

System Level SOC Design

Unit 3: Algorithmic Modeling



PITTSBURGH DIGITAL
GREENHOUSE
growing smart products

Unit 3: Algorithmic Modeling

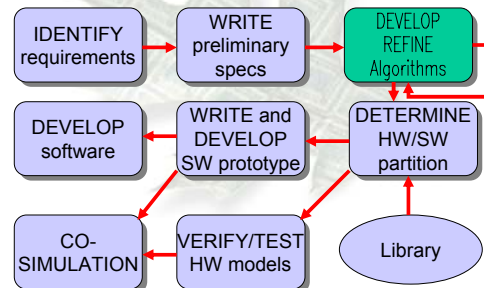
Overview

- 7 course units over 5 days
 - Unit 1: Intro. to System Level Design
 - Unit 2: How to Define Requirements
 - **Unit 3: Algorithmic Modeling**
 - Unit 4: IP Selection
 - Unit 5: Embedded Software Codesign
 - Unit 6: Hardware Codesign/Coverification
 - Unit 7: Testing

Outline

- **Topics**
 - Executable specifications
 - Timing and concurrency in algorithmic model
 - Performance analysis
- **Goal**
 - How to write an algorithmic model
 - Uses and advantages of algorithmic model
 - Understanding performance analysis

The Design Process



Algorithmic Modeling

- **Executable specification is an algorithmic model of the system**
- From algorithmic model, determine
 - Early design verification and testing
 - Trade offs among different algorithms
 - Performance requirements
 - An early idea of SW and HW IP
 - A specification to select and develop IP

Executable Specifications

- **Specifications that are executed and tested like programs**
 - C, C++
 - SystemC
 - Behavioral VHDL
- **Models behavior that is independent of hardware and software yet executable**

C, C++

- **Traditional programming language to model hardware and software at different layers of abstraction**
- Standardized, widely available tools (including open source: GNU)
- Very efficient simulation
- Experience in team
- **C++ brings object-oriented discipline**

Object Oriented Design

- **Careful design leads to “plug ‘n play”**
 - Change modules without concern for their implementation, assuming interfaces are consistent
- Less rigid and fragile designs - *flexible*
- More reusable
- What if scenarios, quick changes to operational flow by plugging in new objects

Object Oriented Design

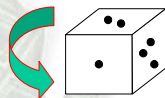
- **Model data transformations and design blocks as “objects”.**
- **Objects are independent**
 - Models functionality of specification component (*behavior*)
 - Well defined interactions (*interface*)
 - Own local data (*state*)
 - Implementation hidden (*encapsulation*)

Object Oriented Design with C++

- **C++ classes provide encapsulation**
 - Local persistent data
 - Methods to access and manipulate data
 - Implementation hidden
- Classes correspond to categories of *data transformations* and *communication* in an operational flow
- Objects are instantiations of classes with specific values

OO Dice

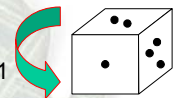
6 faces (an object too!)
one face can be showing
can roll the die



```
class die {  
    private Face currentFace;  
    void roll() { face = new Face(random(1,6)); };  
    Face showing() { return face };  
}
```

Behavior Separate From Interface

An unfair die
Always comes up with 1

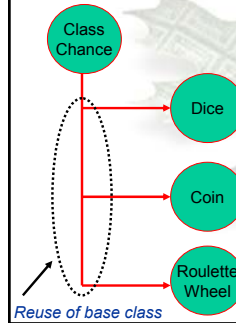


```
class UnfairDie {  
    private Face currentFace;  
    void roll() { face = new Face(1); };  
    Face showing() { return face };  
}
```

Inheritance

- **Base Class provides**
 - Interface
 - Behavior
 - State
 that can be inherited by other classes
- **Re-use of classes, rapid development, base class is well tested and defined**

Inheritance



```

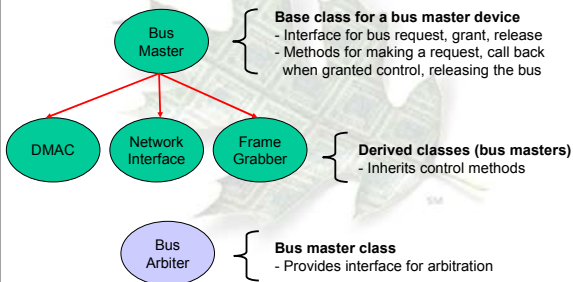
class Chance {
    ...
    int chance() {}; // random chance
    ...
}

class Dice : Chance {
    int chance() { return random face; }
}

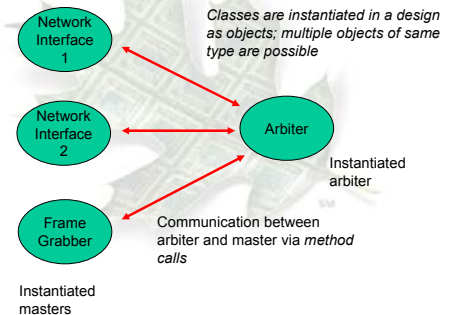
class Coin : Chance {
    int chance() { return random side; }
}

class Roulette : Chance {
    int chance() { return random slot; }
}
    
```

Example: Bus Arbiter



Instantiated Objects



Bus Master and Arbiter Interfaces

Bus Master Base Class

```

method void requestBus();
method void releaseBus();
method void callbackGranted();
method BusMasterID getID();
bool hasBus, requestedBus;
BusMasterID id;
BusArbiter *arbiter;
    
```

```

void requestBus () {
    arbiter->request(this);
    requestedBus = true;
    hasBus = false;
}
    
```

Bus Arbiter Base Class

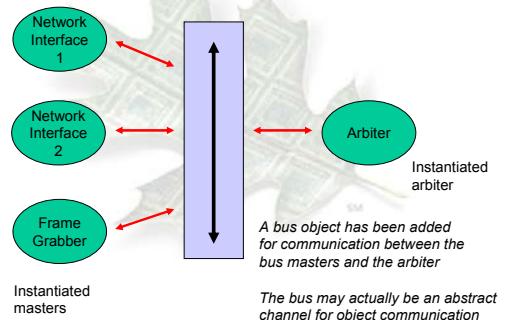
```

method void request(BusMaster *mast);
method void release(BusMaster *mast);
method void grantARequest();
method bool isBusBusy();
BusMaster *masters[];
BusMaster *master;
    
```

```

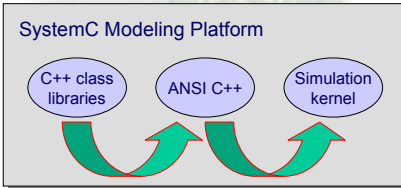
void grantARequest() {
    ...
    id = selectHighestPriority();
    master = id;
    master->callbackGranted();
    ...
}
    
```

Refinement of Bus Example

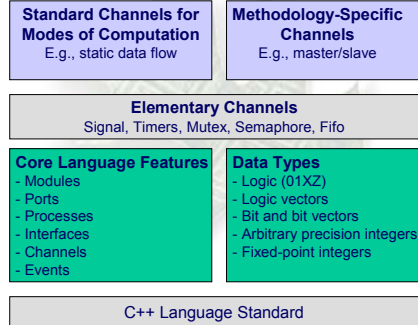


SystemC

- Aiming to be de facto standard for system-level design
- Design at the system-behavioral and register-transfer-levels



SystemC Language Architecture



SystemC vs. HDLs

- **Includes structural modeling**
 - Modules (entities), ports, signals
- **Includes a fixed-point integer data type**
- **Communication modeling for system-level design**
 - Channels (container for communication and synchronization)
 - Interfaces (channel access methods)
 - Events (low-level synchronization)

Behavioral VHDL

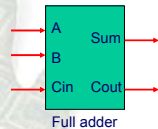
- **Model functional behavior of system**
 - An abstraction of low-level behavior
 - May include timing
- **Model based on processes**
 - Sequential assignment statements
 - Flow control: IF, WHILE, FOR, CASE
 - Process control: WAIT (ON, UNTIL, FOR)
 - No signals!

Behavioral VHDL

ARCHITECTURE example OF full_adder IS
 – No signals in declaration

```

BEGIN
  Summation: PROCESS (A, B, Cin)
  BEGIN
    Sum <= A XOR B XOR Cin;
  END PROCESS Summation;
  Carry: PROCESS (A, B, Cin)
  Cout <= (A AND B) OR (A AND Cin) OR (B AND Cin);
  END PROCESS Carry;
END example;
```



Timing and Concurrency

- **Timing**
 - Refined from functional model
 - High level notion of when something has to be done and how long it will take
- **Concurrency**
 - If model doesn't yet include concurrency, it may be introduced at this point
 - Modeled with processes

Timing and Concurrency

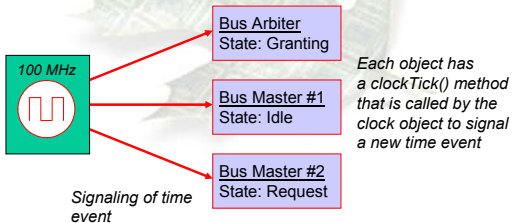
- **Timing and Concurrency**
 - Events and callbacks in model
 - Multithreading
- **Simulation Engine**
 - Global time update simulation
 - Event-driven simulation

Global Time Update

- **Simulation time clock** (a more abstract version of event-driven simulation)
- **Model objects have “time event” call back**
 - **Callback** - for registered object, another object responds with a **method call** to that registered object
 - On **clock tick**, an object is **notified that a clock event** has occurred
 - Model transformations **implement FSM**, where state is advanced on each clock tick

Global Time Update

- Transformations have a “time event” call back
 - On each simulation clock tick, an object is notified that a clock event has occurred



Global Time Update

- **Model transformation behaviorally**
 - May implement portion of FSM
 - When inputs are captured (on specific cycles), then the behavior of the transformation can be done
 - Model writes output on specific cycles
- Faithfully reproduces timing behavior from external view, but internally timing does not need to be cycle accurate

Example: Decompress a Message

```
void clockTick() {
  State IDLE:
  if (inputMessage == True) state = RECEIVE;
  State RECEIVE:
  Message M = receive(); state = DECOMPRESS;
  State DECOMPRESS:
  DecompressedMessage dM = decompress(M);
  counter = 10; state = COUNTDOWN;
  State COUNTDOWN:
  counter = counter - 1;
  if (counter == 0) state = SEND;
  State SEND:
  send(dM); state = IDLE;
}
```

Wait for message

Receive message

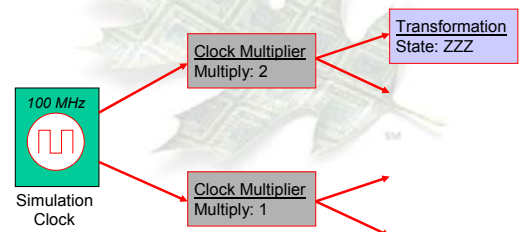
Do the decompress behavior right away but don't send yet

Count down the latency of doing the decompression

Send message

Scaling the Simulation Clock

- Easy to modify timing to fit an individual component - can have clock multipliers



Global Time Update

- **Advantages**
 - Simple, nice conceptual model
 - Easy to quickly code, change behavior
 - Fits OO paradigm
- **Disadvantages**
 - May be inefficient (e.g., count down timer)
 - Simplistic model with all timing managed internally to a transformation
 - Concurrency is simplistic

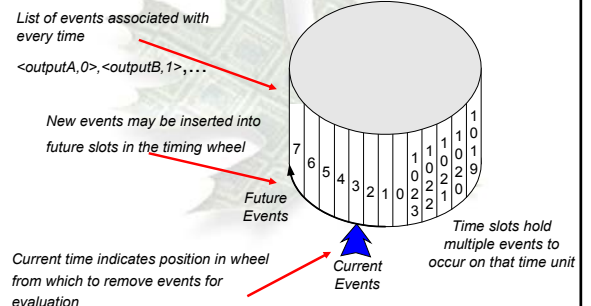
Event-Driven Simulation

- Global time update model
 - Objects receive a clock tick every cycle regardless of whether they have work.
- **Event-driven simulation**
 - Only evaluate when necessary
- Events scheduled when input/output changes
 - Event: Value, time (when), and new value
 - Handling an event generates more events

Timing Wheel

- **Events put into a “to do list”**
- List is a circular queue (“timing wheel”)
- List ordered with major time unit (e.g., 0.1 ns)
- Fractional time units are listed off the base major time unit

Timing Wheel



Model with Concurrency

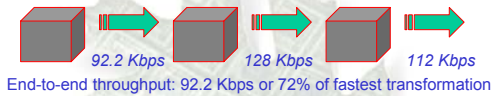
- Treat data transformations as individual, communicating processes
- Simulation engines often provide multithreading (e.g., C-Bridge runs each C model as its own lightweight process)



Performance Analysis

- **Requirements of transformations in the operational flows**
 - **Performance and communication requirements**
- **Throughput**: Total amount of work done in a given amount of time
- **Latency**: Total amount of time to complete a task

Throughput



- Performance of all transformations determines *end-to-end throughput*
- Each transformation affects the throughput
- Maximum end-to-end throughput is limited by the slowest transformation

Types of Throughput

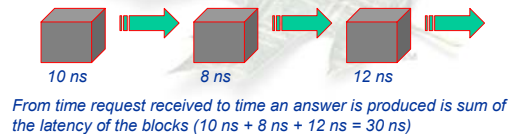
- **Peak throughput:** Maximum data rate ever achieved by a given transformation
 - Rate communication channel supports
- **Maximum throughput:** Data rate with overhead
 - Affects amount of buffering
- **Average throughput:** The typical throughput on average, over time, including overhead and possible burstiness of a transformation

Throughput

- Throughput affects *how you design* and *select components* for data transformations
- If your algorithm requires a particular data rate, then select downstream components based on that data rate
- **Reality check:** When buying IP, you may be forced into picking a faster/slower block

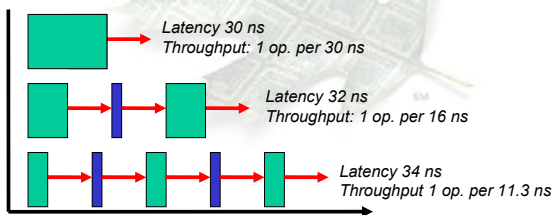
Latency

- Latency: Same as *response time*
 - How long it takes a transformation (or system) to produce an answer
 - Users care about response time!



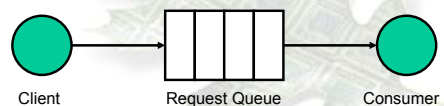
Latency

- Decomposing blocks into smaller blocks can *increase throughput* and *increase latency*
 - E.g., Pipelining: Overlap execution of blocks



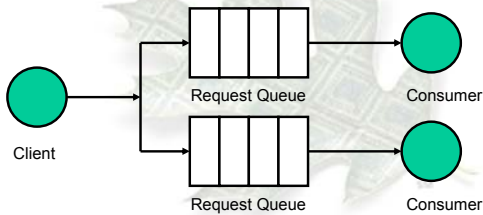
Throughput vs. Latency

Throughput - how many operations completed in a given time
Response time (latency) - how long it takes to complete a job



For best throughput: Keep consumer 100% busy; queue always has pending requests
For best response time: Keep queue empty so a request can be serviced immediately

Throughput vs. Latency



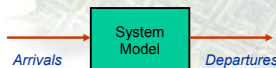
Doubling the consumers (servers): Increases throughput but not latency. Unless the workload is constant and time in queues is less due to more resources.

Performance Analysis

- How do we analytically make decisions about performance trade offs? (e.g., latency vs. response time)
- Formal approach: Queuing theory
 - Caveat: Computer systems in reality are usually too complex for simple queueing theory
 - Have to abstract and simplify the system
 - Still leads to useful understanding and a nudge in the right direction when making decisions
 - But we must ultimately measure - simulation

Queuing Theory Introduction

- Model data flows as a queuing system
- Each transformation is a “black box”



Steady state assumption: Number of tasks entering transformation is the same as number of tasks exiting the transformation (input rate = output rate)

Little's Law

- Relates
 - Average number of tasks in system
 - Average arrival rate of new tasks
 - Average time to perform a task

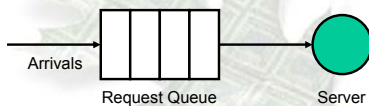
Mean number of tasks =

$$\text{Arrival rate} * \text{Mean response time}$$

For a system in equilibrium - no new requests created or destroyed by the transformation (black box)

Little's Law

- Model the transformation (black box) as a queue and a “server” component



- Define system in terms of
 - Time: how long in queue & server
 - Length: how many requests in queue & being served by server

Example of Little's Law

Average time to do a transformation is 50 ms (0.05 sec)
Transformation gets 200 requests per second (arrival rate)
What is the mean number of requests at the server?

A system in equilibrium implies that both the queue and the server are in equilibrium:

$$\text{Length}_{\text{queue}} = \text{Arrival rate} * \text{Time}_{\text{queue}}$$

$$\text{Length}_{\text{server}} = \text{Arrival rate} * \text{Time}_{\text{server}}$$

Hence, we want to find server length:

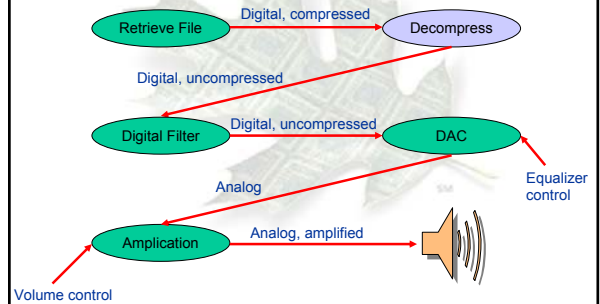
$$\text{Length}_{\text{server}} = 200/\text{sec} * 0.05 \text{ sec} = 10 \text{ requests on average}$$

Insight into how to size the queues

Algorithmic Trade offs

- For a given data transformation, investigate different possibilities for algorithm
- Algorithm can effect
 - Performance, power, area
 - Accuracy
 - Software vs. hardware realizability

Example: Encoding/Decoding Speech for VisorVoice



VisorVoice Speech Coding

Which is most appropriate for VisorVoice?

Trade off is speech quality vs. cost
 Costs: Processing power, memory, storage (for message), implementation complexity

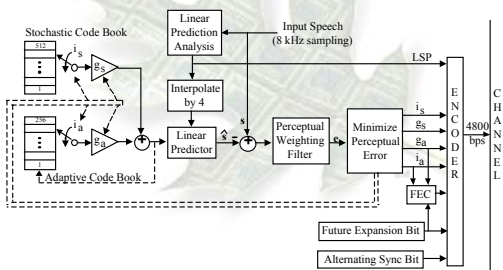
What does VisorVoice need?

Acceptable quality for low to moderately noisy environment
 Minimal cost (processor, power, memory) with rapid development and 1-2 hours

CELP

- **Code Excited Linear Predictive**
 - Breaks speech (bit stream) into frames that are processed as units
 - Based on Linear Predictive Coding: Uses past speech samples to compress speech
 - Code book search is bottleneck
 - Size of code book can be tailored to speech quality and computational needs
 - Good quality with medium bit rate (4800 bps)

CELP Encoder



MELP

- **Mixed-Excitation Linear Predictive**
 - Improves on CELP
 - Good for
 - Natural sounding speech
 - Difficult background noise environments (commercial and military communication)
 - Requires highly efficient implementation
 - Used by DoD, Voice over-Internet Protocol
 - Excellent for noisy backgrounds, very low bit rate (2400 bps)

ADPCM

- **Adaptive Differential Pulse Code Modulation**

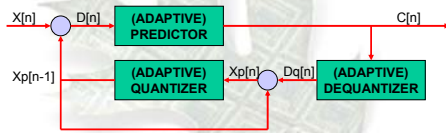
- Widely used wave form encoding (ITU standards G.721, G.723, G.726)
- Input is PCM word (at 8 kHz) and output is a 2, 3, 4, or 5-bit ADPCM word (at 8 kHz)
 - PCM code word is binary representation of signal
 - E.g., 16-bit PCM word to 4-bit ADPCM word

ADPCM

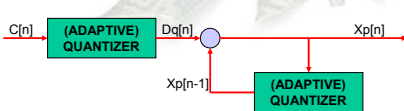
- Records differences between each sample and its predicted value (derived from previous sample or samples)
- Coding scale adjusted dynamically to handle varying changes in the size of the difference
- Good speech encoding but high bit rate

ADPCM

Decoder



Encoder



Speech Encoding Trade Offs

	MELP	CELP	ADPCM
Bit rate	2400	4800	16000
MIPS	20.43	17	7.5
ROM	128KB	128KB	3KB
RAM	98.2KB	14.8KB	1KB

Speech Encoding Trade Offs

	MELP	CELP	ADPCM
Bit rate	2400	4800	16000
MIPS	20.43	17	7.5
ROM	128KB	128KB	3KB
RAM	98.2KB	14.8KB	1KB

We picked ADPCM - minimal performance, leading to a simple solution (likely in software), relatively simple, and from a past project, we had software available for ADPCM

An Initial SW and HW Partition

- **Early design partition**
 - What is likely to be hardware or software
 - Affects trade-offs and costs
- **Lets you**
 - Determine early whether design will meet requirement goals
 - Begin planning for in-house development of some HW/SW IP blocks

Software vs. Hardware

- **Software**
 - Generally **flexible** - can be changed after design released
 - Increased **performance needed** - software is abstraction which adds overhead
 - Efficient use of HW (processor)
- **Hardware**
 - Generally **inflexible** - can't be changed
 - Achieve the **best performance and power** for a particular block

SW vs. HW Guidelines

- **In software -**
 - Complicated algorithms that may need to be fixed or upgraded in the field (e.g., early 56K modem standards were changed - **evolving standards!**)
 - Support multiple standards with same hardware capabilities (processor)
 - Less performance and power critical
 - User interface code (assuming an interface like that with VisorVoice)

SW vs. HW Guidelines

- **In hardware -**
 - Performance and power critical blocks
 - Real-world interfaces (e.g., A/D)
 - Single standard / single capability (typical)
 - Doesn't have to change (standard fixed)

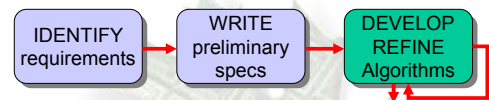
Software vs. Hardware

- Many **other issues** come into play:
 - What's available (as HW/SW)?
 - What does it cost?
 - How restrictive is its licensing?
 - What's the team's experience?
 - How much development time do we have?
- **Involve IP selection** - to come in a little while

Specification for Selecting IP

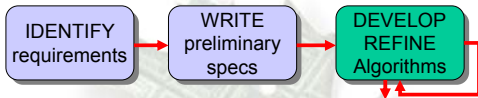
- **Annotated algorithmic specification**
- **Blocks with attributes**
 - Different algorithms and their trade offs
 - Performance (throughput, latency)
 - Area, power
 - Timing

Summary



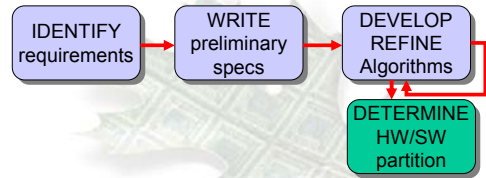
- Algorithmic modeling makes it possible to begin design verification early and start on an initial hardware/software partitioning and selection of IP.
- Algorithmic model is the gold standard against which other parts of the design will be written - worth the effort!

Summary



- At this point, we have the operational flows fully annotated with their requirements and we can begin selecting IP

What's Next



- From the annotated algorithmic model
 - Determine hardware and software blocks
 - Select processor and bus
 - Select IP blocks