Hardware-Software Codesign

張雲南
中山大學資工系
Outline

- Hardware Software Codesign
  - Motivation
  - Domain
  - Problem
- System Specification
- Partition methodology
  - Partition strategies
  - Partition Algorithm
- Design Quality Estimation
- Specification Refinement
- Co-synthesis
Motivation for Co-design

- Cooperative approach to the design of HW/SW system is required.
  - In tradition, HW/SW are developed independently and integrated later.
  - Very little interaction occurring before system integration.
- Co-design improves overall system performance, reliability, and cost effectiveness.
- Co-design benefits the design of embedded systems, which contain HW/SW tailored for a particular applications.
  - The trend of System-on-chip.
  - Support growing complexity of embedded systems.
  - Take advantage of advanced in tools and technologies.
Motivations for Codesign (cont.)

- The importance of codesign in designing hardware/software systems:
  - Improves design quality, design cycle time, and cost
    - Reduces integration and test time
  - Supports growing complexity of embedded systems
  - Takes advantage of advances in tools and technologies
    - Processor cores
    - High-level hardware synthesis capabilities
    - ASIC development
Driving factors for co-design

- Instruction Set Processors (ISPs) available as cores in many design kits.
- System on Silicon
  - Many transistors available in typical processes.
- Increasing capacity of field programmable devices.
- Efficient C compilers for embedded processors.
- Hardware synthesis capability.
Co-design Applications

● Embedded Systems
  – Consumer electronics
  – Telecommunications
  – Manufacturing control
  – Vehicles

● Instruction Set Architectures
  – Application-specific instruction set processors.
    • Instructions of the CPU are also the target of design.
  – Dynamic compiler provides the tradeoff of HW/SW.

● Reconfigurable Systems.
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)
Classic Design & Co-Design

classic design

co-design

HW

SW

HW

SW
Current Hardware/Software Design Process

- Basic features of current process:
  - System immediately partitioned into hardware and software components
  - Hardware and software developed separately
  - “Hardware first” approach often adopted

- 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
Incorrect Assumptions in Current Hardware/Software Design Process

- Hardware and software can be acquired separately and independently, with successful and easy integration of the two later.
- Hardware problems can be fixed with simple software modifications.
- Once operational, software rarely needs modification or maintenance.
- Valid and complete software requirements are easy to state and implement in code.
Typical co-design process

- System Description (Functional)
  - Concurrent processes
  - Programming languages
- FSM-directed graphs
- HW/SW Partitioning
  - Unified representation (Data/control flow)
- SW
  - Software Synthesis
- Interface Synthesis
- Hardware Synthesis
- HW
- System Integration
  - Instruction set level
  - HW/SW evaluation
- Another HW/SW partition
Co-design Process

- System specification.
- HW/SW partition.
  - 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 synthesis.
  - Operational scheduling in hardware
  - Instruction scheduling in compiler
  - Process scheduling in operating systems
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)
Requirements for the Ideal Codesign Environment (cont.)

- 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
Cross-fertilization Between Hardware and Software Design

- Fast growth in both VLSI design and software engineering has raised awareness of similarities between the two
  - Hardware synthesis
  - Programmable logic
  - Description languages

- Explicit attempts have been made to “transfer technology” between the domains
Conventional Codesign Methodology

- Analysis of Constraints and Requirements
  - System Specs.
  - HW/SW Partitioning
    - Hardware Descript.
    - Software Descript.
      - HW Synth. and Configuration
      - Interface Synthesis
      - Software Gen. & Parameterization
      - HW/SW Interfaces
      - Software Modules
    - HW/SW Integration and Cosimulation
      - Configuration Modules
      - Hardware Components
      - HW/SW Interfaces
      - Software Modules
    - Integrated System
      - System Evaluation
      - Design Verification

© IEEE 1994
[Rozenblit94]
Key issues during the codesign process

- Unified representation
- HW/SW partition
  - Partition Algorithm
  - HW/SW Estimation Methods.
- HW/SW co-synthesis
  - Interface Synthesis
  - Refinement of Specification
Unified HW/SW Representation

- Unified Representation --
  - A representation of a system that can be used to describe its functionality independent of its implementation in hardware or software
  - Allows hardware/software partitioning to be delayed until trade-offs can be made
  - Typically used at a high-level in the design process
- Provides a simulation environment after partitioning is done, for both hardware and software designers to use to communicate
- Supports cross-fertilization between hardware and software domains
Specification Characteristics

- Concurrency
  - Data-driven concurrency
  - Control-driven concurrency
- State transition
- Hierarchy
- Programming constructs
- Communication
- Synchronization
- Exception
- timing


Current Abstraction Mechanisms in Hardware Systems

- Abstraction
  - The level of detail contained within the system model

- A system can be modeled at system, instruction set, register-transfer, logic, or circuit level

- A model can describe a system in the behavioral, structural, or physical domain
# Abstractions in Modeling: Hardware Systems

<table>
<thead>
<tr>
<th>Level</th>
<th>Behavior</th>
<th>Structure</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMS (System)</td>
<td>Communicating Processes</td>
<td>Processors Memories Switches (PMS)</td>
<td>Cabinets, Cables</td>
</tr>
<tr>
<td>Instruction Set (Algorithm)</td>
<td>Input-Output</td>
<td>Memory, Ports Processors</td>
<td>Board Floorplan</td>
</tr>
<tr>
<td>Register-Transfer</td>
<td>Register Transfers</td>
<td>ALUs, Regs, Muxes, Bus</td>
<td>ICs Macro Cells</td>
</tr>
<tr>
<td>Logic</td>
<td>Logic Equns.</td>
<td>Gates, Flip-flops</td>
<td>Std. cell layout</td>
</tr>
<tr>
<td>Circuit</td>
<td>Network Equns.</td>
<td>Trans., Connections</td>
<td>Transistor layout</td>
</tr>
</tbody>
</table>

Start here

Work to here

© IEEE 1990

[McFarland90]
Current Abstraction Mechanisms for Software Systems

Virtual machine

A software layer very close to the hardware that hides the hardware’s details and provides an abstract and portable view to the application programmer

Attributes

– Developer can treat it as the real machine
– A convenient set of instructions can be used by developer to model system
– Certain design decisions are hidden from the programmer
– Operating systems are often viewed as virtual machines
Abstractions for Software Systems

Virtual Machine Hierarchy

• Application Programs
• Utility Programs
• Operating System
• Monitor
• Machine Language
• Microcode
• Logic Devices
Abstract Hardware-Software Model

Uses a unified representation of system to allow early performance analysis
Examples of Unified HW/SW Representations

Systems can be modeled at a high level as:

- Data/control flow diagrams
- Concurrent processes
- Finite state machines
- Object-oriented representations
- Petri Nets
Unified Representations (Cont.)

- Data/control flow graphs
  - 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

Example:

```
5  X  4

+  +  +

Control Step 1
```

```
+  +  +

Control Step 2
```

```
+  +  +

Control Step 3
```
Unified Representations (Cont.)

- Concurrent processes
  - Interactive processes executing concurrently with other processes in the system-level specification
  - Enable hardware and software modeling

- Finite state machines
  - Provide a mathematical foundation for verifying system correctness, simulation, hardware/software partitioning, and synthesis
  - Multiple FSMs that communicate can be used to model reactive real-time systems
Unified Representations (Cont.)

- Object-oriented representations:
  - Use techniques previously applied to software to manage complexity and change in hardware modeling
  - Use C++ to describe hardware and display OO characteristics
  - 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
Unified Representations (Cont.)

Object-oriented representation

Example:

3 Levels of abstraction:
Unified Representations (Cont.)

- **Petri Nets**: a system model consisting of places, tokens, transitions, arcs, and a marking
  - Places - equivalent to conditions and hold tokens
  - Tokens - represent information flow through system
  - Transitions - associated with events, a “firing” of a transition indicates that some event has occurred
  - Marking - a particular placement of tokens within places of a Petri net, representing the state of the net

Example:

![Petri Net Diagram]

- Input Places
- Output Place
- Transition
- Token
**Typical DSP Algorithm**

- **Traditional DSP**
  - Convolution/Correlation

- Filtering (FIR, IIR)

- Adaptive Filtering (Varying Coefficient)

- DCT

\[
y[n] = x[n] * h[n] = \sum_{k=0}^{M} x[k] h[n-k]
\]

\[
y[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n]
\]

\[
x[k] = e(k) \sum_{n=0}^{N} x[n] \cos\left(\frac{(2n+1)k\pi}{2N}\right)
\]
**Specification of DSP Algorithms**

- **Example**
  - \( y(n) = ax(n) + bx(n-1) + cx(n-2) \)

- **Graphical Representation Method 1: Block Diagram (Data-path architecture)**
  - Consists of functional blocks connected with directed edges, which represent data flow from its input block to its output block.
Graphical Representation Method 2: Signal-Flow Graph

- SFG: a collection of nodes and directed edges
- Nodes: represent computations and/or task, sum all incoming signals
- Directed edge (j, k): denotes a **linear transformation** from the input signal at node j to the output signal at node k
- Linear SFGs can be transformed into different forms without changing the system functions.
  - *Flow graph reversal* or *transposition* is one of these transformations (Note: only applicable to single-input-single-output systems)
Signal-Flow Graph

- Usually used for linear time-invariant DSP systems representation
- Example:
Graphical Representation Method 3: Data-Flow Graph

- DFG: nodes represent computations (or functions or subtasks), while the directed edges represent data paths (data communications between nodes), each edge has a nonnegative number of delays associated with it.
- DFG captures the data-driven property of DSP algorithm: any node can perform its computation whenever all its input data are available.
**Data-Flow Graph**

- Each edge describes a precedence constraint between two nodes in DFG:
  - Intra-iteration precedence constraint: if the edge has zero delays
  - Inter-iteration precedence constraint: if the edge has one or more delays (Delay here represents iteration delays.)
  - DFGs and Block Diagrams can be used to describe both linear single-rate and nonlinear multi-rate DSP systems

**Fine-Grain DFG**

![Data-Flow Graph Diagram]

- **x(n)** → **D** → **b** → **D** → **c** → **y(n)**
Examples of DFG

- Nodes are complex blocks (in Coarse-Grain DFGs)

- Nodes can describe expanders/decimators in Multi-Rate DFGs
**Graphical Representation Method 4: Dependence Graph**

- DG contains computations for all iterations in an algorithm.
- DG does not contain delay elements.
- Edges represent precedence constraints among nodes.
- Mainly used for systolic array design.
System partitioning

- System functionality is implemented on system components
  - ASICs, processors, memories, buses

- Two design tasks:
  - **Allocate** system components or ASIC constraints
  - **Partition** functionality among components

- Constraints
  - Cost, performance, size, power

- Partitioning is a central system design task
Hardware/Software Partitioning

● Definition
   - The process of deciding, for each subsystem, whether the required functionality is more advantageously implemented in hardware or software

● Goal
   - To achieve a partition that will give us the required performance within the overall system requirements (in size, weight, power, cost, etc.)

● This is a multivariate optimization problem that when automated, is an NP-hard problem
HW/SW Partitioning Issues

- Partitioning into hardware and software affects overall system cost and performance

- Hardware implementation
  - Provides higher performance via hardware speeds and parallel execution of operations
  - Incurs additional expense of fabricating ASICs

- Software implementation
  - May run on high-performance processors at low cost (due to high-volume production)
  - Incurs high cost of developing and maintaining (complex) software
Structural vs. functional partitioning

- **Structural:** Implement structure, then partition
  - Good for the hardware (size & pin) estimation.
  - Size/performance tradeoffs are difficult.
  - Suffer for large possible number of objects.
  - Difficult for HW/SW tradeoff.

- **Functional:** Partition function, then implement
  - Enables better size/performance tradeoffs
  - Uses fewer objects, better for algorithms/humans
  - Permits hardware/software solutions
  - But, it’s harder than graph partitioning
**Partitioning Approaches**

- Start with all functionality in software and move portions into hardware which are time-critical and can not be allocated to software (*software-oriented partitioning*)

- Start with all functionality in hardware and move portions into software implementation (*hardware-oriented 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 collection of indivisible functional objects.
  - Each system component’s functionality is implemented in either hardware or software.

- An important advantage of functional partitioning is that it allows hardware/software solutions.
**Binding Software to Hardware**

- **Binding**: assigning software to hardware components
- After parallel implementation of assigned modules, all design threads are joined for system integration
  - Early binding commits a design process to a certain course
  - Late binding, on the other hand, provides greater flexibility for last minute changes
Hardware/Software System Architecture Trends

- Some operations in special-purpose hardware
  - Generally take the form of a coprocessor communicating with the CPU over its bus
    - Computation must be long enough to compensate for the communication overhead
  - May be implemented totally in hardware to avoid instruction interpretation overhead
    - Utilize high-level synthesis algorithms to generate a register transfer implementation from a behavior description

- Partitioning algorithms are closely related to the process scheduling model used for the software side of the implementation
HW/SW Partition

Formal Definition

- A hardware/software partition is defined using two sets $H$ and $S$, where $H \subseteq O$, $S \subseteq O$, $H \cap S = O$, $H \cap S = O$.

- Associated metrics:
  - $H_{size}(H)$ is the size of the hardware needed to implement the functions in $H$ (e.g., number of transistors).
  - Performance($G$) is the total execution time for the group of functions in $G$ for a given partition $\{H, S\}$.
  - Set of performance constraints, $Cons = (C_1, ..., C_m)$, where $C_j = \{G, timecon\}$, indicates the maximum execution time allowed for all the functions in group $G$ and $G \subseteq O$. 

**Performance Satisfying Partition**

- A performance satisfying partition is one for which \( \text{performance}(C_j G) \leq C_j \text{timecon} \), for all \( j=1 \ldots m \)

- Given \( O \) and \( \text{Cons} \), the hardware/software partitioning problem is to find a performance satisfying partition \( \{H,S\} \) such that \( \text{Hsize}(H) \) is minimized

- The *all-hardware* size of \( O \) is defined as the size of an all hardware partition (i.e., \( \text{Hsize}(O) \))
Basic partitioning issues

- **Specification-abstraction level: input definition**
  - Executable languages becoming a requirement
    - Although natural languages common in practice.
  - Just indicating the language is insufficient
  - Abstraction-level indicates amount of design already done
    - e.g. task DFG, tasks, CDFG, FSMD

- **Granularity: specification size in each object**
  - Fine granularity yields more possible designs
  - Coarse granularity better for computation, designer interaction
    - e.g. tasks, procedures, statement blocks, statements

- **Component allocation: types and numbers**
  - e.g. ASICs, processors, memories, buses
Basic partitioning issues (cont.)

- Metrics and estimations: "good" partition attributes
  - e.g. cost, speed, power, size, pins, testability, reliability
  - Estimates derived from quick, rough implementation
  - Speed and accuracy are competing goals of estimation

- Objective and closeness functions
  - Combines multiple metric values
  - Closeness used for grouping before complete partition
  - Weighted sum common
  - e.g. $k_1 F(\text{area}, c) + k_2 F(\text{delay}, c) + k_3 F(\text{power}, c)$

- Output: format and uses
  - e.g. new specification, hints to synthesis tool

- Flow of control and designer interaction
Issues in Partitioning (Cont.)

High Level Abstraction

Decomposition of functional objects
- Metrics and estimations
- Partitioning algorithms
- Objective and closeness functions

Component allocation

Output

Fall 2002
Specification Abstraction Levels

- Task-level dataflow graph
  - A Dataflow graph where each operation represents a task
- Task
  - Each task is described as a sequential program
- Arithmetic-level dataflow graph
  - A Dataflow graph of arithmetic operations along with some control operations
  - The most common model used in the partitioning techniques
- Finite state machine (FSM) with datapath
  - A finite state machine, with possibly complex expressions being computed in a state or during a transition
**Specification Abstraction Levels (Cont.)**

- **Register transfers**
  - The transfers between registers for each machine state are described

- **Structure**
  - A structural interconnection of physical components
  - Often called a net-list
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.
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
**Metrics in HW/SW Partitioning**

- Two key metrics are used in hardware/software partitioning
  - Performance: Generally improved by moving objects to hardware
  - Hardware size: Hardware size is generally improved by moving objects out of hardware
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., Objfct = k1 * area + k2 * delay + k3 * power
  - 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)
Partitioning Algorithm Issues

- Given a set of functional objects and a set of system components, a partitioning algorithm searches for the best partition, which is the one with the lowest cost, as computed by an objective function.

- While the best partition can be found through exhaustive search, this method is impractical because of the inordinate amount of computation and time required.

- The essence of a partitioning algorithm is the manner in which it chooses the subset of all possible partitions to examine.
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.

A, B - Local minima
C - Global minimum
Iterative Partitioning Algorithms (Cont.)

- **Greedy algorithms**
  - Only accept moves that decrease cost
  - Can get trapped in local minima

- **Hill-climbing algorithms**
  - Allow moves in directions increasing cost (retracing)
    - Through use of stochastic functions
  - Can escape local minima
  - E.g., simulated annealing
Typical partitioning-system configuration
Basic partitioning algorithms

- Random mapping
  - Only used for the creation of the initial partition.
- Clustering and multi-stage clustering
- Group migration (a.k.a. min-cut or Kernighan/Lin)
- Ratio cut
- 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
Hierarchical clustering algorithm

/* Initialize each object as a group */
for each oi loop
  pi=oi
  P=Pupi
end loop

/* Compute closenesses between objects */
for each pi loop
  for each pi loop
    ci, j=ComputeCloseness(pi,pj)
  end loop
end loop

/* Merge closest objects and recompute closenesses*/
While not Terminate(P) loop
  pi,pj=FindClosestObjects(P,C)
  P=P-pi-pjUpij
  for each pk loop
    ci, j,k=ComputeCloseness(pij,pk)
  end loop
end loop
return P
Hierarchical clustering example

Avg(10,10)=10
Avg(15,25)=20
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
Multi-stage clustering

- Start hierarchical clustering with one metric and then continue with another metric.
- Each clustering with a particular metric is called a stage.
Group migration

- Another iteration improvement algorithm extended from two-way partitioning algorithm that suffer 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’s Algorithm

P=P_in
Loop
  /*Initialize*/
  prev_P=P
  prev_cost=Objfct(P)
  bestpart_cost=
  for each o_i loop
    o_i.moved=false
  end loop

  /*create a sequence of n moves*/
  for i in 1 to n loop
    bestmove_cost=
    for each o_i not o_i.moved loop
      cost=Objfct(Move(P, o_i))
      if cost<bestmove_cost then
        bestmove_cost=cost
        bestmove_obj= o_i
      end if
    end loop
    P=Move(P,bestmove_obj)
    bestmove_obj.moved=true
    /*Save the best partition during the sequence*/
    if bestmove_cost<bestpart_cost then
      bestpart_P=P
      bestpart_cost=bestmove_cost
    end if
  end loop

  /*Update P if a better cost was found, else exit*/
  If bestpart_cost<prev_cost then
    P=bestpart_P
  else return prev_P
  end if
end loop
Ratio Cut

- A constructive algorithm that groups objects until a terminal condition has been met.
- A new metric ratio is defined as
  \[
  \text{ratio} = \frac{\text{cut}(P)}{\text{size}(p_1) \cdot \text{size}(p_2)}
  \]
  - \( \text{Cut}(P) \): sum of the weights of the edges that cross p1 and p2.
  - \( \text{Size}(pi) \): size of pi.
- 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 cut sizes without grouping objects that are not close.
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

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})
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.
/*Create first generation with gen_size random partitions*/
G=∅
for i in 1 to gen_size loop
    G=GUCreateRandomPart(O)
end loop
P_best=BestPart(G)

/*Evolve generation*/
While not Terminate loop
    G=Select*G,num_sel U Cross(G,num_cross)
    Mutate(G,num_mutatae)
    If Objfct(BestPart(G))<Objfct(P_best)then
        P_best=BestPart(G)
    end if
end loop

/*Return best partition in final gereration*/
return P_best
Integer Linear Programming

- A linear program formulation consists of a set of variables, a set of linear inequalities, and a single linear function of the variables that serves as an objective function.
  - A integer linear program is a linear program in which the variables can only hold integers.
- For partition purpose, the variables are used to represent partitioning decision or metric estimations.
- Still a NP-hard problem that requires some heuristics.
Partition example

- The Yorktown Silicon compiler uses a hierarchical clustering algorithm with the following closeness as the terminal conditions:

\[
Closeness(p_i, p_j) = \left( \frac{Conn_{i,j}}{\text{MaxConn}(P)} \right)^{k_2} \cdot \left( \frac{\text{size}_{\text{max}}}{\text{Min}(\text{size}_i, \text{size}_j)} \right)^{k_3} \cdot \left( \frac{\text{size}_{\text{max}}}{\text{size}_i + \text{size}_j} \right)
\]

- \( Conn_{i,j} \) \( k_1 \cdot \text{inputs}_{i,j} + \text{wire}_{i,j} \)
- \( \text{inputs}_{i,j} \) \# of common inputs shared
- \( \text{wires}_{i,j} \) \# of op to ip and ip to op
- \( \text{MaxConn}(P) \) maximum Conn over all pairs
- \( \text{size}_i \) estimated size of group \( p_i \)
- \( \text{size}_{\text{max}} \) maximum group size allowed
- \( k_1, k_2, k_3 \) constants
Ysc partitioning example:

YSC partitioning example: (a) input (b) operation (c) operation closeness values (d)
Clusters formed with 0.5 threshold

\[
\text{Closeness}(+, =) = \frac{8+0}{8} \times \frac{300}{120} \times \frac{300}{120 + 140} = 2.9
\]

\[
\text{Closeness}(-, <) = \frac{0+4}{8} \times \frac{300}{160} \times \frac{300}{160 + 180} = 0.8
\]

All other operation pairs have a closeness value of 0. The closeness values between all operations are shown in figure 6.6(c).

Figure 6.6(d) shows the results of hierarchical clustering with a closeness threshold of 0.5 the + and = operations form one cluster, and the < and – operations from a second cluster.
Ysc partitioning with similarities (a) clusters formed with 3.0 closeness threshold (b) operation similarity table (c) closeness values With similarities (d) clusters formed.
Output Issues in Partitioning

- Any partitioning technique must define the representation format and potential use of its output
  - E.g., the format may be a list indicating which functional object is mapped to which system component
  - E.g., the output may be a revised specification
    - Containing structural objects for the system components
    - Defining a component’s functionality using the functional objects mapped to it
Flow of Control and Designer Interaction

- Sequence in making decisions is variable, and any partitioning technique must specify the appropriate sequences
  - E.g., selection of granularity, closeness metrics, closeness functions

- Two classes of interaction
  - Directives
    - Include possible actions the designer can perform manually, such as allocation, overriding estimations, etc.
  - Feedback
    - Describe the current design information available to the designer (e.g., graphs of wires between objects, histograms, etc.)
Comparing Partitions Using Cost Functions

- A cost function is a function $\text{Cost}(H, S, Cons, I)$ which returns a natural number that summarizes the overall quality of a given partition
  - $I$ contains any additional information that is not contained in $H$ or $S$ or $Cons$
  - A smaller cost function value is desired

- An iterative improvement partitioning algorithm is defined as a procedure
  
  $\text{Part\_Alg}(H, S, Cons, I, \text{Cost}())$

  which returns a partition $H'$, $S'$ such that
  
  $\text{Cost}(H', S', Cons, I) \leq \text{Cost}(H, S, Cons, I)$
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
**Typical estimation model example**

<table>
<thead>
<tr>
<th>Design model</th>
<th>Additional tasks</th>
<th>Accuracy</th>
<th>Fidelity</th>
<th>speed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Low</td>
<td>Low</td>
<td>fast</td>
</tr>
<tr>
<td>Mem</td>
<td>Mem allocation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem+FU</td>
<td>FU allocation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem+FU+Reg</td>
<td>Lifetime analysis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem+FU+Reg+Mux</td>
<td>FU binding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem+FU+Reg+Mux+Wiring</td>
<td>Floorplanning</td>
<td>high</td>
<td>high</td>
<td>slow</td>
</tr>
</tbody>
</table>
Accuracy vs. Speed

- Accuracy: difference between estimated and actual value
  \[ A = 1 - \frac{|E(D) - M(D)|}{M(D)} \]
  \[ |E(D)|, |M(D)| : \text{estimated \& measured value} \]

- 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:

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

- \((A, B) = E(A) > E(B), M(A) < M(B)\) ✗
- \((B, C) = E(B) < E(C), M(B) > M(C)\) ✗
- \((A, C) = E(A) < E(C), M(A) < M(C)\) ○

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
Hardware design model
Clock cycles metric

- Selection of a clock cycle before synthesis will affect the practical execution time and the hardware resources.
- Simple estimation of clock cycle is based on maximum-operator-delay method. 

\[ clk(MOD) = \text{Max}_{all t_i} (\text{delay}(t_i)) \]

- The estimation is simple but may lead to underutilization of the faster functional units.
- Clock slack represents the portion of the clock cycle for which the functional unit is idle.
Clock cycle estimation
Clock slack and utilization

- **Slack**: portion of clock cycle for which FU is idle

  \[
  \text{slack} (\text{clk}, \text{ti}) = \left( \left\lceil \frac{\text{delay (ti)}}{\text{dk}} \right\rceil \times \text{dk} \right) - \text{delay (ti)}
  \]

- **Average slack**: FU slack averaged over all operations

  \[
  \text{ave_slack} = \frac{\sum_{i=1}^{T} \left[ \text{occur (ti)} \times \text{slack(clk,ti)} \right]}{\sum_{i=1}^{T} \left[ \text{occur (ti)} \right]}
  \]

- **Clock utilization**: % of clock cycle utilized for computations

  \[
  \text{utilization} = 1 - \frac{\text{ave_slack}(\text{clk})}{\text{clk}}
  \]
Clock utilization

\[ \text{ave\_slack(65 ns)} = 6 + 2 + 2 = 24.4 \text{ ns} \]

\[ \text{utilization(65 ns)} = 1 - \frac{24.4}{65.0} = 62\% \]
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:

\[
csteps(n_j) = \max_{t_i \in T} \frac{\text{occur}(t_j)}{\text{num}(t_j)} \cdot \text{clocks}(t_i)
\]

- The total number of control steps for behavior B is

\[
csteps(B) = \max_{n_i \in N} csteps(n_j)
\]
Operator-use method Example

- Differential-equation example:
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.
Scheduling for DSP algorithms

- Scheduling: assigning nodes of DFG to control times
- Resource allocations: assigning nodes of DFG to hardware (functional units)
- High-level synthesis
  - Resource-constrained synthesis
  - Time-constrained synthesis
Classification of scheduling algorithms

Iterative/Constructive Scheduling Algorithms
- As Soon As Possible Scheduling Algorithm (ASAP)
- As Late As Possible Scheduling Algorithm (ALAP)
- List-Scheduling Algorithms

Transformational Scheduling Algorithms
- Force Directed Scheduling Algorithm
- Iterative Loop Based Scheduling Algorithm
- Other Heuristic Scheduling Algorithms
The DFG in the Example
As Soon As Possible (ASAP) Scheduling Algorithm

- Find minimum start times of each node

---

**Input:** DFG $G = (N, E)$.  
**Output:** ASAP Schedule.

1. $TS_0 = 1; /* Set initial time step */$
2. While (Unscheduled nodes exist) {
   2.1 Select a node $n_j$ whose predecessors have already been scheduled;
   2.2 Schedule node $n_j$ to time step $TS_j = \max \{ TS_i + (C_i) \}$  
     $\forall n_i \rightarrow n_j$; 
}
As Soon As Possible (ASAP) Scheduling Algorithm

- The ASAP schedule for the 2nd-order differential equation
As Late As Possible (ALAP) Scheduling Algorithm

- Find maximum start times of each node

---

Input: DFG $G = (N, E)$, IterationPeriod $= T$.
Output: ALAP Schedule.

1. $TS_0 = T; /*$ Set initial time step $*/$
2. While (Unscheduled nodes exist) {
   2.1 Select a node $n_i$ whose successors have already been scheduled;
   2.2 Schedule node $n_i$ to time step $TS_i = \min \{ TS_j - (C_i) \}$
      $\forall n_i \rightarrow n_j$;
}
As Late As Possible (ALAP) Scheduling Algorithm

- The ALAP schedule for the 2nd-order differential equation
**List-Scheduling Algorithm (resource-constrained)**

- A simple list-scheduling algorithm that prioritizes nodes by decreasing criticalness (e.g. scheduling range)

---

**Input:** DFG $G = (N, E)$, $R = (FU)$.

**Output:** Final Schedule.

1. $TS_0 = 1$; /* Set initial time step */
2. While (Unscheduled nodes exist) {
   2.1 Locate all nodes whose predecessors have already been scheduled and place into list $L$;
   2.2 Sort nodes in $L$ by decreasing criticalness;
   2.3 While (L is not empty) {
      2.3.1 Select the first node $n_j$ from $L$.
      2.3.2 Determine time step $TS_j = \max \{ TS_i + (C_i) \}$
                           $\forall n_i \rightarrow n_j$;
      2.3.3 If (FUs at $TS_j$ are not full)
                     Schedule node $n_j$ to time step $TS_j$
               else
                     Remove node from $L$.  
}}
**Force Directed Scheduling Algorithm (time-constrained)**

- Transformation algorithm

---

**Input:** DFG $G = (N, E)$, *Iteration Period* = $T$.

**Output:** Final FDS Schedule.

1. While (Unscheduled nodes exist) {
   1.1 Compute the time frames for each node;
   1.2 Build the distribution graph;
   1.3 Compute the self-forces;
   1.4 Compute the predecessor and successor forces;
   1.5 Schedule the node into the time step that minimizes the total force;
}
**Force Directed Scheduling Algorithm**

- Figure (a) shows the time frame of the example DFG and the associated probabilities (obtained using ALAP and ASAP).
- Figure (b) shows the DGs for the 2nd-order differential equation.
**Force Directed Scheduling Algorithm**

\[
\text{Self\_Force}(j) = \sum_{i=S_j}^{L_j} [DG(i) \times x(i)]
\]

- **Example:**
  \[
  \text{Self\_Force}_{4}(1) = \text{Force}_{4}(1) + \text{Force}_{4}(2)
  \]
  \[
  = (DG_{M}(1) \times x_{4}(1)) + (DG_{M}(2) \times x_{4}(2))
  \]
  \[
  = (2.833 \times (1 - 0.5)) + (2.333 \times (0 - 0.5))
  \]
  \[
  = (2.833 \times (+0.5)) + (2.333 \times (-0.5))
  \]
  \[
  = +0.25
  \]
**Force Directed Scheduling Algorithm**

- Example (con’d.):
  
  \[
  \text{Self\_Force}_4(2) = \text{Force}_4(1) + \text{Force}_4(2)
  \]
  
  \[
  = (DG_M(1)\times x_4(1)) + (DG_M(2)\times x_4(2))
  \]
  
  \[
  = (2.833\times(-0.5)) + (2.333\times(+0.5))
  \]
  
  \[
  = -0.25
  \]

  \[
  \text{Succ\_Force}_4(2) = \text{Self\_Force}_8(2) + \text{Self\_Force}_8(3)
  \]
  
  \[
  = (DG_M(2)\times x_8(2)) + (DG_M(3)\times x_8(3))
  \]
  
  \[
  = (2.333\times(0-0.5)) + (0.833\times(1-0.5))
  \]
  
  \[
  = (2.333\times(-0.5)) + (0.833\times(0.5))
  \]
  
  \[
  = -0.75
  \]

  \[
  \text{Force}_4(2) = \text{Self\_Force}_4(2) + \text{Succ\_Force}_4(2)
  \]
  
  \[
  = -0.25-0.75
  \]
  
  \[
  = -1.00
  \]
Force Directed Scheduling Algorithm (another example)

A1: \( F_{a1}(0) = 0; \)
A2: \( F_{a2}(1) = 0; \)
A3: T(1): Self \( F_{a3}(1) = 1.5 \times 0.5 - 0.5 \times 0.5 = 0.5 \)
\( \quad \text{Pred } F_{m2}(1) = 0.5 \times 0.5 - 0.5 \times 0.5 = 0 \)
\( F_{a3}(1) = 0.5 \)
T(2): Self \( F_{a3}(2) = -1.5 \times 0.5 + 0.5 \times 0.5 = -0.5 \)
\( F_{a3}(2) = -0.5 \)
M1: \( F_{m1}(2) = 0; \)
M2: T(0): Self \( F_{m2}(0) = 0.5 \times 0.5 - 0.5 \times 0.5 = 0 \)
\( F_{m2}(0) = 0 \)
T(1): Self \( F_{m2}(1) = -0.5 \times 0.5 + 0.5 \times 0.5 = 0 \)
\( \quad \text{Succ } F_{a3}(2) = -1.5 \times 0.5 + 0.5 \times 0.5 = -0.5 \)
\( F_{m2}(0) = -0.5 \)
**Scheduler**

- **Critical path scheduler**
  - Based on precedence graph (intra-iteration precedence constraints)

- **Overlapping scheduler**
  - Allow iterations to overlap

- **Block schedule**
  - Allow iterations to overlap
  - Allow different iterations to be assigned to different processors.
Overlapping schedule

- Example:
  - Minimum iteration period obtained from critical path scheduler is 8 t.u.

```
(2)      (2) 3D  (2)
(2)      (2)
A  2D  C  D  E
```

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>P2</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>P3</td>
<td>C</td>
<td>C</td>
<td>E</td>
<td>C</td>
<td>C</td>
<td>E</td>
<td>C</td>
<td>E</td>
<td>C</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>p4</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>
```
Block schedule

Example:

- Minimum iteration period obtained from critical path scheduler is 20 t.u
Branching in behaviors

- Control steps maybe shared across exclusive branches
  - sharing schedule: fewer states, status register
  - non-sharing schedule: more states, no status registers
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
  \[ \text{execution}(B) = \bigoplus_{b_i \in B} \text{exectime}(b_i) \times \text{freq}(b_i) \]
Probability-based flow analysis

A := A + 1;
for I in 1 to 10 loop
  B := B + 1;
  C := C - A;
  if (D > A) then
    D := D + 2;
  else
    D := D + 3;
  end if;
  E := D * 2;
end loop;
B := B * A;
C := 3
Probability-based flow analysis

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

- 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, \( \text{avgrate}(C) \), is defined as the rate at which data is sent during the entire channel’s lifetime.
- Peak rate of a channel, \( \text{peakrate}(C) \), is defined as the rate at which data is sent in a single message transfer.
Communication rate estimation

- Total behavior execution time consists of
  - Computation time $comptime(B)$
    - Time required for behavior $B$ to perform its internal computation.
    - Obtained by the flow-analysis method.
  - Communication time $commt ime(B,C)$
    - Time spent by behavior to transfer data over the channel
    
    \[
    commtime(B,C) = access(B,C) \cdot portdelay(C)
    \]

- Total bits transferred by the channel,

\[
\text{total\_bits}(B,C) = access(B,C) \cdot \text{bits}(C)
\]

- Channel average rate

\[
\text{average}(C) = \frac{\text{Total\_bits}(B,C)}{comptime(B)+commtime(B,C)}
\]

- Channel peak rate

\[
\text{peakrate}(C) = \frac{\text{bits}(C)}{\text{protocol\_delay}(C)}
\]
Communication rates

- **Average channel rate**
  
  The rate of data transfer over the lifetime of behavior is calculated as:
  
  $$\text{average}(C) = \frac{56\text{bits}}{1000\text{ns}} = 56\text{Mb/s}$$

- **Peak channel rate**
  
  The rate of data transfer of a single message is calculated as:
  
  $$\text{peakrate}(C) = \frac{8\text{bits}}{100\text{ns}} = 80\text{Mb/s}$$
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
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
Clique-partitioning

\[
\begin{array}{c|c|c}
\text{Edge} & \text{Common neighbors} \\
\hline
\varepsilon_{1,3}' & 1 \\
\varepsilon_{1,4}' & 1 \\
\varepsilon_{2,3}' & 0 \\
\varepsilon_{2,5}' & 0 \\
\varepsilon_{3,4}' & 1 \\
\varepsilon_{4,5}' & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{Edge} & \text{Common neighbors} \\
\hline
\varepsilon_{13,4}' & 0 \\
\varepsilon_{2,5}' & 0 \\
\varepsilon_{4,5}' & 0 \\
\end{array}
\]

Clique: \( s_{134} = \{v_1, v_3, v_4\} \)
Clique: \( s_{25} = \{v_2, v_5\} \)
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.
Register Minimization Technique

- Lifetime analysis is used for register minimization techniques in a DSP hardware.
- A ‘data sample or variable’ is live from the time it is produced through the time it is consumed. After that it is dead.
- Linear lifetime chart: Represents the lifetime of the variables in a linear fashion.
  - Note: Linear lifetime chart uses the convention that the variable is not live during the clock cycle when it is produced but live during the clock cycle when it is consumed.
- Due to the periodic nature of DSP programs the lifetime chart can be drawn for only one iteration to give an indication of the # of registers that are needed.
For DSP programs with iteration period N

- Let the # of live variables at time partitions n ≥ N be the # of live variables due to 0-th iteration at cycles n-kN for k ≥ 0. In the example, # of live variables at cycle 7 ≥ N (=6) is the sum of the # of live variables due to the 0-th iteration at cycles 7 and (7-1)N which is 2 + 1 = 3.
Matrix transpose example

The matrix transpose example illustrates the process of transposing a matrix. The original matrix is shown as a row of characters, and the transposed matrix is shown as a column of characters. The transposer is a process that takes the original matrix and produces the transposed matrix.

<table>
<thead>
<tr>
<th>Sample</th>
<th>$T_{in}$</th>
<th>$T_{zout}$</th>
<th>$T_{diff}$</th>
<th>$T_{out}$</th>
<th>Life</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0→4</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>1→7</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>10</td>
<td>2→10</td>
</tr>
<tr>
<td>d</td>
<td>3</td>
<td>1</td>
<td>-2</td>
<td>5</td>
<td>3→5</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>4→8</td>
</tr>
<tr>
<td>f</td>
<td>5</td>
<td>7</td>
<td>2</td>
<td>11</td>
<td>5→11</td>
</tr>
<tr>
<td>g</td>
<td>6</td>
<td>2</td>
<td>-4</td>
<td>6</td>
<td>6→6</td>
</tr>
<tr>
<td>h</td>
<td>7</td>
<td>5</td>
<td>-2</td>
<td>9</td>
<td>7→9</td>
</tr>
<tr>
<td>i</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>12</td>
<td>8→12</td>
</tr>
</tbody>
</table>

To make the system causal, a latency of 4 is added to the difference, so that $T_{out}$ is the actual output time.


Circular Lifetime Chart

- Useful to represent the periodic nature of the DSP programs.
- In a circular lifetime chart of periodicity $N$, the point marked $i$ ($0 \leq i \leq N - 1$) represents the time partition $i$ and all time instances $\{(N/ + i)\}$ where $l$ is any non-negative integer.
- For example: If $N = 8$, then time $i$ time instances $\{3, 11, 19, \ldots\}$.
  - Note: Variable produced during time unit $j$ and consumed during time unit $k$ is shown to be alive from $'j + 1'$ to $'k'$.
  - The numbers in the bracket in the adjacent figure correspond to the # of live variables at each time partition.
Forward-Backward Register Allocation Technique:

Note: Hashing is done to avoid conflict during backward allocation.
Steps of Register Allocation

- Determine the minimum number of registers using lifetime analysis.
- Input each variable at the time step corresponding to the beginning of its lifetime. If multiple variables are input in a given cycle, these are allocated to multiple registers with preference given to the variable with the longest lifetime.
- Each variable is allocated in a forward manner until it is dead or it reaches the last register. In forward allocation, if the register \( i \) holds the variable in the current cycle, then register \( i + 1 \) holds the same variable in the next cycle. If \( (i + 1) \)-th register is not free then use the first available forward register.
- Being periodic the allocation repeats in each iteration, so hash out the register \( R_j \) for the cycle \( l + N \) if it holds a variable during cycle \( l \).
- For variables that reach the last register and are still alive, they are allocated in a backward manner on a first come first serve basis.
- Repeat previous two steps until the allocation is complete.
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 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
Computing datapath area

- Bit-sliced datapath

\[ L_{bit} = \text{tr}(DP) \]
\[ H_{rt} = \frac{\text{nets}}{\text{nets\_per\_track}} \]
\[ \text{area(bit)} = L_{bit} (H_{cell} + H_{rt}) \]
\[ \text{area}(DP) = \text{bitwidth}(DP) \times \text{area(bit)} \]
Pin estimation

- Number of wires at behavior’s boundary depends on
  - Global data
  - Port accessed
  - Communication channels used
  - Procedure calls
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

**dmem3 = dmem1 + dmem2**

#### 8086 Instructions

<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>

#### 68020 Instructions

<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>5</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Technology File for 8086

<table>
<thead>
<tr>
<th>Generic Instruction</th>
<th>Execution Time</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmem3 = dmem1 + dmem2</td>
<td>35 clocks</td>
<td>10 bytes</td>
</tr>
</tbody>
</table>

#### Technology File for 68020

<table>
<thead>
<tr>
<th>Generic Instruction</th>
<th>Execution Time</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmem3 = dmem1 + dmem2</td>
<td>22 clocks</td>
<td>6 bytes</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:
    \[ \text{exectime}(B) = \sum_{b_i \in B} \text{exectime}(b_i) \times \text{freq}(b_i) \]
    - accounts for compiler optimizations
  - accounts for compiler optimizations

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

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

- Variable folding:
  - Implementing each variable in a memory with a fixed word size

- Memory address translation
  - Assignment of addresses to each variable in group
  - Update references to variable by accesses to memory
Variable folding

```vhdl
variable A : bit_vector(3 downto 0);
variable B : bit_vector(15 downto 0);
variable C : bit_vector(11 downto 0);
variable D : bit_vector(11 downto 0);
```

8-bit Memory

```
\begin{center}
\begin{tikzpicture}
\node (A) at (0,0) {A(3 downto 0)};
\node (B) at (0,-1) {B(7 downto 0)};
\node (C) at (0,-2) {B(15 downto 8)};
\node (D) at (0,-3) {C(7 downto 0)};
\node (E) at (0,-4) {C(11 downto 8)};
\node (F) at (0,-5) {D(5 downto 0)};
\node (G) at (0,-6) {D(11 downto 6)};
\end{tikzpicture}
\end{center}
```

to variable C in memory

```
\begin{center}
\begin{tikzpicture}
\node (A) at (0,0) {11};
\node (B) at (0,-1) {8};
\node (C) at (0,-2) {7};
\node (D) at (0,-3) {0};
\node (E) at (1,-1) {4x1};
\node (F) at (1,-2.5) {7.4};
\node (G) at (1,-3.5) {3.0};
\end{tikzpicture}
\end{center}
```

to variable D in memory

```
\begin{center}
\begin{tikzpicture}
\node (A) at (0,0) {11};
\node (B) at (0,-1) {6};
\node (C) at (0,-2) {5};
\node (D) at (0,-3) {0};
\node (E) at (1,-2) {6x1};
\node (F) at (1,-3) {5.0};
\end{tikzpicture}
\end{center}
```
Memory address translation

Original specification

```plaintext
variable J, K : integer := 0;
variable V : IntArray (63 downto 0);
....
V(K) := 3;
X := V(36);
V(J) := X;
....
for J in 0 to 63 loop
SUM := SUM + V(J);
end loop;
....
```

Refined specification

```plaintext
variable J, K : integer := 0;
variable V : IntArray (63 downto 0);
....
V(K) := 3;
X := V(36);
V(J) := X;
....
for J in 0 to 63 loop
SUM := SUM + V(J);
end loop;
....
```

V (63 downto 0) ↓
MEM(163 downto 100)

Assigning addresses to V

Original specification

```plaintext
variable J, K : integer := 0;
variable MEM : IntArray (255 downto 0);
....
MEM(K +100) := 3;
X := MEM(136);
MEM(J) := X;
....
for J in 0 to 63 loop
SUM := SUM + MEM(J);
end loop;
....
```

Refined specification

```plaintext
variable J, K : integer := 100;
variable K : integer := 0;
variable MEM : IntArray (255 downto 0);
....
MEM(K +100) := 3;
X := MEM(136);
MEM(J) := X;
....
for J in 0 to 63 loop
SUM := SUM + MEM(J);
end loop;
....
```
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
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
Communication (cont)

Inter-process communication paradigms: (a) shared memory, (b) message passing
Characterizing communication channels

- For a given behavior that sends data over channel C,
  - **Message size** $\text{bits}(C)$
    - number of bits in each message
  - **Accesses**: $\text{accesses}(B,C)$
    - number of times P transfers data over C
  - **Average rate** $\text{averate}(C)$
    - rate of data transfer of C over lifetime of behavior
  - **Peak rate** $\text{peakrate}(C)$
    - rate of transfer of single message

$\text{bits}(C) = 8$ bits
$\text{averate}(C) = \frac{24\text{bits}}{400\text{ns}} = 60\text{Mbits/s}$
$\text{peakrate}(C) = \frac{8\text{bits}}{100\text{ns}} = 80\text{Mbits/s}$
Characterizing buses

For a given bus $B$

- **Buswidth** $\text{buswidth}(B)$
  - number of data lines in $B$
- **Protocol delay** $\text{protdelay}(B)$
  - delay for single message transfer over bus
- **Average rate** $\text{averate}(B)$
  - rate of data transfer over lifetime of system
- **Peak rate** $\text{peakrate}(B)$
  - maximum rate of transfer of data on bus

$$
\text{peakrate}(C) = \frac{\text{buswidth}(B)}{\text{protdelay}(B)}
$$
Determining bus rates

- Idle slots of a channel used for messages of other channels
- To ensure that channel average rates are unaffected by bus
  \[
  \text{averate}(B) = \left\lfloor \frac{\text{averate}(C)}{B} \right\rfloor
  \]
- Goal: to synthesize a bus that constantly transfers data for channel
  \[
  \text{peakrate}(B) = \text{averate}(C)
  \]
Constraints for bus generation

- **Bus-width**: affects number of pins on chip boundaries
- **Channel average rates**: affects execution time of behaviors
- **Channel peak rates**: affects time required for single message transfer
Bus generation algorithm

- **Compute buswidth range**: \( \text{minwidth}=1, \text{maxwidth} = \text{Max}(\text{bit}(C)) \)
- **For** \( \text{minwidth}: \text{currwidth} \leq \text{maxwidth} \text{ loop} **
  - **Compute bus peak rate**:
    
    \[ \text{peakrate}(B) = \frac{\text{currwidth}}{\text{protdelay}(B)} \]
  
  - **Compute channel average rates**

    \[
    \text{commtime}(B) = \text{access}(B, C) \times [\text{currwidth} \times \text{protdelay}(B)]
    \]

    \[
    \text{average}(C) = \frac{\text{access}(B, C) \times \text{bit}(C)}{\text{commtime}(B) + \text{commtime}(B)}
    \]

    - **If** \( \text{peakrate}(B) \geq \text{averate}(C) \) then
      
      \[
      \text{if } \text{bestcost} > \text{ComputeCost}(\text{currwidth}) \text{ then}
      \]
      
      \[
      \text{bestcost} = \text{ComputeCost}(\text{currwidth})
      \]
      
      \[
      \text{bestwidth} = \text{currwidth}
      \]
Bus generation example

- Assume
  - 2 behavior accessing 16 bit data over two channels
  - Constraints specified for channel peak rates

<table>
<thead>
<tr>
<th>Channel C</th>
<th>Behavior B</th>
<th>Variable accessed</th>
<th>Bits(C)</th>
<th>Access(B, C)</th>
<th>Comptime(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH1</td>
<td>P1</td>
<td>V1</td>
<td>16 data + 7 addr</td>
<td>128</td>
<td>515</td>
</tr>
<tr>
<td>CH2</td>
<td>P2</td>
<td>V2</td>
<td>16 data + 7 addr</td>
<td>128</td>
<td>129</td>
</tr>
</tbody>
</table>
Protocol generation

- Bus consists of several sets of wires:
  - **Data lines**, used for transferring message bits
  - **Control lines**, used for synchronization between behaviors
  - **ID lines**, used for identifying the channel active on the bus
- All channels mapped to bus share these lines
- Number of data lines determined by bus generation algorithm
- Protocol generation consists of six steps
Protocol generation steps

- 1. Protocol selection
  - full handshake, half-handshake etc.

- 2. ID assignment
  - N channels require \( \log_2(N) \) ID lines

```plaintext
behavior P
  variable AD;
  begin
    ..... 
    X <= 32;
    ..... 
    MEM(AD) := X+7;
    ..... 
  end;

behavior Q
  variable COUNT;
  begin
    ..... 
    MEM(60) := COUNT;
    ..... 
  end;
```

variable X; 
  bit_vector(15 downto 0);

variable MEM : bit_vector 
  (63 downto 0, 15 downto 0);

bus B
Protocol generation steps

● 3 Bus structure and procedure definition
  – The structure of bus (the data, control, ID lines) is defined in the specification.

● 4. Update variable-reference
  – References to a variable that has been assigned to another component must be updated.

● 5. Generate processes for variables
  – Extra behavior should be created for those variables that have been sent across a channel.
Protocol generation example

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';
    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
  .....  
  SendCH0(32);
  .....  
  ReceiveCH1(Xtemp);
  SendCH2(AD,Xtemp+7);
  .....  
end;

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

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

process MEMproc
  variable MEM: array(0 to 63);
begin
  wait on B.ID;
  if (B.ID="10") then
    receiveCH2(MEM);
  elseif (B.ID="11") then
    sendCH3(MEM);
  end if;
end;

bus B
Resolving access conflicts

- System partitioning may result in concurrent accesses to a resource
  - Channels mapped to a bus may attempt data transfer simultaneously
  - Variables mapped to a memory may be accessed by behaviors simultaneously

- Arbiter needs to be generated to resolve such access conflicts

- Three tasks
  - Arbitration model selection
  - Arbitration scheme selection
  - Arbiter generation
Arbitration models

STATIC

Dynamic
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
Refinement of incompatible interfaces

- Three situation may arise if we bind functional objects to standard components:
- Neither behavior is bound to a standard component.
  - Communication between two can be established by generating the bus and inserting the protocol into these objects.
- One behavior is bound to a standard component
  - The behavior that is not associated with standard component has to use dual protocol to the other behavior.
- Both behaviors are bound to standard components.
  - An interface process has to be inserted between the two standard components to make the communication compatible.
Effect of binding on interfaces
Protocol operations

- Protocols usually consist of five atomic operations
  - waiting for an event on input control line
  - assigning value to output control line
  - reading value from input data port
  - assigning value to output data port
  - waiting for fixed time interval

- Protocol operations may be specified in one of three ways
  - Finite state machines (FSMs)
  - Timing diagrams
  - Hardware description languages (HDLs)
Protocol specification: FSMs

- Protocol operations ordered by sequencing between states
- Constraints between events may be specified using timing arcs
- Conditional & repetitive event sequences require extra states, transitions
Protocol specification: Timing diagrams

- **Advantages:**
  - Ease of comprehension, representation of timing constraints

- **Disadvantages:**
  - Lack of action language, not simulatable
  - Difficult to specify conditional and repetitive event sequences
Protocol specification: HDLs

- **Advantages:**
  - Functionality can be verified by simulation
  - Easy to specify conditional and repetitive event sequences

- **Disadvantages:**
  - Cumbersome to represent timing constraints between events

```vhdl
port ADDRp : out
bit_vector(7 downto 0);

port DATAp : in
bit_vector(15 downto 0);

port ARDYp : out bit;
port ARCVp : in bit;

port DREQp : out bit;
port DRDYp : in bit;

ADDRp <= AddrVar(7 downto 0);
ARDYp <= '1';
wait until (ARCVp = '1');
ADDRp <= AddrVar(15 downto 8);
DREQp <= '1';
wait until (DRDYp = '1');
DataVar <= DATAp;
```

```vhdl
port MADDRp : in
bit_vector(15 downto 0);

port MDATAp : out
bit_vector(15 downto 0);

port RDp : in bit;
wait until (RDp = '1');

MAddrVar := MADDRp ;
wait for 100 ns;

MDATAp <= MemVar (MAddrVar);
```
Interface process generation

- Input: HDL description of two fixed, but incompatible protocols
- Output: HDL process that translates one protocol to the other
  - i.e. responds to their control signals and sequence their data transfers
- Four steps required for generating interface process (IP):
  - Creating relations
  - Partitioning relations into groups
  - Generating interface process statements
  - Interconnect optimization
IP generation: creating relations

- Protocol represented as an ordered set of relations
- Relations are sequences of events/actions

**Protocol Pa**

```vhdl
ADDRp <= AddrVar(7 downto 0);
ARDYp <= '1';
wait until (ARCVp = '1');
ADDRp <= AddrVar(15 downto 8);
DREQp <= '1';
wait until (DRDYp = '1');
DataVar <= DATAp;
```

**Relations**

- **A1[ (true) ]**
  
  ```vhdl
  ADDRp <= AddrVar(7 downto 0)
  ARDYp <= '1'
  ```

- **A2[ (ARCVp = '1') ]**
  
  ```vhdl
  ADDRp <= AddrVar(15 downto 8)
  DREQp <= '1'
  ```

- **A3[ (DRDYp = '1') ]**
  
  ```vhdl
  DataVar <= DATAp
  ```
IP generation: partitioning relations

- Partition the set of relations from both protocols into groups.
- Group represents a unit of data transfer

<table>
<thead>
<tr>
<th>Protocol Pa</th>
<th>Protocol Pb</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 (8 bits out)</td>
<td>B1 (16 bits in)</td>
</tr>
<tr>
<td>A2 (8 bits out)</td>
<td></td>
</tr>
<tr>
<td>A3 (16 bits in)</td>
<td>B2 (16 bits out)</td>
</tr>
</tbody>
</table>

G1=(A1 A2 B1)  G2=(B1 A3)
IP generation: inverting protocol operations

- For each operation in a group, add its dual to interface process
- Dual of an operation represents the complementary operation
- Temporary variable may be required to hold data values

<table>
<thead>
<tr>
<th>Atomic operation</th>
<th>Dual operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait until (Cp = '1') (\land) (Cp &lt;= '1') (\land) (var &lt;= Dp) (\land) (Dp &lt;= var) (\land) (wait for 100 ns)</td>
<td>(Cp &lt;= '1') (\land) (wait until (Cp = '1')) (\land) (Dp &lt;= TempVar) (\land) (TempVar := Dp) (\land) (wait for 100 ns)</td>
</tr>
</tbody>
</table>

Interface Process

```plaintext
/* (group G1) */
wait until (ARDYp = '1');
TempVar1(7 downto 0) := ADDRp;
ARCVp <= '1';
wait until (DREQp = '1');
TempVar1(15 downto 8) := ADDRp;
RDp <= '1';
MADDRp <= TempVar1;
/* (group G2) */
wait for 100 ns;
TempVar2 := MDATAp;
DRDYp <= '1';
DATAp <= TempVar2;
```
IP generation: interconnect optimization

- Certain ports of both protocols may be directly connected
- Advantages:
  - Bypassing interface process reduces interconnect cost
  - Operations related to these ports can be eliminated from interface process
Transducer synthesis

- **Input**: Timing diagram description of two fixed protocols
- **Output**: Logic circuit description of transducer
- **Steps for generating logic circuit from timing diagrams**:
  - Create event graphs for both protocols
  - Connect graphs based on data dependencies or explicitly specified ordering
  - Add templates for each output node in combined graph
  - Merge and connect templates
  - Satisfy min/max timing constraints
  - Optimize skeletal circuit
Generating event graphs from timing diagrams

e.g. FIFO stack control cell
Deriving skeletal circuit from event graph

- **Advantages:**
  - Synthesizes logic for transducer circuit directly
  - Accounts for min/max timing constraints between events

- **Disadvantages:**
  - Cannot interface protocols with different data port sizes
  - Transducer not simulatable with timing diagram description of protocols
Hardware/Software interface refinement

(a) Partitioned specification

(b) Mapping to architecture
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
Cosynthesis

- Methodical approach to system implementations using automated synthesis-oriented techniques
- Methodology and performance constraints determine partitioning into hardware and software implementations
- The result is "optimal" system that benefits from analysis of hardware/software design trade-offs
Cosynthesis Approach to System Implementation
Co-synthesis

- Implementation of hardware and software components after partitioning
- Constraints and optimization criteria similar to those for partitioning
- Area and size traded-off against performance
- Cost considerations
Synthesis Flow

- HW synthesis of dedicated units
  - Based on research or commercial standard synthesis tools
- SW synthesis of dedicated units (processors)
  - Based on specialized compiling techniques
- Interface synthesis
  - Definition of HW/SW interface and synchronization
  - Drivers of peripheral devices
Co-Synthesis - The POLIS Flow

“ESTEREL” for functional specification language
Hardware Design Methodology

Hardware Design Process: Waterfall Model
Hardware Design Methodology (Cont.)

- Use of HDLS for modeling and simulation
- Use of lower-level synthesis tools to derive register transfer and lower-level designs
- Use of high-level hardware synthesis tools
  - Behavioral descriptions
  - System design constraints
- Introduction of synthesis for testability at all levels
**Hardware Synthesis**

- **Definition**
  - The automatic design and implementation of hardware from a specification written in a hardware description language

- **Goals/benefits**
  - To quickly create and modify designs
  - To support a methodology that allows for multiple design alternative consideration
  - To remove from the designer the handling of the tedious details of VLSI design
  - To support the development of correct designs
Hardware Synthesis Categories

- **Algorithm synthesis**
  - Synthesis from design requirements to control-flow behavior or abstract behavior
  - Largely a manual process

- **Register-transfer synthesis**
  - Also referred to as “high-level” or “behavioral” synthesis
  - Synthesis from abstract behavior, control-flow behavior, or register-transfer behavior (on one hand) to register-transfer structure (on the other)
  - Logic synthesis
  - Synthesis from register-transfer structures or Boolean equations to gate-level logic (or physical implementations using a predefined cell or IC library)
Hardware Synthesis
Process Overview

Specification

Behavioral Simulation

Optional RTL Simulation

Implementation

Behavioral Synthesis

Synthesis & Test Synthesis

Verification

Gate-level Simulation

Gate-level Analysis

Silicon Vendor

Place and Route

Behavioral Functional

RTL Functional

Gate

Layout

Silicon
1. Specification Analysis
2. Concurrent Design
3. System Integration
Software Design Methodology

Software Design Process: Waterfall Model
Software Design Methodology (Cont.)

- Software requirements includes both
  - Analysis
  - Specification
- Design: 2 levels:
  - System level - module specs.
  - Detailed level - process design language (PDL) used
- Coding - in high-level language
  - C/C++
- Maintenance - several levels
  - Unit testing
  - Integration testing
  - System testing
  - Regression testing
  - Acceptance testing
Software Synthesis

- Definition: the automatic development of correct and efficient software from specifications and reusable components

- Goals/benefits
  - To increase software productivity
  - To lower development costs
  - To increase confidence that software implementation satisfies specification
  - To support the development of correct programs
Why Use Software Synthesis?

- Software development is becoming the major cost driver in fielding a system.

- To significantly improve both the design cycle time and life-cycle cost of embedded systems, a new software design methodology, including automated code generation, is necessary.

- Synthesis supports a correct-by-construction philosophy.

- Techniques support software reuse.
Why Software Synthesis?

- More software $\Rightarrow$ high complexity $\Rightarrow$ need for automatic design (synthesis)
- Eliminate human and logical errors
- Relatively immature synthesis techniques for software
- Code optimizations
  - size
  - efficiency
- Automatic code generation
Software Synthesis Flow Diagram for Embedded System with Time Petri-Net

1. Vehicle Parking Management System
2. System Software Specification
3. Modeling System Software with Time Petri Net
4. Task Scheduling
5. Code Generation
6. Code Execution on an Emulation Board
7. Feedback and Modification
8. Test Report
Automatically Generate CODE

- Real-Time Embedded System Model?
  Set of concurrent tasks with memory and timing constraints!

- How to execute in an embedded system (e.g. 1 CPU, 100 KB Mem)?
  Task Scheduling!

- How to generate code?
  Map schedules to software code!

- Code optimizations?
  Minimize size, maximize efficiency!
Software Synthesis Categories

- Language compilers
  - ADA and C compilers
  - YACC - yet another compiler compiler
  - Visual Basic

- Domain-specific synthesis
  - Application generators from software libraries
Software Synthesis Examples

- Mentor Graphics Concurrent Design Environment System
  - Uses object-oriented programming (written in C++)
  - Allows communication between hardware and software synthesis tools
- Index Technologies Excelerator and Cadre’s Teamwork Toolsets
  - Provide an interface with COBOL and PL/1 code generators
- KnowledgeWare’s IEW Gamma
  - Used in MIS applications
  - Can generate COBOL source code for system designers
- MCCI’s Graph Translation Tool (GrTT)
  - Used by Lockheed Martin ATL
  - Can generate ADA from Processing Graph Method (PGM) graphs
Reference