Hardware Software Codesign of SOC

晶片系統之軟硬體共同設計

李宗演

國立台北科技大學電子工程系
tylee@ntut.edu.tw
http://www.ntut.edu.tw/~tylee/

Contents

- Introduction
- SoC Models, Architectures, Languages
- SoC HW-SW Partitioning
- Function-Architecture Codesign
- SoC Co-design Example
Contents

- Introduction
  - SoC Models, Architectures, Languages
  - SoC HW-SW Partitioning
  - Function-Architecture Codesign
  - SoC Co-design Example

Codesign
Definition and Key Concepts

- Codesign Definition
  - The meeting of system-level objectives by exploiting the trade-offs between hardware and software in a system through their concurrent design

- Key Concepts
  - Concurrent: hardware and software developed at the same time on parallel paths
  - Integrated: interaction between hardware and software developments to produce designs that meet performance criteria and functional specifications
Classic Design & Codesign

Why Codesign?

Prototype Availability Gap
**What is an SoC?**

- **System**
  A collection of all kinds of components and/or subsystems that are appropriately interconnected to perform the specified functions for end users.

- **An SoC design is a “product creation process” which**
  - Starts at identifying the end-user needs
  - Ends at delivering a product with enough functional satisfaction to overcome the payment from the end-user

---

**SoC Design Trend**

System-on-Board (SoB)  
System-on-Chip (SoC)
SoC Design Trend

The next design challenge: SoC

- System-on-Chip
  - IP cores
  - Lots of software
- How do you design such systems?
- Codesign!!!
### SoC Codesign Flow & Languages

- **Design Specification** (Models, UML)
- **HW/SW Partitioning**
- **Synthesis**
- **Compiler**
- **Cosimulation**
- **Estimators**
- **SoC Architecture** (Expression, …)
- **Verification**

**HW/SW Partitioning**
- HW: VHDL, Verilog, SystemC
- SW: C, C++, Java

**Synthesis**
- VHDL, Verilog, SystemC

**Compiler**
- C, C++, Java

**Cosimulation**
- Estimators

**Rapid design space exploration**
- Quality tool-kit generation
- Design reuse

### Key Issues in SoC Codesign

- **Unified HW/SW Representations**
  - Models
  - Architectures
  - Languages

- **HW/SW Partitioning**
  - Partitioning Algorithms
  - Function Architecture Mapping

- **HW/SW Cosynthesis**
  - Hardware, Software, and Interface Syntheses
  - Cosimulation Techniques
Contents

- Introduction
  - SoC Models, Architectures, Languages
- SoC HW-SW Partitioning
- Function-Architecture Codesign
- SoC Co-design Example
  - Digital Camera

SoC Models, Architectures, Languages

- Behavior v/s Architecture
- Models
  - State, Activity, Structure, Data, Heterogeneous
- Architectures
  - Function-Architecture, Platform-Based
- Languages
  - Hardware: VHDL / Verilog / SystemVerilog
  - Software: C / C++ / Java
  - System: SystemC / SLDL / SDL
  - Verification: PSL (Sugar, OVL)
Models of an Elevator Controller

(English description)

loop
if (req_floor = curr_floor) then
   direction := idle;
elseif (req_floor < curr_floor) then
   direction := down;
elseif (req_floor > curr_floor) then
   direction := up;
end if;
end loop;

(a) English description

(b) Algorithmic model

(c) State-machine model
Architectures Implementing the Elevator Controller

Virtual Machine Hierarchy

- Application Programs
- Utility Programs
- Operating System
- Monitor
- Machine Language
- Microcode
- Logic Devices
**IP-Based Design of the Implementation**

- Which Bus? PI? AMBA? Dedicated Bus for DSP?
- Which DSP Processor? C50? Can DSP be done on Microcontroller?
- Which Microcontroller? ARM? HC11?
- How fast will my User Interface Software run? How Much can I fit on my Microcontroller?

- Can I Buy an MPEG2 Processor? Which One?
- Can I need a dedicated Audio Decoder? Can decode be done on Microcontroller?

**AMBA-Based SoC Architecture**

- High-performance ARM processor
- High-bandwidth on-chip RAM
- High-bandwidth Memory Interface
- AHB
- DMA bus master
- UART
- Timer
- APB
- Keypad
- PIO
- AHB to APB Bridge
Languages

- Hardware Description Languages
  - VHDL / Verilog / SystemVerilog
- Software Programming Languages
  - C / C++ / Java
- Architecture Description Languages
  - EXPRESSION / MIMOLA / LISA
- System Specification Languages
  - SystemC / SLDL / SDL / Esterel
- Verification Languages
  - PSL (Sugar, OVL) / OpenVERA

Current System Design Methodology

C/C++
System Level Model

Manual Conversion

VHDL/Verilog

Analysis

Results

Refine

- Problems
  - Errors in manual conversion from C to HDL
  - Disconnect between system model and HDL model
  - Multiple system tests

NIU 2004/05/12
**SystemC Design Methodology**

- **Advantages**
  - Refinement methodology
  - Written in a single language
  - Higher productivity
  - Reusable testbenches

**SystemC Programming Model**

- A set of *modules* interacting through *signals*.
- *Module* functionality is described by *processes*.
Vertical Reuse of SystemC Testbench

Contents

- Introduction
- SoC Models, Architectures, Languages
  - SoC HW-SW Partitioning
- Function-Architecture Codesign
- SoC Co-design Example
SoC HW/SW Partitioning

- Partition a set of functions for implementation into hardware and into software, along with interfaces among the hardware and software parts.

SoC Functional Specification

HW/SW Partitioning

Hardware
- Cell-based ASIC, Custom-designed ASIC, Embedded FPGA

Communication Modules
- Memory, Wires, On-Chip-Buses

Software
- p(8051, ARM, MIPS, …)
- Embedded Code (C, C++, Java, …)

SoC HW/SW Partitioning

- Partitioning into hardware and software affects overall system cost and performance.

Hardware implementation
- Provides higher performance via hardware speeds and parallel execution of operations.
- Incurs additional expense of fabricating ASICs.

Software implementation
- May run on high-performance processors at low cost (due to high-volume production).
- Incurs high cost of developing and maintaining (complex) software.
Partitioning Issues

- **Specification-abstraction level**: input definition
  - Executable languages becoming a requirement
    - Although natural languages common in practice
  - Just indicating the language is insufficient
  - Abstraction-level indicates amount of design already done
    - e.g. task DFG, tasks, CDFG, FSMD

- **Granularity**: specification size in each object
  - Fine granularity yields more possible designs
  - Coarse granularity better for computation, designer interaction
    - e.g. tasks, procedures, statement blocks, statements

- **Component allocation**: types and numbers
  - e.g. ASICs, processors, memories, buses

- **Metrics and estimations**: "good" partition attributes
  - e.g. cost, speed, power, size, pins, testability, reliability
  - Estimates derived from quick, rough implementation
  - Speed and accuracy are competing goals of estimation

- **Objective and closeness functions**
  - Combines multiple metric values
  - Closeness used for grouping before complete partition
  - Weighted sum common
    - e.g. \( k_1F(\text{area}, c) + k_2F(\text{delay}, c) + k_3F(\text{power}, c) \)

- **Output**: format and uses
  - e.g. new specification, hints to synthesis tool

- Flow of control and designer interaction
Structural v/s Functional Partitioning

- **Structural**: Implement structure, then partition
  - Good for the hardware (size & pin) estimation.
  - Size/performance tradeoffs are difficult.
  - Suffer for large possible number of objects.
  - Difficult for HW/SW tradeoff.
- **Functional**: Partition function, then implement
  - Enables better size/performance tradeoffs
  - Uses fewer objects, better for algorithms/humans
  - Permits hardware/software solutions
  - But, it’s harder than graph partitioning
SW-Oriented v/s HW-Oriented Partitioning

- **Software-Oriented Partitioning**
  - Start with all functionalities in software
  - Move portions into hardware which are
    - time-critical (bottlenecks: DCT, …)
    - cannot be allocated to software (peripheral I/O, …)

- **Hardware-Oriented Partitioning**
  - Start with all functionalities in hardware
  - Move portions into software implementation which are
    - user-interfaces
    - control actions
    - need large area
    - non-time critical

Constructive v/s Iterative Partitioning

- **Constructive Partitioning Algorithms**
  - Group objects into a complete partition
  - Use closeness metrics to group objects, hoping for a good partition
  - Spend computation time constructing a small number of partitions

- **Iterative Partitioning Algorithms**
  - Modify a complete partition in the hope that such modifications will improve the partition
  - Use an objective function to evaluate each partition
  - Yield more accurate evaluations than closeness functions used by constructive algorithms

- In practice, a combination of constructive and iterative algorithms is often employed
Contents

- Introduction
- SoC Models, Architectures, Languages
- SoC HW-SW Partitioning
  - Function-Architecture Codesign
- SoC Co-design Example

Platform-Based Design

Source: ASV
**Function-Architecture Codesign Methodology**

Refinement

Synthesis

Mapping

Verification

Abstraction

Trade-off

HW

SW

**System Level Design Vision**

Function

Constrained Optimization and Co-design

Architecture sheds light

Refinement

Abstraction

NIU 2004/05/12
Concrete Co-design Flow

Function
Graphical EFSM
ESTEREL
Reactive VHDL

Architecture
Decomposition
SHIFT of FSM Networks

Specification
Behavioral Optimization

Cost-guided Optimization
HW/SW RTOS/Interface Co-synthesis

Estimation and Validation

Macro-level Optimization
Micro-level Optimization
Resource Pool

Micro-estimated Optimization

Functional Optimization
AFFG

Behavioral Optimization

Cost-guided Optimization
HW/SW RTOS/Interface Co-synthesis

Concrete Co-design Flow

POLIS Co-design Environment

Graphical EFSM
ESTEREL
Compilers

SW Synthesis
CFSMs

SW Estimation
HW Estimation

Partitioning

Performance/trade-off Evaluation

SW Code + RTOS

Physical Prototyping

Logic Netlist

Programmable Board of choice
- FPGAs
- FPICs

POLIS Co-design Environment

NIU 2004/05/12

NIU 2004/05/12
Contents

- Introduction
- SoC Models, Architectures, Languages
- SoC HW-SW Partitioning
- Function-Architecture Codesign
  - SoC Co-design Example

Codesign Case Study

- Digital Camera Design with JPEG
  - Frank Vahid, *Embedded System Design*, Chapter 7
  - Typical SoC
Digital Camera SoC Example

- Introduction to a simple digital camera
- Designer’s perspective
- Requirements specification
- Design
  - Four implementations

Introduction to a simple digital camera

- Captures images
- Stores images in digital format
  - No film
  - Multiple images stored in camera
    - Number depends on amount of memory and bits used per image
- Downloads images to PC
- Only recently possible
  - Systems-on-a-chip
    - Multiple processors and memories on one IC
  - High-capacity flash memory
- Very simple description used for example
  - Many more features with real digital camera
    - Variable size images, image deletion, digital stretching, zooming in and out, etc.
**Designer’s perspective**

- Two key tasks
  - Processing images and storing in memory
    - When shutter pressed:
      - Image captured
      - Converted to digital form by charge-coupled device (CCD)
      - Compressed and archived in internal memory
  - Uploading images to PC
    - Digital camera attached to PC
    - Special software commands camera to transmit archived images serially

**Nonfunctional Requirements**

- Performance
  - Must process image fast enough to be useful
  - 1 sec reasonable constraint
    - Slower would be annoying
    - Faster not necessary for low-end of market
  - Therefore, constrained metric

- Size
  - Must use IC that fits in reasonably sized camera
  - Constrained and optimization metric
    - Constraint may be 200,000 gates, but smaller would be cheaper

- Power
  - Must operate below certain temperature (cooling fan not possible)
  - Therefore, constrained metric

- Energy
  - Reducing power or time reduces energy
  - Optimized metric: want battery to last as long as possible
Functional Requirements

- Flowchart breaks functionality down into simpler functions
- Each function’s details could then be described in English
  - Done earlier in chapter
- Low quality image has resolution of 64 x 64
- Mapping functions to a particular processor type not done at this stage

Refined functional specification

- Refine informal specification into one that can actually be executed
- Can use C/C++ code to describe each function
  - Called system-level model, prototype, or simply model
  - Also is first implementation
- Can provide insight into operations of system
  - Profiling can find computationally intensive functions
- Can obtain sample output used to verify correctness of final implementation
Design

- Determine system’s architecture
  - Processors
    - Any combination of single-purpose (custom or standard) or general-purpose processors
  - Memories, buses
- Map functionality to that architecture
  - Multiple functions on one processor
  - One function on one or more processors
- Implementation
  - A particular architecture and mapping
  - Solution space is set of all implementations
- Starting point
  - Low-end general-purpose processor connected to flash memory
    - All functionality mapped to software running on processor
    - Usually satisfies power, size, and time-to-market constraints
    - If timing constraint not satisfied then later implementations could:
      - use single-purpose processors for time-critical functions
      - rewrite functional specification

Implementation 1: Microcontroller alone

- Low-end processor could be Intel 8051 microcontroller
- Total IC cost including NRE about $5
- Well below 200 mW power
- Time-to-market about 3 months
- However, one image per second not possible
  - 12 MHz, 12 cycles per instruction
    - Executes one million instructions per second
  - CcdppCapture has nested loops resulting in 4096 (64 x 64) iterations
    - ~100 assembly instructions each iteration
    - 409,000 (4096 x 100) instructions per image
    - Half of budget for reading image alone
  - Would be over budget after adding compute-intensive DCT and Huffman encoding
Implementation 2: Microcontroller and CCDPP

- CCDPP function implemented on custom single-purpose processor
  - Improves performance – less microcontroller cycles
  - Increases NRE cost and time-to-market
  - Easy to implement
    - Simple datapath
    - Few states in controller
- Simple UART easy to implement as single-purpose processor also
- EEPROM for program memory and RAM for data memory added as well

Analysis of implementation 2

- Total execution time for processing one image:
  - 9.1 seconds
- Power consumption:
  - 0.033 watt
- Energy consumption:
  - 0.30 joule (9.1 s x 0.033 watt)
- Total chip area:
  - 98,000 gates
Implementation 3: Microcontroller and CCDPP/Fixed-Point DCT

- 9.1 seconds still doesn’t meet performance constraint of 1 second
- DCT operation prime candidate for improvement
  - Execution of implementation 2 shows microprocessor spends most cycles here
  - Could design custom hardware like we did for CCDPP
    • More complex so more design effort
  - Instead, will speed up DCT functionality by modifying behavior: floating point $\rightarrow$ fixed point representation

Implementation 3: Microcontroller and CCDPP/Fixed-Point DCT

- Analysis of implementation 3
  - Use same analysis techniques as implementation 2
  - Total execution time for processing one image:
    • 1.5 seconds
  - Power consumption:
    • 0.033 watt (same as 2)
  - Energy consumption:
    • 0.050 joule (1.5 s x 0.033 watt)
    • Battery life 6x longer!!
  - Total chip area:
    • 90,000 gates
    • 8,000 less gates (less memory needed for code)
Implementation 4: Microcontroller and CCDPP/DCT

- Performance close but not good enough
- Must resort to implementing CODEC in hardware
  - Single-purpose processor to perform DCT on 8 x 8 block

Implementation 4: Microcontroller and CCDPP/DCT

- Analysis of implementation 4
  - Total execution time for processing one image:
    - 0.099 seconds (well under 1 sec)
  - Power consumption:
    - 0.040 watt
    - Increase over 2 and 3 because SOC has another processor
  - Energy consumption:
    - 0.00040 joule (0.099 s x 0.040 watt)
    - Battery life 12x longer than previous implementation!!
  - Total chip area:
    - 128,000 gates
    - Significant increase over previous implementations
Summary of implementations

<table>
<thead>
<tr>
<th></th>
<th>Implementation 2</th>
<th>Implementation 3</th>
<th>Implementation 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance (second)</td>
<td>9.1</td>
<td>1.5</td>
<td>0.099</td>
</tr>
<tr>
<td>Power (watt)</td>
<td>0.033</td>
<td>0.033</td>
<td>0.040</td>
</tr>
<tr>
<td>Size (gate)</td>
<td>98,000</td>
<td>90,000</td>
<td>128,000</td>
</tr>
<tr>
<td>Energy (joule)</td>
<td>0.30</td>
<td>0.050</td>
<td>0.0040</td>
</tr>
</tbody>
</table>

- **Implementation 3**
  - Close in performance
  - Cheaper
  - Less time to build

- **Implementation 4**
  - Great performance and energy consumption
  - More expensive and may miss time-to-market window
    - If DCT designed ourselves then increased NRE cost and time-to-market
    - If existing DCT purchased then increased IC cost

**Which is better?**

Reference Books