SPI on RTU

EE149/249A Final Report, Fall 2014

Jerry Chen, Richard Lin, Allen Tang

1 Introduction

Processors in microcontrollers are optimized for speed - executing tasks in order to minimize completion time. Often burdened with interrupts, nondeterministic execution, auxiliary tasks, and non real-time operating systems, processors are unable to make any timing guarantees on most operations. In a traditional setup, interfacing with external components is done via protocol-specific hardware peripherals embedded into silicon. This hardware-based implementation puts restrictions on both the types of communication protocols that can be used and the number of external ports implementing a certain protocol. Furthermore, fixed pin functionality mapping may also cause sub-optimal PCB routing.

To resolve some of these issues, we explore using a timing-predictable co-processor attached to general-purpose input/output (GPIO) pins. Digital communication protocols can be implemented using bit banging, a technique in which software is used to control signals, giving flexibility. The co-processor can also offload cycle-consuming hardware tasks from the main processor, allowing the main processor to utilize its resources elsewhere. Such a co-processor is called a real time unit (RTU).

2 Overview

The goal of this project was to implement bit banging on a co-processor to handle communication with external peripherals. The co-processor we used was the FlexPRET processor, a 5-stage RISC-V processor (implemented in Chisel by Michael Zimmer et al.) designed specifically for real-time embedded systems.

First, we wrote a bit bang Serial Peripheral Interface (SPI) routine, ran the system in simulation, and verified the waveforms. Next, we deployed the FlexPRET processor onto an FPGA and tested it by running programs that communicated with an external SPI accelerometer. We then leveraged the processor's multi-threading to run pulse width modulation (PWM) concurrently with accelerometer SPI. Finally, we improved both our SPI and PWM implementation by making the timing precise down to a processor cycle.

3 Protocols Implemented

3.1 SPI

Serial Peripheral Interface (SPI) is a full-duplex data link used for communication between one master and one or more slaves. The protocol itself normally consists of four signals: serial clock, MOSI (data from master to slave), MISO (data from slave to master), and slave select (used to specify which slave the master is talking to). During a transfer, each clock period carries a data bit - data is sampled on one edge while new data is presented on the other. The actual waveform is specified by the clock frequency along with two parameters: clock phase (CPHA - which clock edge does what), and clock polarity (CPOL - specifying the idle level of clock).

3.2 PWM

Pulse-width modulation (PWM) is a common technique for using digital signals to produce analog signals. It is widely used to control power applications where a higher duty cycle (proportion of time that the signal is high relative to the period) is, the more power it delivers. One example use is for dimming LEDs with digital-only control.

4 Deploying to FPGA

4.1 Accelerometer Interface

While simulation can provide bountiful information and catch almost all the bugs in our code, we cannot rely on it alone. Thus, to test our programs

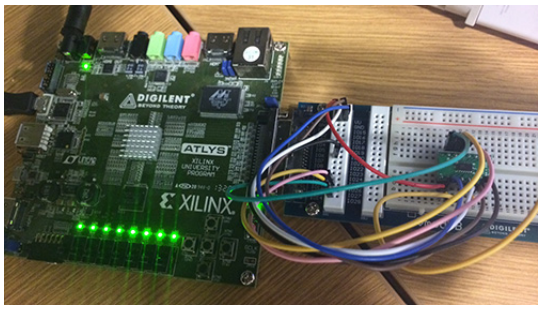


Figure 1: FPGA-Accelerometer Test Setup

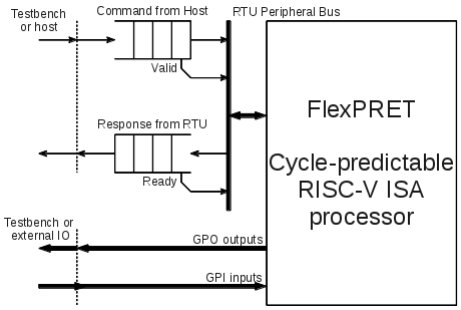


Figure 2: External Connections

running on the FlexPRET processor on real hardware, we deployed it onto an FPGA. We chose the Digilent Atlys development board (with Xilinx Spartan 6 FPGA) because it has been previously used to successfully deploy the processor. Among other I/Os, the board has 8 LEDs, 8 switches, and 28 external GPIOs on a breakout breadboard which we connected via a 68-pin VHDCI connector.

The first step to deploying the processor was setting up the environment and installing the necessary software (Xilinx ISE and iMPACT). This step took much more effort than anticipated because the Xilinx software was not fully compatible with Ubuntu (necessary for the RISC-V toolchain). Generating the bitfile with ISE worked fine, but we ran into problems installing the cable driver when we tried to deploy it in iMPACT (this is a known bug). After numerous days of debugging, we finally worked around the problem by using iMPACT on Windows to deploy the bitfile

With the processor on the FPGA, we ran simple C programs and connected the GPIOs to the switches and LEDs. We successfully verified the behavior of these programs, making sure all the LED and switches were working. Next, we modified our bitbanging SPI code to communicate with Freescale's MMA7455 accelerometer. We connected the SPI pins from the accelerometer to GPIO pins on the board, and we succeeded in reading the 8-bit accelerometer data and displaying it onto the LEDs.

4.2 Connecting to PWM

We also deployed bitbanging PWM onto the FPGA. By varying the duty cycle, we were able to control the ratio of on to off time for the GPIO-connected LEDs, essentially changing their brightness. Once we got the accelerometer interface and PWM working, we looked into running the two programs concurrently. We modified our test program to spawn two threads, one to read data from the accelerometer and one to run PWM. For every new value the accelerometer thread reads, it writes a target duty cycle to a shared global variable for the PWM thread. In all, our program varies the brightness of hte LEDs based on accelerometer readings.

5 Host Interface

As a coprocessor, FlexPRET needs a way to communicate with a host processor. To support this, we added two hardware queues, one in each direction between FlexPRET and the host. FlexPRET can access these queues and the relevant status registers through a memory-mapped IO on a pre-existing peripheral bus. It is expected that the RTU software will check that the queue from the host is valid (has data) before reading from the queue and will check that the queue to the host is ready (not full) before writing to the queue. This is because FlexPRET architecture requires that all memory accesses complete in a single cycle, so stalling the memory system in hardware until the queues are ready is not an option.

5.1 Protocol

For this project, we did not attempt to define general semantics or recommendations for the host interface protocol. Instead, what hardware we wrote is it - the two queues essentially provide a stream of 32-bit words to and from the core. The exact meanings are up to each RTU program.

However, to test that our implementation actually works, we implemented host interface code for both our SPI and PWM modules. On both, we used a simple word-wide command protocol consisting of 8 bits of opcode and 24 bits of data per word. While neither efficient nor extensible, it does provide a good starting point.

For SPI, the main code loop repeatedly checks the command from host queue for valid data. Once data is available, it reads from the queue and parses the command. The available commands are `set_period` which sets the SPI clock period (in nanoseconds), `set_polarity` which sets the CPOL and CPHA parameters, and `transfer` which initiates a SPI data transfer on the IO pins. When `transfer` is done, the returned byte read from the IO is put onto the response queue.

However, the host interface for PWM was different. Since PWM is supposed to be always active, it can't block polling for the host interface. Instead, on each PWM cycle, it latches in the new period and duty cycle from shared variables to a local variable. Host interface code runs in a separate thread, reading data from the command in queue and writing to the shared variables as necessary. While the producer-consumer strategy and single-word-long parameters help prevent threading bugs, no attempts were made to synchronize the period and duty cycle. Possibilities are explored in Section 8.1.

6 Testing

Hardware often goes through extensive testing to make sure it works reliably, and users will likely expect the same from soft peripherals. Therefore, another focus of this project was to provide automated testing infrastructure, ensuring quick detection of buggy code.

6.1 Infrastructure

For the testing framework, we used Chisel's `Tester`, which provides methods to `step` cycles through the circuit, write inputs using `poke`, read outputs using `peek`, and `expect` conditions. While these may be sufficient for general hardware testing, they don't make it simple to express temporal constraints. We therefore augmented the testing library with constructs to `step` until some condition as well as `expect` a condition during a particular time interval. This allows users to write testbenches in the typical hardware style but with timing constraints: send inputs and expect timed outputs.

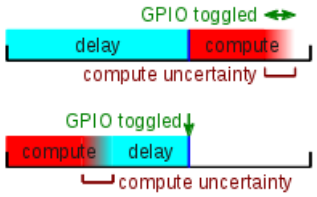
We also added higher-level methods to abstract away common tasks like loading FlexPRET's instruction memory and reading/writing from/to the host interface queues.

6.2 Testbenches

Both our testbenches were simple but reasonably comprehensive: they sent commands through the host interface and expected back GPIO events within certain intervals.

For SPI, the testbench configured the RTU program with the CLK, CPHA, and CPOL, and initiated a transfer. It then expected a SPI waveform where the clock signal met the period specification and the data was stable on all but the transition edge. While the RTU generates the SPI master signals, the testbench fed in the response signals on the MISO line and checks the response queue at the end. Two SPI transfers were done with differing clock phase parameters and data.

For PWM, the testbench similarly loaded the RTU program with the period and duty cycle, then checked to ensure the output waveform was accurate. Several different duty cycles were tested, including the always-low (zero) and always-high (equal to period) edge conditions.



In the first execution, when the GPIO is toggled is dependent on the compute timing uncertainty. In the second execution, the timing uncertainty is absorbed into the delay so when the GPIO is toggled is predictable.

Figure 3: Optimization Illustrated

7 Optimizing

While the testing timing constructs were written in a way to allow events within an interval, a goal was to make everything cycle-precise. The simple linear coding style of computing GPIOs, writing the GPIO, then delaying was prone to slight timing variation because the time to compute the GPIO values is variable based on the computation. For example, this would lead to one SPI clock half-cycle being slightly longer than the other. While these errors are small, they are still annoying.

Our solution was to make the computation happen out-of-phase, during what would be the delay before the GPIO is toggled. This strategy, illustrated in Figure 3, allows the timing-predictable delay to "mask" the timing-volatile portions of the code. While seemingly simple to implement for linear programs, this ends up complicating the control flow of loops as additional state needed to be kept between iterations. While we were able to get perfect cycle-precise waveforms (and set the jitter tolerances within our testbench to zero), this also made the RTU code less readable.

8 Conclusion

We have demonstrated an implementation of several peripherals on the FlexPRET RTU both in simulation and on an FPGA. With the FPGA, we have shown that our SPI code is able to properly communicate with an external digital accelerometer, initialize, and read data from it while generating a PWM waveform to dim LEDs based on that acceleration data.

We have also explored some of the practical issues that might arise when putting a system like this into production. We created both a host interface framework using hardware queues as well as temporally-aware testing constructs. We then integrated these into both the SPI and PWM RTU code, demonstrating an essentially complete system and verified functionality with testbenches.

8.1 Future Work

However, there are ways our work could be extended. First, there are many more timing-constrained IO protocols which could be implemented. While SPI and PWM and hardware are common on most microcontrollers, more rare and complex protocols like CAN and that of the NeoPixel strips would benefit most from a RTU. Testing against more hardware devices would also be important for compatibility.

The infrastructure we have built could also be extended. For example, while we have provided a model for a host interface based on queues, additional research could go into providing best practices or even a framework for high level protocols. DMA compatibility may also be important where bulk data transfer is required. It may also be desirable to have a separate queue for each peripheral, in which case a host interface generator may be helpful. Finally, it may be necessary to several commands to appear to execute atomically - like setting PWM period and duty cycle. This could be accomplished with a separate command telling the RTU to latch in all the new data from the host.

While we have provided basic temporal testing constructs and testbenches for our RTU code, the testing infrastructure could be improved. For example, instead of specifying expected waveforms in a testbench, LTL-like semantics could be associated with RTU code. While we also explored having multiple threads of testing control, we were unable to get Scala's delimited continuations to work with Chisel.