CPU Modeling and Use for Embedded Systems

Lecturer: Trevor Meyerowitz
EE249 Embedded Systems Design
Professor: Alberto Sangiovanni-Vincentelli

October 21st, 2004
Outline

- Introduction
  - Motivation
  - Computer Architecture in 10 Minutes Flat

- Processor Modeling

- Use of Processor Modeling in Embedded Systems

- Conclusions
What is “Computer Architecture”?

- Coordination of many levels of abstraction
- Under a rapidly changing set of forces
- Design, Measurement, and Evaluation
The Instruction Set: a Critical Interface

software

instruction set

hardware
Levels of Representation (61C Review)

High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

Machine Interpretation

Control Signal Specification

Compiler

Assembly

Machine Interpretation

temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

lw $15,0($2)
lw $16,4($2)
sw $16,0($2)
sw $15,4($2)

0000 1001 1100 0110 1010 1111 0101 1000 1010 1111 0101 1000 0000 1001 1100 0110 1100 0110 1010 1111 0101 1000 0000 1001 0101 1000 0000 1001 1100 0110 1010 1111

ALUOP[0:3] <= InstReg[9:11] & MASK
Execution Cycle

- **Instruction Fetch**: Obtain instruction from program storage
- **Instruction Decode**: Determine required actions and instruction size
- **Operand Fetch**: Locate and obtain operand data
- **Execute**: Compute result value or status
- **Result Store**: Deposit results in storage for later use
- **Next Instruction**: Determine successor instruction
Fast, Pipelined Instruction Interpretation

Next Instruction

Instruction Address

Instruction Fetch

Instruction Register

Decode & Operand Fetch

Operand Registers

Execute

Result Registers

Store Results

Registers or Mem

Time

NI  NI  NI  NI
IF  IF  IF  IF  IF
D   D   D   D   D
E   E   E   E   E
W   W   W   W   W

1/22/02
5 Steps of MIPS Datapath

Figure 3.4, Page 134, CA:AQA 2e

- **Data stationary control**
  - local decode for each instruction phase / pipeline stage
Relationship of Caching and Pipelining
A Modern Memory Hierarchy

- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.
- Requires servicing faults on the processor

| Level           | Speed (ns) | Size (Bytes) | Hardware
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>10s</td>
<td>Ks</td>
<td>Processor</td>
</tr>
<tr>
<td>Control</td>
<td>1s</td>
<td></td>
<td>Control</td>
</tr>
<tr>
<td>Datapath</td>
<td>10s</td>
<td>Ms</td>
<td>Datapath</td>
</tr>
<tr>
<td>Registers</td>
<td>100s</td>
<td>Gs</td>
<td>Registers</td>
</tr>
<tr>
<td>On-Chip Cache</td>
<td>1s</td>
<td>Ts</td>
<td>On-Chip</td>
</tr>
<tr>
<td>Second Level</td>
<td>100s</td>
<td>Tertiary</td>
<td>Second</td>
</tr>
<tr>
<td>Cache (SRAM)</td>
<td></td>
<td>Storage</td>
<td>Level</td>
</tr>
<tr>
<td>Main Memory</td>
<td>100s</td>
<td>(10s ms)</td>
<td>Main</td>
</tr>
<tr>
<td>(DRAM)</td>
<td></td>
<td>Secondary</td>
<td>Memory</td>
</tr>
<tr>
<td>Secondary</td>
<td>1,000,000s</td>
<td>(10s sec)</td>
<td>Storage</td>
</tr>
<tr>
<td>Storage (Disk)</td>
<td></td>
<td>Tertiary</td>
<td>Storage</td>
</tr>
<tr>
<td>(Disk/Tape)</td>
<td>1,000,000,000s</td>
<td></td>
<td>(Disk/Tape)</td>
</tr>
</tbody>
</table>
The Other 90% of Architecture

◆ Longer Pipelines
  - The Prescott Pentium 4 CPU has a 31 stage pipeline

◆ Wider Pipelines and Speculation
  - Superscalar – Multi-issue
  - Speculate with branch prediction
  - Out of Order Execution

◆ Caches and Buffers
  - Up to 3 levels of caches + specialized caches
  - Memory Buffers and Reservation Stations

◆ Multiple Everything
  - Multithreading
  - Multiprocessor System On Chip
Outline

- Introduction
- Processor Modeling
  - SimpleScalar
  - Liberty Simulation Environment
  - Metropolis Processor Modeling
- Use of Processor Modeling in Embedded Systems
- Conclusions
SimpleScalar Overview

- The Standard for Microarchitectural Simulation
  - First Released in 1996
  - Developed by Todd Austin and Doug Burger
  - Multiple Levels of Models for Accuracy
  - Written in low-level high-performance sequential C-code

- Supports a Variety of Instruction Sets
  - Alpha, ARM, PowerPC, (x86)

- Supports a Variety of Microarchitectural Features
The Zen of Simulator Design

- *design goals* will drive which aspects are optimized
- the SimpleScalar Tool Set
  - optimizes performance and flexibility
  - in addition, provides portability and varied detail

Performance: speeds design cycle
Flexibility: maximizes design scope
Detail: minimizes risk

Taken From: [http://www.simplescalar.com](http://www.simplescalar.com)
Simulation Suite Overview

- **Sim-Fast**
  - 420 lines
  - functional
  - 4+ MIPS

- **Sim-Safe**
  - 350 lines
  - functional
  - w/ checks

- **Sim-Profile**
  - 900 lines
  - functional
  - lot of stats

- **Sim-Cache/Sim-Cheetah/Sim-BPred**
  - < 1000 lines
  - functional
  - cache stats
  - pred stats

- **Sim-Outorder**
  - 3900 lines
  - performance
  - OoO issue
  - branch pred.
  - mis-spec.
  - ALUs
  - cache
  - TLB
  - 200+ KIPS

Performance

Detail

SimpleScalar Tutorial

Taken From: [http://www.simplescalar.com](http://www.simplescalar.com)
Simulator S/W Architecture

- most of performance core is optional
- most projects will enhance on the “simulator core”
SIM-OUTORDER: H/W Architecture

- implemented in sim-outorder.c and components

*Note: The diagram shows the hardware architecture of SIM-OUTORDER, with stages such as Fetch, Dispatch, Register Scheduler, Memory Scheduler, Exec, Mem, Writeback, and Commit. The diagram also includes components like I-Cache (IL1), I-TLB, D-Cache (DL1), D-TLB, D-Cache (DL2), and Virtual Memory.*

*Source: SimpleScalar Tutorial*
Main Simulation Loop

```c
for (;;) {
    ruu_commit();
    ruu_writeback();
    lsq_refresh();
    ruu_issue();
    ruu_dispatch();
    ruu_fetch();
}
```

- main simulator loop is implemented in `sim_main()`
- walks pipeline from Commit to Fetch
  - backward pipeline traversal eliminates relaxation problems, e.g., provides correct inter-stage latch synchronization
- loop is exited via a `longjmp()` to `main()` when simulated program executes an `exit()` system call
Machine Definition File (ss.def)

- a single file describes all aspects of the architecture
  - used to generate decoders, dependency analyzers, functional components, disassemblers, appendices, etc.
  - e.g., machine definition + ~30 line main = functional sim
  - generates fast and reliable codes with minimum effort
- instruction definition example:

```
DEFINST(ADDI, 0x41, "addi", "t,s,i", IntALU, F_ICOMP | F_IMM, DGPR(RT), NA, DGPR(RS), NA, NA
        SET_GPR(RT, GPR(RS) + IMM))
```
SimpleScalar Conclusions

Solid Framework for Microarchitectural Research
- Used for ~33% of all Computer Architecture Papers
- Good for examining new microarchitectural features
- Fast and Reliable

But...
- Difficult to Retarget and Modify
- Monolithic – hard to use with other tools...
- Core Execution Semantics hard to modify
- Purely Sequential MOC.
**Liberty Simulation Environment**

◆ **A Next-Generation Microarchitectural Environment**
  - From Prof. David August’s Princeton Research Group
  - Compiler and Simulator Framework
  - Advanced Language Features

◆ **Structural Specification**
  - Extended Polymorphism

◆ **Custom Model of Computation**
  - Composability
  - Potential for Optimized Simulation
Liberty Simulation Environment

- Simulator construction system for high reuse

- Two-tiered specifications
  - Leaf module templates in C
  - Netlisting language for instantiation and customization

- Three-signal standard communications contract with overrides (control functions)

- Code is generated
LSS – A Natural Specification Language

- Modularize the model like HW
- Basic components
  - Concurrent computation
  - Communication through ports
- Called structural modeling
- Cornerstone of LSS design

- Compare to C/C++ approach
  - Function encapsulation ≠ hardware block encapsulation
  - Requires re-divide and re-conquer
  - Leads to [MICRO-35]
    - Inaccuracies
    - Long development times

Taken From: http://liberty.cs.princeton.edu
Models of Computation

- System C uses Discrete Event (DE)
- LSE uses Heterogenous Synchronous Reactive (HSR)
  - Edwards (1997)
  - Unparsed code blocks (black boxes)
  - Values begin *unresolved* and resolve monotonically
  - Chaotic scheduling

Taken From: http://liberty.cs.princeton.edu
Creating Static Schedules

- Edwards’ algorithm (1997)
  - Construct a signal dependency graph
  - Break into strongly-connected components (SCC). Schedule in topological order
  - Partition each SCC into a head and tail
  - Schedule tail recursively, then repeat head (any order) and tail’s schedule
  - Coalesce
# Reuse Penalty Revisited

<table>
<thead>
<tr>
<th>Model</th>
<th>Cycles/sec</th>
<th>Speedup</th>
<th>Build time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom SystemC</td>
<td>53722</td>
<td>-</td>
<td>49.1</td>
</tr>
<tr>
<td>Custom LSE</td>
<td>155111</td>
<td>2.88</td>
<td>15.4</td>
</tr>
<tr>
<td>Reusable LSE w/o optimization</td>
<td>40649</td>
<td>0.76</td>
<td>33.9</td>
</tr>
<tr>
<td>Reusable LSE with optimization</td>
<td>57046</td>
<td>1.06</td>
<td>34.4</td>
</tr>
</tbody>
</table>

- Reuse penalty mitigated in part

Reusable LSE model 6% faster than custom SystemC
Component Flexibility

Polymorphism

- Many state and routing components
  - Share identical functionality
  - Store different data types
- Polymorphism allows components to adapt
- But, polymorphism forces over 101 type instantiations for the Itanium 2

Taken From: http://liberty.cs.princeton.edu
Accelerate Modeling with Reuse

- Reuse components to amortize costs [Charest ’02, Emer ’02, Koegst ’98]
- 80% of I2 model’s 183 components from library of 22
  - Similar level of reuse for other non-Itanium processor models
  - Low-overhead customizability critical for this reuse [Radetzki ’98]

**The Liberty Research Group**

Taken From: [http://liberty.cs.princeton.edu](http://liberty.cs.princeton.edu)
Liberty Conclusions

◆ This improves upon SimpleScalar in that...
  - Modular and Structural
  - Domain Specific Language
  - Simulator and Compiler Generation

◆ But...
  - Large learning curve
    ♦ Arcane entry languages
    ♦ Complex communication protocol
  - Retargeting capabilities are unclear
  - Still a monolithic environment
Modeling Microprocessors in Metropolis

◆ Focus on Microarchitectural Design Space Exploration in the context of a System-Level Design framework
  - Intuitive MOC and Simplified Modeling Methodology
  - Connectivity to other tools
  - Retargetability

◆ Outline
  - Modeling using Kahn Process Networks
  - ARM Processor Modeling
  - Instruction Set Retargeting
**Modeling with YAPI + KPN**

- **Kahn Process Networks**
  - Processes communicating via unbounded FIFO’s
  - Blocking Reads / Unblocking Writes
  - Fully deterministic
  - No notion of time

- **YAPI**
  - Extension of KPN
  - Non-deterministic select
  - Refinement to bounded FIFO’s

- **Our Work**
  - Characteristics
    - Synchronous assumption
    - Keeps FIFO lengths fixed
    - Separation of function and timing
  - Microarchitectural Models
    - Single Process Model
    - Out of Order Execution Model
    - 2 Process ARM Models
      - XScale + Strongarm
    - Abstract Speculative OOE Model
**Single Process YAPI Model**

- Add hazard detection and bubble insertion (stalls)
- Parameterize the pipeline depth
- Add a branch predictor
  - Pass prediction and PC down pipeline (new channels)
  - Resolve branch when it commits

- **Single Process Execution Order**
  1. Read operands
  2. Execute
  3. Write to register file
- **Synchronous Assumption**

* Collaboration With: Sam Williams
Out-of-Order Architectures

- Tomasulo style register renaming
- Highly Parameterizable
  - #ports, # integer units, depth, etc.
- Broadcast nature handled with multiple copies of each channel
- ReadWriteIssue must maintain knowledge of which instructions can be issued to which functional units
- All execution units are derived from the Station class
  - N-way Super scalar processor was realized by changing the depth of the instruction channel (depth can be treated as width) from Fetch

* Collaboration With: Sam Williams
ARM Modeling Overview

- Separate between Microarchitectural Performance Model and Program Execution
- We Only Need to Model
  - Operand and Condition Code Dependencies
  - Branch Results
  - Execution Latencies
  - Forwarding Latencies
- Trace Contains
  - Every Instruction
    - Program Counter
    - Read + Write Operands (including cond. codes)
    - Instruction Type
  - (Optional) Data Addresses Accessed
- Advantages
  - Higher execution speed
  - Simplified, reusable microarchitectural modeling

Program Code
- Cross GCC
- ARM ISS
- Inst. Trace
- Microarch Model
- Exec Statistics

Performance Characterization
Double Process Model

- **Needed For:**
  - Modeling Forwarding
  - Modeling Variable Instruction Latencies

- **Leverages FIFO’s for modeling delays**
  - Preexecution Delay
    - Fetch, Decode, etc.
  - Execution Delay
    - Multiple Latencies, Forwarding
  - Synchronization
  - Stalls
    - Issue Stalls
      - Branch Misprediction
      - ICache Misses
    - Result Stalls
      - Operand Dependencies
Double Process

Needed For:
- Modeling Forwarding
- Modeling Variable Instruction Latencies

Leverages FIFO’s for modeling delays
- Preexecution Delay
  - Fetch, Decode, etc.
- Execution Delay
  - Supports Multiple Latencies, Forwarding
- Synchronization

```c
Preload_fifo();
While(true) {
  stall = stall_in.read();
  check_mispredict(stall);
  if (!stall) {
    inst = fetch(inst_num);
    if (inst.type == branch)
      branch_pred(inst);
    inst_out.write(inst);
  }
}
```

```c
Preload_each_results_fifo();
While(true) {
  read_results();
  if (stall == 0)
    ReadInst = FetchedInst.read();
  stall = check_stall();
  compute_memory();
  DoStall.write(stall);
  write_results();
  cycle_count++;
}
```
Models with Memory

◆ Features
  ♦ Cache Models
    ♦ Associative
    ♦ Perfect
    ♦ Statistical
  ♦ Translation Lookaside Buffers
  ♦ Data Cache Write Buffers
  ♦ Shared Bus between Caches

◆ Close to Simplescalar Models ARM
  ♦ 15% for XScale
  ♦ 25% for Strongarm

◆ Still Missing
  ♦ Instruction Buffer
ICache Usage

1. Instruction Fetch:
   get next instruction from trace.
   IssueStall = instruction.issue_stall

2. Instruction Cache Check:
   Query Instruction Cache
   IssueStall += iCache.read(PC)

3. Issue Stalling:
   Write Bubbles To Fetch Queue for
   Issue Stall Cycles, then write
   instruction to Fetch Queue
1. **Load/Store Instruction:**
   foreach (inst.data_address)
   dCache.checkHit(data_address);

2. **L/S Dispatch:**
   if (inCache(addresses))
   dispatch to hitQueue;
   else Dispatch to missQueue;

3. **L/S Commit:**
   Upon Completion:
   Update dCache state
An Abstract Speculative Model

Currently under development in collaboration with Haibo Zeng and Qi Zhu {zenghb, zhuqj}@eecs.berkeley.edu

Adding in Speculation + OOE as an afterthoughts can be difficult

- Why not begin with it and then constrain to a real implementation?

Assume Perfect Model (for a given fetch width)

- Branch prediction
- Perfect Memory and Register Files
- Unlimited Execution Resources and Forwarding

Analyze Performance for Different Applications

- Parallelism
- Resource Usage
- Etc.

Restrict to an actual implementation by adding constraints to the model.
ISA_ML Overview

◆ Main Parts

- A Visual Instruction Set Description Language
  - Currently one describes the encoding of instructions
  - Written using GME*, a UML-based environment for constructing domain specific modeling environments
- Generates a C++-based disassembler and trace-interface code for the given model ISA description

◆ Key Features:

- Two high level models
  - ISA State: Register Files, Memories, Program Counter, etc.
  - Instructions: Encoding and operand fields of each instruction
- Intuitive Visual Interface
- Leverages Hierarchy + Compact Representation
- Extensive Error Checking
- Easy to Retarget to Output Other Formats (e.g. verilog, nML, etc)

<table>
<thead>
<tr>
<th>Results</th>
<th>MIPS Integer Subset</th>
<th>PowerPC Integer Subset</th>
<th>ARM (approximate)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Instructions</td>
<td>10</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>Actual Instructions</td>
<td>55</td>
<td>80</td>
<td>26</td>
</tr>
<tr>
<td>Hours to Enter (approx.)</td>
<td>8</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Header File (# lines)</td>
<td>1357</td>
<td>2134</td>
<td>759</td>
</tr>
</tbody>
</table>

To Appear in 2004 OOPSLA Workshop on Domain Specific Modeling, October 24th, 2004
Title: A Visual Language for Describing Instruction Sets and Generating Decoders
Authors: T. Meyerowitz, J. Sprinkle, A. Sangiovanni-Vincentelli

*GME website: [http://www.isis.vanderbilt.edu/projects/gme/](http://www.isis.vanderbilt.edu/projects/gme/)
# ISA ML State Elements

<table>
<thead>
<tr>
<th>State Elements</th>
<th>Memory</th>
<th>RegFile</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordSize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address Bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program Counter</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Register</td>
<td></td>
<td>PC</td>
</tr>
</tbody>
</table>

## ISA ML Instruction Elements

### Bitfield Operands
- NumBits
- Encoding
- SingleEncoding

### Anchors
- AnchorPoint

### Connection
- Specifies the ordering of bitfields
  - Begin Anchor
  - End Anchor
  - Custom Anchor

### Ordering Connection
## Sample Instructions: Base Instruction

### Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>0...1</th>
<th>2...5</th>
<th>6...9</th>
<th>10..13</th>
<th>14..17</th>
<th>18..21</th>
<th>22..23</th>
<th>24..27</th>
<th>28..31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith Base</td>
<td>11</td>
<td>xxxx</td>
<td>Rm</td>
<td>Rn</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xx</td>
<td>xxxx</td>
<td>Rd</td>
</tr>
<tr>
<td>Add</td>
<td>11</td>
<td>0111</td>
<td>Rm</td>
<td>Rn</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xx</td>
<td>config</td>
<td>Rd</td>
</tr>
<tr>
<td>Subtract</td>
<td>11</td>
<td>0001</td>
<td>Rm</td>
<td>Rn</td>
<td>xxxx</td>
<td>xxxx</td>
<td>00</td>
<td>config</td>
<td>Rd</td>
</tr>
<tr>
<td>MAC</td>
<td>11</td>
<td>0011</td>
<td>Rm</td>
<td>Rn</td>
<td>Rmac</td>
<td>xxxx</td>
<td>xx</td>
<td>xxxx</td>
<td>Rd</td>
</tr>
</tbody>
</table>
Sample Instructions: Other Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>0...1</th>
<th>2...5</th>
<th>6...9</th>
<th>10..13</th>
<th>14..17</th>
<th>18..21</th>
<th>22..23</th>
<th>24..27</th>
<th>28..31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith Base</td>
<td>11</td>
<td>xxxx</td>
<td>Rm</td>
<td>Rn</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xx</td>
<td>xxxx</td>
<td>Rd</td>
</tr>
<tr>
<td>Addition</td>
<td>11</td>
<td>0111</td>
<td>Rm</td>
<td>Rn</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xx</td>
<td>config</td>
<td>Rd</td>
</tr>
<tr>
<td>Subtraction</td>
<td>11</td>
<td>0001</td>
<td>Rm</td>
<td>Rn</td>
<td>xxxx</td>
<td>xxxx</td>
<td>00</td>
<td>config</td>
<td>Rd</td>
</tr>
<tr>
<td>Multiply Accumulate</td>
<td>11</td>
<td>0011</td>
<td>Rm</td>
<td>Rn</td>
<td>Rmac</td>
<td>xxxx</td>
<td>xx</td>
<td>xxxx</td>
<td>Rd</td>
</tr>
</tbody>
</table>
Modeling Microprocessors in Metropolis: Conclusions

◆ **Improvements on Prior Approaches**
  - More Abstract
  - More Retargetable and Modular
  - Methodology for Refinement and DSE

◆ **But...**
  - Ongoing accuracy comparison with other tools
  - Performance needs to improve
  - Currently requires an external ISS to drive it
Outline

◆ Introduction

◆ Processor Modeling

◆ Use of Processor Modeling in Embedded Systems
  ♦ Different Levels of Modeling
  ♦ Co-Simulation
  ♦ Back-Annotation

◆ Conclusions
### Accuracy vs Performance vs Cost

<table>
<thead>
<tr>
<th></th>
<th>Accuracy</th>
<th>Speed</th>
<th>$$$*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Emulation</td>
<td>+++</td>
<td>+-</td>
<td>---</td>
</tr>
<tr>
<td>Cycle accurate model</td>
<td>++</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Cycle counting ISS</td>
<td>++</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Dynamic estimation</td>
<td>+</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Static spreadsheet</td>
<td>-</td>
<td>+++</td>
<td>+++</td>
</tr>
</tbody>
</table>

*$$*$ = NRE + per model + per design
Traditional Cosimulation

**Advantages**
- Allows prototyping without actual hardware
- Consistency between HW and SW models

**Disadvantages**
- Overhead for having 2+ simulators
- Often requires custom microprocessor models
- Doesn’t scale well for multiprocessor systems
Co-Simulation in Metropolis

Application

Mapping

Architecture

Application Process

Program Code, Sync Points, and Data Arguments

Performance Information

uArch Model

 Functional (ISS) Model

 Timing Model

Other Application Processes

Other Architecture Components
Backwards Annotation

Back Annotation Requirements

- User-specified level of granularity
- Flexibility for handling non-trivial interactions
  - RTOS’s, Interrupts, Pipelining, Intra-process variation
- Natural + Flexible Syntax
- Function with Metamodelfunction and Native Code

Our Proposed Solution

- Two functions to annotate the model code
  - CPU.BackAnnotate(begin_label, end_label, atomicity, (arguments))
  - CPU.BB_BackAnnotate(begin_label, end_label, atomicity, (arguments))
- Handle complicated features at the system-level
Back Annotation: Overall Picture

Application Process

Mapping Process

Annotated Mapping Process

Program Code, Sync Points, and Data Arguments

Sync

uArch Model

ISS Model

Timing Model
**Back Annotation: Example**

**Application Process**

```cpp
In_data = InPort.ReadInputs();
Out_data = do_processing(In_data);
OutPort.WriteOutPuts();
```

**Mapping Process**

```cpp
CPU.read(IN_DATA_SIZE);
CPU.execute(CPU.back_annotate(
do_processing.begin,
do_processing.end, true));
CPU.write(OUT_DATA_SIZE)
CPU.execute(EXEC_TIME);
```

**uArch Model and Back Annotator**
Final Words

- Software is a key component in Embedded Systems
  - Fast and Accurate Modeling is Key
  - Time isn’t the only factor to consider
    - Power, Memory Usage, Communication Usage, etc.

- Traditional Microarchitectural Environments are Unsuitable
  - Monolithic designs
  - Retargeting and integration issues

- We’re developing an integrated approach within Metropolis

- What we haven’t covered
  - Software Performance Estimation (Coming Soon...)
    - Estimate based on application, computation, and communication
  - Architecture Description Languages
  - Commercial Offerings
    - Mentor Graphics - Seamless
    - CoWare - ConvergenceSC + LISAtelk
    - VaST Systems
    - Et al.