Lecture 7
SoC Design Process (Part III)

Xuan ‘Silvia’ Zhang
Washington University in St. Louis

ESE 566A: Modern System-on-Chip Design
ese.wustl.edu/~xuan.zhang/ese566
Previously on ESE 566A:

- IDENTIFY requirements
- DEVELOP software
- WRITE preliminary specs
- WRITE and DEVELOP SW prototype
- UNDERTAKE HW/SW partition
- VERIFY/TEST HW models
- CO-SIMULATION
- DEVELOP REFINE Algorithms
- Library
Canonical SoC

- Centered around the processor
- Bus-based communication link
Software IP Challenges

- Similar Problems with HW IP
  - product complexity
  - sophisticated applications
  - multi-domain applications
  - time-to-market

- Unique Challenges
  - proliferation of processor cores
  - a few dominate players: e.g. ARM

Most software IP is developed internally.
Writing SoC Software

• **Assembly Language**
  - performance critical code
  - tailored code structure
  - difficult to maintain and to develop
  - modern compilers do pretty well

• **High Level Language**
  - better abstraction, readability, maintainability
  - shorter time-to-market
  - good reusability
  - performance overhead
  - non-portable constructs and machine dependences
Software Architecture

<table>
<thead>
<tr>
<th>Host Applications</th>
<th>GUI</th>
<th>Error Handler</th>
<th>Memory Allocation</th>
<th>Diagnostics, Debug, Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messaging</td>
<td>Scheduler</td>
<td>State Machine</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application Program Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol Stack</td>
</tr>
<tr>
<td>Display Services</td>
</tr>
<tr>
<td>File Manager</td>
</tr>
</tbody>
</table>

Device Drivers

Hardware
Machine Model

- Interface between SW and HW
  - memory layout
  - interrupt handling mechanism
  - memory-mapped I/O
  - architecturally visible registers
System Memory Layout

- **High address**
  - Memory-Mapped I/O Registers
  - I/O registers mapped into the system memory address space for access via software
- **Application Memory**
  - Memory space for program data, stack, and code
- **Interrupt Service Routines**
  - Routines for handling services for interacting with system devices
- **Low address**
  - Interrupt Vector
  - Interrupt dispatch table
# Program Memory Layout

<table>
<thead>
<tr>
<th>High address</th>
<th>Arguments and program environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Stack (grows downward)</td>
<td>In program environment, includes arguments passed to program</td>
</tr>
<tr>
<td>...</td>
<td>Program stack holds local variables and arguments to functions; traditionally grows downward</td>
</tr>
<tr>
<td>Program Heap (grows upward)</td>
<td>Program heap holds dynamically allocated objects (may not always be present)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process Image</th>
<th>BSS</th>
<th>Static uninitialized data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td></td>
<td>Static initialized data (globals)</td>
</tr>
<tr>
<td>Text</td>
<td></td>
<td>Program code</td>
</tr>
</tbody>
</table>

Low address
Interrupts

- Peripheral Device Control
  - interrupt signal asserted when device needs service

Interrupt Handling

... → Interrupt service routine (ISR)

ff00

... → Vectored interrupts
On interrupt, transfer control to a address for that interrupt that contains a jump to the interrupt service routine

Jmp ff00

Alternative: Table may hold ISR address
Interrupt Service Routines (ISR)

- **Interrupt Latency**
  - time for the ISR to be invoked after interrupt

- **ISR Steps**
  - save context for ISR
  - disable other interrupts
  - handle the interrupt
  - restore context
  - enable other interrupts
  - return from interrupt
System Software

- Collection of routines and services that manage the machine
  - interact and control I/O
  - load a program into memory
  - manage multiple tasks
  - provide means of communication with external devices
  - handle exceptions and interrupt dispatch
  - services to coordinate execution of tasks
System Software

Varies from bare-bones to powerful and a full array of system services.
Example: Software for VisorVoice

- ARM software for (de-)compression
  - single task, simple monitor and interrupts

- Interrupts
  - process incoming data from digital filter (compression)
  - send data to digital filter (decompression)
  - message from user interfaces (e.g. change equalizer settings)
  - reset condition
Software Development Phases

- **Phase 1:** Develop on host workstation
  - implement modules with desired functionality and interface; fast testing of software robustness

- **Phase 2:** Move to target processor ISA simulator
  - get early estimate of instruction count
  - write performance-critical code in assembly and verify

- **Phase 3:** Timing accurate simulation of processor
  - improve on timing estimate on faithful processor model
  - simulation time grows by 100-1000x
  - go back to host and ISA simulator when bugs are found

- **Phase 4:** Integrate software into complete design
  - move to full co-design environment
  - interact with hardware verified
Hardware Co-design and Co-verification

- Iteratively integrating blocks
  - into a single complete design
  - substitute software model with hardware blocks for verification

Codec becomes available and is integrated
Common Problems

- **Mismatched Interfaces**
  - outside IP uses different interface standards
  - bus-to-bus interfaces
  - block-to-bus interfaces
  - block-to-block interfaces

- **Solution: Hardware Wrappers**
  - map interfaces to be consistent
  - between interfaces
  - between different protocols
  - between different data rates
  - between exception models
Example: Wrapper Between Blocks

Wrapper may map a block with a simple handshake protocol, separate address and data, and unidirectional data lines to another block that has a more complex handshake with bidirectional data lines and address/data multiplexing.
Example: Wrapper to a Bus

Block with narrow data interface

Wrapper maps a narrow 8-bit interface to an industry standard bus (e.g., ARM’s AHB) with a 32-bit data width and multiplexed address/data.
Verifying Blocks

- **Test Vectors**
  - input stimuli to test a block with expected output

- **Test Benches**
  - HDL code that generates input and verifies output
  - model interface protocol to block
  - document tests when design changes or questions arise
Example: Testing Memory

• Simple Write/Read Test
  - write to a location
  - read from that location
  - compare read value to written value
Example: Testing Memory

Write/Read/Verify Test
Write 0xf0f0 to location 0x00ff
Read from 0x00ff
Verify read data is 0xf0f0
System-Level Verification

- **Verification Goals**
  - expected functionality
  - meet timing requirements
  - meet area, power requirements

- **Verification Strategy**
  - standalone leaf nodes (lowest individual block)
  - interfaces: transaction and data content
  - run increasingly complex applications
  - prototype full design with full software

Verification is reported to take up to 80% of design effort!
Questions?

Comments?

Discussion?
Lab Tutorial

- Keil MDK-ARM Microcontroller Development Kit
  - µVision5 IDE: debugger, and simulation environment
  - support ARM Cortex-M family processors
  - MDK-Lite free download: https://www.keil.com/demo/eval/arm.htm
Test Bench Generation

• Verify Individual Blocks
  - transaction verification
  - data and behavioral verification

• Construct Test Cases
  - knowledge of designer on common and boundary cases
  - random testing to improve coverage
  - automatic test case generation

• Test Benches as Documentation
  - verification approach and methodology
  - tests performed
  - interfaces to blocks
  - archive for future design re-use
Design Flow Division

- **Front End**
  - specify, explore
  - design, capture
  - synthesize
  - output: structural RTL

- **Back End**
  - input: structural
  - place and route
  - mask making
  - fabrication
Front End Design Flow
Back End Design Flow