Tutorial on Agile Research Test Chips
Introduction

Paul Whatmough    Marco Donato    Glenn G. Ko
Sae-Kyu Lee    David Brooks    Gu-Yeon Wei

School of Engineering and Applied Sciences
Harvard University
CHIPKIT: An agile, reusable open-source framework for rapid test chip development

Paul N. Whatmough\textsuperscript{1,2}, Marco Donato\textsuperscript{3}, Glenn K. Ko\textsuperscript{4}, Sae Kyu Lee\textsuperscript{5}, David Brooks\textsuperscript{1}, and Gu-Yeon Wei\textsuperscript{1}

\textsuperscript{1}Harvard University
\textsuperscript{2}Arm Research
\textsuperscript{3}IBM Research.

Abstract—The current trend for domain-specific architectures (DAs) has led to renewed interest in research test chips to demonstrate new specialized hardware. Tape-outs also offer huge pedagogical value garnered from real hands-on experience to the whole system stack. However, success with tape-outs requires hard-earned experience, and the design process is time-consuming and fraught with challenges. Therefore, custom chips have remained the preserve of a small number of research groups, typically focused on circuit design research. This paper describes the CHIPKIT framework, a reusable SoC subsystem which provides basic ID: an on-chip programmable hard, off-chip hosting, memory and peripherals. This subsystem can be readily extended with new IP blocks to generate custom test chips. Central to CHIPKIT is an agile RTT development flow, including a code generation tool called VGEN.

Finally, we discuss test practices for full-chip validation across the entire design cycle.

Index Terms—Agile design, design reuse, testing, open-source

1 INTRODUCTION

Research test chips are the ultimate demonstration of the true value of novel computer architecture and circuits innovation. In addition, tape-out test chips in a research or academic setting provides huge pedagogical value, offering real insight across the whole stack. Nonetheless, tape-out test chips remains very challenging, especially for the uninitiated. Custom chips are time consuming to design, fabricate and test, and are error prone — often requiring extensive rework to fix problems. In this paper, we...
Who are we?

Paul Whatmough
Arm Research
Harvard University

Marco Donato
Harvard University

Glenn G. Ko
Harvard University

Sae-Kyu Lee
IBM Research

David Brooks
Harvard University

Gu-Yeon Wei
Harvard University
Why build test chips in research?

Circuits research
- Measured test chip required for tier-1 publication

Architecture research
- Understand the whole stack, soup to nuts
- Know you are solving a real problem
- Add real impact to your work; not just another academic paper

All models are wrong, some are useful
- Many things are hard to simulate convincingly
- Data to build models to design better circuits

Training for industry
- Build a deep understanding of real computers
- Problem solving, teamwork, time management
- Extremely valuable depth of experience
Many groups routinely tape out test chips in research environments
  • The first few are typically very challenging, climbing a steep learning curve by trial and error
  • Once you know how to do it, it’s easy – heavily reuse the last chip!

Currently a huge amount of activity around open hardware
  • Open source IP from universities
  • Commercial IP available to academia

Never been a better time to get into research test chips!
  • But there is still a knowledge gap in terms of turn this into working test chips
Who is this tutorial for?

• Those looking to:
  • Do their first tape outs
  • Start develop more sophisticated test chips
  • Reduce the time to develop and maintain their SoCs and custom IPs
Goals of the tutorial

• Provide a source of more fundamental material on research tape outs
  • Provide high-level overview of the front-end design and validation of
    • A simple M-Class (Microcontroller) SoC
    • A simple A-Class (Apps Processor) SoC
    • A simple custom IP block
  • An overview of the physical design flow
  • An overview of bring up and testing

• Bring together researchers with a shared interest in research test chips
  • Andrew Kahng (UCSD) – Open source implementation tools
  • Christopher Batten (Cornell) – PyMTL3
  • Adrian Sampson (Cornell) – Predictable accelerator design
  • Thierry Tambe (Harvard) – High-level synthesis
  • Shuojin Hang (Arm) – Arm academic enablement
Tutorial on Agile Research Test Chips
M-Class SoC Development

Paul Whatmough  Marco Donato  Glenn G. Ko
Sae-Kyu Lee  David Brooks  Gu-Yeon Wei

School of Engineering and Applied Sciences
Harvard University
CHIPKIT: An agile, reusable open-source framework for rapid test chip development

Paul N. Whatmough1,2, Marco Donato1, Glen G. Ro3, David Brooks1, and Gu-Yeon Wei1
1Harvard University
2Arm Research

Abstract—The current trend for domain-specific architectures (DSAs) has led to renewed interest in research test chips to demonstrate new architectural features. Tape-out also offers huge potential for value garnered from real hardware in response to the whole system stack. However, successful tape-out demands hard-won experience, and the design process is time consuming and fraught with challenges. Therefore, custom chips have remained the province of a small number of research groups capable of investing in both the tools and facilities to create a chip. This paper outlines CHIPKIT, an open-source, programmable test chip, focusing on reusability. CHIPKIT is a re-usable test chip designed to demonstrate the benefits of a domain-specific architecture. It is composed of four main modules: (i) an asymmetric programmable test chip, (ii) memory and peripherals. This paper expands on the CHIPKIT framework presented in prior work. An open-source approach is used to generate custom test chips, we also present an agile 45% development flow, including a code generation tool called UGEN. Finally, we outline best practices for full-chip validation across the entire design cycle.

Index Terms—Agile design, design reuse, testing, open-source

1 INTRODUCTION

Research test chips are the ultimate demonstration of the true value of novel computer architecture technologies. In addition, tape-out test chips in a research or academic setting provides huge pedagogical value, offering real insight across the whole stack. Nonetheless, despite all the upsides, tape-out test chips remain very challenging, especially for the uninitiated. Custom chips are time consuming to design, fabricate and test, and often error prone – potentially requiring expensive re-spin to fix problems. In this paper, we explore two key themes of agile and microchip design, to help reduce the barrier to entry for chip designers. Building upon previous work, we define CHIPKIT as an open-source framework for rapid test chip design development. CHIPKIT is a re-usable test chip designed to demonstrate the benefits of a domain-specific architecture. It is composed of four main modules: (i) an asymmetric programmable test chip, (ii) memory and peripherals. This paper expands on the CHIPKIT framework presented in prior work. An open-source approach is used to generate custom test chips, we also present an agile 45% development flow, including a code generation tool called UGEN. Finally, we outline best practices for full-chip validation across the entire design cycle.
Q&A session

CHIPKIT Tutorial
Sun. May 31, 2020
Different complexities of test chips

Analog and Test Structures

M-Class SoCs

A-Class SoCs
Outline

• Opportunities for agile and reusable design
• SoC Bus Fabrics
• M-Class Off-Chip Interfaces
• On-Chip Memories: SRAMs and Control and Status Regs (CSRs)
• Clocks, Resets and Power Domains
• Summary
• Recommended Reading
Opportunities for agile and reusable design
SM2 – 28nm DNN ENGINE Accelerator

- Programmable DNN Classifier for IoT
  - Parallelism/reuse 8-way SIMD, 10X data reuse @ 128b/cycle BW
  - Small data-types 8-bit weights, −30% energy
  - Sparse activation data +4X throughput and -4X energy
  - Algorithmic resilience +50% throughput or -30% energy

[Whatmough et al., ISSCC ’17, JSSC ’18, Hot Chips’17]
M-Class Test Chip Architecture

Everything else you need...

The Interesting Bit!!
M-Class Test Chip Architecture

28nm SoC Test Chip

28m SoC Test Chip

Cortex-M0 Subsystem

ARM Cortex-M0

UART

SCAN

SPI

I2S

AFIO

3.3V

VDD

32b APB

32b AHB

REUSE!

32b AXI

Accelerator Subsystem

V_{DCO} – DCO Output

V_{SCLK} – SCLK Input

V_{SCLK} – SCLK Output

V_{VDD} – Accelerator Logic

V_{VDD} – SRAM Periphery

V_{VDD} – SRAM Core

Everything else you need...

CHIPKIT

Arm DesignStart

AGILE!

CHIPKIT

CHIPKIT

CHIPKIT

Your genius
SoC Bus Fabrics
On-Chip Bus Protocols

• Stick to industry standard bus protocols
  • Exploit well documented public standards
  • Reuse protocol checkers and other verification IP
  • Compatible with a broad IP ecosystem
  • Avoid temptation to “modify”

• AMBA open standards
  • APB – low-performance, low area
  • AHB – general purpose
  • AXI – high-performance

• Many others...
  • WISHBONE (OpenCores)
  • Open Core Protocol (OCP)
  • CoreConnect (IBM)
  • TileLink (RISC-V)
Advanced Peripheral Bus (APB)

• Simple bus for connecting master to multiple slaves
• Low hardware cost due to simple control and small number of signals
• Ideal for low-bandwidth peripherals and control signaling
• Data width up to 32 bits
• Non-pipelined, slave can insert wait cycles
  • Every transfer takes a minimum of two cycles
• Synthesis friendly
  • Simple single-edge clocked timing, no tri-state buses
  • One data bus for read and another for write
• APB is a reasonable option for very low-bandwidth peripherals and IO
APB – Basic Transactions

- **Setup phase (T1)**
  - Transfer starts with address PADDR, write data PWDATA, write signal PWRITE, and select signal PSEL, being registered at the rising edge of PCLK

- **Access phase (T2)**,
  - Enable signal PENABLE, and ready signal PREADY, are registered at the rising edge of PCLK
  - When asserted, PENABLE indicates the start of the Access phase of the transfer.
  - When asserted, PREADY indicates that the slave can complete the transfer at the next rising edge of PCLK.
  - The address PADDR, write data PWDATA, and control signals all remain valid until the transfer completes at the end of the Access phase (T3).
Advanced High-performance Bus (AHB)

• Similar to APB, this is a shared bus protocol for multiple masters and slaves, but higher bandwidth is possible through burst data transfers.
• AHB-lite protocol is a simplified version of AHB, designed for a single-master system
• Large bus-widths up to 1024 bits
• We recommend to use AHB-lite widely where performance is not critical
  • Simple to implement and debug
  • Good eco-system support
  • Very low hardware cost
• A typical AHB transaction consists of an address phase and a subsequent data phase, which are pipelined
• A simple address decoder generates slave select signals and controls the muxing of read data back to the master in the data phase
Every transfer consists of:
- Address phase one address and control cycle
- Data phase one or more cycles for the data

The address phase of any transfer occurs during the data phase of the previous transfer (i.e. they are pipelined)

A slave cannot request that the address phase is extended and therefore all slaves must be capable of sampling the address in one cycle.

However, a slave can request that the master extends the data phase using HREADY
- This signal, when LOW, extends the transfer to give the slave extra time.
• The Advanced Extensible interface (AXI) is very common for higher performance interconnects
• More flexible
  • Scalable point to point interconnect, easy to pipeline and close timing on big complex SoCs
• More features
  • Supports multiple outstanding data transfers, burst data transfers, coherency (with extensions) etc
• Also... more variants
  • AXI-stream protocol is effectively the basis of the AXI protocol, for streaming data from a master to a slave using a simple valid/ready handshake
  • AXI-lite protocol is a simplified version of AXI, removing burst data transfers
  • AXI Coherency Extensions (ACE and ACE-Lite), extend AXI with additional signaling that enables system wide coherency or IO coherency
• We’ll come back to AXI in the A-class SoC section
• AXI-lite can be useful for simpler test-chips, but AHB is even simpler
On-Chip Interconnect IP

• Choose bus interfaces based on required features and performance
  • Strong eco-system around AMBA open standards, but there are other credible choices too 😊

• RTL interconnect implementations need to be robust
  • Critical to operation of SoC, must work 100% - verification is essential
  • Should be flexible and easy/robust to make changes as the tape out project evolves

• Typically need multiple busses
  • Often partition buses (even on simple chips) based on usage and traffic types and volumes
  • Helps with throughput, as well as design and verification
  • Use bridges to inter-connect buses of the same or different or protocols

• Use standard / verified bus interfaces and RTL implementations where possible
  • Time to quality greatly reduced
  • Minimize the chance of critical bugs that could lead to a dead block or even a broken SoC
  • Spend time on the differentiating part
  • Reuse protocol checkers and compatibility with the whole SoC eco-system
CHIPKIT AHB Interconnect IP

- Simple single-layer AHB-lite Interconnect IP
  - Uses bundled SystemVerilog Interfaces to vastly reduce typing
  - Configures address decoder based on a single memory map header file
  - Easy to update as the tape out project evolves, by editing only a single file
  - Automatic default slave in the decoder to catch accesses to unused regions
SystemVerilog interfaces bundle signals, parameters, assertions, ...
CHIPKIT AHB Interconnect IP

- Wrap legacy IP to bundle bus interface
- This example shows an Arm Cortex-M0
CHIPKIT AHB Interconnect IP

• Decoder memory map controlled by a single SystemVerilog package

```systemverilog
define AHB_SLAVE_TIMEOUT = 999;

module cm0_memmap_pkg;
    // _START is first valid address (inclusive),
    // _END address is inclusive

    localparam logic [31:0] S0_ADDR_START = $32'h0000_0000; // SRAM program mem
    localparam logic [31:0] S0_ADDR_END = $32'h0000_00FF; // 64KB
    localparam logic [31:0] S1_ADDR_START = $32'h4000_0000; // SRAM data mem
    localparam logic [31:0] S1_ADDR_END = $32'h4000_00FF; // 64KB
    localparam logic [31:0] S2_ADDR_START = $32'h1003_0000; // APB subsystem peripherals
    localparam logic [31:0] S2_ADDR_END = $32'h1003_FFFF; // 64KB
    localparam logic [31:0] S3_ADDR_START = $32'h1001_0000; // GPIO
    localparam logic [31:0] S3_ADDR_END = $32'h1001_FFFF; // 64KB
    localparam logic [31:0] S4_ADDR_START = $32'h1002_0000; // SPI/I2C
    localparam logic [31:0] S4_ADDR_END = $32'h1002_FFFF; // 64KB
    localparam logic [31:0] S5_ADDR_START = $32'h1005_0000; // SYSCtrl
    localparam logic [31:0] S5_ADDR_END = $32'h1005_FFFF; // 64KB
    localparam logic [31:0] S6_ADDR_START = $32'h1000_0000; // ACC0
    localparam logic [31:0] S6_ADDR_END = $32'h10FF_FFFF; // 2MB - 2KB internal address width
    localparam logic [31:0] S7_ADDR_START = $32'h1004_0000; // CFG
    localparam logic [31:0] S7_ADDR_END = $32'h1004_FFFF; // 64KB
    localparam logic [31:0] S8_ADDR_START = $32'h0000_0000; // NICTOP - lower region
    localparam logic [31:0] S8_ADDR_END = $32'h0FFF_FFFF; // NICTOP - upper region
    localparam logic [31:0] S9_ADDR_START = $32'h2000_0000; // NICTOP - upper region
    localparam logic [31:0] S9_ADDR_END = $32'hFFFF_FFFF;

    // AHB timeout for assertions
    localparam integer SM2_AHB_SLAVE_TIMEOUT = 999;

endmodule
```
M-Class Off-Chip Interfaces
Basic IO

- System clock
  - From PCB crystal
- Power-on Reset (PoR)
  - On a test chip, typically driven from a PoR circuit on the PCB
- Test pins, such as scan test IO
- Real-time clock
- CPU debug interfaces
- General-purpose IO
  - Software programmable, very useful, e.g. for working around problems or testing purposes
UART Slave Peripherals

• Important for software, typically retarget printf() to UARTs
  • Plus use in RTL simulation to control simulation termination

• Speed typically not that important for simple terminal communication
  • But, beware UARTs limiting RTL simulation runtime
  • Include a very high baud rate option just for RTL sims to avoid this problem

• Use a UART to USB transceiver on the PCB
  • This allows a simple USB cable to a laptop, which can then communicate with the test chip interactively using a terminal emulator or in a python script

• Can be useful to have multiple UART slaves, especially in multi-core systems which are more difficult to bring-up and debug
CHIPKIT UART Master Peripheral IP

- For test chips, integrating higher-performance IO is challenging and risky
  - For example, HW and SW integration for on-chip USB is non-trivial
  - In practice, many research chips use simple IO for programming and moving data
- Simple IP for off-chip hosting over UART
  - Very simple and robust bus master interface onto the SoC
  - Perform on-chip bus transactions from laptop terminal over USB
  - No CPU overhead, very simple to integrate HW and SW
- Provides a simple interactive text interface in the terminal, no SW required
  - Write transaction -> w(rite) 0x0700000F 0xDEADBEEF
  - Read transaction -> r(ead) 0x0700000F ( -> 0xDEADBEEF)
- Python tool for loading data and scripting up tests
  - Chip LOad Tool (CLOT)
CHIPKIT VGEN IO Script

- VGEN script for generating and managing IO throughout the project
  - Uses Python VGEN infrastructure for generators (we’ll come back to this)

- CSV database of IO signals
  - Entered manually or automatically generated from the top level SoC netlist
  - Updated automatically as things are added/removed

- Generates/updates lots of useful “code” in seconds
  - Pad ring RTL, pad ring instance, testbench interface, EDA scripts etc
  - Very easily extensible
```plaintext`
default_nettype none

module
  // VGEN: MODULE NAME
  TOP_PADS

  // VGEN: MODULE DECLRARATION
  input wire PAD_HCLK,
  input wire PAD_EXTCLK0,
  input wire PAD_EXTCLK1,
  output wire PAD_TLXFWD_CLK,
  input wire PAD_TLXREV_CLK,
  input wire PAD_PORESETn,

  // Module instanitiation.
  TOP_PADS uTOP_PADS ( 
    .PAD_HCLK, 
    .PAD_EXTCLK0, 
    .PAD_EXTCLK1, 
    .PAD_TLXFWD_CLK, 
    .PAD_TLXREV_CLK, 
    .PAD_PORESETn, 
```
Diagnostic (DIAG) Signal Mux

• Sometimes with test chips, things don’t go as planned 😊
  • Debugging problems inside a fabricated chip can be very challenging

• In the worst case, it can really help to have visibility into the system
  • UART bus master allows access to the on-chip bus to poke registers and RAMs
  • But what if there’s a more subtle problem?

• It helps a lot to have visibility of key signals on-chip, but can’t pin out everything
  • Clocks, resets, power rails, power gate enables, interrupts etc

• Diagnostic (DIAG) signal mux allows more signals to be pinned out by using a big mux to select a signal to drive a pin from a larger group of options
  • Having two DIAG pins allows interaction of multiple things, e.g. check clock and reset at the same time

• Easily generated automatically using a VGEN script
On-Chip Memories: SRAMs and Control and Status Regs (CSRs)
Scratchpad SRAMs

• SRAMs need to be wrapped with a bus interface
  • This can range from very simple to very complex
  • For M-Class systems, typically a simple AHB interface

• Easy to overlook verification things like memory-mapped SRAMs
  • Common mistakes include accidental truncation of address bits somewhere
  • Be sure to test across the full address range, not just the first few words
CHIPKIT Scratchpad SRAMs

- AHB_MEM.sv
Control and Status Registers (CSRs)

• CSRs are very common in memory maps at both SoC and IP -levels
  • Additionally, in test chips it’s common to use a large number of CSRs
  • Enabling features that you want to be able to turn on/off (chicken bits)
  • Debug
  • Test purposes

• Generating and maintaining CSRs involves a large number of files:
  • Multiple RTL files
  • Documentation
  • Programming model and views
  • Test cases

• For agile research test chips, very helpful to avoid doing this by hand!
• CHIPKIT provides a simple flow to automatically generate CSRs from a single database using a VGEN Python script
• Register definition described in comma separated value (CSV) format
  • Can be manually or automatically generated and updated
  • Signal naming pre/post fix convention to identify registers in RTL
  • Very lightweight to update and maintain
• Automatically generates everything required
  • RTL, documentation, tests and software definitions in both C and Python
  • Register definition described in comma separated value (CSV) format
• Other open source alternatives for auto generating CSRs
  • Vregs - https://www.veripool.org/wiki/vregs
Control and Status Registers (CSRs)

- CSRs in CSV database – populate by hand or self-populate (and update as things change) from RTL module(s)

```
$ ./vgen_regs.py --update ../CRG.sv --csv cregs.csv --prefix dc_
```
CHIPKIT VGEN CSR Script

• VGEN script reads the CSV database and automatically generates an RTL module for the registers and an instantiation template...

```python
module

// VGEN: MODULE NAME
cregs

(

// clocks and resets
input logic clk,
input logic rstn,

// Synchronous register interface
reg_intf.sink regbus,

// VGEN: OUTPUTS FROM REGS
output logic [31:0] dc_dummy0 /* idx #0: Dummy 32b register */,
output logic [31:0] dc_dummy1 /* idx #1: Dummy 32b register */,
output logic [31:0] dc_dummy2 /* idx #2: Dummy 32b register */,
output logic [31:0] dc_dummy3 /* idx #3: Dummy 32b register */,
output logic [0:0] dc_pic dco en /* idx #4: */
```

```python
logic [0:0] dc_cap_jtagtop;
logic [0:0] dc_cap_dbgwsenable;
logic [0:0] dc_cap_devicewen;
logic [0:0] dc_erlk_pwr_on_en;

cregs u_cregs

// clocks and resets
.clk(HCLK),
.rstn(HRESETn),

// Synchronous register interface
.regbus (cregs.sink),

// reg file signals
.dc_dummy0(dc_dummy0[31:0]) /* idx 0 */,
.dc_dummy1(dc_dummy1[31:0]) /* idx 1 */,
.dc_dummy2(dc_dummy2[31:0]) /* idx 2 */,
.dc_dummy3(dc_dummy3[31:0]) /* idx 3 */,
.dc_pic dco en(dc_pic dco en[0:0]) /* idx 4 */,
```

```
./vgen_regs.py --generate cregs.csv --clock HCLK --reset HRESETn --output output
```

35 of 11
CHIPKIT VGEN CSR Script

- Automatically generates software views, e.g. Python class or C header

```python
class Cregs(object):
    def __init__(self, base_offset):
        self.base_offset = base_offset

    self.DC_DUMMY0 = self.base_offset + 0x0  # Dummy 32b register
    self.DC_DUMMY1 = self.base_offset + 0x4  # Dummy 32b register
    self.DC_DUMMY2 = self.base_offset + 0x8  # Dummy 32b register
    self.DC_DUMMY3 = self.base_offset + 0xc  # Dummy 32b register
    self.DC_NIC_DCO_EN = self.base_offset + 0x10  #

#ifndef CREGS_H
#define CREGS_H

typedef struct {
    __IOM uint32_t DC_DUMMY0;  /* Offset: 0x0 (R/W) Dummy 32b register */
    __IOM uint32_t DC_DUMMY1;  /* Offset: 0x4 (R/W) Dummy 32b register */
    __IOM uint32_t DC_DUMMY2;  /* Offset: 0x8 (R/W) Dummy 32b register */
    __IOM uint32_t DC_DUMMY3;  /* Offset: 0xc (R/W) Dummy 32b register */
    IOM uint32_t DC_NIC_DCO_EN;  /* Offset: 0x10 (R/W) */
} Cregs;
#endif
```
CHIPKIT VGEN CSR Script

• Automatically generates tests in C or Python

```c
#include "cregs_test.h"

// This test is intended to check initial (reset) values of registers
int cregs_initial_value_test(void) {
    int num_errors=0;
    if (SM2_CREGS->DC_DUMMY0 != 0) {num_errors += 1; puts("ERROR: DC_DUMMY0");}
    if (SM2_CREGS->DC_DUMMY1 != 0) {num_errors += 1; puts("ERROR: DC_DUMMY1");}
    if (SM2_CREGS->DC_DUMMY2 != 0) {num_errors += 1; puts("ERROR: DC_DUMMY2");}
    if (SM2_CREGS->DC_DUMMY3 != 0) {num_errors += 1; puts("ERROR: DC_DUMMY3");}
    if (SM2_CREGS->DC_NIC_DCO_EN != 0) {num_errors += 1; puts("ERROR: DC_NIC_DCO_EN");}
```
CHIPKIT VGEN CSR Script

- Automatically generates documentation in Markdown

---

CHIPKIT

---

Programmers Model

Module: CREGS

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Signal Name</th>
<th>Access</th>
<th>Bit width</th>
<th>Start bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8</td>
<td>DC_DUMMY0</td>
<td>RW</td>
<td>32</td>
<td>0</td>
<td>Dummy 32b register</td>
</tr>
<tr>
<td>0x4</td>
<td>DC_DUMMY1</td>
<td>RW</td>
<td>32</td>
<td>0</td>
<td>Dummy 32b register</td>
</tr>
<tr>
<td>0x0</td>
<td>DC_DUMMY2</td>
<td>RW</td>
<td>32</td>
<td>0</td>
<td>Dummy 32b register</td>
</tr>
<tr>
<td>0xc</td>
<td>DC_DUMMY3</td>
<td>RW</td>
<td>32</td>
<td>0</td>
<td>Dummy 32b register</td>
</tr>
<tr>
<td>0x10</td>
<td>NIC_DCO_EN</td>
<td>RW</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Clocks
Resets
Power Domains
Clocks

• Demands careful design, documentation and RTL hygiene
  • Keep as simple as possible and document very clearly!
  • Ideally, every block only has a single clock and reset and does not span clock domains by design
• Low-frequency clocks can be supplied from the PCB through the package
  • Limited by the PCB/packaging parasitics and IO cell slew limitations – typically works okay up to about 100MHz or so
• Typically also want faster clocks, especially in deep-nm technologies
  • Do not generate clocks inside the IP, deliver from the SoC for better control / visibility / debug
• Internally-generated clocks
  • Fixed-ratio of the bus or system clock
    • Possible to use avoid async bridge for data moving between clock domains
    • But, will need a more careful implementation of clock tree and timing closure
  • Asynchronous requires a full asynchronous bridge CDC
    • Caution: Dragons be here!! If you must do this, use pre-verified code and be careful
CHPIKIT On-chip clock generation

• Standard IO cells do not work well above ~100MHz or so
  - On-chip clock generation for faster clocks

• PLLs and related synthesizers
  - Very low noise and drift
  - High quality reference clock
  - Dedicated clean decoupled power supply
  - Careful layout integration
  - Availability and cost

• Digitally-tuned open-loop oscillator
  - Cheap and straightforward, no hand layout or analog macros
  - Sufficient noise performance for high-performance digital
  - Dedicated clean decoupled power rail

• All-digital implementation!
  - Verilog netlist of standard cell instantiations
  - Verify and tune using SPICE simulation
  - Place and route automatically, or using constraints/directives

“A 40–550 MHz Harmonic-Free All-Digital Delay-Locked Loop Using a Variable SAR Algorithm”, R-J Yang et al., JSSC’07
Resets

- Conceptually straightforward, but can easily cause catastrophic issues
  - Keep as simple as possible and document very clearly!
- Stick to asynchronous active-low resets
  - This uses the async reset pin on standard library flip-flops
  - Don’t get hung up on this, it’s just a convention 😊
- Only one reset per IP block clock domain
  - in which case, use one reset per clock
  - Ensure that reset generation is not done inside IP – do at SoC level instead
- Do not use reset synchronizers when crossing clock domains
  - Use the clock-off strategy instead
Power Domains

• Many reasons to have multiple power domains in your research test chips
  • Functional – e.g. IO cells and analog macros typically need various voltages
  • Performance – e.g. per-cluster or per-core voltage scaling, or clean PLL rails
  • Measurement – split out power consumption of different components (e.g. RAM vs logic power, or cache vs core power)

• But, power domains add a huge amount of complexity in both RTL and (especially) implementation, and a lot of risk!

• For research test chips, we suggest using a lightweight approach
  • Multiple rails where necessary to achieve your research goals
  • Try and avoid power gates and level-shifters, which add a huge amount of EDA complexity and risk
  • Be careful with package design and pin allocation to ensure that power integrity is sufficient
SoC RTL Validation
Setup a Makefile for front-end validation to make it easy to run single validation tasks or longer regressions

- Lint – commercial tools, simple checkers
- Multiple synthesis tool front-ends
- Simulation regressions, RTL, synth/layout netlist +notiming, netlist SDF corners
- Logical equivalence checking (LEC)
- Formal verification
- Power-optimization tools – clock gates and RAM enables

Compiled simulators - VCS/Verilator for fast regressions, possibly modelsim etc for interactive debugging

- Don’t underestimate compute and disk space requirements, especially for regressions
  - Escalates quickly for netlist sims and especially with timing annotation (will need runtime optimization...)

Keep back-end stuff in a separate Makefile flow

- Front-end design will mainly use Synthesis, power analysis, timing analysis, etc
SoC Test Coverage

• Basic tests
  • Clock / reset / power stress tests
• Off-chip communication stress tests
• Per-block exercise test
  • Transactions to cover all regs and mems
    • Can be slow in some cases - optimize for good coverage
    • Ideally write/readback assuming R/W location in memmap
    • Basic functionality tests, ideally fairly fast

• Whole memory map test
  • Transactions to cover everything on the memory map
  • Test unmapped regions too!

• Targeted application tests
  • Ideally every test you want to measure on the silicon when it comes back
M-Class SoC: Summary

• Keep things as simple as possible
  • Essential to balance feature creep against risk and time

• Timing closure will be easier with good hierarchy choices
  • And general hygiene - e.g. hard timing boundaries inside block IO

• Invest in scripting for repetitive tasks
  • Much easier to make changes and maintain
  • Better reuse between projects

• Setup whole SoC as early as possible and build up incrementally
  • Use integration shells for any new IP that does not yet exist
  • Same for back-end implementation
Tutorial on Agile Research Test Chips
Part 3: A-Class SoC

Paul Whatmough  Marco Donato  Glenn G. Ko
Sae-Kyu Lee  David Brooks  Gu-Yeon Wei

School of Engineering and Applied Sciences
Harvard University
M-Class SoC architecture example
Outline

• SMIV – A-Class SoC architecture
• Interconnect fabric
  • NIC400 and AXI interfaces
  • TLX400
• Socrates IP tooling flow
  • Import IPs from arm IP catalog
  • Configure and build a NIC400 interconnect
  • Generate Verilog RTL
A-Class SoC architecture

- Arm Cortex-A53 64-bit CPU Cluster
  - Cortex-A53
  - Cortex-A53
  - 2MB L2

- NIC-400 64-bit Interconnect
  - ACC0
  - ACC1
  - ACC2
  - ACC3
  - Accelerator Coherency Port (ACP)

- NIC-400 128-bit Interconnect
  - ThinLink IO Bridge
  - eFPGA 2x2 EFLX4K
  - 4 Banks x1MB SRAM

- AHB 32-bit Interconnect
  - Cortex M0
  - FC ENGINE +SRAM
  - Periph.

- Always-On (AON) 32-bit Cortex-M0 Cluster

- M-class Sub-system
Interconnect fabric
NIC400 interconnect

• Flexible interconnect that can host up to 128 master and 64 slave AMBA interfaces

• AMIBs and ASIBs provide AMBA master/slave interfaces

• IBs provide domain crossing and buffering functions
AXI interface

- AMBA AXI interface for high-throughput, low-latency
- Flexible
- Backward compatible with AHB and APB

- Interface uses 5 independent channels
  - 2 address channels (read/write)
  - 2 data channels (read/write)
  - 1 response channel (write)

- Allows to operate with separate address/control data phases
  - Supports unaligned data transfers using byte strobes
  - Burst-transactions (fixed, incremented, or wrapped)
AXI handshaking

- AXI uses global clock and reset signals (**ACLK** and **ARESETn**)

- Each of the 5 channels uses **VALID/READY** signals to implement the same handshaking protocol
  - The source generates the **VALID** signal when the information is available
  - The sink generates the **READY** signal when it can accept the information
  - The transaction takes place when both **VALID** and **READY** are high

- Both read and write data channels include a **LAST** signal which is asserted when the last transfer in a burst transaction is being driven
AXI variations

AXI-Lite targets simpler control register-style interfaces that might not need full AXI functionality

• Burst length fixed to 1
• Supports a data bus width of 32-bit or 64-bit

<table>
<thead>
<tr>
<th>Global</th>
<th>Write address channel</th>
<th>Write data channel</th>
<th>Write response channel</th>
<th>Read address channel</th>
<th>Read data channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACLK</td>
<td>AWVALID</td>
<td>WVALID</td>
<td>BVALID</td>
<td>ARVALID</td>
<td>RVALID</td>
</tr>
<tr>
<td>ARESETn</td>
<td>AWREADY</td>
<td>WREADY</td>
<td>BREADY</td>
<td>ARREADY</td>
<td>RREADY</td>
</tr>
<tr>
<td>−</td>
<td>AWADDR</td>
<td>WDATA</td>
<td>BRESP</td>
<td>ARADDR</td>
<td>RDATA</td>
</tr>
<tr>
<td>−</td>
<td>AWPROT</td>
<td>WSTRB</td>
<td>−</td>
<td>ARPROT</td>
<td>RRESP</td>
</tr>
</tbody>
</table>

AXI4-Stream protocol is used as a standard interface to connect components that wish to exchange data

• Used to implement the point-to-point connection for data transfer
TLX Bridge (ThinLink)

- Optional feature for the NIC400 interconnect
  - Reduce number of signals compared to AXI
  - Interconnect routed over a longer distance
  - Supports only a single M/S interface to implement a forward and reverse link
  - Supports clock domain crossing

- The physical layer is AXI-stream compliant

- Used for off-chip Wide I/O bridge to FPGA
  - Access to additional IP (DRAM, PCIe, USB, etc.)
Arm Socrates flow
Arm Socrates IP tooling flow

- Arm IP configuration
  - Cortex A53, system IPs
- External macro configuration
  - Accelerators and custom blocks
- NIC400
  - Clock domains definition
  - Define interfaces
  - Memory map
- Build
  - RTL Verilog
Arm IP configuration - Cortex-A53

IP catalog offers a set of Arm IPs:

- Graphics and multimedia cores (Mali)
- Cortex A-, M-, and R- cores
- System controllers
# Clock domains

```haskell
createClockDomain(deCC, :name => "clk1", :frequency => "100", :powerdomainref => "pd0")
createClockDomain(deCC, :name => "clk2", :frequency => "100", :powerdomainref => "pd0")
createClockDomain(deCC, :name => "clk3", :frequency => "100", :powerdomainref => "pd0")
```

# Clock relationships

```haskell
createClockRelation(deCC, :clockref1 => "clk1", :clockref2 => "clk0", :relationship => "asynchronous")
createClockRelation(deCC, :clockref1 => "clk0", :clockref2 => "clk2", :relationship => "identical")
createClockRelation(deCC, :clockref1 => "clk0", :clockref2 => "clk3", :relationship => "synchronous", :synchronoustype => "1:n")
createClockRelation(deCC, :clockref1 => "clk2", :clockref2 => "clk3", :relationship => "synchronous", :synchronoustype => "m:1")
createClockRelation(deCC, :clockref1 => "clk0", :clockref2 => "clk3", :relationship => "synchronous", :synchronoustype => "m:n")
```
NIC400 configuration

Define interfaces

```python
intf = createMasterIF(deCC, :name => "AXI_Master_ACC0", :clockref => "clk3")
setProtocolAttributes(intf, :protocol => "AXI4MasterProtocol", :datawidth => 128, :addresswidth => 32,
                        :multiregion => false, :trustzonemaster => "secure", :idwidthreduction => false,
                        :programmable => true)
masters = ["AXI4_Master4_ACC0"]

intf = createSlaveIF(deCC, :name => "AXI_Slave_ACC0", :clockref => "clk3")
setProtocolAttributes(intf, :protocol => "AXI4SlaveProtocol", :datawidth => 128, :addresswidth => 32,
                        :programmable => true,
                        :vidwidth => 6, :trustzoneslave => "secure", :readacceptance => 1, :writeacceptance => 1,
                        :localgroupref => "axi_inputs")
createPath(deCC, :source => "AXI_Slave_ACC0", :targets => masters)
slaves = ["AXI_Slave_ACC0"]
```
NIC400 memory map

- Each slave can have its own address map
- Address regions must not overlap
- Regions must be aligned to 4KB boundaries
- Undefined memory regions routed to internal default slave

|Mapped Blocks for selected Memory Map|
|---|---|---|---|
|Offset|Range|Default Target|Region|Resolved Target|
|0x30000000|0x00000000|AXI4_Mstra_UMEM0|\-\-|AXI4_Mstra_UMEM0|
|0x32000000|0x00100000|AXI4_Mstra_HLS|\-\-|AXI4_Mstra_HLS|
|0x33000000|0x00500000|AXI4_Mstra_FLEXNLP|\-\-|AXI4_Mstra_FLEXNLP|
|0x31000000|0x00100000|AHB_Mstra_BGMA|\-\-|AHB_Mstra_BGMA|
|0x08000000|0x00020000|AXI_Mstra_MEM|\-\-|AXI_Mstra_MEM|
|0x00000000|0x00010000|AXI_Mstra_DMEM|\-\-|AXI_Mstra_DMEM|
|0x04000000|0x00000000|TLX_Mstra|\-\-|TLX_Mstra|
|0x0c000000|0x00100000|AXI_GIC_Mstra|\-\-|AXI_GIC_Mstra|
|0x2a020000|0x00010000|APB_Mstra|\-\-|APB_Mstra|
|0x2a030000|0x00020000|APB_RTC|\-\-|APB_RTC|
|0x2a040000|0x00010000|APB_UART0|\-\-|APB_UART0|
|0x2a050000|0x00010000|APB_UART1|\-\-|APB_UART1|
|0x2b000000|0x00100000|GPV|\-\-|GPV|
Q&A session

CHIPKIT Tutorial (Part 3)
Sun. May 31, 2020
10:30 AM - 10:45 AM
Tutorial on Agile Research Test Chips
Custom IP Development

Paul Whatmough    Marco Donato    Glenn G. Ko
Sae-Kyu Lee    David Brooks    Gu-Yeon Wei

School of Engineering and Applied Sciences
Harvard University
CHIPKIT Materials

IEEE Micro paper


Open source Github project

- https://github.com/whatmough/CHIPKIT
Outline

• Overview
• Interface and Control
• Hardware Description
• SystemVerilog Coding Guidelines
• Summary
Overview
Custom IP Development

28nm SoC Test Chip

Cortex-M0 Subsystem

ARM Cortex-M0

UART M

SCAN M

GPIO S

SYSCTL S

I-MEM 64KB S

D-MEM 64KB S

Bridge S

32b APB

32b AHB

Accelerator Subsystem

V_{DCO}

DCO

FCLK

ASYNC

DNN Engine

W-MEM 1MB

128b A1X

V_{ACC} – Accelerator Logic
V_{MEMP} – SRAM Periphery
V_{MEMC} – SRAM Core

OSC

USB

Scan

GPIO

RTC

USB

Timers S

Wdog S

UARTs S

BIST S

Everything else you need...

The Interesting Bit!!
Custom IP Development

REUSE!

AGILE!

Interface and Control

Host Processor

Data In

Data Out

NBUF

Index

DMA

Weight Load

MAC Datapath

Activation

Data Read

IPBUF

XBUF

W-MEM

Writeback

Custom Machine

REUSE!

AGILE!

Interface and Control

Host Processor

Data In

Data Out

NBUF

Index

DMA

Weight Load

MAC Datapath

Activation

Data Read

IPBUF

XBUF

W-MEM

Writeback

Custom Machine
Interface and Control
• Simple slave control – “passive” accelerator programming model
  • Configure accelerator for job
  • Move data into internal RAM
  • Start task and wait for a done signal (IRQ or poll register)
• Slave control with master for data (internal DMA)
  • Configure accelerator for job
  • Accelerator accesses data from provided addresses
  • Run task and (optionally) move results to somewhere in memory
• Job queue control with master for data (internal DMA)
  • Accelerator fetches job for queue in memory
  • Accelerator accesses data from provided addresses
  • Run task and (optionally) move results to somewhere in memory
Accelerator Interfaces

• Very active and arguably understudied aspect of modern SoCs
  • Data movement cost
  • Virtualization
  • Coherency

• Arm A-Class cores feature Accelerator Coherency Port (ACP)
  • Allows access directly into L2 cache of the CPU cluster
  • Coherent to the CPU cache, without implementing dedicated coherency logic

• Arm Coherent Accelerator Interface (ACAI)
  • Implements a coherent (ACE-lite) accelerator interface with RTL & SW
  • All the benefit of coherency, without the complexity of implementing it!
Arm Coherent Accelerator Interface (ACAI)

• Hardware and software framework to enable easy adoption of accelerators on SoC platforms

• Easier hardware accelerator integration
  • ACAI provides accelerator with a coherent cache and virtual addressing capabilities
  • Accelerator interfaces to ACAI using standard AXI protocol
  • Compatible with Xilinx Vivado HLS (High-Level Synthesis)

• Simpler programming model
  • Linux user application written in C/C++ runs on CPU - easier to debug & modify
  • ACAI software libraries and drivers assist with job creation, scheduling and dispatch
  • Support for accelerator virtualization (sharing across different processes)
  • Maximize performance gains and enable fine-grained task acceleration
Arm Coherent Accelerator Interface (ACAI)

Full System Coherence

- HW accelerator connected to ACAI HW IP
- User application written in C/C++ runs on CPU
- ACAI SW libraries assist with job creation, scheduling and dispatch
- ACAI kernel driver sets up application context, page tables and configures ACAI IP
- ACAI HW IP configures the accelerator, provides a coherent cache and memory interface
- User accelerator executes on user described job

Reduced accelerator dispatch time
- No explicit data copies
- No CPU flush/invalidate required
- Supports hardware pointer dereferencing
void main() {
   // initialize acai
   acai *p_acai = new acai();
   p_acai->init();
   
   // setup job chain with a single job
   vector<acaijd> job_chain;
   job_chain.reserve(1);
   
   // setup job descriptor to write 3 registers
   job_chain.push_back(acaijd(3, 0));
   job_chain[0][0] = (uint32_t)length;
   job_chain[0][1] = (uint32_t)src_data;
   job_chain[0][2] = (uint32_t)result_data;
   
   // start and wait on the job to complete
   p_acai->start_job(job_chain[0]);
   p_acai->wait_job(job_chain[0]);
   
   // cpu reads results
}

module ha_fft (  
   input wire clk,
   input wire reset_n,

   // Memory interface (AXI4 Master)
   // ..
   
   // Configuration interface (AXI4-Lite Slave)
   // ..
   
   input wire ai_job_start_i,
   output wire ai_job_complete_o
);
   reg [31:0] length;
   reg [31:0] src_addr;
   reg [31:0] result_addr;

   // more code
endmodule // ha_fft
RTL IP Integration Shell

- Accelerator will need an internal memory map
  - This is usually relative to whatever address it resides at the SoC-level
  - Typically truncate address bits to the required internal address space
  - Ensure RAMs and architectural registers are bus accessible for debug

- An RTL IP integration shell should be the first RTL task
  - Effectively a bus functional model in synthesizable RTL
  - This should be well thought out and documented
  - Very useful for SoC designers and for back-end designers early on
    - All inputs and outputs should be registered for easy timing closure

- The accelerator interfacing should ideally not change, even while the internal blocks are being developed
  - Clear documentation
  - A set of simple (SoC) tests will help ensure this
Control

• Simpler accelerators usually incorporate a control FSM
  • Think carefully about the design of this, bugs may be difficult to work around
• Try and think through all the tests you need to do on silicon
  • Various functional scenarios and use cases
  • Debug scenarios – visibility if when things go wrong
• Measuring performance
  • cycle counters, memory accesses, instruction/operation counts
  • Counter stop/start/clear conditions
• Measuring power
  • Infinite loop mode, self-test mode
RAMs and Control and Status Registers (CSRs)

• Make all RAMs bus accessible
  • Really helps with debug, testing, margining etc

• CSRs are best scripted unless very minimal
  • See previous example on the CSR VGEN script
Hardware Description
Digital Hardware Design Methodologies

• Native Hardware Description Languages (HDLs)
  • Verilog
  • VHDL
  • SystemVerilog

• Add-ons to HDLs
  • Linting tools
  • Genesis2

• New non-native languages
  • Chisel (Scala)
  • MyHDL, PyMTL (Python)
  • RHDL (Ruby)

• High-level synthesis
  • SystemC

Old – not well suited to modern HW
Weakly typed

Strongly typed
Verbose

Big improvement
Poor templating support

Add compile-time checking
Extra step

Fix language templating deficiencies

Nice languages
Extra generation/translation step

More familiar object-oriented language
Serial program with pragmas to infer parallelism
System modeling benefit
Slow tools and tricky to debug

16 of 11
Object-Oriented HLS Design Flow

- C++ to RTL to gates
- MatchLib: library of commonly-used macro-architectural components in HLS-able C++
- All communication between SystemC modules pass through Latency-Insensitive Channels and interconnects
- Reuse of C++/SystemC testbenches for simulation of HLS-generated RTL
- Productivity improvement over manual RTL coding and existing ad-hoc HLS-based flows

[Khailany et al., DAC’18]
SystemVerilog Coding Guidelines
Why SystemVerilog?

• We’ve had very fast results with HLS, potentially great for exploration

• For higher-performance designs, sometimes it’s nice to have more control over pipelining

• We’ve used SystemVerilog extensively
  • Native support in commercial EDA tools
  • Much more compile time checking
  • No translation step

• We’ve used a strict subset of SV for synthesizable code with good results
  • Easy to learn
  • Removes a large number of common bug classes
  • Great PPA results
  • Great correlation through front-end and back-end

• One drawback is less support for some features in open source simulators
Combinational Logic

• Separate combinational logic and sequential logic (flops)
  • Extra compile time checking, easier pipeline adjustments, code readability

• Use logic type exclusively
  • No need for the confusing wire/reg types
  • Except for netlists – use wire and use `default_nettype none` to catch undeclared nets that are usually typos

• Use lower-case signal names with underscores (_)
  • Except for top-level module port signals – all caps by convention

• Use always_comb keyword for all combinational logic
  • Compile-time checking that latches or flops are not inferred (enforces above)

• Be careful with multiplexers

• Be careful with signed numbers in SV
module my_module (  
    // ...signals...  
);  

// Always split out logic and flops!  

logic a, b, c; // Use logic type only  
logic d, e, f; // Lower-case signals  

always_comb a = b & c; // Single-line logic  

always_comb begin // multi-line logic  
    d = e & f;  
    // more code  
end  

endmodule // my_module
Sequential Logic

- Separate combinational logic and sequential logic (flops)
  - Extra compile time checking, easier pipeline adjustments, code readability
- Use the `always_ff` keyword
  - Compile-time checking that no flops are inferred in block (enforces above)
- Stick to positive-edge clocked, active-low asynchronous reset flops
- Use macro definition for inferring flops
  - Enforces separation of logic and flops
  - More compact, but still readable
  - Easily swap out for different flops, e.g. synchronous reset for FPGA
- Use flop/RAM enable terms -> clock/RAM gating
Sequential Logic

```verilog
module my_module (
    // ...signals...
);

// Always split out logic and flops!
logic d_in, q_out;    // Use logic type only

always_ff @(posedge clk, negedge rstn) begin
    if(!rstn)
        q_out <= '0;
    else
        if(en)
            q_out <= d_in;
end

endmodule // my_module
```

```verilog
`include RTL.svh

module my_module (
    // ...signals...
);

// Always split out logic and flops!
logic d_in, q_out;    // Use logic type only

`FF(d_in, q_out, clk, en, rstn, '0);

endmodule // my_module
```
Module Declarations and Instantiations

• Use the SV syntax for module declarations
  • Saves typing

• For module instantiation, use the automatic SV connections

• Use SV packages to group common functions and definitions
module my_module (  
  input logic clk,  
  input logic rstn,  
  // …more signals…  
);  
  
  // Module body  
my_module2 u0 (  
  .clk, // automatic connect  
  .rstn,  
  // …more signals…  
  .signal99(),v // unused output  
  .signal100(other_sig) // override automatic  
);  
endmodule // my_module
VGEN Scripts

• SystemVerilog still has poor support for templating
  • Generate statement and parameterization are both quite hard to use
• The perennial workaround for this is to write generators
• VGEN is a simple Python framework for writing generators quickly
  • CSV database of objects, could be registers, pins, rams, pads, anything
    • Implemented in Python as a list of dictionaries
  • Tools to automatically populate and update this database from RTL, C, Python, or even documentation
    • Easy to use signal or module name pre/post -fix to mark things for automation
  • Tools to generate RTL modules/instances, Python code, C code, Markdown, …
VGEN Scripts

- **RTL Modules**
  - `... my_sig_csr ...`
  - Custom Postfix

- **Human-Editable CSV Database**
  - Update database with new matching signals
  - `... my_sig_csr ...`

- **Generated Code**
  - `vgen -update`
  - `vgen -generate`
  - RTL (*.v)
  - Instance (*.v)
  - Docs (*.md)
  - SW (*.c, *.py)
  - Tests (*.c, *.py)
Module logging

• Opening waveforms should be a last resort during RTL development
  • Simulation is slower
  • Opening and working with a GUI is slow and clumsy

• Printing signals from a block-level test bench is quite clumsy

• Logging directly from an RTL module is a better approach
  • Wrap non-synthesizable debug code in `ifndef SYNTHESIS
  • Use reset signal to mask junk during reset
  • Can generate a lot of clutter in the simulation transcript

• Even better is to log to file with module name
  • You know where to look for module specific debug data
  • Easy to parse in python to check against a model (especially datapath)
module my_module (
  // ...signals...
);

// Module body

// Logging code

`LOGF_INIT  // This macro opens the log
`LOGF(clk, rstn, enable, format_expr)
`LOGF(clk, rstn, enable, format_expr)
  iss.valid,
  ("" | %s | %s |",iss.op,iss.data)
);

endmodule  // my_module
RAMs

- Think about physical design early on and for each node / library
  - Understand the “best” RAM shapes/sizes in terms of PPA and features
- Use a generic RAM module interface to wrap actual instances with
  - Makes changes much easier
  - Wrap multiple smaller instances to make a big logical RAM
  - Swap out RTL model, IP instantiation, FPGA inference template, etc
- Recommend that IP-internal RAMs are memory mapped
  - Makes bring up and test much easier
  - Configuring margin adjustments
  - Bus access timing paths can be made very slow if necessary, to aid timing closure
- Perform validation with RAM initial state of “X”
Special cells

• "Architectural" clock gates are very common
  • Different to inferred clock gates
  • Useful to be able to turn off a whole block or turn off a whole clock domain etc

• Synchronization flip-flops
  • Usually a double or triple back-to-back flop cell with some circuit optimizations
  • Used for any async signals:
    • clock domain crossing signals
    • Input pins
    • Async interrupts
    • Reset synchronizers (but, don’t use these!)

• For all these things, use a simple wrapped model for development, and be sure to swap this for a wrapped version of the library cell at implementation time
  • Same as for RAMs
Front-End Validation

- RTL integration shell + tests
- Block level design
  - Unit tests
  - Linting
  - LINK TO SV UNIT LEVEL TESTING
- Block integration tests
  - Feature complete
  - Clock gate / RAM enable optimization
- SoC integration tests
- Synthesis trials
  - Timing analysis -> timing closure
  - Power analysis -> power closure
- RTL freeze
Summary - Custom IP Development

- Think about IP interfacing carefully
- Use an IP integration shell to define connectivity early on
- Think about the usage model of the IP
- SystemVerilog subset for synthesis
- Front-end validation
Tutorial on Agile Research Test Chips: Physical Design

Paul Whatmough    Marco Donato    Glenn G. Ko
Sae-Kyu Lee    David Brooks    Gu-Yeon Wei

School of Engineering and Applied Sciences
Harvard University
EDA Tools Overview

• Design capture tools
  • netlist entry, schematic, HDL, state diagram entry

• Simulation and verification tools
  • functional (logic) sim and timing sim

• Synthesis and optimization tools
  • creating netlists and optimizing them for timing and power

• Layout tools
  • floorplanning, CTS, routing
Starting 2019.03 CentOS becomes the “build” platform and RHEL will be supported as a “binary compatible” OS. Cadence supports CentOS but disclaims any liability for any errors or bugs in CentOS.
Hierarchical Design

• vs. flat design
  • better for complex chip design
  • runtime is fast (run individual blocks)
  • can fix timing issues on individual blocks
  • incremental functional and timing fixes

• Timing budgeting is required

• Floorplanning is important
  • interconnects between blocks
  • routing congestions
  • I/O budgeting

• Signoff is usually done using a flat view
Design Entry

- Input: RTL or System-C (other high-level language)
- Output: Verified RTL
- Functional simulation and testbench verification
- Tools: Synopsys VCS, Cadence Xcelium
- High-level Synthesis
- Tools: Synopsys Synphony C, Cadence Stratus, Mentor Catapult
  - refer to Part 2: "Closing the algorithm/hardware design and verification loop with speed via high-level synthesis", Thierry Tambe (Harvard)
Synthesis

- Input: RTL, library files
- Output: Gate level netlist
- Synthesizes gate level netlist from RTL
- Reports area, power, timing estimations
- (DFT Insertion)
- Tools: Synopsys Design Compiler, Cadence Genus-RTL
Partitioning

- Partitioning involves dividing the logic into different logical groups and clock groups

- Synopsys IC Compiler II, Cadence Innovus
Top-level Floor Planning

- Inputs: synthesized netlist, view definitions (libraries, corners, constraints, etc.)
  - I/O constraints: aspect ratio, I/O to core clearance, boundaries
- I/O pad, bump allocation
- Placement of macros and SRAMs, halos for them, I/O filler cells
- Synopsys IC Compiler II, Cadence Innovus
Block-level Floor Planning

- Inputs: synthesized netlist, view definitions (libraries, corners, constraints, etc.)
  - I/O constraints: aspect ratio, I/O to core clearance, boundaries
- Place of SRAM cells, halos around SRAM, boundary cells and well traps
- Place Macro ports
- Synopsys IC Compiler II, Cadence Innovus
Block-level Floor Planning Example
Power planning

- Core power ring, macro power ring, metal straps, vias over VDD and VSS power structures

- Tools: Synopsys IC Compiler II (+PrimePower), Cadence Innovus
Placement

- Preplace, in-placement, post-placement optimization before CTS
- Synopsys IC Compiler II, Cadence Innovus
Clock Tree Synthesis

- Buffer sizing, relocation, fix hold, post placement optimization after CTS, clock gating

- Synopsys IC Compiler II, Cadence Innovus
Routing

- Timing and congestion driven, net constraints, global route, track assignments, detail route, search and repair, post route optimization, via and wire length optimization

- Synopsys IC Compiler II, Cadence Innovus
DFM

• Notch and metal filling, filler cell insertion, fiducial insertion

• Synopsys IC Compiler II, Cadence Innovus, Cadence Virtuoso
Timing Analysis and SI Analysis

- Inputs: .SPEF file
- RC extraction
  - Synopsys StarRC, Cadence Quantus
- Signoff static timing analysis using parasitics
  - Synopsys PrimeTime, Cadence Tempus
Inputs: .SDF file

SDF back-annotated functional simulation
  • contains timing information from the layout

Synopsys VCS, Cadence Xcelium,
Formal Verification

- Inputs: .v or .db netlist
- Formal equivalence verification
- Synopsys Formality, Cadence JasperGold
Physical Verification

- Inputs: GDS II
- Design rule checking (DRC) and Layout Versus Schematic (LVS)
- Mentor Calibre + Cadence Virtuoso
  (Or Synopsys IC Validator, Cadence Pegasus)
Complete flow

• May reiterate different parts of the flow as needed
Focus: Logic Design to Physical Design

1. Write RTL
2. Do functional verification
3. Run synthesis with constraints
4. Check if results are satisfactory
   • If not run go back to 3 or 1
5. Run functional and formal verification on the results
• This is an example of when you may want to perform functional verification, STA and Eq. checking to make sure the timing requirement is met and that the resulting gate-level netlist is functionally equivalent to the original RTL.
Example: Block-level
Example: Top-level
Tips

• Watch the disk space
• Sharing machine resources
• Use Skylake or newer machines for layout if possible
• More DRAM the better; 128GB+ or more
• Do DRC and LVS on initial design as soon as you can
  • esp. if first time through the flow
• Do practice tape out to MOSIS well before the deadline
Q&A Session

- CHIPKIT Tutorial (Part 5)
  - Sun. May 31, 2020
  - 11:00 AM - 11:15 AM
Tutorial on Agile Research Test Chips
Part 6: Bring-up and Testing

Paul Whatmough  Marco Donato  Glenn G. Ko
Sae-Kyu Lee  David Brooks  Gu-Yeon Wei

School of Engineering and Applied Sciences
Harvard University
Outline

• Test Board overview
  • Connectivity
  • Functional blocks

• Clot testing software
  • Run programs on the SoC
  • Voltage and Frequency scaling functions
  • Measurement sweep template
Test board overview
A-Class SoC architecture

- Arm Cortex-A53 64-bit CPU Cluster
  - Cortex-A53
  - 2MB L2

- NIC-400 64-bit Interconnect
  - ACC0
  - ACC1
  - ACC2
  - ACC3

- NIC-400 128-bit Interconnect

- eFPGA 2x2 EFLX4K

- 4 Banks x1MB SRAM

- AHB 32-bit Interconnect
  - Cortex M0
  - FC ENGINE +SRAM
  - Periph.

- Always-On (AON) 32-bit Cortex-M0 Cluster

- ThinLink IO Bridge
  - DRAM PCIe

- M-class Sub-system
Test board and FPGA setup

- DARPA CRAFT BGA flip chip socket
- Power supply connections
- USB to UART interface
- General Purpose IO for debugging
- SMA clock connections
- FPGA Mezzanine Card (FMC) connector
  - Kintex KCU105
Power Delivery – mode I

$V_{DD}$ is generated from a 5V switching power supply and an on-board voltage regulator

Useful for nominal operation and initial debugging
Power Delivery – mode I with current read

\(V_{DD}\) is generated from a 5V switching power supply and an on-board voltage regulator

Useful for nominal operation and initial debugging

Supply current can be measured with a DMM
Power Delivery – mode II

The power pin is decoupled from the on-board voltage regulator

\( V_{DD} \) is generated from a programmable PSU

Required for running voltage scaling measurements
Clocking and Resets

Single off-chip clock (HCLK)
- HCLK is connected to the SoC input pin and a SMA connector for debugging
- Faster clocks are generated on chip and can be routed internally to the DIAG block

The board is design with power-on reset capabilities
- The reset can be manually triggered via push-button
- During power-on, PORESETn is asserted when the supply voltage $V_{DD}$ becomes higher than 1.1 V and deasserted after a fixed delay time (200ms)
USB interface

UART

USB - mini B Receptacle Connector 5 Position Through Hole, Right Angle, Horizontal

USB UART IC
Clot testing software
CLOT – Chip LOad Tool

CLOT provides a set of python functions that allow to talk to the UART interface:
• Check the status of the slave interface
• Configure the DIAG mux for debugging internal signals
• Load hex files to a user-specified memory
• Dump memory content to a hex file
• Memory testing capabilities
• Configure internal CRG registers for clock and reset
• Run programs loaded in memory
Uses VISA API to interface with other instrumentation (PSU, Scope, DMM)

M0 load and run example:

```
clot -load imem hex/M0/hello.vhx -run
```
CLOT – Load (1)

- Open hex file
- Read 32bit word
- Compute next address and write

```python
def write_mem_hexfile(ser, base_addr, hex_file, size=None):
    # Keep writing until reach end of file or hit size bytes.
    # Each line in the hex file must either be '', start with '//',
    # or contain no more than 8 hex digits
    
    # Open file
    assert os.path.isfile(hex_file), 'File missing: %s' % hex_file
    f_hex = open(hex_file, 'r')
    # Write to MEM line by line
    i = 0
    for line in f_hex:
        if size != None and not i < (size / 4):
            break
        # Ignore comment lines etc
        line = line.rstrip()
        if line == '' or '//' in line:
            continue
        assert len(line) <= 8
        addr = base_addr + (i * 4)
        wr(ser, addr, int(line, 16))
        i += 1

    if(0):
        pass

    return None
```
CLOT – Run (2)

• Open UART slave port and switch the AHB MASTER MUX from UART to M0

• Read from UART Slave until:
  • End of Transmission (EOT) is detected
  • Timeout

• Switch AHB MASTER MUX back to UART

```python
uart_slave = serial.Serial(**USB_UART_SLAVE0_CONFIG)
print_info('Run')
ahb_master_mux(uart_master,1)

start_time = time.time()
elapsed = 0
done = False
timeout = 10000.0
while not done:
    out = ''
    while uart_slave.inWaiting() > 0:
        out += uart_slave.read(1)
    if out != '':
        print(out)
        elapsed = time.time() - start_time

    if ('\x04' in out):
        done = True
    elif (elapsed > timeout):
        done = True
        print_error('Timed out')

ahb_master_mux(uart_master,0)
uart_slave.close()
```
CLOT – A53 example

M0 is used to run a bootstrap routine for A53 and the SoC
  • Set all DCO clocks to HCLK (50MHz)
  • Release all resets (except A53)
Load test binary on A53 instruction memory

```clot -load imem hex/M0/run_a53_tlx.vhx -load aimem hex/A53/dhrystone.vhx -run
```

DRAM on the FPGA is used for larger binaries which cannot fit on the on-chip SRAM

```clot -load imem hex/M0/run_a53_tlx.vhx -start
clot -load aimem hex/A53/sys/jump_aimem_tlxdram.vhx -load tlx-dram hex/A53-DRAM/dhrystone.vhx```

DVFS functions

- Frequency and voltage sweeps
- Clock Reset Generator configuration with internal registers
- VISA control functions for PSU/Scope programming and data acquisition

```
# Configure DCO A53 Clock
clot -clock A53_CLK true DCO 63 7

# Program and Measure PSU voltage
vicon -set_ps_vol 3 0.8
vicon -meas_vol 3

# Measure Frequency
vicon -meas_freq 2
```
DVFS sweeps

Running C code tests on any of the internal macros using memory mapped registers:

- Write test vectors to internal registers
- Assert macro/accelerator enable signal
- Read back test results and compare to golden values
- Write error flags to test registers

This routine can be repeated for different values of $V_{DD}$ and frequency to find $f_{max}$
C test code

Write test vectors

```c
puts("Write test vectors");
if (write_read_regs((void *) &SM5_PREGS->DP_I_DCST_SIZE, 32, 0x00014000) != 0) err_code |= 1<<0;
if (write_read_regs((void *) &SM5_PREGS->DP_I_DCST_DEST, 8, 0x01) != 0) err_code |= 1<<1;
if (write_read_regs((void *) &SM5_PREGS->DP_I_DCST_WEN, 8, 0x01) != 0) err_code |= 1<<1;
if (write_read_regs((void *) &SM5_PREGS->DP_I_DCST_RBANK, 8, 0x00) != 0) err_code |= 1<<1;
write_regs((void *) &SM5_PREGS->DP_I_DCST, 8, 0x01);
write_regs((void *) &SM5_PREGS->DP_I_DCST, 8, 0x00);

for (i=0; i<input_size; i+=4) {
    write_regs((void *) SM5_PGMA_DCOST0_BASE+i, 32, input_array[i]);
}
```
C test code

Run test and check for done flag

```c
puts("Run test");
int dcst_done;
dcst_done = read_regs((void *) &SM5_PREGS->DP_0_DCST_DONE, 32);
while(dcst_done != 1)
{
    dcst_done = read_regs((void *) &SM5_PREGS->DP_0_DCST_DONE, 32);
}
```
C test code
PASS/FAIL check

```c
puts("Check test results")
int readvalue;
int diffcount=0;
for (i=base_addr; i<mem_size; i+=8) {
    readvalue = read_regs((void *) SM5_PGMA_GLOBAL_BASE+i, 32);
    if (readvalue != goldvalue[i])
    {
        diffcount = diffcount +1;
    }
}
printf("Total different counts: %d\n", diffcount);
if (diffcount != 0) err_code |= 1<<1;
```
Python sweep wrapper

Template for VF sweeps
meas() uses VISA functions to report:
• Average supply voltage/current
• Average power
• Operating frequency

run_binary() runs test code on A53:
• Load binary on imem/aimem
• Reset block under test and set clock configurations
• Run program and return the test signature

The test signature is compared to a gold value to check for PASS/FAIL

```python
rm = visa.ResourceManager('@py')
for vdd in vdds:
    f_max_flag = 0
    time.sleep(wait_time)
    set_ps_volt(rm, 2, vdd)
    time.sleep(1)
    dco_sel = sel_reset
    dco_div = 1
    time.sleep(wait_time)
    avg_volt, avg_cur, avg_pwr, freq = meas(rm, diag_div)

    while(f_max_flag == 0):
        signature = run_binary(rm, vdd, dco_sel, dco_div, diag_div)
        time.sleep(1)
        if signature != gold_signature:
            # Pick dco_sel, dco_div for lower frequency
            else:
                # Pick dco_sel, dco_div for higher frequency
                # All working (dco_sel, dco_div) values have been tested
                set f_max_flag = 1
```
Sweep results

<table>
<thead>
<tr>
<th>Throughput (GOPS)</th>
<th>Energy (GOPS/W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>54.9x 18.9</td>
<td>10^1</td>
</tr>
<tr>
<td>312.4 58.7</td>
<td>10^2</td>
</tr>
<tr>
<td>1.0378 1.0378</td>
<td>10^3</td>
</tr>
</tbody>
</table>

Fixed Function Accelerator
Reprogrammable Hardware
Fully Software Programmable

Energy/Inf. vs Throughput (GOPS/W)

- Dual-A53
- Dual-A53 SIMD
- 2x2 eFPGA
- Quad-CCA

Throughput (GOPS/W) vs Energy (GOPS/W)

- MEP
- NOM
- F_max