ECE 587 – Hardware/Software Co-Design
Lecture 02 Abstraction Levels and Models

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

January 14, 2015
Reading Assignment

- This lecture: 1, 2
- Next lecture: 3.1
Outline

System Design Challenges

Models

An Example Design
The Productivity Gap

- System complexity increases almost exponentially
  - Software: more lines of code
  - Hardware: more transistors to use
- Designer’s ability increases slowly
  - How many components can you manage in your mind?
- There is a huge gap between what is available for us to design and what we can manage to design
- Can we simply increase team size to overcome the complexity?
  - Not always successful according to software engineering practices, especially when robustness and reliability are of concern.
- Commonly accepted solution
  - Raise the level of abstraction in the design process
  - For example, hierarchical designs
The Productivity Gap

- System complexity increases almost exponentially
  - Software: more lines of code
  - Hardware: more transistors to use
- Designer’s ability increases slowly
  - How many components can you manage in your mind?
- There is a huge gap between what is available for us to design and what we can manage to design
- Can we simply increase team size to overcome the complexity?
  - Not always successful according to software engineering practices, especially when robustness and reliability are of concern.
- Commonly accepted solution
  - Raise the level of abstraction in the design process
  - For example, hierarchical designs
The Productivity Gap

- System complexity increases almost exponentially
  - Software: more lines of code
  - Hardware: more transistors to use
- Designer’s ability increases slowly
  - How many components can you manage in your mind?
- There is a huge gap between what is available for us to design and what we can manage to design
- Can we simply increase team size to overcome the complexity?
  - Not always successful according to software engineering practices, especially when robustness and reliability are of concern.
- Commonly accepted solution
  - Raise the level of abstraction in the design process
  - For example, hierarchical designs
The Productivity Gap

- System complexity increases almost exponentially
  - Software: more lines of code
  - Hardware: more transistors to use
- Designer’s ability increases slowly
  - How many components can you manage in your mind?
- There is a huge gap between what is available for us to design and what we can manage to design
- Can we simply increase team size to overcome the complexity?
  - Not always successful according to software engineering practices, especially when robustness and reliability are of concern.
- Commonly accepted solution
  - Raise the level of abstraction in the design process
  - For example, hierarchical designs
The Productivity Gap

- System complexity increases almost exponentially
  - Software: more lines of code
  - Hardware: more transistors to use
- Designer’s ability increases slowly
  - How many components can you manage in your mind?
- There is a huge gap between what is available for us to design and what we can manage to design
- Can we simply increase team size to overcome the complexity?
  - Not always successful according to software engineering practices, especially when robustness and reliability are of concern.
- Commonly accepted solution
  - Raise the level of abstraction in the design process
  - For example, hierarchical designs
Abstraction Levels

- Abstraction helps to hide details, e.g.
  - Logic gates vs. transistors for hardware design
  - Reasonings are easier and more relevant at the higher abstraction level (logic gates) using boolean logic than at the lower one (transistors) using voltages and currents.
  - There are less components at the higher abstraction level.

- To close the productivity gap, it is desired to design the system at higher abstraction levels and not to provide any lower level detail at all.
  - Designers provide *specifications* (descriptions at higher abstraction levels).
  - Design time is reduced by applying design automation that synthesizes *implementations* (details at lower abstraction levels).
  - Avoid error prone manual design to improve robustness and reliability
Abstraction Levels

- Abstraction helps to hide details, e.g.
  - Logic gates vs. transistors for hardware design
  - Reasonings are easier and more relevant at the higher abstraction level (logic gates) using boolean logic than at the lower one (transistors) using voltages and currents.
  - There are less components at the higher abstraction level.

- To close the productivity gap, it is desired to design the system at higher abstraction levels and not to provide any lower level detail at all.
  - Designers provide specifications (descriptions at higher abstraction levels).
  - Design time is reduced by applying design automation that synthesizes implementations (details at lower abstraction levels).
  - Avoid error prone manual design to improve robustness and reliability
More about Abstraction Levels

- How to define an abstraction level?
  - Designers should have consensus on the definition to facilitate communications, e.g. what are logic gates.
  - The definition should involve some kind of mathematics to make automatic synthesis possible, e.g. boolean logic.

- At what abstraction level should designers work?
  - Designers should be able to reason about the system very effectively at such level, as this will help to
  - Reduce design time by ignoring unnecessary details, e.g. a logic gate can be used directly without any understanding on its implementation.
  - Improve design quality by eliminating chances to make mistakes, e.g. you will never implement the logic gate the wrong way.
More about Abstraction Levels

- How to define an abstraction level?
  - Designers should have consensus on the definition to facilitate communications, e.g. what are logic gates.
  - The definition should involve some kind of mathematics to make automatic synthesis possible, e.g. boolean logic.
- At what abstraction level should designers work?
  - Designers should be able to reason about the system very effectively at such level, as this will help to
  - Reduce design time by ignoring unnecessary details, e.g. a logic gate can be used directly without any understanding on its implementation.
  - Improve design quality by eliminating chances to make mistakes, e.g. you will never implement the logic gate the wrong way.
Outline

System Design Challenges

Models

An Example Design
To specify a system at certain abstraction levels, *sufficient details* are needed to predict system behavior with *absolute precision*.

An intuitive way to specify a system is to specify its subsystems and their interactions.

- E.g. hierarchical design

Model: defining an abstraction level by defining a method for decomposition

- Types of the subsystems
- Rules for composing them into the system
To specify a system at certain abstraction levels, *sufficient details* are needed to predict system behavior with *absolute precision*.

An intuitive way to specify a system is to specify its subsystems and their interactions.
- E.g. hierarchical design

Model: defining an abstraction level by defining a method for decomposition
- Types of the subsystems
- Rules for composing them into the system
To specify a system at certain abstraction levels, *sufficient details* are needed to predict system behavior with *absolute precision*.

An intuitive way to specify a system is to specify its subsystems and their interactions.

- E.g. hierarchical design

Model: defining an abstraction level by defining a method for decomposition

- Types of the subsystems
- Rules for composing them into the system
Considerations for Models

- No ambiguity and complete
  - Help to distinguish abstraction levels with subtle differences
  - Make reasonings about the system easier
    - Models come from experiences of expert designers.
    - Modifying a subsystem will also become easier.
- Make communications easier
  - System design is a team work.
Considerations for Models

- No ambiguity and complete
  - Help to distinguish abstraction levels with subtle differences
- Make reasonings about the system easier
  - Models come from experiences of expert designers.
  - Modifying a subsystem will also become easier.
- Make communications easier
  - System design is a team work.
Considerations for Models

- No ambiguity and complete
  - Help to distinguish abstraction levels with subtle differences
- Make reasonings about the system easier
  - Models come from experiences of expert designers.
  - Modifying a subsystem will also become easier.
- Make communications easier
  - System design is a team work.
Examples

- Logic gates can actually be represented at three abstraction levels.
  - Register-transfer level (RTL)
    - Boolean expressions consisting of literals and logic operators
  - Netlist
    - Logic gates and interconnects
  - Standard-cell based designs
    - Placement of standard cells and routings of wires
- The above three models are also examples of a typical classification of models.
Examples

- Logic gates can actually be represented at three abstraction levels.
- Register-transfer level (RTL)
  - Boolean expressions consisting of literals and logic operators
- Netlist
  - Logic gates and interconnects
- Standard-cell based designs
  - Placement of standard cells and routings of wires
- The above three models are also examples of a typical classification of models.
Examples

- Logic gates can actually be represented at three abstraction levels.
- Register-transfer level (RTL)
  - Boolean expressions consisting of literals and logic operators
- Netlist
  - Logic gates and interconnects
- Standard-cell based designs
  - Placement of standard cells and routings of wires
- The above three models are also examples of a typical classification of models.
Examples

- Logic gates can actually be represented at three abstraction levels.
- Register-transfer level (RTL)
  - Boolean expressions consisting of literals and logic operators
- Netlist
  - Logic gates and interconnects
- Standard-cell based designs
  - Placement of standard cells and routings of wires
- The above three models are also examples of a typical classification of models.
Examples

- Logic gates can actually be represented at three abstraction levels.
- Register-transfer level (RTL)
  - Boolean expressions consisting of literals and logic operators
- Netlist
  - Logic gates and interconnects
- Standard-cell based designs
  - Placement of standard cells and routings of wires
- The above three models are also examples of a typical classification of models.
Typical Classification of Models

- **Behavioral model**
  - A blackbox with description of functionality, i.e. input/output relationship
  - Implementation, i.e. how to obtain output from input, is *not* specified

- **Structural model**
  - An implementation of interconnected components
  - Functionality is *not* specified explicitly

- **Physical model**
  - Specify the physical characteristics of components and interconnects
  - Dimensionality and placement

- From the perspective of models, modern ASIC design can be summarized as: RTL (behavioral) → Netlist (structural) → standard cells (physical)

- State-of-the-art software designs also heavily depend on models.
Typical Classification of Models

- **Behavioral model**
  - A blackbox with description of functionality, i.e. input/output relationship
  - Implementation, i.e. how to obtain output from input, is *not* specified

- **Structural model**
  - An implementation of interconnected components
  - Functionality is *not* specified explicitly

- **Physical model**
  - Specify the physical characteristics of components and interconnects
  - Dimensionality and placement

- From the perspective of models, modern ASIC design can be summarized as: RTL (behavioral) → Netlist (structural) → standard cells (physical)
- State-of-the-art software designs also heavily depend on models.
Typical Classification of Models

- **Behavioral model**
  - A blackbox with description of functionality, i.e. input/output relationship
  - Implementation, i.e. how to obtain output from input, is *not* specified

- **Structural model**
  - An implementation of interconnected components
  - Functionality is *not* specified explicitly

- **Physical model**
  - Specify the physical characteristics of components and interconnects
  - Dimensionality and placement

- From the perspective of models, modern ASIC design can be summarized as: RTL (behavioral) → Netlist (structural) → standard cells (physical)

- State-of-the-art software designs also heavily depend on models.
Typical Classification of Models

- **Behavioral model**
  - A blackbox with description of functionality, i.e. input/output relationship
  - Implementation, i.e. how to obtain output from input, is *not* specified

- **Structural model**
  - An implementation of interconnected components
  - Functionality is *not* specified explicitly

- **Physical model**
  - Specify the physical characteristics of components and interconnects
  - Dimensionality and placement

- From the perspective of models, modern ASIC design can be summarized as: RTL (behavioral) → Netlist (structural) → standard cells (physical)

- State-of-the-art software designs also heavily depend on models.
Typical Classification of Models

- **Behavioral model**
  - A blackbox with description of functionality, i.e. input/output relationship
  - Implementation, i.e. how to obtain output from input, is *not* specified

- **Structural model**
  - An implementation of interconnected components
  - Functionality is *not* specified explicitly

- **Physical model**
  - Specify the physical characteristics of components and interconnects
  - Dimensionality and placement

- From the perspective of models, modern ASIC design can be summarized as: RTL (behavioral) $\rightarrow$ Netlist (structural) $\rightarrow$ standard cells (physical)

- State-of-the-art software designs also heavily depend on models.
Other Examples of Models

- **Finite state machine**
  - Pretty much the synonym of RTL for hardware designs
- **Sequential program**
  - Supported by most programming languages
- **Dataflow**
  - Enable parallelism, e.g. MapReduce
- It is usually necessary to extend and to compose existing models to specify a complex system.
- It is usually more rewarding to reason complex functionalities with models instead of separated software and hardware implementations.
Other Examples of Models

- **Finite state machine**
  - Pretty much the synonym of RTL for hardware designs

- **Sequential program**
  - Supported by most programming languages

- **Dataflow**
  - Enable parallelism, e.g. MapReduce

- It is usually necessary to extend and to compose existing models to specify a complex system.

- It is usually more rewarding to reason complex functionalities with models instead of separated software and hardware implementations.
Other Examples of Models

- Finite state machine
  - Pretty much the synonym of RTL for hardware designs
- Sequential program
  - Supported by most programming languages
- Dataflow
  - Enable parallelism, e.g. MapReduce
- It is usually necessary to extend and to compose existing models to specify a complex system.
- It is usually more rewarding to reason complex functionalities with models instead of separated software and hardware implementations.
Other Examples of Models

- **Finite state machine**
  - Pretty much the synonym of RTL for hardware designs
- **Sequential program**
  - Supported by most programming languages
- **Dataflow**
  - Enable parallelism, e.g. MapReduce
- It is usually necessary to extend and to compose existing models to specify a complex system.
  - It is usually more rewarding to reason complex functionalities with models instead of separated software and hardware implementations.
Other Examples of Models

- Finite state machine
  - Pretty much the synonym of RTL for hardware designs
- Sequential program
  - Supported by most programming languages
- Dataflow
  - Enable parallelism, e.g. MapReduce
- It is usually necessary to extend and to compose existing models to specify a complex system.
- It is usually more rewarding to reason complex functionalities with models instead of separated software and hardware implementations.
From Models to System Specifications

- Models are somewhat *conceptual*
  - In designers’ mind
  - On pieces of scratching paper
- Models need to be captured for further processings
  - Especially for design automation tools, e.g. for synthesis and verification
- Specification languages
  - A formal way to capture models
  - A model can be captured in many different languages
  - A language can capture many different models
From Models to System Specifications

- Models are somewhat *conceptual*
  - In designers’ mind
  - On pieces of scratching paper
- Models need to be captured for further processings
  - Especially for design automation tools, e.g. for synthesis and verification
- Specification languages
  - A formal way to capture models
  - A model can be captured in many different languages
  - A language can capture many different models
From Models to System Specifications

- Models are somewhat *conceptual*
  - In designers’ mind
  - On pieces of scratching paper
- Models need to be captured for further processings
  - Especially for design automation tools, e.g. for synthesis and verification
- Specification languages
  - A formal way to capture models
  - A model can be captured in many different languages
  - A language can capture many different models
Natural Languages v.s. Formal Languages

- **Natural languages**
  - Ambiguous: even native speakers may have different explanations
  - Incomplete: cumbersome to elaborate all possible behaviors

- **Formal languages**
  - Based on math: everyone should understand
  - No ambiguity and complete
  - However, training is required to use formal languages effectively

- We will delay the discussions of languages to later lectures.
Natural Languages v.s. Formal Languages

- **Natural languages**
  - Ambiguous: even native speakers may have different explanations
  - Incomplete: cumbersome to elaborate all possible behaviors

- **Formal languages**
  - Based on math: everyone should understand
  - No ambiguity and complete
  - However, training is required to use formal languages effectively

- We will delay the discussions of languages to later lectures.
Natural Languages v.s. Formal Languages

- **Natural languages**
  - Ambiguous: even native speakers may have different explanations
  - Incomplete: cumbersome to elaborate all possible behaviors

- **Formal languages**
  - Based on math: everyone should understand
  - No ambiguity and complete
  - However, training is required to use formal languages effectively

- We will delay the discussions of languages to later lectures.
Outline

System Design Challenges

Models

An Example Design
A System for Summation

Let’s design a system to perform summation

What appears in your mind? An adder?
A System for Summation

Let’s design a system to perform summation

▶ What appears in your mind? An adder?
Mathematical model:
Input: $n$ numbers $a_1, a_2, \ldots, a_n$
Output: $\sum_{i=1}^{n} a_i$

More details are necessary to incorporate such model into a system
- What is $n$?
- What is the type of the numbers?
- What if overflow/underflow happens?

Assumptions
- 16 32-bit integers
- Ignore overflow/underflow

Now the model can be used for simulation without knowing anything about implementation.
Functional Specification

- **Mathematical model:**
  Input: \( n \) numbers \( a_1, a_2, \ldots, a_n \)
  Output: \( \sum_{i=1}^{n} a_i \)

- More details are necessary to incorporate such model into a system
  - What is \( n \)?
  - What is the type of the numbers?
  - What if overflow/underflow happens?

- **Assumptions**
  - 16 32-bit integers
  - Ignore overflow/underflow

- Now the model can be used for simulation without knowing anything about implementation.
Functional Specification

- Mathematical model:
  Input: $n$ numbers $a_1, a_2, \ldots, a_n$
  Output: $\sum_{i=1}^{n} a_i$

- More details are necessary to incorporate such model into a system
  - What is $n$?
  - What is the type of the numbers?
  - What if overflow/underflow happens?

- Assumptions
  - 16 32-bit integers
  - Ignore overflow/underflow

- Now the model can be used for simulation without knowing anything about implementation.
Functional Specification

- **Mathematical model:**
  Input: $n$ numbers $a_1, a_2, \ldots, a_n$
  Output: $\sum_{i=1}^{n} a_i$

- More details are necessary to incorporate such model into a system
  - What is $n$?
  - What is the type of the numbers?
  - What if overflow/underflow happens?

- **Assumptions**
  - 16 32-bit integers
  - Ignore overflow/underflow

- Now the model can be used for simulation without knowing anything about implementation.
Design Constraints

- Latency: complete a summation in 8ns
- Throughput: complete 1,000,000,000 summations per second
Design Constraints

- Latency: complete a summation in 8ns
- Throughput: complete 1,000,000,000 summations per second
Rough HW/SW Partitioning

- **Hardware**
  - Need at least one two-input adder

- **Software**
  - Coordinate hardware to complete summation by adding two numbers a time
  - If a higher precision is required later, software can be updated
Rough HW/SW Partitioning

- **Hardware**
  - Need at least one two-input adder

- **Software**
  - Coordinate hardware to complete summation by adding two numbers a time
  - If a higher precision is required later, software can be updated
Assume adders that can add two 32-bit integers in 1ns are available

- Sequential program
  - Accumulator: 1 adder and 1 32-bit register
  - Smallest size, but won’t meet the latency constraint

- Dataflow
  - 15 adders connected in series
  - 15 adders connected into a tree
  - Will they meet the latency constraint?
  - How many sets of adders are necessary for the throughput constraint?
Assume adders that can add two 32-bit integers in 1ns are available

Sequential program
  - Accumulator: 1 adder and 1 32-bit register
  - Smallest size, but won’t meet the latency constraint

Dataflow
  - 15 adders connected in series
  - 15 adders connected into a tree
  - Will they meet the latency constraint?
  - How many sets of adders are necessary for the throughput constraint?
Assume adders that can add two 32-bit integers in 1ns are available

Sequential program
  ▶ Accumulator: 1 adder and 1 32-bit register
  ▶ Smallest size, but won’t meet the latency constraint

Dataflow
  ▶ 15 adders connected in series
  ▶ 15 adders connected into a tree
  ▶ Will they meet the latency constraint?
  ▶ How many sets of adders are necessary for the throughput constraint?
More on Design Space Exploration

- Is it possible to use less adders to
  - Complete a summation in 8ns?
  - Complete 1,000,000,000 summations per second?
- What if adders with other characteristics are available?
- Note that until now, we haven’t talked about any specific adder design, e.g. carry-ripple and carry-lookahead.
More on Design Space Exploration

- Is it possible to use less adders to
  - Complete a summation in 8ns?
  - Complete 1,000,000,000 summations per second?

- What if adders with other characteristics are available?
- Note that until now, we haven’t talked about any specific adder design, e.g. carry-ripple and carry-lookahead.
More on Design Space Exploration

- Is it possible to use less adders to
  - Complete a summation in 8ns?
  - Complete 1,000,000,000 summations per second?

- What if adders with other characteristics are available?
  - Note that until now, we haven’t talked about any specific adder design, e.g. carry-ripple and carry-lookahead.
More on Design Space Exploration

- Is it possible to use less adders to
  - Complete a summation in 8ns?
  - Complete 1,000,000,000 summations per second?
- What if adders with other characteristics are available?
- Note that until now, we haven’t talked about any specific adder design, e.g. carry-ripple and carry-lookahead.
Summary

- Models define abstraction levels.
- Choose proper models increases designer’s productivity