The RAMP Architecture & Description Language

Greg Gibeling, Andrew Schultz & Krste Asanović
RAMP Gateware Group, UC Berkeley & MIT CSAIL
{gdgib & alschult}@eecs.berkeley.edu, krste@csail.mit.edu

January 17, 2006

1 Introduction

The RAMP (Research Accelerator for Multiprocessors) project is developing infrastructure to support high-speed emulation of large scale, massively parallel multiprocessor systems using FPGA platforms. In this paper, we describe our proposal for a RAMP Design Framework (RDF), which has a number of challenging goals. The framework must support both cycle-accurate emulation of detailed parameterized machine models and rapid functional-only emulations. The framework should hide changes in the underlying RAMP hardware from the module designer as much as possible, to allow groups with different hardware configurations to share designs and to allow RAMP modules to be reused in subsequent hardware revisions. In addition, the framework should not dictate the hardware design language chosen by developers.

Our approach was to develop a decoupled machine model and design discipline, together with an accompanying RAMP Description Language (RDL) and compiler to automate the difficult task of providing cycle-accurate emulation of distributed communicating components.

2 RDF Overview

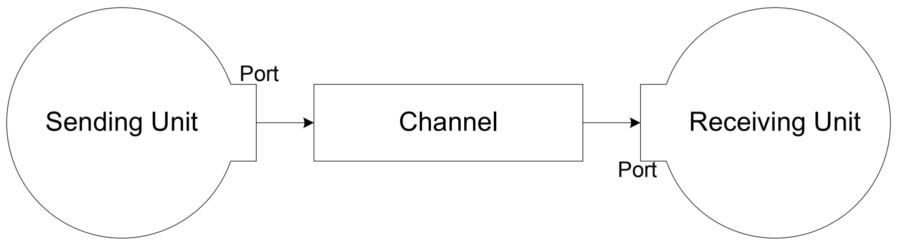
A configured RAMP system models a collection of CPUs connected to form a cache-coherent multiprocessor. The emulated machine is called the *target*, and underlying FPGA hardware (e.g. BEE2) is the *host*. The RAMP design framework is based on a few central concepts. A RAMP configuration is a collection of loosely coupled *units* communicating with latency-insensitive protocols over well-defined *channels*. Figure 1 gives a simple schematic example of two connected units. In practice, a unit will be a large component corresponding to tens of thousands of gates of hardware, e.g., a processor with L1 cache, a DRAM controller, or a network router stage. All communication between units is via messages sent over unidirectional point-to-point inter-unit *channels*, where each channel is buffered to allow units to execute decoupled from each other.

Each unit has a single clock domain. The target clock rate of a unit is the relative rate at which it runs in the target system. For example, the CPUs will usually have the highest target clock rate and all the other units will have some rational divisor of the target CPU clock rate (e.g., the L2 cache might run at half the CPU clock rate). The physical clock rate of a unit is the rate at which the FPGA host implementation is clocked. In some cases, a unit might use multiple physical clock cycles to emulate one target clock cycle, or even require a varying number of physical clock cycles to emulate one target clock cycle. At least initially, we expect that the whole RAMP system will have the same physical clock rate (nominally around 100 MHz), perhaps with some higher physical clock rates in I/O drivers.

All channels are unidirectional and strictly point-to-point between two units. The two units at each end of a channel can have different target clock rates, but, at least for the initial RAMP standard, must have the same physical clock rate. Units are only synchronized via the point-to-point channels. The basic principle is that a unit cannot advance by a target clock cycle until it has received a target clock cycle's worth of activity on each input channel and the output channels are ready to receive another target cycle's worth of activity. This scheme forms a distributed concurrent event simulator, where the buffering in the channels allows units to run at varying physical speeds on the host while remaining logically synchronized in terms of target clock cycles.

Unit designers must produce the RTL code of each unit in their chosen hardware design language or RTL generation framework, and specify the range of message sizes that each input or output channel can carry. For each supported hardware design language, the RAMP framework provides tools to automatically generate a unit wrapper (see following Section) that interfaces to the channels and

Figure 1 Basic RAMP communication model.



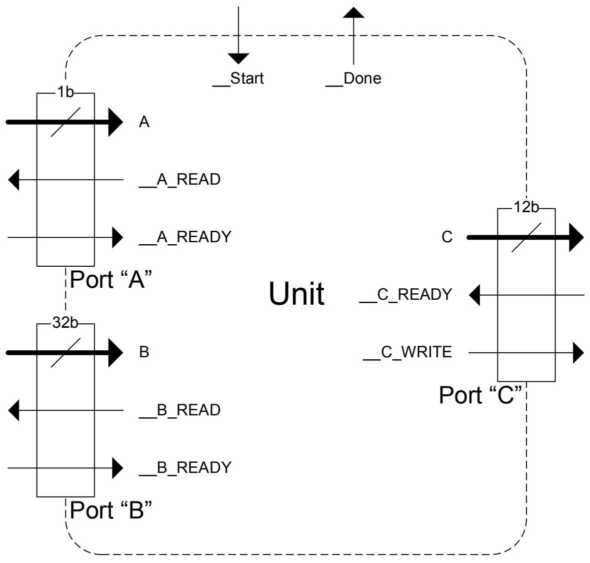
provides target cycle synchronization. The RTL code for the channels is generated automatically by the RDL compiler from an RDL description, which includes a structural netlist specifying the instances of each unit and how they are connected by channels.

The benefit of enforcing a standard channel-based communication strategy between units is that many features can be provided automatically. Users can vary the target latency, target bandwidth, and target buffering on each channel at configuration time. The RAMP configuration tools will also provide the option to have channels run as fast as the underlying physical hardware will allow to support fast functional-only emulation.

We are also exploring the option of allowing these parameters to be changed dynamically at target system boot time to avoid re-running the FPGA tools when varying parameters for performance studies. The configuration tool will build in support to allow inter-unit channels to be tapped and controlled to provide monitoring and debugging facilities. For example, by controlling stall signals from the channels, a unit can be single stepped. Using a separate automatically-inserted debugging network, invisible to target system software, messages can be inserted and read out from the channels entering and leaving any unit.

A higher-level aspect of the RAMP design discipline is that the operation of a unit cannot depend on the absolute or relative latency of messages between units (i.e., all inter-unit communication must be latency insensitive). We do not believe this unnaturally constricts unit design, as latency-insensitive protocols are commonly used in real hardware systems. We emphasize that units are intended to represent large pieces of a design and that it is not intended that channels will be used at a fine-grain, such as within a CPU. Any rigid pipeline timing dependencies must be contained within a unit. For example, primary caches will usually be implemented as part of a CPU unit.

Figure 2 Target-level unit interface.



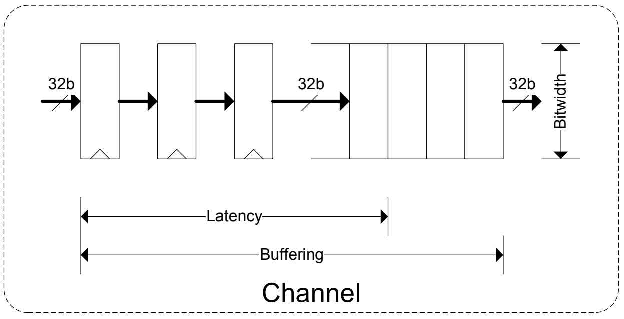
3 Implementation Details

3.1 Unit Interface

Figure 2 shows the interfaces a RAMP unit must support. Ports comprise the input and output interfaces between units and each port is connected to another port via a channel. The example unit has two input ports (A & B), and one output port (C). In addition to the ports there are two connections: `__Start`, which is used to trigger the unit to perform one target clock cycle's worth of action, and `__Done`, which is used to report back to the harness when the unit has completed the target clock cycle. While the ports in the example have a simple fixed message size, the RDL compiler supports complex messages through structures and tagged unions. By automatically building the support machinery (be it hardware or software) to marshall and transport complex messages, the compiler automates a large and tedious part of the emulation design process.

As the figure shows, each port has a FIFO-style interface, which provides a natural match to the

Figure 3 Channel model with parameters.



channel semantics as described in detail in Section 3.2. Input messages are consumed by asserting the appropriate `__Xxx_READ` when the associated `__Xxx_READY` is asserted. Similarly output messages are produced by asserting `__Xxx_WRITE`, when the associated `__Xxx_READY` is asserted. It should be noted that while the above description referred to "signals," which can be "asserted," these constructs can just as easily be represented in software.

We use the term **inside edge** to refer to the interface shown in Figure 2, the goal of which is to decouple the implementation of the unit from the rest of the target (and host) system as much as possible. Currently a complete decoupling is impossible due to the limitaton that the number and types of the ports (see Section 3.2) must be statically assigned to each unit at design time. However it is our hope that parameterization, polymorphism and optional port connections will, in the future, improve design flexibility.

3.2 Channel Model

The key to inter-unit communication lies in the channel model, which, along with the inside edge interface, forms the core of the target model. The channel model can be quickly summarized as lossless, strictly typed, point-to-point, and unidirectional with ordered delivery. This should be intuitively viewed as being similar to a FIFO with a single input and output, which carries strictly typed messages. This section expands the above description with the timing parameters necessary for timing-accurate simulations.

There are three parameters associated with every channel: **bitwidth**, **latency** and **buffering**, as illustrated in Figure 3. The bitwidth of a channel (the number of bits it can carry per target cycle) is the size of a **fragment**. Latency is the minimum number of target cycles which a fragment must take to traverse the channel.

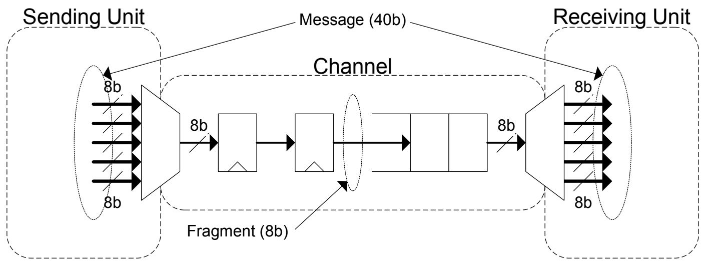
A **message** in RAMP is the unit of data carried between units, however to perform cycle-accurate simulations, messages may be split into fragments. Figure 4 illustrates the difference between a message and fragment. The channel carries exactly zero or one 8-bit fragments on each target cycle, but the units wish to communicate using 40-bit messages. Therefore the messages must be split into five 8-bit fragments for transport over the channel at a rate of one fragment per target cycle. This means that the sending unit may send at *most* one 40-bit message every five target cycles.

Of course, the inverse example is equally valid: a message may be smaller than the fragment size of the channel. In this case a message may be sent on every target cycle. However, a channel can only carry zero or one fragments per target cycle, which means a channel cannot carry more than one message per target cycle.

Fragmentation allows RAMP to decouple the size of messages, which is a characteristic of a unit design, from the size of data moving through the channels. This simplifies experimentation with varying target channel bandwidths, as target channel bandwidths can be modified without changing the target unit design.

The final channel parameter, **buffering**, is then defined as the number of fragments which the sender may send before receiving any acknowledgement of reception (as in a credit-based flow-control scheme). In general, a channel which must support maximum-bandwidth communications will require $\mathit{buffering} \geq 2 * \mathit{latency}$ to tolerate the latency in both directions: the data transfer (fragments moving forward), and the handshaking (credits moving backward). At startup, the sending unit will be given a number of credits equal to the buffering capacity of the channel, thereby allowing it to send that many fragments prior to the receipt of any additional credits. The cost of *latency* cycles for transfer in either direction is the reason for the $\mathit{buffering} \geq 2 * \mathit{latency}$ to achieve $\mathit{bandwidth} = \mathit{bitwidth}$.

Figure 4 Message fragmentation and target cycles.



Program 1 A 32bit Up/Down counter in RDL.

```
unit {
    input bit[1] UpDown;
    output bit[32] Count;
} Counter;
unit {
    instance IO::SwIn UserIn
        (Value(InChannel));
    instance Counter Counter
        (UpDown(InChannel),
         Count(OutChannel));
    instance IO::LEDOut UserOut
        (Value(OutChannel));
    channel fifopipe[1, 1, 1]
        InChannel;
    channel fifopipe[32, 1, 1]
        OutChannel;
} CounterExample;
```

4 RDL

We have developed the RAMP Description Language or RDL (pronounced "riddle") to allow a standard representation of RAMP designs, thereby easing all aspects of developement, and collaboration in particular. RDL is, in essence, a hierarchical netlisting language. The behavior of leaf units is given separately in a host specific language such as Verilog, VHDL, Bluespec, C, C++, or Java.

Program 1 is a very simple fragment of RDL for an up/down counter, and it's instantiation in a simple test. At the time of this writing, we have completed the implementation of the RDL compiler, and this simple example to the point of having a working demo running on a Xilinx FPGA, using switches and LEDs for I/O.

While this is an extremely simplistic example, and much smaller than a typical unit, it illustrates the basics of RDL. The ::Counter is declared to accept unstructured 1-bit messages at its port "UpDown" (::Counter.UpDown) and produce 32-bit messages at its output port "Count" (::Counter.Count). Of course this is a leaf unit, which will be implemented directly in the host language (Verilog, Java or C for example).

RDL and the compiler also support hierarchically defined units like "CounterExample" in this code fragment. Inside this unit, there are two channels, shown with detailed timing models, which are used to connect the three unit instances. Note the use of named port connections similar to Verilog. Positional and explicit port specification is allowed, including the ability to specify connection of a local channel to a port significantly lower in the hierarchy, without explicit pass-through connections at each level.

RDL also supports declarations for host platforms (eg. an FPGA board or computer with specific I/Os) and mappings of a top-level unit onto a platform. Platform declarations include the language (e.g., Verilog or Java) to generate and the specific facilities available for implementing channels on the host. The back end of the compiler is easily extensible to support new languages, and new host implementations.

A mapping from a unit to a platform may also include more detailed mappings to specify the exact implemention of each channel. The compiler can take just such a mapping and produce all of the necessary output to instantiate and connect the various leaf units, which have been implemented in the host language.

5 Status & Future Work

The RDL compiler has been completed and is fully documented. The RAMP description language and the RDL compiler are stable, with working examples, and are ready for research use. Timing-accurate simulations have been mostly implemented, but remain untested. In the future we will be working to expand RDL and the RDL compiler to work with a wider variety of platforms and to more concisely describe very complex systems.