Introduction to RISC-V

RISC-V, an open-source instruction set architecture (ISA), has gained traction for its simplicity, flexibility, and open-source nature. Founded in 2015 by the RISC-V Foundation, it offers both 32-bit and 64-bit addressing, with optional extensions for floating-point arithmetic, vector processing, and cryptography. Its modular design allows for various configurations and extensions to meet different application needs.

With a small set of basic instructions, it is easy to implement, optimize, verify, and secure, making it suitable for embedded systems, IoT devices, and high-performance computing. The open-source ecosystem surrounding RISC-V has led to a vibrant community of developers, researchers, and companies contributing to its development. Companies like SiFive and Western Digital have adopted RISC-V for their processors and storage controllers, respectively. The RISC-V Foundation continues to develop and expand the ISA, ensuring its relevance and applicability in the ever-evolving computing landscape.

In India, RISC-V's open-source nature and flexibility have made it an attractive option for various applications. The Indian government's focus on promoting indigenous hardware and software development aligns well with RISC-V's open-source principles. This has led to a growing interest in RISC-V among Indian companies, educational institutions, and startups.

Several Indian companies have started adopting RISC-V for their processor designs, including those aimed at IoT devices, embedded systems, and data centers. Educational institutions in India have also begun incorporating RISC-V into their curriculum, providing students with the opportunity to learn about and work with this emerging ISA.

Additionally, several Indian startups have emerged that focus on RISC-V development, providing tools, software, and hardware solutions based on the ISA. This growing interest in RISC-V in India is expected to contribute to the broader adoption and development of the ISA globally, further strengthening its position as a viable alternative to proprietary ISAs.

Introduction to RISC-V Compiler

A Linux-based RISC-V compiler is a compiler that runs on a Linux operating system and is capable of compiling code written in the RISC-V instruction set architecture (ISA).

GCC (GNU Compiler Collection) is one of the most popular and widely used compilers for RISC-V. GCC supports a wide range of programming languages, including C, C++, and Fortran, and is available on various platforms, including Linux. To use GCC for RISC-V development on Linux, you need to install the appropriate version of GCC that supports RISC-V.

VSDSquadron Mini

The VSDSquadron Mini RISC-V SoC Kit - Features and Interfaces:

- Core Processor: The board is powered by CH32V003F4U6 chip with 32-bit RISC-V core based on RV32EC instruction set, optimized for high-performance computing with support for 2-level interrupt nesting and supports 48MHz system main frequency in the product function.

- Clock and Reset Systems: Includes a built-in factory-trimmed 24MHz RC oscillator and a 128kHz RC oscillator, plus an external 4-25MHz oscillator option for varied clocking requirements.
- Robust GPIO Support: Boasts 3 groups of GPIO ports, totaling 18 I/O ports, enabling extensive peripheral connections and mapping to external interrupt capabilities Flexible Communication Interfaces: Offers multiple communication protocols including USART, I2C, and SPI for versatile connectivity options.

- Packaging and Debugging: Comes in a compact QFN package with a single-wire serial debug interface (SDI) for streamlined troubleshooting and integration.

- High-Speed Memory: Equipped with 2KB SRAM for volatile data storage, 16KB CodeFlash for program memory, and additional 1920B for bootloader functionalities.

- On-board Programmer: Features on-board CH32V305FBP6 single-wire programming protocol, enhancing development efficiency with seamless code deployment and debugging. NO NEED to purchase any additional adapter.

- The VSDSquadron Mini RISC-V SoC device available on the kit is programmed using the on-board flash programmer which supports the CH32V305FBP6 single-wire programming protocol, enabling streamlined and efficient development workflows directly on the board. Connect the VSDSquadron Mini board using a USB micro B connector to program the CH32V003F4U6 chip.

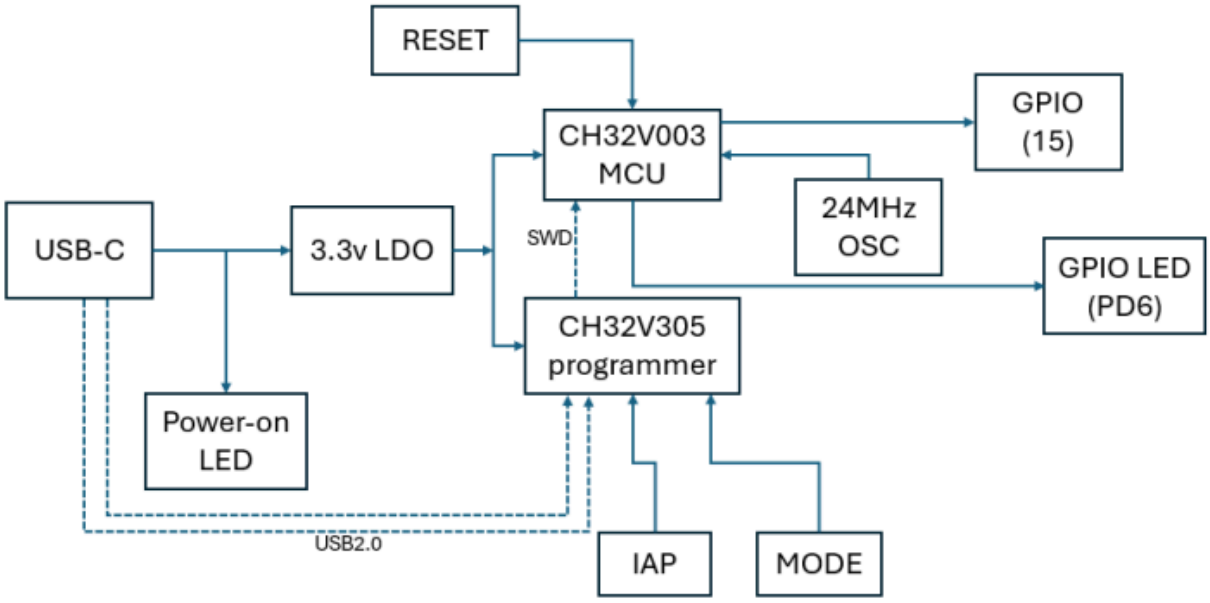
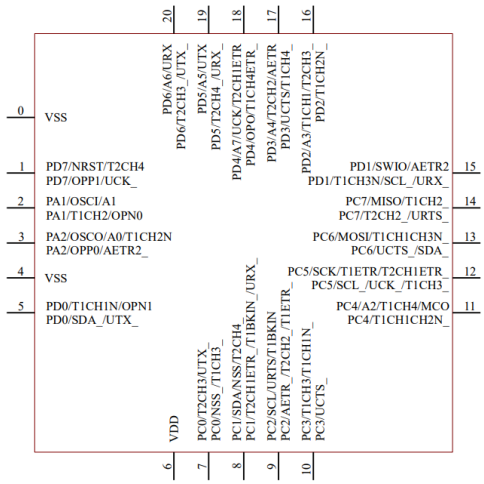


Figure 1: VSDSquadron Mini RISC-V SoC Kit Block Diagram

CH32V003

CH32V003 series is an industrial-grade general-purpose microcontroller designed based on QingKe RISC-V2A core, which supports 48MHz system main frequency in the product function. The series features wide voltage, single line debug, low-power consumption and ultra-small package. It provides commonly used peripheral functions, built-in 1 group of DMA controllers, 1 group of 10-bit analog-to-digital conversion ADC, 1 group of op-amp comparator, multiple timers, standard communication interfaces such as USART, I2C, SPI, etc. The rated operating voltage of the product is 3.3V or 5V, and the operating temperature range is -40℃ ~ 85℃ industrialgrade.

CH32V003F4U6



How to write Basic Code

1. **Include Header Files:**

```
#include <ch32v00x.h>
#include <debug.h>
```

2. **Pin Configurations:**

```
void GPIO_Config(void)
{
   GPIO_InitTypeDef GPIO_InitStructure = {0}; //structure variable used for the GPIO configuration

   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE); // to Enable the clock for Port D

}
```

Input Pin Definition:

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_X | GPIO_Pin_Y | GPIO_Pin_Z ; // Defines which Pin to configure
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
   GPIO_Init(GPIOD, &GPIO_InitStructure);
```

Output Pin Definition:

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_X | GPIO_Pin_Y | GPIO_Pin_Z ; // Defines which Pin to configure
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; // Defines Output Type
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; // Defines speed
   GPIO_Init(GPIOD, &GPIO_InitStructure);
```

PD0 => GPIO_Pin_0
PD1 => GPIO_Pin_1
PD2 => GPIO_Pin_2
PD3 => GPIO_Pin_3
PD4 => GPIO_Pin_4
PD5 => GPIO_Pin_5
PD6 => GPIO_Pin_6
PD7 => GPIO_Pin_7

3. **Main Function:**

```
int main(void)
{
   uint8_t b0, b1, b2, g0 , g1, g2 = 0;
   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
   SystemCoreClockUpdate();
   Delay_Init();
   GPIO_Config();

   while(1)
   {

   }
}
```

4. **Input / Output Statements:**

```
b0 = GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_4); //Input Statement
GPIO_WriteBit(GPIOD, GPIO_Pin_0, RESET); //Reset Output Pin
GPIO_WriteBit(GPIOD, GPIO_Pin_0, SET); //Set Output Pin
```

PROJECT

Problem Statement: Implementing Binary to Gray Code Convertor using VSDSquadron Mini

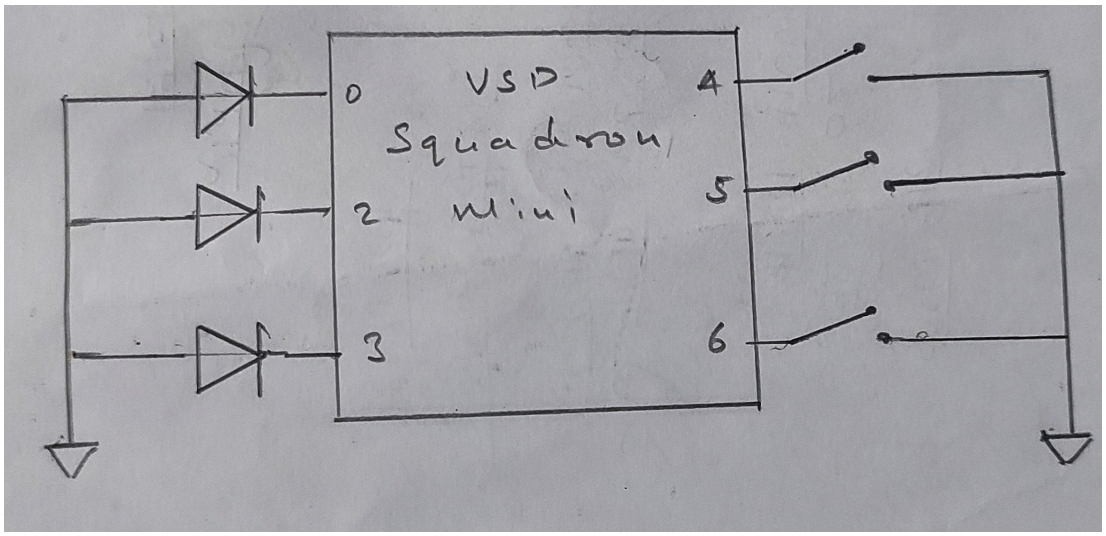
Hardware Required: VSDSquadron Mini, LEDs (L_2, L_1, L_0), Buttons (B_2, B_1, B_0), Breadboard, Jumper Wires

Software Required: PlatformIO, VS Code

Hardware Connections:

LEDs	$L_2(+ve = Pin\_0, -ve = GND), L_1(+ve = Pin\_2, -ve = GND), L_0(+ve = Pin\_3, -ve = GND)$
Buttons	$B_2(+ve = Pin\_4, -ve = GND), B_1(+ve = Pin\_5, -ve = GND), B_0(+ve = Pin\_6, -ve = GND)$

Block Diagram:



Concept: Here 3-Bit Binary ($B_2B_1B_0$) Input is taken from the user and return 3-Bit Gray Code ($G_2G_1G_0$)

Mathematical Expression: $G_2 = B_2$; $G_1 = XOR(B_2, B_1)$; $G_0 = XOR(B_1, B_0)$

Decimal	Binary	Gray Code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

CODE:

```
#include <ch32v00x.h>
#include <debug.h>

int xor(int bit1, int bit2)
{  int xor = bit1 ^ bit2;
   return xor;
}


void GPIO_Config(void)
{  GPIO_InitTypeDef GPIO_InitStructure = {0}; //structure variable used for the GPIO configuration

   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE); // to Enable the clock for Port D

   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_6 ; // Defines which Pin to configure
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; // Defines Input Type
   GPIO_Init(GPIOD, &GPIO_InitStructure);

   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_2 | GPIO_Pin_3 ; // Defines which Pin to configure
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; // Defines Output Type
   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; // Defines speed
   GPIO_Init(GPIOD, &GPIO_InitStructure);
}

int main(void)
{  uint8_t b0, b1, b2, g0 , g1, g2 = 0;
   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
   SystemCoreClockUpdate();
   Delay_Init();
   GPIO_Config();

   while(1)
   {  b0 = GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_4);
      b1 = GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_5);
      b2 = GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_6);
      g0 = xor(0, b0);
      g1 = xor(b0, b1);
      g2 = xor(b1, b2);
      if(g0 == 0)
      { GPIO_WriteBit(GPIOD, GPIO_Pin_0, RESET); }
      else
      { GPIO_WriteBit(GPIOD, GPIO_Pin_0, SET); }
      if(g1 == 0)
      { GPIO_WriteBit(GPIOD, GPIO_Pin_2, RESET); }
      else
      { GPIO_WriteBit(GPIOD, GPIO_Pin_2, SET); }
      if(g2 == 0)
      { GPIO_WriteBit(GPIOD, GPIO_Pin_3, RESET); }
      else
      { GPIO_WriteBit(GPIOD, GPIO_Pin_3, SET); }
   }
}
```

Conclusion: The Binary to Gray Code is implemented successfully using the RISC-V Processors VSDSquadron Mini and the video for the same is attached with this report.