RAMP: Architecture, Language & Compiler

http://ramp.eecs.berkeley.edu
Greg Gibeling
gdgib@berkeley.edu
1/19/2006

Outline

- Introduction to RAMP
- RAMP Architecture
- Target Model
- Host Model
- Tools & Toolflow
- RAMP Description Language
- Status & Future Work

Introduction to RAMP (1)

- FPGAs as a research platfom
  - About ~16 CPUs can fit in Field Programmable Gate Array
  - 1000-CPU system from ~ 60 FPGAs? (15 BEE2s)
  - FPGA generations every 1.5 yrs; 2X CPUs, 2X clock rate
- HW research community does logic design (gateware)
  - Create out-of-the-box, massively parallel system
    - Runs full OS
    - Allows OS, compiler and application developement
  - Gateware: Processors, Caches, Coherency, Ethernet Interfaces, Switches, Routers, …
  - E.g., 1000 IBM Power cache-coherent supercomputer

Introduction to RAMP (2)

- A framework for system emulation
  - Massively parallel (digital hardware) systems
  - Orders magnitude performance enhancement
  - Leverage existing designs
  - Allow community development
    - Share designs, validate experiments, etc…
- Flexible, cross platform designs
  - Requires proper structure
    - Support for automatic debugging
    - Automatic glue logic/code generation
  - Based on the “target model”

RAMP Architecture (1)

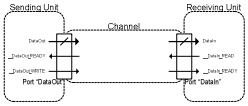
- Target
  - The system being emulated
    - Actually only a model of the system being emulated
    - Can be a cycle accurate model
  - Must conform to the RAMP target model
- Host
  - The system doing the emulation
  - May include multiple platforms
    - Hardware – BEE2, XUP, CaLinx2
    - Software – Java, C, C++

RAMP Architecture (2)

- Fundamental Model
  - Message passing
  - Distributed event simulator
  - Message passing system generator
    - Cross platform
    - Shared development effort
    - Easy to develop, debug and analyze
  - Similar Formalisms
    - Petri Nets
    - Process Networks
    - Research: Click, P2, Ptolmey, Metropolis, etc….

RAMP Target Model (1)

- Units communicate over channels
- Units
  - 10,000+ Gates
    - Processor + L1
  - Implemented in a "host" language
- Channels
  - Unidirectional
  - Point-to-point
  - FIFO semantics
  - Delay Model

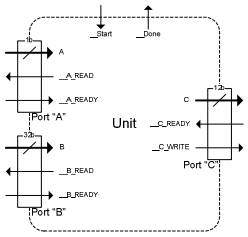


RAMP Target Model (2)

- Transaction style unit semantics
  - Read 0/1 messages from each input
  - Perform some action
  - Write 0/1 messages to each output
  - Units MUST be latency insensitive

Target Model - Units

- Inside edge
  - Ports connect units to channels
    - FIFO signaling
    - Hardware or Software
  - Target cycle control
    - __Start
    - __Done
    - Allows for variable timing, and timing accurate simulation

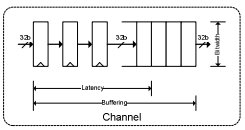


Target Model – Channel (1)

- Channel semantics
  - Arbitrary message size
    - The messages are statically typed
  - Ordered delivery
  - Debugging through monitoring & injection
  - Provides for cross-platform simulations

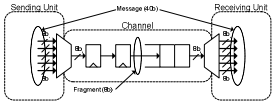
Target Model – Channel (2)

- Channel Params
  - Only used for timing accurate simulations
  - Bitwidth
  - Latency
  - Buffering
- Fragments
  - Smaller than messages
  - Convey the simulation time through idles



Target Model – Channel (3)

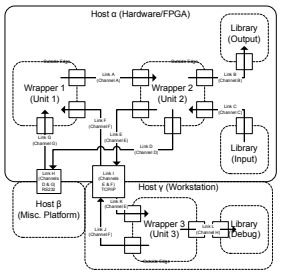
- Fragmentation
  - Messages may be larger than fragments
  - Channels carry (per target cycle):
    - Zero or one fragments
    - Between zero and one messages
- Timing Model
  - Zero fragments still indicates passage of time
  - Implement outside of the units



Target Model - Debugging

- Monitoring
  - All communication is over channels
    - Can be examined and controlled
  - Real time can be paused or slowed down
    - Target cycles are completely subjective
- Injection
  - Makes developing test benches easy
    - Simply inject a sequence of messages
  - Cross platform comm is hidden by RDLC

Host Model

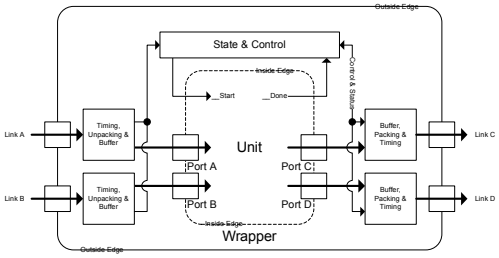


- Cross platform
  - Units implemented in many languages
  - Library units for I/O
  - Links implement channels
- Links
  - Any communication
  - Less defined

Host Model – Wrapper (1)

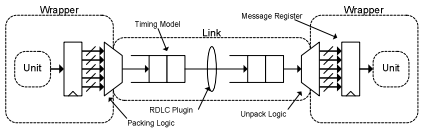
- Inside Edge
  - Wrapper provides this interface to unit
  - Clean, mostly free of implementation
    - Hardware/software differences will be visible
- Outside Edge
  - Implementation dependant
  - Deals with physical links
- Generated by RDLC
  - The wrapper
  - All of the links
  - Uses plugins for easy extensibility

Host Model – Wrapper (2)



Host Model - Link

- Typically Three Components
  - Packing & Unpacking
  - Timing Model
  - Physical Transport



RDL (1)

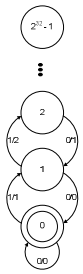
- "RAMP Description Language"
- General message passing system description language
- Compiler includes back-end extensibility
  - Links, other toolflows, external signals
- Does NOT include the functionality of the units!

RDL (2)

- Hierarchical Namespaces
  - Declarations can be external to a namespace
  - Replaces “preprocessing”
  - Allows for communal development
- RDL Target Constructs
  - Channels, Messages and Port types
  - Units include instances, inputs, outputs and connections
- RDL Host Constructs
  - One platform per board or computer
  - Platforms include an implementation language
  - Hierarchy allows for, eg. A board with many FPGAs
- RDL Mappings
  - Hierarchy allows for “compile one, run many”
  - Allows specific units and channels to be precisely mapped

RDL Example (1)

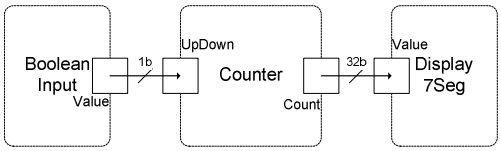
- Up/Down Counter
  - Push button input
  - 4-bit LED output
- Utility as example
  - Proof of concept
  - Simple language example
  - Hands on demo



RDL Example (2)

```
unit {
    input  bit[1]  UpDown;
    output bit[32] Count;
}                  Counter;
```


RDL Example (3)



RDL Example (4)

```
unit {
      instance IO::BooleanInput  BooleanInput;
      instance Counter           Counter;
      instance IO::Display7Seg   Display7Seg;

      channel  fifopipe[1, 1, 1] InChannel
            { BooleanInput.Value -> Counter.UpDown };
      channel  fifopipe[32, 1, 1] OutChannel
            { Counter.Count -> Display7Seg.Value };
}                                CounterExample;
```


RDL Example (5)

```
channel … InChannel { BooleanInput.Value -> Counter.UpDown };
```

```
instance IO::BooleanInput BooleanInput(InChannel);
channel … InChannel;
```

```
instance IO::BooleanInput BooleanInput(Value(InChannel));
channel … InChannel;
```

RDL Example (6)

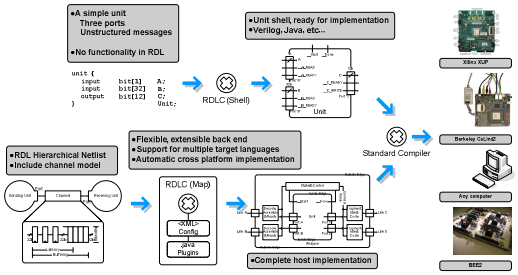
```
platform {
      language            "verilog";
      default link        "RegisterLink";
      engine              "XUPEngine";
}                         XUP;


map {
      platform XUP        BasePlatformInst;
      unit CounterExample BaseUnitInst;
}                         XUPCounter;
```


RDLC Toolflow (1)

- Development Steps
  - Unit Implementation
    - RDL unit descriptions
    - RDLC generates shell code in a specific language (Verilog, Java…)
    - Researcher adds implementation code
  - RDL target design
    - Includes Mapping
    - RDLC generates complete implementation code
    - Includes all links, instantiates all unit shells

RDLC Toolflow (2)



RDLC Toolflow (3)

- Help: rdlc -help
- Support
  - Generates support files for a specific lang
  - rdlc –support:"verilog" dummy.rdl outdir
- Shell
  - Generates an inside edge shell
  - rdlc –shell:"Counter Platforms::XUP" CounterExample.RDL outdir
- Map
  - Generates a complete emulation
  - rdlc –map: "Maps::XUPCounter false" CounterExample.RDL outdir

RDLC Back End (1)

- Language Families
  - Hardware
    - Verilog – Completely finished
    - VHDL – Trivial addition
  - Software
    - Java in progress
    - C/C++ planned
  - RDL
  - Your language here?

RDLC Back End (2)

- Back End Plugins
  - Allow for toolflow integration
    - Important for dealing with complex platforms without requiring detailed knowledge of all the tools involved.
    - XFlow & Impact
    - ModelSim – Next Week
  - Include
    - Allows for including extra files (support modules)
    - Unit implementations!

RDLC Back End (3)

- Configuration
  - Back End XML Config File
    - Add an element to add a language back end
  - Uses Java reflection to load the code
  - Language Specific
    - Link plugins for each language
    - Dependant on the language back end
    - Hardware/Verilog is the model

State of the Project

- **Working hardware implementation!**
  - Compiled RDL to Verilog
  - Tested on a CaLinx2 and XUP Board
  - Java and BEE2 should be done before Feb 1, 2006
- RDL & RDL Compiler
  - RDL is stable
    - Some advanced features are in flux
    - Ready for use!
  - Working compiler, written in java
  - Powerful parser & output generators
    - Easily extensible
    - Software (Java) back end almost complete

Future Work

- RDL & RDLC Features
  - Language Features
    - Generated code, port arrays and compile time parameters
  - Significant additions to back end
    - Languages, platforms, links
    - Debugging automation
- Documentation
  - Architecture, Language & Compiler Technical Report
  - Complete compiler internals documentation
- Automated Testing
  - Regression tests for the compiler
  - Automated test code generation for links and units

Website & Community

- http://ramp.eecs.berkeley.edu
- Planned Features
  - Wiki – Certainly not JSPWiki
  - CVS Repository
  - Mailing list control and archives
  - Communal project listings
- Ideas, Comments & Requests
  - gdgib@berkeley.edu

CounterExample Lab

- The Basic Lab
  - Code walkthrough
  - Use the 3 RDLC commands
  - Put the design on the XUP board
- The Advanced Lab
  - Use the CaLinx2 board
  - Running ModelSim
  - Build your own
  - Crash the compiler