Notice!

- These slides were presented at a MICRO-34 tutorial given on December 1, 2001.
- These slides are necessarily incomplete. Missing items include:
  - Numerous live software demonstrations
  - Slide animation
  - Audience interaction
  - Other verbal communication
- For the latest, please see:

http://liberty.princeton.edu/

Architectural Exploration with Liberty
MICRO-34 Tutorial

The Liberty Computer Architecture Research Group
(http://liberty.princeton.edu/)

Manish Vachharajani, Jason Blome, Neil Vachharajani,
David Penry, Ram Rangan, Sudhakar Govindavajhala,
Jon Wu, Spyridon Triantafyllis, Xinping Zhu

Prof. David August
Princeton University
Princeton, New Jersey

What is Liberty?

Liberty is a Software Project

- The Liberty Tools integrate a collection of generalized components
  - Liberty Simulation Environment – an automated simulator constructor
  - Liberty Compilation Environment – an automatically retargeted optimizing compiler
  - GLAD – a unified, precise (micro)architectural description
- Liberty software is non-copylefted free software

Liberty is a Research Group

- Graduate/Undergraduate, Princeton/Rutgers, EE/CS
- Strong Collaboration with MESCAL (http://www.gigascale.org)

Today's Focus:
The Liberty Simulation Environment (LSE)

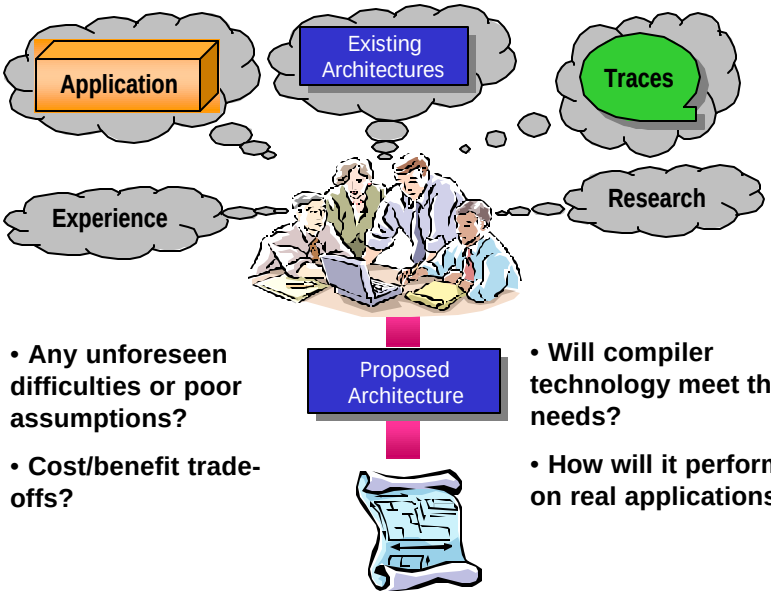
- Excellent, well-written, and popular simulators exist
  - SimpleScalar, RSim, SimOS, etc.
- LSE is NOT a simulator
- LSE is a simulator construction framework
- LSE manages simulator components
  - Rapid Systematic Design Space Exploration
  - Open, Standardized Collaborative Framework
- LSE is about one year old
  1. Class Project: a simple simulator with tight compiler interaction
  2. Target machine disputed
  3. So, we started to generalize…
  4. Liberty Simulator Builder was born

Tutorial Sequence

8:30	Welcome	
8:35	Liberty Introduction and Motivation	August
9:00	The Simulation Environment	M. Vachharajani
	Fundamentals and The Visualizer	Blome
10:00	The Pipeline Builder	N. Vachharajani
10:30	Refreshment Break (10-30 Minutes)	
11:00	Machine Configuration Issues	Penry
	Configuration of "sim-outorder"	
	Complex Machine Configuration	
	BLISS – Both LIberty and SimpleScalar	
12:00	IA-64 Emulation/Simulation	Rangan
	Alpha Emulation/Simulation	Govindavajhala
12:15	Release Info and The Future of Liberty	August
12:30	Adjourn	

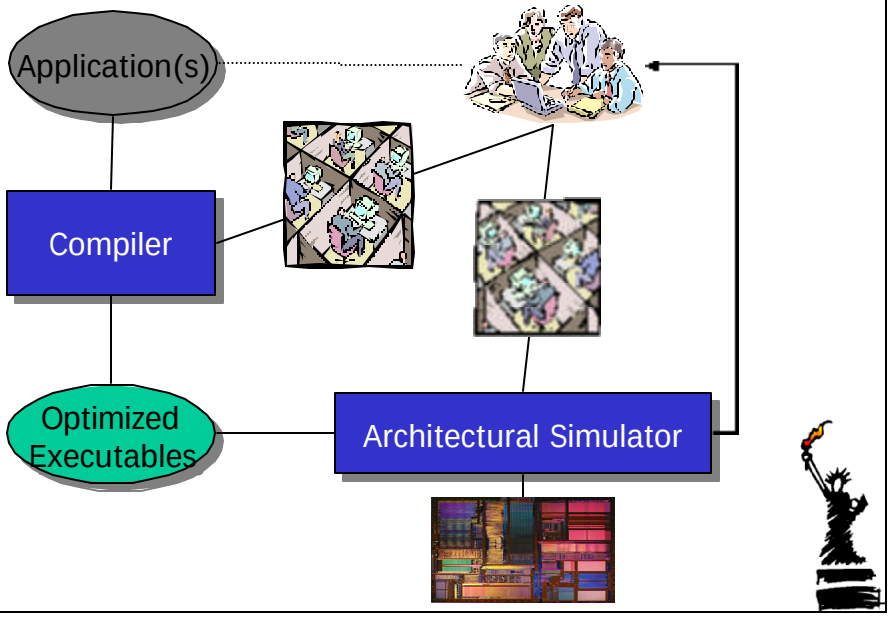
The First Motivator

The Typical Design Process

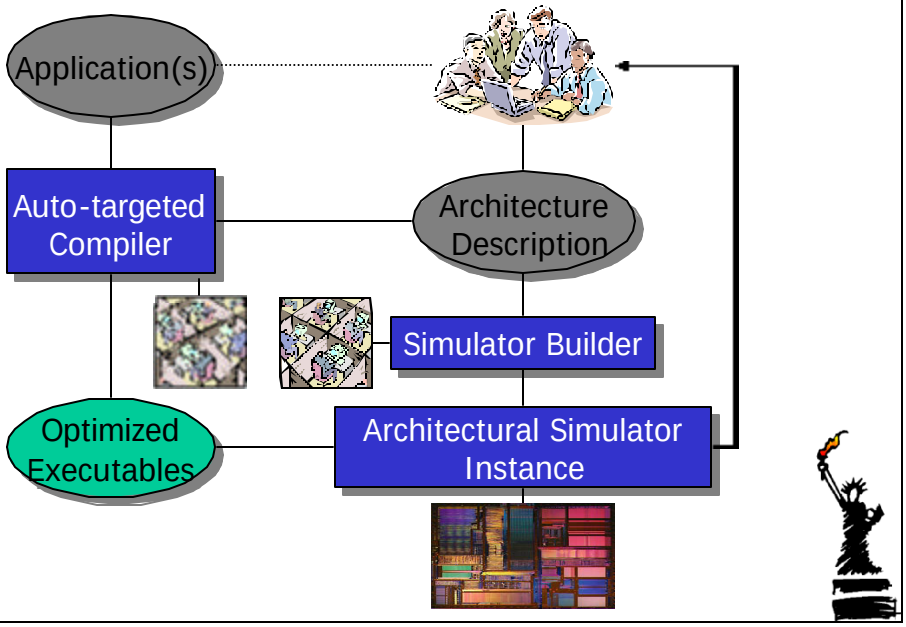


- **Any unforeseen difficulties or poor assumptions?**
- **Cost/benefit trade-offs?**
- **Will compiler technology meet the needs?**
- **How will it perform on real applications?**

The First Motivator
The Typical Design Process



Architectural Exploration with Liberty
Iterate in hours…



The First Goal of Liberty

1. Automatic customization and synthesis of a comprehensive set of tools

- Move people off the critical path
  - Facilitate rapid evolution of designs
  - Tool work is reusable and design independent
- Synchronize simulator, compiler, and associated tools
  - Focus efforts on overall system design quality
  - Ensure tools match
- Complexity exposed only as necessary

The Second Motivator
Collaboration Barriers

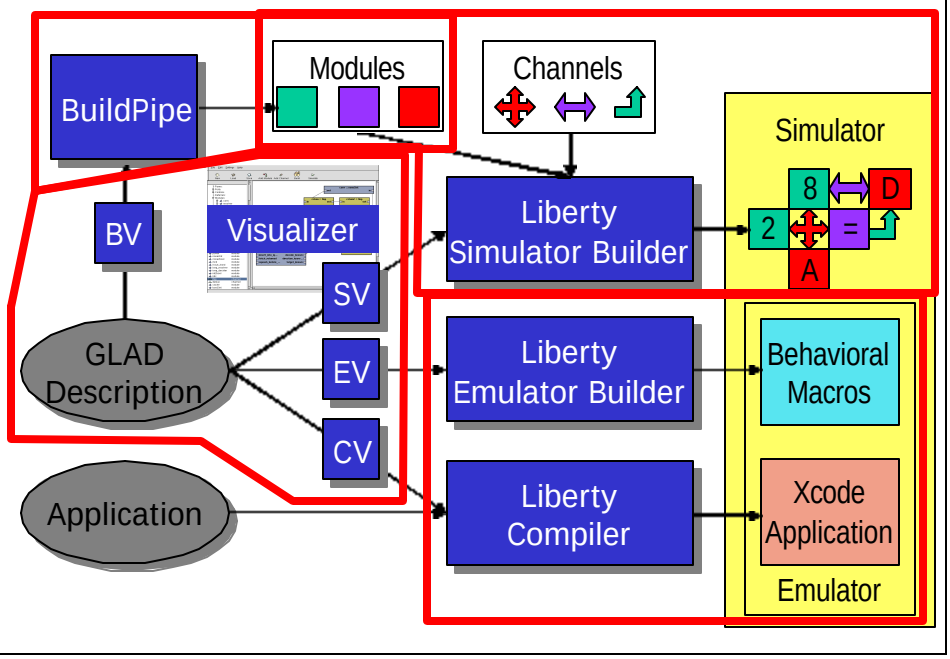
- Alice, Professor of Electrical Engineering
  - Invents a new microarchitectural component
  - Writes a simulator to evaluate it
  - Submits the excellent results for publication
- Bob, an Anonymous Reviewer
  - Likes the logic behind the idea
  - Takes on faith the validity of the simulator
  - Recommends the paper for publication
- Carol, Senior Architect at a Processor Design Company
  - Considers the invention for their next product
  - Cannot justify its use due to evaluation cost in product context
- Doug, Professor of Computer Science
  - Seeks collaboration with Alice for interaction studies
  - Collaboration stalls due to tool incompatibility

The Second Goal of Liberty

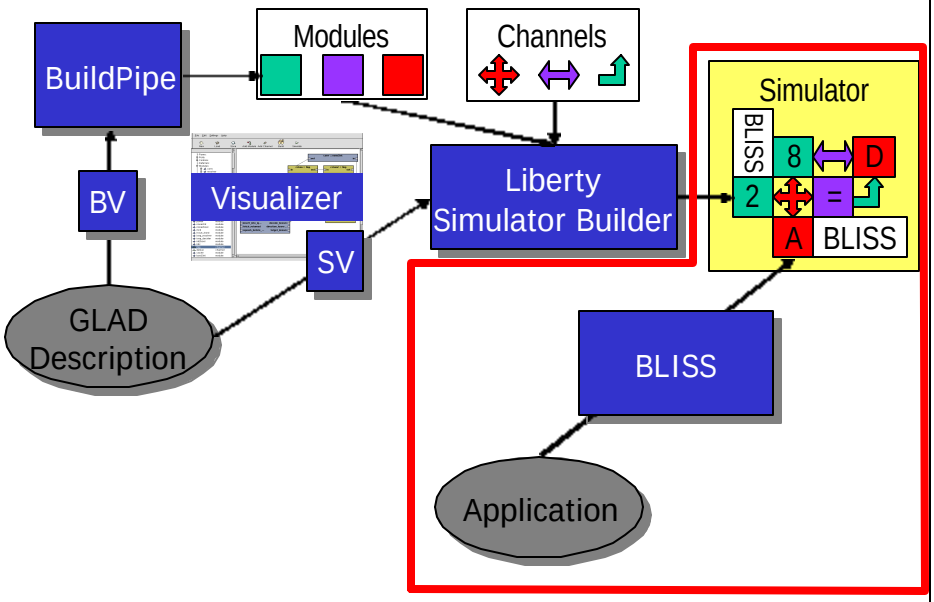
2. Provide a standardized framework for the exchange of architectural components, algorithms, and designs

- Encapsulate components/algorithms
  - Object-orientation
  - No context assumptions
- Evaluation of ideas in a variety of different contexts
  - Bob: meaningful peer review
  - Carol: product inclusion
  - Doug: research collaboration
- Must be open, flexible, and efficient

A Quick Tour of Liberty



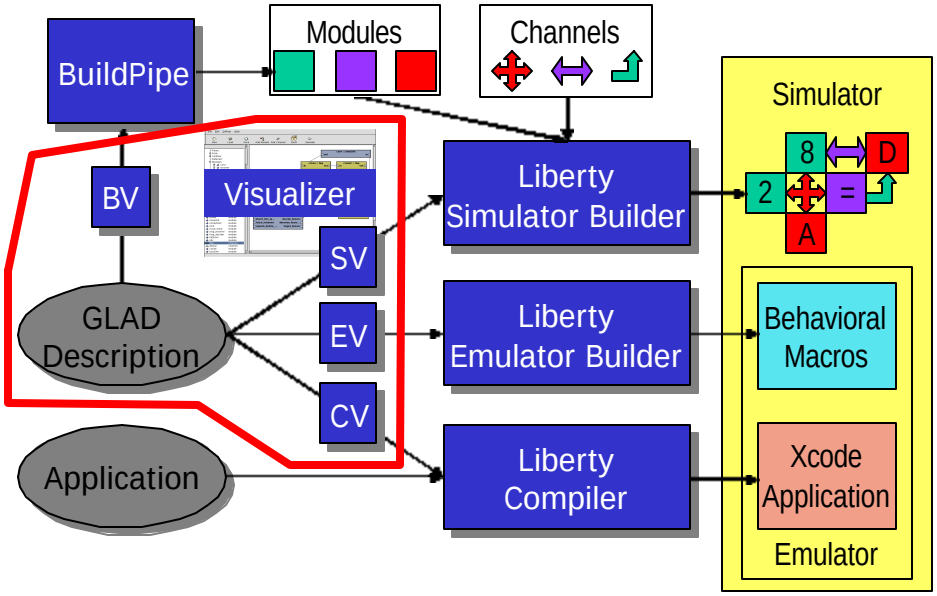
A Tour of Liberty



Terminology Review

Emulator	compiled-code functional simulator: Run alone for validation/statistics, or with model for simulation.
Interpreter	functional simulator, generally slower than emulation
Simulator	A model with a notion of time, a product of the LSE
Simulator Builder	A collection of tools that build simulators, here LSE
BuildSim	The Simulator Builder front-end
BuildPipe	The Pipeline Module Builder
Visualizer	Machine configuration graphical user interface

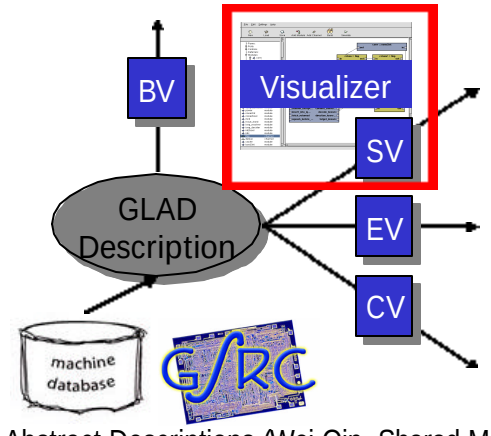
A Quick Tour of Liberty



GLAD

GLAD is Liberty's Architectural Description

- Single description for consistency, ease of use
- However, tools have different views of the architecture
- View Generators create views from GLAD



Abstract Descriptions (Wei Qin, Sharad Malik)

Document View

- Emulation
- Code generator
- Instruction Selection
- Register classes

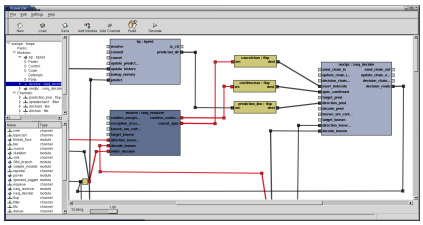
Semantics View

```
ADD:
Resource     0  1  2  3
RegFile      X        X
ALU             X  X
```

Scheduling View

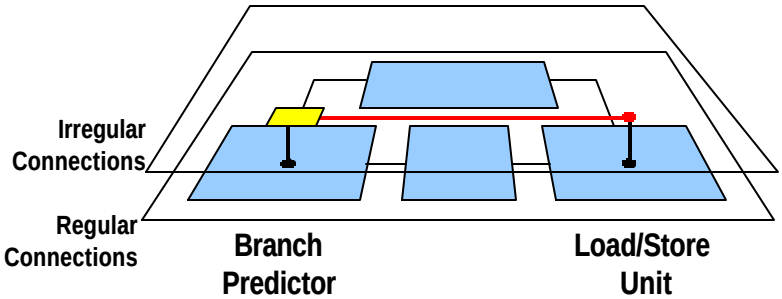
Liberty Simulator View

- Describes instantiation, customization, and interaction of architectural primitives
- Instantiation
  - Cache architectural primitive
  - L1 and L2 cache instantiations
- Customization
  - Parameters (L1 is 4-way)
  - Code points…
- Interaction
  - Port-to-Port network (L1 is connected to L2)
  - Code points…
- Simulator view can be visualized

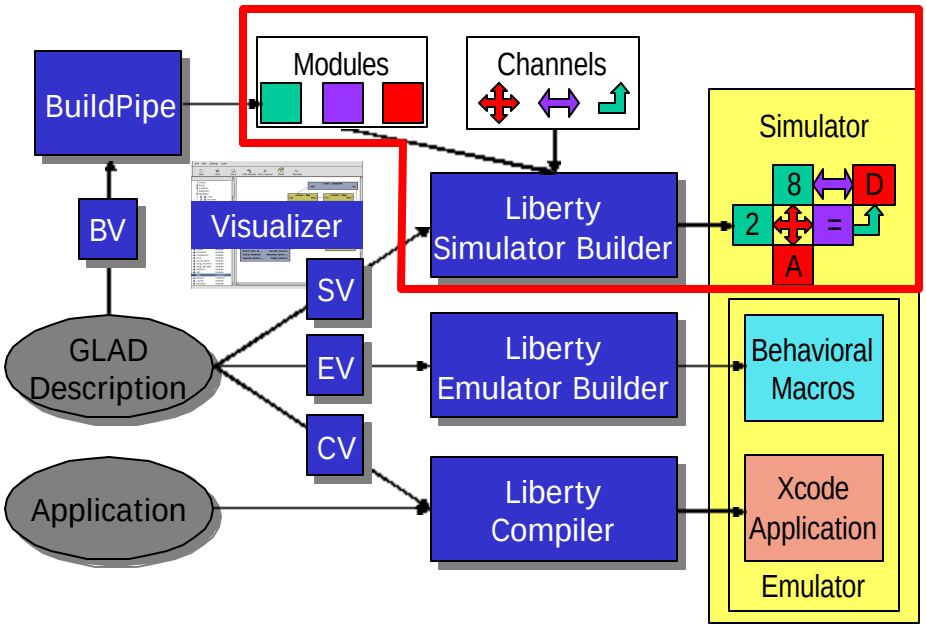


Liberty Simulator View
Code Points

- Parameters and port-to-port networks not sufficient
  - Architecture peculiarities, VLSI design constraints
  - Unanticipated options
- Example: Branch predictor stalls when 4+ loads are outstanding
- Primitive behavior can be modified with code points/functions
- Primitives export state through query functions

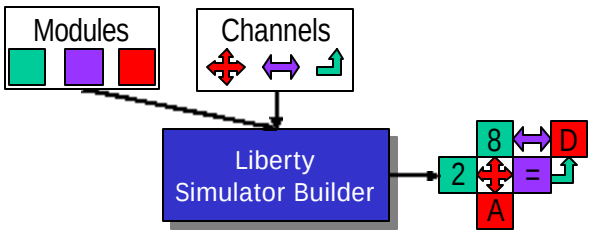


A Quick Tour of Liberty

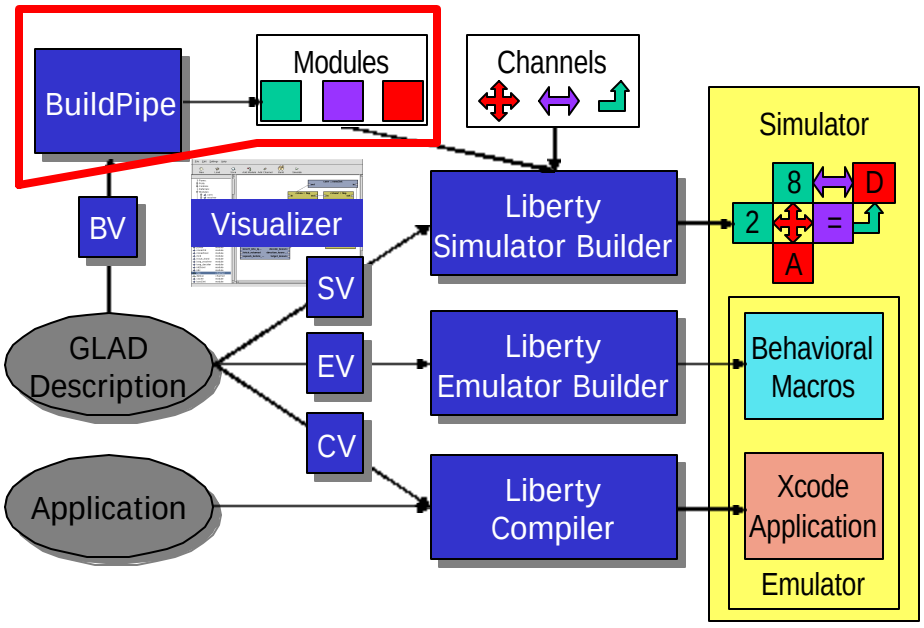


Liberty Simulator Builder

1. Read Description
2. Instantiate primitives prior to compile time
   - Specialize and connect instantiations for given context
   - Ex: size branch predictor, connect to fetch unit
3. Compile and Link
   - Specialization allows optimizations to kick in…
   - Ex: size of branch predictor is constant -> constant prop.



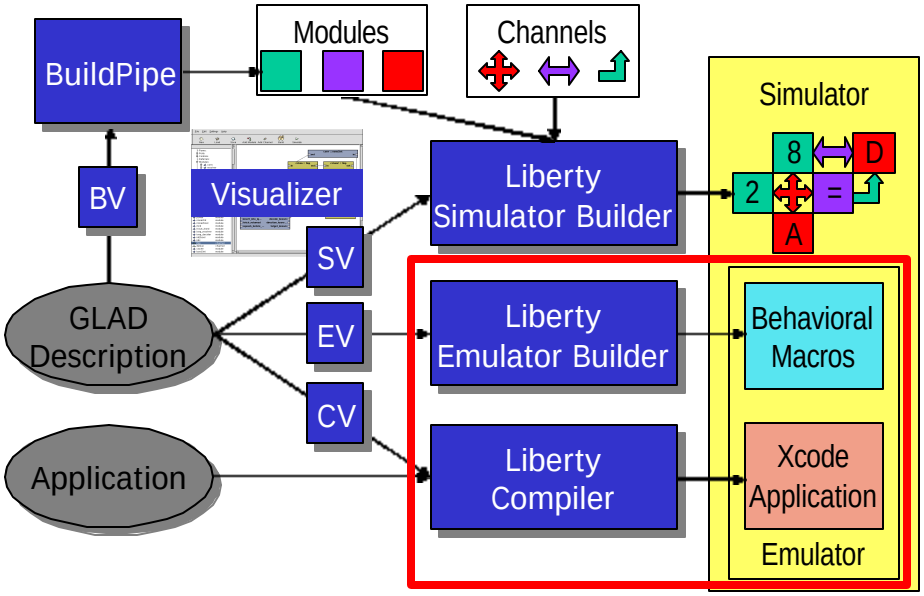
A Quick Tour of Liberty



Liberty BuildPipe
The Not-So-Special Case

- BuildPipe can create pipeline modules
- Pipelines can also be built of standard components
  - Too general
  - Not analyzable
- Pipelines are extremely regular structures
  - Easier to specify regular structure using a specialized language
  - Analysis leads to simulation speed improvements
- Pipeline description can be used for other purposes
  - Scheduling in the compiler
  - Formal verification of pipeline design

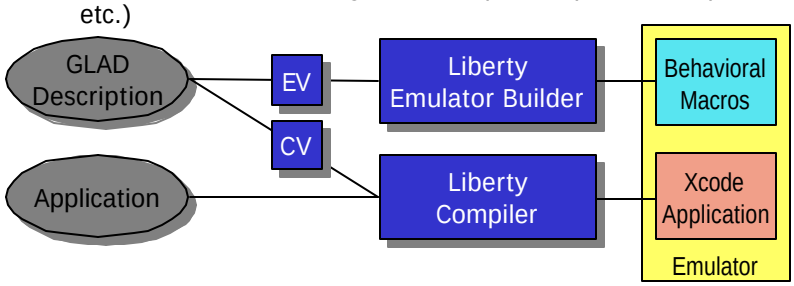
A Quick Tour of Liberty



Liberty Simulation Environment

Enhanced Compiled-Code Emulation

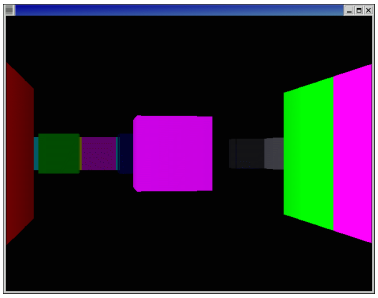
- Code execution determined by program state and simulated machine state
- Improved modeling
  - Wrong-path Speculative Execution
  - Performance monitoring counters (cache, power, temperature, etc.)

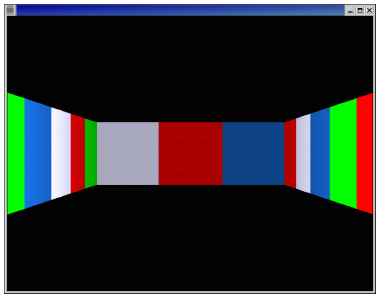


- Can approach native code speed (*static binary translation*)

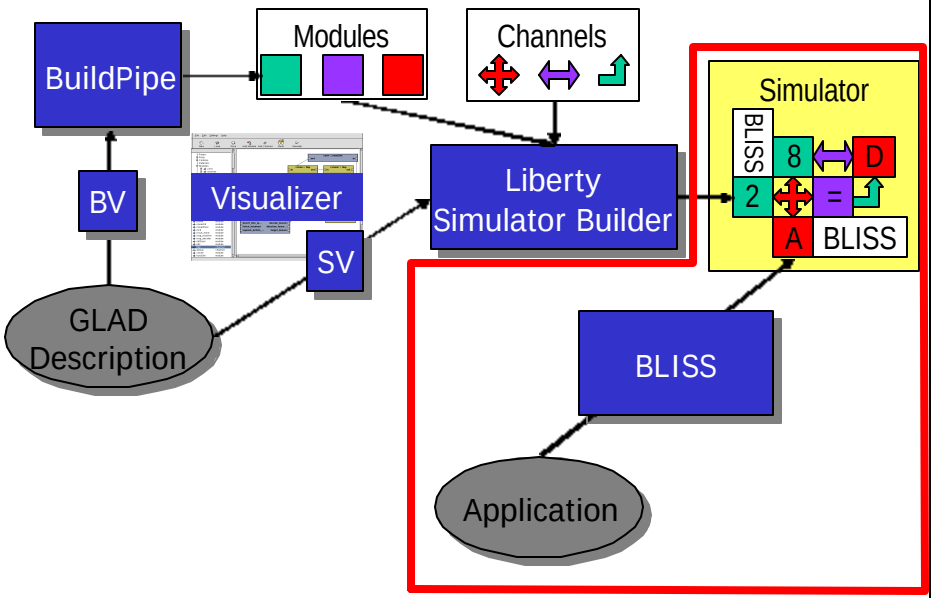
Typical Performance Results

- Various speed/detail trade-offs:
  - Instruction Emulation (verification of compiler, specification)
    - 1-8x slowdown from native
  - Profiling/Statistics (CFG/Cache/Branch)
    - 2-40x slowdown from native
  - Full simulation (Cycle-accurate tuning of architecture)
    - 100x – 4000x slowdown from native
  - IMPACT Lcode Emulation is Faster Than Native!!!





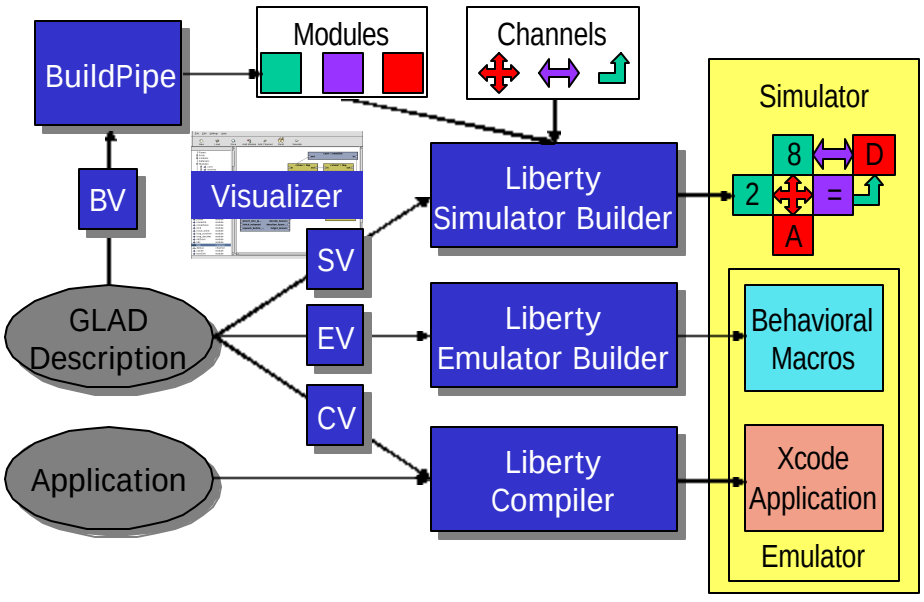
A Tour of Liberty



BLISS
Both Liberty and SimpleScalar

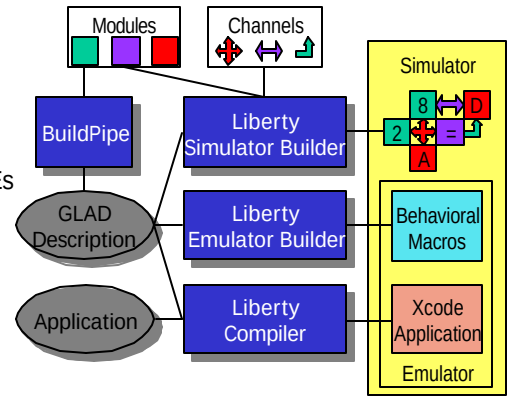
- BLISS optionally integrates Liberty with SimpleScalar
- SimpleScalar front-end provides code interpretation
  - Code interpretation front-ends supported in Liberty
  - Currently no Liberty Interpreter Builder
- SimpleScalar architectural components provide functions
  - Integrated through code points for functionality
  - Wrapped in Liberty Skeleton modules for timing interactions
- Experience with SimpleScalar
  - SimpleScalar code was extremely easy to use, well-written
  - BLISS was extremely easy to create
  - We suspect that any well-written simulator can be Liberated!

A Quick Tour of Liberty



Liberty Simulation Environment Properties

- Diversity
  - General Purpose
    - IA-64, Alpha…
  - Heterogeneous multi-PEs
    - IXP-1200, PowerNP…
- Precision vs. Speed
  - Functional
  - Statistics/Profiling
  - Cycle Accurate
- Built simulators competitive with hand-coded ones
- Take advantage of all available resources
  - Good host compilers
  - Host resources (multiple processors)



Liberty Simulation Environment Properties

Exposes complexity only as necessary

- Statistics gatherer (compiler writer, summer intern)
  - Write statistics queries and run the simulator
- ISA developer (architect)
  - Describe an ISA
- Microarchitectural explorer (microarchitect, Carol)
  - Configure; specify interlocks
- Architectural primitive developer (microarchitect, Alice)
  - Develop and modify modules
- Simulation environment maintainer (Liberty project member)
  - Needs to understand the build process magic

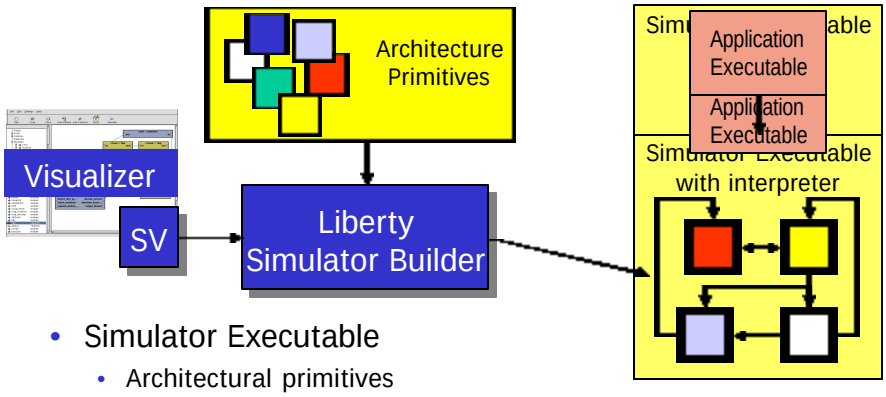
How to evaluate LSE

- LSE is NOT a simulator
  - Comparisons to simulators cannot be made!
    - BLISS
    - Evaluating module code misses the point
  - Compare to other simulator constructors
- LSE is about one year old
  - 90%: exploring generalization mechanisms
  - 10%: writing architectural components
- Like all useful software, LSE is a work in progress
  - Expect additional generalizations – it's in our blood
    - Current set is good, but we are not satisfied
    - More generalizations brewing
  - Expect rapid adoption of good ideas and suggestions
- Live Demo Caveat...

Tutorial Sequence

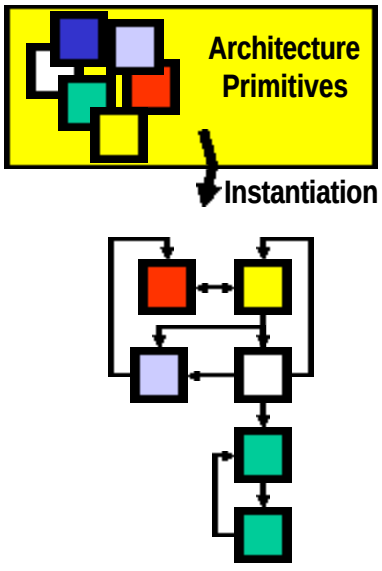
8:30	Welcome	
8:35	Liberty Introduction and Motivation	August
9:00	The Simulation Environment	M. Vachharajani
	Fundamentals and The Visualizer	Blome
10:00	The Pipeline Builder	N. Vachharajani
10:30	Refreshment Break (10-30 Minutes)	
11:00	Machine Configuration Issues	Penry
	Configuration of "sim-outorder"	
	Complex Machine Configuration	
	BLISS – Both LIberty and SimpleScalar	
12:00	IA-64 Emulation/Simulation	Rangan
	Alpha Emulation/Simulation	Govindavajhala
12:15	Release Info and The Future of Liberty	August
12:30	Adjourn	

LSE Is a Simulator Builder!



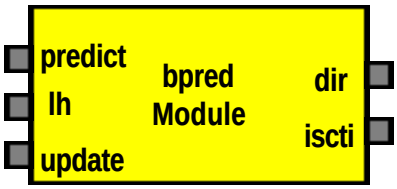
- Simulator Executable
  - Architectural primitives
  - Architecture description
  - ISA Behavioral Macros
- Simulator Executable is specialized for a particular target
- In this part, we'll construct a simple machine...

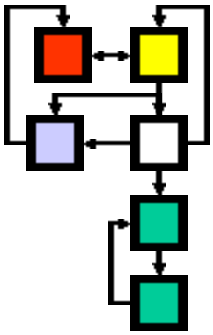
What's Inside the Architecture Description?



- Instantiated Primitives
  - Customization of instances
  - Interconnections between instances
  - User defined control
    - If needed
- First, the primitives...

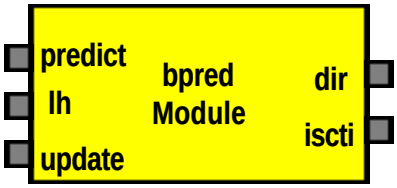
Architecture Primitives





- Large primitives are called **modules**
- Modules roughly correspond to hardware blocks
- We provide usable modules but users can add their own
- Modules are sequential code executing in a concurrent environment
- Modules are flexible
  - via parameters
  - via code points

Modules Parameters

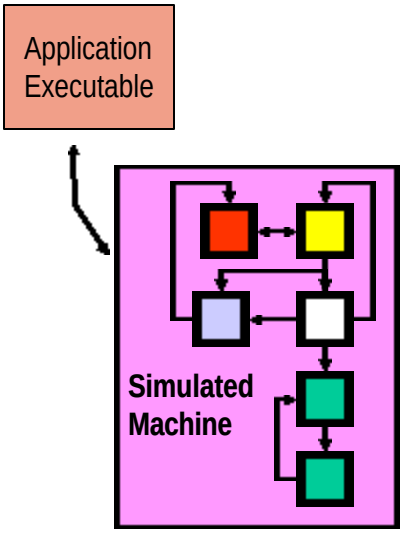


Parameters

history_tag_bits	12
history_bits	8
history_table_size	32
predictor_table_size	1

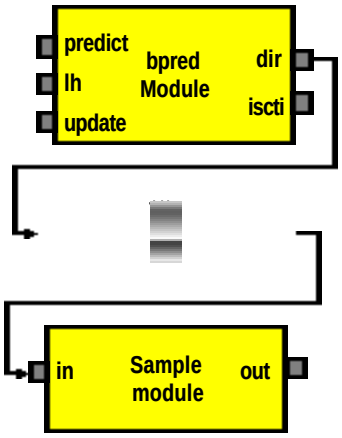
- Module "sizing" and simple configuration options are set through parameters
- We'll configure our branch predictor as shown to the left
- Now, how do we get instructions into the branch predictor?

Modules
Communication with the Application



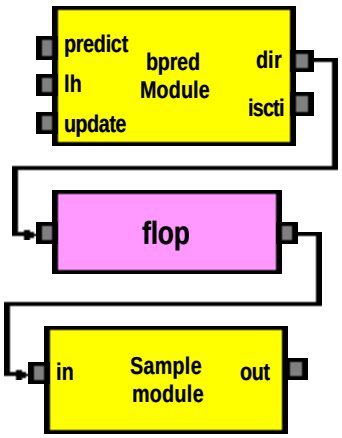
- Certain modules are responsible for communicating with the application
- In most cases this module is the next PC logic of the machine
- Here, this is the iseq_decider unit
- How does the decider communicate with the predictor?

Module Communication
Ports



- Modules communicate to connected modules via message passing
- Module send and receive messages on their ports
- Ports are directional
  - Input
  - Output
- Module ports do not determine communication semantics
  - So they can't be directly connected

Module Communication Channels



- Module ports are connected via **channels**
- Channels embody the semantics of the connection (more later)
- Let's connect the decider and the predictor

Linking the Decider and Branch Predictor

- Every time the decider decides a message is sent on the decision_made port
  - Connect this port to the predict port of the predictor
- Every time a prediction is made a message is sent on predicted_dir
  - Connect this port to the direction_pred port on the decider
- How does the branch predictor know about the instruction?

Linking the Decider and Branch Predictor

- Every message has the a dynamic id corresponding to the dynamic instruction that caused the message
- Modules may also specify additional data for their messages
  - to override dynamic id values
  - to send/receive additional data
- We now have a configuration that can speculate
  - Almost, the decider is too smart to speculate!

Inferred Behavior from Interconnections

- iseq_decider infers that we don't want speculation
  - It cannot resolve speculation since the confirm_spec port is unconnected
- Modules infer their semantics from the interconnections
  - Decider only speculates if the confirm_spec port is connected
  - Decider uses static prediction if the predict ports are unconnected
  - bpred only stores BTB information if the lookup_history port is connected
- Modules optimize code from these inferences
  - Modules can optimize their behavior based on this static knowledge
  - bpred saves memory
  - decider shorts circuits book keeping in simple configurations
- Let's connect the decider's confirm_spec port

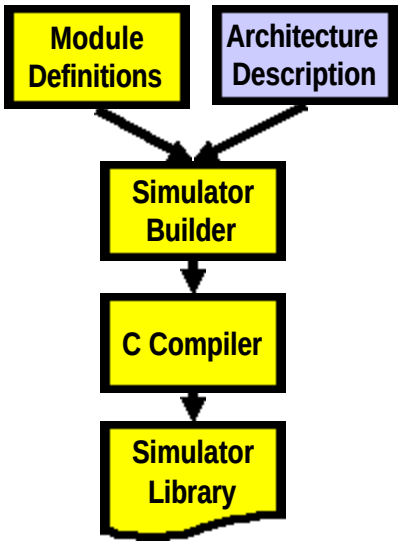
Speculation Resolution

- We need a unit to determine when speculation ends
  - With the iseq_decider, there is a resolver module that can determine this information
- We'll connect the decider and resolver together according to the manual
- But, we need a pipeline to feed resolution information to the resolver and decider!
- For now we'll simulate it using flops and fill in the details later

Simple Branch Predicting Machine

- Now we'll build a simulator corresponding to the machine we just configured
- Overview of the process
  - Simulator Library
  - Application Interaction
  - Preparing the application
  - Building the simulator library
  - Linking the two together

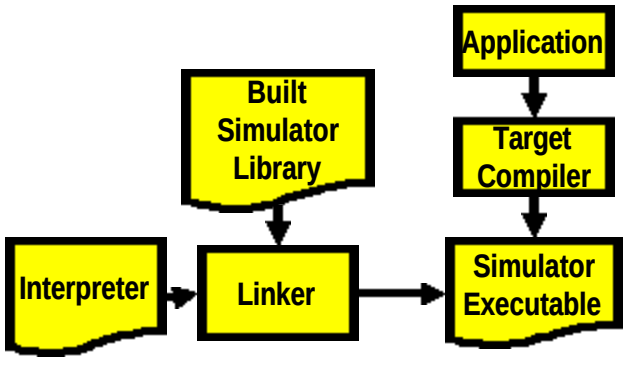
LSE Simulator Library



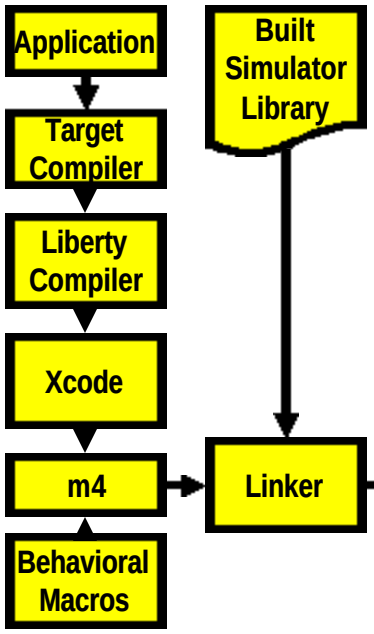
- Simulator Library
  - Architectural primitives
  - Architecture description
  - ISA Behavioral Macros
- Simulator Library is specialized for particular Architecture Description
- Next the application

Running an Application

- We can link an interpreter to the simulation library
  - Flexible and can support self modifying code simulation
  - But, is relatively slow
- What about compiled code simulation?

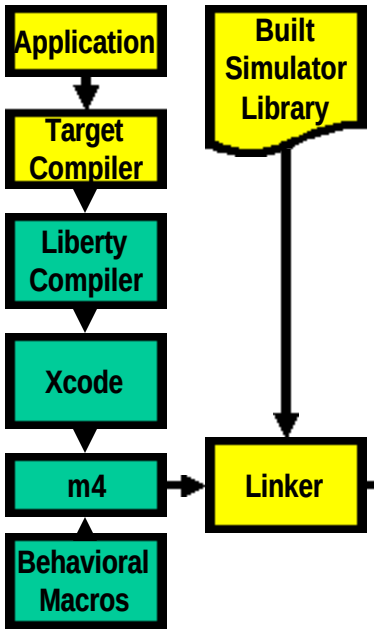


Compiled Code Application Interaction



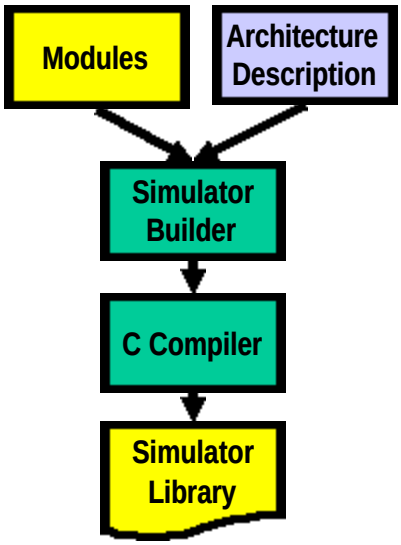
- Application can be linked directly for compiled code simulation
- For this demo, I'll use compiled code Lcode from IMPACT
- Lets build a simulator

Building the Application



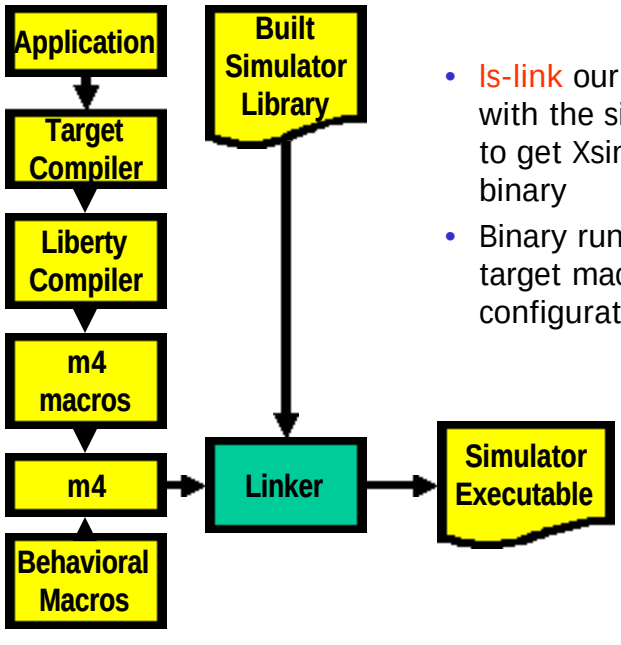
- We use an internal IMPACT so we provide the Lcode for a set of benchmarks
- IA64 is also supported
- Use I-bench to prepare benchmarks for compiled code simulation

Building the Simulator Library



- We'll build our simple simulator with next PC logic and a branch predictor with a dummy pipeline(bpred_test.xml)
- We use ls-build to execute all the stages of simulator library construction

Linking the Simulator



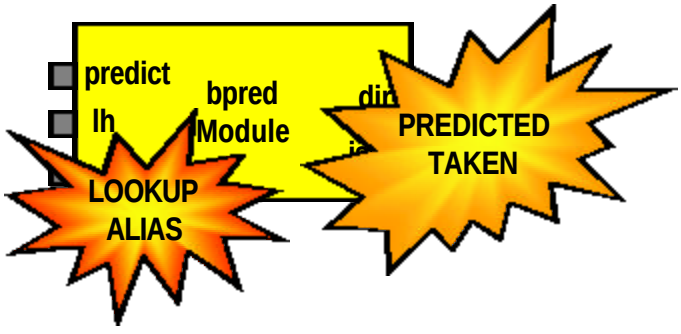
- ls-link our wc benchmark with the simulator library to get Xsim, our simulator binary
- Binary runs wc on the target machine configuration

Running the simulator

- Xsim
  - A binary that behaves like wc
  - But, is actually simulating wc on our configured machine
- Output
  - Xsim outputs the total cycle count for the simulation
- But what about more interesting statistics?

Data Collection
Events and Statistics

- Each module emits **events** during execution
  - Events contain data corresponding to the state of the module when the event occurred
- We can define **statistics** that catch these events and compute the results in which we are interested
- We'll compute a mispredict rate and a history table alias rate



Branch Predictor Events

- HISTORY_ALIAS defines the following information per event
  - id – the name of the input instruction that caused this alias
  - addr – the address we were looking up in the history table
  - haddr – the address we aliased with in the history table
  - alias_oper – the operation type that caused the alias
    - 0 – predict, 1 – update history, 2 – update predictor, 3 – lookup history
- PREDICTION defines the following information per event
  - id – the name of the input instruction that caused this prediction
  - dir – the direction we predicted, 0 – not taken, 1 – taken

Statistics

- There can be a statistic for each event on each instance in the configuration
- Declarations Section
  - Defines variables and includes header files needed to compute this statistic
- Initialization Section
  - Initializes variables declared in the previous section.
  - Receives the data associated with an event
- Record Section
  - Executed each time the corresponding event is generated
- Report Section
  - Executed at the end of simulation to report collected statistics

Computing the Misprediction Rate

```
<STAT inst="mainpe__bp" event="PREDICTION">
   <DECL>
     #include <stdio.h>
     #include "SIM_dynid.h"
     int mispredictions, predictions;
   </DECL>
   <INIT>
     predictions=0;  mispredictions=0;
   </INIT>
   <RECORD>
     if(dir != SIM_dynid_get_taken(id)) mispredictions++;
     predictions++;
   </RECORD>
   <REPORT>
     fprintf(stderr,"Total predictions=%d, mispredictions=%d\n",
                    predictions,mispredictions);
     fprintf(stderr,"Mispredcition rate=%g%%\n",
       predictions?((double)mispredictions)/predictions*100:0);
   </REPORT>
 </STAT>
```

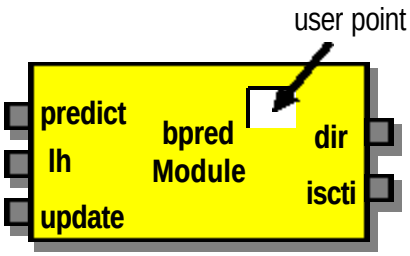
Event and Statistic Bonuses

- Configurer can collect new statistics without muddling around in module code
  - Allows modular collection of data
- Statistics can be used for other things besides real statistics
  - Debugging new modules
  - Trace generation
  - Profiling for Compilation
  - Visualization
- Unused events have no cost
  - Can be used liberally to allow maximum reporting of information

Remaining Issues

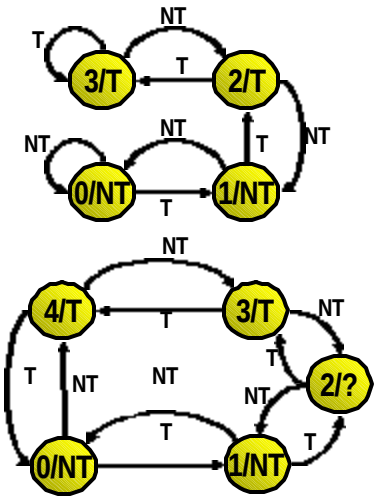
- More complex module configuration
  - Examples: predictor type, saturation, round robin
  - Handled through user points
- Overriding default control
  - Example: only predict every other cycle
  - Handled through control points
- How does the simulator know how to handle different types of instructions?
  - Example: the predictor should only predict branches
  - Handled through decode points

User Point Example
Predictor Type



- Some modules may have complex user definable behavior
- Specified through **user points** and **user functions**
- User points are holes in the module code
- Let's write a user function to change the state machine our branch predictor uses to predict branches

Hypothetical Example



- Suppose we have developed a statistical model for the branch direction trace of our program
- We determine an optimal threshold detector
  - 5 values (states)
  - 0,1 – predict not taken
  - 3,4 – predict taken
  - 2 – randomly select the direction
- Lets see how to use user functions to implement this detector as our predictor state machine

The Branch Predictor User Points

- The bpred module defines the following user points for prediction
  - int predictor_init(void) – called to initialize a predictor
    - Return value is the initial predictor state
  - int predictor_next_state(int state, int dir) – called to update the state of a predictor
    - Return value is the next predictor state
  - int predictor_output(int state) – called to make a prediction based on the state of a predictor
    - Return value is the predicted direction
- We'll set these with our own code

User Function Code

```
<USERFUNC name="predictor_init">
   return 3;
</USERFUNC>
<USERFUNC name="predictor_nextstate">
   if(dir) return (state+1)%5;
   else return (state+4)%5;
</USERFUNC>
<USERFUNC name="predictor_output">
  if(state == 3) return random()%2;
  return (state>=2);
</USERFUNC>
```

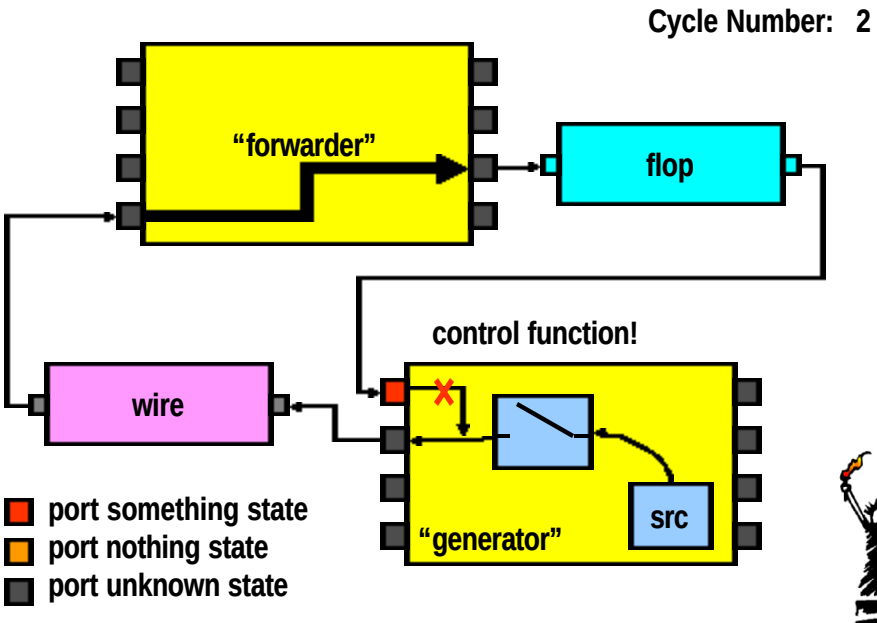
User Point Benefits

- Configurer can change algorithms to suit their needs without worrying about timing implications
- Configurer never needs to see the module code to change these algorithms
  - Allows modularity and greater reuse amongst modules
- Implemented as inline function calls
  - Can be used liberally at no cost to achieve maximum flexibility

Overriding Default Control

- First, what are the default control semantics?
  - Simple, channels accept new data only if they have free slots
  - If not, the sending module stalls
- We override default control by specifying control functions for control points
  - Much like user functions but control module communication and timing

Communication Semantics



Stall Control Point

```
<CONTROLFUNC name="predict">
  if(SIM_time_now % 2) {
    return SIM_decision_no;
  } else {
    return (SIM_decision_send |
            SIM_decision_update);
  }

  SIM_scheduler_run_after(SIM_time_construc
  t(0,1));
</CONTROLFUNC>
```

Control Point Benefits

- Can override default control without muddling in module code
  - As a result control is specified modularly
- Since modules can allow a variety of different timings we gain better reuse

What's the Deal with These Points?

- Control Points, User Points, Decode Points, and even parameters are ways of customizing modules
- Points are essentially parameters whose value is a code snippet rather than a number or a string
  - They act as holes in the module code that a user fills in
  - Defaults are provided for these holes to ease configuration burden
- Why distinguish between the user, control and decode points, aren't they just the same thing?

Code Points Explained

- User points
  - Allow the user to modify a modules algorithm and as such must be written in a conventional programming language
- Control Points
  - Allows modules to co-ordinate data flow with other modules
  - In the future, an analyzable language may take the place of C code
- Decode points
  - Allow a way for modules to discover information about the decoding or classification of instructions.
  - In the future, the information filling a decode point will be drawn from GLAD

Can We Have a Real Pipeline, Please?

- What about real pipelines?
- Won't they be a pain to configure with all these connections?
- How do I configure complex pipeline control with control points without losing my sanity?

Tutorial Sequence

8:30	Welcome	
8:35	Liberty Introduction and Motivation	August
9:00	The Simulation Environment	M. Vachharajani
	Fundamentals and The Visualizer	Blome
10:00	The Pipeline Builder	N. Vachharajani
10:30	Refreshment Break (10-30 Minutes)	
11:00	Machine Configuration Issues	Penry
	Configuration of "sim-outorder"	
	Complex Machine Configuration	
	BLISS – Both LIberty and SimpleScalar	
12:00	IA-64 Emulation/Simulation	Rangan
	Alpha Emulation/Simulation	Govindavajhala
12:15	Release Info and The Future of Liberty	August
12:30	Adjourn	

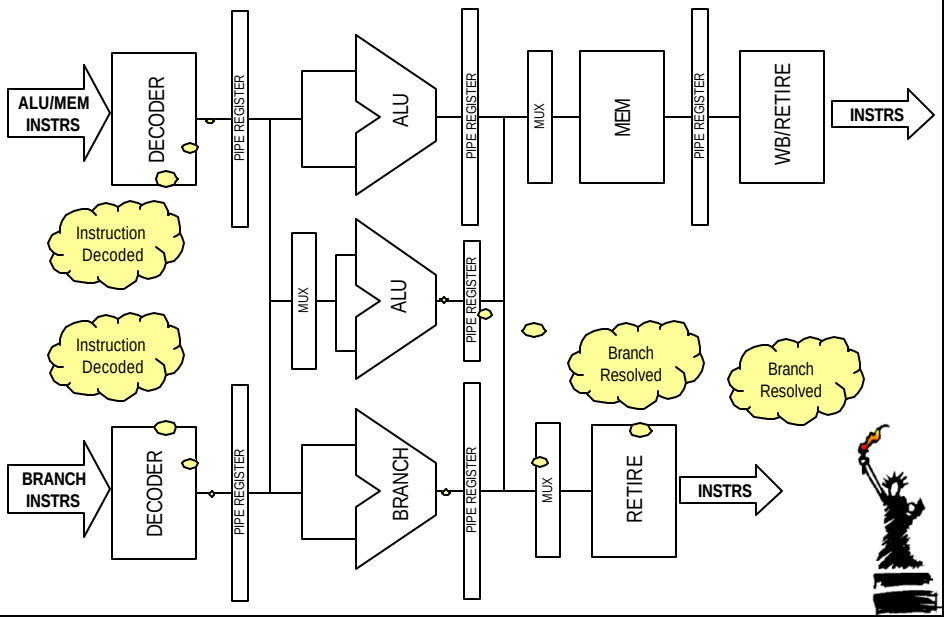
The Answer is….
The Liberty Pipeline Modeler

- The Liberty Pipeline Modeler is a special tool to make Pipeline description easier
- Rather than have a single pipeline module why not have a pipeline module builder!
- The “buildpipe” tool transforms an XML pipeline description into an LSE module
  - This module encompasses the entire pipeline behavior

The Liberty Pipeline Modeler
Why A Special Tool?

- Extremely Regular Structure
  - Easier to specify regular structure using a specialized language
  - Exposing regular structure can lead to simulation speed improvements
- Simplified Control
  - A custom pipeline specification language will allow for even more “implied” or default control
  - Complex intra-pipe control easier to specify
  - Automatic Data Dependency checking
- Compiler Interaction
  - Analysis of regular structure can aide in scheduling during compilation

Pipeline Example
The Data Path



Pipeline Example
Control

- Where do we need control?
  - A path decision needs to be made at the output of each decoder
  - A path decision needs to be made at the output of the 2nd ALU
  - A decision about whether or not to output a branch resolved signal needs to be made at the output of the 2nd ALU
  - Each MUX needs a select control signal
  - Each pipeline register needs a Write Enable signal

The “buildpipe” Way
Steps in Configuring a Pipeline

1. Describe the pipeline resources
   - Functional units
   - Instruction retirement bandwidth
   - External signal bandwidth (e.g. # of branches that can be resolved in a single cycle)
2. Partition the space of instructions into classes
   - Based on instruction type (memory, alu, branch, etc.)
   - Based on possible resource occupation
3. Describe Instruction Flow
   - One description per class of instructions
4. Specify non-implied control

The “buildpipe” Way
Steps in Configuring a Pipeline

1. **Describe the pipeline resources**
   - **Functional units**
   - Instruction retirement bandwidth
   - External signal bandwidth (e.g. # of branches that can be resolved in a single cycle)
2. Partition the space of instructions into classes
   - Based on instruction type (memory, alu, branch, etc.)
   - Based on possible resource occupation
3. Describe Instruction Flow
   - One description per class of instructions
4. Specify non-implied control

Specifying Resources

Functional Units

- All the functional units are grouped together using the <RESOURCES> tag
- Each functional unit is described using the <STAGE> tag
  - The “name” attribute names the resource
  - The “type” attribute tells us the type resource type
    - Most resources can be modeled as the “standard” type
    - Resources that do more than just timing will need to use other types
  - The “latency” parameter gives the resource a default latency

```
<RESOURCES>
    <STAGE name=“decoder1” type=“standard” />
    <STAGE name=“decoder2” type=“standard” />
    <STAGE name=“alu1” type=“standard”>
        <PARAMETERS>
            <PARM name=“latency” value=“2” />
        </PARAMETERS>
    </STAGE>
    …
    <STAGE name=“mem” type=“lsu” />
    <STAGE name=“branch” type=“standard” />
    …
</RESOURCES>
```

The “buildpipe” Way

Steps in Configuring a Pipeline

1. **Describe the pipeline resources**
   - Functional units
   - **Instruction retirement bandwidth**
   - **External signal bandwidth (e.g. # of branches that can be resolved in a single cycle)**
2. Partition the space of instructions into classes
   - Based on instruction type (memory, alu, branch, etc.)
   - Based on possible resource occupation
3. Describe Instruction Flow
   - One description per class of instructions
4. Specify non-implied control

Specifying Resources

Output Bandwidth

- To throttle output bandwidth, specify ports
- In our example
  - Two instruction retirement ports
  - Two external signals (instruction decoded, branch resolved)
- Retirement ports are specified with the <RETIRE> tag
  - The “name” attribute names the retirement port
- Signal ports are with the <SIGNAL> tag
  - The “name” attribute names the signal port
  - The “data_type” describes data type of the signal

```
<PORTS>
    <RETIRE name="out1">
        …
    </RETIRE>
    <SIGNAL name="instr_decoded1" data_type="none">
       …
    </SIGNAL>
    <SIGNAL name="branch_resolved1" data_type="none">
       …
    </SIGNAL>
    …
</PORTS>
```

The “buildpipe” Way

Steps in Configuring a Pipeline

1. Describe the pipeline resources
   - Functional units
   - Instruction retirement bandwidth
   - External signal bandwidth (e.g. # of branches that can be resolved in a single cycle)
2. **Partition the space of instructions into classes**
   - Based on instruction type (memory, alu, branch, etc.)
   - Based on possible resource occupation
3. Describe Instruction Flow
   - One description per class of instructions
4. Specify non-implied control

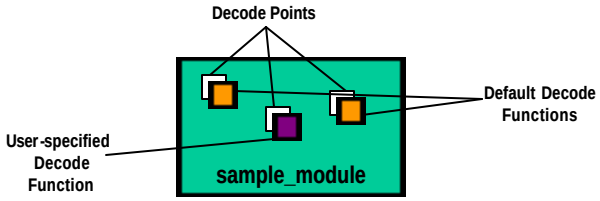
Partitioning Instructions into Classes
Syntax

- Two main paths through the pipeline
  - Branches come in through the lower decoder
  - All other instructions come from the upper decoder
- So we partition into two classes…
- Use the <CLASS> tag to specify instruction classes
  - Use the “name” attribute to name the instruction class
  - Use the “retire” attribute to specify the *default* retire port that this class of instructions will use

```
<INSTRS>
    <CLASS name=“other” retire=“out1”>
      <![CDATA[return !SIM_static_is_cti(info);]]>
    </CLASS>
    <CLASS name=“branch” retire=“out2”>
      <![CDATA[return SIM_static_is_cti(info);]]>
    </CLASS>
</INSTRS>
```

Partitioning Instructions into Classes
Decode Points

- The generated module must classify instructions when they arrive
- Decode Points save the day
  - **Decode Points** are holes in the module that are filled by **Decode Functions**
  - The generated module exports a decode point for each instruction class
  - Pipeline descriptions specify *default* decode functions to be used at these decode points



The "buildpipe" Way

Steps in Configuring a Pipeline

1. Describe the pipeline resources
   - Functional units
   - Instruction retirement bandwidth
   - External signal bandwidth (e.g. # of branches that can be resolved in a single cycle)
2. Partition the space of instructions into classes
   - Based on instruction type (memory, alu, branch, etc.)
   - Based on possible resource occupation
3. **Describe Instruction Flow**
   - One description per class of instructions
4. Specify non-implied control

Instruction Flow

Introduction

- The data path is modeled in pieces
  - Each instruction class describes a piece of the data path
  - Different instruction classes can describe the same part of the data path
  - Overlap allows control information to be inferred (more on this later)
- Instruction Flow
  - Each class specifies instruction flow using a **regular expression language**

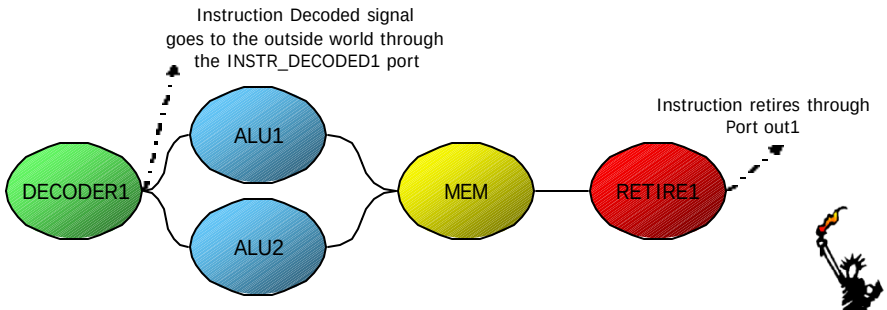
Instruction Flow
Regular Expression Language

- Alphabet
  - The set of pipeline stages form the alphabet for the regular expressions
- Accepted "strings"
  - Strings accepted by the regular expression specify a valid path through the pipeline
  - For simulation, options in the regular expression are marked with "decision points" to allow control to be embedded
- Operators
  - '+', '*', '?' – used to specify repetition (closure) and optional stages
  - ';', '|' – used to specify sequencing and alternation

Instruction Flow
Example

- Instruction Class: Other

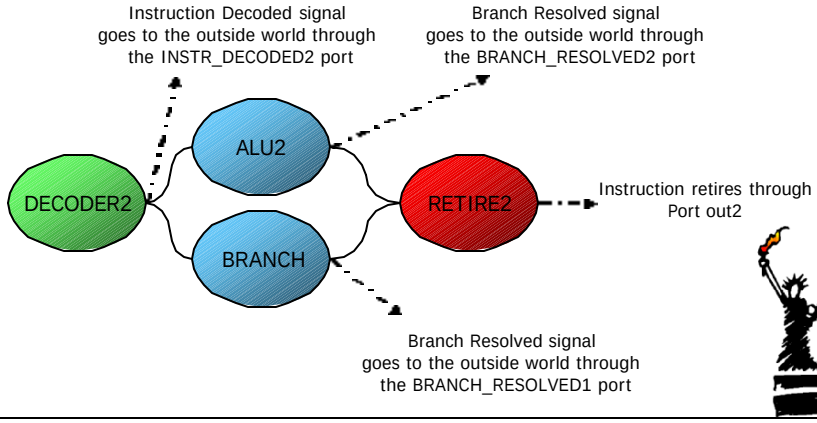
  DECODE1[INSTR_DECODED1 := NONE] ;
  (ALU1 | ALU2) ;
  MEM ;
  RETIRE1



Instruction Flow
Example (continued)

- Instruction Class: Branch

  DECODE2[INSTR_DECODED2 := NONE] ;
  (BRANCH[BRANCH_RESOLVED1 := NONE] | ALU2[BRANCH_RESOLVED2 := NONE]) ;
  RETIRE2



More Than Just Flow…
Implied Control

- Back pressure based control
  - Instructions move forward as long as there is nothing in the way
- Describing the data path in pieces allows for some control to be implied
- If we remember where we came from…
  - We may know where to go
  - We may know what signals to generate
- In the example, an instruction leaving ALU2 knows…
  - To go to MEM if it came from DECODER1
  - To go to RETIRE2 and generate the BRANCH_RESOLVED signal if it came from DECODER2
- In hardware terms, this corresponds to storing control information in pipeline registers
- Although a lot of control can be inferred, there's still more control left…

The “buildpipe” Way
Steps in Configuring a Pipeline

1. Describe the pipeline resources
   - Functional units
   - Instruction retirement bandwidth
   - External signal bandwidth (e.g. # of branches that can be resolved in a single cycle)
2. Partition the space of instructions into classes
   - Based on instruction type (memory, alu, branch, etc.)
   - Based on possible resource occupation
3. Describe Instruction Flow
   - One description per class of instructions
4. **Specify non-implied control**

Explicit Control
Where Do We Still Need Control?

- Most control can be specified using back pressure or inferred using the initial decode decision
- Explicit control is necessary when a decision cannot be inferred
  - Decision between multiple paths
  - Arbitration of a common resource
  - Non-local and irregular stall conditions
- In the example...
  - Instructions leaving each of the decoders need to make a path decision
  - The use of ALU2 needs to be arbitrated between DECODER1 and DECODER2

Explicit Control

How Do We Specify Control

- When specifying the instruction flow, nodes can be marked with **Decision Points**
- A decision point is a "hole" in the generated module's logic that gets filled with a **Decision Function** or gets exported as a **Control Point**
  - Decision functions allow for normal flow control, with user specified routing
  - Exporting a control point allows for routing and flow control decisions to be made jointly
- Decision points should be inserted wherever explicit control is necessary
- In our example, a decision point is needed after DECODER1 and after DECODER2
- This completes our example and the description of the Pipeline Modeler, so…

The Liberty Pipeline Modeler

Completed Example…

Lets see our example running in a simulator…

Tutorial Sequence

8:30	Welcome	
8:35	Liberty Introduction and Motivation	August
9:00	The Simulation Environment	M. Vachharajani
	Fundamentals and The Visualizer	Blome
10:00	The Pipeline Builder	N. Vachharajani
10:30	Refreshment Break (10-30 Minutes)	
11:00	Machine Configuration Issues	Penry
	Configuration of "sim-outorder"	
	Complex Machine Configuration	
	BLISS – Both LIberty and SimpleScalar	
12:00	IA-64 Emulation/Simulation	Rangan
	Alpha Emulation/Simulation	Govindavajhala
12:15	Release Info and The Future of Liberty	August
12:30	Adjourn	

Liberty is a Simulation Framework

- Modules provided with Liberty are just starting points
  - User can add own modules
  - All Liberty modules can be replaced, no "magic"
- Can "liberate" code from other simulators
  - Simplescalar models can be wrapped to form liberty modules
  - Simplescalar's interpreter can be used to drive Liberty

Experience with Liberty Configuration

Goal: Configure a basic SimpleScalar machine

1. Use BLISS-Interpreter to match instruction specifics
2. Configure a machine step by step
3. Demonstrate configuration modification ease

Things to Keep in Mind

- Configuration was done in 3 days
- Configuration was done WITH the Liberty framework
- Only a few architectural primitives available
  - Framework development vs. primitive library mass
  - New modules written along the way, are now in the library!

What is BLISS?

- BLISS is **B**oth **LI**berty and **S**imple**S**calar
- Import pieces of SimpleScalar
  - Interpreter
  - Can use SS functional code inside LSE modules
- SimpleScalar Interpreter is linked to an LSE Simulator
  - Configured simulator controls program execution
- We can also wrap SS code into modules
  - SS code provides functionality
  - LSE module provides timing and control
- We'll use the SS interpreter so we can run the same code as SimpleScalar

Simulator Construction Methodology

- How does one build a real simulator using the LSE?
- We'll configure several machines to illustrate
  - Simple program execution only with BLISS
  - Add a basic pipeline w/ only structural hazards
  - Add data dependencies
  - Prioritize instruction issue
  - Limit commit bandwidth
  - Fix up discrepancies with SS
  - Add speculative execution

Basic BLISS Configuratoin

- Just iseq_decider and the SimpleScalar interpreter

Adding a Pipeline

- Use mqueue in mutliple places with parms to specialize
  - Instruction queue
  - Issue queue/Dispatch Issue
- Choose/routing skeleton prototype
- sample-module for pipeline
- Limited set of primitives – stress framework
- Statistics – USE SS PIPEVIEW

Extend with data deps

- "Tagger" module – determine deps/renaming
- Completion logic – Extend routing skeleton (same inst name)
- EQUAL TO RUU!!!

Assign priorities to Instructions

- Sort function in MQ
- Except – still need: insts removed at WB(us) not at commit(SS)

Limit the Commit Bandwidth

- Limit commit bandwidth like SS
  - Commit in order
  - Need info in RUU sorted in program order, make another copy of RUU
  - No sort channel for now (many different ways to do things)

Remaining Modeling

- Cannot compare to SS – loads and stores issues twice
  - EAC first, LS next
  - Data from write back, keep state (extra state – extend dyn ids) on memory operators – check if mem-op don't complete, send to reservation station for reissue.
  - Limit LS to size of LSQ, limiting entries in RUU

Other Sources of Differences

- Differences remain
  - End program differently than SimpleScalar
  - No pipeline flush on system calls

Speculative Execution

- Add a branch predictor
- Add a resolution unit
  - Maintains program order and squashes miss speculation
  - Works in conjunction with the decider to ensure program order of execution

BLISS

- Now weird stuff
  - Turn off register renaming (code change)

Tutorial Sequence

8:30	Welcome	
8:35	Liberty Introduction and Motivation	August
9:00	The Simulation Environment	M. Vachharajani
	Fundamentals and The Visualizer	Blome
10:00	The Pipeline Builder	N. Vachharajani
10:30	Refreshment Break (10-30 Minutes)	
11:00	Machine Configuration Issues	Penry
	Configuration of "sim-outorder"	
	Complex Machine Configuration	
	BLISS – Both LIberty and SimpleScalar	
12:00	IA-64 Emulation/Simulation	Rangan
	Alpha Emulation/Simulation	Govindavajhala
12:15	Release Info and The Future of Liberty	August
12:30	Adjourn	

ISA and Simulators

- Built simulators can interact with a variety of ISAs
  - Standard interface defined
  - Emulators vs. Interpreters
- BLISS – an interpreter
- Custom Liberty emulators
  - IMPACT Lcode
  - IA-64
  - Alpha
- Multiple ISA demo...
- Future: Emulator and Interpreter builder from GLAD

Tutorial Sequence

8:30	Welcome	
8:35	Liberty Introduction and Motivation	August
9:00	The Simulation Environment	M. Vachharajani
	Fundamentals and The Visualizer	Blome
10:00	The Pipeline Builder	N. Vachharajani
10:30	Refreshment Break (10-30 Minutes)	
11:00	Machine Configuration Issues	Penry
	Configuration of "sim-outorder"	
	Complex Machine Configuration	
	BLISS – Both LIberty and SimpleScalar	
12:00	IA-64 Emulation/Simulation	Rangan
	Alpha Emulation/Simulation	Govindavajhala
12:15	Release Info and The Future of Liberty	August
12:30	Adjourn	

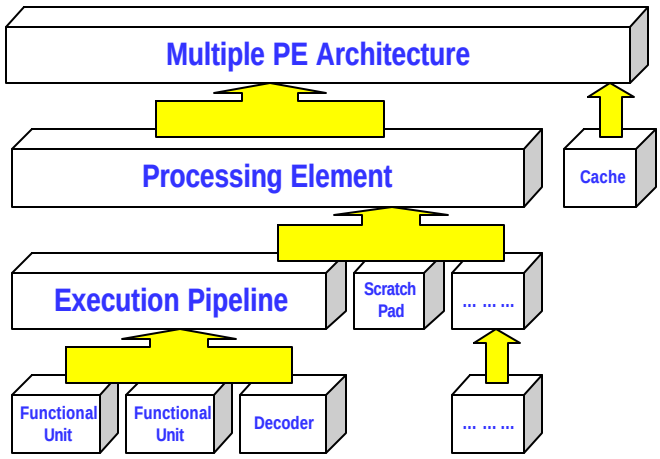
The Future of Liberty

- Near-term
  - Module Extensions, Power Models
  - Hierarchy
  - Emulator, Interpreter Builder
  - Open Component Libraries
- Long term
  - Compiler Release
    - Self-tuning optimizations
    - Run-time optimization
  - Simulator
    - Schedule Analysis
    - Control Specification Language
    - Multiprocessor Support
    - Heterogeneous Processor Support
    - ...

Module Extensions (February)

- Extensions enhance modules
  - Doesn't disturb original module code
  - Can access and modify internal state
- Extends module by adding
  - Parameters, Code Points, Events
  - State
- Code assembled by simulator builder
  - Simulator builder is like a compiler for an object oriented language
  - Module extensions are like of static inheritance
- Example: Power Models (Submission: Hangsheng Wang)
  - Write performance simulator
  - Extend with module extensions

Hierarchical Descriptions (January)



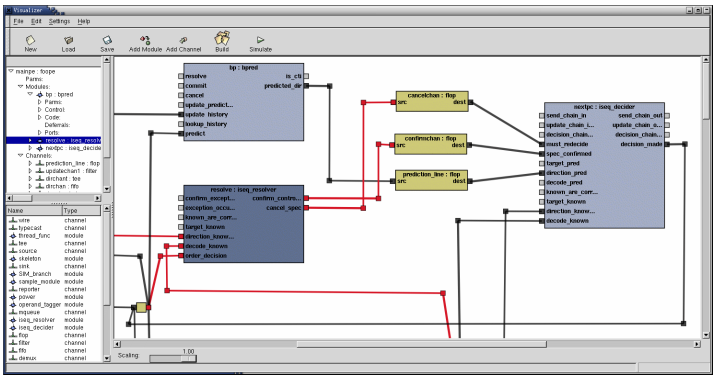
- Useful for describing multiprocessor machines
- Enables sub-description reuse

Emulator Builder (Summer)

- GLAD describes ISA
  - Functional description of each instruction
  - Offers intermediate results – (address calculation, multiply-add, etc.)
  - Creates instruction classes (CTI, Memory, etc.)
- Emulator Builder
  - Builds enhanced compiled-code emulator macros from GLAD
  - Macros are used to compile programs
- Interpreter Builder
  - Use same GLAD description
  - Slower, but supports self modifying code and run-time opti
- Currently – bypass builders
  - Current ISAs: Lcode (IMPACT), Alpha, IA64, x86
  - Compiler "front-ends" for each of these

Open Component Library (Summer)

- Liberty is a framework!
- Libraries of modules, channels, designs on the internet
  - Publish architectural components from your research
  - Anonymizer for peer review?
- Browse libraries from within the visualizer



The Future of Liberty

- Near-term
  - Module Extensions, Power Models
  - Hierarchy
  - Emulator, Interpreter Builder
  - Open Component Libraries
- Long term
  - Compiler Release
  - Self-tuning optimizations
  - Run-time optimization

Liberty Release

- Think bazaar, not cathedral
  - Think Linux
  - Not Windows XP
- Liberty software is non-copylefted free software
  - Think X-Windows license
  - Not Windows XP license

Release Schedule

Initial Release	Simulator Builder, Modules, Channels, BLISS, and Visualizer
Q1-2002, Intel Pro64 Release	IA-64 Emulator, compiler IR and Library

- Check http://liberty.princeton.edu for release details
- Mailing Lists:
  - liberty-sim@cs.princeton.edu - discussion group for simulator
  - liberty@cs.princeton.edu – announcements
  - Subscription request form on web-site

Thank you!

- Let us know what you think, we welcome your feedback!
  - http://liberty.princeton.edu/
  - David August <august@cs.princeton.edu>
- Thank you!!!