Introduction to RISC-V

RISC-V is an open-source instruction set architecture (ISA) recognized for its simplicity, adaptability, and open-source nature. Established by the RISC-V Foundation in 2015, it supports both 32-bit and 64-bit addressing while offering optional extensions for floating-point arithmetic, vector processing, and cryptography. Its modular design enables different configurations to suit various application needs.

With a minimal yet powerful set of instructions, RISC-V is straightforward to implement, optimize, and secure. It is widely used in embedded systems, IoT devices, and high-performance computing. The open-source ecosystem has fostered an active community of developers, researchers, and companies contributing to its advancement. Prominent companies such as SiFive and Western Digital have integrated RISC-V into their processors and storage controllers. The RISC-V Foundation continues to develop and expand the ISA to maintain its relevance in the evolving computing landscape.

In India, the open-source and flexible nature of RISC-V aligns with the government's initiatives to promote indigenous hardware and software development. This has resulted in increasing interest among Indian companies, educational institutions, and startups. Many Indian firms are incorporating RISC-V into processor designs for IoT devices, embedded systems, and data centers. Educational institutions have started including RISC-V in their curriculum to equip students with knowledge and hands-on experience. Additionally, Indian startups are actively developing tools, software, and hardware solutions based on the ISA, further strengthening its global adoption.

Introduction to RISC-V Compiler

A Linux-based RISC-V compiler operates within the Linux environment and is used for compiling code written in the RISC-V ISA.

The GNU Compiler Collection (GCC) is one of the most widely used compilers for RISC-V. It supports multiple programming languages such as C, C++, and Fortran and is compatible with various platforms, including Linux. To use GCC for RISC-V development on Linux, the appropriate version supporting RISC-V needs to be installed.

VSDSquadron Mini

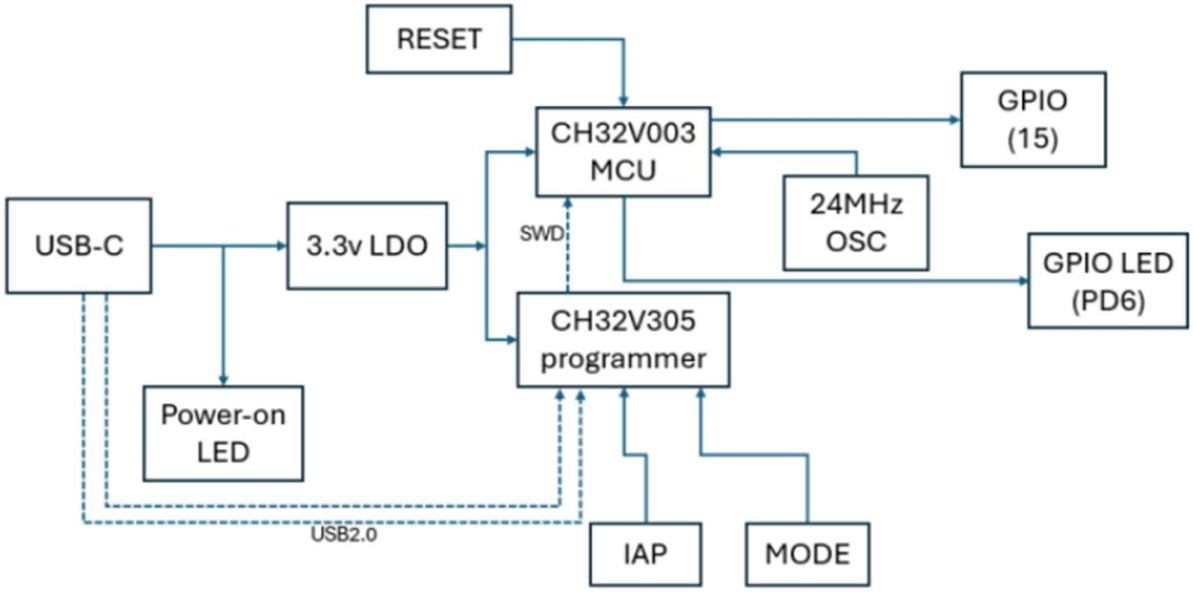

Figure 1: VSDSquadron Mini RISC-V SoC Kit Block Diagram

The VSDSquadron Mini RISC-V SoC Kit offers several features and interfaces:

- **Core Processor:** Utilizes the CH32V003F4U6 chip with a 32-bit RISC-V core based on the RV32EC instruction set, optimized for high-performance computing with a system main frequency of 48MHz and support for two-level interrupt nesting.

- **Clock and Reset Systems:** Includes a factory-trimmed 24MHz RC oscillator, a 128kHz RC oscillator, and an external 4-25MHz oscillator option for diverse clocking requirements.
- **GPIO Support:** Offers three groups of GPIO ports, totaling 18 I/O ports, allowing extensive peripheral connectivity and external interrupt mapping.
- **Communication Interfaces:** Supports USART, I2C, and SPI protocols for flexible communication options.
- **Packaging and Debugging:** Comes in a compact QFN package with a single-wire serial debug interface (SDI) for seamless troubleshooting and integration.
- **High-Speed Memory:** Features 2KB SRAM for volatile storage, 16KB CodeFlash for program memory, and 1920B for bootloader functionalities.
- **On-board Programmer:** Integrated CH32V305FBP6 single-wire programming protocol enables easy code deployment and debugging, eliminating the need for additional adapters.
- **Programming and Development:** The VSDSquadron Mini SoC is programmed using an on-board flash programmer via a USB micro B connector.

CH32V003 Microcontroller

The CH32V003 series is an industrial-grade microcontroller based on the QingKe RISC-V2A core. It supports a 48MHz system main frequency and features a wide voltage range, single-line debugging, low power consumption, and an ultra-small package. The microcontroller includes essential peripheral functions, such as:

- DMA controllers
- 10-bit ADC for analog-to-digital conversion
- Operational amplifier comparator
- Multiple timers
- Standard communication interfaces like USART, I2C, and SPI

Operating at a voltage of 3.3V or 5V, it functions within an industrial-grade temperature range of -40°C to 85°C.

Basic Code Implementation

1. **Including Header Files:**

```
#include <ch32v00x.h>

#include <debug.h>
```

2. **Pin Configurations:**

```
void GPIO_Config(void) {
  GPIO_InitTypeDef GPIO_InitStructure = {0};
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);
}
```

- **Input Pin Definition:**

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_X | GPIO_Pin_Y | GPIO_Pin_Z;

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;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOD, &GPIO_InitStructure);
```

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 Operations:**

```
b0 = GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_4);
GPIO_WriteBit(GPIOD, GPIO_Pin_0, RESET);
GPIO_WriteBit(GPIOD, GPIO_Pin_0, SET);
```

Project: Binary to Gray Code Converter Implementation

- **Hardware Requirements:** VSDSquadron Mini, LEDs, Buttons, Breadboard, Jumper Wires
- **Software Requirements:** PlatformIO, VS Code
- **Hardware Connections:**
  - LEDs connected to specific GPIO pins
  - Buttons connected to designated GPIO pins

Mathematical Expression for Gray Code Conversion:

```
G2 = B2
G1 = XOR(B2, B1)
G0 = XOR(B1, B0)
```

Code Implementation:

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

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

```
void GPIO_Config(void) {
   GPIO_InitTypeDef GPIO_InitStructure = {0};
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_6;
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
   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);
   }
}
```

Conclusion

The Binary to Gray Code conversion was successfully implemented using the RISC-V-based VSDSquadron Mini. A demonstration video is attached with this report.