Design and Implementation of a Risc-V based LoRa Module

Mark Njoroge
MSc Eng Candidate, UCT
Introduction

About me

- BSc Eng Mechatronics, 2020 - UCT
- MSc Eng Candidate - UCT
- Enjoy working with embedded systems
Outline

- Context
- The design of the system
  - Hardware Design - PCB
  - Hardware Design - SOC
- How to program SOC
- A Demonstration
- Planned testing
- Plans for the future
1. Background
Context

IoT is moving to Edge/Fog Computing
  - Frees Up Bandwidth
Existing applications may require extra Computing or connectivity to the cloud
This project fits somewhere in that mould and is an investigation and experiment of using Risc-V and LoRa to create a device that would be accessible to many and would contribute to an edge architecture.
A look at LoRa

- A wireless communication protocol
- Aimed at IoT and used by end nodes to communicate with each other along with gateways.

Desirable features include:
- Low Power
- Free to use (LoRaWAN)
- Highly Sensitive
- Long Range - up to 50km

Designed for Low Bandwidth
Risc-V Instruction Set Architecture

- Open Sourced and Modular
  - Allows sharing and customizing of CPUs
- Increasing development of higher order Application Specific Cores
  - AI
  - Signal Processing
- The processors developed for IoT applications are usually in SoCs, as seen in various literatures.
LoRaDongle

A Low-Cost, fully open source custom PCB design that houses a customizable SOC and offers reconfigurable logic
2. The Design
A look at the choices, constraints, evolution and final choices
PCB Overview

- Two prototypes
- Rev 2 is fully functional
- Improvements made on the second prototype:
  - More applicable ICs
  - Added Breakout Section
  - Reduced physical size
  - Made drastically cheaper
    - Rev 1: ~$300 for two boards
    - Rev 2: ~$25 for five boards
- Price of components: ~$90
- Changes explained in following sections
Functional Block Diagram of PCB
Changing USB -> SPI & UART Bridge

Future Technology Devices International Ltd

FT232H Single Channel Hi-Speed USB to Multipurpose UART/FIFO IC

Future Technology Devices International Ltd

FT2232H Dual High Speed USB to Multipurpose UART/FIFO IC
Updating SPI Flash

Micron Serial NOR Flash Memory
3V, Multiple I/O, 4KB Sector Erase
N25Q032A

W25Q128JV-DTR

3V 128M-BIT
SERIAL FLASH MEMORY WITH
DUAL/QUAD SPI & QPI & DTR
Updating FPGA Package - Improved Pinout, circumventing CLK Issue

ERROR: Cell 'spi_bus0_clk$sb_io' cannot be bound to bel 'X24/Y0/io0' since it is already bound to cell 'SB_IO'
1 warning, 1 error
SOC Design - Tools Used

  - SoC builder for FPGA based designs.
  - Supports multiple optional Risc-V CPUs
  - Also has various other cores/IPs necessary for the SOC.
    - Eg comms: SPI, I2C, JTAG, uart, etc
- Other Open source tools for bitstream generation, Project Icestorm ([https://clifford.at/icestorm](https://clifford.at/icestorm)) (Yosys, arachne-pnr)
SOC design made easy with open source tools

LiteX SOC Builder

- Focuses on the interconnect between the necessary cores.
- Generates:
  - Required documentation
  - Header files
  - Linker Information
- Projects that helped greatly:
  - iCEBreaker
  - FOMU

Pure HDL

- Would have to manually design and build SoC
- Requires great effort in ensuring performance
- Might have to develop on cores and peripherals
SOC Design: Choice of CPU

- Low Cost FPGA used
  - Resource Constrained, only 5K LUTs
- Chosen to reduce price and complexity for first prototype
- Settled on Serv ([https://github.com/olofk/serv](https://github.com/olofk/serv))
  - Incredibly small Risc-V CPU!
  - Already Implemented on LiteX
- Can also use PicoRV32(minimal) for more complexity, ie interrupts.
  - Also in LiteX
SOC Design

- Risc-V CPU - Serv/PicoRV32 (minimal)
- 128KB RAM - Using Internal SPRAM of FPGA
- 8MB ROM - Stored in SPI Flash

Communications cores:
- UART - For debugging & communication with host machine
- SPI - For LoRa device
- i2C - For Gas Sensor

General GPIOs
- TX, RX Leds
- LoRa Configuration Pins
SOC Block Diagram

- Serv/PicoRV32 CPU (Master)
- GPIO
- I2C
- SPI
- timer0
- UART
- ROM
- SRAM

Wishbone Bus

External SPI Flash

CSRs
Resource Utilization based on CPU Chosen

<table>
<thead>
<tr>
<th></th>
<th>Serv</th>
<th>PicoRV32(Minimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs (Percentage of FPGA/5280)</td>
<td>2091 (39%)</td>
<td>3633 (68%)</td>
</tr>
<tr>
<td>Embedded Block Ram (Percentage of FPGA/30)</td>
<td>3 (10%)</td>
<td>6 (20%)</td>
</tr>
<tr>
<td>SPRAM</td>
<td>4 (100%)</td>
<td>4 (100%)</td>
</tr>
</tbody>
</table>
4. A look at the Code

Structure of application, building the SOC and snippets
Generating SOC with LiteX

**Target**
- Configures SOC
- Connects the peripherals together
- Builds and Loads SOC

**Platform**
- Sets FPGA Pins
- Tells LiteX FPGA Part number
- Chooses Synthesis and Programmer

---

Python Scripts for Generating SOC

```
(venv_litex) markn@aurora::~/riscv/LoRaDongle/SOC/main$ python LoRaDongle.py --build --flash
INFO:SoC: Build your hardware, easily!
INFO:SoC:-------------------------------------------
INFO:SoC:Creating SoC... (2022-09-09 12:04:22)
INFO:SoC:FPGA device : ice40-up5k-sg48.
INFO:SoC:System clock: 21.000MHz.
INFO:SoCBusHandler:Creating Bus Handler...
INFO:SoCBusHandler:32-bit wishbone Bus, 4.0 GiB Address Space.
```
# IOs

```python
# IOs

# Clk
(clk, 0, Pins("46"), IOStandard("LVCMOS33")),
(cpu_reset, 0, Pins("21"), IOStandard("LVCMOS33")),

# Leds
(user_led, 0, Pins("37"), IOStandard("LVCMOS33")),
(user_led, 1, Pins("36"), IOStandard("LVCMOS33")),

# LoRa Configuration Pins
(lora_config, 0, Pins("34"), IOStandard("LVCMOS33")),
(lora_config, 1, Pins("35"), IOStandard("LVCMOS33")),
(lora_config, 2, Pins("31"), IOStandard("LVCMOS33")),

# UART
(serial, 0,
  Subsignal("tx", Pins("9")),
  Subsignal("rx", Pins("6")),
  IOStandard("LVCMOS33")),

# SPI Flash
(spiflash, 0,
  Subsignal("cs_n", Pins("16"), IOStandard("LVCMOS33")),
  Subsignal("clk", Pins("15"), IOStandard("LVCMOS33")),
  Subsignal("miso", Pins("17"), IOStandard("LVCMOS33")),
  Subsignal("mosi", Pins("14"), IOStandard("LVCMOS33")),
),
```
# Platform

class Platform(LatticePlatform):
    default_clk_name = "clk"
    default_clk_period = 1e9/50e6

    def __init__(self, toolchain="icestorm"):
        LatticePlatform.__init__(self, "ice40-up5k-sg48", _io, toolchain=toolchain)

    def create_programmer(self):
        return IceStormProgrammer()

    def do_finalize(self, fragment):
        LatticePlatform.do_finalize(self, fragment)
        self.add_period_constraint(self.lookup_request("clk", loose=True), 1e9/50e6)
import argparse
from os import path

from migen import *
from migen.genlib.resetsync import AsyncResetSynchronizer
from migen.genlib.io import CRG

from litex.soc.cores.ram import Up5kSPRAM
from litex.soc.cores.clock import iCE40PLL
from litex.soc.integration.soc_core import SoCCore
from litex.soc.integration.builder import Builder, builder_argdict, builder_args
from litex.build.lattice.programmer import IceStormProgrammer
from litex.soc.integration.soc_core import soc_core_argdict, soc_core_args
from litex.soc.integration.doc import AutoDoc
from litex.soc.integration.soc import SoCRegion

#from Platform.LoRaDongleV1 import Platform
from Platform.LoRaDongleV2 import Platform

from litex.soc.cores.uart import UARTWishboneBridge
from litex.soc.cores import gpio
from litex.soc.cores.spi import SPIMaster
from litex.soc.cores.bitbang import I2CMaster

import litex.soc.doc as lxsocdoc
Target - SOC Class

Instantiation and setting parameters:

```python
class BaseSoC(SoCCore):
    """A SoC on LoRaDongle, optionally with a softcore CPU""

    # Statically-define the memory map, to prevent it from shifting across various litex versions.
    SoCCore.mem_map = [
        "sram": 0x10000000,
        "spiflash": 0x20000000,
        "csr": 0xf0000000,
    ]

def __init__(self, cpu, debug, flash_offset, sys_clk_freq, **kwargs):
    """Create a basic SoC for LoRaDongle."

    Create a basic SoC for LoRaDongle. The 'sys' frequency will run at 21 MHz.

    Returns:
    Newly-constructed SoC

    platform = Platform()

    if cpu == "serv":
        kwargs["cpu_type"] = "serv"
    elif cpu == "picorv32":
        kwargs["cpu_type"] = "picorv32"
        # variant defaults when none are provided
        if "cpu_variant" not in kwargs:
            kwargs["cpu_variant"] = "minimal"

    # Force the SRAM size to 0, because we add our own SRAM with SPRAM
    kwargs["integrated_sram_size"] = 0
    kwargs["integrated_rom_size"] = 0

    kwargs["csr_data_width"] = 32

    # Set CPU reset address
    kwargs["cpu_reset_address"] = self.mem_map["spiflash"] + flash_offset

    # SoCCore
    SoCCore.__init__(self, platform, sys_clk_freq, **kwargs)

    self.submodules.crg = _CRG(platform, sys_clk_freq)
```

Selecting CPU

Telling CPU to find Firmware in external SPI Flash
Target - Setting Up Memory

```python
# UP5K has single port RAM, which is a dedicated 128 kilobyte block.
# Use this as CPU RAM.
spram_size = 128 * 1024
self.submodules.spram = Up5kSPRAM(size=spram_size)
sself.register_mem("sram", self.mem_map["sram"], self.spram.bus, spram_size)

# SPI Flash
from litespi.modules import W25Q128JV
from litespi.opcodes import SpiNorFlashOpCodes as Codes
self.add_spi_flash(mode="1x", module=W25Q128JV(Codes.READ_1_1_1), with_master=False)

# Add ROM linker region
self.bus.add_region("rom", SocRegion(
    origin = self.mem_map["spiflash"] + flash_offset,
    size = 8*1024*1024,
    linker = True)
```
Target - Comms and GPIO

##### LoRa Pins#####

**#Busy Input Pin**

```python
self.submodules.lora_busy = gpio.GPIOIn(platform.request("lora_busy"))
self.add_csr("lora_busy")
```

**#Output Pins**

```python
lora_config = Cat(*[platform.request("lora_config", i) for i in range(3)])
self.submodules.lora_config = gpio.GPIOOut(lora_config)
self.add_csr("lora_config")
```

**# SPI**

```python
self.submodules.SPI = SPIMaster(platform.request("spi_bus", 0),
    data_width = 8,
    sys_clk_freq = sys_clk_freq,
    spi_clk_freq = 8e6)
self.add_csr("spi_bus")
```
def main():
    parser = argparse.ArgumentParser(description="LiteX SoC on LoRaDongle")
    parser.add_argument('--build', action='store_true', help="Build SoC")
    parser.add_argument('--cpu', default="serv", help="Select CPU type")
    parser.add_argument('--flash-offset', default=0x20000, help="Boot offset in SPI Flash")
    parser.add_argument('--sys-clk-freq', type=float, default=21e6, help="Select system clock frequency")
    parser.add_argument('--nextpnr-seed', default=0, help="Select nextpnr pseudo random seed")
    parser.add_argument('--nextpnr-placer', default="heap", choices=["sa", "heap"], help="Select nextpnr placer algorithm")
    parser.add_argument('--debug', action='store_true', help="Enable debug features. (UART has to be used with the wishbone-tool.)")
    parser.add_argument('--document-only', action='store_true', help="Do not build a soc. Only generate documentation.")
    parser.add_argument('--flash', action='store_true', help="Load bitstream")

    builder_args = parser
    soc_core_args = parser
    args = parser.parse_args()

    # Create the SOC
    soc = Basesoc(cpus=cpu, debug=debug, flash_offset=flash_offset, sys_clk_freq=int(sys_clk_freq), **soc_core_args)
    set yosys_nextpnr_settings(nextpnr_placer=nextpnr_placer)

    # Configure command line parameter defaults
    builder_kwargs = builder_args

    if args.document_only:
        builder_kwargs["compile_gateware"] = False

    # Create and run the builder
    builder_kwargs["csr_csv"] = "build/csr.csv"
    builder = Builder(soc, **builder_kwargs)

    if args.build:
        builder.build()
        lxsocdoc.generate_docs(soc, "build/documentation/", project_name="LoRaDongle SOC")

    # If requested load the resulting bitstream onto the LoRaDongle
    if args.flash:
        IceStormProgrammer().flash(0x00000000, path.join(builder.gateware_dir,"{0}.bin".format(soc.build_name)))
Code Structure - Firmware
Csr.h - Collection of functions to interface with peripherals

Setting CSR Base Register

```c
#define CSR_SPI_MOSI_ADDR (CSR_BASE + 0x2808L)
#define CSR_SPI_MOSI_SIZE 1
static inline uint32_t SPI_mosi_read(void) {
    return csr_read_simple(CSR_BASE + 0x2808L);
}
static inline void SPI_mosi_write(uint32_t v) {
    csr_write_simple(v, CSR_BASE + 0x2808L);
}
#define CSR_SPI_MISO_ADDR (CSR_BASE + 0x280cL)
#define CSR_SPI_MISO_SIZE 1
static inline uint32_t SPI_miso_read(void) {
    return csr_read_simple(CSR_BASE + 0x280cL);
}
static inline void SPI_miso_write(uint32_t v) {
    csr_write_simple(v, CSR_BASE + 0x280cL);
}
#define CSR_SPI_CS_ADDR (CSR_BASE + 0x2810L)
#define CSR_SPI_CS_SIZE 1
static inline uint32_t SPI_cs_read(void) {
    return csr_read_simple(CSR_BASE + 0x2810L);
}
static inline void SPI_cs_write(uint32_t v) {
    csr_write_simple(v, CSR_BASE + 0x2810L);
}
```

eg: SPI MOSI, MISO and !SS
sx126x_hal_status_t sx126x_hal_write( const void* context, const uint8_t* command, const uint16_t command_length, const uint8_t* data, const uint16_t data_length )
{
    sx126x_hal_status_t status; // update to remove the 1

    // Setting Command and Message Information
    SPI_control_length_write(8);
    SPI_cs_mode_write(1);
    SPI_cs_sel_write(1);
    usleep(0.7);

    for (uint16_t i; i<command_length; i++) {
        SPI_mosi_write((uint32_t*)command++);
        SPI_control_start_write(0x1);
        usleep(0.5);
    }

    for (uint16_t i; i<data_length; i++){
        SPI_mosi_write((uint32_t*)data++);
        SPI_control_start_write(0x1);
        usleep(0.5);
    }

    SPI_cs_sel_write(0);

    status = (sx126x_hal_status_t) wait_tx_rx_done();

    return status;
}
sx126x.c

sx126x_status_t sx126x_write_register( const void* context, const uint16_t address, const uint8_t* buffer, const uint8_t size )
{
    const uint8_t buf[SX126X_SIZE_WRITE_REGISTER] = {
        SX126X_WRITE_REGISTER,
        ( uint8_t )( address >> 8 ),
        ( uint8_t )( address >> 0 ),
    };

    return ( sx126x_status_t )sx126x_hal_write( context, buf, SX126X_SIZE_WRITE_REGISTER, buffer, size );
}

sx126x_status_t sx126x_read_register( const void* context, const uint16_t address, uint8_t* buffer, const uint8_t size )
{
    const uint8_t buf[SX126X_SIZE_READ_REGISTER] = {
        SX126X_READ_REGISTER,
        ( uint8_t )( address >> 8 ),
        ( uint8_t )( address >> 0 ),
        SX126X_NOP,
    };

    return ( sx126x_status_t )sx126x_hal_read( context, buf, SX126X_SIZE_READ_REGISTER, buffer, size );
}
void transmit(RadioConfig_t *config, uint8_t size, uint8_t *message) {
    led_out_write(0b10);
    PrepareBuffer(config, size, message);
    ConfigureTx(config);
    set_to_transmit(config);
    led_out_write(0b00);
}

void PrepareBuffer(RadioConfig_t *context, uint8_t size, uint8_t *message) {
    uint8_t offset = 0x0;
    sx126x_write_buffer(&context, offset, message, size);
}

void ConfigureTx(RadioConfig_t *config) {
    sx126x_set_dio_irq_params(config, config->irqTx, config->irqTx, SX126X_IRQ_NONE, SX126X_IRQ_NONE);
}

void set_to_transmit(RadioConfig_t *config) {
    ClearAntSW();
    sx126x_set_tx(config, config->txTimeout);
}
#define MESSAGE_SIZE BUFFER_SIZE
RadioConfig_t context;
uint8_t send_message[MESSAGE_SIZE] = {"PING"};
uint8_t receive_message[MESSAGE_SIZE] = {};
uint8_t *send_message_ptr = send_message;
uint8_t *receive_message_ptr = receive_message;

int main(void) {
    time_init();
    uart_init();
    RadioInit(&context);
    SetConfiguration(&context);
    ConfigureGeneralRadio(&context);
    while (1) {
        /*** Transmitting Test *****/
        transmit(&context, sizeof(send_message), send_message_ptr);

        /*** Receiving Test *****/
        receive(&context, sizeof(send_message), receive_message_ptr);
        printf("%s\n", receive_message);
        msleep(1000);
    }
    return 0;
}
3. A quick Demonstration

Building SOC & Pinging between two LoRaDongles
Introduction

LoRaDongle is a project that aims to merge the Open Source LoRaWAN with the wireless modulation protocol LoRa.

Currently, the first prototype is in development in which various different tools and systems are brought together to build the project. The prototype is executed on a custom PCB that is populated with a low-power Lattice FPGA - the ICE40UPF that houses a software SoC. On the same board, a SX1290 LoRa transceiver is populated and configured to transmit at 866MHz.

The two prototypes and the schematic drawing pictured below.
Getting Started

The goal of this section is to build up to a working program on the SoC. The necessary links and steps to get to that will be shown here.

Prerequisites

Hardware

The current supported boards are the CuitaDongle which has the Lattice iCE40ULP, and the Any A7 development board by Xilinx which has the Artix-7 FPGA. To use the Any A7, you would need to clone the `xilinx-branch` branch of the repository.

Software

- **Linux**
- Binutils/GCC toolchain - which can be installed with `clang`
- The necessary synthesis and routing software for the FPGA is XST. For the LoRaDongle (and other Lattice FPGA) this is `ProjectXplore`. For Digilent boards, this is `Icarus`.

Table of Contents

- Installing Prerequisites
- Building the SoC for test
- Programming SoC
Introduction

The demo text application is a working implementation of a program running on the soc. You would obviously want to write your own firmware to run, and so this section describes how to do just that. It focuses on the code specifically, as the linker files are dependent on the soc and it's assumed the soc is unchanged.

Setting up the working directory

The location of the source files are not imperative, the only important thing is to ensure the makefile is aware of where to get them. For this tutorial, we will create a directory in the firmware directory, named blinky.

Copy the `blinky` directory found in the demo application to your new directory. The file is the same as the `soc` is unchanged. Next, you'll need a `main.c` and a `Makefile` to store the code and instructions. Your new directory should now look like this:

```
$ ls
main.c  Makefile
```

Writing C Code

What register to write to

The LED we will be turning on is the fit LED located below header u1k. Looking at the platform definition of the board, we see that the two LEDs are set up under the peripheral `user.leds`.

Documentation!

https://marrrk.github.io/LoRaDongle/
4. Future Plans

Immediate and beyond
Planned Tests: LoRa/Board/SOC Capabilities

- Rev 2.0 In early Stages of Testing
- The testing is planned to cover:
  - Latency
  - Bandwidth & Throughput
  - Range testing
  - Indoor testing
  - Benchmarking of SOC
- Connecting to a TTN LoRaWAN gateway!
Three main goals

**Improve SoC**
- Integrate unused FPGA fabric
- Debug functionality
- Refine build Script

**Embedded Linux**
- Requires a much larger FPGA.
- Would lead to greater contribution to Edge Computing.

**Used in industry**
- Entrepreneurs
- Build a community
- Giving Africa a platform
Thanks!

Contact me:

- Linkedin: [https://www.linkedin.com/in/mark-njoroge-7b4179185](https://www.linkedin.com/in/mark-njoroge-7b4179185)
- Email: marksnjoroge@gmail.com
- Website: [https://marrrk.github.io/](https://marrrk.github.io/)