Hardware-Software Codesign

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
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
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

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
**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)

**Typical Codesign Process**

- System Description (Functional)
- HW/SW Partitioning
- Hardware Synthesis
- Software Synthesis
- Interface Synthesis
- System Integration
- Concurrent processes
- Programming languages
- Unified representation (Data/control flow)
- Instruction set level
- HW/SW evaluation
Codesign Process

- System specification
  - Models, Architectures, Languages
- HW/SW partitioning
  - Architectural assumptions:
    - Type of processor, interface style, etc.
  - Partitioning objectives:
    - Speedup, latency requirement, silicon size, cost, etc.
  - Partitioning strategies:
    - High-level partitioning by hand, computer-aided partitioning technique, etc.
  - HW/SW estimation methods
- HW/SW synthesis
  - Operation scheduling in hardware
  - Instruction scheduling in compiler
  - Process scheduling in operating systems
  - Interface Synthesis
  - Refinement of Specification

Requirements for the Ideal Codesign Environment

- Unified, unbiased hardware/software representation
  - Supports uniform design and analysis techniques for hardware and software
  - Permits system evaluation in an integrated design environment
  - Allows easy migration of system tasks to either hardware or software
- Iterative partitioning techniques
  - Allow several different designs (HW/SW partitions) to be evaluated
  - Aid in determining best implementation for a system
  - Partitioning applied to modules to best meet design criteria (functionality and performance goals)
- Integrated modeling substrate
  - Supports evaluation at several stages of the design process
  - Supports step-wise development and integration of hardware and software
- Validation Methodology
  - Insures that system implemented meets initial system requirements
Models, Architectures, Languages

Models & Architectures

Models are conceptual views of the system’s functionality
Architectures are abstract views of the system’s implementation
**Models of an Elevator Controller**

“If the elevator is stationary and the floor requested is equal to the current floor, then the elevator remains idle.
If the elevator is stationary and the floor requested is less than the current floor, then lower the elevator to the requested floor.
If the elevator is stationary and the floor requested is greater than the current floor, then raise the elevator to the requested floor.”

**Architectures Implementing the Elevator Controller**

(a) Register level
(b) System level
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, Queuing Model, Programming Language Paradigm, Structure Chart

State-Oriented: Finite State Machine (Moore Model, Mealy Model)
**Codesign Finite State Machine (CFSM)**

- Globally Asynchronous, Locally Synchronous (GALS) model

![CFSM Diagram]

**State-Oriented: Petri Nets**

(a) Sequence  
(b) Branch  
(c) Synchronization  
(d) Resource contention  
(e) Concurrency
Activity-Oriented: Data Flow Graphs (DFG)

- Graphs contain nodes corresponding to operations in either hardware or software
- Often used in high-level hardware synthesis
- Can easily model data flow, control steps, and concurrent operations because of its graphical nature

Heterogeneous: Control/Data Flow Graphs (CDFG)

- Graphs contain nodes corresponding to operations in either hardware or software
- Often used in high-level hardware synthesis
- Can easily model data flow, control steps, and concurrent operations because of its graphical nature
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

Heterogeneous: Object-Oriented Paradigms (UML, …)

Object-Oriented Representation

Example:

3 Levels of abstraction:
Separate Behavior from Micro-architecture

- System Behavior
  - Functional specification of system
  - No notion of hardware or software!

- Implementation Architecture
  - Hardware and Software
  - Optimized Computer
**IP-Based Design of the Implementation**

- Can I Buy an MPEG2 Processor? Which One?
- Do I need a dedicated Audio Decoder? Can decode be done on Microcontroller?
- Which DSP Processor? C50? Can DSP be done on Microcontroller?
- Which Bus? PI? AMBA? Dedicated Bus for DSP?
- Which Microcontroller? ARM? HC11?
- How fast will my User Interface Software run? How Much can I fit on my Microcontroller?

**AMBA-Based SoC Architecture**

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

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

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
**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 Language in SOC Codesign Flow**

- Design Specification
- Estimators
- Hw/Sw Partitioning
- HW: VHDL, Verilog
- SW: C
- Synthesis
- Compiler
- Cosimulation
- SW/SW, SW/PU, P2M ...
- IP Library
- Verification
- Rapid design space exploration
- Quality tool-kit generation
- Design reuse
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

System Partitioning
System Partitioning
(Functional Partitioning)

- System partitioning in the context of hardware/software codesign is also referred to as functional partitioning.
- Partitioning functional objects among system components is done as follows:
  - The system’s functionality is described as a collection of indivisible functional objects.
  - Each system component’s functionality is implemented in either hardware or software.
- Constraints:
  - Cost, performance, size, power.
- An important advantage of functional partitioning is that it allows hardware/software solutions.
- This is a multivariate optimization problem that when automated, is an NP-hard problem.

Issues in Partitioning

- High Level Abstraction
- Decomposition of functional objects
  - Granularity
  - Metrics and estimations
  - Partitioning algorithms
  - Objective and closeness functions
- Component allocation
- Output
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.

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.
**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

**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*
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

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., \( \text{Objfct} = k_1 \times \text{area} + k_2 \times \text{delay} + k_3 \times \text{power} \) (equation 1)
  - Because constraints always exist on each design, they must be taken into account
    - e.g. \( \text{Objfct} = k_1 \times F(\text{area, area\_constr}) + k_2 \times F(\text{delay, delay\_constr}) + k_3 \times F(\text{power, power\_constr}) \)
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

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
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

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
**Ratio Cut**

- A constructive algorithm that groups objects until a terminal condition has been met.
- A new metric **ratio** is defined as
  \[ \text{ratio} = \frac{\text{cut}(P)}{\text{size}(p_1) \times \text{size}(p_2)} \]
  - \( \text{Cut}(P) \): 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.

**Greedy partitioning for HW/SW partition**

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

Repeat
  
P_orig=P
  for i in 1 to n loop
    if Objfct(Move(P,o))<Objfct(P) then
      P=Move(P,o)
    end if
  end loop
Until P=P_orig
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.

Simulated annealing

- Iterative algorithm modeled after physical annealing process that 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
Simulated annealing algorithm

\[ \text{Temp}=\text{initial temperature} \]
\[ \text{Cost}=\text{Objfct}(P) \]

While not Frozen loop
   while not Equilibrium loop
      \[ P_{\text{tentative}}=\text{Move}(P) \]
      \[ \text{cost}_{\text{tentative}}=\text{Objfct}(P_{\text{tentative}}) \]
      \[ \Delta\text{cost}=\text{cost}_{\text{tentative}}-\text{cost} \]
      if(Accept(\(\Delta \text{cost}, \text{temp}\))>Random(0,1)) then
         \[ P=P_{\text{tentative}} \]
         \[ \text{cost}=\text{cost}_{\text{tentative}} \]
      end if
   end loop
   \[ \text{temp}=\text{DecreaseTemp}(\text{temp}) \]
End loop

where:\n\[ \text{Accept}(\Delta \text{cost}, \text{temp}) = \min(1, e^{\frac{\Delta \text{cost}}{-\text{temp}}}) \]

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: random selected partition.
  - Crossover: randomly selected from two strong partitions.
  - Mutation: randomly selected partition after some randomly modification.
- Produce good result but suffer from long run times.

/*Evolve generation*/
While not Terminate loop
   \[ G=\text{Select}(G, \text{num\_sel}) \cup \text{Cross}(G, \text{num\_cross}) \]
   \[ \text{Mutate}(G, \text{num\_mutatae}) \]
   If Objfct(BestPart(G))<Objfct(P\_best) then
      \[ P\_best=\text{BestPart}(G) \]
   end if
end loop
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

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

- Simplified estimation models yield fast estimator but result in greater estimation error and less accuracy.
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)  
(B, C) = E(B) < E(C), M(B) > M(C)  
(A, C) = E(A) < E(C), M(A) < M(C)  

Fidelity = 33%

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
Scheduling

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

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
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:

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

(1)

- The total number of control steps for behavior B is

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

(2)

---

Operator-use method Example

- Differential-equation example:
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
  \[ \text{execution}(B) = \sum_{b_i \in B} \text{exectime}(b_i) \times \text{freq}(b_i) \]  

Probability-based flow analysis
Probability-based flow analysis

- Flow equations:
  - $\text{freq}(S) = 1.0$
  - $\text{freq}(v1) = 1.0 \times \text{freq}(S)$
  - $\text{freq}(v2) = 1.0 \times \text{freq}(v1) + 0.9 > \text{freq}(v5)$
  - $\text{freq}(v3) = 0.5 \times \text{freq}(v2)$
  - $\text{freq}(v4) = 0.5 \times \text{freq}(v2)$
  - $\text{freq}(v5) = 1.0 \times \text{freq}(v3) + 1.0 > \text{freq}(v4)$
  - $\text{freq}(v6) = 0.1 \times \text{freq}(v5)$

- Node execution frequencies:
  - $\text{freq}(v1) = 1.0$
  - $\text{freq}(v2) = 10.0$
  - $\text{freq}(v3) = 5.0$
  - $\text{freq}(v4) = 5.0$
  - $\text{freq}(v5) = 10.0$
  - $\text{freq}(v6) = 1.0$

- Can be used to estimate number of accesses to variables, channels or procedures

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 $C$, 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, peakrate$(C)$, is defined as the rate at which data is sent in a single message transfer.
  \[
  \text{peakrate}(C) = \frac{\text{bits}(C)}{\text{protocol \_ delay}(C)}
  \]
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.

Software estimation models
Deriving processor technology files

Generic instruction

$d_{mem3} = d_{mem1} + d_{mem2}$

### Technology file for 8086

<table>
<thead>
<tr>
<th>Instruction</th>
<th>clock</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ax, word ptr[bp+offset1]</td>
<td>(10)</td>
<td>3</td>
</tr>
<tr>
<td>add ax, word ptr[bp+offset2]</td>
<td>(9+EA1)</td>
<td>4</td>
</tr>
<tr>
<td>mov word ptr[bp+offset3], ax</td>
<td>(10)</td>
<td>3</td>
</tr>
</tbody>
</table>

### Technology file for 68020

<table>
<thead>
<tr>
<th>Instruction</th>
<th>clock</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov a6@[offset1], do</td>
<td>(7)</td>
<td>2</td>
</tr>
<tr>
<td>add a6@[offset2], do</td>
<td>(2+EA2)</td>
<td>2</td>
</tr>
<tr>
<td>mov d0,a6@[offset3]</td>
<td>(6)</td>
<td>2</td>
</tr>
</tbody>
</table>

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:
    \[
    \text{exectime}(B) = \delta \times \left( \sum_{i} \text{exectime}(b_i) \times \text{freq}(b_i) \right)
    \]
    \[\delta \text{ accounts for compiler optimizations}\]
  - accounts for compiler optimizations

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

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

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

```
V (63 downto 0) => MEM(163 downto 100)
```

```latex
\begin{align*}
\text{variable } J, K & : \text{integer} := 0; \\
\text{variable } V & : \text{IntArray (63 downto 0)}; \\
\ldots & \\
V(K) & := 3; \\
X & := V(36); \\
V(J) & := X; \\
\ldots & \\
\text{for } J \text{ in 0 to 63 loop} & \\
\text{SUM := SUM + V(J);} & \\
\text{end loop;} & \\
\ldots & \\
\text{variable } J, K & : \text{integer} := 0; \\
\text{variable } MEM & : \text{IntArray (255 downto 0)}; \\
\ldots & \\
MEM(K + 100) & := 3; \\
X & := MEM(136); \\
MEM(J + 100) & := X; \\
\ldots & \\
\text{for } J \text{ in 0 to 63 loop} & \\
\text{SUM := SUM + MEM(J + 100);} & \\
\text{end loop;} & \\
\ldots & 
\end{align*}
```
**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

---

**Inter-process communication paradigms:**

(a) shared memory, (b) message passing
Channel refinement

- Channels: virtual entities over which messages are transferred
- Bus: physical medium that implements groups of channels
- Bus consists of:
  - wires representing data and control lines
  - protocol defining sequence of assignments to data and control lines
- Two refinement tasks
  - Bus generation: determining bus width
    - number of data lines
  - Protocol generation: specifying mechanism of transfer over bus

Protocol generation

- Protocol selection
  - full handshake, half-handshake etc.
- ID assignment
  - N channels require log2(N) ID lines
- Bus structure and procedure definition.
- Update variable-reference.
- Generate processes for variables.
Protocol generation example

```vhdl
type HandShakeBus is record
  START, DONE : bit;
  ID : bit_vector(1 downto 0);
  DATA : bit_vector(7 downto 0);
end record;

signal B : HandShakeBus;

procedure ReceiveCH0( rxdata : out bit_vector) is
begin
  for J in 1 to 2 loop
    wait until (B.START = '1') and (B.ID = "00")
    rxdata (8*J-1 downto 8*(J-1)) <= B.DATA;
    B.DONE <= '1';
  wait until (B.START = '0');
  B.DONE <= '0' end loop;
end ReceiveCH0;

procedure SendCH0( txdata : in bit_vector) is
begin
  bus B.ID <= "00";
  for J in 1 to 2 loop
    B.data <= txdata(8*J-1 downto 8*(J-1));
    B.START <= '1';
    wait until (B.DONE = '1');
    B.START <= '0';
    wait until (B.DONE = '0');
  end loop end SendCH0;
```

Refined specification after protocol generation

```
process P
  variable AD Xtemp
begin
  SendCH3(32);
  [...] ReceiveCH1(Xtemp)
  SendCH2(AD,Xtemp+7);
  [...] end

process Q
  variable COUNT
begin
  SendCH3(60, COUNT);
  [...] end

process Pproc
  variable X
begin
  wait on B.ID;
  if (B.ID="00") then receiveCH0(X)
  else if (B.ID="01") then sendCH1(X)
  end f;
end;
process Xproc

process MEMproc
  variable MEM: array(0 to 53) of std_log2(2);
begin
  wait on B.ID;
  if (B.ID="10") then receiveCH2(MEM)
  else if (B.ID="11") then sendCH3(MEM)
  end f;
end;
```
Arbitration schemes

- Arbitration schemes determines 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

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
Hardware/Software interface refinement

(a) Partitioned Specification  (b) Mapping to Architecture

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.
Contents

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

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) Methodology

FAC System Level Design Vision
**Main Concepts**

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

**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
**Target Architectural Exploration and Estimation**

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

**Architectural Exploration in POLIS**

![Diagram of Architectural Exploration in POLIS]
Reactive System Cosynthesis

EFSM: Extended Finite State Machines
CDFG: Control Data Flow directed acyclic Graph

EFSM is suitable for describing EFSM reactive behavior but

- Some of the control flow is hidden
- Data cannot be propagated
Data Flow Optimization

Function/Architecture Codesign

Design Representation
Abstract Codesign Flow

Unifying Intermediate Design Representation for Codesign
Models (in POLIS / FAC)

- **Function Model**
  - “Esterel”
    - as “front-end” for functional specification
    - Synchronous programming language for specifying reactive real-time systems
  - Reactive VHDL
  - EFSM (Extended FSM): support for data handling and asynchronous communication

- **Architecture Model**
  - SHIFT: Network of CFSM (Codesign FSM)

---

**CFSM**

- **Includes**
  - Finite state machine
  - Data computation
  - Locally synchronous behavior
  - Globally asynchronous behavior

- **Semantics:** GALS (Globally Asynchronous and Locally Synchronous communication model)
CFSM Network MOC

MOC: Model of Computation

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.
(Architecture) Function Flow Graph

Refinement

Architecture Independent

Architecture Dependent

Design

Functional Decomposition

Constraints

EFSM Semantics

I/O Semantics

FFG/CLIF

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

Data Flow/Control Optimizations
**Function Flow Graph (FFG)**

- FFG is a triple $G = (V, E, N_0)$ where
  - $V$ is a finite set of nodes
  - $E = \{(x,y)\}$, 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).

**C-Like Intermediate Format (CLIF)**

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

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;

FFG / CLIF Example

Legend: constant, output flow, dead operation
S# = State, S#L# = Label in State S#

Function
Flow Graph
S1:

CLIF
Textual Representation
S1:
x = x + y;
x = x + y;
a = b + c;
a = x;
cond1 = (y == cst1);
cond2 = !cond1;
if (cond2) goto S1L0
output = a;
goto S1; /* Loop */
S1L0:
output = b;
goto S1;
Tree-Form FFG

Function/Architecture Codesign

Function/Architecture Optimizations
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

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.
**FFG Optimization Algorithm**

- FFG Optimization algorithm (G)

  ```
  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
  ```

**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
**Sample DFA Problem**

**Available Expressions Example**

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

\[
\begin{align*}
AE &= \emptyset \\
S_1 &\quad t := a + 1 \\
AE &= \{a+1\} \\
S_2 &\quad t_1 := a + 1 \\
&\quad t_2 := b + 2 \\
AE &= \{a+1, b+2\} \\
S_3 &\quad a := a * 5 \\
&\quad t_3 := a + 2 \\
AE &= \{a+2\}
\end{align*}
\]

**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 \( D \).
**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 $L$.

**Solving Data Flow Problems**

Data Flow Equations

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

$$\text{Out}(S3) = (\text{In}(S3) - \text{Kill}(S3)) \cup \text{Gen}(S3)$$

$AE = \text{Available Expression}$
Solving Data Flow Problems

- Solve data flow problems using the \textit{iterative method}
  - General: does not depend on the flow graph
  - Optimal for a \textit{class of data flow problems}
  - Reaches fixpoint 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
Function/Architecture Codesign

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

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.
Architecture Dependent Optimizations

```
Architecture
Independent
```

```
lib
Architectural
Information

EFSM → FFG → OFFG → AFFG → CDFG
```

```
EFSM in AFFG (State Tree) Form
```

```
S0
F0 → F1 → F2

S1
F3 → F4

S2
F5 → F6 → F7 → F8
```
**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

**Motivating Example**

Eliminate the redundant needless runtime re-evaluation of the a+b operation
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):
  
  \[ \text{begin} \]
  
  \[
  \begin{align*}
  &\text{Data Flow and Control Optimization} \\
  &\text{Reverse Sweep (dead operation addition, Normalization and available operation elimination, dead operation elimination)} \\
  &\text{Forward Sweep (optional, minimize the lifetime)} \\
  &\text{Final Optimization Pass}
  \end{align*}
  \]

  \[ \text{end} \]

Cost-Guided Operation Motion

Cost Estimation

- User Input
- Profiling
  
  \[ \text{Inference Engine} \]

Design Optimization

- FFG (back-end)
- Attributed FFG
  
  \[ \text{Relaxed Operation Motion} \]
**Function Architecture Co-design in the Micro-Architecture**

System Constraints → Decomposition → System Specs

- \( t_1 = 3b \)
- \( t_2 = t_1 + a \)
- Emit \( x(t_2) \)

AFFG

**Operator Strength Reduction**

Reducing the multiplication operator:

1. \( t_1 = 3b \)
2. \( t_2 = t_1 + a \)
3. \( x = t_2 \)

Expressed as:

- \( \text{expr1} = b + b; \)
- \( t_1 = \text{expr1} + b; \)
- \( t_2 = t_1 + a; \)
- \( x = t_2; \)
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)**
  - SHIFT = CFSMs + Functions

Macro-Architectural Organization

![Diagram of SW and HW partitions with interfaces and processors](image)
**Architectural Organization of a Single CFSM Task**

CFSM

![CFSM Diagram]

**Task Level Control and Data Flow Organization**

![Task Diagram]
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: CFSMs + Functions
**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

**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
**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

---

**Distributing the Reactive Controller**

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

![Distributing the Reactive Controller Diagram](image)
**Multiplexing Inputs**

\[ c = a \]

\[ T = b + c \]

\[ \text{Control} \{ 1, 2 \} \]

\[ a \]

\[ c \]

\[ b \]

**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**

\[ T_2 = a + b \]

\[ \text{Out} = T_1(a+b); \]

\[ \text{emit} (\text{Out}) \]

\[ a \]

\[ b \]

\[ x \]

\[ \text{Out} \]
Function/Architecture Codesign

Hardware/Software Cosynthesis and Estimation

Co-Synthesis Flow
Concrete FA Codesign Flow

POLIS Co-design Environment
**POLIS Co-design Environment**

- Specification: FSM-based languages (Esterel, ...)
- Internal representation: CFSM network
- Validation:
  - High-level co-simulation
  - FSM-based formal verification
  - Rapid prototyping
- Partitioning: based on co-simulation estimates
- Scheduling
- Synthesis:
  - S-graph (based on a CDFG) based code synthesis for software
  - Logic synthesis for hardware
- Main emphasis on unbiased verifiable specification

**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
RTOS Synthesis and Evaluation in POLIS

1. Provide communication mechanisms among CFSMs implemented in SW and between the OS and the HW partitions.
2. Schedule the execution of the SW tasks.

Estimation for CDFG Synthesis

Costs in clock cycles on a 68HC11 target processor using the Introl compiler for a simple CFSM
**Estimation-Based Co-simulation**

![Diagram of Estimation-Based Co-simulation](image)

**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
Co-simulation Approach

- 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
- Future Work
  - *Architectural view instead of component view*

Coverification Tools

- Mentor Graphics Seamless Co-Verification Environment (CVE)

*The patented seamless Coherent Memory Server empowers you to switch dynamically between detailed hardware verification and high-speed software execution.*
Seamless CVE Performance Analysis

References

Codesign Case Studies

- ATM Virtual Private Network
  - CSELT, 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
ATM EXAMPLE OUTLINE

- INTRODUCTION
- THE POLIS CO-DESIGN METHODOLOGY
- IP INTEGRATION INTO THE CO-DESIGN FLOW
- THE TARGET DESIGN: THE ATM VIRTUAL PRIVATE NETWORK SERVER
- RESULTS
- CONCLUSIONS

NEEDS FOR EMBEDDED SYSTEM DESIGN

- EASY DESIGN SPACE EXPLORATION
- EARLY DESIGN VALIDATION
- HIGH DESIGN PRODUCTIVITY
- HIGH DESIGN 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

**THE POLIS CODESIGN FLOW**

Graphical EFSM → ESTEREL → CFSMs → HW/SW CO-SIMULATION PERFORMANCE / TRADE-OFF EVALUATION → PHYSICAL PROTOTYPING → SW CODE + RTOS CODE → SW SYNTHESIS → SW ESTIMATION → PARTITIONING → HW ESTIMATION → HW SYNTHESIS → FORMAL VERIFICATION → COMPILERS
GOALS

- ASSESSMENT OF POLIS ON A TELECOM SYSTEM DESIGN
  - CASE STUDY: ATM VIRTUAL PRIVATE NETWORK SERVER

- INTEGRATION OF THE POLIS DESIGN FLOW
CASE STUDY: AN ATM VIRTUAL PRIVATE NETWORK SERVER

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
**DESIGN IMPLEMENTATION**

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

**CONTROL UNIT:**
- 25 CFSMs

**VIPTM LIBRARY MODULES**
- HW/SW CODESIGN MODULES
- COMMERCIAL MEMORIES

**ALGORITHM MODULE ARCHITECTURE**

**DATA PATH:**
- ATM 155 Mbit/s

**CONTROL UNIT:**
- 25 CFSMs
DESIGN SPACE EXPLORATION

- **CONTROL UNIT**
  - Source code: 1151 ESTEREL lines
  - Target processor family: MIPS 3000 (RISC)
- **FUNCTIONAL VERIFICATION**
  - Simulation (PTOLEMY)
- **SW PERFORMANCE ESTIMATION**
  - Co-simulation (POLIS VHDL model generator)
- **RESULTS**
  - 544 CPU clock cycles per time slot
  - $\downarrow$ 200 MHz clock frequency

**PROCESSOR FAMILY CHANGED**
(MOTOROLA PowerPC™)

<table>
<thead>
<tr>
<th>MODULE</th>
<th>SIZE</th>
<th>I</th>
<th>II</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSD TECHNIQUE</td>
<td>180</td>
<td>HW</td>
<td></td>
</tr>
<tr>
<td>CELL EXTRACTION</td>
<td>95</td>
<td>HW</td>
<td></td>
</tr>
<tr>
<td>VIRTUAL CLOCK</td>
<td>95</td>
<td>HW</td>
<td>HW</td>
</tr>
<tr>
<td>SCHEDULER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL TIME SORTER</td>
<td>300</td>
<td>HW</td>
<td>HW</td>
</tr>
<tr>
<td>ARBITER #1</td>
<td>33</td>
<td>HW</td>
<td>HW</td>
</tr>
<tr>
<td>ARBITER #2</td>
<td>34</td>
<td>HW</td>
<td>HW</td>
</tr>
<tr>
<td>ARBITER #3</td>
<td>37</td>
<td>HW</td>
<td>HW</td>
</tr>
<tr>
<td>LQM INTERFACE</td>
<td>75</td>
<td>HW</td>
<td>HW</td>
</tr>
<tr>
<td>SUPERVISOR</td>
<td>120</td>
<td>HW</td>
<td></td>
</tr>
</tbody>
</table>

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
### 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>66</td>
<td>106</td>
<td>114</td>
<td>1,600</td>
</tr>
<tr>
<td>CELL EXTRACTION</td>
<td>26</td>
<td>55</td>
<td>66</td>
<td>564</td>
</tr>
<tr>
<td>VIRTUAL CLOCK SCHEDULER</td>
<td>77</td>
<td>71</td>
<td>95</td>
<td>1,280</td>
</tr>
<tr>
<td>REAL TIME SORTER</td>
<td>261</td>
<td>731</td>
<td>52</td>
<td>10,504</td>
</tr>
<tr>
<td>ARBITER #1</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>114</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>16</td>
<td>9</td>
<td>17</td>
<td>159</td>
</tr>
<tr>
<td>LQM INTERFACE</td>
<td>20</td>
<td>39</td>
<td>27</td>
<td>603</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>

### DATA PATH 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>UTOPIA RX INTERFACE</td>
<td>120</td>
<td>251</td>
<td>37</td>
<td>16,300</td>
</tr>
<tr>
<td>UTOPIA TX INTERFACE</td>
<td>140</td>
<td>265</td>
<td>43</td>
<td>16,700</td>
</tr>
<tr>
<td>LOGIC QUEUE MANAGER</td>
<td>247</td>
<td>332</td>
<td>31</td>
<td>5,360</td>
</tr>
<tr>
<td>ADDRESS Lookup</td>
<td>87</td>
<td>96</td>
<td>82</td>
<td>1,700</td>
</tr>
<tr>
<td>ADDRESS CONVERTER</td>
<td>14</td>
<td>13</td>
<td>17</td>
<td>240</td>
</tr>
<tr>
<td>PARALLELISM CONVERTER</td>
<td>48</td>
<td>31</td>
<td>47</td>
<td>480</td>
</tr>
<tr>
<td>DATAPATH TOTAL</td>
<td>658</td>
<td>1001</td>
<td>47</td>
<td>42,000</td>
</tr>
</tbody>
</table>
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)

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…)

171

172
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

Digital Camera SoC Example

Outline

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

Introduction to a simple digital camera

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


**Designer’s perspective**

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

---

**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.
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

![Covered cells and Zero-bias adjustment](image)

Compression

- Store more images
- Transmit image to PC in less time
- JPEG (Joint Photographic Experts Group)
  - Popular standard format for representing digital images in a compressed form
  - Provides for a number of different modes of operation
  - Mode used in this example provides high compression ratios using DCT (discrete cosine transform)
  - Image data divided into blocks of 8 x 8 pixels
  - 3 steps performed on each block
    - DCT
    - Quantization
    - Huffman encoding
DCT step

- Transforms original 8 x 8 block into a cosine-frequency domain
  - Upper-left corner values represent more of the essence of the image
  - Lower-right corner values represent finer details
    - Can reduce precision of these values and retain reasonable image quality
- FDCT (Forward DCT) formula
  - \( C(h) = \begin{cases} 1/\sqrt{2} & \text{if } h = 0 \\ 1.0 & \text{else} \end{cases} \)
  - Auxiliary function used in main function \( F(u,v) \)
  - \( F(u,v) = \frac{1}{8} \times C(u) \times C(v) \times \sum_{x=0}^{7} \sum_{y=0}^{7} D_{xy} \times \cos\left( \frac{\pi}{8} (2u + 1)u/16 \right) \times \cos\left( \frac{\pi}{8} (2y + 1)v/16 \right) \)
    - Gives encoded pixel at row \( u \), column \( v \)
    - \( D_{xy} \) is original pixel value at row \( x \), column \( y \)
- IDCT (Inverse DCT)
  - Reverses process to obtain original block (not needed for this design)

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

<table>
<thead>
<tr>
<th>After being decoded using DCT</th>
<th>Divide each cell's value by 8</th>
<th>After quantization</th>
</tr>
</thead>
<tbody>
<tr>
<td>144</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>
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

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 nodes 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
**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

**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
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

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
Nonfunctional 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

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
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

Executable model of digital camera

CCD module

- Simulates real CCD
- CcdInitialize is passed name of image file
- CcdCapture reads “image” from file
- CcdPopPixel outputs pixels one at a time

```c
#include <stdio.h>
#define SZ_ROW 64
#define SZ_COL (64 + 2)
static FILE *imageFileHandle;
static char buffer[SZ_ROW][SZ_COL];
static unsigned rowIndex, colIndex;

void CcdInitialize(const char *imageFileName) {
    imageFileHandle = fopen(imageFileName, "r");
    rowIndex = -1;
    colIndex = -1;
}

void CcdCapture(void) {
    int pixel;
    rewind(imageFileHandle);
    for(rowIndex=0; rowIndex<SZ_ROW; rowIndex++) {
        for(colIndex=0; colIndex<SZ_COL; colIndex++) {
            if (fscanf(imageFileHandle, "%i", &pixel) == 1) {
                buffer[rowIndex][colIndex] = (char)pixel;
            }
        }
    }
    rowIndex = 0;
    colIndex = 0;
}

char CcdPopPixel(void) {
    char pixel;
    if (pixel = buffer[rowIndex][colIndex]) {
        if (++colIndex == SZ_COL) {
            colIndex = 0;
            if (++rowIndex == SZ_ROW) {
                rowIndex = -1;
            }
        }
        rowIndex = -1;
    }
    return pixel;
}
```
CCDPP (CCD PreProcessing) module

- Performs zero-bias adjustment
- CcdpCapture uses CcdCapture and CcdPopPixel to obtain image
- Performs zero-bias adjustment after each row read in

```c
#include <stdio.h>

#define SZ_ROW 64
#define SZ_COL 64
static char buffer[SZ_ROW][SZ_COL];
static unsigned rowIndex, colIndex;

void CcdppInitialize() {
    rowIndex = -1;
    colIndex = -1;
}

void CcdppCapture() {
    char bias = 0;
    CcdCapture();
    for(rowIndex=0; rowIndex<SZ_ROW; rowIndex++) {
        for(colIndex=0; colIndex<SZ_COL; colIndex++) {
            buffer[rowIndex][colIndex] = CcdPopPixel();
        }
        bias = (CcdPopPixel() + CcdPopPixel()) / 2;
        for(colIndex=0; colIndex<SZ_COL; colIndex++) {
            buffer[rowIndex][colIndex] -= bias;
        }
    }
    rowIndex = 0;
    colIndex = 0;
}

char CcdppPopPixel() {
    char pixel = buffer[rowIndex][colIndex];
    if( ++colIndex == SZ_COL ) {
        colIndex = 0;
        if( ++rowIndex == SZ_ROW ) {
            colIndex = -1;
            rowIndex = -1;
        }
    }
    return pixel;
}
```

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

```c
#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);
}
```
**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
- CodecPopPixel called 64 times to retrieve encoded block from obuffer

```c
static short ibuffer[8][8], obuffer[8][8], idx;
void CodecInitialize(void) { idx = 0; }
void CodecDoFdct(void) {
  int x, y;
  for(x=0; x<8; x++) {
    for(y=0; y<8; y++)
      obuffer[x][y] = FDCT(x, y, ibuffer);
  }
  idx = 0;
}
void CodecPushPixel(short p) {
  if( idx == 64 ) idx = 0;
  ibuffer[idx / 8][idx % 8] = p; idx++;
}
short CodecPopPixel(void) {
  short p;
  if( idx == 64 ) idx = 0;
  p = obuffer[idx / 8][idx % 8]; idx++;
  return p;
}
```

**CODEC (cont.)**

- Implementing FDCT formula
  \[
  C(h) = \begin{cases} \frac{1}{\sqrt{2}} & h = 0 \\ 1 \end{cases}
  
  F(u,v) = \frac{1}{8} \sum_{x=0}^{7} \sum_{y=0}^{7} D_{xy} \times \cos\left(\frac{\pi}{16}(2u + 1)u\right) \times \cos\left(\frac{\pi}{16}(2y + 1)v\right)
  
  Only 64 possible inputs to \text{COS}, so table can be used to save performance time
  - Floating-point values multiplied by 32.678 and rounded to nearest integer
  - 32.678 chosen in order to store each value in 2 bytes of memory
  - Fixed-point representation explained more later

  FDCT unrolls inner loop of summation, implements outer summation as two consecutive loops

```c
static const short COS_TABLE[8][8] = {
  { 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, -27245, -12539,  32138 },
  { 32768, -18204, -12539,  32138, -23170,  -6392,  30273, -27245 },
  { 32768, -27245,  12539,   6392, -23170,  32138, -30273,  18204 },
  { 32768, -32138,  30273, -27245,  23170, -18204,  12539,  -6392 }
};
static int FDCT(int u, int v, short img[8][8]) {
  double s[8], r = 0; int x;
  for(x=0; x<8; x++)
    s[x] = img[x][0] * COS(0, v) + img[x][1] * COS(1, v) +
       img[x][2] * COS(2, v) + img[x][3] * COS(3, v) +
       img[x][4] * COS(4, v) + img[x][5] * COS(5, v) +
       img[x][6] * COS(6, v) + img[x][7] * COS(7, v);
  for(x=0; x<8; x++)
    r += s[x] * COS(x, u);
  return (short)(r * .25 * C(u) * C(v));
}
```

```c
static 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
- CntrlInitialize for consistency with other modules only
- CntrlCaptureImage uses CCDPP module to input image and place in buffer
- CntrlCompressImage breaks the 64 x 64 buffer into 8 x 8 blocks and performs FDCT on each block using the CODEC module
  - Also performs quantization on each block
- CntrlSendImage transmits encoded image serially using UART module

```c
void CntrlSendImage(void) {
    for(i=0; i<SZ_ROW; i++)
        for(j=0; j<SZ_COL; j++) {
            temp = buffer[i][j];
            UartSend(((char*)&temp)[0]);    /* send upper byte */
            UartSend(((char*)&temp)[1]);    /* send lower byte */
        }
}
```

```
#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;
```

```c
void CntrlCaptureImage(void) {
    CcdppCapture();
    for(i=0; i<SZ_ROW; i++)
        for(j=0; j<SZ_COL; j++)
            buffer[i][j] = CcdppPopPixel();
}
```

```c
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();
            buffer[i*8+k][j*8+l] >>= 6;
        }
}
```

**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();
    return 0;
}
```
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
    - Usually satisfies power, size, and time-to-market constraints
    - If timing constraint not satisfied then later implementations could:
      - use single-purpose processors for time-critical functions
      - rewrite functional specification

Implementation 1: Microcontroller alone

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

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

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

![Block diagram of Intel 8051 processor core](image-url)
**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

**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
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

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 its ready for next byte
      - UART may be much slower than processor
  - Similar modification for CCDPP code

- All other modules untouched

Original code from system-level model

```c
#include <stdio.h>
static FILE *outputFileHandle;
void UartInitialize(const char *outputFileName) {
  outputFileHandle = fopen(outputFileName, "w");
}
void UartSend(char d) {
  fprintf(outputFileHandle, "%i\n", (int)d);
}
```

Rewritten UART module

```c
static unsigned char U_TX_REG _at_ 65535;
static unsigned char U_STAT_REG _at_ 65534;
void UARTInitialize(void) {}
void UARTSend(unsigned char d) {
  U_TX_REG = d;
  while( U_STAT_REG == 1 ) { /* busy wait */
    // do nothing
  }
}
```

Software
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

Obtaining design metrics of interest

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
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

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
        - mult, add
      - 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
**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 $2^s$ (# of bits used for fractional part)
  - Round to nearest integer
  - E.g., represent 3.14 as 8-bit integer with 4 bits for fraction
    - $2^4 = 16$
    - $3.14 \times 16 = 50.24 \approx 50 = 00110010$
    - 16 ($2^4$) possible values for fraction, each represents 0.0625 (1/16)
    - Last 4 bits (0010) = 2
    - $2 \times 0.0625 = 0.125$
    - $3(0011) + 0.125 = 3.125 \approx 3.14$ (more bits for fraction would increase accuracy)

**Fixed-point arithmetic operations**

- Addition
  - Simply add integer representations
  - E.g., $3.14 + 2.71 = 5.85$
    - $3.14 \rightarrow 50 = 00110010$
    - $2.71 \rightarrow 43 = 00101011$
    - $50 + 43 = 93 = 01011101$
    - $5(0101) + 13(1101) \times 0.0625 = 5.8125 \approx 5.85$

- Multiply
  - Multiply integer representations
  - Shift result right by # of bits in fractional part
  - E.g., $3.14 \times 2.71 = 8.5094$
    - $50 \times 43 = 100001100110$
    - $50 + 43 = 93 = 01011101$
    - $5(0101) + 13(1101) \times 0.0625 = 5.8125 \approx 5.85$

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

```c
static unsigned char C(int h) { return h ? 64 : ONE_OVER_SQRT_TWO;}

static int F(int u, int v, short img[8][8]) {
    long s[8], r = 0;
    unsigned char x, j;
    for(x=0; x<8; x++) {
        s[x] = 0;
        for(j=0; j<8; j++)
            s[x] += (img[x][j] * COS_TABLE[j][v]) >> 6;
    }
    for(x=0; x<8; x++) r += (s[x] * COS_TABLE[x][u]) >> 6;
    return (short)(((r * (((16*C(u)) >> 6) *C(v)) >> 6)) >> 6) >> 6;
}
```

---

**Implementation 3: Microcontroller and CCDPP/Fixed-Point DCT**

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

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

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

Rewritten CODEC software

```c
static unsigned char xdata C_STAT_REG _at_ 65527;
static unsigned char xdata C_CMND_REG _at_ 65528;
static unsigned char xdata C_DATAI_REG _at_ 65529;
static unsigned char xdata C_DATAO_REG _at_ 65530;

void CodecInitialize(void) {}
void CodecPushPixel(short p) { C_DATAO_REG = (char)p; }
short CodecPopPixel(void) {
    return ((C_DATAI_REG << 8) | C_DATAI_REG);
}
void CodecDoF dct(void) {
    C_CMND_REG = 1;
    while( C_STAT_REG == 1 ) { /* busy wait */ }
}
```
Implementation 4: Microcontroller and CCDPP/DCT

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

Summary of implementations

<table>
<thead>
<tr>
<th></th>
<th>Implementation 1</th>
<th>Implementation 2</th>
<th>Implementation 3</th>
<th>Implementation 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance (seconds)</td>
<td>9.1</td>
<td>1.5</td>
<td>0.099</td>
<td>0.099</td>
</tr>
<tr>
<td>Power (watt)</td>
<td>0.033</td>
<td>0.033</td>
<td>0.040</td>
<td>0.040</td>
</tr>
<tr>
<td>Total gates</td>
<td>98,000</td>
<td>90,000</td>
<td>128,000</td>
<td>128,000</td>
</tr>
<tr>
<td>Energy (joule)</td>
<td>0.30</td>
<td>0.050</td>
<td>0.0040</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?
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!