28nm IoT SoC Design Review

March 16-18, 2021
UC Berkeley EE290C
Course History

- **2017**
  - David Burnett, Osama Khan
  - Taped out analog, RF
- **2018**
  - Osama Khan, Edward Wang
- **2019**
  - Edward Wang, Aviral Pandey, Hall Chen
- **2021**
  - Bora, Ali, Kris
  - Dan Fritchman, Aviral Pandey
Research Infrastructure

Bora Nikolić
28th-year grad student
Leveraging Research Infrastructure

- Processor core, interfaces
- Software tools

https://github.com/ucb-bar/chipyard

- Custom BLE digital baseband, accelerator wrappers

https://www.chisel-lang.org/

Berkeley Analog Generator (BAG) -
generated SAR ADC
Chip Intro & Overview
OSCI BEAR

Open-source SoC for IoT with BLE, AES, and Radio
Project Goals

● Lightweight BLE compatible SoC for IoT applications
  ○ Wearables, localized sensor networks, smart keys, etc

● Demonstrate agile hardware development flow
  ○ Using Berkeley designed tools

● Tape-out in TSMC 28nm technology
  ○ 1 mm² total die area
  ○ Off chip radio and clock components will be attached on PCB level
Chip Overview

PCB

Voltage/Clock Reference

Power/Clocking

On Chip

Digital Blocks

AES Accelerator

CPU

Digital Baseband

Off Chip Radio Components

Transceiver

RX

TX

switch1

switch2

Ref_CLK

VDD_D

GND

VDD_A

CLK_D

RoCCIO

DMA TL

SPI

JTAG

GPIO

MMIO TL

DMA TL

VCO ctrl

Gain ctrl

Filt ctrl
Compute Complex

- RV32IMAFC Core
- 32KB I$
- 32KB Data Memory (DTIM)
- JTAG
- SPI flash
- Interrupt Controller (PLIC)
AES Accelerator/Co-processor

- Efficient data security
- Performs AES Encryption/Decryption
  - Different modes orchestrated by software
- Built from open-source Secwork AES
  - Heavily tested
  - Taped-out verilog
- Non-blocking interaction with core
  - Buffer custom RISC-V instructions
  - DMA to handle memory operations
  - Supports both interrupts and polling
BLE Baseband and Modem

- BLE baseband paired with GFSK modem
- Receives instructions from core via MMIO and data from scratchpad via DMA
  - Notifies core of incoming messages via interrupt
- Supports all packet types for LE 1M specification
BLE Transceiver

- Low IF receiver architecture with tunable LO
- Analog PLL
- Digital control for GFSK modulation
## Target Specs

<table>
<thead>
<tr>
<th>Digital Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Clock</td>
</tr>
<tr>
<td>Core Power</td>
</tr>
<tr>
<td>AES Speedup</td>
</tr>
<tr>
<td>BLE Transmit Latency</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Analog Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
</tr>
<tr>
<td>BER</td>
</tr>
</tbody>
</table>
Infrastructure: Chipyard
Infrastructure: Hammer VLSI Flow
CPU
Team Introduction

Nayiri Krzysztofowicz
1st Yr PhD Student, advised by Bora

Josh Alexander
4th Yr EECS Undergrad

Cheng Cao
3rd Yr EECS Undergrad
Overview

- RV32IMAFC Core
- 32KB I$
- 32KB DTIM
- JTAG
- SPI flash
  - With XIP
  - Quad read/write
Memory subsystem

- Single core, uses TileLink protocol to connect peripheral devices
- The Data Tightly Integrated Memory in place of the L1D$, and can back the L1I$
- The TSI (Tethered Serial Interface module) can tunnel TileLink messages and provide memory backing through a debugging interface
- The QSPI flash has Execute In Place mode to back the instruction cache
- The debug module has a small SRAM to provide backing in case of other memory blocks are inaccessible
Boot Process (Boot ROM)

Two options:

1. **Boot select pin is high (Self-boot)**
   a. Boot directly into the SPI flash with execute in place mode
   b. The program can copy itself to the DTIM if needed

2. **Boot select pin is low (Tethered debug)**
   a. Setup a trap handler, and enter a wait for interrupt loop
   b. Use JTAG, TSI, or other external debugging tools to program the on-chip memory
   c. Use JTAG, TSI, or other external tools to trigger an exception
   d. Boot ROM receives exception, jumps to the programmed memory
JTAG Debug

**simulation:**

**chip bringup:** (**preliminary diagram**)
TSI Debug

**Simulation:**

- FESVR
- HTIF : TSI
- TSI Protocol
- Test Harness
- SerialAdapter
- SimSerial.v
- SimSerial.cc
- TLSerdesser
- Serialized TL
- Design Under Test (DUT)
- TLSerdesser
- TileLink Messages
- TileLink Memory Transactions

**Chip Bringup:**

- FPGA
- RISC-V Softcore SoC
- FESVR
- HTIF : TSI
- SerialAdapter
- DRAM
- TToAXI
- TLSerdesser
- Serialized TL
- Design Under Test (DUT)
- TLSerdesser
- TileLink Messages
- TileLink Memory Transactions
JTAG Simulation

**Host**

- **GDB**
- **OpenOCD**
- **JTAG**

**Test Harness**

**Simulation**

**JTAG Protocol**

**SimJTAG.v**

**SimJTAG.cc**

**Debug Transfer Module (DTM)**

**TileLink Messages**

---

**current output:**

```
[chipyard] mayi@1024cc:~/scratch/mayi/tstech28/chipyard3/sim/vcs (vcs4-digital) $ [chipyard] mayi@1024cc:~/scratch/mayi/tstech28/chipyard3/sim/vcs (vcs4-digital) $ open -f -e /home/mayi/cfg Open on-Chip Debugger (0.19.870-vdec-00049-cocb3c5501d (2021-03-08 12:43) Licensed under SP8 (v) v For bug reports, read http://openocd.org/doc/openjtag/bugs.html Info: only one transport option: autosel 'jtag' Info: Initializing remote breathing driver Info: Connecting to local device: 192.168.0.101 Info: Remote adapter 'jtag' on port 18 Info: [si] This adapter doesn't support configurable speed Info: JTAG uses 7-bits. tap/device found: 0x00000000 (tap: 0x00000000), part: 0x0000, ver: 0x0 for data0:0x0, progfuse:0x4 Info: Disabling abstract command reads from GDBs. Info: Examined RISC-V core; found 1 harts Info: hart #1: X1, X2, X3, X4, X5, X6, X7, X8 Info: Listening on port 3333 for gdb connections Info: Listening on port 3334 for tcl connections Info: Accepting gdb connection on tcp/3333 Warn: keep_all cấp was not invoked in the 100ms deadline. GOB alive packet not sent (28930). Workaround: increase "set remittence" in GOB Warn: keep_all cấp was not invoked in the 100ms deadline. GOB alive packet not sent (28930). Workaround: increase "set remittence" in GOB Warn: keep_all cấp was not invoked in the 100ms deadline. GOB alive packet not sent (28930). Workaround: increase "set remittence" in GOB Warn: keep_all cấp was not invoked in the 100ms deadline. GOB alive packet not sent (28930). Workaround: increase "set remittence" in GOB Warn: keep_all cấp was not invoked in the 100ms deadline. GOB alive packet not sent (28930). Workaround: increase "set remittence" in GOB Warn: keep_all cấp was not invoked in the 100ms deadline. GOB alive packet not sent (28930). Workaround: increase "set remittence" in GOB Warn: keep_all cấp was not invoked in the 100ms deadline. GOB alive packet not sent (28930). Workaround: increase "set remittence" in GOB Warn: keep_all cấp was not invoked in the 100ms deadline. GOB alive packet not sent (28930). Workaround: increase "set remittence" in GOB
```

---

**chipyard**

```
while (wait) {
    int = 0;
    while (!text[i]) {
        char lower = text[i] & 32;  
        if (lower == 'a' && lower == 'g')
            text[i] += 13;
        else if (lower > 'a' && lower < 'z')
            text[i] = text[i] += 13;
        i++;
    }
}
```

---

**chipyard**

```
chipyard] mayi@1024cc:~/scratch/mayi/tstech28/chipyard3/sim/vcs (vcs4-digital) $ make run-CI BIN=chipyard E2E0808Config BINARY=hellobug.exec CM_ROOT=src/main/resouces
```
Simulation & Software

- compile C benchmarks into rv32ui-imafc format
- able to load programs through JTAG
  - next steps: debug helloworld, software stack for cpu benchmarks, aes, baseband
- Chipyard default TSI infrastructure working in simulation
- OS / Software stack
  - RTOS is nice to have & possible to run
  - Hardware specd to be able to run a bluetooth stack on a RTOS
    - 32K data memory, instruction cache, etc.
  - Not our focus and not required, the test software will be baremetal
Interface with RF/Accelerators

RF

- Send
  - Send interrupt to send 2-158 bytes
  - Software handles encrypt/decrypt
  - Store accel response in register

- Receive
  - Set base address at onset of transmission
  - Command when to stop
  - Interrupts resolved in order at end
    - How many bytes
    - Valid/Invalid

AES

- Interacts directly with Rocket Core
- Uses DMA along with RF
- Space for text to decrypt/encrypt
Physical Design

- Digital Top
  - $VDD = 0.9 \text{ V}$
  - clock freq = 100MHz
  - LVT devices

- area = 0.193 mm$^2$
  (allocated 0.38 mm$^2$)
AES Accelerator
Team Introduction

Anson Tsai
5th Yr M.S., advised by Bora

Eric Wu
EECS M.Eng., advised by Kris

Daniel Fan
3rd Yr EECS undergrad
Overview Design

- RoCC (Rocket Custom Co-processor) AES Accelerator
- Building from open-source Secworks AES core
- Designing HW wrapper logic for rocket integration
- Custom RISC-V instructions
- Software stack to compile C code
Starting at the Core: Secworks AES

- Performs AES encryption and decryption
  - 128b and 256b keys
- Internal registers to hold key and text
- 51 cycles/block for 128b key
- 71 cycles/block for 256b key
- Well tested and mature
- Taped-out in other designs

```plaintext
Secworks AES Core

clock      = Clock
reset_n    = Bool
cs         = Bool
we         = Bool
address    = UInt(8.W)
write_data = UInt(32.W)
read_data  = UInt(32.W)
```
AES Core Usage

1. Load the key to be used by writing to the key register words.
2. Set the key length by writing to the config register.
3. Initialize key expansion by writing a one to the init bit in the control register.
4. Wait for the ready bit in the status register to be cleared and then to be set again. This means that the key expansion has been completed.
5. Write the cleartext block to the block registers.
6. Specify encryption/decryption by writing to the config register.
7. Start block processing by writing a one to the next bit in the control register.
8. Wait for the ready bit in the status register to be cleared and then to be set again. This means that the data block has been processed.
9. Read out the ciphertext block from the result registers.
## AES Core Usage

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Load the key to be used by writing to the key register words.</td>
<td>KEY SETUP</td>
</tr>
<tr>
<td>2.</td>
<td>Set the key length by writing to the config register.</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Initialize key expansion by writing a one to the init bit in the control register.</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Wait for the ready bit in the status register to be cleared and then to be set again. This means that the key expansion has been completed.</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Write the cleartext block to the block registers.</td>
<td>DATA LOAD</td>
</tr>
<tr>
<td>6.</td>
<td>Specify encryption/decryption by writing to the config register</td>
<td>ENCRYPT/DECRYPT</td>
</tr>
<tr>
<td>7.</td>
<td>Start block processing by writing a one to the next bit in the control register.</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>Wait for the ready bit in the status register to be cleared and then to be set again. This means that the data block has been processed.</td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>Read out the ciphertext block from the result registers.</td>
<td>DATA WRITE</td>
</tr>
</tbody>
</table>
What we need to implement:

- Instructions
  - Key Load + Setup
  - Data Load
  - Start Encryption/Decryption
  - Query Status

- Decoupler
  - Receive/process instructions for controller

- Controller
  - Handles AES Core operation
  - Queries DMA for data reading/writing

- DMA + Buffers
  - Handles memory operations
  - Buffers DMA packets and outputs 32b blocks
Custom RISC-V Instructions

1. Load key and set key size
   - funct7: 0 for 128b, 1 for 256b key size
   - rs1: key address
   - rs2: N/A
   - rd: N/A
   - (xd,xs1,xs2): (0,1,0)

2. Load text src and dest address
   - funct7: 2
   - rs1: text src address
   - rs2: text dest address
   - rd: N/A
   - (xd,xs1,xs2): (0,1,1)

3. Encrypt/Decrypt Blocks
   - funct7: 3 for encrypt, 4 for decrypt
   - rs1: # of 128b blocks to process
   - rs2: N/A
   - rd: N/A
   - (xd,xs1,xs2): (0,1,0)

4. Query Status (blocking)
   - funct7: 5
   - rs1: N/A
   - rs2: N/A
   - rd: destination register
   - (xd,xs1,xs2): (1,0,0)

Reserved opcodes for RoCC Accelerators

Register Control Fields
Instruction Interface - RoCCIO

- **Rocket Custom Co-processor IO**
- **Wrapper for RISC-V Instructions**
  - Commands (from CPU to accelerator)
  - Responses (from accelerator to CPU)
- **Status signals**
  - Busy, Exception, Interrupt
- **Other interfaces (not used)**
  - Page Table Walker (virtual mem)
  - FPU Requests and Response

---

RoCC Command

- `funct` = Bits(7.W)
- `rd` = Bits(5.W)
- `rs1` = Bits(5.W)
- `rs2` = Bits(5.W)
- `rd` = Bits(5.W)
- `opcode` = Bits(7.W)
- `rs1_data` = Bits(32.W)
- `rs2_data` = Bits(32.W)

---

RoCC Response

- `rd` = Bits(5.W)
- `rd_data` = Bits(32.W)

---

RoCC Decoupler

- `cmd` = RoCCCommand
- `resp` = RoCCResponse
- `busy` = Bool
- `interrupt` = Bool
- `exception` = Bool
RoCC Decoupler

- Receives/processes instructions
- Buffers data for controller
  - Outputs operate by Ready-Valid
  - Data grouped by operation/step
    - Key, Addr, Start
- Non-blocking interaction with CPU

<table>
<thead>
<tr>
<th>RV GROUP</th>
<th>OUTPUT (to ctrl)</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>key_addr</td>
<td>Address of Key</td>
</tr>
<tr>
<td>key</td>
<td>key_size</td>
<td>Size of Key</td>
</tr>
<tr>
<td>addr</td>
<td>src_addr</td>
<td>Address of source data</td>
</tr>
<tr>
<td>addr</td>
<td>dest_addr</td>
<td>Address to write result</td>
</tr>
<tr>
<td>start</td>
<td>block_count</td>
<td># of 128b blocks to process</td>
</tr>
<tr>
<td>start</td>
<td>op_type</td>
<td>Enc. or Dec.</td>
</tr>
</tbody>
</table>
RoCC Decoupler <> Controller Interface

<table>
<thead>
<tr>
<th>RoCC Decoupler</th>
<th>AES Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>excp_ready = Bool</td>
<td>excp_ready = Bool</td>
</tr>
<tr>
<td>excp_valid = Bool</td>
<td>excp_valid = Bool</td>
</tr>
<tr>
<td>interrupt = Bool</td>
<td>interrupt = Bool</td>
</tr>
<tr>
<td>key_ready = Bool</td>
<td>key_ready = Bool</td>
</tr>
<tr>
<td>key_valid = Bool</td>
<td>key_valid = Bool</td>
</tr>
<tr>
<td>key_size = UInt(1.W)</td>
<td>key_size = UInt(1.W)</td>
</tr>
<tr>
<td>key_addr = UInt(32.W)</td>
<td>key_addr = UInt(32.W)</td>
</tr>
<tr>
<td>addr_ready = Bool</td>
<td>addr_ready = Bool</td>
</tr>
<tr>
<td>addr_valid = Bool</td>
<td>addr_valid = Bool</td>
</tr>
<tr>
<td>src_addr = UInt(32.W)</td>
<td>src_addr = UInt(32.W)</td>
</tr>
<tr>
<td>dest_addr =UInt(32.W)</td>
<td>dest_addr = UInt(32.W)</td>
</tr>
<tr>
<td>start_ready = Bool</td>
<td>start_ready = Bool</td>
</tr>
<tr>
<td>start_valid = Bool</td>
<td>start_valid = Bool</td>
</tr>
<tr>
<td>op_type = Bool</td>
<td>op_type = Bool</td>
</tr>
<tr>
<td>block_count = UInt(32.W)</td>
<td>block_count = UInt(32.W)</td>
</tr>
</tbody>
</table>
Controller - AES Core Usage Recap

1. Load the key to be used by writing to the key register words.

2. Set the key length by writing to the config register.

3. Initialize key expansion by writing a one to the init bit in the control register.

4. Wait for the ready bit in the status register to be cleared and then to be set again. This means that the key expansion has been completed.

5. Write the cleartext block to the block registers.

6. Specify encryption/decryption by writing to the config register.

7. Start block processing by writing a one to the next bit in the control register.

8. Wait for the ready bit in the status register to be cleared and then to be set again. This means that the data block has been processed.

9. Read out the ciphertext block from the result registers.
Controller

- Controller consists of a main controller and a memory controller

- Main Controller
  a. Receive information from RoCC Decoupler
  b. Handle the workflow of the AES operations

- Memory Controller
  a. Query key and text data from DMA with DMA buffer
  b. Read/Write data from AES core
AES Core Usage

1. Load the key to be used by writing to the key register words.
2. Set the key length by writing to the config register.
3. Initialize key expansion by writing a one to the init bit in the control register.
4. Wait for the ready bit in the status register to be cleared and then to be set again. This means that the key expansion has been completed.
5. Write the cleartext block to the block registers.
6. Specify encryption/decryption by writing to the config register.
7. Start block processing by writing a one to the next bit in the control register.
8. Wait for the ready bit in the status register to be cleared and then to be set again. This means that the data block has been processed.
9. Read out the ciphertext block from the result registers.
Main Controller - FSM

When data_wr_done && remain > 0
remain = 0
key_rdy = 1
addr_rdy = 0
start_rdy = 0

When data_wr_done && remain == 0
When key_valid
key_rdy = 0
When keyexp done
Wait Data
When addr_valid
addr_rdy = 0
Wait Start
When start_valid
start_rdy = 1
When data_ld_done && remain == 0
remain -= 1
When AES_done
remain -= 1
When data_wr_done && remain > 0
When data_wr_done && remain == 0
Data Write

When data_id_done & remain > 0
When data_id_done & remain == 0
wait Data Setup
Wait Data Setup
Wait Start
Set AES operation type
Remain = X
Wait Result
Set aes start
register
Set key length
Set key init
register
AES Core Usage

1. Load the key to be used by writing to the key register words. *KEY SETUP*
2. Set the key length by writing to the config register.
3. Initialize key expansion by writing a one to the init bit in the control register.
4. Wait for the ready bit in the status register to be cleared and then to be set again. This means that the key expansion has been completed.
5. Write the cleartext block to the block registers. *DATA LOAD*
6. Specify encryption/decryption by writing to the config register *ENCRYPT/DECRYPT*
7. Start block processing by writing a one to the next bit in the control register.
8. Wait for the ready bit in the status register to be cleared and then to be set again. This means that the data block has been processed.
9. Read out the ciphertext block from the result registers. *DATA WRITE*
Main Controller - FSM

Idle

- When reset
  - remain = 0
  - key_rdy = 0
  - addr_rdy = 0
  - start_rdy = 0

Key Setup

- When key_valid
  - key_rdy = 0
  - Set key init register

Key Exp

- When key_ld_done
  - addr_rdy = 1

Wait Data

- When data_wr_done && remain > 0
  - remain = 0
  - key_rdy = 1
  - addr_rdy = 0
  - start_rdy = 0
- When data_wr_done && remain == 0

Data Setup

- When keyexp done
  - addr_rdy = 1

Wait Start

- When key_valid
  - key_rdy = 0
- When keyexp done
  - addr_rdy = 1

Wait Result

- When start_valid
  - start_rdy = 0
  - Set AES operation type
  - remain = X

AES Run

- When AES_done
  - remain = 1

Data Write

- When data_wr_done && remain > 0
- When data_wr_done && remain == 0

Set AES start register

Set key length

Set key init register
## AES Core Usage

1. Load the key to be used by writing to the key register words.  
   **KEY SETUP**
2. Set the key length by writing to the config register.
3. Initialize key expansion by writing a one to the init bit in the control register.
4. Wait for the ready bit in the status register to be cleared and then to be set again. This means that the key expansion has been completed.
5. Write the cleartext block to the block registers.  
   **DATA LOAD**
6. Specify encryption/decryption by writing to the config register  
   **ENCRYPT/DECRYPT**
7. Start block processing by writing a one to the next bit in the control register.
8. Wait for the ready bit in the status register to be cleared and then to be set again. This means that the data block has been processed.
9. Read out the ciphertext block from the result registers.  
   **DATA WRITE**
Main Controller - FSM

When reset
remain = 0
key_rdy = 0
addr_rdy = 0
start_rdy = 0

When key_valid
key_rdy = 1
Set key length

When key ld done
Set key init register

When keyexp done
addr_rdy = 1

When data_wr_done && remain > 0
remain = 0
key_rdy = 1
addr_rdy = 0
start_rdy = 0

When data_wr_done && remain == 0

When key_valid
key_rdy = 0

When keyexp done
Wait Data

When addr_valid
addr_rdy = 0
Wait Start

When start valid
start_rdy = 1
Set AES operation type
remain = X

When AES_done
remain -= 1

When data ld done & remain > 0
set aes start register

When data ld done & remain == 0

When data ld done & data_wr_done & remain > 0

Data Write

Set aes start register

AES Core Usage

1. Load the key to be used by writing to the key register words.
2. Set the key length by writing to the config register.
3. Initialize key expansion by writing a one to the init bit in the control register.
4. Wait for the ready bit in the status register to be cleared and then to be set again. This means that the key expansion has been completed.
5. Write the cleartext block to the block registers.
6. Specify encryption/decryption by writing to the config register
7. Start block processing by writing a one to the next bit in the control register.
8. Wait for the ready bit in the status register to be cleared and then to be set again. This means that the data block has been processed.
9. Read out the ciphertext block from the result registers.
Main Controller - FSM

- **Idle**
  - \( \text{When \ reset} \)
  - \( \text{remain} = 0 \)
  - \( \text{key}_{\text{rdy}} = 0 \)
  - \( \text{addr}_{\text{rdy}} = 0 \)
  - \( \text{start}_{\text{rdy}} = 0 \)

- **Key Setup**
  - \( \text{When \ key}_{\text{valid}} \)
  - \( \text{key}_{\text{rdy}} = 0 \)
  - \( \text{Set \ key \ length} \)

- **Key Exp**
  - \( \text{When \ key}_{\text{id \ done}} \)
  - \( \text{Set \ key \ init \ register} \)

- **Wait Data**
  - \( \text{When \ key}_{\text{exp \ done}} \)
  - \( \text{addr}_{\text{rdy}} = 1 \)

- **Data Setup**
  - \( \text{When \ addr}_{\text{valid}} \)
  - \( \text{addr}_{\text{rdy}} = 0 \)

- **Wait Start**
  - \( \text{When \ start}_{\text{valid}} \)
  - \( \text{start}_{\text{rdy}} = 1 \)

- **AES Run**
  - \( \text{Set \ aes \ start \ register} \)

- **AES Done**
  - \( \text{remain} -= 1 \)

- **Data Write**
  - \( \text{When \ data}_{\text{wr \ done}} \&\& \text{remain} > 0 \)
  - \( \text{When \ data}_{\text{wr \ done}} \&\& \text{remain} == 0 \)
Idle

When reset
remain = 0
key_rdy = 0
addr_rdy = 0
start_rdy = 0

Key Setup

When key_valid
done
key_rdy = 0
Set key init
_register

Key Exp

When keyExp
done
addr_rdy = 1

Wait Data

When addr_valid
remain > 0
addr_rdy = 0

Data Setup

When addr_valid
remain = 0
start_rdy = 0

Start

When start_valid
remain > 0

Wait Result

Set aes start
register

AES Run

When AES_done
remain = 1

Wait Data

When data_wr_done
&& remain > 0

When data_wr_done
&& remain == 0

Main Controller - FSM

Set key length

Set AES
operation type
remain = X

Set key init
_register
Memory Controller - FSM

- **Write Req**
  - When enqueue fire
  - Send AES write
- **Write Into Mem**
  - When write_data
  - counter += 1
  - Send AES read
- **Idle**
  - When reset
  - Counter = 0
  - When load_data
- **Read Req**
  - When DMA read req fire
  - Send DMA read req
  - counter += 1
- **Read Into AES**
  - When dequeue fire
  - Send AES write
  - counter += 1

counter = mem_target

counter = 0

counter = 0
- Services memory requests from controller
- Contains reader and writer sub-blocks
  - Concurrent read + write operations
- Interfaces with memory bus via TileLink
  - TileLink is a chip-scale interconnect in Rocket
- Shared DMA design in baseband module
  - More details presented in that presentation

**DMA Read Request**

\[
\text{addr} = \text{UInt}(32.W) \\
\text{totalBytes} = \text{UInt}(9.W)
\]

**DMA Write Request**

\[
\text{addr} = \text{UInt}(32.W) \\
data = \text{UInt}(X.W) \\
\text{totalBytes} = \text{UInt}(Y.W)
\]

**DMA Read Data**

\[
data = \text{UInt}(X.W)
\]
DMA Buffers

- AES Core operates with 32-bit blocks
- DMA “packet size” may be different
- DMA Input Buffer
  - Takes 32-bit blocks and breaks into packet-sized blocks if required
- DMA Output Buffer
  - Takes packet-sized blocks and breaks/combines into 32-bit blocks
AES Accelerator Top-Level Diagram

AES RoCC Accelerator

Verilog Black Box
(Secworks AES)

RoCC Decoupler

Controller

DMA Buffer

DMA

Rocket Core

RoCCIO

Frontend Bus

TL-UL
Verification: Secworks AES Core

- Given TB alongside RTL
- Flexible design
  - Option to add keys/input/expected
  - Generate own randomized keys/input
  - AES Library to calculate expected

```plaintext
*** TC 22 ECB mode test started.
cycle: 0x0000000000000069
State of DUT
----------
ctrl_reg: init = 0x0, next = 0x0
config_reg: encdec = 0x0, length = 0x1
block: 0xb6ed21b9, 0x9ca6f4f9, 0xf153e7b1, 0xbeafed1d

*** TC 22 successful.

*** TC 23 ECB mode test started.
cycle: 0x0000000000000055
State of DUT
----------
ctrl_reg: init = 0x0, next = 0x0
config_reg: encdec = 0x0, length = 0x1
block: 0x23304b7a, 0x39f9f3ff, 0x067d8d8f, 0x9e24ecc7

*** TC 23 successful.

*** All 16 test cases completed successfully

*** AES simulation done. ***
```
Verification: Accelerator Testbench Design

- **Stimulus Generator (Random addr, keys, text)**
- **Input Driver (RoCCIO)**
- **Input Monitor (RoCCIO)**
- **AES Accelerator**
- **Result Checker (Standard AES Library)**
- **TL Memory Model**

The diagram shows the flow of the testbench design, where the stimulus generator provides random addresses, keys, and text to the input driver and monitor (both RoCCIO). These components interact with the AES accelerator and the TL memory model. The result checker uses a standard AES library to verify the correctness of the accelerator's output.
Software

● How does a user code for the accelerator?
  ○ Include the C header to access accelerated crypto routines
  ○ Different routines implement different cipher modes in software
  ○ Routines also handle key setup and polling
  ○ Provides a level of abstraction above embedded assembly
RoCC Assembly

- Handy pre-existing repo (IBM/rocc-software) defines some useful RoCC generating macros
- Header file implements our extensions with said macros
Encryption modes

- The accelerator only implements the block cipher, so we have to implement individual cipher modes in software.
- For ECB, we can tell the accelerator the number of blocks we want to encrypt.
- For other modes like CBC, CTR, we have to encrypt blocks one at a time.

```c
static int AES_ECB_encrypt(uint8_t* key, int keylen, uint8_t* src, uint8_t* dest, int length) {
    if (keylen == 128) {
        aes_extended_keysetup128(*key);
    } else if (keylen == 256) {
        aes_extended_keysetup256(*key);
    } else {
        return 1; // other key lengths not supported
    }
    aes_extended_addressload(src, dest);
    aes_extended_encryptblocks((int) (length / 16)); // AES block length == 128 bits == 16 bytes
    int status = 1;
    while (status)
        aes_extended_querystatus(status);
    return 0;
}
```
Status

- **Implementation - Completed**
  - All sub-blocks are implemented and sanity checked with unit-level tests

- **Software Stack**
  - Completed intrinsics of custom RISC-V instructions
  - In process of implementing various AES Modes

- **Verification**
  - Subcomponent verification completed
  - Currently setting up accelerator-level testbench
  - Integrated into SoC with software tests pending
RF Baseband
Team Introduction

Ryan Lund
5th Year M.S. Student

Griffin Prechter
5th Year M.S. Student
Functional Goals

- Send and receive 1M Uncoded Link Layer Packets
  - With a design that can be expanded for more packet types (2M, LE Coded) over time
- Manage RF tuning constants
  - Some are CPU set while automatically tuned (AGC)
- Resilience to real world circumstances
  - Demodulation resilient to noise
  - Packet disassembler gracefully handled malformed packets
- Target 20μS transmit latency (command start to analog RF start)
- Meet BLE timing requirements
  - Allow CPU to have enough control over TX and RX; appropriate interrupts
Design: Overview

- Five main functional blocks
  - DMA
  - MMIO Frontend
  - Baseband
  - Controller
  - Modem

- Interface between digital and analog
  - Modem interacts with ADC/DAC
  - Frontend contains tuning registers for RF components
Design: MMIO Frontend

- TL register router component exposes registers as memory mapped devices
  - Command register from CPU
  - Status registers for CPU
  - Analog tuning parameters (e.g. I/Q filter resistors)
- Houses connection to interrupt bus and PLIC
- Parameterizable to allow for attachment at different base addresses
Design: DMA Interface

- Stream reader and writer generate TL transactions
  - Get, put, put partial
- Single command for reads
- Multiple commands for writes
  - TL bus width of data per command
- Allows for simultaneous reading and writing
Design: Baseband

- **Send Command:**
  - Packet Assembler begins the bit-streaming process, pulling data from specified address via DMA.

- **Receive Command:**
  - Packet Disassembler begins to wait for preamble; will write to specified address via DMA.

- **Debug Command:**
  - Both TX and RX chains active, loopback enabled to allow for loopback testing.

- **PD sends interrupts to alert the CPU of packet reception and success/failure.**
Design: Baseband

Bit Stream Processing

- Cyclic Redundancy Check (CRC): implemented as an LFSR

- Data Whitening: intended to avoid continuous streams of all 1s or all 0s, implemented as an LFSR
Design: Modem

- Designed for GFSK
- **TX side**
  - Gaussian filter on bitstream input from baseband (with per bit over-sampling)
  - Resulting value mapped to LO / PLL control signals via programmable LUT
- **RX side**
  - Async FIFO accounts for non-sync ADC logic
  - AGC performed on FIFO output
  - Image rejection uses Hilbert Filter
  - Non-coherent demodulation with bandpass and envelope detection
Design: Modem

Digital Image Rejection

- Hilbert Filter implemented as a 29 tap FIR filter with fixed point arithmetic
- CPU-configurable control signal to choose between adding and subtracting between I and Q signals.
Design: Modem

Modulation and Demodulation

- Modulation performed using integral tap Gaussian FIR filter
- Demodulation performed using dual bandpass filters at $W_0$ and $W_1$ followed by envelope detection
  - Decision made by symbol time majority

Data Input vs Gaussian FIR Output
Design: Controller

- Commands contain 4 fields
  - Inst. 1: Primary function
  - Inst. 2: Sub-function
  - Data: Small values
  - Additional data: Wide values

- Synchronizes DMA interface, baseband, and modem for command execution
  - Configures loopbacks for debug execution

- Manages interrupts and status registers

<table>
<thead>
<tr>
<th>Bits</th>
<th>31-8</th>
<th>7-4</th>
<th>3 - 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
<td>Data</td>
<td>Instruction 2</td>
<td>Instruction 1</td>
</tr>
<tr>
<td>Bits</td>
<td>31-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Additional Data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Design: Controller

Controller FSM

- TX Chain Controller
- RX Chain Controller

States:
- IDLE
- SEND CMD
- RECEIVE CMD
- DEBUG CMD
- LOOPBACK DONE
- FINISHED SENDING
- FINISHED RECEIVING

Transitions:
- TX → IDLE: Finished Sending
- IDLE → TX: SEND CMD
- RX → IDLE: Received CMD
- IDLE → RX: RECEIVE CMD
- DEBUG → IDLE: DEBUG CMD
- IDLE → DEBUG: LOOPBACK DONE
- IDLE → TX: CONFIG CMD
Verification: Testbench Design
Verification: Stimulus Generation

- Fully Deterministic Stimulus
- Partially Random Stimulus
- Fully Random Stimulus
- Directed Stimulus

- Bug Fix
- Test Failure
Software

- MMIO controls are easy to interface with in C
  - No special ISA extensions required
  - Generic mmio.h file allows for register reads and writes from 8 to 64-bits
- Series of macros defined to streamline interface
- Work to be done to integrate with chosen RTOS

```c
// Address map
#define BASEBAND_INST 0x8000
#define BASEBAND_ADDITIONAL_DATA 0x8004
#define BASEBAND_STATUS0 0x8008
#define BASEBAND_STATUS1 0x800C
#define BASEBAND_STATUS2 0x8010
#define BASEBAND_STATUS3 0x8014
#define BASEBAND_STATUS4 0x8018

// Instruction macro
#define BASEBAND_INSTRUCTION(primaryInst, secondaryInst, data) ((primaryInst & 0xF) + ((secondaryInst & 0xF0) << 4) + ((data & 0xFFFFFFFF) << 24))

// Primary instructions
#define BASEBAND_CONFIG 0
#define BASEBAND_SEND 1
#define BASEBAND_RECEIVE 2
#define BASEBAND_RECEIVE_EXIT 3
#define BASEBAND_DEBUG 15

// Secondary instructions
#define BASEBAND_CONFIG_CRC_SEED 0
#define BASEBAND_CONFIG_ACCESS_ADDRESS 1
#define BASEBAND_CONFIG_CHANNEL_INDEX 2
#define BASEBAND_CONFIG_ADDITIONAL_FRAME_SPACE 3
#define BASEBAND_CONFIG_LO_LUT 4

reg_write32(BASEBAND_ADDITIONAL_DATA, 5);
reg_write32(BASEBAND_INST, BASEBAND_INSTRUCTION(BASEBAND_CONFIG, BASEBAND_CONFIG_CHANNEL_INDEX, 0));
```
Into the Weeds: How the Baseband is Integrated

- Baseband attachment required custom Chisel
  - Four sections that need connecting: DMA to SBus, MMIO to FBus, Interrupt to IBus, Analog IO to ChipTop
- Accomplished via mixin, IO punch throughs, and harness binders (for simulation)
  - The magic: This is all automated, the connections can be made invariant to Baseband changes
  - Allows us to accommodate a continual stream of analog interface changes

```java
trait CanHavePeripheryBLEBasebandModem { this: BaseSubsystem =>
  val baseband = p(BLEBasebandModemKey).map { params =>
    val baseband = LazyModule(new BLEBasebandModem(params,
        fbus.beatBytes))
    pbus.toVariableWidthSlave(Some("baseband")) { baseband.mmio }
    fbus.fromPort(Some("baseband"))():= baseband.mem
    ibus.fromSync := baseband.intnode
    val io = InModuleBody {
      val io = IO(new
        BLEBasebandModemAnalogIO(params)).suggestName("baseband")
      io <> baseband.module.io
      io
    }
  }
}

class WithBLEBasebandModemPunchthrough(params: BLEBasebandModemParams = BLEBasebandModemParams()) extends OverrideIOBinder{
  (system: CanHavePeripheryBLEBasebandModem) => {
    val ports: Seq[BLEBasebandModemAnalogIO] = system.baseband.map({ a =>
      val analog = IO(new
        BLEBasebandModemAnalogIO(params)).suggestName("baseband")
      analog <> a
      analog
    }).toSeq
    (ports, Nil)
  }
}
```
Status

- **Baseband**
  - Fully implemented
  - Highly tested on a unit level
  - Integrated into SoC with software tests pending
- **Modem**
  - Work in progress, constantly evolving
  - Targeting completion by end of Spring Break
- **Software stack**
  - In preliminary phase
  - To date, used mainly for SoC integration testing of components
Integration
Introduction

Troy Sheldon
3rd Year EECS Undergrad

Dylan Brater
3rd Year EECS Undergrad

Jackson Paddock
5th Yr M.S., advised by Prof. Pister
Focus on analog circuit design

Kareem Ahmad
5th Yr M.S., advised by Prof. Sophia Shao
Focus on HW for ML

Dylan Brater
3rd Year EECS Undergrad
Floorplan
Layout and Planning

- Manual Layout
- 9 Metal layers
- M8 and M9 = Power Grid
- M6 and M7 = Inter-block connections
Verification Methodology

- test_bench.v
  - Transmitter_testbench.v
  - Check
  - Data Generation (to Rx)
  - Probe (from TX/GPIO)

- Chip_top.v
  - power_top.v
  - Digital_top.v
  - Analog_top.sv
Verification Methodology

On Chip RF IC

Behavioral Verilog Model

Analog_top.sv

testbench.v

Digital Baseband
Verification Methodology

Early Stage Goals:
● Minimal design that only replicates Rx
● Ignores tuning bits from Baseband
● Receives a high frequency simple square wave from Receiver and outputs a lower frequency square wave from adc->modem

End Goals:
● Replicates both Tx and Rx
● Receives full sinusoid signal to be modulated based on tuning bits from baseband
● Outputs descretised sinusoid from adc->baseband
● Models LO and outputs full sinusoid from Tx->off chip
RF Transceiver
Team Introduction

Alex Moreno  
Ph.D. Student

Shreesh S  
Ph.D. Student

Kerry Yu  
4th Year Undergrad

Leon Wu  
4th Year Undergrad

Sherwin  
4th Year Undergrad

Felicia Guo  
Ph.D Student

Jeffrey Ni  
4th Year Undergrad
Overview

Target <5mW power consumption
RX Specifications

● Input range of -70dBm ~ -10dBm
  ○ Translates to a gain range of 2-60dB

● BER of 0.1% under the following conditions (-67dBm input signal power)
  ○ Translates to SNR of 12.5dB

<table>
<thead>
<tr>
<th>Interference Band</th>
<th>Pcarrier</th>
<th>Pinterference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Co-channel</td>
<td>21dB</td>
<td>-89dBm</td>
</tr>
<tr>
<td>Adj. 1MHz</td>
<td>15dB</td>
<td>-82dBm</td>
</tr>
<tr>
<td>Adj. 2MHz</td>
<td>-17dB</td>
<td>-44dBm</td>
</tr>
<tr>
<td>Adj. &gt;= 3MHz</td>
<td>-27dB</td>
<td>-34dBm</td>
</tr>
<tr>
<td>Image</td>
<td>-9dB</td>
<td>-58dBm</td>
</tr>
<tr>
<td>Image +/- 1MHz</td>
<td>-15dB</td>
<td>-52dBm</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Band</th>
<th>Pinterference</th>
<th>Meas. Res</th>
</tr>
</thead>
<tbody>
<tr>
<td>30MHz - 2000MHz</td>
<td>-30dBm</td>
<td>10MHz</td>
</tr>
<tr>
<td>2003MHz - 2399MHz</td>
<td>-35dBm</td>
<td>3MHz</td>
</tr>
<tr>
<td>BLE channels here</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2484MHz - 2997MHz</td>
<td>-35dBm</td>
<td>3MHz</td>
</tr>
<tr>
<td>3000MHz - 12.75GHz</td>
<td>-30dBm</td>
<td>25MHz</td>
</tr>
</tbody>
</table>
TX Specifications

● Symbol period: 1us
● Gaussian Mask - BT=0.5
● Modulation index between 0.45 ~ 0.55
  ○ 0.495~0.505 for ‘stable modulation’
  ○ Target 0 = fc-250kHz; 1 = fc+250kHz
● Max center freq. drift: 150kHz
● Maximum freq drift: 50kHz
● Maximum drift rate: 400Hz/us
● Transmission power met using off chip PA
Receiver Link Budget + Modeling

<table>
<thead>
<tr>
<th>Antenna</th>
<th>Rin (ohms)</th>
<th>Rout (ohms)</th>
<th>Gain (dB, 20log)</th>
<th>Max_in (V)</th>
<th>Output Noise (V integrated)</th>
<th>Total Gain (dB)</th>
<th>Total SNR (dB)</th>
<th>input voltage</th>
<th>signal amp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matching Network</td>
<td>50.00</td>
<td>225.00</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-6.021</td>
<td>-12.041</td>
<td>2.50E-04</td>
<td></td>
</tr>
<tr>
<td>Mixer</td>
<td>225.00</td>
<td>400.00</td>
<td>30</td>
<td>0.007</td>
<td>2.14E-04</td>
<td>2.14E-04</td>
<td>17.959</td>
<td>17.919</td>
<td>7.00E-03</td>
</tr>
<tr>
<td>Buffer</td>
<td>1.00E+99</td>
<td>1.000.00</td>
<td>0</td>
<td>0.1</td>
<td>7.00E-05</td>
<td>7.00E-05</td>
<td>17.959</td>
<td>17.334</td>
<td>7.18E-03</td>
</tr>
<tr>
<td>VGA</td>
<td>1.00E+99</td>
<td>1.000.00</td>
<td>12</td>
<td>0.1</td>
<td>8.00E-04</td>
<td>8.00E-04</td>
<td>29.959</td>
<td>15.845</td>
<td>7.91E-03</td>
</tr>
<tr>
<td>BPF</td>
<td>1.00E+99</td>
<td>1.000.00</td>
<td>12</td>
<td>0.1</td>
<td>8.00E-04</td>
<td>8.00E-04</td>
<td>31.959</td>
<td>15.610</td>
<td>9.74E-04</td>
</tr>
<tr>
<td>VGA</td>
<td>1.00E+99</td>
<td>1.000.00</td>
<td>12</td>
<td>0.1</td>
<td>8.00E-04</td>
<td>8.00E-04</td>
<td>43.959</td>
<td>15.348</td>
<td>3.83E-04</td>
</tr>
</tbody>
</table>

- Modelling also in virtuoso
  - Combination of verilogA and discrete ideal components
  - Similar to link budget in variables that can be adjusted
Mixer

- Passive mixer for low power
- Differential Voltage gain: 37dB
- Maximum input voltage: ~6mV
- $\text{NFdsb: } 3.56 \text{ dB integrated}$
- Power consumption: ~1mW
  - ~500uW for the LO buffers
  - ~450uW for the TIA
VGA

- **VGA design**
  - 0-30 dB gain range for each VGA
  - $R_1$ steps in 6 dB, $R_3$ steps in 2 dB
  - $BW(30 \text{ dB}) = 6.85 \text{ MHz}$ with $300\text{pF}$ load
  - $R_{1,\text{min}} = 5 \text{ Kohm}$, $R_{3,\text{min}} = 80 \text{ Kohm}$

- **Fully differential amplifier design**
  - $V_{i,\text{CM}} = V_{o,\text{CM}} = 0.6\text{V}$
  - $A = 98 \text{ dB}$, $BW = 3.15 \text{ kHz}$
  - Current consumption = 270uA
  - Miller compensation, $PM = 57$
  - Input swing 0.3-0.9 V; output swing 0.15-0.8 V

- **Noise @ 30dB gain**
  - Input spot(2MHz) = 15 nV/$\sqrt{\text{Hz}}$
  - Integrated output noise (1-3 MHz) = 678 uV
VGA

Monte Carlo @ 30dB gain:

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>std dev</th>
<th>max</th>
<th>min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalent input offset (mV)</td>
<td>0.006</td>
<td>1.25</td>
<td>3.58</td>
<td>-3.47</td>
</tr>
<tr>
<td>PSRR (dB)</td>
<td>60.6</td>
<td>9.69</td>
<td>110</td>
<td>46.2</td>
</tr>
<tr>
<td>CMRR (dB)</td>
<td>82.0</td>
<td>11.1</td>
<td>130</td>
<td>56.6</td>
</tr>
</tbody>
</table>

Corner simulation @ 30dB gain:

<table>
<thead>
<tr>
<th></th>
<th>TT(27°)</th>
<th>Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC gain (dB)</td>
<td>30</td>
<td>29.9</td>
</tr>
<tr>
<td>Gain @ 2MHz (dB)</td>
<td>29.5</td>
<td>28.6</td>
</tr>
<tr>
<td>3dB BW (MHz)</td>
<td>5.44</td>
<td>3.39</td>
</tr>
<tr>
<td>output noise (uV)</td>
<td>684</td>
<td>813</td>
</tr>
</tbody>
</table>

Transient analysis: Switching from unity to 6dB gain
Bandpass Filter

- Multiple feedback high pass followed by low pass
  - Opamp: single stage with CMFB
- Gain of 3dB unloaded, and about unity when loaded by next stage
- Resistors adjustable ±50% to account for variation and maintain passband across temperature

- Noise:
  - Input spot @2MHz: 80nV/√Hz
  - Output integrated (1-3MHz): 170uVrms
- Group delay: 168ns (0) to 139ns (1)

No tuning.
LO:

- $F = 4.8 \text{ GHz}$
- $P = 360 \text{ uW}$
- $V_{pp} = 0.814 \text{ V}$
- $Q = 15$
- $L = 3.472 \text{ nH}$

Figure 5.12: Local Oscillator Schematic
LO:

- $F = 4.8$ GHz
- $P = 360$ uW
- $V_{pp} = 0.814$ V
- $Q = 15$
- $L = 3.472$ nH
LO:

- $F = 4.8 \text{ GHz}$
- $P = 360 \text{ uW}$
- $V_{pp} = 0.814 \text{ V}$
- $Q = 15$
- $L = 3.472 \text{ nH}$
LO:

- $F = 4.8 \text{ GHz}$
- $P = 360 \mu\text{W}$
- $V_{pp} = 0.814 \text{ V}$
- $Q = 15$
- $L = 3.472 \text{ nH}$

<table>
<thead>
<tr>
<th>Code</th>
<th>Freq (GHz)</th>
<th>-27</th>
<th>27</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>5.134</td>
<td>5.126</td>
<td>5.120</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>error</td>
<td>4.575</td>
<td>4.572</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>PN (dBc/Hz)</th>
<th>-121.7</th>
<th>-120.8</th>
<th>-119.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>-121.7</td>
<td>-120.8</td>
<td>-119.8</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>error</td>
<td>-119.8</td>
<td>-118.9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Amp</th>
<th>815.417</th>
<th>801.446</th>
<th>777.969</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>815.417</td>
<td>801.446</td>
<td>777.969</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>error</td>
<td>585.71</td>
<td>589.93</td>
<td></td>
</tr>
</tbody>
</table>
**LO:**

- F = 4.8 GHz
- P = 360 uW
- Vpp = 0.814 V
- Q = 15
- L = 3.472 nH

<table>
<thead>
<tr>
<th>Code</th>
<th>Freq (GHz)</th>
<th>-27</th>
<th>27</th>
<th>70</th>
<th>FF+-27</th>
<th>SS+70</th>
<th>PN (dBc/Hz)</th>
<th>-121.7</th>
<th>-120.8</th>
<th>-119.8</th>
<th>117.2</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>5.134</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-121.7</td>
<td>-120.8</td>
<td>-119.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>error</td>
<td>4.575</td>
<td>4.572</td>
<td>5.307</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>error</td>
<td>-120.8</td>
<td>-119.8</td>
<td>-117.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>error</td>
<td>-119.8</td>
<td>-118.9</td>
<td>-117.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>Amp (mV)</td>
<td>815.417</td>
<td>801.446</td>
<td>777.969</td>
<td>856.788</td>
<td>error</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>error</td>
<td>585.71</td>
<td>589.93</td>
<td>779.946</td>
<td></td>
<td>error</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LO: Cap DAC

- 4 bit Cap Bank
- $F_{\text{min}}$ target = 4.56 GHz
- $F_{\text{max}}$ target = 5.21 GHz
- ~40 MHz per unit cap
LO: Varactor

- Range 200mV-800mV
- Delta C = 7.2556 fF
- ~70MHz
- KVCO=115MHz/V
PLL

$F_{ref} = 2$ MHz

$\phi_{in}$

PFD

Charge Pump

Loop Filter

Coarse Tune

GFSK Tune

$F_{vco} = 4.804 - 4.960$ GHz

$\phi_{out}$

$F_{out} = 2.402 - 2.480$ GHz

/N

/2

12

Channel Select
PLL Loop Filter

- Closed loop bandwidth placed low enough to stay locked while not rejecting 1 MHz GFSK during TX
- \( f_{BW} = 100 \text{ kHz} \)
- \( f_z/f_{BW} = 8 \)
- Estimated Area: 250um x 250um

\[ \Phi_{\text{out}} / \Phi_{\text{ref}} \]
\[ \Phi_{\text{out}} / \Phi_{\text{vn}} \]

0 dB at 1 MHz
Charge Pump

- Cascode current source with wide swing current mirrors
  - Designed for $I_{OUT} = 150 \, \mu A$
  - Estimated Power Consumption: 300 uW

![Charge Pump Diagram]

![Graphs of $I_{UP}$ vs $V_o$ and $I_{DOWN}$ vs $V_o$]
Phase/Frequency Detector

Stability issues when $\phi_{\text{ref}}$ and $\phi_{\text{fb}}$ both high
17GHz oscillations on up and down

FB / REF / UP / DOWN
Phase/Frequency Detector

Fixed by adding buffers and using slower NAND

FB / REF / UP / DOWN
Pre-scalar/Mixer LO Distribution

- CML Logic for better differential output balance
  - <4ps output pulse width variation
- Takes 1 clock cycle to settle
- 130uW power consumption
Current Receiver Results

- Functional first pass integration measured at -70dBm input
- Sufficient Gain (35dB voltage gain measured)
- NFdsb of 4.2dB
- Power Consumption: 2.817mW
- Next Steps
  - Operating range expansion
  - Use real components for passives
Analog & Power
Introduction

- Jackson Paddock
  - 5th year MS student in Prof. Kristofer Pister’s lab
  - Focus on analog circuit design (specifically LDOs)
Power Domains

- Three power domains on the chip: analog, digital, and I/O
- Analog:
  - Includes: RF (PLL, LO, mixer, amplifiers, ADCs)
  - Analog LDO (VDDA) supplies 5.75mA at 0.9V
- Digital:
  - Includes: CPU, accelerators, digital baseband
  - Digital LDO (VDDD) supplies 47.2mA at 0.9V
- I/O:
  - Includes: pad ring
  - VDDIO comes from an off-chip supply
Current Reference Topology

- Currently, there is no bandgap reference on chip so the voltage and current references are from off chip (0.9V and 10uA respectively)
- With an input NMOS device, the 10uA reference is mirrored up to PMOS devices on:
  - Vbat, and then brought down for the tail current mirrors of the analog and digital LDOs
  - VDDA and VDDD for current references in the analog and digital circuits on chip
- All devices shown have max channel length to reduce variation from VDS
LDO Topology

- Both the digital and analog LDOs use the same topology: a simple diff pair amplifier with a PMOS pass device
- Both LDOs use the same amplifier and have outputs at 0.9V
- The dominant pole is at the output with an off chip capacitor (due to area) for stability
LDO Design Tradeoffs

- **Pole location:**
  - Not enough information is known about the load capacitance from other blocks at the moment.
  - A dominant pole at the gate of M5 reduces PSRR bandwidth significantly and may cause stability issues if the chip’s built-in capacitance is too large.

- **Power vs Stability:**
  - As the relative width of all devices in the amplifier increases, the output resistance decreases and pushes the secondary pole higher (increasing the phase margin).
  - Wider devices mean higher current consumption in the amplifier.
**LDO Specs (TT, 27°C)**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Error</td>
<td>35.4μV</td>
</tr>
<tr>
<td>Amplifier Current</td>
<td>211.1μA</td>
</tr>
<tr>
<td>Phase Margin</td>
<td>75.35°</td>
</tr>
<tr>
<td>PSRR</td>
<td>51.88dB</td>
</tr>
<tr>
<td>PSRR bandwidth</td>
<td>1.992MHz</td>
</tr>
<tr>
<td>Line Regulation*</td>
<td>99.95m</td>
</tr>
<tr>
<td>Load Regulation*</td>
<td>412.0μ</td>
</tr>
<tr>
<td>Load Pole (C_L=100nF)</td>
<td>4.498kHz</td>
</tr>
<tr>
<td>Second Pole (Gate of M5)</td>
<td>1.434MHz</td>
</tr>
</tbody>
</table>

*Simulated with output device set for 5.75mA
**Simulated with 20% variance peak to peak of reference voltage and load current

![LDO Circuit Diagram]
## Corner and Temperature Simulations

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Temperature (TT)</th>
<th>Corners (TT, FF, SS, FS, SF)</th>
<th>(TT, 27°)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>0°</td>
<td>27°</td>
<td>84°</td>
</tr>
<tr>
<td>Static Error</td>
<td>-69.07uV</td>
<td>35.4uV</td>
<td>275.0uV</td>
</tr>
<tr>
<td>Amp Current</td>
<td>207.4uA</td>
<td>211.1uA</td>
<td>218.0uA</td>
</tr>
<tr>
<td>PM</td>
<td>74.66°</td>
<td>75.35°</td>
<td>76.78°</td>
</tr>
<tr>
<td>PSRR</td>
<td>51.81dB</td>
<td>51.88dB</td>
<td>51.99dB</td>
</tr>
<tr>
<td>PSRR BW</td>
<td>2.112MHz</td>
<td>1.992MHz</td>
<td>1.803MHz</td>
</tr>
<tr>
<td>Line Reg</td>
<td>99.96m</td>
<td>99.95m</td>
<td>99.93m</td>
</tr>
<tr>
<td>Load Reg</td>
<td>384.7u</td>
<td>412.0u</td>
<td>471.9u</td>
</tr>
</tbody>
</table>

*Simulated with output device set for 5.75mA
Overview

- RV32IMAFC Core
- 32KB I$ Memory
- 32KB Data Memory
- BootROM
- TSI
- JTAG
- SPI flash
- Interrupts
- GPIO