SOC Design Process
SOC Design Process

- SOC design flow
- System level design issues
- Macro design flow
1. SOC Design Flow

- To meet challenges of SOC, design flow changes from
  - From a waterfall model to a **spiral model**
  - From a top-down to a **combination** of top-down and bottom-up
Traditional ASIC Design Flow

- Waterfall model
- Recursive
  - “From error to where?”
- Verification Strategy
  - “Design is becoming COMPLEX!”
- Time-To-Market Pressure
- What’s the problem
  - Handoff are rarely clean
  - Larger, deep submicron designs
    - co-development for HW and SW
    - Physical issues
SOC Design Process

• Evolution: waterfall to spiral model
  – Addressing these problems concurrently
    • Functionality,
    • Timing,
    • Physical design and
    • Verification
  – Incrementally improving as design converges
• Top-down to combination of top-down and bottom-up
  – Bottom-up with critical low-level blocks, reuse soft or hard macros
Spiral Model

Goal: Maintain parallel interacting design flows

SYSTEM DESIGN AND VERIFICATION

PHYSICAL
- Physical specification: area, power, clock tree design
- Preliminary floorplan
- Updated floorplans
- Trial placement

TIMING
- Timing specification: I/O timing, clock frequency
- Block timing specification
- Top-level synthesis

HARDWARE
- Hardware specification
- Block selection/design
- Block verification
- Top-level verification

SOFTWARE
- Software specification
- Algorithm development & macro decomposition
- Application prototype development
- Application prototype testing
- Application development
- Application testing

Final place and route
Tapeout

Copyright ©2003 All rights reserved
Waterfall v.s. Spiral

• Waterfall
  – Work well up to 100K gate and down .5u
  – Serial H/W and S/W development

• Spiral
  – For large, deep submicron designs
  – Parallel development of H/W & S/W
  – Parallel verification and synthesis
  – Floorplaning and P & R in synthesis process
  – Use predesigned Macros (Hard/Soft)
  – Planned iteration throughput

“H/W and S/W development concurrently : functionality, timing, physical design, and verification”
Top-Down vs. Bottom-Up

• Classical top-down
  – Begin with spec and decomposition
  – End with integration and verification
  – Assuming lowest level block, pre-designed
    • Too ideal to be easily broken and cause unacceptable iteration

• Real-world design team
  – **Mixture** of top-down and bottom-up design
  – Building critical low-level blocks early
  – Libraries of reusable hard and soft macros helps this process
“Construct by Correction”

- **Construct by correction**
  - Made the **first pass ASAP**, and refine later
  - Why
    - allow for multiple iterations
  - Used in Sun Microsystem’s UltraSPARC design methodology
    - “One of the most successful in Sun Microsystem’s History”
  - Take from architecture definition through P & R
  - Foresee impact of architectural decision on final design: area, power, performance
  - Target
    - larger, complex designs
  - **Correction by construct**
    - Make the first pass completely right
    - Target
      - small designs
Key to SOC Design Process

- **Iteration** is an inevitable part of the design process
- The problem is **how large the loop is**
- **Goal**
  - Minimize the overall design time
- **But How**
  - Planned for iterations
  - Minimize iteration numbers
    - especially major loops (Spec to chip)
  - Local loop is preferred
    - coding, verifying, synthesizing small blocks
  - IP clearly help due to pre-verified
  - Parameterized blocks offer more tradeoff between area, performance and functionality
- Carefully designed **spec** is the best way to minimize the loops
Specification Problems

• First part of design process
  – Most crucial, challenging, lengthy phase of project

• Why it is so important
  – Specification is your destination
    • If you know it exactly, you can spot the error path and fix it quickly
    • If not, you may not spot major errors until late

• Now the question
  – When shall you document your specification
    • Early phase in the design cost less and more valuable
    • Later phase may only delays the project or be skipped
Purpose of Specification

• Specification for Integration
  – Functional/Physical/Design requirements
  – The block diagram
  – Interfaces to external system
  – Manufacturing test methodology
  – Software model
  – Software requirements

• Specification for block Design
  – Algorithm spec
  – Interface spec
  – Authoring guide
  – Test Spec – lint & coverage
  – Synthesis constraints
  – Verification environment, tools used
Types of Specifications

• Written in natural language
  – Traditional, ambiguous, incompleteness, erroneous

• Formal specification
  – Desired characteristic (functionality, timing, power, area,…), independent to implementation
  – Not widely used, important research topic

• Executable specification
  – Description of functional behavior
  – Parallel with RTL Model in the TestBench
Executable Specification

• Procedural language for behavioral modeling
  – Design productivity
    • Easy to model complex algorithm
    • Fast execution
    • Simple testbench
  – Tools
    • Native C/C++ through PLI/FLI
    • Extended C/C++ : SpecC, SystemC

• Verify it on the fly!
  – Test vector generation
  – Compare RTL code with behavioral model
  – Coverage test
Using Executable Specifications

• Ensure **completeness** of specification
  – Even components (e.g. peripherals) are so complex
  – Create a program that behave the same way as the system

• Avoid **unambiguous** interpretation of the specification
  – Avoids unspecified parts and inconsistencies
  – IP customer can evaluate the functionality up-front

• **Validate system functionality** before implementation
  – Early feedback from customer
  – Create early model and validate system performance

• Refine and test the implementation of the specification
  – Test automation improves time-to-market
Executable Spec Motivation

- **Customer System**
- **Paper Spec**
- **HDL Design**
- **Netlist**
- **Layout**
- **Silicon**

**Verification, Error Checking Bottleneck**

**HDL TestBench with C-Interface (PLI/FLI)**

- **Customer System**
- **Executable Spec**
- **HDL Design**
- **Netlist**
- **Layout**
- **Silicon**
Time Spent in Design Phases

Conventional methodologies:
- Product Planning: 14%
- System Design: 12%
- Logic Design: 18%
- Physical Design & Assembly: 13%
- Prototype Debug: 43%

Time Spent Debugging:
- Product Requirements Mis-communicated By customer: 20%
- Specification incorrectly Translated or ambiguous: 50%
- Incorrect Login in Design: 30%

Source: Toshiba/Collet/STOC
Specification Based Design

C/C++ System Level Model

Analysis

Results

Test Bench

- C-to-HDL Interface (PLI/FLI)
- Test Vector (VCD/WAVES)
- Waveform Compare

Executable Spec.

Conversion

HDL

Simulation

Synthesis

Netlist

Waveform Compare

P & R

Netlist

Simulation

Silicon
**System Design Process**

1. IDENTIFY
   - system requirements

2. WRITE
   - preliminary specifications

3. DEVELOP
   - high-level algorithmic model
     - C/C++/MATLAB/SES/
       - NuThena/Bones/COSSAP

4. REFINE and TEST
   - algorithms
     - C/C++/COSSAP/SPW/SDL

5. DETERMINE
   - hardware/software partition

6. WRITE
   - hardware specification

7. DEVELOP
   - behavioral model for hardware

8. PARTITION
   - into macros
     - Macro 1 ... Macro n

9. WRITE
   - preliminary specification for macro

10. DEFINE
    - interfaces

11. WRITE
    - software specification

12. DEVELOP
    - prototype of software

13. COSIMULATION

14. DEVELOP software

Characterized library of hardware/software
macros & interface protocols
SoC Design Characteristics

• Design Level
  – RTL / Behavioral > Architectural / VC Evaluation

• Design Team
  – Small, Focused > Multidisciplinary> Multi-Group, Multidisciplinary

• Primary Design
  – Custom Logic > Blocks, Custom Interface> Interface to System / Bus

• Design Reuse
  – Opportunistic Soft, Firm and Hard > Planned Firm and Hard

• Optimization Focus
  – Synthesis, Gate-level > Floor planning, Block Architecture > System Architecture
SoC Test Characteristics

- Test Architecture
  - Scan/JTAG/BIST/Custom
    > Hierarchical, Parallel scan/JTAG/BIST/custom
- Bus Architecture
  - Custom > Standardized / Multiple app-specific
- Verification Level
  - Gate/RTL > Bus functional/RTL/Gate
    > Mixed (ISS to RTL with H/W and S/W)
- Partitioning Focus
  - Synthesis limitation > Functions / Communication
SoC Layout Characteristics

- Placement
  - Flat > Flat with limited hierarchical > Hierarchical
- Routing
  - Flat > Flat with limited hierarchical > Hierarchical
- Timing
  - Flat > Flat with limited hierarchical > Hierarchical
- Physical Verification
  - Flat > Flat with limited hierarchical > Hierarchical
Transition of SoC Design Methodology

- From area-driven to timing-driven design
- From block-based to platform-based design

Design Methodology
SoC Design Methodology

- Transition of Design Methodology
  - ADD > TDD > BBD > PBD
- Reuse—the key to SoC design
  - Personal > Source > Core > Virtual Component
- Integration approach
  - IP-Centric vs. Integration-Centric Approach
- SoC and productivity
  - Executable specification
    - Test automation
    - Real-world stimuli
    - Higher-level algorithmic system modeling
2. System-Level Design Issues

Key Aspects of Design Reuse

- **Fundamentals**
  - Well-designed IP is the key to successful SOC design
- **System level design guidelines**
  - To produce well-designed IP
  - To integrate well-designed IP to an SOC design
  - Driven by the needs of IP integrator and chip designer
- **Principles behind these guidelines**
  - **Discipline**
    - Consistent good practices
  - **Simplicity**
    - The simpler the design, the easier to fix the bugs
  - **Locality**
    - Make timing and verification problem local by careful block and interface design
Full Custom Design in Reuse

- Full custom design
  - Design that are not from synthesis

- Major problems
  - Performance gain is limited
  - Non-portable, hard to modify designs
  - Redesign take time

- **Limit** full custom design for only small part of design
  - Even aggressive processor designer uses full custom only for data path
Interface and Timing Closure

• Timing problems due to deep submicron process
  – Dominated wire delay
  – Imprecise wireload model due to uncertainty of wire delays

• Solution
  – Tools
    • Timing driven P&R, Physical synthesis
  – Tactics for fundamental good design
    • Register all inputs/outputs of the macro
      – Unit for floorplan
    • Register all outputs of the subblock of macro
      – Unit for synthesis
  • Exception
    – Cache interface
    – Design likes PCI interface that needs glue logic at the interface
Synchronous v.s. Asynchronous

• Synchronous
  – Avoid asynchronous and multi-cycle paths
  – Tools work best for synchronous design
    • Accelerate synthesis and simulation
  – Ease static timing analysis

• Register based
  – Use (positive) edge triggered DFF
  – Latches shall be used only in small memory or FIFOs
Clocking

• Clock planning
  – Minimize the number of clock domains
  – Isolate the interface between clock domains
  – Careful synchronizer design to avoid metastability
  – Isolate clock generation and control logic

• Document the clock scheme
  – Required clock frequencies and PLL
  – Interface timing requirements to other parts of the system

• PLL
  – Disabling/bypassing scheme
  – Ease testing

• For hard blocks
  – Eliminate the clock delay using a PLL
  – Balance the clock insertion delay
Reset

- Synchronous reset
  - Easy to synthesize
  - Requires a free-running clock

- Asynchronous reset
  - Do not require a free-running clock
  - Not affect flip-flop data timing due to separated input
  - Harder to implement, like clock, CTS is required
  - Synchronous de-assertion problem
  - Make STA and cycle-based simulation more difficult

- **Asynchronous reset** is preferred
Internal Generated Reset

- Internal generated reset causes unwanted reset during scan shift
- Solution
  - Force internal generated reset signal inactive during test

Diagram:
- Power-on reset
- Test mode
- Reset to all FF
Design for Verification

• Principle of **locality**
• **Plan** before design starts
• Testbenches should reflect the system environment
• Best strategy
  – Bottom-up verification
  – Challenges: developing testbench
  – Solution
    • Macros with clean, well-designed interface
    • High level verification languages + code coverage tool
System Interconnection

- Tri-state bus is not good
  - Bus contention problem
    - Reduce reliability
    - One and only one driver at a time
      - Harder for deep submicron design
  - Bus floating problem
    - Reduce reliability
    - Bus keeper
  - ATPG problem
  - FPGA prototyping problem

- Multiplexer-based bus is better
IP-to-IP Interface

- Direct connection (via FIFO)
  - Higher bandwidth
  - Redesign for different IP
  - Become unmanageable when the IP number increases
  - Only suitable for design connected to analog block, e.g. PHY

- **Bus-based**
  - Eliminate direct link
  - Layered approach can offer higher bandwidth
  - All IPs talk to bus only, thus only **IP-to-bus problem**
  - The mainstream of current IP-based SOC integration

- Choose the **standard bus** whenever possible
On-chip Bus (OCB)

• **ARM AMBA**
  – Advanced Microcontroller Bus Architecture
  – Dominant player
  – V 3.0 is on the road
  – Available solution
    • Synopsys DW_AMBA, …

• Sonics OCP
• VSIA OCB 2.1
• WishBone Silicore
• IBM CoreConnect
• ….
AMBA Bus System

**AMBA Advanced High-performance Bus (AHB)**
- High performance
- Pipelined operation
- Burst transfers
- Multiple bus masters
- Split transactions

**AMBA Advanced Peripheral Bus (APB)**
- Low power
- Latched address and control
- Simple interface
- Suitable for many peripherals
Design for Debug: On-chip Debug

• Experienced teams assume chip won’t work when first power up and plan accordingly.

• Challenges for IP test
  – IPs are deeply embedded within the SOC design
  – Disaster to the system and S/W engineers

• Solution
  – Principle: increase controllability and observability
  – Add debug support logic to the hardware
  – MUX bus to existing I/O pins
Low Power (1/3)

\[ P = \sum \alpha CV^2 f \]

\( \alpha \): switching activity, \( C \): capacitance, \( V \): supply voltage, \( f \): frequency

- **Reduce the supply voltage**
  - Process improvement

- **Reduce capacitance**
  - Low power cell and I/O library
  - Less logic for the same performance

- **Reduce switching activity**
  - Architecture and RTL exploration
  - Power-driven synthesis
  - Gate-level power optimization
Low Power (2/3)

- Memory
  - Dominated power consumption
  - Low-power memory circuit design
  - Partition a large memory into several small blocks
  - Gray-coded address interface
Low Power (3/3)

- Clock gating
  - 50% - 70% power consumed in clock network reported
  - gating the clock to an entire block
  - gating the clock to a register

```verilog
always @(posedge clk)
  if(en)
    q <= q_nxt;
assign clk1 = clk & en;
always @(posedge clk1)
  q <= q_nxt;
```
Design for Test

• Memory test
  – Memory BIST is recommended

• Processor test
  – Chip level test controller (including scan chain controller and JTAG controller)
  – Use shadow registers to facilitate full-scan testing of boundary logic

• Other macros
  – **Full scan** is strongly recommended

• Logic BIST
  – Embedded stimulus generator and response checker
  – Not popular yet
3. Macro Design Process

- Top-level macro design
- Subblocks design
- Integrate subblocks
- Macro productization
Problem in SoC Era

- Productivity gap
- Time-to-market pressure
- Increasing design complexity
  - HW/SW co-development
  - System-level verification
  - Integration on various levels and areas of expertise
  - Timing closure due to deep submicron

Solution: Platform-based design with reusable IPs
Design for Reuse IPs

- Design to maximize the flexibility
  - configurable, parameterizable
- Design for use in multiple technologies
  - synthesis script with a variety of libraries
  - portable for new technologies
- Design with complete verification process
  - robust and verified
- Design verified to a high level of confidence
  - physical prototype, demo system
- Design with complete document set
Parameterized IP Design

• Why to parameterize IP?
  – Provide flexibility in interface and functionality
  – Facilitate verification

• Parameterizable types
  – Logic/Constant functionality
  – Structural functionality
    • Bit-width, depth of FIFO, regulation and selection of sub-module
  – Design process functionality (mainly in test bench)
    • Test events
    • Events report (what, when and where)
    • Automatic check event
  – Others (Hardware component Modeling, 1996)
IP Generator/Compiler

• User specifies
  – Power dissipation, code size, application performance, die size
  – Types, numbers and sizes of functional unit, including processor
  – User-defined instructions.

• Tool generates
  – RTL code, diagnostics and test reference bench
  – Synthesis, P&R scripts
  – Instruction set simulator, C/C++ compiler, assembler, linker, debugger, profiler, initialization and self-test code
Logic/Constant Functionality

- Logic Functionality
  - Synthesizable code
    ```
    always @(posedge clock) begin
      if (reset==`ResetLevel) begin
        ...
      end
      else begin
        ...
      end
    end
    ```

- Constant Functionality
  - Synthesizable code
    ```
    assign tRC_limit= (`RC_CYC > (`RCD_CYC + burst_len)) ? `RC_CYC - (`RCD_CYC + burst_len) : 0;
    ```

  - For test bench
    ```
    initial begin
      #(`T_CLK) event_1;
      #(`T_CLK) event_2;
      ...
    end
    ```
Reusable Design - Test Suite

- Test events
  - Automatically adjusted when IP design is changed
  - Partition test events to reduce redundant cases when test for all allowable parameter sets at a time
- Debug mode
  - Test for the specific parameter set at a time
  - Test for all allowable parameter sets at a time
  - Test for the specific functionality
  - Step control after the specific time point
- Display mode of automatic checking
  - display[0]: event current under test
  - display[1]: the time error occurs
  - display[2]: expected value and actual value
  - ...

Reusable Design - Test Bench

- Use Global Connector to configure desired test bench
  - E.g.: bus topology of IEEE 1394
Characteristics of Good IP

• Configurability
• Standard interface
• Compliance to defensive design practices
• Complete set of deliverables
  – Synthesizable RTL
  – Verification suite
  – Related scripts of EDA tools
  – Documentations
IP Core Macro Design Process

Develop functional specification

Develop behavioral model

Develop testbench

Test behavioral model

Create behavioral model

Partition design into subblocks

Write functional specification

Write technical specification

Develop timing constraints

Write RTL

Run lint

Synthesis

Simulate

Perform power analysis

Measure test coverage

PASSES - READY FOR INTEGRATION

Completed behavioral model for HW/SW cosimulation and test development

Perform these steps for each subblock

Meets timing, power, & area requirements

Coverage tool passes

Macro Integration Process


Copyright ©2003 All rights reserved
Four Major Phases

- Design top-level macro
  - macro specification; behavior model
  - macro partition
- Design each subblock
  - specification and design
  - testbench; timing, power check
- Integration subblocks
- Macro productization
Specification at Every Level

• Overview
• Functional requirements
• Physical requirements
• Design requirements
• Block diagram
• Interface to external system
• Manufacturing test methodology
• Software model
• Software requirement
• Deliverables
• Verification
Top-Level Macro Design Flow

Macro specification

DEVELOP detailed technical specification

CODE behavioral model
C/Verilog/VHDL

PARTITION
the block into subblocks

TEST behavioral model

CODE testbench
C/Verilog/VHDL/Vera/Specman

Completed behavioral model for HW/SW cosimulation and test development

CERETE BEHAVIOURAL MODEL

Top-Level Macro Design

• Updated macro hardware specification
  – document

• Executable specification
  – language description
  – external signals, timing
  – internal functions, timing

• Behavioral model
  – SystemC, HDL

• Testbench
  – test vector generation, model for under test unit, monitoring and report

• Block partition
Subblock Design Flow

WRITE functional specification

WRITE technical specification

DEVELOP timing constraints

WRITE RTL

RUN Lint

DEVELOP testbench

SYNTHESIS

Design Compiler

SIMULATE

Verilog/VHDL

PERFORM power analysis

PowerCompiler/QuickPower

MEASURE testbench coverage

VHDLCover/VeriSure/CoverMeter

Meets timing, power, & area requirements

Coverage tool passes

PASSES - READY FOR INTEGRATION

Subblock Design

• Design elements
  – Specification
  – Synthesis script
  – Testbench
  – Verification suite
  – RTL that pass lint and synthesis
Linter

- Fast static RTL code checker
  - preprocessor of the synthesizer
  - RTL purification
    - syntax, semantics, simulation
  - timing check
  - testability checks
  - reusability checks
- Shorten design cycle by avoiding lengthy iterations
Subblock Integration Flow

**Top-level HDL**

- **DETERMINE** configuration and **GENERATE** top-level HDL
- **FUNCTIONAL VERIFICATION**
  - Verilog/VHDL simulator
  - ModelSim, VSS, VCS
- **RUN lint**
  - Verilint, VHDLlint
- **GENERATE** top-level synthesis scripts
  - SYNTHESIZE with reference library
    - Design Compiler
- **SYNTHESIZE**
  - Scan insertion, ATPG, coverage analysis
  - Test Compiler, DFTAdvisor, FastScan/FlexTest
- **PERFORM analysis**
  - QuickPower, Power Compiler
- **READY FOR PRODUCTION**
- **PRODUCTIZE** as soft macro
- **PRODUCTIZE** as hard macro

**Subblock 1**
Subblock Integration

- Integration process is complete when
  - top-level RTL, synthesis script, testbench complete
  - macro RTL passes all tests
  - macro synthesizes with reference library and meets all timing, power and area criteria
  - macro RTL passes lint and manufacturing test coverage
Macro Productization

From block integration:

1. DEVELOP specification for prototype chip
2. DESIGN chip
3. SYNTHESIS chip
4. Scan insertion, ATPG and coverage analysis
5. FLOORPLAN
6. PLACE and ROUTE
7. VERIFY timing
8. FABRICATE
9. TEST chip in demo board
10. TRANSLATE Verilog ↔ VHDL
11. REGRESSION TEST on translated code
12. RUN TESTS on multiple simulators
13. RUN Pre-sim on one technology
14. Formal Verification RTL vs. gates
15. CREATE user documents: e.g.,
   - user guide
   - Verification guide
   - Integration guide
   - Test guide
16. Release
Soft Macro Production

• Produce the following components
  – Verilog version of the code, testbenches, and tests
  – Supporting scripts for the design
    • installation script
    • synthesis script
  – Documentation