DRIVING GPIO PINS WITH RISC-V INSTRUCTION SET ARCHITECTURE

Darius Miles Gatson, Houston Baptist University
Mr. Ryan Duane Barnes, Houston Baptist University

Ryan Barnes is a Junior student pursuing a BSc in Computer Science at Houston Baptist University with a strong foundation of object-oriented programming in python, java and javascript.
Abstract

In this paper we are demonstrating the use of RISC-V assembly code to operate a 4-bit full adder. This project was implemented as a partial fulfillment of our Computer Architecture (COSC 3341) course project where we learned about the latest open-source RISC-V Instruction Set Architecture (ISA). To implement this project, we used the SN74LS283N 4-bit full adder IC, the SiFive HiFive1 Rev B board that hosts a 32-bit RISC-V processor, breadboard, LEDs, and few resistors. To program the circuit, we used Visual Studio Code with the PlatformIO extension to allow deployment and debugging of the RISC-V assembly code on the HiFive board. The goal of this project was to demonstrate the full 4-bit adder truth table by cycling through different inputs and displaying the adder output using LED lights. The input to the adder was driven by onboard GPIO pins that were actuated through RISC-V assembly code. Using the SiFive HiFive1 Rev B manuals, we identified the base memory addresses of the GPIO pins and the respective offsets of the individual pins. Through experimentation and debugging of our assembly code, we were able to see both the effects of running the code using debug mode, which is un-optimized code, and the effects of compiled and optimized binary code. The un-optimized code ran the circuit in a deterministic way and the output was as expected, while the optimized code deviated from expectations due to out-of-order instructions execution which caused the RISC-V registers to be updated out of sequence, making the output non-deterministic. The optimized code produced results in which a single GPIO pin would stay on for the duration of the truth table operation and skewed the output as a result. In the un-optimized code (debug mode), the GPIO pins held HIGH and LOW when expected. When attempting to disable optimization manually, no attempts were successful. These results imply that code optimizations done by compilers could sometimes lead to unpredictable outputs.

Introduction

RISC-V is a new Instruction Set Architecture (ISA) designed to support computer architecture research and education\(^2\). Originated in 2010 at UC Berkley and slowly became endorsed internationally by different companies and industries, which led to the establishment of the RISC-V International non-profit organization in 2015 (RISC-V)\(^1\). The RISC-V ISA is delivering new levels of free extensible software and hardware freedom in architecture and is paving the way for unlimited opportunities of computing design and innovation. RISC-V ISA has two options 32-bit and 64-bit user
level address with both having 31 general purpose registers and the 32\textsuperscript{nd} register being a hardwired zero register used primarily for clearing registers and comparisons.

Driving GPIO pins using RISC-V Instruction Set Architecture (ISA) was an assigned project given to challenge our knowledge and skills learned in Computer Architecture (COSC 3341). The course introduces RISC-V ISA and assembly code, instructions decoding, CPU clocking, performance enhancements, memory hierarchy, cache types, pipelining, pipelining hazards, and much more. In this course, we learned the interfacing between hardware and software in modern computing systems and how instructions are executed at the processor level. The objective of the project was to drive GPIO pins on a SiFive HiFive1 Rev B\textsuperscript{3} board to activate the input pins of an SN74LS283N 4-bit adder IC and display the 4-bit adder output using LED lights. Controlling these GPIO pins was accomplished programmatically using RISC-V assembly code wrapped in C code.

Overall, this project was used to put our skills learned in COSC 3341 to the test to demonstrate how high-level programming language correlates to assembly level code. We observed after building the circuit and developing the code that the truth table displayed using the LEDs was skewed by not exhibiting the pattern that we programmed. After testing, fixing bugs and trying different solutions, the circuit operated correctly in debug mode. After much discussion, we concluded that the C optimizer was executing the code out-of-order, but in debug mode, code optimizations are disabled by default and the instructions executed in the way we expected.

Project Design and Implementation

The project design and implementation involved two main phases: 1- Circuit design and testing, 2- Coding and testing the circuit using RISC-V assembly instructions. Continuing, we will present each phase in more details.

1. Circuit Design and Testing

To implement this project, we used the SN74LS283N 4-bit full adder IC, the SiFive HiFive1 Rev B board that hosts a 32-bit RISC-V processor, breadboard, LEDs, and few resistors. To send voltage to the 4-bit adder and produce the expected output, the SiFive HiFive1 Rev B onboard GPIO pins were employed. As shown in Figure 1, 8 pins in total were used to connect the board with the 4-bit adder via breadboard, 4 pins (labeled A1 – A4) allocated to one 4-bit number and the other 4 pins (labeled B1 – B4) allocated to the second 4-bit number to be added. In addition to the adder input connections, 5 LEDs were used to show the output of the adder, with each bit being represented by a single LED, including the carry bit (labeled $\Sigma_1 - \Sigma_4$ and C4). The Vcc and GND pins are connected to the 3.3V positive and negative pins of the board for power.

To test the full functionality of our 4-bit adder before code development, we manually applied active high and low voltages to the input bits and verified the results as given by the LED’s connected to the result pins. Once it was verified that the adder was fully functional, we continued to the second phase of the project: providing input voltages to the adder via the onboard GPIO pins using RISC-V assembly code.
Proceedings of the 2022 ASEE Gulf-Southwest Annual Conference
Prairie View A&M University, Prairie View, TX
Copyright © 2022, American Society for Engineering Education

2. RISC-V Coding of the Circuit
The GPIO pin assignments are numbered differently from the numbers printed on the circuit board. For example, for our project we used pins 0-7 as labeled on the board, but in reality these are GPIO pins 16-23 as indicated by the board manual. The mapped GPIO pin number must be set to ‘1’ in a 32-bit word. For example, enabling GPIO 23, or pin 7 as printed on the circuit board, would require setting the 23rd bit in the 32-bit word to 1, which would look like 0000 0000 1000 0000 0000 0000 0000 0000 (0x00800000). Once the combined word value for all pins is identified, the pins can be configured as output pins by writing the word value to the GPIO_OUTPUT_EN memory address as shown in the sample code in Figure 2 (Lines 1-3).

All of the obtained word values for all of the pins we used are defined in our source code file memorymap.inc as shown in Figure 3. This configuration file is also used to store the base address for accessing the pins (GPIO_CTRL_ADDRESS), and the offsets used for enabling output (GPIO_OUTPUT_EN) and changing pin state (GPIO_OUTPUT_VAL) which are found in FE310-G002 manual. This configuration file is included in all of our .S assembly files.

1 li t0, GPIO_CTRL_ADDRESS # Store into t0 GPIO BASE address
2 li t1, GPIO_PINS # Store combined value for all pins
3 sw t1, GPIO_OUTPUT_EN(t0) # Enable output
4 sw x0, GPIO_OUTPUT_VAL(t0) # Set all pins to 0

Figure 2. setupGPIO.S code snippet

After we have enabled all the pins we needed, we set all pins to zero (active LOW state) using the offset from GPIO_CTRL_ADDRESS defined in GPIO_OUTPUT_VAL as shown in Figure 2 (line 4). To change the state of a pin to HIGH, we use an or instruction to set the pin’s bit to 1 (HIGH) while maintaining all other values. To change the pin to a LOW state, we use an xor instruction to set the pin’s bit to 0 (LOW) while maintaining all other values. These processes are shown in Figure 4.

Figure 1. Schematic diagram of the circuit design
Proceedings of the 2022 ASEE Gulf-Southwest Annual Conference
Prairie View A&M University, Prairie View, TX
Copyright © 2022, American Society for Engineering Education

1. equ GPIO_CTRL_ADDRESS, 0x10012000
2. equ GPIO_OUTPUT_EN, 0x08
3. equ GPIO_OUTPUT_VAL, 0x0C
4. equ GPIO_OUTPUT_XOR, 0x40
5. equ GPIO_PINS, 0x00FF0000
6. equ A1_PIN, 0x00800000  # Pin 7 = A1
7. equ A2_PIN, 0x00400000  # Pin 6 = A2
8. equ A3_PIN, 0x00200000  # Pin 5 = A3
9. equ A4_PIN, 0x00100000  # Pin 4 = A4
10. equ B1_PIN, 0x00080000 # Pin 3 = B1
11. equ B2_PIN, 0x00040000  # Pin 2 = B2
12. equ B3_PIN, 0x00020000 # Pin 1 = B3
13. equ B4_PIN, 0x00010000  # Pin 0 = B4

Figure 3. memorymap.inc code snippet

1 pinHigh:                       # File pinHIGH.S
2     . . .
3    li t0, GPIO_CTRL_ADDRESS    # Load the GPIO Address
4    lw t1, GPIO_OUTPUT_VAL(t0)  # Get the current value of the pins
5    or t1, t1, a0               # Set pin to HIGH
6    sw t1, GPIO_OUTPUT_VAL(t0)  # Store into memory

1 pinLow:                        # File pinLOW.S
2     . . .
3    li t0, GPIO_CTRL_ADDRESS    # Load the GPIO Address
4    lw t1, GPIO_OUTPUT_VAL(t0)  # Get the current value of the pins
5    xor t3, a0, t1              # XOR pin value with state addresses
6    sw t3, GPIO_OUTPUT_VAL(t0)  # Store into memory

Figure 4. pinHigh.S and pinLow.S code snippet

10 while(1){
11    for(int i = 0; i < NUM_A_PINS; i++) { // Loop through A pins
12        pinHigh(A_PINS[i]);          // A[i] pin turned HIGH
13        delay(DELAY);              // Delay for DELAY time
14        for(int j = 0; j < NUM_B_PINS; j++) {
15            pinHigh(B_PINS[j]);      // B[i] pin turned HIGH
16            delay(DELAY);         // Delay again
17            pinLow(B_PINS[j]);    // Turn B[i] pin LOW
18        }
19    } pinLow(A_PINS[i]);           // A[i] pin turned LOW
20 }}

Figure 5. main.c code snippet

Finally, to execute our assembly instructions and loop through the truth table for the 4-bit adder, we wrapped our assembly code in C code. As shown in Figure 5, the main.c file contains the main method that loops through the different combinations of inputs to produce the truth table. In-between state changes, a delay function implemented using RISC-V assembly instructions is invoked. The delay is induced until a stopping time calculated as (delay x frequency) + current timer value is less than the timer value maintained in memory as shown in Figure 6.
Observations

When deploying the code onto our SiFive HiFive1 Rev B board, the output of the 4-bit adder as indicated by the LEDs did not have the correct pattern that we programmed, and one LED remained on for the duration of the faulty truth table. To remedy this, we tried using different GPIO pins, but this did not resolve the issue. During our troubleshooting, we attempted to manually write out the state changes line by line, we tested to make sure each pin was giving an active LOW signal after initialization, we tried re-enabling the pins after each state change in case the enabled status was being overwritten somehow. Finally, when we started executing the code in debug mode for line-by-line verification, the program produced the correct truth table. This led to the conclusion that the out-of-order execution optimizations done by the C compiler was the reason why our circuit was not functioning in the deterministic way that we programmed it.

Summary and Conclusions

In summary, this project challenged our comprehension of skills gained in the Computer Architecture course COSC 3341. The goal of this project was to display the 4-bit adder truth table using a 4-bit adder and a SiFive HiFive1 Rev B board. To accomplish this project we programmed the board using RISC-V assembly instructions to send voltage to the input pins of the 4-bit adder. However, with the project only working in debug mode, we concluded that compilation optimizations, namely, out-of-order code execution that is used by many modern compilers and processors to optimize performance by maximizing the usage of the instructions pipeline, prevented the correct execution of our programmed circuit. The solution to this problem would be to disable compiler optimizations in gcc, though we were unable to do this, and our attempts to bypass PlatformIO yielded similar results. This predicament shows that optimizations, though useful, can be an obstacle to the correct execution of programs given that compilers have no semantical information about the code logic.

References