This part of the course will be given in 6+1 weeks. There will be several labs that familiarize the usage of HDL simulators such as ModelSim, the Mentor Graphics Seamless Coverification Environment, and other Cadence synthesis/simulation tools. We will have one term project based on the Digital Camera SoC found in Chapter 7 of Frank Vahid’s book on Embedded System Design.

Outline

- Introduction to Hardware-Software Codesign
- System Modeling, Architectures, Languages
- System Partitioning
- Co-synthesis Techniques
- Function-Architecture Codesign Paradigm
- Case Study
  - ATM Virtual Private Network Server

The outline of this part will be given as follows: First, we’ll brief introduce what HW-SW codesign is. Next, we’ll talk about the concepts of the models, architectures and languages which are mainly used to specify the system we want to design. Then, we’ll talk about how to partition the system and decide which part will be run in SW and what will be implemented as ASIC. After partitioning, SW and HW part of the system has to be developed, and we’ll present some co-synthesis technique which can accelerate the development of HW and SW and their integration. In order to give an overall view of the codesign process, we’ll introduce one of very famous academic codesign flow built in UC Berkeley called “function-architecture codesign”. In the end, we will present two case studies: ATM VPN (from CODES’2001) and Digital Camera SoC (from Vahid’s Chapter 7).
Classical system design process partitions system requirements into hardware and software, which are developed separately by different teams. Their designs are integrated only at the end of the design process. When hardware faults are discovered, software must try to compensate for the hardware faults because hardware is difficult to change (compared to the ease of software changes). In addition, since HW and SW are developed separately, their effect of their interaction and integration cannot be explored until the last phase. This often results in suboptimal designs, costly modifications, delay to market, schedule slippages, etc. The main problem here is that faults are found TOO late!

### Classic Hardware/Software Design Process

- **Basic features of current process:**
  - System immediately partitioned into hardware and software components
  - Hardware and software developed separately
- **Implications of these features:**
  - HW/SW trade-offs restricted
    - Impact of HW and SW on each other cannot be assessed easily
  - Late system integration
- **Consequences of these features:**
  - Poor quality designs
  - Costly modifications
  - Schedule slippages

Codesign Definition and Key Concepts

- **Co-design**
  - 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

The common definitions for HW/SW codesign are presented above. The two key concepts involved in codesign are concurrent development of HW and SW, and integrated design. Integrated design allows interaction between the design of HW and SW and help to make trade-offs between HW and SW. Codesign techniques using these two key concepts take advantage of design flexibility to create systems that can meet stringent performance requirements with a shorter design cycle.

Since the main goal of codesign is to accelerate the system design which consists of hardware and software, any technique used to help shortening the design time can be categorized as codesign issue. In general, it is preferred that the clear separation point between of HW and SW through the whole design process can be deferred as late as possible.
Motivations for Codesign

- Co-design helps meet time-to-market because developed software can be verified much earlier.
- Co-design improves overall system performance, reliability, and cost effectiveness because defects found in hardware can be corrected before tape-out.
- Co-design benefits the design of embedded systems and SoCs, which need HW/SW tailored for a particular application.
  - Faster integration: reduced design time and cost
  - Better integration: lower cost and better performance
  - Verified integration: lesser errors and re-spins

Codesign has the following advantages:

(1) Shorter time to market, because software debug can be performed much earlier
(2) More optimal designs, because defects can be found and repaired earlier without having to compensate for others’ mistakes
(3) Faster, better, verified integration, because of better design space explorations.

Co-design is especially important for the design of embedded systems or SOC because not only most of them require both the HW and SW in nature, but also they are often designed under some cost and performance constraints which codesign can help to meet.

Driving Factors for Codesign

- Reusable Components
  - Instruction Set Processors
  - Embedded Software Components
  - Silicon Intellectual Properties
- Hardware-software trade-offs more feasible
  - Reconfigurable hardware (FPGA, CPLD)
  - Configurable processors (Tensilica, ARM, etc.)
- Transaction-Level Design and Verification
  - Peripheral and Bus Transactors (Bus Interface Models)
  - Transaction-level synthesis and verification tools
- Multi-million gate capacity in a single chip
- Software-rich chip systems
- Growing functional complexity
- Advances in computer-aided tools and technologies
  - Efficient C compilers for embedded processors
  - Efficient hardware synthesis capability

There are several driving factors for the codesign methodology as listed in the slide. Mainly, systems are getting more and more complex in terms of either functionality or the gate counts, so it is no longer designed from scratch, instead it is now more of an integration of reusable components, such as hardware IP and software COTS. Further, due to FPGA, hardware has now become more and more flexible just as the software of the old days. All these flexibilities have resulted in the need for a better methodology with a wider coverage and lesser design faults such as the codesign technique. Especially many CAD tools and techniques have been making good progress, mature and efficient tools for either software generation through the compiler and hardware generation through some high level synthesis techniques are available nowadays. How to move forward the overall system design considering both HW and SW together has become a very challenging and interesting topics.
Categories of Codesign Problems

- Codesign of embedded systems
  - Usually consist of sensors, controller, and actuators
  - Are reactive systems
  - Usually have real-time constraints
  - Usually have dependability constraints
- Codesign of ISAs
  - Application-specific instruction set processors (ASIPs)
  - Compiler and hardware optimization and trade-offs
- Codesign of Reconfigurable Systems
  - Systems that can be personalized after manufacture for a specific application
  - Reconfiguration can be accomplished before execution or concurrent with execution (called evolvable systems)

Listed are some of the ways in which the codesign methodology can be applied to different kinds of systems. The first category of system is about the embedded system which will be mainly what we’ll focus in this cost. Their system consists of microcontroller and some hardware components which involve SW and HW issues. Besides they are often subject to real-time constraints. What the popular SOC targets is just various embedded system applications. The other category of system codesign often apply is about the application-specific instruction set processors. This kind of processors doesn’t have fixed architecture and instruction sets. Instead, according to the application requirement, specific instruction sets CPU can be developed. Since the instruction sets are not fixed, how to generate the software and estimate the performance has become very important. The third category of system listed here is about some reconfigurable system. These systems provide some reconfiguration capability which can be configured for different applications. How to generate the software to configure these system will be one of the codesign issues.

Typical Codesign Process

A typical codesign process starts with the system description in some kind of executable language such as SystemC or a combination of HDLs (hardware description languages) Verilog/VHDL and HLLs (high level languages) C/C++/Java. Partitioning is then performed on the functional specification by using profiling tools or some other estimation techniques that tell a designer what percentage of time is spent in executing which functions and where are the major bottlenecks or critical path of the system. Except for some regular systems such as DSP, hw/sw partitioning is often not easily automated and needs human intervention due to the variety of functional applications. In addition, the search of optimal hw/sw partitioning solution is NP-hard problem so some heuristic algorithms are required in practice. Once a decision is made as to what functions are to be implemented in hardware and what in software, they are designed simultaneously by different design teams, but they must use a common medium of verification such as a cosimulation environment in which system errors that result from hardware and software interactions can be detected and fixed. The interface between HW/SW has also to be implemented. Finally, the whole system is integrated at a lower instruction set level. If the current partition of HW/SW can not meet the constraints of the system. Another iteration of the design partition can be tried. Different partitionings can be synthesized to perform a guided exploration of the design space.
Some details of what codesign process involves are listed in this slide. (The main concepts were discussed in the previous slide.) The first step in designing a system is specifying its functionality which can be described by suitable languages. However, the feature of the system functionality can be best captured by constructing the appropriate abstraction model for the system. Next step of the design has to map the model of the system to some underlying architecture. Here architecture describes what style of the target system will be implemented. The mapping of the functionality to architecture has to be decided by partitioning. To proceed the partitioning, we have to decide what the object of the partitioning is, and partitioning strategies. Besides, some estimation methods have to be applied to evaluate the partitioning results. After some partitioning decisions have been made, the HW and SW part of the system has to be generated respectively. The details of HW and SW synthesis will involve the scheduling of operation or instructions, the generation of the interface between HW and SW, etc. The partitioning result of HW/SW has also to be back annotated to refine the original specification.

There are several requirements for an ideal codesign environment, including a specification language that is not biased towards either hardware or software, such as using SystemC, UML, or some other more generic modeling technique such as Petri nets, dataflow diagrams, etc. Further, the codesign environment should be able to allow a designer to iteratively change the hardware software partitions for efficient design space exploration. Hardware and software must be cosimulated and coveredified in a common modeling substrate such as Mentor Graphics Seamless CVE (coverification environment).
We’ll go to the details of the codesign process by first introducing the concepts of models, architectures and the languages which are used to describe the system.

Models are specifications and architectures are implementations of the specification through a refinement process. Languages help in this design process by making things executable, simulatable, verifiable, and thus less error-prone.

The system design process starts from the specification of the system functionality with some given design constraints and then follow the design flow and go through a series of design process to achieve final implementation of the system. Here the starting point of the system design is the system specification, and models provides just some conceptual views of the system’s functionality at different design phases. Models can be defined as a formal system consisting of objects and composition rules, and is used for describing a system’s characteristics. Typically we can use a particular model to decompose a system in to pieces, and then generate a specification by describing theses pieces in a particular language. A language can capture many different models, and a model can be captured in many different languages. Designers can choose different models during different design phases to emphasize the aspects of the systems. For example, during the specification phase, the designer knows nothing beyond the functionality of the system, so he tends to use a model that won’t reflect any implementation details while during the implementation phase, the designer can switch to a model that can capture the system’s structure.

Models can only describe exactly how the system work but don’t tell how the system is to be implemented. Therefore, the model has to be finally mapped to some architecture which serves to define the model’s implementation by specifying the hardware components as well as the connection between them. In other words, architectures are abstract views of the system’s implementation.

Models and architecture are conceptual and implementation views on the highest level of abstraction. The design process is the set of design tasks that transforms a model into an architecture.
One examples of model, architecture, and languages can be illustrated from an elevator controller example. Here the target elevator system can be describe by English as shown in figure (a). The natural language such as English is not a good specification language to specify the system as it is ambiguous and not executable. Figure (b) and (c) shows two different models of the elevator controller. Fig (b) represents the controller as a set of programming statements while Fig (c) represents the controller as a state machine. These two models all consists of a set of objects and the interactions among them. The state machine model consists of a set of states and transitions of states which is best suited to represent a system’s temporal behavior as it allows a designer to explicitly express the modes and mode transitions caused by some external or internal events. The algorithm models, consists of a set of statements that are executed under a control sequence that uses branching and looping. Although this model cannot represent the concept of explicit states, it can specify a system’s input-output relationship in terms of a sequence of statements, and best suited to represent the procedural view of the system.

As described before, the models of the system will be finally mapped to some underlying architectures for implementation. This slides shows two alternative architectures at different abstraction levels for the elevator controller system. Figure (a) is a register-level implementation which use state register to hold the current state, and the combinational logic to implement state transitions and values of output signals. Figure (b) the system is implemented by a microcontroller where a variable in a program is used to represent the current state and the program to calculate the state transitions and values of output signals. In this architecture, the program is stored in the memory and executed by the processor.

These two architectures represent total two different implementation approaches. By selecting different target system architecture at the beginning of the design phase, it will surely affect the following design processes.
There are numerous methods that are candidates to be used for a unified representation. These models can be classified as five distinct categories. Most all of them have been tried in one codesign system or another with varying levels of success. Typically the methods are more suited to systems of a certain type, e.g., data flow diagrams are more suited to data driven applications like Digital Signal Processing (DSP) systems.

A state-oriented model represents the system as a set of states and a set of transition between them which are triggered by external events. This model is most suitable for control systems where the system’s temporal behavior is the most important aspect of the design.

An activity-oriented model describes the system as a set of activities related by data or execution dependencies. This model is suitable for some transformation systems such as DSP systems.

A structure-oriented model is usually used to describe a system’s physical modules and interconnections between them. This model is usually used for representing the system’s architecture.

A data-oriented model emphasizes the data structure used in the system. It represent the system as a collection of data related by their attributed, class membership, etc. This model would be most suitable for information systems such as database where the function of the system is less important than the data organization.

HW/SW System Models

- State-Oriented Models
  - Finite-State Machines (FSM), Petri-Nets (PN), Hierarchical Concurrent FSM
- Activity-Oriented Models
  - Data Flow Graph, Flow-Chart
- Structure-Oriented Models
  - Block Diagram, RT netlist, Gate netlist
- Data-Oriented Models
  - Entity-Relationship Diagram, Jackson’s Diagram
- Heterogeneous Models
  - UML (OO), CDFG, PSM, Queueing Model, Programming Language Paradigm, Structure Chart

A finite-state machine is an example of a state-oriented model and also the most popular model for describing control systems since the temporal behavior of such systems is most naturally represented in the forms of states and transitions between states. The transition between states are triggered by some external events such as clock, primary inputs, etc. There are general two forms of the FSM which distinguish in how to associate with the output of the system. In (transition-based) Mealy FSM, the outputs of the system depend on both the state and the input value; while in (state-based) Moore FSM, the outputs of the system depends only on the states. This slide represents the same function of the elevator controller in the state-based Moore FSM model. In principal, the Moore and Mealy state machines can be translated from each other. However, the Moore state model will require more states than the Mealy model in the representation for the same functionality.

In addition to the simple Mealy and Moore FSM, there is another important extended FSM called FSM with a datapath where the events influencing the transitions between the states now can be represented as a set of status expressions as logic relations. Besides, the output values can also be described as the expression of some equations. The salient feature of FSMD is that it can be suitable for both control and computation dominated systems.

The merits and demerits of the FSM can be summarized as follows:

**Merits**

- **Represent system’s temporal behavior explicitly**
- **Suitable for control-dominated systems**
The conventional FSM is not suitable for describing complex systems as it cannot explicitly support concurrency and hierarchy. Due to this, the FSM representation may lead to the problem of state explosions for complex systems. This slide shows some various versions of FSM called codesign finite state machine, one of the GALS model. Here this example consists of three local synchronous FSM which are running concurrently. The interaction between these three FSM are through some asynchronous communication channel. The communication channel will act like the external events for the local FSM.

Codesign Finite State Machine (CFSM)

- Globally Asynchronous, Locally Synchronous (GALS) model

The Petri net model is another type of state-oriented model defined to model systems that comprise interacting concurrent tasks. The Petri net consists of a set of places, a set of transitions, and a set of tokens. Tokens reside in places and circulate through the Petri net by being consumed and produced whenever a transition fires. A transition can fire only if each of its input places has at least one token. When a transition fires, all of its enabling tokens from input places will be removed, and deposited one token into each output place. This slide shows several system characteristics that Petri Nets can model. Fig (a) shows the model of “sequencing” where transition t1 fires after t2. Fig (b) shows the modeling of “non-deterministic branching” as only one of t1 or t2 will fire. Fig (c) models the “synchronization” as t1 can fire only after both input places have tokens. Fig (d) represents the concept of “resource contention” where t1 and t2 compete for the same token which resides in the place of the center. Fig (e) models the concept of “concurrency” as some transition like t1 and t4 can fire at the same time.

In addition to being able to represent several key system characteristics, Petri net models can be used to check and validate certain useful system properties such as safeness and liveness. Safeness guarantees that the number of tokens in the net won’t grow indefinitely. Liveness guarantees a dead-lock will not happen in the system.

The merits and demerits of the Petri nets can be summarized as follows:

**Merits**

- Good at modeling and analyzing concurrent systems
- Extensive theoretical and experimental works

**Demerits**

- Complex to understand and use
- Space and time complexity can be high
So far we have introduced several models which are basically suitable to describe the system state changes in response to some external events. These state transition characteristic is inherent in most of the control systems. However, for some transformational systems where the outputs are determined by a set of computations on the system’s input, they are hard to capture using state-oriented model. In fact, they are captured by a so called “data flow graph” model. DFG consists of a set of nodes and a set of edges. There are various types of nodes including input, and output or the activity node which represent the activities of computation that transform or manipulate data. The nodes are connected by directed edges which represents the data dependence between the nodes. For the example shown in the slide, this system consists of two activities A1, and A2, the latter can be further decomposed into A21, A22, and A23. Here after A1 has been computed and the data Y is produced, the activity of A2 can start.

The merits and demerits of the DFG can be summarized as follows:

**Merits**
- Support hierarchy
- Suitable for specifying complex transformational systems
- Represent problem-inherent data dependencies

**Demerits**
- Do not express control sequencing or temporal behaviors
- Weak for modeling embedded systems

Since most of the systems exhibit several characteristics, single model representation very often cannot capture the system very well. Therefore, some traditional models can be modified or combined as a heterogeneous model. This figure presents one of that example in so called control/data flow graphs (CDFG) which is able to explicitly show both the data dependence as well as the control sequence of a system. In the example shown in this slide, the result of (5+X) is scheduled to be computed at control step 1, and its result will be used in another addition node scheduled in step 2. This mode is used in many high level specification tools such as Ptolemy, SES Workbench, etc.
Object-Oriented Paradigms (UML, ...)  

- Use techniques previously applied to software to manage complexity and change in hardware modeling  
- Use OO concepts such as  
  - Data abstraction  
  - Information hiding  
  - Inheritance  
- Use building block approach to gain OO benefits  
  - Higher component reuse  
  - Lower design cost  
  - Faster system design process  
  - Increased reliability  

As the system becomes more and more complex, the conventional models discussed in previous slides become somehow hard to capture complex system functionality. Therefore, some modeling techniques previously used in software engineering have now been applied in hardware modeling. One of such key modeling techniques is so called the object-oriented paradigms. The salient features of object-oriented representation is that it can support data abstraction, information hiding and inheritance. The detailed manipulation process of data will be hided from outside. Each OO building block interact with each other through abstract channel. In SOC design, the concept of IP (intellectual property) or component reuse has become a very popular practice. Through the OO modeling, the feature of component reuse can be easily captured.

In object-oriented representation, each OO block can support any level of abstraction. This is a simple example of three different levels of hardware abstraction that can be described in an OO representation.

[Kumar95]
This slide shows the application of UML on the overall design phase of the SOC design.

Characteristics of conceptual models

- Concurrency
  - Data-driven concurrency
  - Control-driven concurrency
- State transitions
- Hierarchy
  - Structural hierarchy
  - Behavior hierarchy
- Programming constructs
- Behavior completion
- Communication
- Synchronization
- Exceptional handling
- Non-determinism
- Timing

The specification languages are used to specify the system characteristics. From the description of the conceptual models, it can be seen that different conceptual models possess different characteristics. This slide presents some of the characteristics most commonly found in conceptual models. In general, it will be preferred that more characteristics can be captured in a single specification language.

For the design of the embedded system, the specification requirements include state transition, behavior hierarchy, concurrency, exception, programming constructs, and behavior completion.
So far, we have introduced several models and how they can be used to describe a system's functionality, data, control and structure. An architecture is intended to supplement these descriptive models, specifying how the system will be actually implemented. Architectures can range from simple controllers to massively parallel processors. Depending on different target architectures, the mapping from the model to the architecture implementation may require different design process. A system behavioral description should be unbiased and implementation independent, thus it will allow the maximum flexibility in system architecture exploration and optimization.

This shows what issues we will face during the implementation of the typical DSP application. The architecture we use is based on some common bused interconnect by some IP. IP can be regarded as pre-verified hardware library blocks which are designed for the intention of reuse in several system. Through the reuse of the IP, the design productivity of the system can be highly increased.
This slide shows a typical AMBA-based SoC architecture which is a very common platform for SOC implementation. This platform consists of two buses: AHB (Advanced High Performance Bus) and APB (Advanced Peripheral Bus) interconnected by a bridge which acts as a slave on the AHB side and a master on the APB side. The microprocessor and other high performance cores are attached to the AHB bus and other slower peripheral devices are attached as slaved on the APB bus.

Conceptual models we have mentioned before are used to understand, organize and define a system’s functionality. To further capture these models in a concrete form, system specification languages will be required. Especially in the earliest design phase, it is preferred that the system’s conceptual view can be described in terms of an executable specification language which is capable of capturing the system functionality in a machine-readable and simulatable form. Such an approach has several advantages. First, simulating such a specification allows the designer to verify the correctness of the system’s intended functionality. Secondly, the executable specification can serve as an input to synthesis tools. Third, this specification can also be a comprehensive documentation.

This slides lists several popular specification languages which specify the system from different aspects. Depending on the application, one of more languages may be used in the same design of the system. Among the language list, hardware description language and software programming languages should be already addressed in some other courses. The system specification language is also introduced before. Here the architecture description and verification language will be discussed.
Here are some of the current state-of-art hardware description languages. Verilog is used on a large scale in Taiwan. VHDL is the more preferable language in Europe. In USA, both VHDL and Verilog are used on an equal basis. VHDL is more typed than Verilog. SystemVerilog is still under development and standardization.

SystemVerilog 3.0 has several enhancements compared to Verilog 2.0. There are several more built-in C types allowing designers more flexibility in writing HDL code. Mismatches between simulation and synthesis result in unsyntesizable designs. SystemVerilog tries to solve these mismatches. Assertions that help design verification and embedding of designer intents into a system at the code level have been added to SystemVerilog. There are several more enhanced communication interfaces.
SystemVerilog 3.1 includes more testing and verification mechanisms such as testbench data structures, classes, inter-process communication and randomization and temporal assertions.

This diagram shows the SystemVerilog environment that was used to design an Ethernet MAC system. The blue assertions and the green testbenches have been made possible by SystemVerilog.
**Architecture Description**

- Objectives for Embedded SOC
  - Support automated SW toolkit generation
    - exploration quality SW tools (performance estimator, profiler, ...)
    - production quality SW tools (cycle-accurate simulator, memory-aware compiler...)
  - Specify a variety of architecture classes (VLIWs, DSP, RISC, ASIPs, ...)
  - Specify novel memory organizations
  - Specify pipelining and resource constraints

Architecture description languages are intended to describe a set of various architectural alternatives available for system design. It may specify different architecture class such as DSP, RISC etc or different memory organization. It may also specify different pipelining and resource constraints. Hopefully through the different specification of architectures, it can also aid for the estimation and generation of the SW.

This figure shows how architectures can be described using an ADL (architecture description language) and then used throughout of the SoC hardware-software codesign process. Here the IP can be described using ADL which can support some information for the performance estimation, or the SW compilation and the HW/SW cosimulation.
**Property Specification Language (PSL)**

- Accellera: a non-profit organization for standardization of design & verification languages
- PSL = IBM Sugar + Verplex OVL
- System Properties
  - Temporal Logic for Formal Verification
- Design Assertions
  - Procedural (like SystemVerilog assertions)
  - Declarative (like OVL assertion monitors)
- For:
  - Simulation-based Verification
  - Static Formal Verification
  - Dynamic Formal Verification

Property Specification Language is a language standard which aids for the verification of the system. Based on this standard language, some formal verification techniques can be applied to provide another verification approach.

**System Partitioning**

After introducing several key issues in developing a functional system specification including model, architecture and languages, we will now turn our focus on the system design itself. In the system design, one of the most challenging and important tasks is finding a way to partition the system’s functionality among various system components so that all of the design constraints can be satisfied. We’ll address the general partitioning issues while the HW/SW partitioning can be regarded as a special case.
For the system or functional partitioning approach, we first decompose the system's functionality into non-divisible pieces called functional objects. Those objects can be partitioned among system components, after which we can implement each component's functionality either as hardware or as software.

It is analogous to Structural Partitioning in which the structure of a system is refined into lower level hardware components. However, in Structural Partitioning, functionality cannot be moved from hardware to software.

[Gajski94].
**Granularity Issues in Partitioning**

- The granularity of the decomposition is a measure of the size of the specification in each object.
- The specification is first decomposed into functional objects, which are then partitioned among system components.
  - Coarse granularity means that each object contains a large amount of the specification.
  - Fine granularity means that each object contains only a small amount of the specification.
    - Many more objects
    - More possible partitions
    - Better optimizations can be achieved

For functioning partitioning, the system function is first decomposed into several functional objects. The granularity is a measure of the size of each object. Coarse granularity means that each object contains a large amount of the specification while the fine-granularity contains less amount. Although finer granularity may potentially lead to better result, there are several drawbacks it may suffer. First, since fine granularity will produce more objects, partitioning must use more computation time or it will yield poor results. Second manual interaction is more difficult to support. Besides, the partitioning output is less comprehensible to humans. Finally, estimation is more difficult. Therefore a reasonable partitioning granularity should be considered in practice. Note that it is the system functionality that is being partitioned here in order to achieve a better allocation and assignment to hardware or software. That is, a number of objects in a partition defined here may be assigned to the same hardware or software later.

**System Component Allocation**

- The process of choosing system component types from among those allowed, and selecting a number of each to use in a given design.
- The set of selected components is called an allocation.
  - Various allocations can be used to implement a specification, each differing primarily in monetary cost and performance.
  - Allocation is typically done manually or in conjunction with a partitioning algorithm.
- A partitioning technique must designate the types of system components to which functional objects can be mapped.
  - ASICs, memories, etc.

An integral part of the partitioning problem is allocating portions of the system to actual components for their implementation. Obviously, the system “tasks” must be allocated to components that are capable of performing them. The set of selected components is called an allocation. Various allocations may be able to implement a given specification, each differ in cost and performance. Allocating is typically either done manually, or performed in conjunction with a partitioning algorithm.

[Gajski94].
**Metrics and Estimations Issues**

- A technique must define the attributes of a partition that determine its quality
  - Such attributes are called metrics
    - Examples include monetary cost, execution time, communication bit-rates, power consumption, area, pins, testability, reliability, program size, data size, and memory size
  - Closeness metrics are used to predict the benefit of grouping any two objects
- Need to compute a metric’s value
  - Because all metrics are defined in terms of the structure (or software) that implements the functional objects, it is difficult to compute costs as no such implementation exists during partitioning

A technique must define the attributes of a partition that determine if the partition is good or bad. Such attributes are called metrics which could be the area, execution time, power consumption, pins and reliability. Metrics must be defined to determine a partition’s relative cost vs. other potential partitionings. Obviously, some metrics, such as execution time of a given task on a specific processors, are impossible to measure precisely until a final implementation is made. Therefore, accurate, fast "cost" estimation is mandatory for a good partitioning algorithm.

In addition to those metrics which evaluate a complete partition, there is also another type of metric called closeness metric. Such metric is used to predict the benefit of grouping two functional objects. For example, if two objects which share many same variables, their closeness could be higher.

[Gajski94].

**Computation of Metrics**

- Two approaches to computing metrics
  - Creating a detailed implementation
    - Produces accurate metric values
    - Impractical as it requires too much time
  - Creating a rough implementation
    - Includes the major register transfer components of a design
    - Skips details such as precise routing or optimized logic, which require much design time
    - Determining metric values from a rough implementation is called estimation

There are two basic approaches to computing metrics. The first is to create a detailed implementation and directly measure the metrics of interest. The second is to estimate the given metric from the abstract system model in use at the time. Although the first approach can provide very accurate metric, it is nearly impractical because it takes too much time to finish an implementation not mentioning that there are usually many candidate partitions existing. Therefore, second approach is usually preferred that relies some algorithms to estimate the metric of partitions in very short period.

[Gajski94].
Estimation of Partitioning Metrics

- Deterministic estimation techniques
  - Can be used only with a fully specified model with all data dependencies removed and all component costs known
  - Result in very good partitions
- Statistical estimation techniques
  - Used when the model is not fully specified
  - Based on the analysis of similar systems and certain design parameters
- Profiling techniques
  - Examine control flow and data flow within an architecture to determine computationally expensive parts which are better realized in hardware

Metrics must be used to guide the partitioning process. The type of metrics used depends a great deal on the level of description of the system.

How to estimate the metric will largely depend on what metric itself is evaluated. However, the estimation techniques can be classified into three categories. The deterministic techniques provide good estimation but the fully specified model should be built in advance. While this technique produces accurate metric values, it is impractical, because it requires too much time. A fully specified model may require days or months to create.

The statistical estimation method can be used based on the analysis of similar system such that the model may not be fully specified.

[DeMicheli93].

Objective and Closeness Functions

- Multiple metrics, such as cost, power, and performance are weighed against one another
  - An expression combining multiple metric values into a single value that defines the quality of a partition is called an Objective Function
  - The value returned by such a function is called cost
  - Because many metrics may be of varying importance, a weighted sum objective function is used
    - e.g., Objfct = k1 * area + k2 * delay + k3 * power (equation 1)
  - Because constraints always exist on each design, they must be taken into account
    - e.g Objfct = k1 * F(area, area_constr) + k2 * F(delay, delay_constr) + k3 * F(power, power_constr)

Since partitioning evaluation is not evaluate based on a single metric, multiple metrics may be considered simultaneously to determine the overall evaluation result of the partition. Therefore, the object functions base on varying weights for area, timing, and power constraints may be used to reflect their relative importance in each different system being designed. For example, the following objective function is a weight sun function in which three metric values, area, delay, and power are weighted by constant k1, k2, and k3 respectively, and then summed. Giving k1, a large value than k2, and k3 makes area the most important metric. Since most design decision are driven by constraints, simple function such as equation 1 are rarely used. Constraints must be incorporated into the function so that partitions that meet constraints are considered better than those that don’t. For example, we can extend the above objective function as follows. Where F is a function indicating how the metric’s estimate is to the given constraint. A common form of F returns the amount by which the metric’s estimate violates the constraint, returning zero when there is no violation. This form of F caused the objective function to return zero when a partition meets all constraints, making the goal of partition to obtain a cost of zero.

Closeness function, similar to the objection function, consists of the expression of several metrics. However, closeness function is usually used to guide the partitioning during the partitioning process. While an objective function combines metrics to evaluate a partition, a closeness function combines closeness metrics to indicate the desirability of grouping objects, before a complete partition exists.

[Gajski94].
**Partitioning Algorithm Classes**

- **Constructive 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 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

A partitioning algorithm can be classified into two general categories. One is a constructive algorithm which starts from decomposed functional objects. The partitioning is built and constructed by organizing or grouping these functional objects. How to group the objects usually depends on the defined closeness function. The computation time in a constructive algorithm is spent constructing a small number of partitions.

Another class of algorithm belongs to iterative algorithm which will iteratively modify the initial partitioning to in the hope to achieve the better result. Such algorithm use an objective function to evaluate each partition, which yields more accurate evaluations than closeness functions used by constructive algorithms.

In practice, a combination of constructive and iterative algorithms is often employed. Basic partition algorithms are described later.

**Iterative Partitioning Algorithms**

- The computation time in an iterative algorithm is spent evaluating large numbers of partitions
- Iterative algorithms differ from one another primarily in the ways in which they modify the partition and in which they accept or reject bad modifications
- The goal is to find global minimum while performing as little computation as possible

The concept of a local minimums is demonstrated in the following graph, which shows a sequence of moves and the cost after each move. A move is a partition modification that remaps an object from one group to another. The leftmost point represents the cost of an initial partition. After several moves, the cost decreases to A. But the next move increases the cost. A is referred to as a local minimum. B is also a local minimum. C represents a solution which would consume fewer resources (in HW and SW) than A or B, yet it might be very difficult to find, and solution A may be "good enough." An algorithm that accepts only moves that decrease cost is call a greedy algorithm. Such algorithms cannot escape local minimums. An algorithm that can escape a local minimums is often called hill-climbing algorithm.

Multivariate optimization is a much-studied problem in CS. The interested reader should investigate this area to understand the problems inherent in optimal partitioning.

[Gajski94].
**Basic Partitioning Algorithms**

- **Constructive Algorithm**
  - Random mapping
    - Only used for the creation of the initial partition.
  - Clustering and multi-stage clustering
  - Ratio cut

- **Iterative Algorithm**
  - Group migration
  - Simulated annealing
  - Genetic evolution
  - Integer linear programming

A partition algorithm maps each function object to exactly one group, where each group represents a system component. Ideally, the partition that is produced by the algorithm is the one which yields the minimal cost, as computed by the selected objective function.

Several common partitioning algorithms are listed here. The most trivial algorithm is Random mapping which can be used to generate initial partitioning. In this mapping, where each object is randomly assigned to one of the given components. The computational complexity of the algorithm is \(O(n)\), where \(n\) is the number of objects. The remaining algorithms will be discussed in the subsequent slides.

**Hierarchical Clustering**

- One of constructive algorithm based on closeness metrics to group objects

- **Fundamental steps:**
  - Groups closest objects
  - Recompute closenesses
  - Repeat until termination condition met

- **Cluster tree maintains history of merges**
  - Cutline across the tree defines a partition

Hierarchical clustering belongs to the class of constructive algorithm. This algorithm starts with a initial set of functional objects. The initial set can be created using random mapping algorithm. Hierarchical clustering will try to iteratively group the closest objects together. The closeness of the object will be judged by the closeness function.

Closeness metrics are intended to yield a partition with good global metric values. The approach groups the closest objects, recomputes closenesses after the grouping, and repeats until some termination condition is met. The complexity of the algorithm is dominated by the computation of closeness between all object pairs, which is \(O(n^2)\).

After one iteration of grouping, the closeness function will be recomputed so next stage of functional grouping can start.

For example, Figure (a) shows four objects – o1, o2, o3 and o4 – and their closeness values. The two closest objects are o1 and o2, with a closeness value of 30. In Figure (b), we merge o1 and o2, and approximate the closeness values between the new object and o3 and o4 as a average of previous closeness values. In Figure (c), we again merge the closest objects and compute a new closeness value. Assuming that the algorithm terminates when no closeness exceeds a threshold of 15, the final partition is o1, o2, o3 in one group and o4 in another group.
**Ratio Cut**

- A constructive algorithm that groups objects until a terminal condition has been met.
- A new metric *ratio* is defined as
  \[ \text{ratio} = \frac{\text{cut}(P)}{\text{size}(p_i) \times \text{size}(p_j)} \]
  - \( \text{Cut}(P) \): sum of the weights of the edges that cross \( p_1 \) and \( p_2 \).
  - \( \text{Size}(p) \): size of \( p \).
- The ratio metric balances the competing goals of grouping objects to reduce the cutsize without grouping distance objects.
- Based on this new metric, the partition algorithms try to group objects to reduce the cutsizes without grouping objects that are not close.

Ratio cut, in fact, can be regarded as one of the multi-stage clustering. The metric used in ratio cut will include the size of the functional objects. One simple closeness function example is called cut-size which represents number of edges between a pair of objects. If the clustering algorithm is built based on the cutsize, because very often the tiny object has smaller number of edges, it is possible that a large object will always be grouped with a tiny object although they are not really close. To take into account the size of objects, the ratio cut will be used as the closeness function instead. Ratio cut algorithm groups objects until a termination condition has been met, meaning that no objects are considered close enough to be merged. The new termination condition relies upon the definition of a new partition metric. A partition can be evaluated by the cutsizes of its groups, since small cutsize indicate that close objects have been grouped, as desired. However, if cutsize is the only metric, then the best cutsize is reached by merging all objects into one group, even if the objects are not considered close. To avoid ending up with a single group we could impose a constraint on the number of objects or the size of a group, but such a constraint would prevent us from finding good partitions consisting of unbalanced group sizes. Base on the above considerations, the goal of ratio cut partition is to group objects to reduce the cutsize without grouping objects that are not close, and without constraining the size of a group. This goal is achieved by replacing the cutsize metric with a new metric called *ratio*. The ratio metric will balance the competing goals of grouping objects to reduce the cutsize without grouping distant objects. The denominator of the ratio equation encourages maintaining multiple groups with balanced sizes.

**Greedy Partitioning**

- Two-way partition algorithm between the groups of HW and SW.
- Suffer from local minimum problem.

This slides shows one of the iterative partitioning algorithm called two-way partitioning. Here this greedy algorithm starts from a initial partitioning which consists of two sets. This algorithm will try to move the object one at a time from one set into another set to check if it will lower the partitioning cost. If the cost increase, the object will not be moved at all. This algorithm is quite simple, but most of the time it can also provide us the local minimum result.

The algorithm begins by creating an all-hardware partitioning, thus guaranteeing that a performance-satisfying partition is found if it exists (actually, certain behaviors considered unconstrainable are initially placed in software). A performance-satisfying partition is one in which all performance constraints are satisfied. To move an object in the algorithm require not only that cost be improves, but also that all performance constraints still be satisfied (actually they require that maximum interfacing constraints between groups be satisfied). Once a behavior is moved, the algorithm tries to move closely related objects first. While greedy algorithms are fast, their chief drawback is that they cannot escape a local minimum.
**Group Migration**

- Another iteration improvement algorithm extended from two-way partitioning algorithm that suffers from local minimum problem.
- The movement of objects between groups depends on if it produces the greatest decrease or the smallest increase in cost.
  - To prevent an infinite loop in the algorithm, each object can only be moved once.

Group migration algorithm can improve the simple two-way partitioning that suffers the local minimum problem. This algorithm will include some control strategy. In two-way partitioning, the object can be moved from one group into another group only if this move can decrease the cost. This algorithm cannot detect a case in which no single move decreases the cost, but a sequence of two or more moves does decrease the cost. To solve this problem, the condition of move will be relaxed and the move of objects between groups in each iteration is allowed if either it produces the greatest decrease of the smallest increase in cost. To prevent an infinite loop in the algorithm, each object can only be moved once. After all the objects have been moved once, we select the lowest-cost partition we have encountered. The entire algorithm is iterated using the new partition as the initial partition, until we no longer encounter a lower-cost partition we repeats is usually less than five.

The algorithm is easily extended for multiway partitioning. In two-way partitioning, we tentatively moved every object to its opposite group to see which object move produced the lowest cost. In multiway partitioning, we tentatively moved every object to every other group. In an alternative extension for multiway partitioning, we first create two groups using two-way partitioning, and then repeatedly partition each group into two groups, again using two-way partitioning. We continue partitioning groups until we obtain the desired number of groups.

**Simulated Annealing**

- Iterative algorithm modeled after physical annealing process to avoid local minimum problem.
- **Overview**
  - Starts with initial partition and temperature
  - Slowly decreases temperature
  - For each temperature, generates random moves
  - Accepts any move that improves cost
  - Accepts some bad moves, less likely at low temperatures
- **Results and complexity depend on temperature decrease rate**

The group migration algorithm escapes local minimum by accepting cost-increasing moves if they are part of a sequence of moves that leads to a lower-cost partition. However, the complexity is limited by moving each object only once in the sequence. The simulated annealing algorithm also accepts cost-increasing moves. In contrast to group migration, simulated annealing may move each object more than once, limiting the complexity by decreasing over time the tolerance for accepting cost-increasing moves.

The simulated annealing, intended to model the annealing process in physics, where a material is melted and its minimal energy state is achieved by lowering the temperature slowly enough that equilibrium is reached at each temperature. The algorithm starts with an initial partition and initial simulated temperature, and then the temperature is slowly decreased. For each temperature, random moves are generated. The algorithm accepts any move that improves the cost; otherwise, it may still accept the move but such acceptance becomes less likely at lower temperatures.
Simulated Annealing Algorithm

Temp = initial temperature
Cost = Objfct(P)
While not Frozen loop
  while not Equilibrium loop
    P_tentative = Move(P)
    cost_tentative = Objfct(P_tentative)
    ∆cost = cost_tentative - cost
    If Accept(∆cost, temp) > Random(0,1) then
      P = P_tentative
      cost = cost_tentative
    end if
  end loop
  temp=DecreaseTemp(temp)
End loop

where Accept(∆cost, temp) = min(1, e^-∆cost/temp)

This slides shows the algorithm of simulated annealing partition method. The key part of this algorithm is the acceptance procedure which is used to determine whether to accept a move based on the cost improvement and current temperature. This procedure will return a value and if that value is greater than a random number between 0 and 1, the move will be accepted. As we can see, if we lower the temperature (i.e. decrease the value of temp), the value procedure returns will become smaller, and the possibility of acceptance will become small. The algorithm promises to achieve a good results while the temperature has gradually decreased.

Theoretical studies have shown that the simulated-annealing algorithm can climb out of a local minimum and find the globally optimal solution if the processes reaches an equilibrium state at each temperature, and if the temperature is lowered infinitely slowly. The above conditions require an infinite number of iterations at an infinite number of temperatures, which is clearly impractical, so several heuristic approaches have been developed to control the simulated-annealing process. These heuristics define the equilibrium state and describe how to lower temperature.

Genetic Evolution

- Genetic algorithms treat a set of partitions as a generation, and create a new generation from a current one by imitating three evolution methods found in nature.
- Three evolution methods
  - Selection: randomly select partition
  - Crossover: randomly select two strong partitions, replicate trait
  - Mutation: randomly select partition, randomly modify (move object)
- Produce good result but suffer from long run times.

The group migration and simulated annealing algorithms improve a current partition by moving a number of objects, saving the best partition encountered, and iterating the process with this best partition. However, we need not restrict ourselves to save only partition from one iteration to the next.

Genetic evolution modeled after the genetic evolution process can also be applied to the partitioning applications. This algorithm will regard a set of partitions as a generation. Genetic algorithm create a new generation from three possible methods. One is selection which randomly choose a low-cost partition and copies it to the next generation. The second method is crossover which randomly selects two strong two partitions and replicates a trait of one in the other. The final method is called mutation which randomly selects a partition and modifies it by moving some randomly selected objects.

Genetic algorithm complexity is heavily influenced by the Terminate procedure’s form. Like simulated annealing, genetic algorithm usually produce good results but suffer from long run times. Also, since genetic algorithms maintain multiple partitions, they require more memory.
Estimation

- Estimates allow
  - Evaluation of design quality
  - Design space exploration
- Design model
  - Represents degree of design detail computed
  - Simple vs. complex models
- Issues for estimation
  - Accuracy
  - Speed
  - Fidelity

In the previous slides, we described some of the basic issues and techniques for partitioning a system’s functionality among various system components, such that constraints on various design metrics, such as performance and area, would be satisfied. In order to determine if these constraints have actually been satisfied, however, we must be able to obtain metric estimates as rapidly as possible. Here, we will describe some of the techniques that can be used for such rapid estimation of both hardware and software quality metrics.

The goal of estimation used in the partition or system design is to evaluate the design quality of the current design or partition based on some given design metric. This evaluation can provide us if the constraint of the design will be met or it can help to explore the possible design space.

While we make some estimation of the given design, we’ll assume some architectures which the design will be implemented on. In other word, a design model should be built for the quality metric to estimate. Surely the model can be simple or complex depending on our estimation goal. In general, we want the estimation can be as more accurate as possible, but it may suffer longer computation time. We won’t every estimation will take lot of time because it may reduce the design space we can explore. Another estimation issue is fidelity. Fidelity reflects the relative correctness of the estimation results.

Accuracy vs. Speed

- Accuracy: difference between estimated and actual value
  \[
  A = 1 - \frac{|E(D) - M(D)|}{M(D)}
  \]
  \[E(D), M(D)\] : estimated & measured value
- Speed: computation time spent to obtain estimate

The accuracy of an estimate is a measure of how close the estimate is to the actual value of the metric measured after design implementation. Perfect estimation has an accuracy equals 1. The accuracy depends on the degree of the detail in the model. Estimators based on simple models execute rapidly but may not provide the accuracy good enough.

A design model may incorporate several aspects of the design. For example, a detailed area estimate would require determining the number and the size of the memories, functional units, registers and multiplexers in addition to the wiring area. These calculations would in turn involve performing tasks such as functional-unit allocation, variable lifetime analysis, functional-unit binding and floorplanning. Estimators based on such detailed design models require longer computation times, but also produce more accurate estimates, and therefore allow a better selection of design alternatives.
Fidelity

- Estimates must predict quality metrics for different design alternatives
- Fidelity: % of correct predictions for pairs of design implementations
- The higher fidelity of the estimation, the more likely that correct decisions will be made based on estimates.
- Definition of fidelity:

\[
F = 100 \times \frac{2}{n(n-1)} \sum_{i=1}^{n} \sum_{j=i+1}^{n} u_{i,j}
\]

(A, B) = E(A) > E(B), M(A) < M(B)  \[\times\]
(B, C) = E(B) < E(C), M(B) > M(C)  \[\times\]
(A, C) = E(A) < E(C), M(A) < M(C)  \[\bigcirc\]

Fidelity = 33%

If estimation of metric is to check if the design can meet the target constraint, then the accuracy of the estimation result will become very important. The fidelity of an estimation method is defined as the percentage of correctly predicted comparisons between design implementations.

However, if the estimation is to compare different implementations or different partitions in order to find the best one, fidelity becomes a more important issue. Fidelity will reflect the correctness of the predictions. The higher the fidelity of the estimation, the more likely that correct design decisions will be made based on comparing the estimates of two implementations.

During the selection of one of several design implementations, predictions of design quality based on estimates with high fidelity will result, on average, in better designs. Fidelity depends upon the design model used to estimate the design parameter. In general, the more accurate the model, the higher the fidelity of estimation.

Quality Metrics

- **Performance Metrics**
  - Clock cycle, control steps, execution time, communication rates
- **Cost Metrics**
  - Hardware: manufacturing cost (area), packaging cost (pin)
  - Software: program size, data memory size
- **Other metrics**
  - Power
  - Design for testability: Controllability and Observability
  - Design time
  - Time to market

The quality metrics we are often interested in system design are discussed. The essential two metrics includes the performance metrics and the cost metric. The performance of the system can be regarded from many aspects including execution time, communication rates, clock cycle, or control steps.

Performance metric can be divided into computation and communication metrics. Computation metrics measure the time required to perform the computations within a behavior. These metrics can be classified according to the type of the time units used in computation of the metric. Communication metrics are related to the time spent by a behavior in interacting with the other behaviors in the system. Although the execution time can reflect the performance best, it will need more efforts to estimate.

The cost of system can be measure in silicon area or the program size depending on what applications you are interested. In addition to metrics, there are other metrics you can consider including power, time-to-market, and the testability.
**Scheduling**

- A scheduling technique is applied to the behavior description in order to determine the number of control steps.
- It is quite expensive to obtain the estimate based on scheduling.
- Resource-constrained vs time-constrained scheduling.

The scheduling technique may also be applied to the behavior in order to determine the number of control steps. Usually the scheduling algorithm can produce very good estimation of control step, however, the computation complexity of the algorithm may be high. Scheduling techniques can roughly be divided into two classes. One is resource constraint scheduling which assumes constrained number of functional units are given in advance. Given resource-constraints, the operations in the behavior are scheduled with a view to minimizing the total number of control steps. For each type of operation in the behavior, the list scheduling algorithm maintains a priority list from which operations are assigned to control steps. At any stage, the priority list for an operation type constrains the set of operations that have all their predecessors already scheduled. Thus, all predecessors of an operation must be executed before that operation itself is executed. The other one is time-constrained scheduling which is to find out a schedule that can meet the timing requirement with the least amount of resources.

**Control Steps Estimation**

- Operations in the specification assigned to control step
- Number of control steps reflects:
  - Execution time of design
  - Complexity of control unit
- Techniques used to estimate the number of control steps in a behavior specified as straight-line code
  - Operator-use method
  - Scheduling

A control step corresponds to a single state of the control unit state machines. Operations in the functional specification are assigned to these control steps during synthesis. Each operation may take different number of control steps. Therefore, the estimation of control steps can be said to find out the number of cycles required to execute the system. The number of steps can reflect the execution time and the complexity of the control unit.

The number of control steps needs for a behaviors execution can be estimated in several ways. We first describe two techniques - Operator-use method and Scheduling algorithm – for estimating the number of control steps in a behavior specified as straight-line code. We then present a technique for estimating the number of control steps for a behavior that contains control constructs such as branching and iteration.

Here, we’ll introduce two simple ways to obtain the control step number in the behavior. We will first focus on the behaviors where no branches will occur.
**Operator-use Method**

- Easy to estimate the number of control steps given the resources of its implementation.
- Number of control steps for each node can be calculated:

\[
csteps(n_j) = \max_{t_i \in T} \left[ \frac{\text{occur}(t_i)}{\text{num}(t_i)} \times \text{clocks}(t_i) \right]
\]  

(1)

- The total number of control steps for behavior B is

\[
csteps(B) = \max_{n_i \in N} csteps(n_j)
\]

(2)

First method is called operator-use method. It will first divide all statement in a behavior into a set of nodes in such a way that all the statements in a node could be executed concurrently. Therefore, this method simply considers the number of operations required and the number of functional units available.

Let \( T \) represent the number of distinct types of operations in a behavior B. Let \( \text{num}(t_i) \) and \( \text{clock}(t_i) \) represent the number and delay (in clock cycles) of functional units available to implement operations of type \( t_i \). Then, if there are \( \text{occur}(t_i) \) occurrences of an operation, Equation (1) gives the number of steps needed to execute operations of type \( t_i \). The number of control steps needed for any node \( n_j \), \( csteps(n_j) \), is equal to the maximum number of control steps needed to perform operations of any type in the node. Equation (1) gives the details.

Once the control steps for each node have been determined, the total number of control steps required by behavior B is determined as Equation (2), where \( N \) is the number of nodes in B.

**Operator-use Method Example**

- **Differential-equation example:**

This slide shows an example of operator-use method. The differential equation’s behavior can be first described as a set of statements. Next, we can analyze the data dependence existing in these statements. Then, independent statements will be grouped together and the operator-use method will be applied to each block. Finally, the final estimation result is sum of the control steps required for each block.

The operator-use method can be extended to incorporate pipeline functional units and include memory accesses.

The operator-use method can provide fairly rapid estimates of the number of control steps required by a behavior. If there are \( n \) operations in the behavior, the method has a computational complexity of \( O(n) \). However, there is potential for error, since the method operate at a statement-level granularity and ignores the dependencies between the operation within a statement.

In conclusion, the operator-use method will generate more accurate estimates if each statement in the specification is restricted to one operation.
Execution Time Estimation

- Average start to finish time of behavior
- **Straight-line code behaviors**
  \[ \text{execution}(B) = \text{csteps}(B) \times \text{clk} \]  
- **Behavior with branching**
  - Estimate execution time for each basic block
  - Create control flow graph from basic blocks
  - Determine branching probabilities
  - Formulate equations for node frequencies
  - Solve set of equations
  \[ \text{execution}(B) = \sum_{B \in B} \text{exectime}(b_i) \times \text{freq}(b_i) \]

The estimation of the execution time for the straight-line code is straightforward. We can estimate first the number of control step \( \text{csteps}(B) \) and then multiply it by the clock period. However, in the general case, a behavior may consist of sequential statements that have branching and iteration constructs. Since branching may be dependent upon input data, we have to analyze the frequency of each statement.

We first determine the set of basic blocks for the behavior. Since each basic block consists of a set of sequential assignment, we can determine the execution time for each block by first determining the number of control steps it requires. To determine the execution time for the entire behavior, the execution time \( \text{exectime}(b_i) \) for each basic block \( b_i \) needs to be weighed by the execution frequency of the basic block. The execution frequency, \( \text{freq}(b_i) \), for a basic block \( b_i \) is defined as the number of times that the basic block will be executed on the average, during a single execution of the behavior to which it belongs. Once the execution frequencies have been determined, \( \text{exectime}(B) \) is estimated as in Equation (2).

In order to estimate the execution frequency for each basic block, the control flow graph model has to be first built. Based on the model, we can determine the execution frequencies of each node using flow analysis based on the branching probability. Branch probability measures how often a branch is executed after evaluating the condition associated with a branching statement. The probability can be computed statically for the loop case or obtained dynamically by simulating the behaviors on several sets of sample data, recording how often the various branched were executed and consequently, deriving the probabilities for the individual branches.

We will illustrate the probability-base flow analysis method with an example of the VHDL behavior in the Figure(a). The basic blocks for VHDL statements are shown in Figure(b). The control flow graph of Figure(c) has a vertex for each basic block in the behavior. An edge exists between two vertices if control could be transferred between the corresponding basic blocks.
Probability-based flow analysis

- Flow equations:
  \[ \text{freq}(S) = 1.0 \]
  \[ \text{freq}(v_1) = 1.0 \times \text{freq}(S) \]
  \[ \text{freq}(v_2) = 1.0 \times \text{freq}(v_1) + 0.9 \times \text{freq}(v_5) \]
  \[ \text{freq}(v_3) = 0.5 \times \text{freq}(v_2) \]
  \[ \text{freq}(v_4) = 0.5 \times \text{freq}(v_2) \]
  \[ \text{freq}(v_5) = 1.0 \times \text{freq}(v_3) + 1.0 \times \text{freq}(v_4) \]
  \[ \text{freq}(v_6) = 0.1 \times \text{freq}(v_5) \]

- Node execution frequencies:
  \[ \text{freq}(v_1) = 1.0 \text{ freq}(v_2) = 10.0 \]
  \[ \text{freq}(v_3) = 5.0 \text{ freq}(v_4) = 5.0 \]
  \[ \text{freq}(v_5) = 10.0 \text{ freq}(v_6) = 1.0 \]

This slide shows an example for probability-based flow analysis. The execution frequency for any node depends on the weighted execution frequency of all its immediate predecessor nodes. We can derive a set of flow equations based on this relation. The flow equation can be solved to give you the node execution frequencies.

In the above method, we obtained the execution time of the entire behavior by associating the execution time of the corresponding basic block with each node and performing flow analysis. Probability-based flow analysis can also be applied to determine other useful design characteristics by associating different types of information with each node in the control flow graph. For example, if each node in the control flow graph represents the number of memory accesses in the corresponding basic block, flow analysis will yield the average number of memory accesses during execution of the entire behavior. Similarly, if we associate the number of calls made to a certain procedure in the basic block with each node, we can determine the number of calls made by the entire behavior to the procedure. Finally, by associating the number of data transfers over a channel in the corresponding basic block with each node, flow analysis will yield the total number of channel accesses by the entire behavior.

Communication rate

- Communication between concurrent behaviors (or processes) is usually represented as messages sent over an abstract channel.
- Communication channel may be either explicitly specified in the description or created after system partitioning.
- Average rate of a channel \( C \), \( \text{average}(C) \), is defined as the rate at which data is sent during the entire channel’s lifetime.
  \[ \text{average}(C) = \frac{\text{Total bits}(B,C)}{\text{comptime}(B) + \text{commtime}(B,C)} \]
- Peak rate of a channel, \( \text{peakrate}(C) \), is defined as the rate at which data is sent in a single message transfer.
  \[ \text{peakrate}(C) = \frac{\text{bits}(C)}{\text{protocol delay}(C)} \]

Communication channel may be either specified in the behavior description or created when a variable being accessed by a portion of the behavior is assigned to a different chip or block. Since the channel is part of the overall system, its effect has to be analyzed as well.

To evaluate the channel, several terms such as average rate and peak rate have been defined.

The total number of bits sent over a channel \( C \) during the lifetime of behavior \( B \) is \( \text{total bits}(B,C) \).

Computation time, \( \text{comptime}(B) \), is defined as the time required by the behavior \( B \) to perform its internal computations. Communication time is defined as the time spent by the behavior in accessing data external to the behavior. The communication time required by a behavior \( B \) to transfer data over channel \( C \) is denoted by \( \text{commtime}(B,C) \). \( \text{bits}(C) \) represent the number of bits received or sent over channel \( C \) in a single message. \( \text{protocol delay}(C) \) represents the delay associated with the transfer of a single message over the channel \( C \).
Area Estimation

- Two tasks:
  - Determining number and type of components required
  - Estimating component size for a specific technology (FSMD, gate arrays etc.)
- Behavior implemented as a FSMD (finite state machine with datapath)
  - Datapath components: registers, functional units, multiplexers/buses
  - Control unit: state register, control logic, next-state logic
- Area can be accessed from the following aspects:
  - Datapath component estimation
  - Control unit estimation
  - Layout area for a custom implementation

Estimation of the size of design has to first determines the number and the type of components required and the size of each component for a variety of technology. If we assume that our design is implemented on a FSMD, then the component of FSMD includes data path component like functional unit, registers etc and the control unit. Therefore, the estimation of area can be done for data path and control unit.
Clique-partitioning

- Commonly used for determining datapath components
- Let $G = (V,E)$ be a graph, $V$ and $E$ are set of vertices and edges
- Clique is a complete subgraph of $G$
- Clique-partitioning
  - divides the vertices into a minimal number of cliques
  - each vertex in exactly one clique
- One heuristic: maximum number of common neighbors
  - Two nodes with maximum number of common neighbors are merged
  - Edges to two nodes replaced by edges to merged node
  - Process repeated till no more nodes can be merged

To estimate the number of data path component required, a clique partitioning approach is often used. A graph $G = (V,E)$ is first derived from the behavior where node $V$ represents a operation and edge $E$ will connect nodes that can be mapped to the same functional units. A complete subgraph or a clique of $G$ represents a functional unit for all the operation in the subgraph. The number of functional units required is to find the minimal number of cliques that cover all the nodes.
Storage Unit Estimation

- Variables used in the behavior are mapped to storage units like registers or memory.
- Variables not used concurrently may be mapped to the same storage unit
- Variables with non-overlapping lifetimes have an edge between their vertices.
- Lifetime analysis is popularly used in DSP synthesis in order to reduce number of registers required.

Storage units like registers are required for holding data values represent by the constants, variables and arrays in the behavior. Similar to the functional unit, we also want to reuse the storage units to reduce the number of registers required. In order to do that, we can use the clique partitioning technique or we can use the lifetime analysis method.
Functional-unit and interconnect-unit Estimation

- Clique-partitioning can be applied
- For determining the number of FU’s required, construct a graph where
  - Each operation in behavior represented by a vertex
  - Edge connects two vertices if corresponding operations assigned different control steps and there exists an FU that can implement both operations
- For determining the number of interconnect units, construct a graph where
  - Each connection between two units is represented by a vertex
  - Edge connects two vertices if corresponding connections are not used in same control step

Clique-partitioning can be applied to estimate the minimum number of functional units and register requires. The interconnection channel required for connecting the functional units can also be derived by this technique. Here we assume the functional units shares some common communication channels.

Software Estimation Model

- Processor-specific estimation model
  - Exact value of a metric is computed by compiling each behavior into the instruction set of the targeted processor using a specific compiler.
  - Estimation can be made accurately from the timing and size information reported.
  - Bad side is hard to adapt an existing estimator for a new processor.
- Generic estimation model
  - Behavior will be mapped to some generic instructions first.
  - Processor-specific technology files will then be used to estimate the performance for the targeted processors.

Processor-specific: adv. The estimates obtained are very accurate, since the behavior is actually compiled for execution on the selected processor. However, such an estimation approach requires a dedicated compiler for each processor. Consequently, it is difficult to adapt an existing estimator for a new processor. In addition, compiling a behavior for the estimation purposes is computationally expensive.

Compared with the processor-specific model, the generic estimation model has several advantages. First, the generic estimation model does not require different compilers and estimators for different target processors. Instead, only a single compiler, estimator and a set of technology files is requires for software estimation. Second, the generic model makes retargeting the estimator to a new processor much easier. Retargeting consists of providing a technology file for the new processor. In the processor-specific model, we would require a compiler for the new processor in addition to the timing and size information of the processor’s instruction set. Finally, the peculiarities of each type of processor can be reflected in the technology file for the processor.

A disadvantage of generic model is the lower accuracy of its estimates largely because the generic instruction set represent only a small portion of processor’s entire instruction set.
Software estimation models

The behavior is first compiled into a set of generic three-address instructions. Processor-specific technology files are made available containing information about the number of clock cycles and bytes that each type of generic instruction requires.

The estimator computes the software metrics for the behavior based on the generic instructions and the technology files for the target processors.

The proposed generic instruction set consists of the following five classes:
1. Arithmetic/logic/relational instructions: \(<\text{dest} \leftarrow \text{src1 op src2}>\)
2. Move/load/store instructions: \(<\text{dest} \leftarrow \text{src}>\)
3. Conditional jump instruction: \(<\text{if cond goto label}>\)
4. Unconditional jump instruction: \(<\text{goto label} e>\)
5. Procedure call instruction: \(<\text{call label}>\)

Deriving processor technology files

The generic instruction given in figure is compiled into a sequence of three 8086 instructions.

Here, \(d\text{mem}\) indicates a direct memory addressing mode. The generic instruction is first mapped to a sequence of target processor instructions. The total number of clock cycles of the generic instruction is obtained by summing the clock cycles of the generic instruction in the sequence. EA1 and EA2 in figure are the effective address calculation times used for displacement memory addressing mode, which are six and eight clock cycles on the 8086 and 68020 processors, respectively.

Thus, the generic instruction will take 35 and 22 clock cycles on 8086 and 68020 processors, respectively. Using a similar approach we can derive the number of bytes that each type of generic instruction will require if it is compiled in 8086 or 68020 processor.

The total size of the three instructions is 10 bytes, which is entered in the technology file for the 8086 processor as the size of the generic instruction under consideration.
Software estimation

- **Program execution time**
  - Create basic blocks and compile into generic instructions
  - Estimate execution time of basic blocks
  - Perform probability-based flow analysis
  - Compute execution time of the entire behavior:
    \[ \text{execute}(B) = \delta \times (\sum_{b_i \in B} \text{exectime}(b_i) \times \text{freq}(b_i)) \]  
    (1)
  - \( \delta \) accounts for compiler optimizations

- **Program memory size**
  \[ \text{progsize}(B) = \sum_{g \in G} \text{instr\_size}(g) \]

- **Data memory size**
  \[ \text{datasize}(B) = \sum_{d \in D} \text{datasize}(d) \]

The program execution time can be estimated by first dividing the program into the basic blocks which consist of only straight line statements. Next, the probability flow analysis can be applied to find out the execution frequencies of each basic block. The overall execution time can be summed by the execution time of each basic block.

Finally, equation 1 can be used to obtain the average-case software execution time for the entire behavior.

In order to estimate the performance corresponding to the optimized code, we need to know the optimization ratio of the compiler used by the designer to generate the machine instructions. The performance-optimization ratio, \( \delta \), is defined as the ratio of optimized code performance to the non-optimized code performance.

Based on the size of each generic instruction, the program-memory size if behavior is then computed as the sum of those generic instructions in the behavior. If a behavior \( B \) is compiled into a set of generic instructions \( G \), and instr_size\( (g) \) represents the size of the generic instruction \( g \), then the size of the program required for behavior \( B \) is computed as progsize\( (B) \).

The data memory size is determined by examining the data declarations in the functional specification. The data memory size, datasize\( (d) \), of a declaration \( d \) is determined by the size of \( d \)'s base type and number of elements in \( d \). The base type of any declaration is an indivisible type defined in the language. The data memory size of a behavior \( B \) can be computed by summing that for each its declarations. The data memory size estimation describe here is accurate under the assumption that all variables have lifetimes equal to the execution time of the behavior, which means we do not share memory locations for two variables.

Refinement

- **Refinement is used to reflect the condition after the partitioning and the interface between HW/SW is built**
  - **Refinement** is the update of specification to reflect the mapping of variables.

- **Functional objects are grouped and mapped to system components**
  - Functional objects: variables, behaviors, and channels
  - System components: memories, chips or processors, and buses

- **Specification refinement is very important**
  - Makes specification consistent
  - Enables simulation of specification
  - Generate input for synthesis, compilation and verification tools

After partitioning the system behavior to several functional objects and mapping these objects to suitable objects. The original specification should be further refined to reflect the update architecture. It can make the specification consistent in all aspects, and it also makes the specification simulatable. In addition, the refined specification that is generated serves as an input for verification, synthesis and compilation tools that may follow the system design.

Here, we present the set of specification refinement tasks. First, we describe the refinement tasks associated with the implementation of variable and channel groups. Second, we introduce mechanisms for resolving access conflicts that may arise when several behaviors concurrent access variables or channels that have been grouped together into a memory or a bus. Third, we discuss the effect of binding behaviors of the pin structure and protocols used for communication. Methods for interfacing two standard components with incompatible protocols are also presented. Finally, we discuss issues related to the implementation of communication between behaviors that have been assigned to hardware and software system components,
Refining variable groups

- The memory to which the group of variables are reflected and refined in specification.
- Memory address translation
  - Assignment of addresses to each variable in group
  - Update references to variable by accesses to memory

\[
\begin{align*}
V(K) & := 3; \\
X & := V(36); \\
V(J) & := X;
\end{align*}
\]

\[ \text{for } J \text{ in } 0 \text{ to } 63 \text{ loop} \]
\[ \text{SUM} := \text{SUM} + V(J); \]
\[ \text{end loop;} \]

\[
\begin{align*}
\text{MEM}(K +100) & := 3; \\
X & := \text{MEM}(136); \\
\text{MEM}(J +100) & := X;
\end{align*}
\]

\[ \text{for } J \text{ in } 0 \text{ to } 63 \text{ loop} \]
\[ \text{SUM} := \text{SUM} + \text{MEM}(J +100); \]
\[ \text{end loop;} \]

The first step to refine the original specification is to refine the variable groups. The variables used in the behaviors have to be folded into the memory location or the registers. In addition, all the memory reference of the variables used in the specification should also be translated into the real position.

System partitioning maps a group of variables to an allocated memory with a fixed number of words and width, different variables may have different sizes. Variable folding refers to the assignment of the bits in a variable to the bits in each word of memory.

The assignment of addresses to variables mapped to memory and the modification of all references to those variables in the specification is known as memory address translation. To assign memory addresses, one can consider the variables in any order. The number of memory words required for each variable can be determined by variable folding. Elements of an array variable must be assigned contiguous addresses in the memory. For scalar variables, memory address translation is relatively straightforward. For an array variable that has been grouped with other variables, the addresses that index the variable must be updated while refining references to that variable.

Communication

- Shared-memory communication model
  - Persistent shared medium
  - Non-persistent shared medium
- Message-passing communication model
  - Channel
    - uni-directional
    - bi-directional
    - Point-to-point
    - Multi-way
  - Blocking
  - Non-blocking
- Standard interface scheme
  - Memory-mapped, serial port, parallel port, self-timed, synchronous, blocking

A system usually consists of several concurrent behavior which need to communicate some data or control information with each other. Usually this communication can be conceptualized in either share-memory or message passing paradigms. In shared-memory communication model, the sending behavior or process writes data to a shared medium while the receiving process can read the data from the medium. Shared medium can also be of the persistent of non-persistent. A persistent medium such a a storage element can retain the value written into it by the process until that value has been rewritten by a new one. Non-persistent medium cannot retain the value such as the wire connection.

In the message-passing model, the details of data transfer between processes are replaced by communication over an abstract medium called a channel, over which the data or messages are sent. A channel is a virtual entity free of any implementation details. Only after synthesis would a channel be implemented in the form of a bus consisting of a set of wires and protocols to sequence the data transferred over these wires. There are many variations including uni-directional, bi-directional, point-to-point or multiway for the channel model. A further distinction rests in whether the message-passing communication is blocking or non-blocking. Blocking means if a process which communicates over a channel will suspend until the other process is ready for the data transfer. The main advantages of blocking is that two processes can synchronize at the data transfer, and no additional storage is required. However, the drawback is that the process will be suspend which can result in performance deterioration. Non-blocking communication does not have this problem, however, external storage must be implicitly associated with the channel usually in the form of a queue.
The slides show two conceptual communication models which are mentioned in previous slides. For shared-memory communication model, two processes or behaviors are exchanging information through some common shared memory. For message passing model, the processes communicate with each other by sending a message through the channel.

Concurrent behaviors in a specification communicate with another by sending a message over abstract communication channels. In order to minimize the interconnect cost, the channels in the system are grouped in such a way that each group of channels is implemented by a bus. To coordinate the use of the bus with concurrent behaviors, some protocols have to be implemented with the bus. The task of generating buses and their protocols for each group of channels is called interface refinement. Therefore, the refinement of channel includes two tasks: generation of bus with proper length and the associated protocol.

A simple case of buswidth generation in which all the channels in a group have a identical message size is presented. In this case, channels are merged in such a way that all channels in any group are used exclusively over time to communicate between the same two behaviors. Consequently, each channel group is implemented with a buswidth identical to the size of any channel. In more general case, behaviors communicating over channels that have been grouped together may want to transfer data over the shared physical medium simultaneously. In addition, different channels may be transferring messages of different sizes of different sizes between the behaviors.
Various access protocol to bus can be used such as full-handshake, half-handshake or the hardwired port. Different protocol will require different control lines.

The behavior that will use the bus should be assigned an unique ID to differentiate the user of the bus at any time.

If N channels are implemented on the same bus, \( \log_2(N) \) lines will be required to encode the channel ID. A unique ID is assigned to each channel. The structure determined for the bus is defined in the specification. For each channel mapped to the bus, appropriate send and receive procedures are generated, encapsulating the sequence of assignments to the bus control, data and ID lines to execute the data transfer. References to a variable that has been assigned to another system component by system partitioning must be updated in behaviors that were originally referencing it directly. Access to variables are replaced by the send receive procedure calls corresponding to the channel over which the variable is accessed. In order to obtain a simulatable system specification, a separate behavior is created for each group of variables accessed over a channel. Appropriate send and receive procedure calls are included in the behavior to respond to access requests to the variable over the bus.

---

This slide shows the declaration of an eight bit bus, with two control lines and two ID lines.

Bus B is declared to be a global variable so that all behaviors can access it. Behavior P writes to the 16-bit variable X over channel CH0. Since the buswidth is only eight bits, procedures SendCH0 and ReceiveCH0 transfer the 16-bit message associated with channel CH0 over the bus, in two transfers of eight bits each.
Refined specification after protocol generation

In previous figure, behavior P writes the values “32” directly to the variable X. Channel CH0 represents the write to variable X. The statement “X<=32” is replaced by the send procedure call “sendCH0(32)” in this slide. The statement “MEM(60) :=COUNT” in behavior Q is updated to “sendCH3(60,COUNT)”, indicating that the value in COUNT is to be written to address 60 of array MEM.

In previous figure, the variable X and MEM were assigned to different system components, as shown by dashed lines. In the slide, behaviors Xproc and MEMproc have been created for these two variables.

The protocol generation has several advantages. First, the refined specification is simulatable, and the design functionality, after insertion of buses and communication protocols, can be verified. Second, by encapsulating data transfer over the bus in terms of send and receive procedures, the description of the behavior remains less cluttered than it would be if we were to insert the assignments for the control and data lines at each communication point in the behavior. Finally, if at a later stage another communication protocol were selected for communication over the bus, only the bus declaration and send and receive procedures need be changed. The system’s behavior descriptions, including the send and receive procedure calls, remain unchanged.

Arbitration schemes

- Arbitration schemes determine the priorities of the group of behaviors’ access to solve the access conflicts.
- Fixed-priority scheme statically assigns a priority to each behavior, and the relative priorities for all behaviors are not changed throughout the system’s lifetime.
  - Fixed priority can be also pre-emptive.
  - It may lead to higher mean waiting time.
- Dynamic-priority scheme determines the priority of a behavior at the run-time.
  - Round-robin
  - First-come-first-served

The accesses by a set of behaviors to the shared resource must be prioritized. Given a group of behaviors needing to access a given resource, an arbitration scheme determines their relative priority in order to resolve potential access conflicts. Arbitration schemes may be classified as fixed priority and dynamic priority.

Determining the fixed priority for various behaviors depends on some metric which has to be optimized.

For example: mean waiting time, size of the data associated with accesses made by a behavior and frequency of accesses made by the behavior to shared resource.

Mean waiting time: represents the average time spent by any behavior waiting to gain access to a shared resource.

Behavior should be assigned priorities in a manner that minimizes the mean waiting time. For a specific assignment of priorities to behaviors, the resulting mean waiting time can only be determined dynamically by simulating the specification. To determine priorities of the behaviors statically, the mean waiting time is usually approximated by metrics that can be evaluated easily. One such metric is the size of the data associated with accessed made by a behavior.

A dynamic-priority scheme determines the priority of a behavior according to the state of the system at run-time. For example, a round-robin scheme assigns the lowest priority to the behavior that most recently accessed the shared resource. A first-come-first-served scheme will grant access privileges to behaviors in the order they requested the access. Such schemes are characterized by the absence of any absolute order in which behaviors are granted access to a resource.

Consequently, dynamic arbitration will not have to wait indefinitely to gain access to the shared resource.
Tasks of hardware/software interfacing

- Data access (e.g., behavior accessing variable) refinement
- Control access (e.g., behavior starting behavior) refinement
- Select bus to satisfy data transfer rate and reduce interfacing cost
- Interface software/hardware components to standard buses
- Schedule software behaviors to satisfy data input/output rate
- Distribute variables to reduce ASIC cost and satisfy performance

Any behaviors description can be implemented either as software or as hardware. If we choose to implement it as software, the behavioral description will be compiled into the instruction set of the chosen processor. If we choose to implement it as hardware, the behavior has to be synthesized into ASIC hardware. Suppose some behaviors in the system are implemented as SW, and some as HW, we have to refine the interface between hardware/software behaviors. The tasks related to the HW/SW interfacing are summarized in this slide. The interface between two behaviors include the data and control transfer. Therefore, we have to refine the data access for those shared variables, and then refine control access that coordinate the execution sequence between behaviors. Next, an appropriate bus width, protocol and architecture should be chosen to satisfy the required data transfer rate. (AMBA is a popular standard bus used in many SOC systems.) After the bus structure has been decided, the software and hardware behaviors has to be modified to be compatible to the bus standard. In many cases, the software behaviors can be properly scheduled to meet the data input and output rate. Finally, since some variables are shared between behaviors, the allocation of variables to either memory, or ASIC buffer may affect the overall performance, we have to pay attention to how to distribute these variables.

This slide shows an example of the refinement of a hardware/software. Figure (a) shows an example of a partitioned specification. In this specification, v1 to v6 represent variables, B1 to B4 are behaviors, and p1 to p3 are ports through which the behaviors communicates data with outside. The edges in the figure represent data accesses by the behaviors to the variables or ports. A behavior like B1 or B2 that is mapped to the software partition is called a software behavior, and similarly, the one mapped to the hardware partition is called a hardware behavior such as B3 or B4. Variables in the behaviors can be categorized as hardware variable or software variable and they can reside in different places. In this example, variable v1 is shared by the behaviors in the same partition and it can be simply taken by software compilation. Hardware variables v6 resides in the ASIC. Variable v2, v4 shared by the behaviors in different partition can be allocated into ASIC or memory. In this example, v2 will reside in the main memory while v4 is implemented in the hardware buffer. The hardware ASIC, software processor and the memory all communicate through a bus through which the data access will happen.
Data and control access refinement

- Four types of data access in HW/SW interface:
  - Software behaviors access memory locations.
  - Hardware behavior access memory locations.
  - Software behaviors access ports of the ASIC’s buffer.
  - Hardware behavior access the ASIC’s buffer.

- Control access refinement’s tasks:
  - Insert the corresponding communication protocols in the software and the hardware behaviors.
  - Insert any necessary software behaviors such as interrupt service routines.
  - Refine the accesses to any shared variables that have been introduced by the insertion of the protocols.

This slide summarizes the task of data and control access between behaviors. There are four types basic types of data access that may occur between the HW/SW interface. First, the software behaviors may access the software variables which reside in the memory. This type of data access is usually carried out through the processor’s load/store instructions. Second, the ASIC hardware IP may access the data in the memory. This type of data access is carried out through a direct-memory access (DMA) mechanism. They are similar to load and store operations except that ASIC first has to gain control of the bus. The third possibility is the software behaviors access ports of the ASIC’s buffer. In this scenarios, the data access can be carried out either through the process’s in/out or move instruction or memory-mapped I/O mechanism. Finally, the hardware behaviors may also access the ASIC’s buffer. This data access is usually realized during the implementation of the ASIC.

Unlike data access channel which exists between behavior and variable, control channel exists between two behaviors to indicate the starting or completion of behavior. This control can be built by inserting hand-shaking protocols which use some shared variables (start and done) to indicate the starting and completing points. However, if the behaviors are implemented as software, in order to improve the processor efficiency, usually the interrupting mechanism is incorporated such that the processor can execute other behavior while waiting for the completion of some behavior implemented in HW.

Contents

- Function-Architecture Codesign
  - Design Representation
  - Optimizations
  - Cosynthesis and Estimation
- Case Study
  - ATM Virtual Private Network Server
  - Digital Camera SoC

In this part of the codesign course, we will be mainly covering a well-known methodology called Function-Architecture Codesign (FAC) from UC Berkeley. FAC stresses on different phases of architecture-independent and architecture-dependent models and model-oriented optimizations, most of which are from conventional data-flow analysis. FAC is more wide in scope than the conventional hardware-software codesign in the fact that the final architecture of the system is taken into consideration during codesign, hence the name function-architecture codesign.

Two case studies will be presented to highlight the features of the methodology including IP re-use and SoC design. The ATM VPN example is from CSELT, Italy and the digital camera SoC is from Chapter 7 of Frank Vahid’s book on Embedded System Design.
Function Architecture Codesign (FAC) Methodology

- FAC is a top-down (synthesis) methodology
- More realistic than hardware-software codesign
  - More suitable for SoC design
- Maps function to architecture
  - Application functions
  - SoC Target Architecture
- Trade-offs between hardware and software implementations

Function Architecture Codesign (FAC) is a top-down design methodology for hardware-software systems. FAC was initially proposed in the following book, which was also a Ph.D. Dissertation of Dr. Bassam Tabbara.


Compared to the conventional hardware-software codesign, the FAC methodology is more suitable for SoC hardware–software codesign because of the complexity of an SoC. SoC is not merely hardware and software, which is what a conventional hardware-software codesign methodology would assume. Rather, SoC is a complex integration of hardware IPs and software modules, which requires a more detailed mapping of application functionalities to affordable hardware components and to efficient software components. The FAC methodology is a good solution to the above mapping problem since it maps functions to architecture. Hence, FAC is more suitable for SoC.

FAC is basically a top-down synthesis methodology based on a set of different models that are differentiated into architecture independent and architecture dependent models. FAC mainly requires two inputs as follows.

1. a functional description of the application under development, and
2. a description of the target system architecture.

For an SoC, the inputs will be a functional description of the SoC application and an architectural description of the to-be-designed SoC. For example, the FAC-based design of a Digital Camera (DigCam) SoC will require

- (1) a functional description of the DigCam SoC application
- (2) an architectural description of the DigCam SoC

This slide shows how functions and architectures must be mapped to generate the final architecture through an iterative refinement process and a verification process, while at the same time making trade-offs between hw and sw. There are basically two steps:

1. Trade-off between function and architecture: through an estimation of the architecture, one is able to make some trade-offs between function and architecture, for example, if the computation required by a function is more than what a selected processor in an architecture can handle, then it is required to either
   a) upgrade the CPU, for example use a more powerful CPU, or
   b) downgrade the functional requirements, for example use a simpler computation.

2. Trade-off between hardware and software: When functions are mapped to concrete components, either hardware or software, profiling can be done to estimate its performance. If either performance or cost does not satisfy user-given constraints, then trade-offs are required so that user-given constraints are met.

The above 2 steps constitute the refinement (synthesis) process in FAC. The reverse direction is verification, which requires abstraction of implemented systems. For example, a hardware-software implementation of a DigCam SoC need to be abstracted into a high-level model to check / verify if it satisfies user-given system-level functional or non-functional requirements.

An iteration of the above two directions, namely synthesis and verification, will result in an optimized target system.
This shows how functions can be implemented on an architecture by constraining and optimization through codesign. It appears as if the function light (torch) is reflected and constrained by the architecture tub (containing water?). A mutually constraining relationship exists between function and architecture as described in the following.

1. Some functions are constrained by the architecture due to either time or space limitations. For example, due to memory restrictions, some functions might have to use “fixed-point” computations which will result in quality degradation of the final computation results.
2. A given architecture will also provide some functionalities which are not required by a given functional description of a system. This part of the architecture should be minimized as much as possible for a cost-optimized final system.

Through the FAC methodology, one is able to consider this mutually constraining relationship and obtain an optimized system, which is not what a conventional hardware-software codesign methodology could possibly handle.

From ASV: Albert Sangiovanni Vincentelli slides
We first map different kinds of application requirements onto a single platform which is then used to derive a platform instance that represents the target system design. A two-phase process is the general methodology used in a platform-based design.

1. An application is first mapped to a platform, for example, a design reference board,
2. Then, the platform-based design is then transformed into a real design which maps the system to an architecture.

Since the platform plays an important role in the design process, one has to be careful not to be trapped by the platform, that is, we have to be sure of what parts of the platform were actually used and what parts are actually required, because otherwise we would end up having a system that uses more than what our cost estimates considered. An infeasible system would be the final result in this case. Thus, the platform brings convenience to a designer but at the same time the flexibility provided by a platform may be a cause of underestimation.

FAC is like a platform-based design process but it does not have the above shortcoming because its design process is a direct mapping to the final architecture without going through an “intermediate” platform stage.
Main Concepts

- Decomposition
- Abstraction and successive refinement
- Target architectural exploration and estimation

In FAC, the main concepts are decomposition, refinement, and exploration with estimation.

1. Decomposition: a complex application must be decomposed into modular functionalities or functionalities that can be implemented by some hardware IP or software COTS. A clever decomposition is also a key-step in a successful FAC mapping.

2. Abstraction and successive refinement: The FAC design process is an iteration of abstraction and refinement because a system is designed by mapping the abstraction of what an architecture can provide with the refinement of functionalities in an application.

3. Target architectural exploration and estimation: A target architecture can still possess several flexibilities such as the amount of memory, etc. which need to be explored and estimated for the final design results.

Decomposition

- Top-down flow
- Find an optimal match between the application function and architectural application constraints (size, power, performance).
- Use separation of concerns approach to decompose a function into architectural units.

Decomposition is a top-down flow involving an optimal match between application function and architectural constraints. We need to use separation of concerns approach to decompose a function into architectural units. For example, some typical considerations are as follows.

1. Computation: consider only the computation required by a function and if the computation power provided by an architectural unit such as a processor can match the requirements.

2. Communication: consider the overall performance such as response time, throughput, required by a function and if that provided by a chosen interconnection architecture, such as AMBA bus, or mesh NoC (network-on-chip) can match the requirements.
**Abstraction & Successive Refinement**

- Function/Architecture formal trade-off is applied for mapping function onto architecture
- Co-design and trade-off evaluation from the highest level down to the lower levels
- Successive refinement to add details to the earlier abstraction level

Refinement is the process of adding details to the currently developed or designed architectural components. A design is developed using an iterative process, which consists of architectural abstractions and function refinements. These two actions must meet somewhere, that is, there is a trade-off to be achieved such that a feasible system satisfying all user-given constraints is constructed. A system always consists of parts that are abstract and parts that are already refined. The abstract parts need to be refined and bound to some architectural parts. A match is to be found between:

1. the functional parts that need to be “refined,” and
2. the architectural parts whose functionalities can be “abstracted.”

**Target Architectural Exploration and Estimation**

- Synthesized target architecture is analyzed and estimated
- Architecture constraints are derived
- An adequate model of target architecture is built

Target architecture is synthesized through a series of estimation, analysis, and refinement. Traditional design space exploration techniques can be used here. The choices among the types and the number of each component are made at this stage. For example, the bandwidth of a bus channel, the number of ports, the interconnection capabilities of a network interface, the amount of each type of memory modules, etc. are all considered at this stage. The goal of this stage is to find an optimal architecture within the user-given bounds.
Architectural Exploration in POLIS

This shows how FAC architectural explorations are done in POLIS. There are 3 levels: functional, mapping, and architectural.

1. Functional Level: Users specify system requirements at the functional level by describing two types of models: functional model and architecture model. Each of the models are then verified for correctness.

2. Mapping Level: Functional model is then mapped to architecture model using a data flow analysis framework. The mapping is then verified for performance requirements.

3. Architectural Level: The mapping is then refined into hardware-software micro-architecture, which can then be optimized and verified at the micro-architecture level.

Reactive System Cosynthesis

A reactive system can be co-synthesized by decomposing the design requirements into a set of EFSM (Extended Finite State Machine) which are then mapped to a set of CDFG (Control Data Flow Graph) by scheduling and estimation. Later, these CDFG are then synthesized into hardware and software modules after partitioning and estimation. The EFSM is more suitable for describing the functionalities of a control-dominated design application. The CDFG is more suitable for implementation into hardware and software. An example will be given in later slides.
Reactive System Cosynthesis

CDFG is suitable for describing EFSM reactive behavior but

• Transition Function is suitable for describing EFSM
  • Reactive behavior of hardware or software breaks across re-invocations
  • Data is “abstracted” for easier verification
  • People can leverage on hardware synthesis flows
  • For software it makes ESTIMATION easier
  • but not for performing data flow optimization …
  • Need a different view

The slide shows an example, where the EFSM consists of 3 states S0, S1, and S2. The corresponding CDFG is shown on the right part of the slide. There are 3 cases based on the 3 states. Corresponding assignments are shown on the different paths in the CDFG. CDFG is more suitable for expressing reactive behavior but some control flow is hidden and data cannot be propagated.

Data Flow Optimization

An EFSM representation can be optimized by term rewriting and value propagation as shown in the slide. The data value of variable “a” can be propagated from S0 to S1. Since “a” is not changed between S0 and S1, its value can be directly set as 6 in S1 and then emitted in S2. This saves an adder for incrementing the variable value.
We will discuss the design representations used in FAC in this section of the slides.

An Intermediate Design Representation (IDR) must be used for allowing different types of inputs and for functional mapping into a single final architecture. The IDR must at the same time be unbiased towards either hardware or software. The abstract codesign flow will be made concrete in future slides.
After functional decomposition, the IDR is used to represent an architecture independent model of the system under development. Architecture independent optimizations are performed on the IDR. Then, architectural constraints are considered for architecture dependent optimizations giving SW and HW partitions that can be further synthesized and integrated. More on optimizations will be discussed in later slides.

The POLIS system is an implementation for FAC.

The function model can be composed from Esterel language specifications, reactive VHDL, and a set of EFSM. Esterel is a synchronous language that has been used heavily in Europe for the design of hardware and reactive systems. It is well-supported by many academic as well as commercial tools.

The architecture model is composed from a network of CFM (Codesign FSM), called SHIFT (Software-Hardware Intermediate FormaT). SHIFT will be discussed in more details later.
**CFSM**

- Includes
  - Finite state machine
  - Data computation
  - Locally synchronous behavior
  - Globally asynchronous behavior
- Semantics: GALS (Globally Asynchronous and Locally Synchronous communication model)

A CFSM is basically a finite state machine along with data computations and the GALS semantics. GALS is an important notion in developing complex architectures such as distributed real-time embedded systems including sensor networks, etc. GALS was proposed long ago, but its actual use is only now appreciated with the advent of complex systems such as multiprocessor SoC, distributed sensor networks, pervasive systems, etc. The two-level synchronization approach makes the design of such systems easier and precise.

---

**CFSM Network MOC**

A network of CFSM can collaborate as a model of computation along with communication events. The GALS concept is well implemented in this architecture. For example, the system-level channels are basically asynchronous, however each CFSM is locally synchronous, that is, it is driven by some clock. Events such as F, G, C, A, and B are used for communication among the different CFSM processes.
**Intermediate Design Representation (IDR)**

- Most current optimization and synthesis are performed at the low abstraction level of a DAG (Direct Acyclic Graph).
- Function Flow Graph (FFG) is an IDR having the notion of I/O semantics.
- Textual interchange format of FFG is called C-Like Intermediate Format (CLIF).
- FFG is generated from an EFSM description and can be in a Tree Form or a DAG Form.

The IDR in FAC is a Function-Flow Graph (FFG), which has I/O semantics and can be textually written in a C-like Intermediate Format (CLIF). FFG or CLIF can be generated from an EFSM description. There are two forms of FFG, namely tree form and DAG form. FFG is a unified model that is unbiased towards either hardware or software. Further, FFG can be optimized in several ways using conventional software analysis approaches. FFG can also be annotated with architectural attributes for implementation. Finally, the attributed FFG is optimized and then mapped to either hardware circuits or software code automatically.

**At the architecture independent phase, I/O semantics is used for optimizations, while at the architecture dependent phase EFSM semantics is used.**

Attributed or Architecture FFG (AFFG) is the annotated version of FFG which is derived by adding architecture attributes to the original FFG. Attributes may be derived from some hardware or software libraries and by profiling.

Refinements and optimizations can be two-fold, architecture independent that work on FFG and architecture dependent that work result in AFFG and its mapping to SW and HW.
**FFG/CLIF**

- Develop Function Flow Graph (FFG) / C-Like Intermediate Format (CLIF)
  - Able to capture EFSM
  - Suitable for control and data flow analysis

---

**Function Flow Graph (FFG)**

- FFG is a triple $G = (V, E, N_0)$ where
  - $V$ is a finite set of nodes
  - $E \subseteq V \times V$, a subset of $V \times V$; $(x, y)$ is an edge from $x$ to $y$ where $x \in \text{Pred}(y)$, the set of predecessor nodes of $y$.
  - $N_0 \in V$ is the start node corresponding to the EFSM initial state.
  - An unordered set of operations is associated with each node $N$.
  - Operations consist of TESTs performed on the EFSM inputs and internal variables, and ASSIGNs of computations on the input alphabet (inputs/internal variables) to the EFSM output alphabet (outputs and internal (state) variables).

This is the formal definition of FFG. It is a directed acyclic graph (DAG), with the following features.

1. $V$ is a finite set of nodes that represent some sequence of computations,
2. $E$ is a set of edges that represent predecessor/successor relationships among the nodes, that is, the precedence between two sequence of computations,
3. $N_0$ is an initial node.
4. Each node is associated with a set of unordered operations.
5. Operations consist of TEST and ASSIGN. TEST checks the value of EFSM inputs and internal variables. ASSIGN defines the value of EFSM output alphabets after some computations on the EFSM input alphabets.

---

EFSM is captured in FFG (which have I/O semantics) and then data flow/control optimizations are performed to obtain optimized FFG. CDFG is the final result mapped from the optimized FFG.
**C-Like Intermediate Format (CLIF)**

- Import/Export Function Flow Graph (FFG)
- “Un-ordered” list of TEST and ASSIGN operations
  - \([\text{if (condition)}] \text{ goto label} \)
  - \( \text{dest} = \text{op(src)} \)
    - \( \text{op} = \{ \text{not, minus, } \ldots \} \)
  - \( \text{dest} = \text{src1 op src2} \)
    - \( \text{op} = \{ +, *, /, ||, &&, |, & \ldots \} \)
  - \( \text{dest} = \text{func(arg1, arg2, } \ldots \) \)

*Preserving I/O Semantics*

```c
input inp;
output outp;
int a = 0;
int CONST_0 = 0;
int T11 = 0;
int T13 = 0;

S1:
goto S2;
S2:
a = inp;
T13 = a + 1 CONST_0;
T11 = a + a;
outp = T11;
goto S3;
S3:
outp = T13;
goto S3;
```

The I/O semantics of CLIF is shown here. There are three states S1, S2, and S3, with transitions from S1 to S2, from S2 to S3, and a loop from S3 to S3. inp is an input and outp is an output.
**FFG / CLIF Example**

Function Flow Graph

CLIF Textual Representation

S1:

\[
\begin{align*}
    x &= x + y; \\
    x &= x + y; \\
    a &= b + c; \\
    a &= b + c; \\
    cond1 &= \text{if } y \text{ equal } \text{cst1}; \\
    cond2 &= \text{cond1}; \\
    \text{if } (\text{cond2}) \text{ goto S1L0} \\
    \text{output} &= a; \\
    \text{goto } S1; /* Loop */ \\
\end{align*}
\]

S1L0:

\[
\begin{align*}
    \text{output} &= b; \\
    \text{goto } S1; \\
\end{align*}
\]

Legend: constant, output flow, dead operation

S# = State, S#L# = Label in State S#

S# = state

S#L# = labels (branches) within state

An FFG and its corresponding CLIF representation are shown here for a simple example. It is a rather straightforward mapping.

This is the tree form of FFG, where clusters of nodes form a state. This gives a hierarchical structure to FFG.
In this part of the lecture, we will focus on FAC optimizations that can reduce the complexity of FFG and AFFG models so that the hardware and software can be successfully synthesized.

FAC Optimizations

- **Architecture-independent phase**
  - Task function is considered solely and control data flow analysis is performed
  - Removing redundant information and computations

- **Architecture-dependent phase**
  - Rely on architectural information to perform additional guided optimizations tuned to the target platform

In FAC, optimizations are classified into:

1. architecture-independent optimizations that focus on control and data flow analysis to remove redundant information and computation, and
2. architecture-dependent optimizations that perform target platform related optimizations
**Function Optimization**

- Architecture-independent optimization objective:
  - Eliminate redundant information in the FFG.
  - Represent the information in an optimized FFG that has a minimal number of nodes and associated operations.

Architecture-independent optimizations are applied to FFG to eliminate redundancies such that it has a fewer number of nodes and associated operations.

---

**FFG Optimization Algorithm**

- FFG Optimization algorithm(G)
  ```plaintext
  begin
  while changes to FFG do
  Variable Definition and Uses
  FFG Build
  Reachability Analysis
  Normalization
  Available Elimination
  False Branch Pruning
  Copy Propagation
  Dead Operation Elimination
  end while
  end
  ```

This is the FFG optimization algorithm. Most of these techniques come from classical software engineering. Refer to an SE textbook for details on these functional optimization techniques. Some of them are introduced briefly as follows.

1. **Variable Definition and Uses**: Based on the “definition” and “use” of variables, we can eliminate considering the value of some variables during some parts of a computation, we can also propagate the value of variables so that redundant information is eliminated.

2. **Reachability Analysis**: A reachability graph is constructed for an FFG and unreachable states can thus be eliminated, making the FFG more compact and smaller in size.

3. **Normalization**:

4. **Available Elimination (AE)**: AE is a technique that is used to collect all the expressions that have been computed at each node and can be propagated to successor nodes. This technique prevents repeated computation of the same expression. Details on this technique will be illustrated in later slides.

5. **False Branch Pruning**: Some branches of a choice (switch-case) will never be taken in some contexts and can thus be eliminated.

6. **Copy Propagation**: A copy of variable and expression values can be propagated to eliminate redundant computations

7. **Dead Operation Elimination**: Operations which are never executed can be eliminated.
Optimization Approach

- Develop optimizer for FFG (CLIF) intermediate design representation
- Goal: Optimize for speed, and size by reducing
  - ASSIGN operations
  - TEST operations
  - variables
- Reach goal by solving sequence of data flow problems for analysis and information gathering using an underlying Data Flow Analysis (DFA) framework
- Optimize by information redundancy elimination

The two other main methods (besides iterative) are:
- Elimination: analogous to Gaussian elimination, work on a restricted class of flow graphs (Graham-Wegman)
- Path algebra (Tarjan)

The 3 properties shown are intended to address:
1) Feasibility
2) Optimality
3) Speed of convergence to optimal solution…

Optimality really coming from “restrictions” on the input language mainly no aliasing etc., as well as properties of the data flow problems solved (monotone, distributive…).

Kildall has developed data propagation algorithms in a general lattice theoretic framework ("Global expression optimization during compilation", Proc. ACM Conf. on PoPL, Oct. 1973)

Kildall was the first to formulate data flow problems in a framework that combines flow graph structure with lattice properties (semilattice with meet or join operators)

Framework 4-tuple: (flow graph, semilattice, class of functions, choice of particular defining equation (assignment map M))

Sample DFA Problem

Available Expressions Example

- Goal is to eliminate re-computations
  - Formulate Available Expressions Problem
    - Forward Flow (meet) Problem

Available Expressions (AE) are generated on-the-fly for each node and then reused if required to eliminate re-computations of similar expressions.

In the example, AE is initially empty for S1 and S2 since there are no computations done before those two nodes. After S1, we have AE={a+1} and after S2 we have AE={a+1, a+2}. The intersection of those two sets can be passed to S3, namely {a+1} as the propagated AE. However, since S3 destructs the value of a+1 by changing the value of variable a, thus a+1 is not propagated anymore beyond S3. There is a new computation, namely a+2 which can be propagated to nodes succeeding S3.
**Data Flow Problem Instance**

- A particular (problem) instance of a monotone data flow analysis framework is a pair \( I = (G, M) \) where \( M: N \rightarrow F \) is a function that maps each node \( N \) in \( V \) of FFG \( G \) to a function in \( F \) on the node label semi-lattice \( L \) of the framework \( D \).

This is the formal definition of an instance of monotone data flow analysis framework, which gives a mapping from an FFG to a semilattice. This framework can be used to perform data flow analysis and optimizations on the FFG based on the analysis results.

**Data Flow Analysis Framework**

- A monotone data flow analysis framework \( D = (L, \wedge, F) \) is used to manipulate the data flow information by interpreting the node labels on \( N \) in \( V \) of the FFG \( G \) as elements of an algebraic structure where
  - \( L \) is a bounded semilattice with meet \( \wedge \), and
  - \( F \) is a monotone function space associated with \( L \).

This is the more general definition of a data flow analysis framework. An algebraic structure called the semi-lattice is used to analyze data in the framework. This is the theory behind FFG optimization. (Students may just understand that it is based on some solid theory and not some ad hoc analysis.)
**Solving Data Flow Problems**

Data Flow Equations

\[
\text{In}(S3) = \bigcap_{P \in \{S1, S2\}} \text{Out}(P)
\]

\[
\text{Out}(S3) = (\text{In}(S3) \setminus \text{Kill}(S3)) \cup \text{Gen}(S3)
\]

The data flow equations given for the example in the slide show how a new AE set is computed from propagated AE and the newly performed computations. S3 gets the intersection of the AE from S1 and S2, labeled as In(S3). S3 propagates only those expressions that it has received (i.e. In(S3)) and which have not been destroy (i.e. In(S3) \setminus \text{Kill}(S3)), plus those newly computed by S3 (i.e. Gen(S3)).

Thus, the AE propagated by S3 is Out(S3) = \{a+2\}.

---

**Solving Data Flow Problems**

- Solve data flow problems using the *iterative method*
  - General: does not depend on the flow graph
  - Optimal for a *class of data flow problems*
  - Reaches fixpoint in polynomial time (O(n^2))

An iterative method is used for solving data flow problems such that fixpoints can be reached in polynomial time (O(n^2)).
FFG Optimization Algorithm

- Solve following problems in order to improve design:
  - Reaching Definitions and Uses
  - Normalization
  - Available Expression Computation
  - Copy Propagation, and Constant Folding
  - Reachability Analysis
  - False Branch Pruning

- Code Improvement techniques
  - Dead Operation Elimination
  - Computation sharing through normalization

This is the list of FFG optimization algorithms (refer to SE textbooks for details or refer to the notes of slide 32, where they were briefly explained).

Function/Architecture Codesign

A diagrammatic representation of FAC. After decomposition, an intermediate format such as FFG and AFFG are used to represent the function and architecture attributed function of a system. There is a tradeoff between function and architecture such that either function has to be degraded to meet architectural constraints or architecture upgraded to meet functional requirements. An example on the ATM VPN will illustrate this fact at the end of this set of slides.
**Function Architecture Optimizations**

- **Function/Architecture Representation:**
  
  - Attributed Function Flow Graph (AFFG) is used to represent architectural constraints impressed upon the functional behavior of an EFSM task.

AFFG = FFG + architectural constraints

AFFG is the attributed version of FFG after it has been mapped to some hardware-software system architecture, such as an AMBA-based SoC architecture. AFFG is architecture platform dependent and thus has more information for further optimizations based on the features or restrictions of the architecture.

This shows the transformation between one model and its optimized or attributed forms. EFSM is first converted into FFG (or CLIF textual format), which is optimized into OFFG. The architectural library is considered for attributing the FFG into AFFG, which is optimized and finally converted into the CDFG for hardware-software implementation. In POLIS, the final CDFG is the SHIFT model. Details can be found in later slides.
This is the tree form of AFFG, which corresponds to the tree form of FFG, giving a hierarchical structure to the graph.

**Architecture Dependent Optimization Objective**

- Optimize the AFFG task representation for speed of execution and size given a set of architectural constrains
- Size: area of hardware, code size of software

AFFG can be further optimized for speed of execution and size. For hardware, it is the area that is reduced and for software, it is code size which represents the amount of memory required and thus the cost for the system.
Motivating Example

This is a motivating example for optimization of AFFG. As you can see from the example, there are lots of “a+b” computations which are needlessly repeated. The redundant computations may be eliminated by analyzing the data flow such as using the AE technique.

Cost-guided Relaxed Operation Motion (ROM)

- For performing safe and operation from heavily executed portions of a design task to less visited segments
- Relaxed-Operation-Motion (ROM):

  \begin{verbatim}
  begin
  Data Flow and Control Optimization
  Reverse Sweep (dead operation addition, Normalization and available operation elimination, dead operation elimination)
  Forward Sweep (optional, minimize the lifetime)
  Final Optimization Pass
  end
  \end{verbatim}

ROM is the optimization procedure for AFFG. The algorithm consists of three steps are shown in the slide.

1. Reverse Sweep tries to optimize the AFFG by adding dead operations so that the AE can be reused and dead operations are also eliminated.
2. Forward Sweep is optional and tries to minimize the lifetime of a variable by moving the “definition” of a variable as close to its “use” as possible. A reduced lifetime represents more efficient use of memory by reducing the time the value of a variable needs to be stored for later use.
3. Final Optimization Pass tries to repeat the above steps as long as changes are made to the AFFG.
Cost estimation is required for ROM optimizations, which is performed by profiling through an inference engine. For example, the code size and variable lifetime are estimated only through profiling.

Micro-architecture optimizations result in optimized AFFG. For example, instruction selection and operator strength reduction can be performed on the function and the architecture, respectively, for optimizing AFFG.
**Operator Strength Reduction**

\[
\begin{align*}
\text{t1} &= 3 \times b \\
\text{t2} &= \text{t1} + a \\
x &= \text{t2}
\end{align*}
\]

\[
\begin{align*}
\text{expr1} &= b + b; \\
\text{t1} &= \text{expr1} + b; \\
\text{t2} &= \text{t1} + a; \\
x &= \text{t2};
\end{align*}
\]

Reducing the multiplication operator

This shows how multiplication operators can be reduced to the weaker and cheaper addition operators. The example shows that an expensive multiplication can be reduced to 3 cheap additions.

---

**Architectural Optimization**

- **Abstract Target Platform**
  - Macro-architectures of the HW or SW system design tasks

- **CFSM (Co-design FSM): FSM with reactive behavior**
  - A reactive block
  - A set of combinational data-flow functions

- **Software Hardware Intermediate Format (SHIFT)**
  - \( \text{SHIFT} = \text{CFSMs} \times \text{Functions} \)

An abstract target platform is used to represent the macro-architectures of hardware and software design tasks.

CFSM (Co-design FSM) is an FSM with reactive behavior, that is, a reactive block with a set of combinational dataflow functions. A CFSM is a basic mapping unit for hardware and software tasks.

SHIFT (Software Hardware Intermediate Format) is a network of CFSMs along with functions. SHIFT is used to represent the architecture of a system.
Macro-architectural organization of the mapped architecture. It is assumed that there is only ONE software partition which includes the RTOS running a microprocessor. The hardware can be partitioned into more than one partition. This assumption is valid for a single processor system, which has only one software running. The hardware and software partitions communicate through bus interface.

Architectural Organization of a Single CFSM Task

This shows the architectural organization of a single CFSM task. It is basically a CDFG in a state.
This shows the hardware implementation of a CFSM. The reactive controller is used to control the outputs based on the inputs. This hardware implementation corresponds to the CFSM in the previous slide.

---

**CFSM Network Architecture**

- Software Hardware Intermediate FormaT (SHIFT) for describing a network of CFSMs
- It is a hierarchical netlist of
  - Co-design finite state machine
  - Functions: state-less arithmetic, Boolean, or user-defined operations

SHIFT = CFSM + functions
SHIFT is a hierarchical netlist of CFSM with functions that are state-less arithmetic just like combinational circuits.
SHIFT: CFSMs + Functions

An example of SHIFT: 2 HW partitions and 1 SW partition

FAC assumes there is only 1 final SW partition corresponding to a single processor system, FAC can also be extended to multiprocessor systems.

In this example, CFSM1 and CFSM2 are in hardware partition 1 and CFSM3 is in hardware partition 2. The results of CFSM1 can be propagated to CFSM3 through an e1 channel. Similarly, CFSM4, CFSM5, CFSM6, CFSM7 are all in a single software partition. The hardware and software communicate through two channels e2 and e3. Ports are used to abstract the different communication interfaces either among partitions or among CFSMs within a single partition. For example, port3 is a bi-directional port for communication between the hardware CFSM2 and the software CFSM7.

Architectural Modeling

- Using an AUXiliary specification (AUX)
- AUX can describe the following information
  - Signal and variable type-related information
  - Definition of the value of constants
  - Creation of hierarchical netlist, instantiating and interconnecting the CFSMs described in SHIFT

Auxiliary specifications include other details of a SHIFT architecture such as variable types, constant values, etc. Auxiliary specifications also specify the hierarchical netlist among CFSMs in a SHIFT architecture.
Mapping AFFG onto SHIFT

- Synthesis through mapping AFFG onto SHIFT and AUX (Auxiliary Specification)
- Decompose each AFFG task behavior into a single reactive control part, and a set of data-path functions.

Mapping AFFG onto SHIFT Algorithm (G, AUX)

begin
    foreach state s belong to G do
        build_trel (s.trel, s, s.start_node, G, AUX);
    end foreach
end

This shows how an AFFG is mapped onto SHIFT and AUX. Each AFFG task is decomposed into a single reactive control part and a set of data-path functions. The transition relationship (trel) is built for each state s in the FFG G.

Architecture Dependent Optimizations

- Additional architecture information leads to an increased level of macro- (or micro-) architectural optimization
- Examples of macro-arch. Optimization
  - Multiplexing computation Inputs
  - Function sharing
- Example of micro-arch. Optimization
  - Data Type Optimization

Architecture dependent optimizations include macro and micro level ones. Examples for macro-architecture optimization include multiplexing computation inputs and sharing functions. Details are in later slides. Example for micro-architecture optimization include data type optimization.
Distributing the Reactive Controller

Move some of the control into data path as an ITE assign expression

This shows how a complex reactive controller can be simplified by moving some control into the data path and using a multiplexer. The diagram in the slide shows how some control can be implemented as ITE (if-then-else) in the data path and thus the reactive controller becomes simpler for implementation. This is a type of macro-architecture optimization.

Multiplexing Inputs

This shows how inputs can be multiplexed so that a reactive controller can be simpler. In the above example, c may take different values when T=b+c is computed. To account for the different values, a 2x1 multiplexer can be used for the input c. In this way, only one adder is required, instead of the original two adders.
Micro-Architectural Optimization

- Available Expressions cannot eliminate $T_2$
- But if variables are registered (additional architectural information) we can share $T_1$ and $T_2$

S1:
- $T_1 = a + b$
- $x = T_1$
- $a = c$

S2:
- $T_2 = a + b$
- $Out = T(a+b)$
- $emit(Out)$

Micro-architectural information allow better optimizations. In the above example, if variables are stored in registers, then only one adder, instead of two, is necessary to implement the above FFG. This is a type of micro-architecture optimization.

Hardware/Software Cosynthesis and Estimation

This part introduced cosynthesis and estimation techniques in FAC.
This shows the overall cosynthesis flow consisting of all model transformations, optimizations, and the final SHIFT architecture that can be implemented in hardware and software. This flow is integrated in the POLIS tool. Software can be compiled into object code and hardware can be synthesized into netlist.

A FAC codesign flow is shown in the slide. Graphical EFSM, Esterel language specifications, and reactive VHDL are the inputs to FAC along with architectural constraints. After functional decomposition, a set of FFGs (function flow graphs) are generated which then undergo functional optimizations under the SHIFT architectural constraints resulting in Attributed FFGs (AFFGs). Macro-level and micro-level optimizations are then applied to AFFG and finally the full system architecture.
Some essential phases and parts of the POLIS codesign environment are shown here. Software synthesis is based on S-graph (software graphs) which can be used to automatically generate software code. The synchronous Esterel language is used for system requirement input, which can be formally validated. The internal representation is a CFSM network. Validation can be performed in three ways: high-level cosimulation, formal verification, and rapid prototyping. Partitioning and scheduling are performed on the CFSM.

This is the POLIS codesign flow that implements FAC. CFSM is the core model used for hardware-software cosynthesis. For software, S-graph is used for profiling and synthesis.
Hardware/Software Co-Synthesis

- Functional GALS CFSM model for hardware and software
  - initially unbounded delays refined after architecture mapping
- Automatic synthesis of:
  - Hardware
  - Software
  - Interfaces
  - RTOS

Hardware software cosynthesis in FAC / POLIS includes the functional GALS (globally asynchronous locally synchronous) CFSM for hardware and software implementations and the automatic synthesis of hardware, software, interfaces, and RTOS.

RTOS Synthesis and Evaluation in POLIS

This shows the flow of RTOS synthesis and evaluation in POLIS. S-graph can be used to provide communication mechanisms among CFSMs implemented in software and between the OS and the hardware partitions. It can also be used to schedule the execution of software tasks.
Estimation for CDFG Synthesis

This shows how software costs are estimated for a CDFG in FAC/POLIS. The costs are in terms of clock cycles on a 68HC11 target processor using the Introl compiler for a simple CFSM. The different weights (number of clock cycles) on the branches from a branch node represent the different number of clock cycles required for different branch cases. The CDFG can then be used to schedule/synthesize the software.

Estimation-Based Co-simulation

Cosimulation requires cost and performance estimations for both software and hardware. Hardware-software cosimulation evaluates the performance of a hardware-software partition and suggests some trade-offs for improving the performance.
Co-simulation Approach

- Fills the “validation gap” between fast and slow models
  - Performs performance simulation based on software and hardware timing estimates
- Outputs behavioral VHDL code
  - Generated from CDFG describing EFSM reactive function
  - Annotated with clock cycles required on target processors
- Can incorporate VHDL models of pre-existing components

Hardware logic simulation models are intrinsically slow while software execution models such as ISS (instruction set simulator) can be very fast. Models are different abstractions (such as FIFO channels and buffer implementations) also simulate at different speeds. There is a large gap between how far the models can execute. Hardware-software cosimulation tries to fill this gap through memory models and various kinds of optimizations.

Simulation is done based on software and hardware timing estimates and not the real hardware or software modules themselves.

In FAC/POLIS, behavioral VHDL code is generated from CDFG describing EFSM reactive function and then annotated with clock cycles required on target processors. The methodology can also reuse VHDL models of existing components.

---

Future Work

- Models of mixed hardware, software, RTOS and interfaces
- Mimics the RTOS I/O monitoring and scheduling
  - Hardware CFSMs are concurrent
  - Only one software CFSM can be active at a time

- Architectural view instead of component view

This cosimulation approach consists of mixed modules in hardware, software, RTOS, and interfaces. Models are often at different abstraction levels, which increases the complexity of cosimulation and also necessitates the use of wrappers for interconnecting components at different abstraction levels.
Coverification Tools

- Mentor Graphics Seamless Co-Verification Environment (CVE)

This is a well-known and widely used cosimulation tool from Mentor Graphics called Seamless CVE (Co-Verification Environment). CVE supports integration and concurrent execution of one or more processor ISS (instruction set simulator) with a hardware logic simulator. CVE v5.0 also supports the execution of C models through the C Bridge. Software is simulated by supporting different processor instruction set simulators in different packages such as ARM 920T, Palm, (supported by CiC now). There are other PSP (processor support package) not currently available in CiC and must be purchased separately. As far as logic simulator is concerned, Mentor Graphics’ ModelSim is a popular one which is supported by CVE. CVE uses a coherent memory server to optimize the execution speeds of the different rated ISS and logic simulator.

Seamless CVE Performance Analysis

CVE v5.0 can display three types of profiling information during cosimulation, including the bus loading, the code profile (which function took what time), and memory transactions (to see where is the major bottleneck in hw/sw communication). A user after observing these information can try to improve the design for better performance, for example remove the bottleneck in communication.
References


These are the references for the materials in this set of slides for codesign. The FAC methodology is from the first reference book. Gajski’s book is mainly on high-level synthesis, but it is a very good reference book for all sorts of high-level design algorithms and methods.

Codesign Case Studies

- ATM Virtual Private Network
  - CSEL, Italy
  - Virtual IP library reuse
- Digital Camera SoC
  - Chapter 7, F. Vahid, T. Givargis, Embedded System Design
  - Simple camera for image capture, storage, and download
  - 4 design implementations

Two case studies will be discussed for hw/sw codesign as listed in the following.
(1) ATM (Asynchronous Transfer Mode) VPN (Virtual Private Network) node, and
(2) A digital camera SoC

The ATM example was developed by CSELT from Italy and the digital camera is an example from the Chapter 7 of Frank Vahid’s book on Embedded System Design. These two examples illustrate how hardware-software codesign must be practiced. The ATM example shows how POLIS is integrated with the reuse of IP from the VIP library. The camera example shows how different hardware-software partitions can be explored and finally how an SoC can be designed to meet all user-given requirements.
This case study is about the design of an ATM (asynchronous transfer mode) network VPN (virtual private network) node. It is developed by CSELT from Italy in collaboration with University of California, Berkeley, USA. Mainly, we will introduce POLIS, how VIP library is used with POLIS to design the ATM node hardware-software, and how POLIS must be extended to deal with real system design.

Outline
- The key aspects in the architectural evolution of switching nodes and the technological key issues.
- An example of a switching architecture is given as the state of art for an industrial product.
- Architectural and technological solutions to expand switching capacity using currently available technology.
- How the optical technology is used to enhance the performance of packet switching nodes.
- Conclusions.
This case study shows why IPs may be reused in hw/sw codesign for high design productivity and reliability.

THE POLIS EMBEDDED SYSTEM CO-DESIGN ENVIRONMENT

- HW-SW CO-DESIGN FOR CONTROL-DOMINATED REAL-TIME REACTIVE SYSTEMS
  - AUTOMOTIVE ENGINE CONTROL, COMMUNICATION PROTOCOLS, APPLIANCES, ...
- DESIGN METHODOLOGY
  - FORMAL SPECIFICATION: ESTEREL, FSMS
  - TRADE-OFF ANALYSIS, PROCESSOR SELECTION, DELAYED PARTITIONING
  - VERIFY PROPERTIES OF THE DESIGN
  - APPLY HW AND SW SYNTHESIS FOR FINAL IMPLEMENTATION
  - MAP INTO FLEXIBLE EMULATION BOARD FOR EMBEDDED VERIFICATION

This is a brief description of the POLIS codesign environment (as we discussed earlier in FAC). The design methodology is summarized here for introducing this example.
This is the POLIS codesign flow (as discussed earlier in FAC). This shows how ATM VPN is going to be designed.

This is the VIP library that can be used for DSP applications. All IPs are available in RTL VHDL code. The IPs mainly cover fast packet switching such as ATM, TCP/IP and video and multimedia functions. The features of VIP library modules are: medium architectural complexity, high usability, high programmability, and reasonable performance.
The goal of this case study is to show how an ATM VPN server can be designed using codesign technology and the VIP library.

This is the architecture of an ATM VPN server which takes an input from the ATM network at the speed of 155 Mbit/s and must output the cells at the same rate. In this ATM VPN server node, there is a congestion control algorithm called MSD (message selective discarding) which discards cells of an entire message if the buffer queues are full, otherwise the cells are queued up in their respective channels. A WFQ (weighted fair queuing) scheduler is used to schedule and dispatch the cells out to the ATM network based on a real-time sorter.
CRITICAL DESIGN ISSUES

- **TIGHT TIMING CONSTRAINTS**
  - Functions to be performed within a cell time slot (2.72 μs for a 155 Mbps flow) are:
    - Process one input cell
    - Process one output cell
    - Perform management tasks (if any)

- **FREQUENT ACCESS TO MEMORY TABLES**
  - That store routing information for each connection and state information for each queue

These are the design constraints. For 155 Mbps flow, we need to process one cell within 2.72 microseconds which is very short time, so we need both hardware and software to handle all those data cells. There is frequent access to memory tables that store routing information for each connection and state information for each queue. Thus, memory access must be designed efficiently.

DESIGN IMPLEMENTATION

**DATA PATH:**
- 7 VIP LIBRARY™ MODULES
- 2 COMMERCIAL MEMORIES
- SOME CUSTOM LOGIC (PROTOCOL TRANSLATORS)

**CONTROL UNIT:**
- 25 CFSMs

- VIP™ LIBRARY MODULES
- HW/SW CODESIGN MODULES
- COMMERCIAL MEMORIES

The different colors differentiate which modules were implemented using the VIP library, which modules were bought from the market, and which modules were codesigned. There are 7 VIP modules, 2 commercial memory modules, and 25 CFSMs that need to be implemented in hw and sw.
Within the full ATM VPN data flow, on the right we have the architecture of the MSD algorithm module, which is going to be codesigned. Supervisor will be in software.

This is the hardware schematic for an ATM VPN server. There are three arbiters, a real-time sorter, a supervisor, a collision detector, and the MSD technique.
This slide shows that we need to design interfaces between VIP module and our codesigned module because different protocols are used by them. We also need to develop the sw/hw interface between POLIS sw codesign module and a VIP library module. Mainly protocol translators must be designed and tested by integration. High-level synthesis algorithm from Gajski’s book may be used here for protocol translator design. The hardware-software interface design method can also be looked up in Gajski’s book.

This slide shows why the processor choice was changed to Motorola PowerPC because the MIPS 3000 RISC CPU could not give the desired performance.

On the right, we have two different hardware/software partitions. The MSD technique, the cell extraction, arbiter #2, and arbiter #3 are implemented differently in the two partitions. The virtual clock scheduler, the real-time sorter, and arbiter #1 need high performance, hence they are all implemented in hardware for both the partitions. The supervisor does not have real-time requirements hence it is implemented in software for both partitions.
DESIGN VALIDATION

- VHDL co-simulation of the complete design
  - Co-design module code generated by POLIS
- Server code: ~ 14,000 lines
  - VIP LIBRARY™ modules: ~ 7,000 lines
  - HW/SW co-design modules: ~ 6,700 lines
  - IP integration modules: ~ 300 lines
- Test bench code: ~ 2,000 lines
  - ATM cell flow generation
  - ATM cell flow analysis
  - Co-design protocol adapters

This shows the size of the HDL code for both the design and the testbench. As we can see, reusing libraries save us around half of the code to be written. But, of course, we still need to write the testbench (2000 lines of HDL). The testbench consists mainly of ATM cell flow generation, analysis, and codesign protocol adapters. The codesign modules were automatically generated by POLIS.

CONTROL UNIT MAPPING RESULTS

<table>
<thead>
<tr>
<th>MODULE</th>
<th>FFs</th>
<th>CLBs</th>
<th>I/Os</th>
<th>GATES</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSD TECHNIQUE</td>
<td>45</td>
<td>166</td>
<td>204</td>
<td>1,009</td>
</tr>
<tr>
<td>CELL EXTRACTION</td>
<td>30</td>
<td>86</td>
<td>56</td>
<td>386</td>
</tr>
<tr>
<td>VIRTUAL CLOCK SCHEDULER</td>
<td>77</td>
<td>74</td>
<td>95</td>
<td>1,286</td>
</tr>
<tr>
<td>REAL TIME SORTER</td>
<td>261</td>
<td>753</td>
<td>52</td>
<td>10,064</td>
</tr>
<tr>
<td>ARBITER #1</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>115</td>
</tr>
<tr>
<td>ARBITER #2</td>
<td>10</td>
<td>7</td>
<td>10</td>
<td>127</td>
</tr>
<tr>
<td>ARBITER #3</td>
<td>10</td>
<td>7</td>
<td>17</td>
<td>153</td>
</tr>
<tr>
<td>LQM INTERFACE</td>
<td>28</td>
<td>39</td>
<td>27</td>
<td>906</td>
</tr>
<tr>
<td>PARTITION I</td>
<td>409</td>
<td>892</td>
<td>120</td>
<td>13,224</td>
</tr>
<tr>
<td>PARTITION II</td>
<td>443</td>
<td>961</td>
<td>256</td>
<td>14,228</td>
</tr>
</tbody>
</table>

This table shows the hardware features of the different modules found in the control path and the hardware sizes of the two partitions. The listed hardware features include FFs (Flip Flops), CLBs (Control Logic Blocks), I/Os (Input/Output blocks), and gates. The first partition is smaller in gate count than the second partition.
This table shows the hardware features of the modules that are found in the data path of the ATM VPN server node architecture. The data path is the same for the two partitions because codesign is performed only on the control path modules.

**WHAT DO WE NEED FROM POLIS?**

- **IMPROVED RTOS SCHEDULING POLICIES**
  - AVAILABLE NOW:
    - ROUND ROBIN
    - STATIC PRIORITY
  - NEEDED:
    - QUASI-STATIC SCHEDULING POLICY

- **BETTER MEMORY INTERFACE MECHANISMS**
  - AVAILABLE NOW:
    - EVENT BASED (RETURN TO THE RTOS ON EVENTS GENERATED BY MEMORY READ/WRITE OPERATIONS)
  - NEEDED:
    - FUNCTION BASED (NO RETURN TO THE RTOS ON EVENTS GENERATED BY MEMORY READ/WRITE OPERATIONS)

So, does POLIS satisfy our requirements for a codesign/cosimulation environment that can reuse IP modules? There is a lack of static scheduling techniques and memory interface mechanisms currently in POLIS.
WHAT ELSE DO WE NEED FROM POLIS?

MOST WANTED: EVENT OPTIMIZATION

- EVENT DEFINITION
- INTER-MODULE COMMUNICATION PRIMITIVE
  BUT:
  - NOT ALL OF THE ABOVE PRIMITIVES ARE ACTUALLY NECESSARY
  - UNNECESSARY INTER-MODULE COMMUNICATION LOWERS PERFORMANCE

SYNTHESIZABLE RTL OUTPUT

- SYNTHESIZABLE OUTPUT FORMAT USED: XNF
- PROBLEM: COMPLEX OPERATORS ARE TRANSLATED INTO EQUATIONS
  - DIFFICULT TO OPTIMIZE
  - CANNOT USE SPECIALIZED HW (adders, comparators...)

Also, there is a lack of event optimization to enhance performance. Further, synthesizable RTL output is also needed from POLIS.

ATM EXAMPLE CONCLUSIONS

- HW/SW CODESIGN TOOLS PROVED HELPFUL IN REDUCING DESIGN TIME AND ERRORS
  - CODESIGN TIME = 8 MAN MONTHS
  - STANDARD DESIGN TIME = 3 MAN YEARS

- POLIS REQUIRES IMPROVEMENTS TO FIT INDUSTRIAL TELECOM DESIGN NEEDS

- EVENT OPTIMIZATION + MEMORY ACCESS + SCHEDULING POLICY

- EASY IP INTEGRATION IN THE POLIS DESIGN FLOW
  - FURTHER IMPROVEMENTS IN DESIGN TIME AND RELIABILITY

The ATM case study was developed in 8 man months instead of the 3 man years that would have been required without codesign/cosimulation environment such as POLIS. This case study showed how IP reuse needs integration techniques that may not be provided by a system-level design tool and must be hand-crafted.
This is Chapter 7 of Frank Vahid’s book “Embedded Systems Design: A Unified Hardware/Software Introduction”.

The digital camera SoC is used an illustration example for the hardware-software codesign part of this course.

It shows how different design choices can be made by trade-off between hardware and software. It also shows how a digital camera SoC may be designed.

The example is a simple digital camera SoC, with image capture, storage, and transfer capabilities. We will discuss the system requirements from a user’s point of view and then give four implementation choices starting from a pure software one. Since the performance requirement is a maximum time interval of one second between two presses of the camera shutter (image captures), some hardware is introduced gradually as we improve the performance. Software-based approximations such as fixed point representation is also used to enhance performance in one of the design architectures.
**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.

The main functions of the simple digital camera are as follows.

(a) Captures images,
(b) Stores images in digital format, and
(c) Downloads images to PC.

The full set of camera functions could be designed on a single chip only recently due to the advance in fabrication technology resulting in the possibility of SoC designs.

The camera example is a simple one. More complex functions such as image processing also exists in real digital cameras and can be incorporated into this example.

**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

The details of the two key tasks, namely image capture and download, are given here. A CCD device is used for digital image capture and it is archived in internal memory. The image must be downloadable to a PC through some serial communication interface.
**Charge-coupled device (CCD)**

- Special sensor that captures an image
- Light-sensitive silicon solid-state device composed of many cells

When exposed to light, each cell becomes electrically charged. This charge can then be converted to an 8-bit value where 0 represents no exposure while 255 represents very intense exposure of that cell to light.

Some of the columns are covered with a black strip of paint. The light-intensity of these pixels is used for zero-bias adjustments of all the cells.

The electromechanical shutter is activated to expose the cells to light for a brief moment.

The electronic circuitry, when commanded, discharges the cells, activates the electromechanical shutter, and then reads the 8-bit charge value of each cell.

These values can be clocked out of the CCD by external logic through a standard parallel bus interface.

The CCD is a light-sensitive silicon solid-state device composed of many cells that can be used to capture digital images. The details of the CCD are as described in the slide. Each cell is charged when exposed to light and this charge is then converted into an 8-bit digital format. Due to manufacturing instability, there are offsets introduced for cells of different rows. This is compensated for by covering two or more columns of the CCD with a black strip of paint such that the cells of those columns are always hidden from light. Details on this “zero-biasing” procedure will be given in later slides. There is electronic circuitry for reading the charges from each cell, column-wise, such that the image can be stored in memory.

**Zero-bias error**

- Manufacturing errors cause cells to measure slightly above or below actual light intensity
- Error typically same across columns, but different across rows
- Some of left most columns blocked by black paint to detect zero-bias error
  - Reading of other than 0 in blocked cells is zero-bias error
  - Each row is corrected by subtracting the average error found in blocked cells for that row

The zero-bias error is caused by manufacturing instability that causes cells to register some charge even if they are not exposed to any light at all. The zero-bias error is the same for cells of the same row, but different for cells of different rows. Thus, the error can be corrected by covering some of the left most columns with black paint so that light is blocked from those cells and then the charge registered by the blocked out cells is used compensate for the error. An example of an 8x8 image before and after zero-bias adjustment is shown in the slide. The zero bias adjustment is calculated as an average of the charges in the blacked out cells of a particular row. This adjustment is then deducted from the charges of the other cells.
Images are captured and compressed before archiving them. The standard format is JPEG, which is also very popularly used. There are several operation modes in JPEG, of which the mode used in this example is high compression ratios using DCT. Image is processed block by block where each block is 8 x 8 pixels large. There are three steps performed on each block, including DCT, quantization, and Huffman encoding.

It must be noted here that for a color image, some kind of color filter such as Bayer’s Color Filter Array (CFA) must be used before the 8 x 8 pixels is compressed and the image color must also be reconstructed from the 8 x 8 pixels.
Quantization step

- Achieve high compression ratio by reducing image quality
  - Reduce bit precision of encoded data
  - Fewer bits needed for encoding
  - One way is to divide all values by a factor of 2
    - Simple right shifts can do this
  - Dequantization would reverse process for decompression

After being decoded using DCT

This is the quantization step that is performed after DCT in JPEG. The main purpose of quantization is to reduce the amount of memory space required to store the image. Larger values require more space, hence the values are reduced (quantized) by some factor. In the example, each value in the 8 x 8 DCT-encoded pixels is divided by 8 to obtain the right hand side smaller valued pixel array. Fewer bits are thus required to store the right hand side quantized array than the original left hand side one.

Huffman encoding step

- Serialize 8 x 8 block of pixels
  - Values are converted into single list using zigzag pattern
- Perform Huffman encoding
  - More frequently occurring pixels assigned short binary code
  - Longer binary codes left for less frequently occurring pixels
- Each pixel in serial list converted to Huffman encoded values
  - Much shorter list, thus compression

This is the Huffman encoding step which is performed after quantization in JPEG. An 8 x 8 pixel array captured by the electronic circuitry in a CCD is first transformed by DCT, then quantized, and finally archived using the Huffman encoding method. Huffman encoding is a lossless encoding method, which adopts a zigzag pattern of storing the 8 x 8 pixel array, starting from the upper-left corner to the lower-right corner (see image in slide). The details of how Huffman encoding is performed are in the next slide.
Huffman encoding example

- Pixel frequencies on left
  - Pixel value -1 occurs 15 times
  - Pixel value 14 occurs 1 time
- Build Huffman tree from bottom up
  - Create one leaf node for each pixel value and assign frequency as node’s value
  - Create an internal node by joining any two nodes whose sum is a minimal value
  - This sum is internal node’s value
    Repeat until complete binary tree
- Traverse tree from root to leaf to obtain binary code for leaf’s pixel value
  - Append 0 for left traversal, 1 for right traversal
- Huffman encoding is reversible
  - No code is a prefix of another code

The Huffman encoding uses the shortest string to represent the most frequently used value and the longest string for the least frequently used. In the example above, -1 occurs the most frequently (15 times) and is represented by 00. A Huffman tree is constructed bottom up. A leaf node is created for each pixel value. Two leaf nodes are joined to an internal node if their sum is currently the minimal value. The total value is thus 64 in the above Huffman tree example. The string for a leaf node (a value) is obtained by traversing the tree top-down. A 0 is appended for each left branch and a 1 for each right branch. Huffman encoding is lossless and reversible because no code is a prefix of another code.

Archive step

- Record starting address and image size
  - Can use linked list
- One possible way to archive images
  - If max number of images archived is N:
    - Set aside memory for N addresses and N image-size variables
    - Keep a counter for location of next available address
    - Initialize addresses and image-size variables to 0
    - Set global memory address to N x 4
    - Assuming addresses, image-size variables occupy N x 4 bytes
    - First image archived starting at address N x 4
    - Global memory address updated to N x 4 + (compressed image size)
- Memory requirement based on N, image size, and average compression ratio

This step is performed after an image is captured and compressed. Normally, some kind of filesystem such as FAT16 or FAT32 is used in Flash or other memory media for storing compressed images. Here, a simpler archiving method is used. Memory is allocated for N addresses and N image-size variables. A counter points to the next available address. 4 bytes of addresses is assumed and hence the global memory address requires N x 4 bytes. First image is stored at address N x 4. Image size variations can be accommodated in this archiving method.
Uploading to PC

- When connected to PC and upload command received
  - Read images from memory
  - Transmit serially using UART
  - While transmitting
    - Reset pointers, image-size variables and global memory pointer accordingly

The uploading of compressed and stored images to PC can be accomplished in this step. Images are read from memory and transmitted serially using UART. Transmitted images are automatically deleted from the archive by resetting pointers.

Requirements Specification

- System’s requirements – what system should do
  - Nonfunctional requirements
    - Constraints on design metrics (e.g., “should use 0.001 watt or less”)
  - Functional requirements
    - System’s behavior (e.g., “output X should be input Y times 2”)
    - Initial specification may be very general and come from marketing dept.
      - E.g., short document detailing market need for a low-end digital camera that:
        - captures and stores at least 50 low-res images and uploads to PC,
        - costs around $100 with single medium-size IC costing less that $25,
        - has long as possible battery life,
        - has expected sales volume of 200,000 if market entry < 6 months,
        - 100,000 if between 6 and 12 months,
        - insignificant sales beyond 12 months

The requirements for a system can be segregated into functional and non-functional ones. Non-functional requirements include performance, reliability, security, safety, availability, and standards conformance. Functional requirements include all the functions that a system should perform including mainly its behavior.

Some examples of system specification for a digital camera SoC that come from the marketing department are as shown in the slides. The market time window is very important because a delayed time to market represents a system getting more and more useless.
Nonfunctional requirements

- Design metrics of importance based on initial specification
  - Performance: time required to process image
  - Size: number of elementary logic gates (2-input NAND gate) in IC
  - Power: measure of avg. electrical energy consumed while processing
  - Energy: battery lifetime (power x time)
- Constrained metrics
  - Values must be below (sometimes above) certain threshold
- Optimization metrics
  - Improved as much as possible to improve product
- Metric can be both constrained and optimization

Non-functional requirements for a digital camera also include the power or energy consumption because a high power consumption means a shorter battery life and thus a poorer quality product to sell. Details are in next slide.

Metrics can be of two types: constrained and optimization. Constrained metrics specify the threshold (minimum, maximum) values. For example, a max cost, a min performance, a max memory size, etc. Optimization metrics specify the goals of a product that must be met by designers while designing a system. For example, we might need to minimize the cost under some performance constraints or we might need to maximize the performance under some cost constraints.

Non-functional requirements (cont.)

- 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

Non-functional requirements are detailed in this slide for a digital camera SoC. The performance requirement is a one second constraint on the time interval between two shutter presses. This is a constrained metric because a longer delay would be annoying (for example, you will be frustrated if you missed out on some good football actions while the camera is processing a previous image!). A shorter delay is not required for the low-end market because that would mean an unaffordable price for the low-end users.

The other nonfunctional requirements include constrained hardware size in gate count (200,000 gates), power (must operate without fan), and energy (must be optimized).
**Informal functional specification**

- 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

The flowchart is a popular model for describing the functionalities of a system as is the example given for the digital camera in the slide.

There is a loop for processing all the 8 x 8 pixel blocks, a loop for transmitting all the bits serially through UART and a reactive loop for processing the next image from CCD input.

**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

The detailed functional specification is given in this slide in the form of an executable model in terms of the C program files: ccd.c, ccdpp.c, cntrl.c, codec.c, and uart.c. Profiling of the executables can provide information on the existence of bottlenecks in the system. Each of the executable modules are described in the later slides.
The CCD module simulates a real CCD by generating one pixel at a time. The image input however is pre-stored in a file. The format used for each module is similar: a initialization function, a capture function, and a pop function. The image is assumed to be of size 64 x 64 pixels. There are two extra columns (totally 66) which are used for zero-bias adjustment.

The CCDPP module performs zero-bias adjustment. It has the same format as the CCD module. The adjustment is done in the ccdppcapture function.
**UART module**

- Actually a half UART
  - Only transmits, does not receive
- UartInitialize is passed name of file to output to
- UartSend transmits (writes to output file) bytes at a time

The UART module simulates the transmitter half of a real UART. This module outputs to a file.

**CODEC module**

- Models FDCT encoding
- ibuffer holds original 8 x 8 block
- obuffer holds encoded 8 x 8 block
- CodecPushPixel called 64 times to fill ibuffer with original block
- CodecDoFdct called once to transform 8 x 8 block
  - Explained in next slide
- CodecPopPixel called 64 times to retrieve encoded block from obuffer

The codec module is responsible for FDCT encoding. The CodecPopPixel function is applied to an 8 x 8 block of pixels, so it is called 64 times for a 64 x 64 pixel sized image.
**CODEC (cont.)**

- Implementing FDCT formula
  \[ C(h) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } h = 0 \\ 1.0 & \text{otherwise} \end{cases} \]

\[ F(u,v) = \frac{1}{4} \sum_{x=0}^{7} \sum_{y=0}^{7} D_{xy} \cos\left(\frac{\pi}{16}(2u + 1)x\right) \cos\left(\frac{\pi}{16}(2y + 1)v\right) \]

- Only 64 possible inputs to \( \text{COS} \), hence we can use a table for precomputing and storing all the \( \text{COS} \) values in a lookup table. The equation for \( \text{COS} \) is implemented in the FDCT function.

\[
\text{static const short COS_TABLE[8][8] = } \begin{bmatrix}
32768, 32138, 30273, 27245, 23170, 18204, 12539, 6392 \\
32768, 27245, 12539, -6392, -23170, -32138, -30273, -18204 \\
32768, 18204, -12539, 32138, -23170, 6392, 30273, 27245 \\
32768, 6392, -30273, -18204, 23170, 27245, -12539, -32138 \\
32768, -6392, -30273, 18204, -23170, 32138, 30273, 27245 \\
32768, -18204, -12539, 32138, -23170, -6392, 30273, -27245 \\
32768, -27245, 12539, 6392, -23170, 32138, -30273, 18204 \\
32768, -32138, 30273, -27245, 23170, -18204, 12539, -6392 \\
\end{bmatrix}
\]

- \( \text{FDCT} \) unrolls inner loop of summation, implements outer summation as two consecutive for loops

```c
static const short ONE_OVER_SQRT_TWO = 23170;
static double COS(int xy, int uv) {
    return COS_TABLE[xy][uv] / 32768.0;
}
static double C(int h) {
    return h ? 1.0 : ONE_OVER_SQRT_TWO / 32768.0;
}
```

**CNTRL (controller) module**

- Heart of the system
- \( \text{CNTRL} \) initializes and interacts with other modules only
- \( \text{CNTRL} \) uses \( \text{CCDPP} \) module to input an image and place it in buffer
- \( \text{CNTRL} \) uses \( \text{FDCT} \) module to perform \( \text{FDCT} \) on each block using the \( \text{CODEC} \) module
- Also performs quantization on each block
- \( \text{CNTRL} \) transmits the encoded image serially using \( \text{UART} \) module

```c
#define SZ_ROW          64
#define SZ_COL          64
#define NUM_ROW_BLOCKS (SZ_ROW / 8)
#define NUM_COL_BLOCKS (SZ_COL / 8)

static short buffer[SZ_ROW][SZ_COL], i, j, k, l, temp;

void CntrlInitialize(void) {}
void CntrlCaptureImage(void) {
    CcdppCapture();
    for(i=0; i<SZ_ROW; i++)
        for(j=0; j<SZ_COL; j++)
            buffer[i][j] = CcdppPopPixel();
}
void CntrlCompressImage(void) {
    for(i=0; i<NUM_ROW_BLOCKS; i++)
        for(j=0; j<NUM_COL_BLOCKS; j++) {
            for(k=0; k<8; k++)
                for(l=0; l<8; l++)
                    CodecPushPixel((char)buffer[i * 8 + k][j * 8 + l]);
            CodecDoFdct();/* part 1 - FDCT */
            for(k=0; k<8; k++)
                for(l=0; l<8; l++)
                    buffer[i * 8 + k][j * 8 + l] = CodecPopPixel();/* part 2 - quantization */
            buffer[i*8+k][j*8+l] >>= 6;
        }
}
```

The \( \text{CNTRL} \) (controller module) is the heart of the system. It initializes all the modules, uses \( \text{CCDPP} \) module to input an image and place it in buffer, breaks the 64 x 64 buffer into 8 x 8 blocks, and performs \( \text{FDCT} \) on each block. Finally, it transmits the image serially using \( \text{UART} \) module.

There are only 64 different input values for \( \text{COS} \), hence we can use a table for precomputing and storing all the \( \text{COS} \) values in a lookup table. The equation for \( \text{COS} \) is implemented in the \( \text{FDCT} \) function.
Putting it all together

- Main initializes all modules, then uses CNTRL module to capture, compress, and transmit one image
- This system-level model can be used for extensive experimentation
  - Bugs much easier to correct here rather than in later models

```c
int main(int argc, char *argv[]) {
    char *uartOutputFileName = argc > 1 ? argv[1] : "uart_out.txt";

    /* initialize the modules */
    UartInitialize(uartOutputFileName);
    CcdInitialize(imageFileName);
    CcdppInitialize();
    CodecInitialize();
    CntrlInitialize();

    /* simulate functionality */
    CntrlCaptureImage();
    CntrlCompressImage();
    CntrlSendImage();
}
```

This is the main C function which uses the CNTRL module.

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

An architecture is decided for the system by selecting the processor, memories, and buses. The functionality is then mapped to the architecture and finally implemented. A purely software implementation is first executed to evaluate the system’s performance. Usually power, size, and time-to-market constraints are satisfied, however timing constraint may not be satisfied. A single-purpose processor is can be used to accelerate parts of the system that require heavy computations.
**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
  - 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

This is the first implementation using a single microcontroller such as Intel 8051. The total IC cost including NRE is about US$5 and the power is well below the 200 mW power constraint. The time-to-market is about 3 months. However, this implementation cannot meet the performance requirements of 1 second per image capture. 8051 is 12 MHz, that is, 12 cycles per instruction and can execute one million instructions per second. However, CcdppCapture itself requires nearly half of this budget. Thus, this implementation cannot meet the performance constraints because DCT and Huffman encoding will be even more compute-intensive.

**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

To accelerate the software implementation (previous slide), the ccdpp function is implemented on a custom single-purpose processor. UART is also implemented as a single-purpose processor. EEPROM and RAM are added, too.
**Microcontroller**

- Synthesizable version of Intel 8051 available
  - Written in VHDL
  - Captured at register transfer level (RTL)
- Fetches instruction from ROM
- Decodes using Instruction Decoder
- ALU executes arithmetic operations
  - Source and destination registers reside in RAM
- Special data movement instructions used to load and store externally
- Special program generates VHDL description of ROM from output of C compiler/linker

The microcontroller is an Intel 8051 VHDL model at the RTL. It fetches instruction from ROM, decodes it, and executes using the ALU. Special data movement instructions are used to load and store externally. There is a special program (given in the downloadable package) to compile the executable derived from C compiling/linking into ROM code for fetching and executing by the 8051 microcontroller.

**UART**

- UART in idle mode until invoked
  - UART invoked when 8051 executes store instruction with UART’s enable register as target address
- Memory-mapped communication between 8051 and all single-purpose processors
- Lower 8-bits of memory address for RAM
- Upper 8-bits of memory address for memory-mapped I/O devices
- Start state transmits 0 indicating start of byte transmission then transitions to Data state
- Data state sends 8 bits serially then transitions to Stop state
- Stop state transmits 1 indicating transmission done then transitions back to idle mode

The UART is described by an FSM in the slide, where it starts in the Idle mode and when invoked starts transmitting data bit-by-bit. When a byte is transmitted, it stops and returns back to the Idle mode. The UART transmitter hardware is implemented using this FSM.
**CCDPP**

- Hardware implementation of zero-bias operations
- Interacts with external CCD chip
  - CCD chip resides external to our SOC mainly because combining CCD with ordinary logic not feasible
- Internal buffer, $B$, memory-mapped to 8051
- Variables $R$, $C$ are buffer’s row, column indices
- GetRow state reads in one row from CCD to $B$
  - 66 bytes: 64 pixels + 2 blacked-out pixels
- ComputeBias state computes bias for that row and stores in variable $Bias$
- FixBias state iterates over same row subtracting $Bias$ from each element
- NextRow transitions to GetRow for repeat of process on next row or to Idle state when all 64 rows completed

The CCDPP FSM is also illustrated in the slide. It starts in the Idle mode, when invoked gets a row of pixel values, computes the corresponding bias, which is then used to fix all the row pixels, and another is obtained and the process repeats. $B$ is the internal buffer used to store the row pixels, that is, totally 66 bytes.

**Connecting SOC components**

- Memory-mapped
  - All single-purpose processors and RAM are connected to 8051’s memory bus
- Read
  - Processor places address on 16-bit address bus
  - Asserts read control signal for 1 cycle
  - Reads data from 8-bit data bus 1 cycle later
  - Device (RAM or SPP) detects asserted read control signal
  - Checks address
  - Places and holds requested data on data bus for 1 cycle
- Write
  - Processor places address and data on address and data bus
  - Asserts write control signal for 1 clock cycle
  - Device (RAM or SPP) detects asserted write control signal
  - Checks address bus
  - Reads and stores data from data bus

All the SoC components described in the previous slides are then interconnected. The basic transactions after interconnecting the components are read and write, which are very similar to microprocessor transactions.
Software

- System-level model provides majority of code
  - Module hierarchy, procedure names, and main program unchanged
- Code for UART and CCDPP modules must be redesigned
  - Simply replace with memory assignments
    - `xdata` used to load/store variables over external memory bus
    - `*_at_` specifies memory address to store these variables
    - Byte sent to `U_TX_REG` by processor will invoke UART
  - `U_STAT_REG` used by UART to indicate it’s ready for next byte
    - UART may be much slower than processor
- All other modules untouched

```
static unsigned char xdata U_TX_REG _at_ 65535;
static unsigned char xdata U_STAT_REG _at_ 65534;
```

Rewritten UART module

```
#include <stdio.h>
static FILE *outputFileHandle;

void UartInitialize(const char *outputFileName) {
  outputFileHandle = fopen(outputFileName, "w");
}

void UartSend(char d) {
  fprintf(outputFileHandle, "%d\n", (int)d);
}
```

Original code from system-level model

```
void UARTInitialize(void) {}
void UARTSend(unsigned char d) {
  while( U_STAT_REG == 1 ) {
    /* busy wait */
  }
  U_TX_REG = d;
}
```

Analysis

- Entire SoC tested on VHDL simulator
  - Interprets VHDL descriptions and functionally simulates execution of system
    - Recall program code translated to VHDL description of ROM
    - Tests for correct functionality
    - Measures clock cycles to process one image (performance)
- Gate-level description obtained through synthesis
  - Synthesis tool like compiler for SPPs
    - Simulate gate-level models to obtain data for power analysis
      - Number of times gates switch from 1 to 0 or 0 to 1
      - Count number of gates for chip area

```
Power equation
```

Obtaining design metrics of interest

The entire SoC was tested on a VHDL simulator, where the software code in C was compiled, linked, and translated into VHDL ROM code for logic simulation. The simulation was cycle accurate and could measure the number of cycles required for processing an image. Power is estimated only after the system is synthesized into gate level and re-simulated at the gate level. The size of the system can be estimated in terms of the gate count.
**Implementation 2: Microcontroller and CCDPP**

- 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

This slide shows the estimation results for implementation 2. It requires totally 9.1 seconds for processing an image, 0.033 watt power consumption, 0.30 joule of energy consumption (9.1 s x 0.033 watt), and the total chip area is 98,000 gates.

The gate count and the power/energy consumption were within the specified constraints but the speed of processing images (9.1 seconds) is much larger than the 1 second limit.

**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

Since implementation 2 does not meet the performance constraint of 1 second per image, it is found through profiling that DCT, which is the most compute-intensive (more than 80% of total execution time), must be accelerated. We will first look at a software improvement on the floating point computations which are very expensive.
**DCT floating-point cost**

- Floating-point cost
  - DCT uses ~260 floating-point operations per pixel transformation
  - 4096 (64 x 64) pixels per image
  - 1 million floating-point operations per image
  - No floating-point support with Intel 8051
    - Compiler must emulate
      - Generates procedures for each floating-point operation
      - Each procedure uses tens of integer operations
    - Thus, > 10 million integer operations per image
    - Procedures increase code size
- Fixed-point arithmetic can improve on this

The DCT function requires around 260 floating-point operations per pixel transformation, which amounts to 1 million such operations per image (64 x 64 pixels). There is no floating-point support in 8051, hence compiler must emulate the floating-point operation using integer operations. Each floating point operation is emulated using an average of 10 integer operations. Thus, we need totally more than 10 million integer operations per image. The procedures used to emulate floating-point operation increases the overhead.

The solution for this is “fixed-point arithmetic,” as discussed later.

**Fixed-point arithmetic**

- Integer used to represent a real number
  - Constant number of integer’s bits represents fractional portion of real number
    - More bits, more accurate the representation
  - Remaining bits represent portion of real number before decimal point
- Translating a real constant to a fixed-point representation
  - Multiply real value by $2^\#$ (where # of bits used for fractional part)
  - Round to nearest integer
    - E.g., represent 3.14 as 8-bit integer with 4 bits for fraction
      - Multiply by $2^4 = 16$
      - $3.14 \times 16 = 50.24 = 50.0010010$
      - 16 ($2^4$) possible values for fraction, each represents 0.0625 (1/16)
      - Last 4 bits (0010) = 2
      - $2 \times 0.0625 = 0.125$
      - 3(0011) + 0.125 = 3.125
      - Thus, 3.14 (more bits for fraction would increase accuracy)

Fixed-point representation is used mainly to reduce the complexity of arithmetic operations for floating-points. An integer is used to represent a real number. Some bits in an integer representation is used to represent the fractional part of a real number and the remaining bits are used to present the integer portion (before decimal point). A real value is multiplied by $2^\#$ (where # of bits used for fractional part) and then rounded to nearest integer. An example is given here. Suppose an integer has 8 bits, 4 of which are used to represent the fraction. The transformation for 3.14 into its fixed-point representation 50 is shown in the slide. An inverse calculation process starting from 50 does not result in 3.14, it becomes 3.125. Thus, there is an approximation.
**Fixed-point arithmetic operations**

- **Addition**
  - Simply add integer representations
  - E.g., \(3.14 + 2.71 = 5.85\)
    - \(3.14 \rightarrow 011010010\)
    - \(2.71 \rightarrow 00101011\)
    - \(50 + 43 = 93 = 01011101\)
    - \(5(0101) + 13(1101) \times 0.0625 = 5.8125 \approx 5.85\)

- **Multiply**
  - Multiply integer representations
  - Shift result right by # of bits in fractional part
  - E.g., \(3.14 \times 2.71 = 8.5094\)
    - \(50 \times 43 = 2150 = 100001100110\)
    - \(\gg 4 = 10000110\)
    - \(8(1000) + 6(0110) \times 0.0625 = 8.375 \approx 8.5094\)

- Range of real values used limited by bit widths of possible resulting values

Addition and multiplication for real numbers represented in fixed-point format can be performed directly in their fixed point formats. Examples are given in the slide.

**Fixed-point implementation of CODEC**

- **COS_TABLE** gives 8-bit fixed-point representation of cosine values
- 6 bits used for fractional portion
- Result of multiplications shifted right by 6

The CODEC module is transformed into a fixed-point implementation version. The COS_TABLE gives 8-bit fixed point representation of cosine values, where 6 bits are used for fractional portion.
**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)

By converting the CODEC module into fixed-point, we have implementation 3 which becomes faster at the expense of some approximation in the image pixel values. The execution time for processing a single image is 1.5 seconds, the power is 0.033 watt, the energy is 0.050 joule, and the total chip area is 90,000 gates. Except for the execution time, all other constraints are satisfied.

**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

Performance in implementation 3 is close but not good enough, hence we finally resort to implementing the CODEC hardware using a single-purpose processor, which performs DCT on 8 x 8 pixel block.
**CODEC design**

- 4 memory mapped registers
  - C_DATAI_REG/C_DATAO_REG used to push/pop 8 x 8 block into and out of CODEC
  - C_CMND_REG used to command CODEC
    - Writing 1 to this register invokes CODEC
  - C_STAT_REG indicates CODEC done and ready for next block
    - Polled in software
- Direct translation of C code to VHDL for actual hardware implementation
  - Fixed-point version used
- CODEC module in software changed similar to UART/CCDPP in implementation 2

The hardware implementation for CODEC is shown here. 4 memory mapped registers are used to record status, write commands, data input, and data output. The fixed-point version of the C code is directly translated into VHDL code. The software code for CODEC is changed in a way similar to that for UART/CCDPP in implementation 2. Commands are written to the command register in CodecDoF dct.

**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

This hardware implementation of DCT results in implementation 4 and the execution time is reduced to 0.099 seconds, which is well under 1 second. Power consumption is 0.04 watt, energy consumption is 0.0004 joule, and the total chip area is 128,000 gates.
Summary of implementations

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Performance (second)</th>
<th>Power (watt)</th>
<th>Size (gate)</th>
<th>Energy (joule)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation 1</td>
<td>9.1</td>
<td>0.033</td>
<td>98,000</td>
<td>0.30</td>
</tr>
<tr>
<td>Implementation 2</td>
<td>1.5</td>
<td>0.033</td>
<td>90,000</td>
<td>0.050</td>
</tr>
<tr>
<td>Implementation 3</td>
<td>0.099</td>
<td>0.040</td>
<td>128,000</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?

A summary of the features of the 4 different implementations is tabulated here. Implementation 3 is close in performance, cheaper, and requires less time to build. Implementation 4 has great performance and energy consumption, however, it is more expensive than implementation 3. Which is better? May be implementation 3 is suitable for users who can accommodate a slight performance degradation by paying less for the camera, whereas implementation 4 is for users who are willing to pay for high performance. This is the very reason why we have different product versions for the same brand product with similar functionalities.

Summary

- Digital camera example
  - Specifications in English and executable language
  - Design metrics: performance, power and area
- Several implementations
  - Microcontroller: too slow
  - Microcontroller and coprocessor: better, but still too slow
  - Fixed-point arithmetic: almost fast enough
  - Additional coprocessor for compression: fast enough, but expensive and hard to design
  - Tradeoffs between hw/sw!

As a summary, digital camera example was specified in English and an executable language. There were 3 design metrics, including performance, power, and area. There is tradeoff between hardware and software during the implementation of the 3 different design alternatives for the digital camera.