Generic Netlist Representation for System and PE Level Design Exploration

Bita Gorjiara, Mehrdad Reshadi, Pramod Chandraiah, Daniel Gajski
Center for Embedded Computer Systems, University of California, Irvine
{bgorjiar, reshadi, pramodc, gajski}@cecs.uci.edu

ABSTRACT

Designer productivity and design predictability are vital factors for successful embedded system design. Shrinking time-to-market and increasing complexity of these systems require more productive design approaches starting from high-level languages such as C. On the other hand, tight constraints of embedded systems require careful design exploration at system level (coarse grained exploration) and at the processing-element (PE) level (fine grained exploration).

In this paper we presented GNR, a formal modeling approach, developed to improve productivity of designing systems and processing elements, the same way that traditional ADLs improved productivity for designing processors. The GNR is an order of magnitude shorter than state-of-the-art ADLs with RTL generation capabilities and yet can capture any structural details that affect the implementation quality. Using relatively short GNR description, we explored several designs for implementing an MP3 decoder and achieved 3.25 speedup compared to MicroBlaze processor. We have also developed a web-based interface for our tools, so that users can upload and evaluate new architectures described in GNR. Our toolset and GNR is an intermediate step towards synthesis of TLM to RTL.

Categories and Subject Descriptors

B.5.2 [**Design Aids**] Automatic synthesis; C.0 [**General**] Systems specification methodology, Modeling of computer architecture.

General Terms

Design, Performance, Languages.

Keywords

Architecture Description Language, application-specific processor, system design, modeling, synthesis, NISC, GNR.

1. INTRODUCTION

Designer productivity and *design predictability* are vital factors for successful embedded system design. Shrinking time-to-market and increasing complexity of these systems require more productive design approaches. Hence, embedded systems are increasingly designed using software (high-level languages such as C) rather than directly implementing them in RTL. Tight constraints of embedded systems require careful design exploration at system level (coarse grained exploration) and at the processing-element (PE) level (fine grained exploration). Such explorations can result in considerable improvement in terms of performance, power consumption, area, and manufacturability. Furthermore, we believe that design flows that give more control to the designers over the final implementation will generate more predictable results.

Architecture Description Languages (ADLs) have been proven to be productive for design of Application Specific Instruction-set Processors (ASIP). The ADL captures the behavior or structure of the processor and is used by the tools that compile the application and simulate the results. A few approaches have also offered automated or semi-automated RTL synthesis of the processor, which can improve the designer's productivity. It is desired to extend the ADL-based approaches to capture the entire systems as well. However, ADL-based design flows always assume that the architecture has a predefined instruction-set. This assumption creates three problems: (a) they cannot be used for dedicated hardware executing a fixed application (IP), where instructions impose unnecessary overhead; or for the entire system, where no instruction-set can be defined; (b) such ADLs are lengthy and complex because they contain either behavioral description of all instances of instructions, or structural description of the instruction decoder.; (c) generated RTL from instruction behaviors has unpredictable quality.

To address the above issues, in this paper we present a Generic Netlist Representation (GNR) that can be used for generating programmable and dedicated custom pipelined IPs from high level C description of the application. It can capture a single IP or a system composed of several IPs. In contrast to ASIP approaches, our target processing elements (PEs), called No-Instruction-Set-Computers (NISC), do not have a predefined instruction-set. In our approach, the accurate netlist of the datapath components is described GNR. Using this GNR, a *cycle-accurate compiler* compiles C code of the application directly on the input datapath and generates the control words for each clock cycle. The outputs of this compiler and the input GNR is used to generate the simulatable and synthesizable RTL code of the PE. Generally, most of the designer's experience, skill and innovation go into the design of datapath. Our approach improves *design predictability* by giving the designer complete control over the datapath. On the other hand, design of the controller is tedious, time consuming and error-prone process. By automating this process and by allowing reuse of previously designed datapaths and components, *designer productivity* is also significantly improved in our approach.

The GNR can also capture a system containing several communicating custom IPs. It can be used as the output of TLM-based synthesis tools. After modeling and verifying a system in transaction level, it can be converted to GNR for synthesis. Each low level TLM communication command (e.g. send/receive) is mapped to an intrinsic C function representing a communication component at the hardware level. In this paper, we present a formalism for modeling a system and its components including programmable and dedicated custom pipelined IPs. The GNR is formal and hence it allows checking rules and reducing semantic errors in the design. It provides support for third-

CODES+ISSS'06, October 22–25, 2006, Seoul, Korea.

party cores, and the same GNR description is used for compilation, simulation and RTL generation. Since the designer does not describe the controller in our approach, the GNR descriptions are much shorter than other ADLs. We have developed a web-based interface for our toolset, so that users can upload and evaluate new architectures described in GNR. Our compiler supports various architectural features such as controller/datapath pipelining, multi-cycle/pipelined units, and heterogeneous forwarding paths. The compilation algorithm and the datapath optimizations have been discussed in [9] and [10], respectively.

The rest of the paper is organized as follows. Section 2 and 3 explain the GNR modeling approach and its syntax. Section 4 discusses the details of GNR using several examples. Section 5 presents the flow of our tools, followed by experimental results in Section 6. Section 7 presents related works and Section 8 concludes the paper.

2. GNR MODELING APPROACH

GNR models a system as a hierarchical description of components (objects) and their connections (composition). GNR contains a set of predefined components and port types. These types are used for enforcing the composition rules. A typical system consists of several RTL components and processing elements (PEs). The behavior of each PE is captured in C language. In GNR, the PEs are represented by components of type *behavioralIP*. A *behavioralIP* may contain a custom datapath that is captured by a component of type *NiscArchitecture*. The *NiscArchitecture* contains basic RTL components that are used by our compiler. Figure 1 shows a simple example of a system with two PEs (BIP1, BIP2), a bus, and an arbiter. BIP2 is implemented by a programmable *NISC* and has a control memory (Cmem) and data memory (Dmem). In the rest of this section, we present the details of the GNR objects and compositions rules.

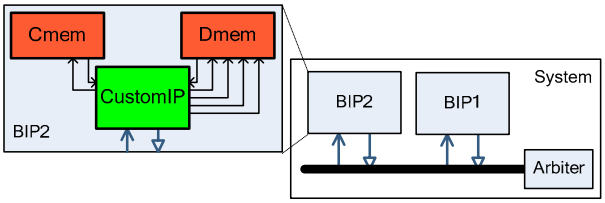


Figure 1- A sample system in GNR.

2.1 GNR formalism

In GNR, a component x is represented by (τ_x, P_x, C_x, L_x, A_x), where τ_x is the component's type, P_x is the set of ports, C_x is the set of components inside x, L_x is the set of its internal point-to-point connections, and A_x is the list of aspects that describe behavior of x for different tools in the toolset. Component type τ_x is defined as follows:

$\tau_x \in T$, T={*register*, *register-file*, *bus*, *mux*, *tri-state buffer*, *functional-unit*, *memory-proxy*, *controller*, *NiscArchitecture*, *behavioralIP*, *module, system*}

Where, *NiscArchitecture*, *behavioralIP, module*, *system,* and *controller* are hierarchical components and contain an internal netlist, while others are basic RTL components with no internal netlist.

Each port p in P_x has a bit-width β_p, and a type θ_p defined as follows:

$\theta_p \in$ {*clkPort*, *ctrlPort*, *inPort*, *outPort*, *cwPort*}

Type *clkPort* shows the port is a clock, and type *ctrlPort* shows the port is used to control the component. For example, a register has one port of each type *clkPort*, *inPort*, *outPort*, and *ctrlPort* (i.e. load enable). Type *cwPort* means the port is a control-word port and is used to drive the control ports of the components in the *NiscArchitecture* (see Section 2.2).

The set of connections L_x is defined between a bit-slice of a port $p1$ and a similarly sized bit-slice of port $p2$ as follows:

$$L_x = \{(p1, p2, s1, e1, s2, e2) \mid p1, p2 \in \left(P_x \cup (\bigcup_{y \in C_x} P_y) \right) \text{and}$$
$$0 \le s1 \le e1 < \beta_{p1}, \text{and } 0 \le s2 \le e2 < \beta_{p2}, \text{and } e1 - s1 = e2 - s2\}$$

where, $s1$ and $s2$ are the start index of $p1$ and $p2$ and $e1$ and $e2$ are the end index of $p1$ and $p2$.

A_x is a list of aspects required by different tools for processing component x. Aspects are defined based on components types. Currently, in our toolset, each component has three aspects: compilation aspect CA_x, simulation aspect MA_x, and synthesis aspect NA_x. Compilation aspect usually captures the relation between the component's behavior and the C-language operations, or application functions. Simulation and synthesis aspects usually contain the description of the component in an HDL, or the information required for generating a hardwired core (e.g. memory, divider, etc.). For some component types, if an aspect is not specified by the designer, the toolset will generate it automatically. For example, the simulation/synthesis aspects of hierarchical components can either be generated automatically from their internal components, or be explicitly specified by the designer. This feature allows modeling of third party cores and pre-laid-out components that have special technology or manufacturability considerations. Aspects are also used in defining *proxy* components in a *NiscArchitecture*. A proxy component is a component that resides outside of the IP block but the IP controls it. For example, a memory proxy represents a memory or cache hierarchy that resides outside of the IP. The HDL implementation of a proxy may be as simple as input to output wirings. However, its compiler aspect captures the information for controlling the external component. The *NiscArchitecture* and *behavioralIP* component types have additional properties as follows:

NiscArchitecture: The *NiscArchitecture* represents our target architecture that does not have instruction-set and its control words are generated by the cycle-accurate compiler. The compiler aspect of a *NiscArchitecture* ξ is modeled by CA_ξ=($freq_\xi$, $CNST_\xi$, Γ_ξ, sPt_ξ, fPt_ξ). The $freq_\xi$ specifies the clock frequency of the *NiscArchitecture* and is used by the compiler to generate the proper control words considering the component delays. A control word contains the control values of components as well as a set of constant fields $CNST_\xi$. The constant fields are used for jump and other operations with a constant operand. Each constant field f in $CNST_\xi$ has a bit-width or size denoted by β_f. The Γ_ξ is a function that defines the ordering of the constant and control fields in the control word. This ordering is used by the compiler to generate the correct control words. The sPt_ξ and fPt_ξ are storage components used for stack pointer and frame pointer. The storage components can be separate registers or registers in a register file.

BehavioralIP: *behavioralIP* is a component that its behavior is specified in C language, and is handled by our cycle-accurate compiler, a traditional compiler, or a high-level synthesis (HLS) tool. The compiler aspect of the *behavioralIP* specifies the set of application files (e.g. header files and C files) that execute on that IP. In our approach, the netlist of *behavioralIP* contains a *NiscArchitecture* and, if necessary, a memory subsystem (Figure 1). The cycle-accurate compiler compiles the application C code directly on the datapath of *NiscArchitecture*. The *behavioralIP* can cover instruction-set based general-purpose or custom processors as well, where the synthesis aspect is usually a third-party core.

2.2 GNR Rules

Our formal and typed description allows us to define rules to validate the correctness of the given netlist. Enforcing such rules significantly improves the productivity of the designer by identifying most of the problems without simulation. Depending on the component type, the rules can restrict number and types of the ports, instantiated components, and their connectivity. There are two groups of rules: general rules, and NISC-specific rules.

General rules:

- Clock ports can only connect to clock ports:

$\forall(p1,p2,\ldots)\in L_x$, $\tau_{p1}=clkPort$ if and only if $\tau_{p2}=clkPort$

- Connections in L_x are defined between source ports (i.e. *outPort*) and the destination ports (i.e. *inPort*). For boundary connections (i.e. the connections that involve ports in P_x), the input ports of P_x must be source and its output ports must be the destination.
- Maximum of one connection is allowed to any bit of any destination port. The only exception is for input ports of *bus*-type components, where multiple connections are valid. In digital design, connecting several output ports to a single input port is not valid, unless through tri-state buffers.

$\forall(p1,p2,s1,e1,s2,e2), (p3,p4,s3,e3,s4,e4)\in L_x$, if $p2=p4$, then ($p2\in P_x$ and $\tau_x=bus$) or ($s2>e4$) or ($s4>e2$)

NISC-specific rules:

- Each *NiscArchitecture* ξ has one and only one component of type *controller*:

$\exists!\ x\in C_\xi$, where $\tau_x=controller$

- Only component x with $\tau_x=controller$ can have one and only one port of type *cwPort*:

$\exists!\ p\in P_x$ and $\theta_p=cwPort$ if and only if $\tau_x = controller$

- Each *NiscArchitecture* ξ has at least one component of type *register-file*:

$\exists\ x\in C_\xi$, where $\tau_x=register\text{-}file$

- In *NiscArchitecture*ξ, the bit-width of the *cw* port of controller component must be equal to sum of the bit-widths of all control ports, plus the sum of the bit-widths of all control fields in $CNST_\xi$.

$$\forall cw \in P_c, \text{if } \theta_{cw} = cwPort, \text{then } \beta_{cw} = \sum_{p\in CP_\xi}\beta_p + \sum_{f\in CNST_\xi}\beta_f$$

$$\text{where, } CP_\xi = \{p \mid p \in \bigcup_{x\in C_\xi} P_x \text{ and } \theta_p = ctrlPort\}$$

- Control connections in *NiscArchitecture* ξ are defined between the *cw* port and the control ports of components in C_ξ.

$\forall(p1,p2,s1,e1,s2,e2)\in L_x$, if $p2\in CP_\xi$, then $\theta_{p1}=cwPort$ and $s2=0$ and $e2=(e1-s1)=\beta_{p2}-1$

3. GNR SYNTAX

We use XML language [12] to describe IP models in GNR. We define GNR syntax in XML Schema [13] to enforce syntax and semantics checking on the given input model. The Schema can also be used for code completion, which further increases the productivity of the designers. Figure 2 shows the partial block diagram of the Schema for modeling a custom IP (*NiscArchitecture*). The IP has several children tags including: <Ports>, <Components>, <Connections>, <CwFields>, <Compiler-aspect>, <Simulation-aspect>, and <Synthesis-aspect>, representing P_ξ, C_ξ, L_ξ, Γ_ξ, CA_ξ, MA_ξ, and NA_ξ, respectively. All components in GNR have a <Params> tag that parameterizes that component. For example, the delay or bit-width of the component can be specified as parameters.

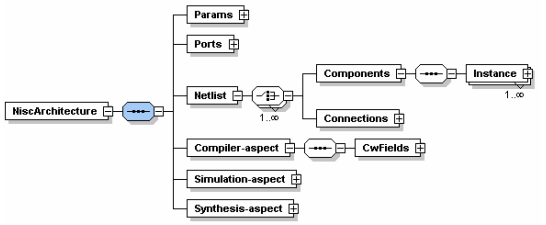


Figure 2- Block diagram of GNR schema for *NiscArchitecture*.

4. EXAMPLE GNR MODELS

In this section, we discuss modeling IPs in more details using several examples. We first explain how a simple component, namely an ALU, is defined in GNR. Then, we explain how components are integrated to form a simple IP that can execute C code. Finally, we show how this IP is extended for system.

4.1 Modeling a custom ALU

ALU is a component of type *functional-unit*. Figure 3 shows the GNR description of a custom ALU that executes three operations: Add, Sub, Not. The component has two parameters: BIT_WIDTH and DELAY. The parameters are initialized during the instantiation of a component in a datapath. This ALU has two input ports, one output port and a control port. Since this ALU executes three operations, the size of the *ctrl* port is at least two. The simulatable and synthesizable code of the ALU are described in the <Simulation-aspect> and <Synthesis-aspect> (not shown in the figure). For some components, it is also possible to generate the HDL description automatically from the component entity information and compiler aspect.

```
- <FU typeName="ALU">
  - <Params>
      <Param n="BIT_WIDTH" />
      <Param n="DELAY" val="1" />
    </Params>
  - <Ports>
      <InPort n="i0" bitWidth="{@BIT_WIDTH}" />
      <InPort n="i1" bitWidth="{@BIT_WIDTH}" />
      <OutPort n="o" bitWidth="{@BIT_WIDTH}" />
      <CtrlPort n="ctrl" bitWidth="2" default="00" />
    </Ports>
    <Simulation-aspect />
    <Synthesis-aspect />
  - <Compiler-aspect>
    - <Operations>
      - <Operation n="Add" delay="{@DELAY}">
          <Output port="o" />
          <Input port="i0" />
          <Input port="i1" />
          <Ctrl val="00" port="ctrl" />
        </Operation>
      - <Operation n="Sub" delay="{@DELAY}">
          <Output port="o" />
          <Input port="i0" />
          <Input port="i1" />
          <Ctrl val="01" port="ctrl" />
        </Operation>
      - <Operation n="Not" delay="{@DELAY}">
          <Output port="o" />
          <Input port="i0" />
          <Ctrl val="10" port="ctrl" />
        </Operation>
      </Operations>
    </Compiler-aspect>
</FU>
```

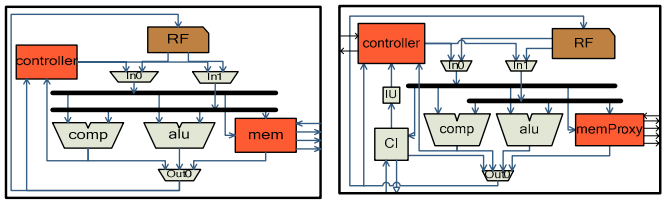
Figure 3- Partial description of a custom ALU in GNR.

In <Compiler-aspect> the operations that the ALU executes are described in details. Each operation has a *name* and a *delay* attribute: the *name* is selected from the list of valid C operations, and the *delay* is specified in terms of number of cycles or nanoseconds, according to the selected target technology. Each operation has a set of input ports and at most one output port. An operation may also require a specific value on one or more control ports. The values are specified using <Ctrl> tag. Using this modeling approach, new functional units can be described and added to the library.

Some functional units are more complex than others. For example, some of them are pipelined, or may require instantiation of hardwired cores provided by a third party. In case of a pipelined unit, a netlist of the main functional unit and the pipeline registers are defined as a *module* in GNR. Most of today's synthesis tools apply retiming to the netlist, and generate proper pipelined functional unit. In case of hardwired cores, the information of the third party tool that must be called for core generation is specified in <Synthesis-aspect>.

4.2 Modeling a simple IP

Figure 4(a) shows the block diagram of a simple *NiscArchitecture* that can execute simple C codes. The architecture consists of a controller, a register file (RF), a data memory proxy, an ALU, a comparator, and a few multiplexers. The bus-width of the IP is 32 bits. The register file has 32 registers, and two read ports and one write port.



(a) **(b)**

Figure 4- Block diagram of a simple IP (a) without, (b) with communication Interface.

```
- <NiscArchitecture  type="simpleIP">
  - <Ports>
      <Clock n="clk" bitWidth="1" />
      <InPort n="reset" bitWidth="1" />
      <InPort n="dm_r" bitWidth="32" />
      <OutPort n="dm_addr" bitWidth="32" />
      <OutPort n="dm_w" bitWidth="32" />
      <OutPort n="dm_readEn" bitWidth="1" />
      <OutPort n="dm_writeEn" bitWidth="1" />
    </Ports>
  - <Netlist>
    - <Components>
        <Instance n="controller" type="Controller" />
      - <Instance n="RF" type="RF2x1">
          <SetParam n="BIT_WIDTH" val="32" />
          <SetParam n="REG_COUNT" val="32" />
        </Instance>
        <Instance n="In0" type="Mux" />
        <Instance n="In1" type="Mux" />
        <Instance n="Out0" type="Mux" />
        <Instance n="comp" type="Comparator" />
        <Instance n="alu" type="ALU" />
        <Instance n="mem" type="DataMemProxy" />
      </Components>
    - <Connections>
        <Conn src="controller" sPort="cw" dest="In0" dPort="i0" extend="signed" s="9" e="0" />
        <Conn src="controller" sPort="cw" dest="In1" dPort="i0" extend="signed" s="9" e="0" />
        <Conn src="comp" sPort="o" dest="controller" dPort="status" s="0" e="0" />
        <Conn src="RF" sPort="r0" dest="In0" dPort="i0" />
        <Conn src="RF" sPort="r1" dest="In1" dPort="i1" />
        <Conn src="Out0" sPort="o" dest="RF" dPort="w0" />
        <Conn src="In0" sPort="o" dest="comp" dPort="i0" />
        <Conn src="In1" sPort="o" dest="comp" dPort="i1" />
        <Conn src="comp" sPort="o" dest="Out0" dPort="i0" />
        <Conn src="In0" sPort="o" dest="alu" dPort="i0" />
        <Conn src="In1" sPort="o" dest="alu" dPort="i1" />
        <Conn src="alu" sPort="o" dest="Out0" dPort="i1" />
        <Conn src="In0" sPort="o" dest="mem" dPort="addr" />
        <Conn src="In1" sPort="o" dest="mem" dPort="w" />
        <Conn src="mem" sPort="r" dest="Out0" dPort="i2" />
        <Conn src="" sPort="dm_r" dest="mem" dPort="dm_r" />
        <Conn src="mem" sPort="dm_addr" dest="" dPort="dm_addr" />
        <Conn src="mem" sPort="dm_w" dest="" dPort="dm_w" />
        <Conn src="mem" sPort="dm_readEn" dest="" dPort="dm_readEn" />
        <Conn src="mem" sPort="dm_writeEn" dest="" dPort="dm_writeEn" />
        <!-- ## 2 clock connections##  -->
        ...
        <!-- ## 13 control connections ##  -->
        <Conn src="controller" sPort="cw" dest="alu" dPort="ctrl" s="10" e="10" />
        ...
      </Connections>
    </Netlist>
  - <Compiler-aspect defaultIntegralRF="RF" defaultDMem="mem">
    - <CwFields n="cwFields">
        <Field n="const0" bitWidth="10" />
        <!-- ## 13 control field ##  -->
        <CtrlField component="alu" ctrlPort="ctrl" />
        ...
      </CwFields>
      ...
    </Compiler-aspect>
  </NiscArchitecture>
```

Figure 5- GNR description of the IP in Figure 4(a).

In this IP, suppose that a constant field of 10 bits is used for operations with a constant operand. Figure 5 shows the GNR description of the IP. The IP has one clock port, a reset port, and several IO ports for communicating with data memory unit. The <Netlist> tag shows the components and connections of the IP. For each instantiated component the proper parameters such as BIT_WIDTH and REG_COUNT are initialized. Thirty four connections are defined for this IP. Each connection determines the source component *src*, source port *sPort*, destination component *dest*, and destination port *dPort*. Among these connections, 19 are shown in Figure 5, and the rest are clock and control connections.

In <Compiler-aspect> the ordering of the control fields are specified by listing the fields in tag <CwFields>. This information is used by the compiler for generating the control words. In this architecture, the total bit-width of the control ports is 35 bits, and the constant width is 10 bits. Therefore, the bit-width of the control words is 45 bits.

4.2.1 Automatic generation of control and clock connections

In order to further simplify the datapath description, if the control connections are not explicitly specified, we generate them automatically by analyzing the components added to the architecture. This improves the productivity significantly because adding the control connections is very error-prone. Our modeling approach allows automatic generation of control connections and control fields, because we distinguish the control ports from other types of ports. Similarly, the clock connections can be added automatically. In this architecture, automatically adding the control connections and control fields reduces the description size by 25%, while reducing the design and validation time by more than two times.

4.2.2 Expanding the IP for communication

In order to use the simple IP of Figure 4(a) in a system we need to add communication capability to it. For example, to connect the component to a double-handshake bus protocol in message-passing mode, we need to add an interrupt unit (IU) and a proper communication-interface unit (CI) to the datapath of the IP. The CI has two send and receive queues controlled by a control port. The block diagram of the new IP is shown in Figure 4(b). In the C code of the application, the CI component is programmed through a set of intrinsic-functions that are described in GNR description of CI. The cycle-accurate compiler detects these functions in the code and translates them to proper control signals for the CI. The details of the bus protocol and CI drivers are available in [16]. This IP is instantiated inside a *behavioralIP* as shown in Figure 1.

5. GENERATING RTL FROM GNR

Figure 6 shows the block diagram of our toolset. The inputs of the toolset are GNR description of the system and the application C codes. The outputs include synthesizable and simulatable RTL codes.

The *Pre-Processor* first verifies the syntax of the given GNR file using the GNR Schema. Next, it completes the netlist by (a) resolving the parameters of the components, (b) adding the missing clock and control connections, and (c) adding the control fields, as explained in Section 4.2. The semantic correctness of the completed netlist is verified afterwards, and proper warning and error messages are reported by *Pre-Processor*. The netlist checker reports unconnected ports, invalid connectivity, and non-existing referenced component and port names. GNR modeling enables additional checking that is not possible using HDL-based structural descriptions or even SystemC. For example, in GNR, if a data port is mistakenly connected to a clock port, or if multiple output-ports are connected to one input port of a non-bus

component, then it is possible to detect and report the problem. Note that such connections are valid in HDLs but they result in an incorrect design behavior. Using such simple checking in GNR, most architecture problems are quickly determined.

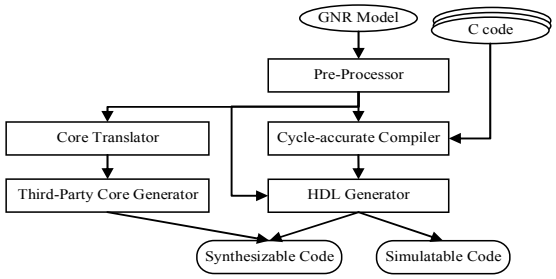


Figure 6-The flow of our toolset.

The *Cycle-accurate compiler* compiles the C code of each PE on the given datapath using the algorithm presented in [9]. If a specific operation required by C code is not supported by a given datapath, then compiler displays proper error messages. After compilation, the compiler generates the contents of data and control memories. The *HDL Generator* uses the GNR and the outputs of the compiler to produce the final simulatable and synthesizable codes. The simulatable code is mostly behavioral and simulates much faster than the synthesizable code. The *Core Translator* generates the input files for third-party core generator by extracting proper information and parameters from the GNR model. The produced cores are combined with the generated HDL code to form the synthesizable code. An online version of the toolset is available at [11].

6. EXPERIMENTAL RESULTS

As experimental results, we designed different system architectures using GNR, and ran a fixed-point MP3 decoder (11000 lines of C code downloaded from [14]) on them. We explored system-level customizations and PE-level customizations in order to maximize the performance gain. For all experiments, we generated Verilog RTL code, and simulated and synthesized them on a Xilinx Virtex II FPGA using Xilinx ISE 8.1 toolset. We measured the execution delay of the MP3 decoder for processing one frame.

We profiled the MP3 decoder to identify its computationally intensive parts. The profiling results showed that during processing of each frame most of the execution time is spent inside DCT and IMDCT filters. Therefore, we can accelerate the execution of these filters using dedicated DCT and IMDCT cores. In this section, we present five system architectures: *System1* includes a MicroBlaze and an OPB bus (Xilinx cores) for off-chip memory communication; *System2* extends *System1* by adding one DCT IP; *System3* extends *System1* by two parallel DCT IPs; *System4* adds one DCT IP and two IMDCT IPs to *System1*; and *System5* includes only one custom IP that runs the entire MP3 decoder. For the filters and the entire MP3 we designed two custom datapaths and used our cycle-accurate compiler to compile the corresponding code on them. The customizations include adding multiple constant fields, proper pipelining and data forwarding. In System5, we also added an integer divider core provided by Xilinx LogiCore.

Our current component library has several communication-interface components for double-handshake bus protocols (DHS). However, OPB uses a master-slave protocol that is not yet implemented in our library. Therefore, in order to communicate between MicroBlaze and our custom IPs, we used a bridge (similar to [15]) that converts the two protocols to each other. Figure 7 shows the block diagram of the *System4* that includes MicroBlaze, OPB bus, bridge, DHS bus, and three custom IPs (One DCT and two IMDCTs).

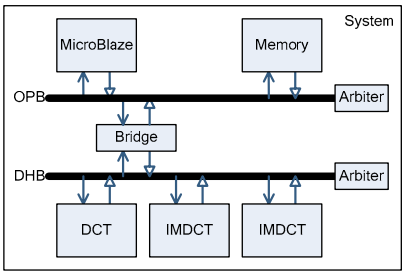


Figure 7- Block diagram of system 4.

Table 1- Performance, memory and area of the five systems.

	# Cycles (millions)	Delay (s)	Speedup	# FPGA Slices
System 1	2.7	0.0540	1.00	1270
System 2	2.54	0.0508	1.06	6008
System 3	2.47	0.0494	1.09	8376
System 4	1.24	0.0248	2.17	10750
System 5	0.83	0.0166	3.25	2600

We captured all these five systems including the two custom IPs in GNR and used our tools to compile the partitioned C code and generated Verilog RTL code for simulation and synthesis. Table 1 shows the performance and area of the five systems. The second column shows the total number of cycles for decoding a frame. The third column shows the overall delay of systems running at 50MHz clock frequency. The fourth column shows the speedup of the systems compared to *System1*. The fifth column shows the area of the designs in terms of number of FPGA slices. To play 38 frames per second (as required by MP3 standard), processing one frame should not take more than 0.026 seconds. System1 processes each frame in 0.054s, and therefore cannot meet the deadline. Among four other systems, only *System4* and *System5* can meet the deadline. *System4* and *System5* run 2.17 and 3.25 times faster than *System1*. However, *System5* consumes 4.1 times less area compared to *System4*. Therefore, *System5* is a better design choice for MP3 application.

Table 2- Specification vs. Genereated code size.

	GNR lines of code				Verilog lines of code	
	system	IP	Total	modified	simulatable code	synthesizable code
system 1	70	NA	70	-	NA	NA
system 2	181	363	544	474	2600	22000
system 3	220	363	583	40	4100	41000
system 4	285	363	648	65	6400	50200
system 5	25	432	457	150	2400	32000

Table 2 shows the size of the GNR files compared to the size of the generated RTL files. The second column of the table shows the GNR lines of code for description of the systems. This includes instantiating the RTL and behavioralIP components in the systems and connecting them together. In case of *System5*, only one IP is instantiated and hence the size of this file is very small. The third column of the table shows the GNR lines of code for describing the IPs. Note that the same IP (with 363 lines), with different parameters, is instantiated once, twice, and three times in *System2*, *System3*, and *System4*, respectively. In *System5*, the IP is more complex and hence has more lines of GNR code (432). The fourth column of the table show the total number of GNR lines of code for each system, i.e. sum of GNR lines for describing the system and its IPs. Note that, since in our experiments

we changed one system to create the next, we did not need to rewrite the whole description again. The number of modified lines of code in each step is shown in the fifth column of the table. For example, when generating *System3* from *System2*, we reused the IP description and only need to modify the system description to instantiate and connect it (40 lines). The last two columns of the table show the size of the Verilog and other core related files that are generated automatically. Note that, while the GNR descriptions are only a few hundred lines of code, the generated files are several thousand lines. This shows the productivity gain of using the GNR.

Overall, we could perform different system level (coarse-grained) and IP level (fine-grained) architecture explorations using relatively small GNR descriptions. The productivity gain was due to several factors including: parametrizable component descriptions, static rule checking, and automatic compilation and RTL generation for the custom IPs. Since GNR enabled us to make detailed architectural adjustments, we were able to achieve significant performance improvement while meeting the area constraints.

7. RELATED WORKS

Over the past years, several ADLs and their supporting software tools have been introduced. A complete survey of these ADLs can be found in [1], [2]. Among these ADLs only the followings have directly or indirectly addressed synthesis of the architecture.

LISA [3], a sate-of-the-art commercial product, and EXPRESSION [4] are behavioral ADLs that capture a processor in terms of its instruction-set behavior and a high level block diagram of its pipeline. They were originally designed for compilation and simulation and have been recently extended to generate the RTL of the processor by synthesizing the instruction behaviors. Since instruction behaviors are described in a very high abstraction level in order to be used by the compiler, achieving a high quality synthesis in these approaches is less likely. Furthermore, the designer has no control over the details of final implementation and is limited to describing the functionality of instructions. Since these ADLs are behavioral, they must capture all possible configurations of instructions. This can lead to very lengthy descriptions. For example, in LISA the description of two RISC processors with four and seven pipeline stages has been reported to be more than 2000 and more than 9000 lines of code, respectively [8].

UDL/I [5] is a hardware description language (HDL) that captures the architecture at the Register-Transfer (RT)-level. A target specific compiler can be generated based on the instruction set extracted from the UDL/I description. UDL/I cannot support architecture with any instruction level parallelism.

MIMOLA [6] is another HDL that captures the architecture netlist at RT-Level and is used for hardware synthesis, simulation, test generation, and code generation. The RECORD compiler [7] extracts behavioral model of instructions from MIMOLA HDL. It processes the structure of the datapath from destination storages towards source storages to extract valid register transfers (RTs). After analyzing the controller and the instruction decoder, it rejects illegal RTs that do not correspond to an instruction, and uses the remaining RTs in the compiler. MIMOLA does not support pipelined architectures and assumes single cycle operations. Furthermore, designer must describe the instruction decoder from which the compiler will extract the set of valid operations. Although RT-level descriptions are more amicable to hardware designers, describing the instruction decoder at RT-level is very tedious. Also instruction set extraction from RT-level is very difficult and is typically possible only for limited target scope.

8. CONCLUSION AND FUTURE WORK

In this paper we presented GNR, a formal modeling approach, developed to improve productivity of designing systems and processing elements, the same way that traditional ADLs improved productivity for designing processors. GNR captures a system as a hierarchical netlist of components annotated by compilation, simulation and synthesis aspects. Our tools and GNR improve the productivity of system design by means of using parametrizable component descriptions, static rule checking, and automatic compilation and RTL generation for the custom PEs.

Furthermore, GNR enhances the designer control over structural details of the design and hence improves design predictability. Using relatively short GNR description, we explored several designs for implementing an MP3 decoder and achieved 3.25 speedup compared to MicroBlaze processor. The future work will address TLM to GNR translation.

9. REFERENCES

[1] P. Mishra and N. Dutt, "Architecture Description Languages for Programmable Embedded Systems", *IEE Proc. on Computers and Digital Techniques (CDT), Special issue on Embedded Microelectronic Systems: Status and Trends*, vol. 152, no 3, 2005.

[2] W. Qin and S. Malik, "Architecture Description Languages for Retargetable Compilation", in *The Compiler Design Handbook: Optimizations & Machine Code Generation*. Y. N. Srikant and Priti Shankar, CRC Press, 2002.

[3] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, A.Wieferink, and H. Meyr. A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language. IEEE Transactions on Computer-Aided Design, 20(11):1338–1354, Nov. 2001.

[4] P. Mishra, A. Kejariwal, and N. Dutt, "Synthesis-driven Exploration of Pipelined Embedded Processors", *International Conference on VLSI Design*, 2004.

[5] H. Akaboshi, "A Study on Design Support for Computer Architecture Design", *Doctoral Thesis, Depart. of Information Systems, Kyushu Univ.*, Japan, Jan. 1996

[6] R. Leupers and P. Marwedel, "Retargetable Code Generation based on Structural Processor Descriptions," *Design Automation for Embedded Systems*, vol. 3, no. 1, 1998.

[7] R. Leupers, P. Marwedel, "Retargetable Generation of Code Selectors from HDL Processor Models", *European Design and Test*, 1997.

[8] A. Chattopadhyay, D. Kammler, E. Witte, O. Schliebusch, H. Ishebabi, B. Geukes, R. Leupers, G. Ascheid, "Automatic Low Power Optimizations during ADL-driven ASIP Design", VLSI-DAT, 2006.

[9] M. Reshadi, D. Gajski, "A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths", CODES+ISSS, 2005.

[10] B. Gorjiara, D. Gajski, "Custom Processor Design Using NISC: A Case-Study on DCT algorithm", ESTIMEDIA, 2005.

[11] http://www.cecs.uci.edu/~nisc

[12] XML: http://www.w3.org/XML/

[13] XML Schema: http://www.w3.org/XML/Schema

[14] http://www.underbit.com/products/mad/

[15] H. Cho, S. Abdi, D. Gajski, "Design and Implementation of Transducer for ARM-TMS Communication", *In Proc. ASPDAC*, Design Contest, 2006.

[16] B. Gorjiara, M. Reshadi, D. Gajski, "NISC Communication Interface", Center for Embedded Computer Systems (CECS) Technical Report TR 06-05, 2006.