{Q^2} - 2\right) + \omega_0^4}} = \frac{1}{\sqrt{\left(\frac{\omega_0}{\omega}\right)^4 + \left(\frac{\omega_0}{\omega}\right)^2 \cdot \left(\frac{1}{Q^2} - 2\right) + 1}}$$

If $\frac{1}{Q^2} - 2 > 0$, i.e., $Q < \sqrt{0.5} = 0.707$, $|H_s(\omega)|$ is monotonically ascending with respect to ω , so corner frequency will be defined at 3dB point, which is

$$\left| \frac{H_s(\omega)}{H_s(0)} \right| = \frac{1}{\sqrt{\left(\frac{\omega_0}{\omega}\right)^4 + \left(\frac{\omega_0}{\omega}\right)^2 \cdot \left(\frac{1}{Q^2} - 2\right) + 1}} = \frac{1}{\sqrt{2}}$$

So we have $\omega_0 = \omega \cdot \sqrt{1 - \frac{1}{2 \cdot Q^2} + \frac{1}{2} \cdot \sqrt{\frac{1}{Q^4} - \frac{4}{Q^2} + 8}}$ at corner frequency.

PulseRain GRV3000D – Programmer's Guide

If $\frac{1}{Q^2} - 2 < 0$, i.e., $Q > \sqrt{0.5} = 0.707$, its peak was achieved at frequency $\omega = \omega_0 \cdot \sqrt{1 - \frac{1}{2 \cdot Q^2}}$

$$\left| \frac{H_s(\omega_0 \cdot \sqrt{1 - \frac{1}{2 \cdot Q^2}})}{H_s(\infty)} \right| = \frac{2 \cdot Q^2}{\sqrt{4 \cdot Q^2 - 1}}$$

Now move on to the Z domain, since sample rate f_s is much greater than the frequency of interest, frequency warping can be ignored. Doing Bi-linear transformation, replace S with $S = 2 \cdot f_s \cdot \frac{z-1}{z+1}$, we have

$$H_z(Z) = \frac{4 \cdot f_s^2 \cdot (1 - 2 \cdot Z^{-1} + Z^{-2})}{4 \cdot f_s^2 + \frac{2 \cdot \omega_0 \cdot f_s}{Q} + \omega_0^2 + (2 \cdot \omega_0^2 - 8 \cdot f_s^2) \cdot Z^{-1} + (\omega_0^2 + 4 \cdot f_s^2 - \frac{2 \cdot \omega_0 \cdot f_s}{Q}) \cdot Z^{-2}}$$